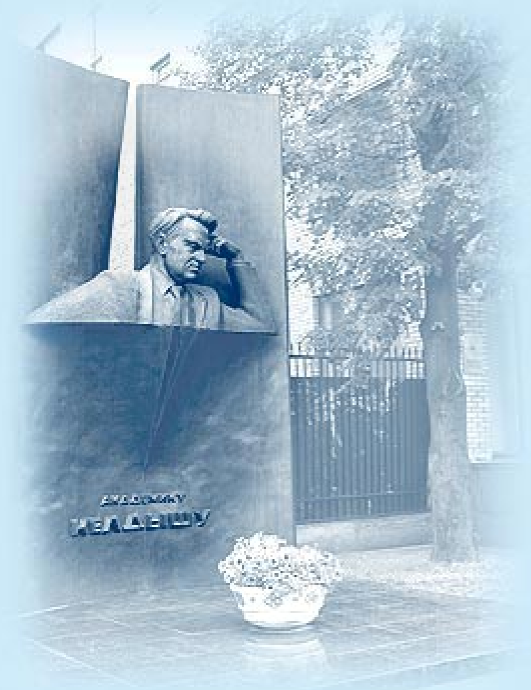
ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 58 за 2024 г.



ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

А.В. Агеев, А.А. Богуславский, С.М. Соколов

Модификация алгоритма HEFT для планирования параллельных работ на гетерогенных вычислителях

Статья доступна по лицензии
Creative Commons Attribution 4.0 International

Рекомендуемая форма библиографической ссылки: Агеев А.В., Богуславский А.А., Соколов С.М. Модификация алгоритма HEFT для планирования параллельных работ на гетерогенных вычислителях // Препринты ИПМ им. М.В.Келдыша. 2024. № 58. 34 с.
https://doi.org/10.20948/prepr-2024-58
https://library.keldysh.ru/preprint.asp?id=2024-58

Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Российской академии наук

А.В. Агеев, А.А. Богуславский, С.М. Соколов

Модификация алгоритма HEFT для планирования параллельных работ на гетерогенных вычислителях

Москва – 2024

Агеев А.В., Богуславский А.А., Соколов С.М.

Модификация алгоритма HEFT для планирования параллельных работ на гетерогенных вычислителях

Современная робототехника требует всё большей степени автономности робототехнических комплексов, что влечёт увеличение количества задач, решаемых на борту робототехнического комплекса (РТК) в реальном времени. Одним из признанных направлений компоновки бортовых вычислительных ресурсов является создание специализированных гетерогенных вычислителей и формирование распределённой бортовой вычислительной сети. Для работы с такими системами необходимо оперативное планирование и эффективное распределение поступающих задач между доступными гетерогенными вычислителями. Известно достаточно много алгоритмов планирования заданий, но в этих алгоритмах нет учёта специфики бортовых вычислителей РТК. В статье предлагается модификация алгоритма HEFT для планирования параллельных работ на гетерогенных вычислителях, которая позволяет организовать конвейерную обработку потока сенсорных данных.

Ключевые слова: *алгоритм планирования, планирование заданий, резервирование ресурсов, периодическая задача, HEFT, реальное время, потоковая обработка данных*

Aleksey Vladimirovich Ageev, Andrey Alexandrovich Boguslavsky, Sergey Michailovich Sokolov

Modification of the HEFT algorithm for scheduling parallel jobs on heterogeneous computers

Modern robotics requires an increasing degree of autonomy of robotic complexes, which leads to an increase in the tasks solved on board the robotic system (RS) in real time. One of the recognized directions of on-board computing resources arrangement is the creation of specialized heterogeneous computers and formation of a distributed on-board computing network. To work with such systems, it is necessary to plan and efficiently distribute incoming tasks between available heterogeneous computers. Quite a lot of task scheduling algorithms are known, but these algorithms do not take into account the specifics of on-board RS computers. The paper proposes a modification of the HEFT algorithm for scheduling parallel jobs on heterogeneous computers, which allows organizing pipeline processing of sensor data stream.

Key Words: *scheduling algorithm, job scheduling, resource allocation, periodic task, HEFT, real time, data stream processing*

1. Введение

При построении робототехнического комплекса (РТК) высокой степени автономности требуется создание бортовой вычислительной системы для исполнения ресурсоёмких алгоритмов в масштабах реального времени, которые обрабатывают потоки данных от системы датчиков РТК. При построении таких систем с целью обеспечения достаточной производительности в условиях ограничений по габаритам и питанию выполняется компоновка вычислительной системы из специализированных вычислителей. В полевых условиях функционирования цели РТК могут меняться. Таким образом, список вычислительных задач может меняться, соответственно, требуется создание системы управления гетерогенными вычислительными ресурсами. Нами были уже проделаны шаги для построения каркаса данной системы [1], но в реализованной системе использовался простейший алгоритм распределения ресурсов для задач Round Robin. Теперь необходимо реализовать алгоритм, позволяющий организовать конвейерную обработку потока данных от системы датчиков, установленной на РТК.

Задача управления гетерогенными вычислениями решена для кластерных, массово-параллельных компьютеров и грид вычислений. Эти системы решают следующие задачи:

1. Распределение ресурсов между пользователями вычислительной системы.
2. Планирование времени исполнения набора заданий.
3. Выделение ресурсов для выполнения задачи в заданный момент времени.

Решение задачи (1) является выделением заданных долей ресурса вычислительной системы между многими пользователями [2], но в нашем случае эта задача не требует решения, поскольку у вычислительной системы только один пользователь, которому отдаются все ресурсы — это система управления РТК.

Для организации планирования исполнения заданий (2) существуют следующие классы алгоритмов [3]:

1. Методы разделения пространства ресурсов.
2. Методы разделения времени.

Методы разделения пространства ресурсов (1) заключаются в том, что использование вычислительных ресурсов во времени описывается некоторым способом и для приходящих заданий выполняется поиск доступных ресурсов в этом описании. Для задачи набор ресурсов выдается в эксклюзивное использование. В эту категорию входит такой алгоритм, как FCFS (First Come First Served) [3]. Суть этого алгоритма заключается в следующем. Как только задача поступает в систему планирования, она попадает в очередь. Если очередь пуста, то ищутся свободные ресурсы для немедленного запуска, если их нет, то задача остается в очереди и дожидается их освобождения, остальные

задания, приходящие в систему, добавляются в конец очереди и ожидают своего момента времени начала планирования (алгоритм Round Robin является более простой реализацией такого подхода). Недостаток этого алгоритма заключается в том, что задача может зависнуть в очереди на неопределенное время, что не является удовлетворительным для систем реального времени. Таким образом, необходимо для каждой задачи выполнять резервирование ресурсов, как только задача попадает в систему планирования – это подход, используемый алгоритмом Backfill [3]. В пакетных системах планирования Backfill используется как оптимизация с целью повышения использования ресурсов системы путем нарушения порядка в очереди. Ресурсы для больших заданий резервируются, а маленькие задания немедленно запускаются несмотря на то, что они не первые в очереди.

Методы разделения времени (2) предполагают запуск на одних и тех же ресурсах нескольких задач, которые путем механизма вытеснения постоянно чередуются друг с другом, алгоритмы данного класса позволяют планировать задания с неопределенным временем исполнения, но такие алгоритмы приводят к замедлению получения результатов [4]. Поскольку для бортовой информационно-управляющей системы (БИУС) РТК важно получать результаты как можно быстрее, то мы будем проектировать алгоритм из категории «Методы разделения пространства ресурсов».

Для планирования времени исполнения заданий между пользователем и планировщиком создается интерфейс, который заключается в обмене следующими сущностями:

- Описание имеющихся ресурсов, которыми может распоряжаться планировщик;
- Задачи, которые необходимо исполнять.

Описание ресурсов – это статическая информация, задается в конфигурационном файле планировщика. Задачи поступают по мере необходимости: от самой системы управления РТК или как результат исполнения задачи – задача сама ставит на исполнение новые задачи. Для организации конвейерной обработки потоковых данных задачу необходимо описывать в виде этапов, каждый из которых требует свой набор вычислительных ресурсов. Этапы необязательно могут идти друг за другом, а могут образовывать параллельные ветви, поэтому конвейер выполнения задания будем описывать в виде ориентированного графа без циклов. Для определенности введем следующие понятия.

Каждая задача описывается в виде графа, элементами которого являются:

- Узлы графа – это атомарные работы (jobs), которые необходимо исполнить; являются этапами конвейерной обработки или источниками данных для этапов обработки;
- Ребра графа – потоки информации между источниками и потребителями.

Для реализации алгоритма планирования времени исполнения задач, которые описываются в виде ориентированного ациклического графа (Directed Acyclic Graph - DAG), за основу предлагается использовать алгоритм HEFT [5]. Данный алгоритм вводит правила, по которым выполняется топологическая сортировка работ (вершин графа) таким образом, чтобы предоставить порядок исполнения в соответствии с зависимостями работ.

Алгоритм HEFT пытается составить расписание исполнения для работ, которые занимают одного исполнителя – это может быть один поток или вся вычислительная машина, что нам не подходит. Алгоритм не может учитывать потребление множества ресурсов (каждая работа обладает своими потребностями), он предназначен для планирования только одного задания – графа работ. В нашем случае необходимо планирование нескольких параллельных заданий, каждая работа в которых может использовать несколько разных ресурсов. В задачах, решаемых БИУС РТК, необходим учет использования множества ресурсов, например, таких как:

- Ядра CPU;
- Набор GPU;
- Объем оперативной памяти;
- Использование дисковой памяти: объем, IOPS (Input/Output Operations Per Second);
- Дополнительные вычислители (FPGA, MTU).

Важность учета потребления множества ресурсов при планировании иллюстрируется следующим примером: БИУС не может запустить две однопоточные работы (каждая работа требует одного ядра CPU) одновременно на многоядерном процессоре, если их суммарное потребление памяти превышает свободную оперативную память. Другой пример: если две работы активно используют дисковую память, то их тоже не стоит запускать параллельно на одном вычислительном узле, поскольку конкурентный доступ к диску увеличивает задержку времени ответа от диска, несмотря на то что операционная система Linux сглаживает медленность диска через буферизацию (дополнительное потребление оперативной памяти), что приводит к замедлению исполнения параллельных работ.

Для решения вышеописанной проблемы мы предлагаем планирование задач разбить на две связанные операции:

- Упорядочивание работ для составления расписания;
- Учет и выделение ресурсов вычислительной системы.

Эти две операции взаимосвязаны, они не могут быть реализованы независимо друг от друга.

Поскольку в дальнейшем для построения системы планирования мы основываемся на алгоритме HEFT, то целью этой работы является разработка

модифицированной версии алгоритма HEFT, которая учитывает потребности БИУС РТК. Алгоритм HEFT будет изменяться в следующих направлениях:

- поддержка периодических заданий;
- поддержка резервирования нескольких ресурсов разного типа для работ;
- поддержка нескольких конфигураций для одной работы.

Планирование периодических заданий означает то, что в расписании будет несколько экземпляров одной и той же задачи, только для обработки данных, поступающих в систему в разное время, что позволяет организовать обработку потока данных. Для этого при размещении очередного экземпляра задачи в расписании необходимо учитывать смещение относительно ранее запланированной копии задачи в соответствии с периодом формирования исходных данных.

Для поддержки резервирования нескольких ресурсов разного типа вместо исходной последовательности действий алгоритма HEFT (который выполняет выбор только одного ресурса) будет использоваться предложенный нами алгоритм учета и выделения ресурсов.

Конфигурация – это реализация работы, использующая определенный набор ресурсов. Разные конфигурации используют разные наборы ресурсов. Необходимы изменения в ранжировании работ графа с целью учета нескольких возможных реализаций одной и той же работы.

Данная работа организована следующим образом. Во втором разделе представлено описание модифицированного алгоритма HEFT (PMR HEFT), который удовлетворяет требованиям БИУС РТК. В третьем разделе представлено описание алгоритма для поиска и выделения ресурсов, который является составной частью алгоритма PMR HEFT. В четвертом разделе представлен пример планирования периодической задачи (графа работ) разработанным алгоритмом.

2. Модифицированный алгоритм HEFT – Periodic Multiple Resources HEFT

Алгоритм планирования Periodic Multiple Resources Heterogeneous Earliest Finish Time (PMR HEFT) выполняет планирование графа задачи:

$$G = \langle V, E \rangle, \qquad (2.1)$$

где G – это ориентированный граф без циклов; $V = \{v\}$ – множество типизированных вершин графа (в оригинальном алгоритме HEFT тип вершины только один, каждая вершина является работой – программой, реализующей некоторый алгоритм). Для работы с графом введем следующие обозначения:

- $v.type$ – тип вершины v;
- $succ(v)$ – множество входящих ребер в вершину v графа G;
- $pred(v)$ – множество исходящих ребер из вершины v графа G;

– $size(M)$ – количество элементов в некотором множестве М.

Существуют следующие типы вершин:

1. $TOPIC_SOURCE$ (тема-источник) – это вершина графа, которая является источником данных. Используется для передачи данных от датчика в систему планирования. Эти данные формируются с известной периодичностью. Также данные могут поступать не только от датчика, но и от другой работы, которая исполняется в системе. Вершина указывает на один элемент из множества источников данных:
$$Dsrc = \{dsrc\}, \tag{2.1}$$
каждый элемент которого содержит следующие свойства:
   – $dsrc.dt$ – тип данных, который передается в систему;
   – $dsrc.dt.size()$ – функция, возвращающая объем используемой оперативной памяти;
   – $dsrc.name$ – название источника данных;
   – $dsrc.p$ – периодичность источника данных;
   – $dsrc.q$ –процессор, на котором расположен источник данных, является частью множества Q, которое описано в формуле (2.3);
   – $dsrc.plannedEmmit$ – планируемый момент времени формирования данных периодическим источником. Экземпляр данных, который формируется в этот момент времени, планируется обрабатывать работой, от которой он зависит. Значение свойства заполняется на шаге алгоритма PMR HEFT 2.2 и 2.5.
2. $TOPIC_DESTINATION$ (тема-приемник) – это вершина графа, которая принимает на вход данные от работы и необходима для хранения результатов исполнения работ и обмена данными между разными графами работ. Вершина указывает на один элемент из множества приемников данных:
$$Ddst = \{ddst\}, \tag{2.2}$$
каждый элемент которого содержит следующие свойства:
   – $ddst.dt$ – тип данных, который принимается данной вершиной;
   – $ddst.name$ – название приемника данных.
3. JOB (работа) – это вершина графа, которая описывает реализацию некоторого алгоритма в виде процесса. Вершина указывает на один элемент из множества работ:
$$Job = \{job\}. \tag{2.3}$$

Для каждой задачи существует свое множество Job, каждый элемент которого содержит следующие свойства:

- $job.id$ – идентификатор работы в рамках задачи. Идентификатор необходим, чтобы отличать копии работ, которые встречаются в одном графе задачи, но относятся к разным узлам графа;
- $job.name$ – название работы;
- $job.InArgs = \{in_arg\}$ – множество входных аргументов в работу. Каждый аргумент содержит атрибуты:
  - $in_arg.name$ – название аргумента;
  - $in_arg.dt$ – тип данных аргумента.
- $job.OutArgs = \{\text{out_arg}\}$ – множество выходных аргументов из работы. Каждый выходной аргумент содержит те же атрибуты, что и входной аргумент;
- $job.C = \{c\}$ – множество конфигураций работы (что такое конфигурация, описано ниже);
- $job.startTime$ – время запуска работы по расписанию. Изначально данный атрибут равняется inf, как сигнал того, что работа еще не добавлена в расписание. Значение устанавливается, как только работа будет добавлена в расписание (шаг алгоритма 2.6);
- $job.stopTime$ – время завершения работы по расписанию. Изначально данный атрибут равняется inf, как сигнал того, что работа еще не добавлена в расписание. Значение устанавливается, как только работа будет добавлена в расписание (шаг алгоритма 2.6);
- $job.q$ – процессор, на котором будет запущена работа. Изначально данный атрибут пустой, заполняется при добавлении работы в расписание. q является частью множества Q, которое описано в формуле (2.5);
- $job.c$ – выбранная конфигурация работы, которая будет запущена на вычислительном узле $job.q$. $job.c \in job.C$. Значение атрибута назначается алгоритмом планирования.

$E = \{e\}$ – множество связей между вершинами графа, каждая из которых обладает следующими свойствами:

1. $e.srcName$ – название источника данных:
   - Если вершина-источник имеет тип Topic-Source, то название совпадает с названием источника данных;
   - Если вершина-источник имеет тип Job, то название совпадает с названием одного из выходных аргументов работы.
2. $e.dstName$ – название приемника данных:
   - Если вершина-приемник имеет тип Topic-Destination, то название совпадает с названием приемника данных;

- Если вершина-приемник имеет тип Job, то название совпадает с названием одного из входных аргументов работы.

3. $e.start$ – вершина, из которой выходит ребро.
4. $e.end$ – вершина, в которую входит ребро.
5. Типы данных источника и приемника должны совпадать.

Каждая задача описывается сущностью $task$, которая имеет следующие атрибуты:

1. $task.id$ – уникальный идентификатор задачи.
2. $task.G$ – граф задачи, каждый узел графа ссылается на один элемент из трех множеств ($task.Job, task.Dsrc, task.Ddst$).
3. $task.Job$ – множество работ, из которых состоит задача.
4. $task.Dsrc$ – множество источников данных для задачи.
5. $task.Ddst$ – множество приемников результатов исполнения работ задачи.

Каждая связь графа $e = (v_i, v_j) \in E$ накладывает ограничение на возможное время запуска работ. В данном случае работа v_j может быть запущена только после того, как источник данных v_i сформирует данные, так как v_j использует данные v_i как входные аргументы. Вершина v_i может быть как работой, так и темой-источником.

Каждая конфигурация работы (c) – это реализация работы с использованием определенного набора ресурсов. Таким образом, описание конфигурации работы содержит следующие атрибуты:

1. $c.R = \{r\}$ – множество ресурсов, необходимых для исполнения конфигурации работы. Каждый элемент содержит следующие атрибуты:
   - $r.name$ – название ресурса;
   - $r.amount$ – количество необходимого ресурса в условных единицах. Для каждого типа ресурса условные единицы несут свою семантику. Например, для оперативной памяти это количество мегабайт, потребляемое алгоритмом; для центрального процессора это количество используемых ядер.

Все типы данных, которые используются в системе, представлены во множестве:

$$DT = \{dt\}, \tag{2.4}$$

каждый элемент, которого содержит следующие атрибуты:

2. $dt.name$ – название типа данных.
3. $dt.type$ – разновидность типа данных (статический или динамический).
4. $dt.size()$ – функция, возвращающая объем потребляемой памяти типом данных.

Типы данных разделяются на два класса:

1. Статический тип данных – это тип данных, который имеет постоянный заранее известный объем, измеряемый в байтах.
2. Динамический тип данных – это тип данных, который может иметь разный объем в зависимости от параметров. К такому типу данных, например, относится изображение, размер в байтах которого зависит от разрешения и способа кодирования. В этом случае размер типа данных указывает источник этого типа данных.

В графе G вершина без родителей называется входной вершиной (истоком), а вершина без дочерних элементов – выходной вершиной (стоком). Далее предполагается, что в графе исток и сток представлены в единственном экземпляре. Если это не так, то в граф необходимо добавить фиктивную вершину исток, которая соединит ребрами исходные истоки, добавить фиктивную вершину сток, с которой соединяются ребрами исходные вершины стоки. Таким образом получаем граф, в котором ровно один исток и один сток.

Мы предполагаем, что целевая вычислительная система состоит из множества исполнителей, соединенных сетью в полносвязную топологию

$$Q = \{q\}. \tag{2.5}$$

Каждый исполнитель из множества Q обладает следующими атрибутами:

1. Множеством ресурсов $R = \{r\}$, которым обладает исполнитель. Каждый ресурс обладает следующими атрибутами:
   - r.name – название ресурса;
   - r.amount – количество ресурса, которым обладает вычислитель q в условных единицах.
2. $Slots$ – структура данных, которая хранит расписание доступности ресурсов вычислительного узла. Структура данных и алгоритм манипулирования этой структурой описаны в разделе 3.

Предполагается планирование без вытеснения.

В пункте 3 описания алгоритма учета и выделения ресурсов используемая терминология отличается. Алгоритм учета и выделения ресурсов оперирует сущностью «задача». Эта сущность совпадает с сущностью «работа», которой оперирует алгоритм PMR HEFT. В контексте алгоритма PMR HEFT сущность «задача» — это граф сущности $task$, которая была описана выше.

Для описания свойств компьютерной сети используется квадратная матрица B, которая описывает пропускную способность сети между всеми узлами:

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1,\mathrm{size}(Q)} \\ b_{21} & & & b_{2,\mathrm{size}(Q)} \\ \vdots & & b_{ij} & \vdots \\ b_{\mathrm{size}(Q),1} & b_{\mathrm{size}(Q),2} & \cdots & b_{\mathrm{size}(Q),\mathrm{size}(Q)} \end{bmatrix} \tag{2.6}$$

Здесь b_{ij} – пропускная способность сети при отправке данных от узла i к узлу j; B является квадратной матрицей, и ее размер зависит от количества ВУБ $(\mathrm{size}(Q) \times \mathrm{size}(Q))$.

Также каждый ВУБ обладает накладными расходами на запуск коммуникаций. Эти накладные расходы будем описывать массивом

$$L = \{l_1, l_2, \ldots, l_i, \ldots l_{\mathrm{size}(Q)}\}, \tag{2.7}$$

где l_i – время в миллисекундах, необходимое вычислительному узлу i на подготовку к передаче данных.

Для планирования работ необходимо знать, сколько времени требуется для исполнения каждой конфигурации каждой работы на каждом ВУБ. Для этого зададим следующую матрицу:

$$W = \begin{bmatrix} \{w_k\}_{11} & \{w_k\}_{12} & \cdots & \{w_k\}_{1,\mathrm{size}(Q)} \\ \{w_k\}_{21} & & & \{w_k\}_{2,\mathrm{size}(Q)} \\ \vdots & & \{w_k\}_{ij} & \vdots \\ \{w_k\}_{\mathrm{size}(Job),1} & \{w_k\}_{n2} & \cdots & \{w_k\}_{\mathrm{size}(Job),\mathrm{size}(Q)} \end{bmatrix} \tag{2.8}$$

Здесь $\{w_k\}_{ij}$ – множество времен исполнения конфигураций работы i на ВУБ j, которые могут быть запущены на ВУБ j. Количество элементов этого множества равняется или меньше количества конфигураций работы i, так как не каждая конфигурация работы i может быть запущена на ВУБ j. Каждый отдельный элемент этого множества будем обозначать следующим образом:

$$w_{ijk}. \tag{2.9}$$

w_{ijk} – время исполнения конфигурации k работы i на ВУБ j; соответственно, индекс строки – это указатель на работу, индекс колонки – это указатель на ВУБ. Размер матрицы $\mathrm{size}(Job) \times \mathrm{size}(Q)$.

Определим структуру данных расписания ($schedule$). Расписание содержит следующие атрибуты:

1. $schedule.tasks$ – список запланированных задач.
2. $schedule.lastPlanedJob: jobInTaskId \rightarrow job$:
   - отображение идентификатора работы на последний запланированный экземпляр работы. Данное отображение используется для определения сдвига по времени при планировании копии задачи для организации непрерывной обработки сенсорных данных. Такое отображение может быть реализовано через хеш-таблицу (такая реализация будет предполагаться дальше в описании алгоритма).
3. $jobInTaskId$ – это составной идентификатор, который состоит из:
   - $jobInTaskId.taskId$ – уникального идентификатора задачи;
   - $jobInTaskId.jobId$ – идентификатора работы в рамках задачи.

Составной идентификатор необходим для определения конкретного запланированного экземпляра работы, поскольку одна работа может

быть составной частью для нескольких задач, а также одна и та же работа может несколько раз встречаться в графе задачи.

4. $schedule.currentTime$ – текущее реальное время, в которое был запущен алгоритм планирования для добавления задачи в расписание.
5. $schedule.offset$ – сдвиг по времени. При планировании самое раннее время старта для планируемой работы:

$$schedule.currentTime + schedule.offset. \quad (2.10)$$

Этот сдвиг используется из-за того, что задача добавляется в расписание не мгновенно. Поэтому планировать задачи необходимо на будущее (с некоторым сдвигом по времени относительно текущего). Этот сдвиг является конфигурационным параметром.

Алгоритм PMR HEFT состоит из следующих этапов:

1. Топологическая сортировка узлов графа, которые имеют тип «работа», состоит из следующих этапов:
   1.1. Вычисление среднего времени исполнения каждой работы.
   1.2. Вычисление средних накладных расходов на начало коммуникации.
   1.3. Вычисление матрицы среднего времени передачи данных между работами.
   1.4. Ранжирование работ.
   1.5. Сортировка работ по убыванию ранга.

2. Цикл. В отсортированном порядке выполняется обход работ, для каждой из которых выполняются следующие этапы:
   2.1. Определение самого позднего времени завершения среди родительских работ.
   2.2. Вычисление самого раннего допустимого времени старта в зависимости от периодических источников данных и дубликатов запланированных работ.
   2.3. Вычисление матрицы времени передачи данных от родительских узлов графа.
   2.4. Поиск и резервирование слота на одном из ВУБ, который дает раннее время завершения работы.
   2.5. Корректировка планируемого времени получения данных от периодического источника, которое было запланировано на шаге 2.2.
   2.6. Добавление работы в расписание в соответствии с зарезервированным слотом.

3. Задача, у которой запланированы все работы, добавляется в конец списка запланированных задач.

Опишем каждый этап алгоритма более подробно.

Шаги первого этапа:

1.1 Вычисление среднего времени исполнения каждой работы ($\overline{w_i}$)

$$\overline{w_i} = \frac{1}{\text{size}(Q)} \sum_{j=1}^{\text{size}(Q)} \frac{1}{\text{size}(\{w_k\}_{ij})} \sum_{k=1}^{\text{size}(\{w_k\}_{ij})} w_{ijk}, \quad (2.11)$$

где $\overline{w_i}$ – среднее время выполнения задачи i на всех ВУБ (с учетом конфигураций доступных для запуска на каждом ВУБ).

1.2 Вычисление средних накладных расходов на начало коммуникации ($\bar{L}$)

$$\bar{L} = \frac{1}{\text{size}(L)} \sum_{i=1}^{\text{size}(L)} l_i. \quad (2.12)$$

1.3 Вычисление матрицы среднего времени передачи данных между работами

Среднее время передачи вычисляется относительно средней пропускной способности вычислительной сети:

$$\bar{B} = \frac{2}{\text{size}(Q) \times \text{size}(Q) - \text{size}(Q)} \sum_{i=1}^{\text{size(Q)}} \sum_{j=1}^{i} b_{ij} \quad (2.13)$$

Для каждой связи $e \in E$ графа G, выполняется вычисление среднего времени передачи данных по следующей формуле:

$$\bar{\text{T}}(e) = \bar{L} + \frac{e.dt.\text{size}}{\bar{B}}; \quad (2.14)$$

1.4 Ранжирование работ

Для каждой работы необходимо вычислить ранг. Ранг вычисляется по следующей формуле:

$$rank_u(v_i) = \overline{w_i} + max_{e_j \in \text{succ}(v_i)} \left(\bar{T}(e_j) + rank_u(e_j.start)\right). \quad (2.15)$$

Функция по вычислению рангов вершин графа реализуется на основе алгоритма обхода графа в ширину. Сначала выполняется транспонирование графа, а затем его обход в ширину, начиная со стока.

1.5 Сортировка работ по убыванию ранга

Множество работ $Job = \{job\}$ сортируется по убыванию ранга, который был вычислен для соответствующих вершин графа G.

Шаги второго этапа:

На предыдущем этапе множество Job было отсортировано по рангам. Выполняется итерационный цикл по данному множеству в отсортированном порядке. Пусть job_i – это работа на шаге итерации. Дальше все шаги выполняются внутри этого цикла.

2.1 Определение самого позднего времени завершения среди родительских работ

$$parentJobStop(job_i) = \max_{e_j \in \mathrm{succ}(job_i)} \left(I_{job}(e_j.start) \cdot e_j.start.stoptime\right), \quad (2.16)$$

где $I_{job}(v_j) = \begin{cases} 1 & v_j.type == JOB \\ 0 & \text{иначе} \end{cases}$ – сигнальная функция, которая проверяет тип вершины графа: «Имеет ли вершина тип работы?».

Предполагается, что если значение сигнальной функции равняется нулю, то дальнейшие вычисления не производятся. Соответственно, не выполняется обращение к несуществующему атрибуту у вершины графа. Таким образом, в этой формуле в вычислениях участвуют только вершины графа, которые имеют тип «работа». Также поскольку множество работ было отсортировано в соответствии с рангом, то это приводит к топологической сортировке работ в соответствии с графом. Таким образом, работа либо не имеет родительских работ, либо родительские работы уже запланированы на исполнение (для них определен атрибут $stopTime$).

2.2 Вычисление самого раннего допустимого времени старта в зависимости от периодических источников данных и дубликатов запланированных работ

1. Определяется множество периодических источников данных работы job_i:

$$SRC = \{e.start \mid e \in succ(job_i), e.start.type = TOPIC_SOURCE\}. \quad (2.17)$$

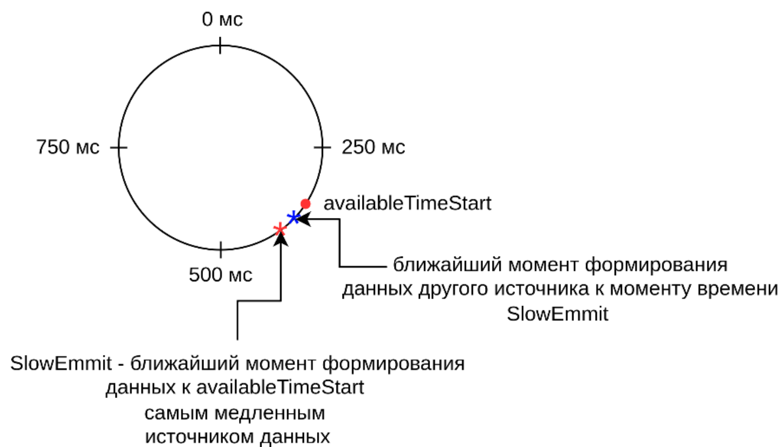


Рис 2.1. Круг времени, на котором выполняется расчет времени формирования данных для планирования исполнения работы по обработке этих данных

2. Определяется самый медленный источник данных:

$$\exists v_{slow} \in SRC, \forall v_s \in SRC: v_{slow}.p \geq v_s.p. \tag{2.18}$$

3. Поскольку периодичность источников данных определяется с точностью до одной миллисекунды из частоты формирования данных за одну секунду, то далее мы будем работать на конечном множестве целых чисел, состоящих из 1000 элементов:

$$\{0\text{мс}, 1\text{мс}, 2\text{мс}, \ldots 999\text{мс}\}, \tag{2.19}$$

каждый элемент которого описывает текущее положение в пространстве времени (в данной секунде реального времени). Это множество графически можно представить в виде круга, на котором равномерно расположены элементы этого множества (рис. 2.1). Красной точкой на круге отмечено текущее положение системы планирования во времени ($availableTimeStart$). $availableTimeStart$ на текущий момент определяется как:

$$\begin{gathered} availableTimeStart = parentJobStop(job_i) \\ if availableTimeStart < (schedule.currentTime + \\ schedule.scheduleOffset)\ then; \\ availableTimeStart = schedule.currentTime \\ +schedule.scheduleOffset. \end{gathered} \tag{2.20}$$

4. Необходимо вычислить текущее положение на круге времени:

$$timeOnRing = availableTimeStart\%1000, \tag{2.21}$$

где % – это операция вычисления остатка от деления.

5. Определяем индексы используемых экземпляров данных идентичной уже запланированной работы (этот шаг необходим, чтобы запланировать обработку следующих экземпляров данных от источника). Для этого выполняем формирование составного идентификатора:

$$jobInTask.jobId = i; jobInTask.taskId = task.id. \tag{2.22}$$

6. Выполняем поиск информации об используемых данных уже запланированной идентичной задачей в хеш-таблице:

$$\begin{gathered} jobData_{same} = schedule.lastPlannedJob[jobInTask], \\ jobData_{same} = \{dataIndex_i\}, len(jobData_{same}) = len(SRC), \end{gathered} \tag{2.23}$$

где $jobData_{same}$ – массив индексов обрабатываемых данных; $dataIndex_i$ – номер экземпляра данных периодического источника $v_i \in SRC$, обрабатываемый последним запланированным экземпляром задачи.

Задача job_i должна обрабатывать следующие данные, экземпляры данных, индекс которых больше, чем индексы, представленные в массиве $jobData_{same}$.

7. Если в хеш-таблице $schedule.lastPlannedJob$ найдена запись, то выполняются следующие действия:

7.1 Выполняем формирование $jobData_i$ – массива индексов обрабатываемых данных работой job_i:

$$jobData_i = jobData_{same}. \quad (2.24)$$

7.2 Все элементы массива $jobData_i$ необходимо увеличить на единицу.

7.3 Выполняем итерацию по всем элементам множества SRC, для каждого элемента $v_i \in SRC$ выполняются следующие действия:

$$\begin{gathered} shift = periodMax/v_i.p; \\ step = round(shift * n); \\ dataIndex_i = step, dataIndex_i \in jobData_i; \\ time = step * period; \\ v_i.plannedEmmit = time, \end{gathered} \quad (2.25)$$

где $periodMax$ – периодичность самого медленного источника данных; n – минимальное значение из массива $jobData_{same}$.

8. Если в хеш-таблице $schedule.lastPlannedJob$ отсутствует запись с указанным составным идентификатором, то выполняются следующие действия:

8.1 Вычисляем индекс ближайшего экземпляра данных самого медленного источника (на рис. 2.1 этот момент времени обозначен красной звездочкой).

$$n = ceil(timeOnRing/periodMax). \quad (2.26)$$

8.2 Относительно самого медленного источника данных вычисляем ближайшие моменты времени формирования данных остальными источниками (на рисунке 2.1 в качестве примера это положение указано синей звездочкой). Выполняем итерацию по всем элементам множества SRC. Для каждого элемента $v_i \in SRC$ выполняются следующие действия:

$$\begin{gathered} doubleshift = periodMax/v_i.p; \\ step = round(shift * n); \\ dataIndex_i = step, dataIndex_i \in jobData_i; \\ time = step * v_i.p; \\ v_i.plannedEmmit = time, \end{gathered} \quad (2.27)$$

где $periodMax$– периодичность самого медленного источника данных.

9. Во время выполнения циклов из шага 7 или 8 также выполняется вычисление самого позднего времени формирования данных: $maxTime$.
10. Обновляем доступное время для запуска работы:

$$\begin{gathered} timeShift = maxTime - timeOnRing; \\ availableTimeStart = availableTimeStart + timeShift. \end{gathered} \tag{2.28}$$

2.3 Вычисление матрицы времени передачи данных от родительских узлов графа

Матрица имеет следующую структуру:

$$TransferTime = \begin{bmatrix} tt_{11} & tt_{12} & \cdots & tt_{1,\text{size}(Q)} \\ tt_{21} & & & tt_{2,\text{size}(Q)} \\ \vdots & & tt_{kj} & \vdots \\ tt_{\text{size}(\text{succ}(job_i)),1} & tt_{\text{size}(\text{succ}(job_i)),2} & \cdots & tt_{\text{size}(\text{succ}(job_i)),\text{size}(Q)} \end{bmatrix} \tag{2.29}$$

tt_{kj} – время, необходимое для передачи данных от источника данных v_k, который расположен на вычислительном узле $v_k.q$, до вычислительного узла q_j. Каждый элемент матрицы вычисляется по следующей формуле:

$$tt_{kj} = \frac{B[e_k.start.q, q_j]}{dateType(e_k).\text{size}}, \tag{2.30}$$

где k – это индекс, который итерируется по списку входящих ребер в узел job_i:

$$e_k \in \text{succ}(job_i), k = 1, \ldots, \text{size}\big(\text{succ}(job_i)\big). \tag{2.31}$$

Каждый узел графа обладает атрибутом q. Таким образом, выражение $e_k.start.q$ означает, на каком вычислительном узле расположен узел графа, из которого выходит ребро e_k.

B – это матрица, которая описывает пропускную способность сети, где $B[q_i, q_j]$ – пропускная способность сети между узлами q_i и q_j, определяемая формулой (2.6).

$dataType: e \to dt, e \in E, dt \in DT$ – функция, которая возвращает тип данных, передаваемый по ребру графа e. ВУБ, на котором расположен источник данных («тема»), известен заранее из конфигурационного файла. Местоположение источника типа «работа» известно, поскольку все предшествующие работы для текущей уже запланированы в расписании и, соответственно, им назначен узел исполнитель.

Размерность матрицы TransferTime: $\text{size}(\text{succ}(job_i)) \times \text{size}(Q)$.

2.4 Поиск и резервирование слота на одном из ВУБ, который дает раннее время завершения работы

Слот – это конечный временной интервал на вычислительном узле, обладающий фиксированным объемом доступных ресурсов, которые могут быть использованы для запуска работы.

Определим функцию, которая на основании матрицы TransferTime вычисляет максимальное время ожидания получения входных данных для задачи job, если мы хотим запустить ее на ВУБ q:

$$transferTime(q, job) = \max_{e \in \mathrm{succ}(job)} (TransferTime[e.start, q]). \qquad (2.32)$$

Функция вычисляет максимальный элемент в столбце q – матрицы TransferTime.

Тогда функция поиска слота запуска работы с ранним временем завершения работы среди всех ВУБ и конфигураций для работы job_i будет иметь вид

$$slot = \min_{\substack{jc \in job_i.C \\ q \in Q}} \left\{ FindSlot \begin{pmatrix} q, \\ W[job_i, q][jc], \\ jc, \\ availableTimeStart + \\ +transferTime(q, job_i) \end{pmatrix} .endTime \right\}. \qquad (2.33)$$

Функция $FindSlot$ зависит от 4 аргументов:

1. Вычислительный узел, на котором необходимо вести поиск свободного слота.
2. Длина искомого слота по времени.
3. Набор ресурсов, который должен быть доступен в слоте.
4. Время, с которого необходимо начать поиск подходящего слота. Аренда слота не должна начинаться раньше этого времени, только позже.

Функция $FindSlot$ возвращает структуру $slot$, которая обладает следующими атрибутами:

1. $slot.startTime$ – момент времени, в который начинается аренда ресурсов вычислительного узла (в этот момент времени будет запущена работа).
2. $slot.endTime$ – момент времени, в который заканчивается аренда ресурсов вычислительного узла. В этот момент времени работа завершит свое исполнение.

Алгоритм (2.33) выполняет перебор всевозможных пар вида:

$$\langle \text{Конфигурация работы; ВУБ} \rangle, \qquad (2.34)$$

среди которых ищется пара, дающая слот для запуска работы с самым ранним временем завершения. После того как необходимый слот будет найден,

необходимо зарезервировать ресурсы выбранного слота. Таким образом, результатом работы этого этапа является кортеж:

$$\langle q_{found}; slot_{found}; jc_{found} \rangle, \tag{2.35}$$

где q_{found} – вычислительный узел, на котором резервируются ресурсы;
$slot_{found}$ – временной слот, который выделяется для исполнения задачи;
jc_{found} – выбранная для запуска конфигурация работы.

Алгоритмы функции FindSlot и резервирование ресурсов под найденный слот описаны в разделе 3.

2.5 Корректировка планируемого времени получения данных от периодического источника, которое было запланировано на шаге 2.2

На шаге 2.2 выполнялось вычисление левой границы формирования данных. После определения доступного места для работы время ожидания момента времени выбранного слота для запуска может значительно превышать время передачи данных. Это приводит к тому, что работа будет обрабатывать устаревшие данные. Задача системы планирования – предоставлять в обработку наиболее свежие данные, насколько это возможно, поэтому необходимо пересчитать индексы обрабатываемых данных.

1. Необходимо вычислить новое самое позднее время формирования данных относительно запланированного слота исполнения работы job_i.
2. $maxAvailableTimeDataEmmit = 0$ – самое позднее допустимое время формирования данных, которое будет вычислено в следующем цикле.
3. Циклически перебираем элементы множества SRC. Для каждого элемента множества $v_i \in SRC$ выполняем следующие действия:

(2.36)

$$transferTime = \frac{v_i.dt.\text{size}}{B[v_i.q, job_i.q]} + L[v_i.q];$$
$$availableTimeDataEmmit = job_i.startTime - transferTime$$
$$if(availableTimeDataEmmit < maxAvailableTimeDataEmmit)then$$
$$maxAvailableTimeDataEmmit = availableTimeDataEmmit.$$

4. Проходим по массиву SRC и определяем источник с самым большим интервалом формирования данных: v_{slow}

$$v_{slow} = \max_{v_i \in SRC}\{v_i.p\}. \tag{2.37}$$

5. Пересчитываем время формирования данных для самого медленного источника:

$$v_{slow}.plannedEmmit = floor(maxAvailableTimeDataEmmit/v_{slow}.p) * v_{slow}.p, \quad (2.38)$$

где $floor$ – округление дробного числа вниз.

6. Циклически перебираем элементы множества SRC. Для каждого элемента множества $v_i \in SRC$ выполняем следующие действия:

$$v_i.plannedEmmit = round(v_{slow}.plannedEmmit/v_i.p) * v_i.p, \quad (2.39)$$

где $round$ – округление дробного числа.

2.6 Добавление работы в расписание в соответствии с зарезервированным слотом

1. В структуре job_i заполняем следующие атрибуты:

$$job_i.q = q_{found};$$
$$job_i.c = jc_{found};$$
$$job_i.startTime = slot.startTime;$$
$$job_i.endTime = slot.endTime. \quad (2.40)$$

2. Сохраняем экземпляр работы в хеш-таблице последних запланированных работ для ее учета при периодическом планировании копии работы. Хеш-таблица не содержит копии работ: если работа с составным идентификатором уже существует, то информация перезаписывается.
   2.1. Формируем составной идентификатор:

$$jobInTask.jobId = i; jobInTask.taskId = task.id. \quad (2.41)$$

   2.2. Добавляем описание запланированной работы в хеш-таблицу, где ключом является составной идентификатор:

$$schedule.lastPlannedJob[jobInTask] = job_i. \quad (2.42)$$

3. $schedule.currentTime = now()$ – устанавливаем значение переменной времени реальным временем, значение которого обеспечивается операционной системой (wall clock).

После прохода по списку всех работ задачи $task$ эту задачу необходимо добавить в список запланированных задач.

3. Задача, у которой запланированы все работы, добавляется в конец списка запланированных задач:

$$shcedule.tasks.append(task). \quad (2.43)$$

3. Алгоритмы учета, планирования и выделения ресурсов

Задачу планирования использования ресурсов в кластерах обычно рассматривают как задачу о рюкзаке в различных вариациях. Например, задачу упаковки набора прямоугольников в группу полубесконечных полос различной ширины (multiple strip packing problem), задачу об упаковке в контейнеры (bin-packing problem). В зависимости от целей системы планирования выбирается та или иная вариация задачи о рюкзаке для моделирования и построения алгоритма.

В случае multiple strip packing problem выполняется планирование одного типа ресурса, например, одновременного использования нескольких ядер процессора одной работой. В этом случае каждый вычислительный узел описывается в виде полубесконечной полосы, в которые укладываются прямоугольники (без возможности их вращения). Ширина полосы и прямоугольника – это количество ядер, доступное на узле, и количество ядер, требуемое работой соответственно, а высота прямоугольника – это время, на которое необходимо зарезервировать ресурсы для работы [6]. Такое представление задачи планирования нам не подходит, поскольку предоставляется возможность оперировать только одним ресурсом, а также в связи с геометрической интерпретацией работе выделяются только «соседние» ядра процессора, идентификаторы которых образуют непрерывную последовательность целых чисел. В некоторых случаях это бывает полезно. Например, если имеется большое количество вычислительных узлов, то строится сложная сетевая архитектура, в которой разные вычислительные узлы имеют разную задержку при передаче сообщений друг другу. Например, для архитектуры сети Fat-Tree. Тогда в этом случае алгоритм планирования может оперировать местоположением вычислительных узлов, и выгоднее располагать процессы одной работы на узлах, которые находятся ближе друг к другу, в соответствии с сетевой топологией. В случае bin-packing problem возможно планирование использования множества ресурсов. Данное представление задачи планирования мы и будем использовать для описания алгоритмов учета и выделения ресурсов.

Для моделирования проблемы планирования использования ресурсов параллельными работами, такими как задача об упаковке в контейнеры, i-й вычислительный узел представляется как d-мерный контейнер, емкость которого описывается вектором:

$$\vec{C}_i = (C_{i1}, C_{i2}, \dots, C_{id}), \tag{3.1}$$

где C_{ij} – количество ресурса j, которым обладает узел i; $C_{ij} \geq 0$. Всего в распределенной вычислительной системе используется d типов ресурсов.

Работа описывается вектором потребностей в каждом типе ресурса:

$$\vec{J_i} = (J_{i1}, J_{i2}, \dots, J_{id}), \quad (3.2)$$

где J_{ij} – количество ресурса j, который требуется задаче i; $J_{ij} \geq 0$.

Целью алгоритма упаковки является нахождение разбиения B множества задач $J = (\vec{J_1}, \vec{J_2}, \dots, \vec{J_n})$ на минимальное количество контейнеров B ($J = \bigcup_{B_i \in B} B_i$) таким образом, чтобы выполнялось условие:

$$\forall B_i \in B: \sum_{\vec{J_i} \in B_i} \vec{J_i} \leq \vec{C}, \quad (3.3)$$

где $\vec{C}$ – вектор, описывающий максимальную емкость контейнера по каждому его измерению. Условие (3.3) задает требование, чтобы все задачи, расположенные в контейнере B_i, не требовали больше ресурсов, чем его емкость. Также предполагается, что все контейнеры B_i имеют одинаковую емкость. Для эффективной упаковки задач в контейнеры алгоритм оперирует пакетами задач (очередью). Алгоритм выполняет сортировку по некоторому критерию, а затем последовательно складывает задачи в контейнер. Если в контейнер не влезает очередная задача, то создается новый контейнер, и так до тех пор, пока все задачи не будут упакованы в контейнеры [7].

Контейнер B_i является d-мерным, и в размерности отсутствует компонента времени. Контейнер B_i может представлять разную сущность. Например, bin-packing problem может использоваться для построения алгоритмов планирования с вытеснением, и тогда B_i описывает квант времени, который выделяется операционной системой для выполнения части инструкций процессов. Емкость B_i описывает объем доступных ресурсов узла, на котором выполняется планирование, а длина всех задач, упаковываемых в этот контейнер, одинаковая и равняется кванту времени исполнения процесса. Тогда разбиение B описывает, какая часть какой задачи будет выполняться в какой промежуток времени [4].

В нашем случае мы строим алгоритм планирования без вытеснения, и к вышеописанной постановке задачи надо добавить время исполнения для каждой задачи. Выполним переход от bin-packing problem к задаче учета и выделения ресурсов. Для этого мы зафиксируем количество контейнеров как N – количество вычислительных узлов, т.е. каждый контейнер описывает один вычислительный узел:

$$C = (\vec{C}_1, \vec{C}_2, \dots, \vec{C}_N), \quad (3.4)$$

где C – множество вычислительных узлов; $\vec{C_i}$ – вектор, описывающий объем ресурсов каждого типа на узле i. Каждый узел имеет разный объем ресурсов каждого типа.

Задачу будем описывать кортежем:

$$J_i = \langle T_i, \vec{D}_i \rangle, \quad (3.5)$$

где T_i – время, необходимое для выполнения работы i; $\vec{D}_i$ – вектор требований к ресурсам размерности d, необходимых для запуска задачи.

Введем функцию учета количества доступных ресурсов в зависимости от времени:

$$\vec{U}_i(t) = \big(U_{i1}(t), U_{i2}(t), \dots, U_{id}(t)\big), \tag{3.6}$$

где $U_{ij}(t)$ – количество свободного ресурса типа j на узле i в момент времени t.

На значение данной функции влияет текущий план исполнения работ. Если на узле i не запланировано ни одной работы, то тогда:

$$\forall t' \in [0; \infty): \vec{U}_i(t') = \vec{C}_i. \tag{3.7}$$

Если на узле i запланирована одна работа, которая будет исполняться во временном промежутке $[0; a]$ и которая требует объем ресурсов $\vec{D}_i$, то тогда функция (3.6) будет принимать следующие значения:

$$\begin{gathered} \forall t_a \in [0; a]: \vec{U}_i(t_a) = \vec{C}_i - \vec{D}_i; \\ \forall t_b \in (a, \infty): \vec{U}_i(t_b) = \vec{C}_i. \end{gathered} \tag{3.8}$$

Далее опишем реализацию функции (3.6). Данную функцию будем реализовывать подобно алгоритмам из работ [3, 8], которые выполняют разбиение пространства (свободные ресурсы; время) на слоты. Слотом является промежуток времени, в котором определенный набор ресурсов доступен для использования. Алгоритмы из [3, 8] заключается в поиске такого слота, который обладает достаточным объемом ресурсов и дает самое раннее время завершения задачи.

Алгоритм и структура данных для реализации функции $U_i(t)$ и поиска свободных временных интервалов:

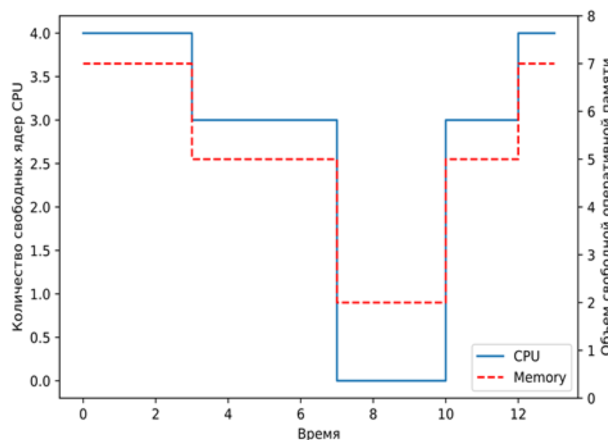


Рис. 3.1 Пример возможных значений функции $\vec{U}_i(t)$ для двух ресурсов: CPU и Memory

На рисунке 3.1 представлено возможное значение функции $\vec{U}_i(t)$ для двух ресурсов. Поскольку значение функции зависит от текущего расписания узла i, то для реализации данной функции предлагается использовать табличное представление, в котором строка таблицы описывает значение ресурсов во временном интервале. Для реализации табличного представления используется отсортированный связанный список, в котором каждый элемент описывает строку таблицы. Представление связанного списка показано на рисунке 3.2. Каждый элемент этого списка назовем **слотом**.

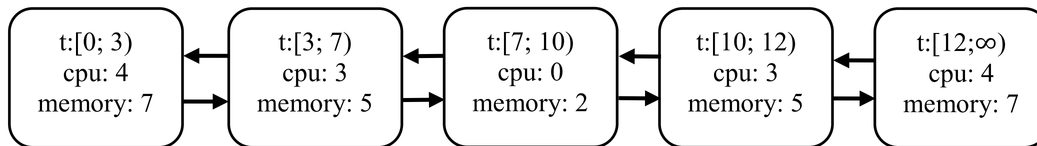


Рис. 3.2 Описание реализации табличного представления функции (показанной на рис. 3.1) в виде двусвязного списка

Для дальнейшего описания алгоритмов, которые выполняют манипуляции над слотами, рассмотрим используемые структуры данных. Каждый слот ($slot$) – это структура, которая содержит следующие атрибуты:

1. $slot.startTime$ – время начала возможной аренды ресурсов.
2. $slot.endTime$– время завершения возможной аренды ресурсов.
3. $slot.RA$ – отображение $RA: name \rightarrow resource$, переводящее название ресурса в структуру $resource$, описывающую объем ресурса. Отображение RA реализуется через хеш-таблицу. Структура $resource$, содержит следующие атрибуты:
   3.1. $resource.name$ – название ресурса.
   3.2. $resource.amount$ – объем доступного ресурса.

Слоты организуются в двусвязный список $slots$, в котором они расположены в порядке по возрастанию $slot.startTime$ и $slot.endTime$, слоты во времени не пересекаются, и в связанном списке нет пропусков по времени, то есть время завершения одного слота всегда совпадает со временем начала следующего слота.

Рассмотрим структуру $context$, которая предназначена для описания найденной последовательности слотов, которая будет резервироваться для запуска работы. Структура содержит следующие атрибуты:

1. $context.start$ – указатель на элемент из списка $slots$, с которого начинается резервирование ресурсов.
2. $context.end$ – указатель на элемент из списка $slots$, на котором заканчивается резервирование ресурсов.
3. $context.RA$ – какое количество ресурсов необходимо резервировать (здесь используется та же структура данных, что и в атрибуте $slot.RA$).

4. $context.startTime$ – время начала аренды ресурсов.
5. $context.endTime$ – время завершения аренды ресурсов.

Алгоритм 3.1 – Поиск последовательности слотов для дальнейшего резервирования (Функция FindSlot из раздела 2)

Вход:

1. slots – список слотов доступных для резервирования.
2. length – время, на которое необходимо арендовать ресурсы.
3. RA – хеш-таблица ресурсов, необходимая для резервирования.
4. timeStart – самое раннее время, с которого необходимо начать поиск свободных ресурсов.

Выход:

1. context – описание последовательности слотов, которые возможно зарезервировать.

Алгоритм:

1. Проходим по списку slots до тех, пор пока не найдем слот, с которого возможно начать поиск последовательности слотов или не переберем все слоты. Такой слот должен удовлетворять следующим условиям:
   1.1. $slot.startTime \leq timeStart \wedge slot.endTime > timeStart \wedge (slot.RA \geq RA)$.
   1.2. Обозначим найденный слот как $startSlot$.
   1.3. Если мы перебрали все слоты и не нашли ни одного подходящего, то завершаем работу алгоритма с информацией о том, что слот не найден.
2. $curentLength = startSlot.endTime - timeStart$ – длина обработанной последовательности ресурсов.
3. $currentRA = startSlot.RA$ – максимально доступное количество ресурсов в последовательности слотов (это минимальный объем ресурсов среди всех слотов).
4. Выполняем итерацию по связному списку $slots$, начиная с элемента, который идет после $startSlot$. Обозначим через $slot_i$ рассматриваемый элемент на текущей итерации:
   4.1. $currentLength = currentLength + slot_i.endTIme - -slot_i.startTime.$
   4.2. $currentRA = \min(currentRA; slot_i.RA)$.
   4.3. Проверяем, достаточно ли ресурсов в текущей последовательности слотов для запуска задания: $currentRA < RA$.
      4.3.1. Если ресурсов меньше, чем необходимо, то

$$startSlot = slot_{i+1}$$
$$currentLength = startSlot.endTime - startSlot.startTime$$

Переходим на следующую итерацию.

4.4. Проверяем длину текущей последовательности слотов: $currentLength \geq length$.

4.4.1. Если условие выполняется, то формируем структуру данных context и завершаем выполнение алгоритма:

$$context.start = startSlot$$
$$context.end = slot_i$$
$$context.startTime = max(timeStart; startSlot.startTime)$$
$$context.endTime = context.startTime + length$$
$$context.RA = RA$$

5. Проверяем длину текущей последовательности слотов: $currentLength \geq length$.

5.1. Если условие выполняется, то проверяем, достаточно ли ресурсов в текущей последовательности слотов для запуска задания: $currentRA \geq RA$.

5.1.1. Если условие выполняется, то формируем структуру данных context и завершаем выполнение алгоритма

$$context.start = startSlot$$
$$context.end = slot_i$$
$$context.startTime = max(timeStart; startSlot.startTime)$$
$$context.endTime = context.startTime + length$$
$$context.RA = RA$$

В алгоритме 3.1 используются следующие операции над таблицей ресурсов:

1. Бинарная операция сравнения для хеш-таблицы ($\geq, <$) – это поэлементное сравнение соответствующих значений. Все значения должны удовлетворять условию. Соответствие элементов устанавливается по ключу. Множество ключей двух таблиц совпадает.
2. $\min(currentRA; slot_i.RA)$ – как и в случае бинарной операции сравнения, выполняется попарное вычисление операции минимума. Пары формируются из значений двух таблиц с одинаковым ключом (по одному значению из каждой таблицы).

После того как мы нашли пространство на вычислительном узле (это пространство описывается структурой $context$), нам необходимо его зарезервировать для задачи. Резервирование ресурсов описано в алгоритме 3.2. Суть резервирования ресурсов заключается в коррекции значений структуры $slots$, которая показывает, какое количество ресурсов доступно для использования. Необходимо в соответствии с найденным временным

промежутком для запуска задачи, уменьшить доступный объем ресурсов в структуре $slots$ в соответствующем временном промежутке.

Алгоритм 3.2 — Резервирование ресурсов

Вход:
1. slots – список слотов, доступных для резервирования.
2. context – описание последовательности слотов, которые возможно зарезервировать.

Выход:
1. Модификация структуры данных slots.

Алгоритм:
1. Проверка на условие, необходимо ли разделить первый слот: $context.start.startTime \neq context.startTime$.
   1.1. Если условие выполняется, то выполняем разделение: $splitSlot(slots, context.start, context.startTime, context.endTime)$
2. Выполняем итерацию по связанном списку слотов, начиная со слота $context.start$. Итерирование по списку слотов выполняется до слота $context.end$, не обрабатывая его. Обозначим через $slot_i$ обрабатываемый элемент на шаге цикла.
   2.1. Уменьшаем количество ресурсов, доступное в слоте $slot_i$: $$slot_i.RA - context.RA.$$
3. Проверка на условие, необходимо ли разделить последний слот: $context.end.endTime \neq context.endTime$.
   3.1. Если условие выполняется, то выполняем разделение $$splitSlot(slots, context.end, context.startTime, context.en$$
4. Уменьшаем количество ресурсов, доступное в слоте $context.end$: $context.end.RA - context.RA$.

Алгоритм 3.2 использует бинарную операцию вычитания для хеш-таблиц ($slot_i.RA - context.RA$). Под этой операцией предполагается следующее:
1. Множество ключей двух таблиц совпадает.
2. Выполняется поэлементное вычитание объема ресурсов. То есть из значения объема ресурса A таблицы $slot_i.RA$ вычитается значение объема ресурса A таблицы $contex.RA$.

Также алгоритм 3.2 использует функцию $splitSlot$, которая описана в алгоритме 3.3.

Алгоритм 3.3 – Разделение слота (функция $splitSlot$)

Вход:
1. slots – связанный список слотов.
2. slot – указатель на элемент списка slots, который необходимо разделить.
3. startTime – начальное время, по которому необходимо разделить слот.
4. endTime — конечное время, по которому необходимо разделить слот.

Выход:
1. Модификация структуры данных slots.

Алгоритм:
1. Если $slot.startTime < startTime$, то:

$$newSlot = slot$$
$$slot.endTme = startTime$$
$$newSlot.startTime = startTime$$
$$slots.insertAfter(slot, newSlot)$$

2. Если $slot.endTime > endTime$, то:

$$newSlot = slot$$
$$newSlot.endTime = endTime$$
$$slot.startTime = endTime$$
$$slots.insertBefore(slot, newSlot)$$

– $slots.insertAfter(slot, newSlot)$ – вставка элемента $newSlot$ в связанный список $slots$ после элемента $slot$;
– $slots.insertBefore(slot, newSlot)$ – вставка элемента $newSlot$ в связанный список $slots$ перед элементом $slot$.

Алгоритм выделения ресурсов:

После того как мы забронировали ресурсы на заданное время исполнения работы, необходимо дождаться времени освобождения ресурсов, а затем запустить работу на исполнение. Для этого предлагается использовать структуры данных и принцип их обработки, представленный в Алгоритме 3.4. Алгоритм 3.5 содержит описание алгоритма выделения ресурсов, который выполняется при запуске задач.

Алгоритм 3.4 – Учет ресурсов

1. Для отслеживания свободных и занятых ядер процессора достаточно хранить целое положительное число свободных ядер. Когда запускается задача, уменьшаем это число на количество потоков, которое задача запланировала использовать. На каких ядрах будут запущены потоки, операционная система Linux решит самостоятельно.
2. Для описания доступной оперативной и дисковой памяти подход аналогичен описанию свободных ядер процессора.

3. Для GPU поддерживаем два списка. Список занятых GPU и список свободных GPU.
   3.1. Когда запускается задача, которая потребляет n графических ускорителей, то из списка свободных GPU извлекаем n идентификаторов Ids.
   3.2. Идентификаторы из списка Ids перекладываются в список занятых GPU.
   3.3. При запуске работы идентификаторы Ids передаются работе как параметры запуска, чтобы процесс правильно настроил использование нужных GPU, установленных на вычислительном узле.
   3.4. Как только некоторая работа, использовавшая GPU, завершает свое исполнение, то идентификаторы GPU, которые были использованы этой задачей из списка занятых GPU, перекладываются в список свободных.
4. Для MTU и FPGA подход учета аналогичен подходу для GPU.

Алгоритм 3.5 – Выделение ресурсов

1. Каждый вычислительный узел i имеет очередь работ Q_i.
2. Каждая работа в очереди Q_i представлена картежем J (5), который описывает, в какой промежуток времени и какой объем ресурсов задача будет занимать. Очередь отсортирована по возрастанию времени начала исполнения задач.
3. В очередь Q_i добавляется очередная задача после ее планирования алгоритмом PMR HEFT (алгоритм описан в разделе 2).
4. Из начала очереди Q_i задача извлекается и запускается, как только требуемый объем ресурсов становится доступен. При проверке доступности и запуске задачи выполняется модификация структур данных, описывающих свободные и занятые ресурсы как описано в алгоритме 3.4.

4. Пример работы алгоритма

В данном разделе рассмотрим результат работы алгоритма применительно к периодической задаче, представленной на рисунке 4.1. В графе имеются фиктивные вершины, которые автоматически добавляются к графу задачи перед тем, как запустить алгоритм планирования, с целью, чтобы в графе существовал один исток и один сток. Задача состоит из 4 работ и двух источников данных изображений. Это камеры, которые формируют кадры с

частотой 30 (источник frames1) и 25 (источник frames2) кадров в секунду. Источники данных в графе обозначены через овалы, работа – это прямоугольник, приемник данных – параллелограмм. Фиктивные узлы обозначены через шестиугольник. Для каждого источника данных в квадратных скобках обозначен период формирования новых данных. Каждая работа может иметь несколько конфигураций, каждая из которых требует свой набор ресурсов. Набор ресурсов для конфигурации работы обозначен в квадратных скобках. Для каждой конфигурации набор ресурсов указан в отдельной строке. Например, работа job1 имеет две конфигурации. Первая конфигурация требует одно ядро центрального процесса (CPU) и один графический укоритель (GPU). Вторая конфигурация работы требует 4 ядра центрального процессора.
Каждое ребро графа имеет метки:

- в начале ребра стоит название выходного аргумента из задачи;
- в конце ребра стоит название входного аргумента в задачу;
- в круглых скобках указано название типа данных, который передается по ребру.

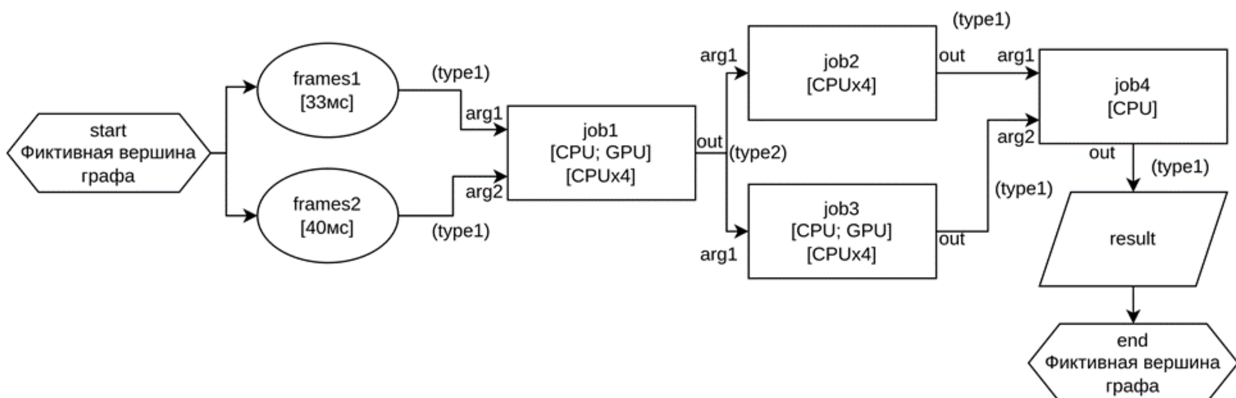


Рис. 4.1 Граф задачи, который будет планироваться алгоритмом PMR HEFT

Для примера были выбраны два типа данных, которые обладают следующими характеристиками:

- type1 имеет объем, равный 6 220 800 байт (RGB изображения разрешения 1920x1080 без сжатия) – время передачи такого объема данных по сети с пропускной способностью 1 гигабит в секунду равняется 49 мс:

$$1\ \text{Гигабит} = 10000000\ \text{бит} = 125000000\ \text{байт}$$
$$\frac{6220800\ \text{байт}}{125000000\ \text{байт/с}} \to \frac{6220800\ \text{байт}}{125000\ \text{байт/мс}} \approx 49\ \text{мс} \tag{4.1}$$

  (вычисления в реализации алгоритма выполняются в целых числах)
- type2 имеет объем, равный 500000 байт. Время передачи ≈ 0 мс;
- В алгоритме используется параметр «Накладные расходы на начало передачи данных», который в примере принят за 1 мс. Поэтому при

составлении расписания считается, что type1 передается по сети за 50 мс, а type2 за 1 мс.

Фиктивные вершины никаких данных не передают и нужны только на стадии расчета рангов вершин графа для выполнения топологической сортировки.

Для исполнения этого задания предполагается, что имеются 3 вычислительных узла, которые обладают следующими ресурсами (рис. 4.2):

– Первый ВУБ обладает 4-ядерным процессором и графическим ускорителем;
– Второй ВУБ обладает 4-ядерным процессором;
– Третий ВУБ обладает 4-ядерным процессором.

Источник данных frames1 – камера, расположенная на втором ВУБ, а источник данных frames2 расположен на третьем ВУБ. Пропускная способность сети – 1 гигабит в секунду.

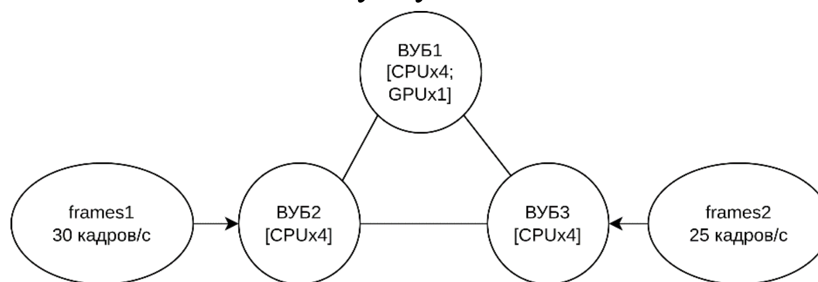


Рис. 4.2 Схема вычислительных узлов. Для каждого ВУБ указаны его вычислительные ресурсы, показано какая камера к какому узлу подключена, приведена топология сети с пропускной способностью 1 гигабит/с

После выполнения стадии топологической сортировки (сортировки списка задачи в соответствии с вычисленными рангами) работы будут планироваться в следующем порядке: job1; job3; job2; job4. На рисунке 4.3 показана диаграмма Ганта при планировании одной периодической задачи, граф которой представлен на рисунке 4.1. Было выполнено планирование 5 экземпляров задачи, каждый из которых обрабатывает собственный набор входных данных, состоящий из двух изображений.

Каждое изображение формируется своим источником (frame1 и frame2), которые расположены на ВУБ2 и ВУБ3. Планирование выполнялось с отступом в 100 мс от текущего времени (текущее время считаем за 0 мс). Каждая работа выделена цветом. Каждый цвет соответствует своему экземпляру задачи. Составленное расписание исполнения задач имеет следующую длительность:

– первый экземпляр (белый цвет) обрабатывает изображения за 181 мс;
– второй экземпляр (красный цвет) обрабатывает изображения за 221 мс;
– третий экземпляр (синий цвет) обрабатывает изображение за 221 мс;

– четвертый экземпляр (зеленый цвет) обрабатывает изображение за 221 мс;
– пятый экземпляр (желтый цвет) обрабатывает изображение за 221 мс.

Каждый экземпляр задачи (из пяти представленных) планируется последовательно. Используется информация только об уже запланированных работах, при планировании работы не учитывается тот факт, что в очереди на планирование существуют еще работы. Таким образом, получается, что при планировании первого экземпляра задания вычислительный кластер полностью свободен и есть возможность использовать графический ускоритель для всех работ, для которых существует соответствующая конфигурация. Для первого экземпляра задачи в расписании используется графический ускоритель для работ job1, job3. При планировании остальных экземпляров задачи на графическом ускорителе запускается только работа job3. Это связано с тем, что в момент составления расписания ищется такое размещение работ, при котором уменьшается время ожидания получения результата. Графический ускоритель занят, и, таким образом, результат от работы job1 будет получен быстрее на центральном процессоре, чем если дождаться освобождения графического ускорителя и запустить эту работу на нем. Таким образом, GPU является критическим ресурсом, так как он все время занят, а моменты простоя связаны с доставкой данных на вычислительный узел.

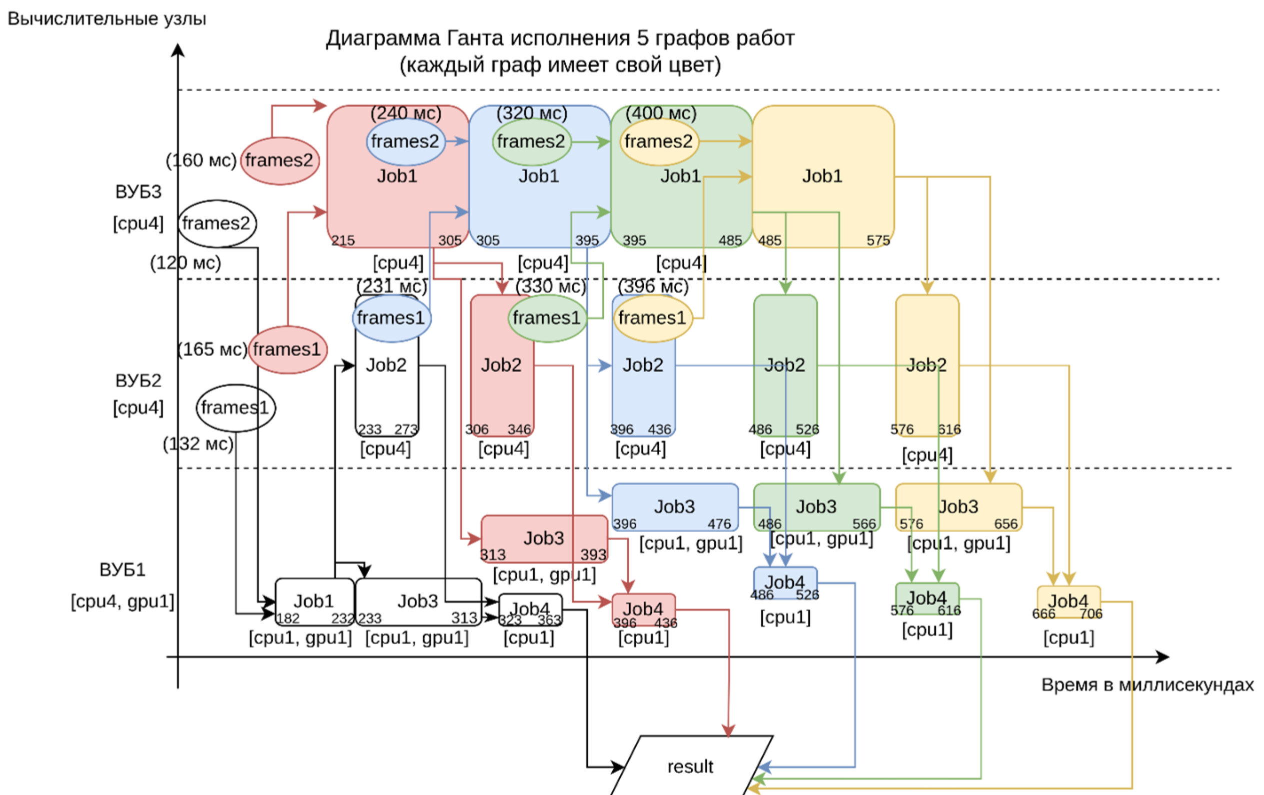


Рис. 4.3 Диаграмма Ганта. План исполнения 5 экземпляров одной периодической задачи (легенда представлена на рис. 4.4)

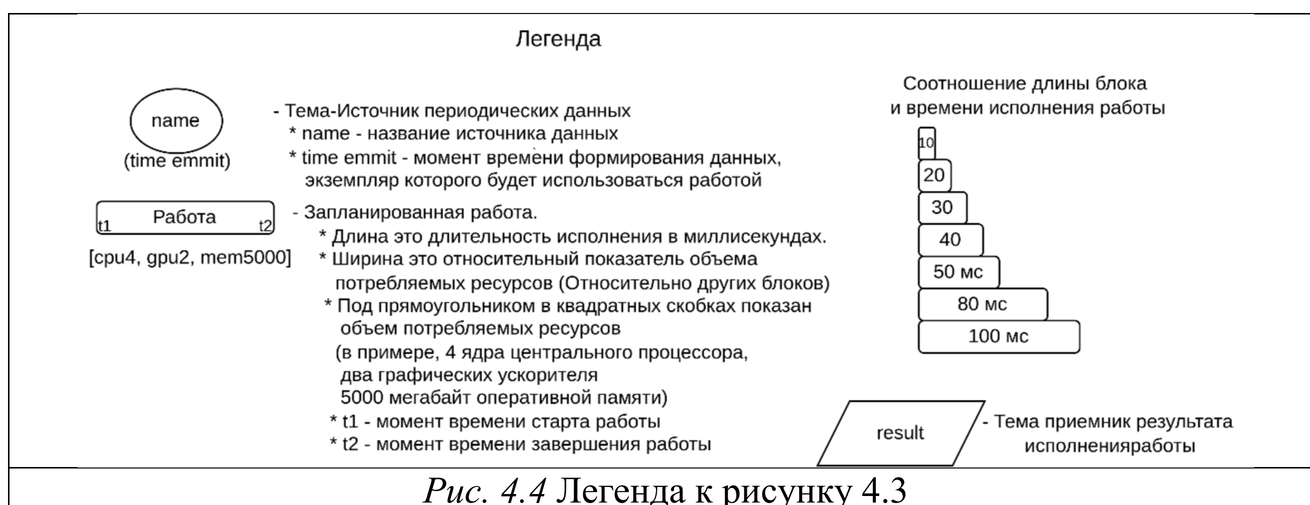


Рис. 4.4 Легенда к рисунку 4.3

Заключение

На основании алгоритма HEFT, который выполняет топологическую сортировку графа связанных работ, был разработан и реализован алгоритм планирования Periodic Multiple Resources HEFT (PMR HEFT). Этот алгоритм позволяет составлять расписание исполнения периодических заданий, для выполнения которых требуется определенный набор ресурсов. Каждая работа может иметь несколько реализаций, каждая из которых требует свой набор ресурсов. Использование различных реализаций одной работы (конфигураций) позволяет сократить время ожидания завершения исполнения заданий, поскольку нет необходимости дожидаться освобождения определенного набора ресурсов, так как для исполнения работы будет выбран другой набор вычислительных ресурсов. Разработанный алгоритм позволяет реализовать конвейерную обработку потока сенсорных данных.

Список источников и литературы

1. Агеев А.В., Богуславский А.А., Соколов С.М. Планирование заданий в бортовой вычислительной системе // Препринты ИПМ им. М.В.Келдыша. 2023. № 43. 27 с. https://doi.org/10.20948/prepr-2023-43 https://library.keldysh.ru/preprint.asp?id=2023-43
2. Ghodsi A. et al. Dominant resource fairness: Fair allocation of multiple resource types //8th USENIX symposium on networked systems design and implementation (NSDI 11). – 2011.
3. Управление параллельными заданиями в гриде с неотчуждаемыми ресурсами / В. Н. Коваленко [и др.] // Препринты ИПМ им. М.В.Келдыша. 2007. № 63. 28 с. https://library.keldysh.ru/preprint.asp?id=2007-63

4. Ousterhout J. K. et al. Scheduling Techniques for Concurrent Systems //ICDCS. – 1982. – Т. 82. – С. 22-30.
5. Topcuoglu, Haluk; Hariri, Salim; Wu, M. (2002). "Performance-effective and low-complexity task scheduling for heterogeneous computing". IEEE Transactions on Parallel and Distributed Systems. **13** (3): 260–274. *CiteSeerX* *10.1.1.119.122*. *doi*:*10.1109/71.993206*. *S2CID* *17773509*.
6. Лазарев Д.О., Кузюрин Н.Н. Об онлайновых алгоритмах для задач упаковки в контейнеры и полосы, их анализе в худшем случае и в среднем. Труды Института системного программирования РАН. 2018;30(4):209-230. https://doi.org/10.15514/ISPRAS-2018-30(4)-14
7. Leinberger W., Karypis G., Kumar V. Job scheduling in the presence of multiple resource requirements // Proceedings of the 1999 ACM/IEEE conference on Supercomputing. – 1999. – С. 47-es.
8. Toporkov V. V. et al. Resource co-allocation algorithms in distributed job batch scheduling // Ubiquitous Computing and Communication Journal. – 2012. – Т. 7. – №. 2. – С. 1232.

Оглавление

1. Введение 3
2. Модифицированный алгоритм HEFT – Periodic Multiple Resources HEFT....... 6
3. Алгоритмы учета, планирования и выделения ресурсов 21
4. Пример работы алгоритма.. 29
Заключение.. 33
Список источников и литературы 33