**Содержание**

[Введение 2](#_Toc280834657)

[1. Постановка задачи 3](#_Toc280834658)

[1.1. Входные данные 3](#_Toc280834659)

[1.2. Топология вычислительной системы 3](#_Toc280834660)

[2. Проектирование алгоритма планирования 4](#_Toc280834661)

[2.1. Критерии выбора 4](#_Toc280834662)

[2.2. Описание алгоритма 5](#_Toc280834663)

[2.3. Алгоритм расчета критического пути 6](#_Toc280834664)

[2.4. Детали реализации 7](#_Toc280834665)

[3. Проектирование системы симуляции 9](#_Toc280834666)

[4. Результаты работы планировщика 10](#_Toc280834667)

[Выводы 13](#_Toc280834668)

[Литература 14](#_Toc280834669)

# Введение

В настоящее время параллельная обработка данных является приоритетным направлением развития компьютерных систем. Одной из основных проблем в данной области является задача распределения связанных подзадач по исполнителям.

Данная задача является NP-полной и, следовательно, не имеет алгоритма для получения гарантированно оптимального решения. Тем не менее, создано множество стратегий планирования, которые достаточно хорошо справляются с задачей распределения. Существуют, как приспособленные для любых задач, так и специализированные стратегии, учитывающие особенности задачи (к примеру, степень связности задач, топологию графа задач, топологию компьютерной системы и т.д.). Каждая стратегия имеет свои достоинства и недостатки и не может давать одинаково хорошие решения для всех типов задач.

Есть несколько приоритетных параметров при планировании, таких как: оптимизация времени решения, оптимизация количества используемых ресурсов, уменьшение количества пересылок. Необходимо искать компромисс в значениях этих параметров, так как они зависят друг от друга, а мы везде стремимся к минимуму. Полученное планирование будет лежать между крайними случаями: все задачи на одном процессоре и все задачи на отдельных процессорах.

Не последнюю роль в распределении играет и учет особенности топологии используемой компьютерной системы. То есть, выбор стратегии распределения должен основываться на особенностях, как задачи, так и топологии системы.

# Постановка задачи

## Входные данные

На вход планировщик принимает направленный ациклический граф (DAG – directed acyclic graph) задачи. Каждый узел графа имеет вес, который определяет время выполнения задачи, представленной узлом. Каждое ребро в графе направлено по направлению передачи данных и имеет вес, описывающий объем (а следовательно и длительность) пересылки.

## Топология вычислительной системы

Топология вычислительной системы – «общая шина с маркерным доступом».

Стандарт [1] описывает сущность маркерного доступа следующим образом:

1. Маркер управляет правом доступа к физической среде: станция, которая удерживает (владеет) маркер(ом), немедленно получает возможность управления физической средой.
2. Маркер передается станциями, подключенными к среде. В процессе передачи маркера от станции к станции формируется логическое кольцо.
3. Операция устойчивого состояния состоит из фазы передачи данных и фазы передачи маркера.

# Проектирование алгоритма планирования

## Критерии выбора

Алгоритм должен удовлетворять всем поставленным в задании требованиям. Во-первых, он должен одинаково хорошо работать для любых типов задач, которые могут быть заданны при помощи направленного ациклического графа. Во-вторых, он должен быть оптимизирован для работы на системе с топологией «общая шина с маркерным доступом».

Под «любыми типами» задач понимается отсутствие ограничений на структуру графа. К примеру: количество входных и выходных вершин может быть любым, на веса задач и пересылок не накладывается никаких ограничений.

Следует рассмотреть используемую топологию более подробно. Общая шина с маркерным доступом основана на простой общей шине (то есть все вычислительные узлы подключены к одной передающей среде), в которой проблема общего доступа к передающей среде решается при помощи маркера. Маркер представляет собой некоторый сетевой пакет, который с течением времени переходит от узла к узлу в определенном постоянном направлении. Узлы связанны между собой логическим кольцом, по которому и переходит маркер.

Только вычислительный узел, который в текущий момент владеет маркером, имеет возможность пользоваться передающей средой (передавать сообщения). Узел удерживает маркер до тех пор, пока не передаст все необходимые данные.

В общем случае, маркер переходит между узлами очень быстро, но для выделения особенности топологии в данном случае считается, что маркер пытается переходить от узла к узлу, через промежутки времени, равные одному вычислительному такту.

Так как передающая среда это общий ресурс, разделяемый всеми узлами нужно минимизировать время ее использования каждым из узлов. В таком случае, время ожидания возможности передачи у каждого узла будет минимальным. Именно этот фактор и являлся ключевым при выборе алгоритма планирования.

## Описание алгоритма

Исходя из приведенных выше критериев, был выбран алгоритм обнуления ребер (The EZ (edge-zeroing) algorithm) предложенный Саркаром (Sarkar) в 1989 году [2].

Основная его идея в том, что бы убрать из графа задачи все наиболее тяжелые ребра (то есть исключить наиболее длительные пересылки), при этом не ухудшая критический путь в графе. Здесь, под поднятием «критический путь» понимается наиболее длинная цепочка заданий, которые выполняются последовательно (возможно на разных вычислительных узлах). То есть критический путь определят наименьшее время решения для задачи заданной данным графом, для идеального случая распределения по вычислительным узлам.

Вершины, между которыми было обнулено ребро, помещаются в один кластер. Стоимость пересылки всех задач в пределах одного кластера равна нулю (так как, это задачи, которые будут помещены на один процессор, и пересылка будет происходить через общую память).

Основные шаги алгоритма:

1. Составить список всех ребер графа и отсортировать его в порядке убывания веса ребер
2. Начать рассматривать ребра, начиная с начала списка
3. Обнулить текущее ребро
4. Пересчитать критический путь. Если он не увеличился, то добавить вершины, которые соединялись рассматриваемым ребром, в один кластер. Если соединяемые задачи принадлежали двум разным кластерам – кластера объединяются (соответственно, все ребра между задачами новообразованного кластера обнуляются). Если критический путь увеличился – вернуть вес ребра. Перейти к следующей вершине.
5. Повторять пункты 3-4 до тех пор, пока не будут рассмотрены все вершины из списка ребер

В пределах одного кластера, вершины идут в порядке убывания значения статического уровня. Статический уровень вершины, это максимальная сумма весов всех задач, которые лежат на максимально длинном пути от одной из конечных вершин до данной вершины (если корневых вершин несколько, считается максимальный путь от каждой из них, после чего выбирается максимум).

Общее количество полученных кластеров и вершин, не вошедших не в один из кластеров, определяет необходимое количество вычислительных узлов. Вершины, входящие в один кластер, погружаются на один и тот же процессор.

## Алгоритм расчета критического пути

Весьма нетривиальной, представляется задача расчета критического пути в графе. По сути, значение критического пути, это длина максимальной цепочки последовательно выполняемых команд. Для поиска этого значения, применяется поиск в глубину по графу задач.

Поиск в глубину, реализован при помощи рекурсивной функции. Каждая задача в графе хранит значение максимального пути к ней. Функция вызывается во всех корневых вершинах и работает по следующему алгоритму:

1. Получить список наследников текущей задачи
2. Рассчитать длину пути к текущей вершине, через каждого наследника, для этого:

* Если текущая задача и наследник находятся в разных кластерах, то путь рассчитывается как:

<путь к наследнику> + <вес ребра соединяющих текущую задачу и наследника> + <вес текущей задачи>

* Если текущая задача и наследник находятся в одном кластере, то вес ребер между ними нулевой, но эти задачи выполняются на одном процессоре и между их выполнением может вклиниться выполнения других задач из данного кластера. Поэтому длина пути рассчитывается как:

<вес наследника> + <вес задач идущих по времени выполнения между текущей задачей и наследником> + <вес текущей задачи>

* Если у данной вершины нет наследников, то установить путь равный весу данной вершины

1. Найти максимум среди рассчитанных длин путей
2. Сравнить найденный максимум и текущий путь, записанный в задаче. Если максимум больше – установить его как текущий путь.
3. Вызвать функцию подсчета максимального пути для всех предков данной задачи

Объясним некоторые понятия, которые использовались при определении алгоритма:

* Наследник – задание, которое зависит от текущего задания (для начала его работы нужны данные от текущего задания). Наследники находятся на концах ребер, выходящих из вершины графа, представляющего данное задание.
* Предок – задание, от данных которого зависит данное задание. Предки находятся на началах ребер, входящих в вершину графа, представляющего данное задание.

## Детали реализации

При реализации данного алгоритма возникла проблема формы хранения графа. Так как для расчета критического пути был выбран обход графа в глубину, представляется разумным хранения графа как массива объектов, где каждый объект представляет одно задание. Каждое задание хранит в себе список входных и выходных ребер, содержит поле текущего пути и статического уровня.

Для создания кластеров, в процессе работы алгоритма, было решено создать пул кластеров, который реализует возможность создания новых пулов и добавления в них новых вершин. Пул инкапсулирует процесс создания новых пулов: для создания нового пула в него передается задание-ключ и задание-значение. Вершина-ключ ищется в уже существующих пулах. Если такой пул найден – задание-значение добавляется в него, иначе создается новый кластер, в который добавляется и задание-ключ и задание-значение.

# Проектирование системы симуляции

По заданию, результаты работы планировщика должны быть временные диаграммы Ганта для каждого вычислительного узла. Для их получения, нужно создать систему, которая будет эмулировать компьютерную систему с данной топологией и заданным количеством вершин.

## Общая структура системы

Для начала симуляции, задается общее количество вычислительных узлов (общему количеству кластеров и задач не входящих не в один кластер) на которые распределяются все кластера и задачи, не вошедшие ни в один кластер. То есть, каждому кластеру или не распределенной вершине выделяется отдельный процессор.

Следующий шаг это начало симуляции. Симуляция производится потактово. При этом для каждого процессора, ведется история событий, которая включает в себя временные промежутки. Промежутки могут быть следующих типов:

* Вычисление задачи
* Передача сообщения
* Прием сообщения
* Процессор в данный момент удерживает маркер

Все процессоры обходятся по кольцевому списку, и у каждого вызывается метод совершения шага.

После обхода всех процессоров, совершается попытка отбора маркера у процессора, который удерживал его в данном такте и передачи его следующему процессору. Процессор «отдает» маркер, в том случае, если он не совершает передачу данных. Таким образом, на всем протяжении передачи, процессор держит маркер у себя и ни один другой процессор не может начать передачу.

Процесс симуляции завершается, когда все процессоры закончили работу. Процессор считается закончившим работу, если у него не осталось заданий для выполнения и пересылок.

## Внутренняя структура процессора

Каждый процессор содержит в себе очереди вычислительных задач и пересылок. Процессор может одновременно производить вычисления и передавать/принимать сообщение (только одно, использование маркера гарантирует это).

Процессор может находится в состоянии вычисления задачи и/или пересылки. Если процессор, при вызове у него метода совершения шага, не выполняет ни одну задачу, предпринимается попытка выбрать из очереди задач задачу для выполнения. Из очереди выбирается первая задача, она проверяется на готовность (есть ли все данные для начала ее работы). Если задача готова, она удаляется из очереди и процессор начинает выполнять ее. В противном случае ничего не происходит, и задача остается в очереди.

Если задача завершается, она должна передать результат всем своим наследникам. Если наследник находится в том же кластере, что и данная задача, то пересылка не требуется. Иначе создается пересылка и ставится в очередь пересылок.

Каждый процессор хранит множество полученных данных, для определения готовности задач к выполнению. При завершении пересылки сообщения с другого процессора или завершения работы задачи из того же кластера, соответствующие данные заносятся в множество полученных данных.

Если процессор в данный момент содержит маркер, то он пытается начать передачу, если очередь сообщений не пуста. Маркер удерживается процессором все время передачи.

## Оптимизация на этапе симуляции

Процессоры формируют сообщения пересылок между собой, динамически, в процессе симуляции. Может возникнуть ситуация, когда в одном процессоре формируется очередь пересылок. В этом случае имеет смысл сортировать эту очередь в порядке убывания статического уровня, вершин к которым происходит передача. Так как статический уровень определяет порядок выполнения задач и чем он больше, тем раньше задача поступит на выполнение, то есть вероятно информация понадобится ей раньше чем другим задачам.

# Результаты работы планировщика

Созданная программа предоставляет графический интерфейс пользователя, при помощи которого он может ввести граф и через который выводятся результаты планирования в виде временных диаграмм Ганта. На диаграммах отображаются временные оси для каждого процессора системы. На временной оси каждого процессора отображаются промежутки вычисления задач, пересылки и принятия сообщений, а так же наличие маркера у процессора на данном такте.

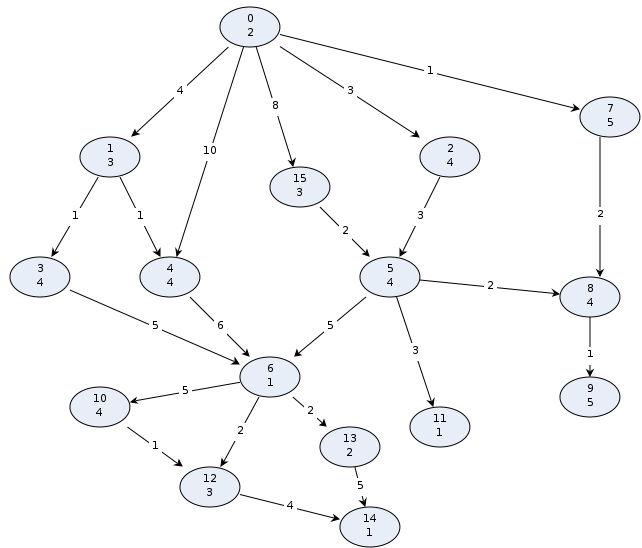
На рисунках 4.1 и 4.2 показаны примеры работы планировщика (графы задач и диаграммы Ганта для них).

## Анализ результатов работы

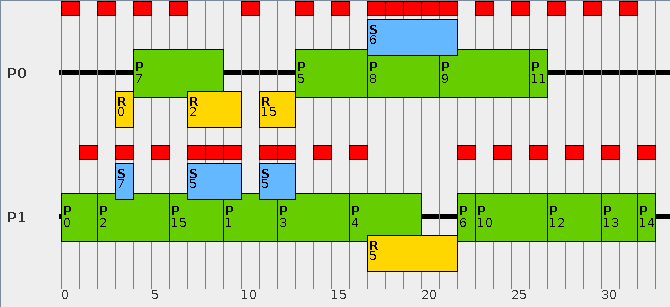
Из полученных примеров, можно сделать вывод, что выбранная стратегия дает достаточно хорошие результаты для шинной топологии с маркерным доступом. Упор сделан на исключении наиболее затратных пересылок и, как можно увидеть по диаграммам Ганта, это разумное решение.

Упор на минимизации использования передающей среды, хорошо сочетается с маркерным методом доступа. Обычно вершины ожидают своей очереди на передачу не очень долго, так как большинство пересылок достаточно коротки.

Алгоритм Саркара в общем случае рассчитан на работу при полносвязной топологии вычислительной системы. Система с общей шиной похожа на полносвязную в том, что любой вычислительный узел может напрямую связаться с любым другим узлом. Отличие в том, что в один момент времени может быть установлена только одна такая связь. Отсюда можем сделать вывод, что чем меньше пересылок в системе и чем меньше их общий вес, тем ближе характеристики шинной топологии к полносвязной, а следовательно алгоритм дает более аптимальные результаты, так как изначально рассчитан на полноствязную систему.



4.1 Пример 1. Граф задачи



4.2 Пример 1. Временная диаграмма Ганта

# S:\Univer\VII_semestr\OS\Cours\pics\Снимок-OS Scheduler-2.png

4.3 Пример 2. Граф задачи

# S:\Univer\VII_semestr\OS\Cours\pics\Снимок-OS Scheduler-3.png

4.4 Пример 2. Временная диаграмма Ганта

# Выводы

В результате проведенной работы получен статический планировщик заданий для вычислительных систем «общая шина с маркерным доступом». Планировщик основывается на алгоритме обнуления ребер в графе задач (алгоритма Саркара). Основная идея алгоритма – убрать все наиболее тяжелые ребра из графа задач, не увеличивая при этом критический путь графа. В результате работы алгоритма, получаются кластеры, то есть множества вершин, которые будут погружены на один процессор. Количество кластеров и вершин вне кластеров определяет необходимое количество вычислительных узлов.

Алгоритм не дает гарантированно оптимальных результатов, но основываясь на результатах, полученных при распределении им задач для нескольких графов задач, можно сделать вывод, о том что он дает удовлетворительные результаты.

При выполнении данной работы я познакомился с теорией статических планировщиков, узнал об их классификации и особенностях. Реализация алгоритма Саркара стала полезным опытом в создании планировщиков и симуляции работы компьютерных систем.

# Литература

1. ГОСТ 34.913.4-91
2. «Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors» YU-KWONG KWOK, ISHFAQ AHMAD, ACM Computing Surveys, Vol. 31, No. 4, December 1999
3. Wikipedia: Critical path method <http://en.wikipedia.org/wiki/Critical_path_method>
4. “A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors”, Tao Yang, Apostolos Gerasoulis