Министерство образования и науки Российской Федерации
федеральное государственное автономное образовательное учреждение
высшего образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Институт кибернетики
Направление подготовки 09.03.01 «Информатика и вычислительная техника»
Кафедра информационных систем и технологий

БАКАЛАВРСКАЯ РАБОТА

Тема работы
Исследование стратегий балансировки нагрузки в системах распределенной обработки данных

УДК № 004.75:004.03

Студент

Группа	ФИО	Подпись	Дата
8В3Б	Нагиев Андрей Евгеньевич		

Руководитель

Должность	ФИО	Ученая степень, звание	Подпись	Дата
Доцент каф. ИСТ	Ботыгин И.А.	к.т.н.		

КОНСУЛЬТАНТЫ:

По разделу «Финансовый менеджмент, ресурсоэффективность и ресурсосбережение»

Должность	ФИО	Ученая степень, звание	Подпись	Дата
Доцент каф. МЕН	Спицын В.В.	к.э.н.		

По разделу «Социальная ответственность»

Должность	ФИО	Ученая степень, звание	Подпись	Дата
Ассистент каф. ЭБЖ	Невский Е.С.	-		

ДОПУСТИТЬ К ЗАЩИТЕ:

Зав. кафедрой	ФИО	Ученая степень, звание	Подпись	Дата
ИСТ	Мальчуков А.Н.	к.т.н.		

Томск – 2017

ЗАПЛАНИРОВАННЫЕ РЕЗУЛЬТАТЫ ПО ОСНОВНОЙ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЕ ПОДГОТОВКИ БАКАЛАВРОВ 09.03.01 «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА», ИК ТПУ, ПРОФИЛЬ «СИСТЕМЫ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ»

Код результатов	**Результат обучения (выпускник должен быть готов)**	**Требования ФГОС, критерии АИОР**
Р1	Применять базовые и специальные естественнонаучные и математические знания в области информатики и вычислительной техники, достаточные для комплексной инженерной деятельности.	Требования ФГОС (ОК-7, ОПК-5, ПК-3), критерий 5 АИОР (п. 1.1) 06.019, специалист по технической документации в области информационных технологий; 06.027, Специалист по администрированию сетевых устройств информационно-коммуникационных систем
Р2	Применять базовые и специальные знания в области современных информационных технологий для решения инженерных задач.	Требования ФГОС (ОК-7, ОПК-2, 5, ПК-1, 3), критерий 5 АИОР (п.1.1, 1.2) 06.001, программист; 06.003, архитектор программного обеспечения.
Р3	Ставить и решать задачи комплексного анализа, связанные с созданием аппаратно-программных средств информационных и автоматизированных систем, с использованием базовых и специальных знаний, современных аналитических методов и моделей.	Требования ФГОС (ОК-6, ОПК-1, ПК-2, 4, ПК-6), критерий 5 АИОР (п. 1.2) Профессиональные стандарты (код, название): 06.001, программист; 06.028, Системный программист; 06.027, Специалист по администрированию сетевых устройств информационно-коммуникационных систем
Р4	Разрабатывать программные и аппаратные средства (системы, устройства, блоки, программы, базы данных и т. п.) в соответствии с техническим заданием и с использованием средств автоматизации проектирования.	Требования ФГОС (ОК-7, ОПК-2, 4, ПК- 1, 2, ПК-6), критерий 5 АИОР (п. 1.3) Профессиональные стандарты (код, название): 06.001, программист; 06.028, Системный программист; 06.027, Специалист по администрированию сетевых устройств информационно-коммуникационных систем
Р5	Проводить теоретические и экспериментальные исследования, включающие поиск и изучение необходимой научно-технической информации, математическое моделирование, проведение эксперимента, анализ и интерпретация полученных данных, в	Требования ФГОС (ОК-5, ОПК-5, ПК-1, 2, 3), критерий 5 АИОР (п.1.4) Профессиональные стандарты (код, название): 06.001, программист; 06.028, Системный программист; 06.027, Специалист по администрированию сетевых устройств информационно-

	области создания аппаратных и программных средств информационных и автоматизированных систем.	коммуникационных систем
Р6	Внедрять, эксплуатировать и обслуживать современные программно-аппаратные комплексы, обеспечивать их высокую эффективность, соблюдать правила охраны здоровья, безопасность труда, выполнять требования по защите окружающей среды.	Требования ФГОС (ОК-8, 9, ОПК-1, 2, 4, ПК-3, 4, 5, ПК-6), критерий 5 АИОР (п. 1.5) Профессиональные стандарты (код, название): 06.001, программист; 06.003, архитектор программного обеспечения.
	Универсальные компетенции	
Р7	Использовать базовые и специальные знания в области проектного менеджмента для ведения комплексной инженерной деятельности.	Требования ФГОС (ОК-3, ОПК-3, 5), критерий 5 АИОР (п. 2.1) Профессиональные стандарты (код, название): 06.016, руководитель проектов в области информационных технологий
Р8	Владеть иностранным языком на уровне, позволяющем работать в иноязычной среде, разрабатывать документацию, презентовать и защищать результаты комплексной инженерной деятельности.	Требования ФГОС (ОК-5, 7, ПК-3, 4), критерий 5 АИОР (п. 2.2) Профессиональные стандарты (код, название): 06.019, специалист по технической документации в области информационных технологий
Р9	Эффективно работать индивидуально и в качестве члена группы, состоящей из специалистов различных направлений и квалификаций, демонстрировать ответственность за результаты работы и готовность следовать корпоративной культуре организации.	Требования ФГОС (ОК-2, 6, 7), критерий 5 АИОР (п. 2.3, 2.4) Профессиональные стандарты (код, название): 06.016, руководитель проектов в области информационных технологий
Р10	Демонстрировать знания правовых, социальных, экономических и культурных аспектов комплексной инженерной деятельности.	Требования ФГОС (ОК-1, 2, 3, 4, 5), критерий 5 АИОР (п. 2.5)
Р11	Демонстрировать способность к самостоятельному обучению в течение всей жизни и непрерывному самосовершенствованию в инженерной профессии.	Требования ФГОС (ОК-5, 7), критерий 5 АИОР (п. 2.6)

Министерство образования и науки Российской Федерации
федеральное государственное автономное образовательное учреждение
высшего образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Институт кибернетики
Направление подготовки 09.03.01 «Информатика и вычислительная техника»
Кафедра информационных систем и технологий

УТВЕРЖДАЮ:
Зав. кафедрой
_____ _____ Мальчуков А.Н.
(Подпись) (Дата) (Ф.И.О.)

ЗАДАНИЕ
на выполнение выпускной квалификационной работы

В форме:

Бакалаврской работы

Студенту:

Группа	**ФИО**
8В3Б	Нагиеву Андрею Евгеньевичу

Тема работы:

Исследование стратегий балансировки нагрузки в системах распределенной обработки данных	
Утверждена приказом директора (дата, номер)	

Срок сдачи студентом выполненной работы:	

ТЕХНИЧЕСКОЕ ЗАДАНИЕ:

Исходные данные к работе	Объектно-ориентированный язык программирования Java, пакет JDK 8, среда разработки Intellij IDEA.
Перечень подлежащих исследованию, проектированию и разработке вопросов	Анализ методов и средств распараллеливания вычислений и балансировки нагрузки; Исследование возможностей распараллеливания вычислений в многоядерных центральных процессорных устройствах и графических процессорных устройствах; Изучение уровней и задач распараллеливания вычислений; Описание механизмов и основных технологий распараллеливания на базе графических процессоров; Исследование основных целей, задач, аппаратных и программных решений балансировки нагрузки;

	Изучение методов и алгоритмов балансировки нагрузки; Разработка алгоритма балансировки нагрузки на основе динамического вычисления рангов вычислителей по заданным критериям; Проектирование вычислительных графов различных размерностей; Разработка функциональной структуры инструментальных средств имитационной среды моделирования; Проведение экспериментов по обработке задач имитационной средой моделирования с использованием балансировки нагрузки.
Перечень графического материала	Метод балансировки нагрузки Random; Метод балансировки нагрузки Weighted Random; Предлагаемый метод балансировки нагрузки в неоднородной вычислительной системе; Выполнение вычислительной задачи для одинаковых вычислителей, но с предпочтением разных характеристик для обработки; Сравнение результатов обработки методами балансировки нагрузки.

Консультанты по разделам выпускной квалификационной работы

Раздел	Консультант
Финансовый менеджмент, ресурсоэффективность и ресурсосбережение	Спицын Владислав Владимирович
Социальная ответственность	Невский Егор Сергеевич

Дата выдачи задания на выполнение выпускной квалификационной работы по линейному графику	

Задание выдал руководитель:

Должность	ФИО	Ученая степень, звание	Подпись	Дата
Доцент кафедры ИСТ	Ботыгин И.А.	к.т.н.		

Задание принял к исполнению студент:

Группа	ФИО	Подпись	Дата
8В3Б	Нагиев Андрей Евгеньевич		

РЕФЕРАТ

Выпускная квалификационная работа содержит 165 с., 57 рис., 11 табл., 72 источника.

Ключевые слова: параллелизм, балансировка нагрузки, распределенные вычислительные системы, имитационное программное моделирование, Java, распределение вычислительной нагрузки.

Объектом исследования является распределение вычислительной нагрузки в многоядерных системах.

Цель работы – исследование стратегий балансировки нагрузки в системах распределенной обработки данных.

В процессе исследования были изучены и проанализированы основные существующие методы балансировки нагрузки, такие как: Random, Weighted Random, Round Robin, Weighted Round Robin, Agent-Based Adaptive, Geo-locating, Dynamic Weighted Round Robin, Data Center Aware, Token Aware, Least Connections, Weighted Least Connections, Dynamic Weighted Least Connection, Locality-Based Least Connection Scheduling, Locality-Based Least Connection Scheduling with Replication Scheduling, Least Response Time, Load-Based и т.д.

В результате исследования был разработан алгоритм балансировки нагрузки на основе динамического вычисления рангов вычислителей по заданным критериям. Частными случаями критериев являются: производительность процессора узла вычисления, его объем оперативной памяти и время доступа к узлу.

Для оценки эффективности спроектированного алгоритма балансировки, была разработана имитационная среда моделирования, в которой на основе генерируемого графа вычислений с различным числом обрабатываемых модулей с различной степенью параллелизма их выполнения, с генерацией времени выполнения каждого модуля на основе датчика псевдослучайных чисел, проводились имитационные эксперименты

по обработке графов вычислений с учетом различных видов балансировки нагрузки.

Степень внедрения: опытная эксплуатация имитационной среды моделирования.

Область применения: использование разработанных алгоритмов балансировки при решении вычислительно-ёмких задач в центрах обработки данных на высокопроизводительных кластерах.

Экономическая эффективность/значимость работы: проведенный анализ конкурентных технических решений по построенной оценочной карте показал то, что предлагаемый алгоритм балансировки нагрузки на основе динамического вычисления рангов вычислителей при дальнейшем исследовании и проектировании может стать конкурентным программным решением для целого ряда существующих алгоритмов балансировки.

В будущем планируется дальнейшая модификация и улучшение алгоритма балансировки нагрузки для сильно-связанных вычислительно-ёмких задач.

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ, СОКРАЩЕНИЯ И НОРМАТИВНЫЕ ССЫЛКИ

GPGPU (также GPGP, GP²U, General-Purpose computing for Graphics Processing Units – неспециализированные вычисления на графических процессорах) – технология, разработанная в 2001-м году, которая позволяет использовать возможности графического процессора (ГП) для обработки задач ЦП.

FLOPS (FLoating-point Operations Per Second) – количество операций с плавающей точкой в секунду.

Thread – поток выполнения, наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

HTT (Hyper-Threading Technology – технология гиперпоточности) – технология, разработанная компанией Intel, которая реализует идею «одновременной мультипоточности». При включении технологии каждое физическое ядро процессора определяется операционной системой как два логических ядра.

SMT (Simultaneous MultiThreading – одновременная многопоточность) – технология, которая позволяет исполнять инструкции из нескольких независимых потоков выполнения на множестве функциональных модулей суперскалярного микропроцессора в одном цикле.

SFU (special function unit) – блоки для вычисления специальных функций, которые предназначены для вычисления сложных (например, тригонометрических) вычислений.

GDDR (Graphic Double Data Rate memory) – память с удвоенной скоростью передачи данных, которая предназначена для графической обработки, объем которой значительно больше объема кэш-памяти у центрального процессора.

DDR SDRAM (Double Data Rate Synchronous Dynamic Random Access Memory) – синхронная динамическая память с произвольным доступом и удвоенной скоростью передачи данных.

SIMT (Single Instruction stream / Multiple Thread) – микроархитектура, которая поддерживает один поток команд, множество нитей**.**

SIMD (Single Instruction stream / Multiple Data stream) – микроархитектура, которая поддерживает один поток команд, множество потоков данных.

MIMD (Multiple Instruction stream / Multiple Data stream) – микроархитектура, которая поддерживает множество потоков команд, множество потоков данных.

CUDA (Compute Unified Device Architecture – архитектура вычислительного устройства с поддержкой универсальных вычислений) – аппаратно-программная платформа, разработанная компанией NVidia и предназначенная для разработки приложений с использованием параллельных вычислений и возможностей графической карты.

ГП, GPU (Graphics Processing Unit) – графический процессор.

ЦП, CPU (Central Processing Unit) – центральный процессор.

SSL (Secure Sockets Layer – уровень защищённых сокетов) – криптографический протокол, который подразумевает более безопасную связь.

OpenCL (Open Computing Language – открытый язык вычислений) – фреймворк, который является открытым стандартом и предназначен для унификации написания параллельных программ на графических и центральных процессорах.

OpenACC (Open Accelerators) – программный стандарт, который разработан компаниями PGI, NVidia, Cray, CAPS и предназначен для упрощения программирования программ, которые задействуют как центральный, так и графической процессор.

C++ AMP (C++ Accelerated Massive Parallelism) – библиотека, разработанная компаний Microsoft и которая позволяет реализовывать параллельные программы на языке C++, перенося часть вычислений на графические процессоры.

Java.util.concurrent – пакет, внедренный в JDK 7 и в котором определяются основные возможности альтернативных вариантов встроенных способов параллелизма (синхронизаторы, исполнители, параллельные коллекции, каркас Fork/Join Framework).

Оглавление

Введение 18
1 Описание методов и средств распараллеливания вычислений и балансировки нагрузки 22
1.1 Исследование возможностей распараллеливания вычислений в многоядерных центральных процессорных устройствах и графических процессорных устройствах 22
1.1.1 Уровни и задачи распараллеливания вычислений 22
1.1.2 Описание механизмов и основных технологий распараллеливания на базе графических процессоров 31
1.2 Обзор и сравнительный анализ алгоритмов балансировки нагрузки в многопроцессорных вычислительных системах 39
1.2.1 Основные цели, задачи, аппаратные и программные решения балансировки нагрузки 39
1.2.2 Методы и алгоритмы балансировки нагрузки 48
2 Разработка алгоритма балансировки нагрузки на основе динамического вычисления рангов вычислителей по заданным критериям 75
2.1 Проектирование вычислительных графов различных размерностей .. 75
2.2 Алгоритм балансировки нагрузки на основе динамического вычисления рангов вычислителей по заданным критериям 84
2.3 Задание характеристик вычислителей 92
3 Разработка функциональной структуры и инструментальных средств имитационной среды моделирования 96
3.1 Обобщенная функциональная структура программно-имитационной среды моделирования 96
4 Результаты экспериментов по обработке задач имитационной средой моделирования с использованием балансировки нагрузки 101
5 Социальная ответственность 111
6 Финансовый менеджмент, ресурсоэффективность и ресурсосбережение. 120
6.1 Область практического применения балансировки нагрузки 120

6.2 Анализ конкурентных технических решений 121
6.3 Определение возможных альтернатив проведения научных исследований 123
6.4 Планирование научно-исследовательских работ 128
6.4.1 Структура работы в рамках научного исследования 128
6.4.2 Определение трудоёмкости выполнения работ 128
Заключение 135
Список публикаций студента 137
Список использованных источников 139
Приложение А 148
Приложение Б 151
Приложение В 155
Приложение Г 157
Приложение Д 159
Приложение Е 162

Введение

В настоящий момент с интенсивным развитием многоядерных и многопроцессорных архитектур и постоянным увеличением количества ядер и одновременно работающих потоков выполнения на различных вычислительных устройствах, а также в связи с повышением скорости передачи по каналам связи, невозможно представить обработку сложных задач без использования распараллеливания вычислений.

Распараллеливание вычислений является наиболее простым и эффективным способом повышения скорости обработки данных и представляет собой разбиение задачи на удобные для дальнейшей обработки многоядерной системой составные части, то есть – модули. Таким образом, каждая составная часть (модуль) вычислительной задачи может обрабатываться отдельным ядром или процессором системы в отдельном потоке выполнения, что значительно повышает скорость вычислений. Ввиду этого, распределенные вычислительные системы играют весомую роль в повышении быстродействия и уменьшении времени обработки.

Одним из важнейших компонентов в высокопроизводительных вычислительных кластерах является диспетчеризация вычислений на его узлах (балансировка). Тема выпускной квалификационной работы имеет непосредственную связь с балансировкой нагрузки в горизонтально масштабируемых, территориально распределенных системах обработки данных. Балансировка нагрузки позволяет значительно снизить финансовые затраты на проектировку горизонтально распределенной вычислительной системы и повышает эффективность использования имеющихся ресурсов.

Ввиду высокой практической значимости балансировки существует большое количество различных технологий, алгоритмов и способов распределить нагрузку между вычислителями, эффективность которых зависит от имеющихся ресурсов, типов обрабатываемых данных и сложности поставленной задачи, например: Round Robin, Least Connection, Weighted

Round Robin и т.д. Классификация методов балансировки нагрузки, их подробное описание, а также преимущества и недостатки каждого метода приведены далее в теоретической части настоящей выпускной квалификационной работы.

Над балансировкой нагрузки в распределенных вычислительных системах в настоящий момент работают крупнейшие мировые исследовательские центры и IT-компании, среди которых: Amazon (Elastic Load Balancing), Oracle (Real Application Clusters, Grid Infrastructure), Microsoft (Windows Clustering), Google (Cloud Load Balancing) и т.д.

Существует бесчисленное множество задач, для вычисления которых применяется балансировка нагрузки. Среди задач, которые требуют применения балансировки нагрузки, имеются такие, как: анализ спутниковых метеоданных и прогнозирование климатических изменений, моделирование городских условий на примере большого города, предсказание различных экстремальных процессов и явлений, моделирование кровеносной системы тела человека, анализ радиосигналов из радиотелескопов для поиска внеземных цивилизаций и т.д. [1, 2] Всё эти задачи являются достаточно трудоёмкими и требуют постоянного использования больших вычислительных ресурсов. Без использования балансировки нагрузки невозможно было бы максимально эффективно и быстро просчитывать столь сложные задачи.

В ходе выполнения выпускной квалификационной работы на различных программно-генерируемых имитационных моделях вычислительных задач апробирован предлагаемый метод балансировки нагрузки, который основан на определенных характеристиках производительности каждого вычислителя. Данный метод описан далее в пояснительной записке выпускной квалификационной работы. Он представляет собой еще один способ равномерно распределить нагрузку между вычислителями, а также максимально эффективно использовать имеющиеся ресурсы для своевременного получения результатов, справляясь

с изменениями сложности вычислительной задачи и используя всю гибкость имеющейся системы при вычислениях и обработке данных на пиковых нагрузках. Метод позволяет динамически подключать имеющиеся вычислительные узлы на основе следующих характеристик: тактовой частоты процессора (-ов), объема оперативной памяти, а также время доступа к вычислителю.

Статистическая информация программных экспериментов обработана и представлена в качестве результатов работы.

Цель работы состоит в исследование стратегий балансировки нагрузки в системах распределенной обработки данных и оценке эффективности предлагаемого метода балансировки нагрузки на основе генерируемой программно-имитационной модели вычислительной задачи с различным числом обрабатываемых модулей, с различной степенью их параллелизма и генерацией времени выполнения каждого модуля на основе датчика псевдослучайных чисел.

Разработанная программная имитационная среда моделирования позволяет оценить результат применения алгоритма балансировки на графах вычислений с числом вершин до двух тысяч.

Различные исследования, проведенные автором, по схемам и алгоритмам балансировки нагрузки докладывались и обсуждались на следующих конференциях:

1) VIII Сибирская конференция «Параллельные и высокопроизводительные вычисления», 28.10.2015 г., г. Томск;

2) XIII Международная научно-практическая конференция студентов, аспирантов, молодых ученых «Молодежь и современные информационные технологии», 09.11.2015 г., г. Томск;

3) IX Международная научно-техническая конференция «Современные проблемы машиностроения», 01.12.2015 г., г. Томск;

4) XIII Всероссийская научно-практическая конференция студентов, аспирантов и молодых ученых «Технологии Microsoft в теории и практике программирования», 23.03.2016 г., г. Томск;

5) III Международная научная конференция «Информационные технологии в науке, управлении, социальной сфере и медицине», 24.05.2016 г., г. Томск;

6) VIII Международная научно-практическая конференция «Физико-технические проблемы в науке, промышленности и медицине», 03.06.2016 г., г. Томск;

7) XIV Международная научно-практическая конференция студентов, аспирантов, молодых ученых «Молодежь и современные информационные технологии», 11.11.2016 г., г. Томск.

1 Описание методов и средств распараллеливания вычислений и балансировки нагрузки

В настоящей главе рассматриваются возможности применения параллельных вычислений с использованием различных вычислительных ресурсов, а также алгоритмы и методы распределения вычислительной нагрузки на архитектурах, допускающих параллельные независимые вычисления.

1.1 Исследование возможностей распараллеливания вычислений в многоядерных центральных процессорных устройствах и графических процессорных устройствах

1.1.1 Уровни и задачи распараллеливания вычислений

На текущий момент большинство задач и алгоритмов, требующих вычисления, может быть разделено на определенные наборы одновременно вычисляемых подзадач. Таким образом, каждый набор общей задачи вычисляется в своём потоке выполнения на своём ядре или процессоре и не влияет на скорости выполнения других наборов в других потоках. Именно эта технология легла в основу распараллеливания вычислений. Распараллеливание вычислений значительно увеличивает скорость выполнения задачи.

Распараллеливание – не единственный способ повысить скорость вычислений. Альтернативой ему является увеличение тактовой частоты процессорных устройств. Однако данный способ является гораздо более дорогим и энергозатратным ввиду того, что помимо необходимости уменьшения длины или разделения инструкций, которые обрабатывает процессор, стоит учитывать еще один факт – при увеличении тактовой частоты процессора в два раза, отвод тепла от процессора необходимо увеличить в восемь раз [3]. Всё вышеперечисленное, несомненно, усложняет

проектирование подобных вычислительных систем, повышает себестоимость разработки, цену самой продукции и удорожает её обслуживание из-за дополнительных систем охлаждения и отвода тепла.

Цель распараллеливания вычислений состоит в уменьшении времени обработки вычислений. Это особенно критично для задач, выполняющихся в масштабе реального времени, т.к. необходимо учитывать следующие критерии:

1) Время отклика системы на внешние воздействия, т.е. время реакции системы [4]. Чем меньше данное время, тем выше степень эффективности обработки.

2) Способность выдерживать определенные строго заданные интервалы между запуском задачи и получение результата [5]. На основе данного критерия эффективности, системы реального времени подразделяются на следующие виды:

а) жесткие, где превышение заданного интервала вычислений ведет к отказу системы;

б) твердые, где допускается небольшой выход за пределы интервала;

в) мягкие, где эффективность системы снижается при выходе за интервал, однако система продолжает выполняться [6].

Таким образом, в системах реального времени даже небольшая задержка, и несвоевременное предоставление результатов обработки может повлечь за собой серьезные проблемы.

Существуют различные уровни параллелизма вычислений, между которыми нет строгих границ. Каждый уровень параллелизма тем или иным образом может затрагивать и соседние уровни.

Приведем краткое описание каждого уровня:

1) Параллелизм на уровне задач [7] – применяется в случаях, при которых задача или процесс явным образом состоят из определенных независимых частей (подзадач, подпроцессов и так далее). Данный уровень параллелизма предоставляет операционная система, которая запускает на

различных ядрах вычислительной системы несвязанные между собой задачи или приложения.

Пример параллелизма на уровне задач представлен на рисунке 1.

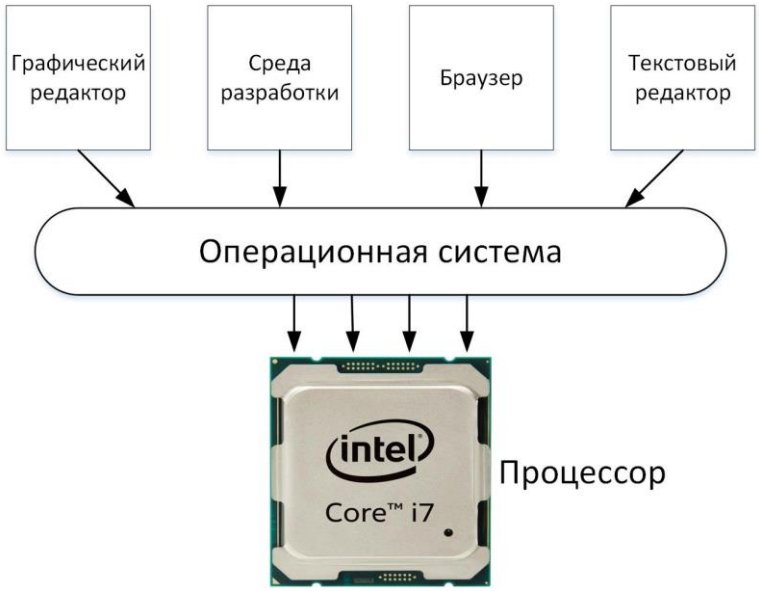

Рисунок 1 – Параллелизм на уровне задач

На рисунке 1 изображено параллельное выполнение на отдельных ядрах процессора следующих приложений: браузера, текстового редактора, среды разработки графического редактора. При этом, как уже было сказано, выполнение каждого приложения осуществляется независимо от других.

Существенным недостатком данного метода является невозможность распараллеливания однородной задачи, состоящей из зависимых между собой частей ввиду того, что каждая часть программы будет ожидать выполнение другой. В таком случае распараллеливание только уменьшит скорость выполнения. Для распараллеливания однородной задачи применяют более низкоуровневые виды параллелизма.

2) Параллелизм на уровне данных [7] – применяется в случае выполнения одной и той же задачи для множества (массива) данных. На данном уровне происходит разбиение данных на блоки, которые

обрабатываются на различных вычислительных узлах (ядрах и процессорах) системы.

Пример применения параллелизма на уровне данных показан на рисунке 2.

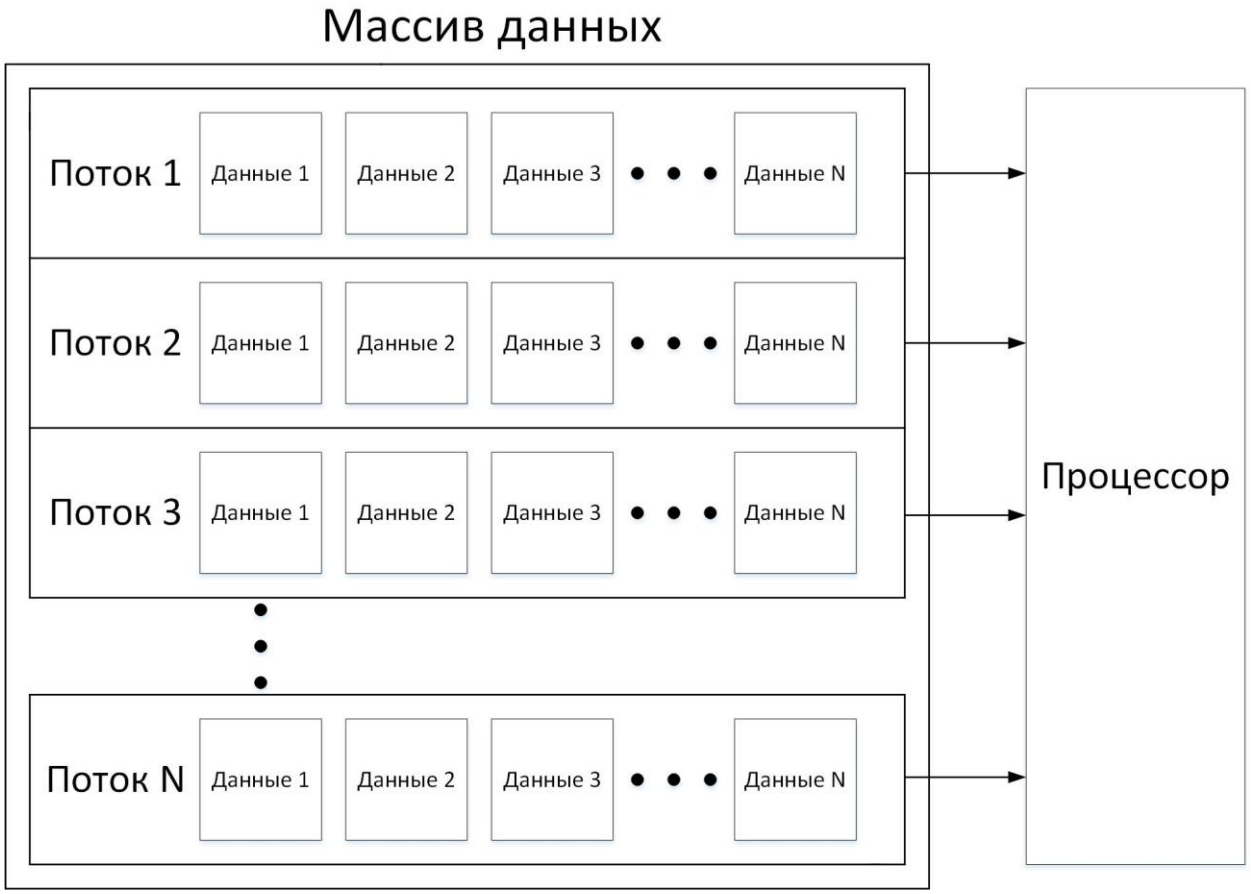

Рисунок 2 – Параллелизм на уровне данных

На рисунке 2 изображен массив данных, для всех элементов которого необходимо выполнить одинаковые операции. Для этого данный массив разбивается на составные части. Процессор производит выполнение каждой части в отдельном потоке, что значительно увеличивает скорость обработки массива.

Примером параллелизма на уровне данных является программная модель MapReduce, которая разработана компанией Google [8] и позволяет обрабатывать петабайты данных в распределенных компьютерных кластерах.

Данная модель состоит из двух шагов: map и reduce, название которых совпадает с названием функций высших порядков.

Работа фреймворка MapReduce представлена на рисунке 3.

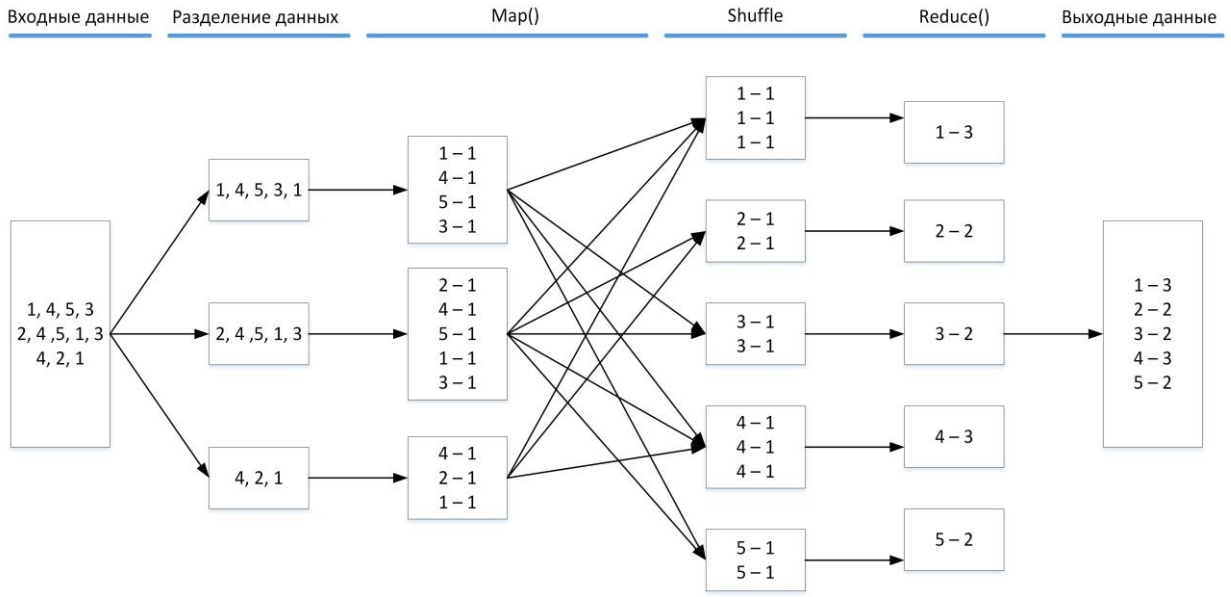

Рисунок 3 – Пример работы фреймворка MapReduce

Исходя из данных рисунка 3 принцип работы MapReduce следующий:

I. Входные данные разделяются на определенные фрагменты главным узлом системы (master node) для повышения удобства последующей обработки. Главный узел передает разделенные части другим вычислительным устройства – рабочим узлам (worker node).

В описанном на рисунке 3 случае, входные данные – это двумерный список чисел от 1 до 5. С помощью программной модель MapReduce необходимо подсчитать количество повторений каждого числа в списке. Для этого, главный узел сначала разделяет список на составные части – одномерные списки.

II. Рабочие узлы применяют функцию Map() ко всем элементам входных данных. Функция Map() либо вернет ноль, либо создаст экземпляры ключ–значение объектов [9].

Для каждого числа создается пара ключ-значение. Ключом является число, а значением – количество заданных чисел в списке.

III. Рабочие узлы сортируют (shuffling) все пары ключ–значение, при этом создав новые экземпляры объектов, где все значения будут сгруппированы по ключу [9].

Ключом в данном случае является число из списка входных данных.

IV. Главный узел системы принимает данные от рабочих узлов и выполняет операцию Reduse() – свертку (т.е. подсчет одинаковых пар ключ-значение) обработанных данных для получения результата. Значение ключа при этом не меняется.

У подхода MapReduce имеются следующие преимущества: возможность параллельного выполнения операций обработки (Map) и свертки (Reduce), причем, ограничение в параллелизме зависит только от имеющихся вычислительных узлов; многократное выполнение операций Map() и Reduce() [10]; повышенная отказоустойчивость, так как при отказе одного узла его работу всегда может выполнить другой при условии доступности данных.

3) Параллелизм на уровне алгоритмов – применяется в алгоритмах параллельной сортировки, при умножении матриц, решении линейных уравнений [7].

Данный уровень состоит из различных наборов директив компилятора и библиотечных процедур. Таким образом, при распараллеливании программа начинается как один поток, но в определенных участках текущий поток создает дополнительные потоки, которые параллельно выполняют вычисления этого участка программы. После вычисления участка, работу продолжает только главный поток, получив результаты от дополнительных потоков. Описываемый уровень параллелизма достаточно сложен для реализации, но обладает высокой гибкостью и возможностями распараллеливания.

Параллелизм на уровне алгоритмов представлен на рисунке 4.

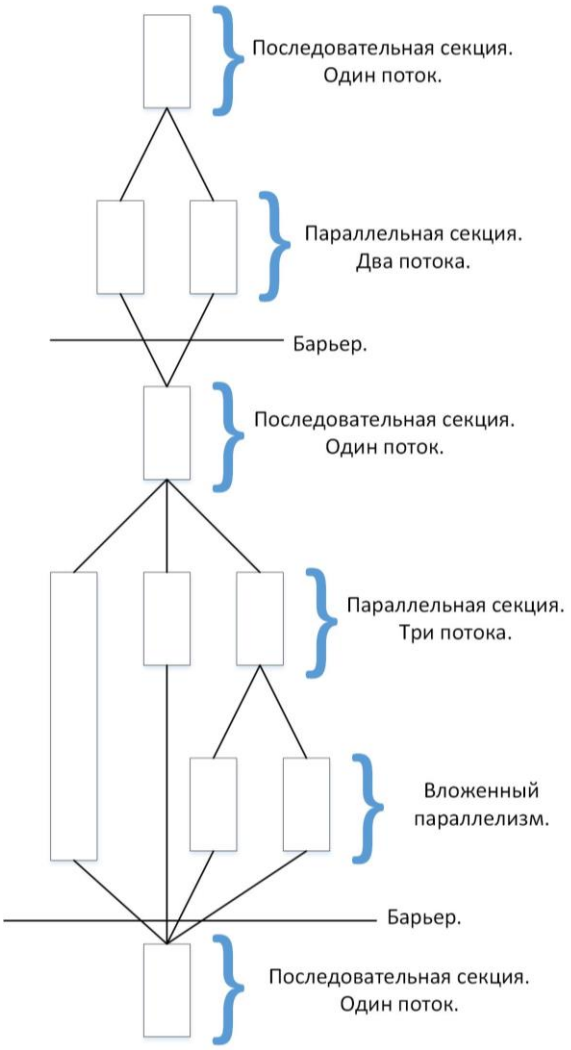

Рисунок 4 – Параллелизм на уровне алгоритмов

На рисунке 4 можно увидеть определенные участки программы, на которых происходит разделение основного потока выполнения на дополнительные, которые позволяют выполнять вычисления параллельно. Дополнительный поток может также разделиться на потоки, что является вложенным параллелизмом. После параллельного вычисления дополнительные потоки закрываются, и вычисление продолжает только главный поток.

4) Параллелизм на уровне инструкций (команд) [7] – самый низкий уровень параллелизма. На данном уровне осуществляется параллельная обработка процессором определенного количества различных инструкций.

Программа в представляет собой последовательность инструкций, которые выполняются процессором. Основную часть распараллеливания выполняет компилятор, который имеет возможность изменить порядок выполнения инструкций, распределить инструкции для выполнения на различных ядрах, произвести объединение инструкций в группы.

Пример параллелизма на уровне инструкций представлен на рисунке 5.

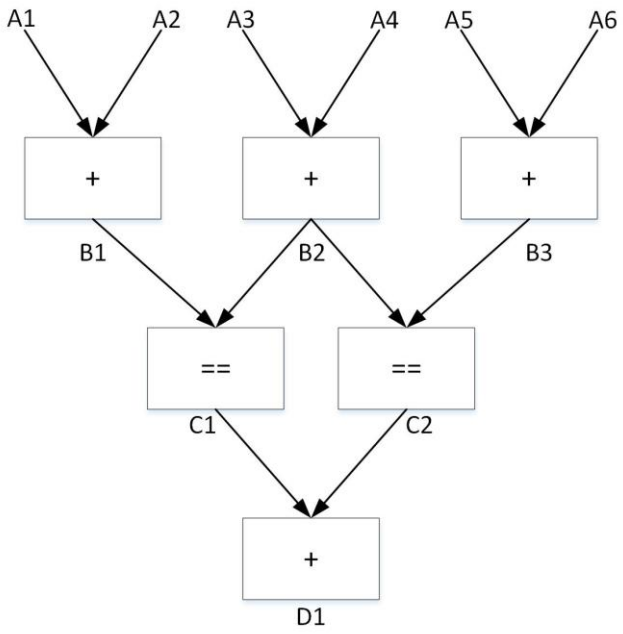

Рисунок 5 – Параллелизм на уровне инструкций

На рисунке 5 изображены следующие переменные: A1-A6, B1-B3, C1, C2 и D1. При этом можно увидеть, что выполнение операций сложения над переменными A1 и A2 не зависит от выполнения операций сложения над переменными A3, A4 и A5, A6. Это означает, что данные операции могут быть выполнены параллельно. Одновременно с ними не могут быть выполнены операции над переменным B1-B3 и над C1-C2 ввиду того, что их значения зависят от выполнения текущих операций A1-A6. Выполнив операции над переменными A1-A6, можно произвести параллельное сравнение переменных B1-B3. Затем, после выполнения данных операций будет выполнено сложение переменных C1 и C2.

Таким образом, если предположить, что 1 операция выполняется за 1 единицу времени, то 6 операций будет выполнено за 3 единицы времени, в отличие от выполнения операций без наличия параллелизма, когда количество выполняемых инструкций в данном примере по числовому значению совпадало бы с количеством единиц времени, которое необходимо для их выполнения. Т.е., при отсутствии параллелизма 6 инструкций было бы выполнено за 6 единиц времени. Прирост в скорости обработки инструкций в вышеописанном примере составляет 2 раза.

Для того чтобы понять основы распараллеливания, стоит также описать основную задачу параллелизма. Параллелизм вычислений предназначен для повышения скорости обработки данных, что позволяет использовать его для высокопроизводительных численных расчетов (задачи прогнозирования погоды, моделирование процессов мировой экономики, вычисление взаимодействия лекарственных препаратов с рецепторами мозга и т.д.) [11].

Однако у параллелизма имеются и определенные недостатки. Основной недостаток параллелизма – сложность реализации, которая ложится на плечи программиста ввиду дополнительного условия: программа должна не просто выполняться, а выполняться параллельно на различном числе процессоров, ядер и потоков, с учетом того, что данная программа запускается на вычислительных системах с различной конфигурацией узлов и с различной архитектурой. Каждый процессор обладает своим процессорным временем, что дополнительно усложняет задачу программирования на многопроцессорных системах. Необходимо учитывать возможность борьбы за ресурсы между потоками при разработке программ и различную скорость передачи данных по каналам связи, а также разное расстояние между узлами распределенной вычислительной системы.

Также недостатком параллельного программирования является, тот факт, что, даже несмотря на то, что в системе может быть доступно несколько процессоров, другие задачи (и сама операционная система) будут соперничать с прикладной программой за время центрального процессора

(ЦП, CPU – central processing unit). Поэтому не стоит особенно полагаться на то, что у прикладной программы будет неограниченный доступ ко всем имеющимся в системе процессорам. Кроме того, различные процессоры в одной и той же программе могут показать разные характеристики времени выполнения из-за отличий в загруженности задачами [12].

1.1.2 Описание механизмов и основных технологий распараллеливания на базе графических процессоров

Помимо распараллеливания посредством возможностей центрального процессора существует также возможность распараллелить вычисления с помощью графической карты вычислительного устройства. Для этого используется технология GPGPU.

GPGPU (также GPGP, GP²U, General-Purpose computing for Graphics Processing Units – неспециализированные вычисления на графических процессорах) – технология, разработанная в 2001-м году, которая позволяет использовать возможности графического процессора (ГП) для обработки задач ЦП [13].

Отличительной особенностью технологии GPGPU является возможность двунаправленной передачи данных между центральным и графическим процессором, что значительно увеличивает скорость обработки вычислений и позволяет более гибко использовать возможности вычислительного устройства при разработке параллельно-вычисляемых программ.

Применение технологии GPGPU стало возможным после добавления к возможностям графического процессора использование программируемых шейдеров (программы для вычисления средствами графических процессоров), которые позволяют выполнять вычисления общего назначения, используя для этого любое имеющееся в микросхеме

арифметико-логическое устройство (АЛУ), а также возможности выполнения операций над числами с плавающей точкой [14].

Технология GPGPU используется при обработке больших наборов данных; при анализе молекулярных формул и исследованиях белков в органической химии, что помогает при разработке лекарственных препаратов против рака и болезни Альцгеймера; в обработке изображений компьютерного зрения; при проектировании дизайна автомобилей, самолетов и т.д. [15].

Примерами реализации GPGPU являются технологии:

1) CUDA от компании NVidia;
2) ATI Stream Technology от компании AMD;
3) OpenCL от компании Apple;
4) OpenACC от компаний CAPS, Cray, NVidia и PGI;
5) C++ Accelereated Massive Parallelism (C++ AMP) от компании Microsoft и т.д.

Преимуществом параллелизма посредством графической карты является наличие гораздо большего количества ядер и потоков, чем количество ядер и потоков в CPU. Например, современная графическая карта от компании NVidia – GTX TITAN Xp имеет в наличии 3840 скалярных потоковых процессоров (streaming processors) (ядер, которые поддерживают технологию CUDA и OpenCL), которые объединены в 30 мультипроцессоров (streaming multiprocessors) по 128 ядер в каждом с тактовой частотой 1583 МГц [16, 17]. В свою очередь, современный процессор i7 7700K на базе микроархитектуры Kaby Lake от компании Intel имеет 4 ядра и 8 потоков с тактовой частотой 4,5 ГГц [18].

В таблице 1 приведены сравнительные характеристики FLOPS (FLoating-point Operations Per Second – количество операций с плавающей точкой в секунду) процессора Intel i7 7700K архитектуры Kaby Lake [19, 20] и видеокарты NVidia GTX TITAN Xp [21].

Таблица 1 – Сравнение производительности между ЦП и графической картой

Производительность	Intel i7 7700K, GFLOPS	GTX TITAN Xp, GFLOPS
FP32	133,87	11366,4
FP64	94,8	355,2

Примечание: FP32 – число одинарной точности; FP64 – число двойной точности.

Исходя из данных таблицы 1, можно сделать вывод, что, несмотря на повышенную тактовую частоту ЦП, производительность графической карты значительно выше, что позволяет увеличить скорость обработки данных в вычислительной системе.

Потоковые процессоры играют непосредственную роль в обработке данных. Помимо потоковых процессоров в графической карте имеются дополнительные модули – блоки для вычисления специальных функций (SFU – special function unit), которые предназначены для вычисления сложных (например, тригонометрических) вычислений [22].

В зависимости от модели графической карты, конфигурация и количество мультипроцессоров меняется. При этом также меняется и количество потоковых процессоров в каждом мультипроцессоре.

Производительность при обработке чисел одинарной точности значительно выше производительности при обработке чисел двойной точности в графической карте ввиду того, что в мультипроцессоре графической карты блоков для выполнения чисел двойной точности гораздо меньше, чем одинарной, при этом, скорость самих блоков также ниже.

Большое количество ядер связано с тем, что графическая карта предназначена в первую очередь для обработки и анализа изображений, что требует выполнения одинаковых операций над каждым выводимым пикселем. Также преимуществом видеокарты является наличие собственной быстродействующей памяти (GDDR – Graphic Double Data Rate memory – память с удвоенной скоростью передачи данных, которая предназначена для

графической обработки), объем которой значительно больше объема кэш-памяти у центрального процессора. Доступ к памяти графическим процессором осуществляется быстрее, чем доступ центрального процессора к оперативной памяти (DDR SDRAM - Double Data Rate Synchronous Dynamic Random Access Memory – синхронная динамическая память с произвольным доступом и удвоенной скоростью передачи данных), т.е. графическая карта имеет повышенную пропускную способность относительно центрального процессора.

GDDR обладает более низким энергопотреблением (для ГП порядка 10 ГФлопс/Вт, для ЦП порядка 1 ГФлопс/Вт [23]) и тепловыделением относительно DDR, повышенной частотой и использует специальные методы управления буфером ввода-вывода, что увеличивает пропускную способность. [24]

Однако стоит отметить недостаток графических ядер, который заключается в изначальной приспособленности только к вычислениям направленным на обработку графики, что значительно сужает круг вычислительных задач, которые могут быть обработаны с помощью технологии GPGPU.

Графические процессоры поддерживают только SIMT (Single Instruction stream / Multiple Thread – Один поток команд, множество нитей) архитектуру, которая является частью SIMD [25] архитектуры по классификации М. Флинна [26] (Single Instruction stream / Multiple Data stream – Один поток команд, множество потоков данных). Это означает, что один и тот же набор команд выполняется для всех входных данных.

Устройство SIMD-архитектуры представлено на рисунке 6.

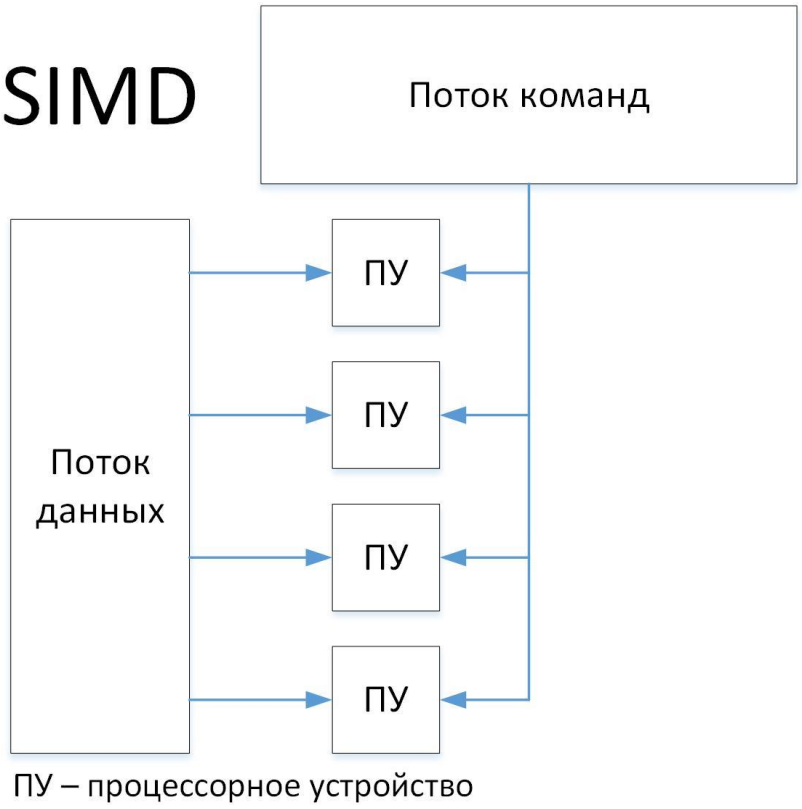

Рисунок 6 – Устройство SIMD-архитектуры

Данная архитектура достаточно эффективна при обработке графики и вычислении больших объемов данных с применением одинаковых операций над ними. Однако имеется только определенный спектр задач, который позволяет эффективно использовать данную архитектуру и управлением потоком тяжелых задач нельзя ускорить при помощи SIMD [27].

В свою очередь, ядра центрального процессора обладают более высокой гибкостью и поддерживают архитектуру MIMD (Multiple Instruction stream / Multiple Data stream – Множество потоков команд, множество потоков данных) по классификации М.Флинна [26].

Архитектура MIMD позволяет одновременно выполнять разные команды для различных данных, что значительно увеличивает спектр возможных задач распараллеливания в сравнении с архитектурой SIMD.

Устройство архитектуры MIMD представлено на рисунке 7.

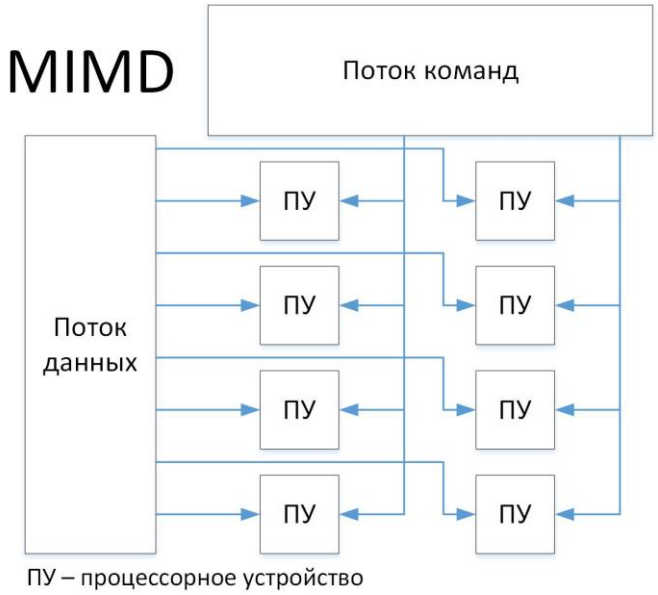

Рисунок 7 – Устройство MIMD-архитектуры

Таким образом, рассмотрев особенности параллелизма на основе графической карты можно сделать вывод о том, что графическая карта позволяет значительно повысить скорость на уровне параллелизма данных, однако при выполнении сложных к распараллеливанию задач или при одновременном выполнении различных операций над данными нельзя обойти без центрального процессора.

Далее подробно рассмотрим основные технологии, которые входят в состав GPGPU:

1) CUDA (Compute Unified Device Architecture – архитектура вычислительного устройства с поддержкой универсальных вычислений) – аппаратно-программная платформа, разработанная компанией NVidia и предназначенная для разработки приложений с использованием параллельных вычислений и возможностей графической карты. Релиз данной платформы состоялся 15 февраля 2007-го года на базе видеокарт GeForce 8-й серии [28].

Помимо самой графической карты, платформа включает в себя драйвер CUDA (который включает также драйвер GPU (Graphics Processing Unit) – графический процессор), CUDA SDK (набор средств для разработки

приложений на платформе) и CUDA Toolkit (содержит библиотеки для работы с платформой, а также компилятор nvcc для преобразования исходного кода программы в промежуточный ассемблерный код) [29].

Цель CUDA – предоставить разработчику возможности использовать GPU для параллельного программирования.

В качестве языка программирования используется язык Си, который дополнен определенными командами, позволяющими производить вычисления с помощью графической карты. Платформа CUDA поддерживает OpenCL с 2010 года, когда был выпущен CUDA Toolkit 3.0.

В настоящий момент существует большое количество API (Application Programming Interface – программный интерфейс приложения) CUDA на различных языках программирования:

а) Java – JCUDA [30]

б) Ruby, Python – KappaCUDA [31];

в) Haskell – Data.Array.Accelerate [32];

г) .NET – CUDA.NET [33] и т.д.

2) ATI Stream Technology (ранее ATI FireStream и AMD Stream Processor) – аппаратно-программная платформа, которая разработана компанией AMD (Advanced Micro Devices, Inc.) и является конкурентом CUDA. Принцип работы ATI Stream Technology аналогичен принципу работы CUDA: ряд программно-аппаратных оптимизаций, быстрые операции с плавающей точкой, двусторонняя передача данных, поддержка памяти с коррекцией ошибок, непосредственный доступ к памяти, наличие собственного AMD APP SDK (который включает в себя OpenCL, драйвер Catalyst Omega, а также библиотеку Open Source C++ под названием «Bolt» и библиотеку для поддержки OpenCV (Open Source Computer Vision – библиотека компьютерного зрения с открытым исходным кодом) [34].

Отличием является микроархитектура графических карт компании AMD от компании NVidia. В настоящее время компанией AMD используется микроархитектура Vega (6-е поколение архитектуры GCN (Graphic Core

Next) – графическое ядро следующего поколения), тогда как NVidia использует собственную микроархитектуру Pascal [35].

3) OpenCL (от англ. Open Computing Language – открытый язык вычислений) – фреймворк, который является торговой маркой компании Apple [36] и разработан компанией совместно с промышленным консорциумом Kronos Group в 2008-м году. OpenCL поддерживается видеокартами компаний AMD и NVidia.

Фреймворк является полностью открытым стандартом и предназначен для унификации написания параллельных программ на графических и центральных процессорах. Создание стандарта связано с тем, что раньше вычисления на графических картах от разных производителей было программно несовместимым и значительно усложняло задачу при проектировании многопоточных приложений на вычислительных кластерах [37].

В состав фреймворка входит язык программирования C на базе стандарта C99 и API. OpenCL обеспечивает параллелизм как на уровне инструкций, так и на уровне данных. Существует большое количество API, применяемых для различных языков программирования. Например, для языка программирования Java используется пакет JavaCL.

4) OpenACC (Open Accelerators) – программный стандарт, который разработан компаниями PGI, NVidia, Cray, CAPS и предназначен для упрощения программирования программ, которые задействуют как центральный, так и графической процессор [38].

Благодаря имеющемуся набору директорий с помощью данного стандарта можно выделить те участки программы, которые необходимо распараллелить или выполнить с применением графического процессора. Стандарт использует языки C++, C и Fortran [39].

5) C++ Accelerated Massive Parallelism (C++ AMP) – библиотека, разработанная компаний Microsoft и которая позволяет реализовывать

параллельные программы на языке C++, перенося часть вычислений на графические процессоры.

При этом если программа не может исполняться на графических процессорах, то она автоматически запускается на центральном процессоре [23].

Таким образом, в настоящий момент имеется значительно количество библиотек, технологий, фреймворков и аппаратно-программных платформ, который позволяют максимально эффективно использовать возможности параллелизма как графических, так и центральных процессоров в гетерогенных вычислительных системах.

Помимо компаний AMD и NVidia активно разработку аппаратных графических средств ведет Intel, используя технологию Intel HD Graphics – интеграцию центральных и графических процессоров. В настоящий момент встроенный графический процессор имеет до 72 исполнительных устройств с частотой до 1150 МГц и производительностью до 1152 GFLOPS. Стоит также отметить, что технология Intel HD Graphics поддерживает OpenCL.

1.2 Обзор и сравнительный анализ алгоритмов балансировки нагрузки в многопроцессорных вычислительных системах

1.2.1 Основные цели, задачи, аппаратные и программные решения балансировки нагрузки

Увеличение количества потоков, ядер и процессоров в вычислительных устройствах, применение параллелизма на различных уровнях системы, а также появление аппаратных и программных возможностей использовать параллелизм вычислений посредством графической карты значительно увеличили скорость обработки задач и расширили возможности систем обработки данных.

Использование распараллеливания помогает максимально эффективно применять имеющуюся вычислительную технику. Однако стоит отметить особенность, что наращивание вычислительной мощности не всегда приводит к увеличению производительности и уменьшению времени обработки задачи. Иногда случаются моменты, когда большое количество ядер, процессоров и потоков начинают конфликтовать и бороться за данные и ресурсы, значительно снижая скорость обработки информации и ухудшая показатели вычислительной системы. Примером из реальной жизни является забег на стадионе. Если будет бежать один человек, и никто ему не будет мешать, то он покажет куда лучший результат, нежели, если на этом же стадионе вместе с ним побежит еще пятьдесят человек ввиду того, что ему придется обгонять других людей, маневрировать между другими бегунами, меняя скорость и сбивая дыхание.

Таким образом, для каждого проекта и для каждой трудоёмкой вычислительной задачи необходимо учитывать такое понятие, как горизонтальная масштабируемость. Система должна справляться как с ростом нагрузки, так и с добавлением новых ресурсов в систему.

Масштабируемость является линейной, если отношение увеличения производительности к добавленным ресурсам близко к единице [40]. Достичь единицы в реальной системе не представляется возможным.

Стоит также учитывать то, что помимо борьбы за ресурсы в распределенной вычислительной системе может возникнуть ситуация, когда один вычислительный узел принимает всю нагрузку на себя, а остальные узлы простаивают. Это значительно снижает скорость, увеличивает время обработки данных и сводит на нет все преимущества параллельной обработки на различных узлах.

В качестве решения проблемы возникновения конфликтов между процессорами или ядрами используется балансировка нагрузки – понятие, которое неразрывно связано с распараллеливанием вычислений. Балансировка нагрузки позволяет оптимизировать вычисления и

синхронизировать процессоры, ядра и отдельные вычислительные узлы для согласования обмена информацией в горизонтально-масштабируемой архитектуре информационных и вычислительных ресурсов сложных систем, а также позволяет повысить отказоустойчивость.

В качестве «балансировщика» нагрузки выступает специальный диспетчер балансировки (load balancer) (рисунок 8), который программно позволяет распределять нагрузку от поступающих запросов между вычислительными узлами, что повышает гибкость и быстродействие системы.

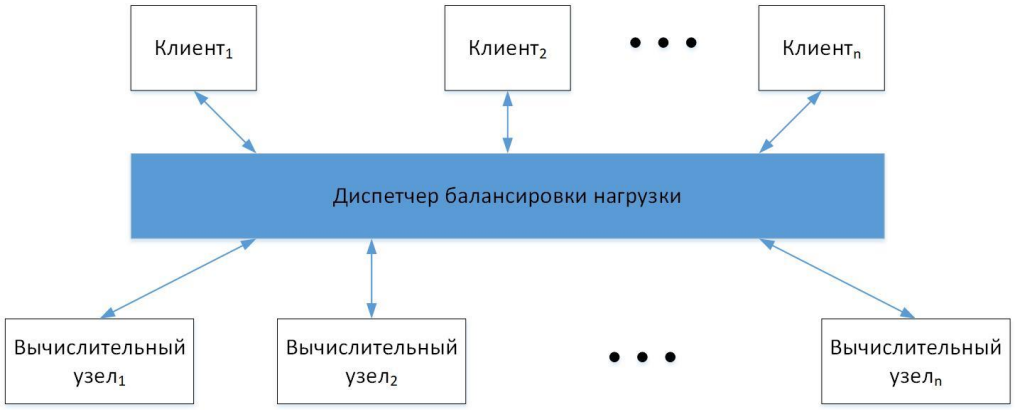

Рисунок 8 – Применение диспетчера балансировки нагрузки

Работа диспетчера балансировки нагрузки основана на выполнении определенных алгоритмов балансировки, эффективность которых зависит как от самой вычислительной системы, так и от задачи, которая будет выполняться. Таким образом, диспетчер балансировки нагрузки выступает в качестве посредника между клиентами, которые отправляют запрос на обработку и вычислительными узлами, которые производят его обработку и отправляют результат обратно.

При выборе метода балансировки нагрузки следует руководствоваться целями, которые необходимо достичь [41]:

1) справедливость – гарантия того, что все вычислительные запросы будут обработаны своевременно, без возникновения ситуации, при которой

происходит постоянная обработка запроса от определенного клиента при простое в очереди остальных;

2) эффективность – максимально эффективная загрузка существующих вычислительных узлов без простоя в ожидании запросов на обработку;

3) сокращение времени выполнения запроса – обеспечение минимального времени от отправки запроса на вычисления, до завершения обработки запроса;

4) сокращение времени отклика – минимизация времени ответа на запрос.

Также, необходимо, чтобы алгоритм балансировки нагрузки обладал следующими свойствами [41]:

1) предсказуемость;

2) равномерная загрузка имеющихся ресурсов вычислительной системы;

3) сохранение свойств алгоритма при увеличении масштаба системы и нагрузки на вычислительные узлы.

Балансировка нагрузки может быть как программной, так и аппаратной. Аппаратная балансировка нагрузки представлена решениями:

1) Cisco (CSS L4, ACE L7) [42];

2) Citrix NetScaler [43];

3) F5 BIG-IP [44];

4) Radware ADC (Application Delivery Controller) [45] и т.д.

Преимуществом аппаратных решений является широкий функционал, более глубокие механизмы балансировки, основанные на различных уровнях модели OSI (open systems interconnection basic reference model – базовая эталонная модель взаимодействия открытых систем), а также встроенные средства безопасности (например, встроенный SSL (Secure Sockets Layer) – уровень защищённых сокетов, а также защита от Dos-атак (denial-of-service) – атак на отказ в обслуживании, т.е. доведение системы до отказа) [42].

Недостатком является высокая цена, которая гораздо больше цен на программные решения.

Помимо аппаратной балансировки существует огромное количество различных средств программной балансировки нагрузки. Основное ПО (программное обеспечение) балансировки нагрузки представлено следующими решениями:

1) LVS (Linux Virtual Server) – средство управления высоконадежными и высокопроизводительными кластерными системами посредством балансировки нагрузки, которое работает в ОС (операционная система) Linux. Архитектура кластера является полностью прозрачной. Взаимодействие пользователей с системой происходит так, как будто вычислительный кластер представлен одним высокопроизводительным виртуальным сервером [46].

Работа LVS основана на двух одинаково настроенных вычислителях: активном LVS-маршрутизаторе и резервном LVS-маршрутизаторе. Активный LVS-маршрутизатор выполняет задачи балансировки нагрузки и проверку работоспособности на каждом вычислительном узле. Резервный LVS-маршрутизатор берет на себя аналогичные функции в случае выхода активного LVS-маршрутизатора из строя [47];

2) Nginx [engine x] – это HTTP-сервер (HyperText Transfer Protocol – протокол передачи гипертекста), обратный прокси-сервер, почтовый прокси-сервера, а также TCP (Transfer control protocol – протокол управления передачей)/UDP (User Datagram Protocol — протокол пользовательских датаграмм) прокси сервер общего назначения, который позволяет эффективно распределять нагрузку и поддерживать высокую отказоустойчивость для высоконагруженных сайтов (Netflix, Wordpress,com, Mail.ru, Yandex.ru и т.д.). Согласно статистике Netcraft nginx обслуживает 28,72% наиболее высоконагруженных сайтов (по состоянию на апрель 2017-го года) [48];

3) HAProxy – это быстрый и надежный прокси-сервер, который предлагает балансировку нагрузки, на основе протоколов TCP и HTTP. В настоящий момент поддерживается большим количеством наиболее посещаемых веб-сайтов (Twitter, Reddit, Stack Overflow, GitHub, Tumblr и т.д.) [49];

4) Perlbal – свободное ПО, обратный прокси- и веб-сервер, написанный на языке программирования Perl, и предлагающий использование балансировки нагрузки [50].

Имеется несколько аппаратно-программных способов балансировки нагрузки относительно различных сетевых механизмов и уровней сетевой модели OSI:

1) Уровень 4 (транспортный) сетевой модели OSI. Virtual Server via NAT (Network Address Translation — преобразование сетевых адресов).

Данный вид NAT'а был создан специально для балансировки нагрузки между вычислительными узлами, которые работают по TCP-протоколу. Трансляция срабатывает только при инициировании соединения со стороны клиента.

Пример использования NAT'а для балансировки нагрузки представлен на рисунке 9.

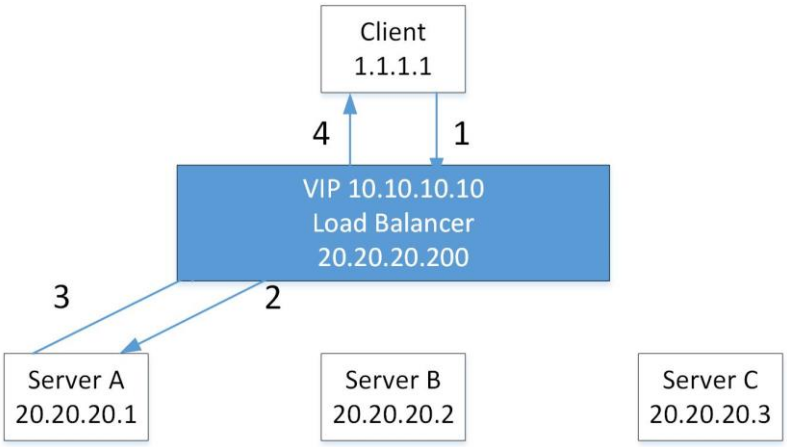

Рисунок 9 – Применение NAT для балансировки нагрузки

На основе рисунка 9 можно увидеть использование NAT. Имеется три web-сервера (20.20.20.1 – 20.20.20.3), на которых расположен один и тот же веб-сайт (например, это frontend-сервера). Порт у серверов один (например, HTTP – 80). Допустим, система балансирует с помощью метода Round Robin (который описан далее).

Клиент отправляет запрос на веб-сайт (прямая трансляция) и сначала происходит проверка, является ли входящий запрос TCP-трафиком. Если это не TCP-трафик, то он будет отправлен на первый адрес в пуле, если это TCP-трафик, то происходит проверка на соответствие NAT. При соответствии адрес назначения меняется на следующий в пуле. Трансляция заносится в список трансляций и пакет с измененным адресом подвергается маршрутизации. Далее выполняется обратная трансляция на сторону клиента.

У метода имеется следующее преимущество – сервера могут быть в разных физических сетях, но также имеется недостаток – большая нагрузка на процессор, весь трафик идет через диспетчер балансировки нагрузки [51].

2) Уровень 4 (транспортный) сетевой модели OSI. Virtual Server via IP (Internet Protocol – межсетевой протокол) Tunneling.

IP-туннелирование предназначено для транспортировки порций данных сетевым протоколом, в котором инкапсулирован другой, более низкий сетевой протокол [52].

Отличительной особенностью текущего способа от способа балансировки посредством NAT является то, что диспетчер балансировки нагрузки отправляет запрос на реальные вычислительные узлы, используя IP-туннелирование.

Пример применения IP-туннелирования для балансировки нагрузки представлен на рисунке 10.

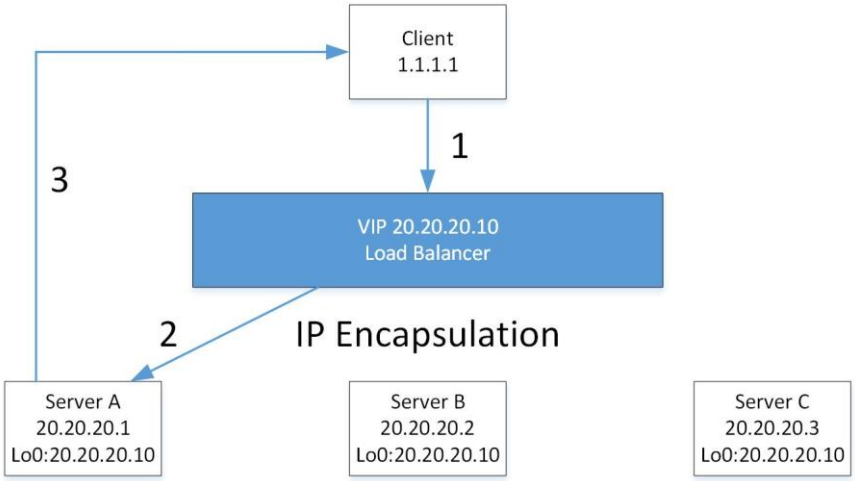

Рисунок 10 – Применение туннелирования для балансировки нагрузки

Исходя из данных рисунка 10, клиент отправляет запрос на вычислительный кластер. Кластер принимает пакет, который предназначен для виртуального IP-адреса. Диспетчер балансировки нагрузки проверяет адрес и порт назначения пакета и, если они совпадают, то происходит выбор реального вычислителя из кластера для обработки запроса в соответствии с работающим в системе алгоритмом балансировки нагрузки.

Диспетчер балансировки нагрузки инкапсулирует переданный клиентом пакет в IP-дейтаграмму и пересылает выбранному узлу.

Вычислительный узел принимает инкапсулированный пакет, производит его декапсуляцию и обработку. Результат, в отличие от способа балансировки посредством NAT, отправляется не обратно через диспетчер балансировки нагрузки, а непосредственно клиенту в соответствии с его таблицей маршрутизации.

Если соединение не завершено и отправляет новый пакет на обработку, то выбирается текущий вычислительный узел, который найден в хеш-таблице. После завершения соединения запись подключения удаляется их хеш-таблицы.

Преимуществом метода является то, что узлы могут иметь любой IP-адрес и не обязаны находиться в одной физической сети. Недостатком является достаточная сложность настройки [52].

3) Уровень 3 (сетевой) модели OSI. Virtual Server via Direct Routing.

Принцип балансировки нагрузки с применением прямой маршрутизации схож с предыдущими двумя методами, однако в нём не используется NAT или IP-туннелирование для инкапсуляции порций данных.

Использование прямой маршрутизации представлено на рисунке 11.

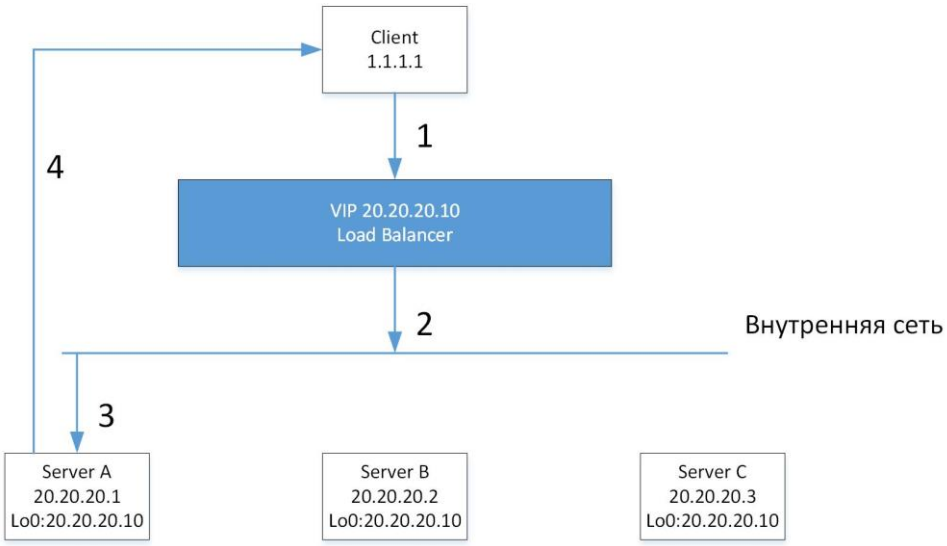

Рисунок 11 – Применение прямой маршрутизации для балансировки нагрузки

Преимуществом является высокая производительность метода и простота настройки, т.к. нет необходимости использовать дополнительные механизмы, и порции данных передаются непосредственно вычислителям диспетчером балансировки нагрузки на основе используемого алгоритма. Недостатком является то, что все вычислительные узлы должны находиться в одной физической сети [53].

Помимо транспортного и сетевого уровней модели OSI балансировка нагрузки может использоваться также в прикладном уровне (уровень 7). В отличие от низких уровней, которые описывают способность надежной передачи пакетов между узлами в сети, прикладной уровень взаимодействует с программным обеспечением, которое обменивается данными через сеть. Преимуществом прикладного уровня в балансировке нагрузки является то,

что он позволяет взаимодействовать с протоколами высокого уровня (например, HTTP), тогда как низкий уровень позволяет выполнять только простые операции: получение и переадресация трафика.

Прикладной уровень позволяет анализировать тип передаваемого трафика и выполняться переадресацию на основе анализа данных [54]. Примером применения прикладного уровня является веб-сервер Nginx (описанный ранее), который распределяет запросы между фронтедом и бэкендом, за что отвечает модуль Upstream [41]. Преимуществом использования прикладного уровня является наиболее равномерное распределение нагрузки и высокая гибкость при конфигурации. Недостатком является производительность, которая ниже, чем при использовании более низких уровней сетевой модели OSI.

1.2.2 Методы и алгоритмы балансировки нагрузки

Количество алгоритмов и методов балансировки нагрузки постоянно растет. При этом эффективность каждого алгоритма зависит от вычислительной системы и типов данных, для которых данный алгоритм будет использоваться. Приведем описание основных программных методов балансировки нагрузки:

1) Random – метод балансировки нагрузки, при котором нагрузка распределяется по вычислительным узлам системы в произвольном порядке, т.е. на основе данных датчика псевдослучайных чисел [55].

Датчик псевдослучайных чисел выбирает узел, на который отправляется текущий запрос на обработку. Далее выбирается другой любой узел.

Метод Random представлен на рисунке 12.

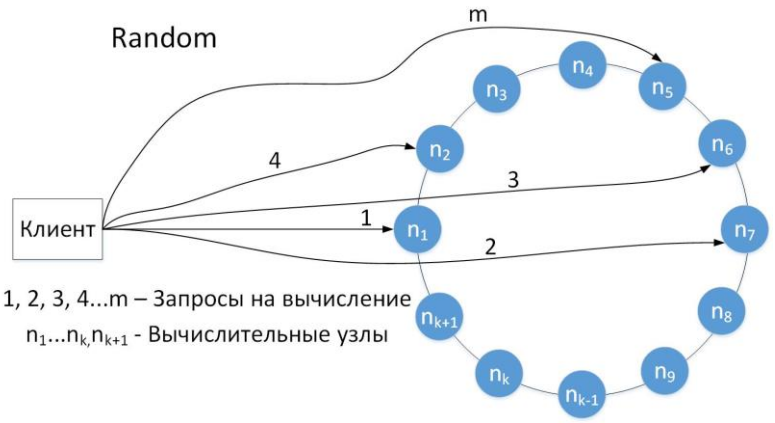

Рисунок 12 – Метод балансировки нагрузки Random

На рисунке выше изображена вычислительная система, состоящая из определенного набора вычислительных узлов от n_1 до n_{k+1}. Также имеется клиент, который отправляет запросы на обработку для данной системы. Таким образом, используя метод Random на основе датчика псевдослучайных чисел, клиент отправит первый запрос вычислительному узлу n_1, второй – узлу n_7 и так далее.

Алгоритм является несовершенным ввиду того, что не учитывает характеристики и особенности каждого вычислительного узла и линий связи между клиентами и узлами, а также не позволяет равномерно распределить нагрузку между вычислителями (имеется вероятность, что на один вычислительный узел подряд будет отправлено два или более запроса).

2) Weighted Random (Взвешенный Random) является улучшенной версией метода Random [55]. Улучшение состоит в добавлении веса каждому вычислительному узлу системы на основе их вычислительных возможностей. Более мощному вычислительному узлу, который может обработать запросы быстрее присваивается большой вес, менее мощному – меньший. Датчик псевдослучайных чисел выбирает любой сервер для последующей обработки запроса из предоставляемого списка. В список добавляются все вычислительные узлы системы, но, в отличие от алгоритма невзвешенного метода Random, в данном алгоритме вычислительные узлы будут добавляться в список на основе их весов.

Пример применения алгоритма Weighted Random представлен на рисунке 13.

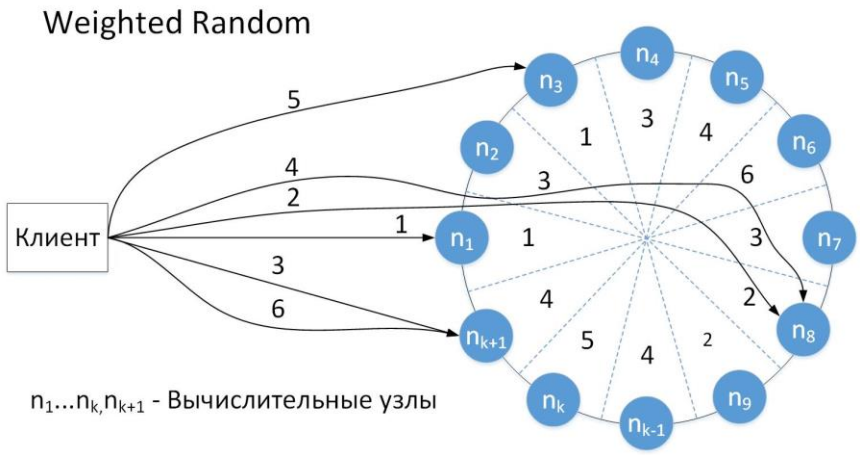

Рисунок 13 – Метод балансировки нагрузки Weighted Random

В список добавляются узлы относительно их веса: n_1, n_2, n_2, n_2, n_3, n_4, n_4, n_4 и т.д. Ввиду того, что в список добавляются узлы с учетом их вычислительной мощности, данный алгоритм позволяет лучше сбалансировать нагрузку относительно простого алгоритма Random. Однако, ввиду несовершенства подхода к выбору вычислительного узла (на основе датчика псевдослучайных чисел), остается возможность многократного последовательного выбора одного вычислителя для отправки запросов, число которых может превысить его возможности для их обработки, что может создать очередь и снизить скорость вычисления всей системой.

3) Round Robin – метод балансировки нагрузки, который основан на алгоритме кругового обслуживания, который позволяет отправлять запросы «по кругу», т.е. циклически [55]. Имеется определенный список всех вычислителей на основе которого происходит перебор для отправки запроса.

После начала обработки очередного запроса система запоминает, какой вычислительный узел в данный момент производит вычисления. При этом, следующий запрос на обработку будет отправлен на следующий вычислитель в списке.

Метод Round Robin представлен на рисунке 14.

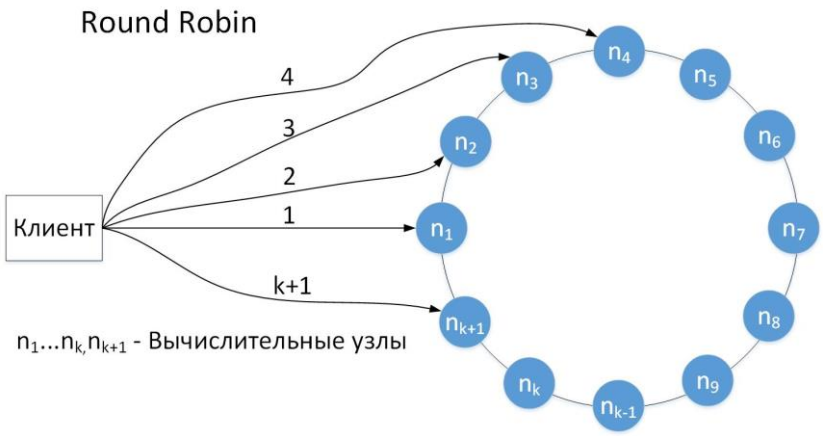

Рисунок 14 – Метод балансировки нагрузки Round Robin

Исходя из рисунка выше, можно убедиться, что первый запрос будет отправлен на вычислительный узел n_1, второй - на узел n_2, третий – на узел n_3 и т.д. исходя из списка вычислителей.

После того, как последний вычислительный узел в списке начал обработку, последующий поступивший запрос подаётся снова на начало списка, т.е. распределение вычислительных узлов относительно отправляемых запросов начинается сначала.

Данный алгоритм является более эффективным, чем алгоритм Random ввиду того, что он позволяет повысить равномерность распределения нагрузки между вычислителями, учитывая общее их количество. Однако он не учитывает характеристики системы, принимая то, что узлы обладают одинаковой производительностью.

Примером применения алгоритма является Round Robin DNS, который использует данный метод балансировки нагрузки для управления ответами DNS-серверов [56]. Round Robin DNS используется в серверах компании Google. Каждый DNS-сервер предоставляет идентичный сервис. Однако при отправлении запроса на сервера Google выдается список из 11-ти DNS-серверов (рисунок 15).

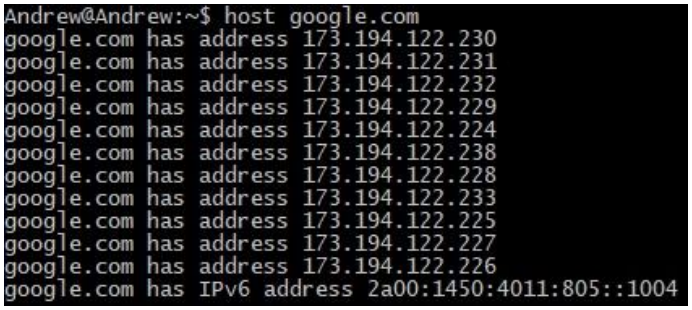

Рисунок 15 – Отправка запроса на google.com

Принцип работы основан на том, что с каждым запросом меняется DNS-сервер, который производит ответ (который стоит на вершине списка адресов), что показано на рисунке 16.

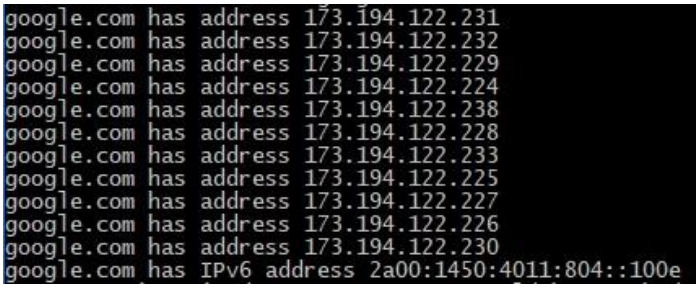

Рисунок 16 – Отправка повторного запроса на google.com

У метода Round Robin DNS имеются определенные недостатки – отсутствует учет доступности услуг, т.е. если сервис на одном вычислительном узле не является доступным, то алгоритм продолжит раздавать адрес этого узла, и клиенты будут продолжать пытаться связаться с неработающим вычислительным узлом. Для решения этой задачи имеются модифицированные DNS-сервера, которые могут регулярно опрашивать сервера для проверки их доступности и степени нагрузки.

4) Weighted Round Robin (Взвешенный Round Robin). Метод является доработанной версией Round Robin, которая позволяет учитывать вычислительную мощность каждого узла в системе.

Таким образом, на узлы с большей производительностью будут отправляться запросы чаще, чем на узлы с меньшей. Это позволяет повысить равномерность распределения и эффективность обработки поступающих запросов [55].

Метод Weighted Round Robin представлен на рисунке 17.

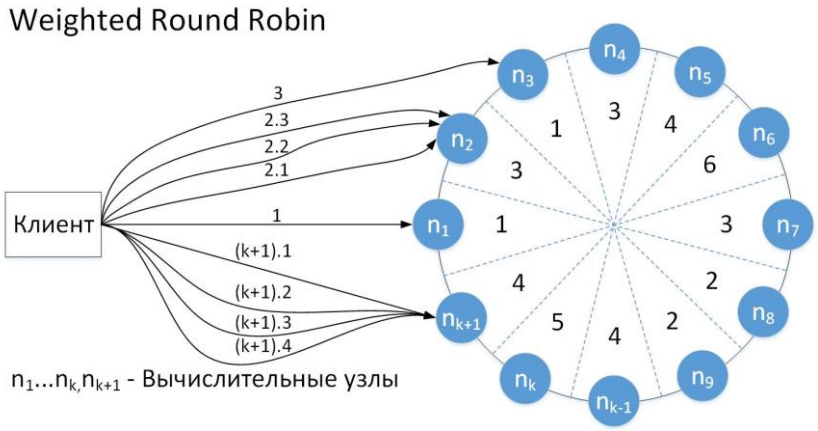

Рисунок 17 – Метод балансировки нагрузки Weighted Round Robin

На рисунке выше приведен пример работы данного алгоритма. Вычислительный узел n_1 имеет вес 1, а значит, допустим, в данном цикле на него может поступить только один запрос на обработку, узел n_2 имеет вес 3 – в цикле на него может поступить три запроса и т.д.

При проектировании любого взвешенного метода балансировки нагрузки можно использовать различные весовые системы – целые числа; десятичные числа, которые должно в сумме составлять 1.0 (100%) и т.д. [55].

Очередь предоставления серверов на обработку может выглядеть следующим образом: n_1, n_2, n_2, n_2, n_3, n_4, n_4, n_4 и т.д.

Если на узел n_1 отправлять за цикл два запроса, то он не успеет обработать их и отправить данные. Второй запрос будет ожидать выполнения и выполнится уже в другом цикле, когда поступят новые два запроса. Постепенно с каждым новым циклом, очередь на обработку запросов в

вычислительном узел n_1 будет расти, что может привести к потере запросов и снижении эффективности обработки всей системой.

Снижение эффективности также произойдет и при поступлении на вычислительный узел меньшего количества запросов за цикл, чем данный узел может обработать. Например, на узел n_6 будет поступать не 6 запросов за цикл, а 3.

Недостатком метода, является то, что веса для вычислительных узлов устанавливаются вручную и при изменениях в системе (улучшении характеристики определенного узла путём добавления новых аппаратных решений или резком неожиданном снижении вычислительной мощности другого узла из-за возникших аппаратных или программных неисправностей), количество отправляемых запросов на обработку останется неизменным (рисунок 18).

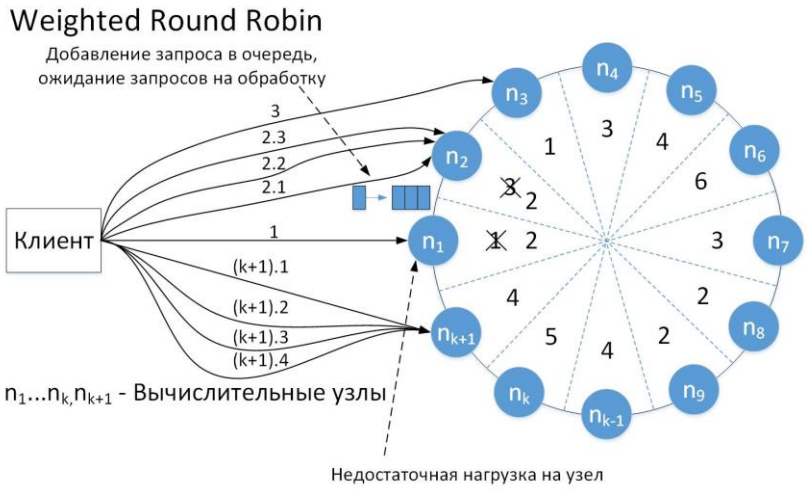

Рисунок 18 – Снижение производительности вычислительной системы при выполнении метода Weighted Round Robin

Снижение производительности одного узла может привести к существенному снижению производительности и скорости обработки запросов всей системы, а также потерю поступающих запросов, которые будут накоплены в очереди неисправного вычислителя.

5) Agent-Based Adaptive – метод балансировки нагрузки, который позволяет в реальном времени следить за нагрузкой каждого вычислительного узла. Для этого каждый вычислительный узел имеет специальный агент, который сообщает о текущей нагрузке диспетчер балансировки нагрузки [57].

6) Geo-locating load balancing [58] – метод балансировки нагрузки, который, позволяет обрабатывать запросы тем вычислительным узлам в распределенной вычислительной системе, которые находятся ближе всех к клиенту, который отправил запрос на обработку. Пример данного метода балансировки представлен на рисунке 19. На карту нанесены условные вычислительные серверы в разных частях света.

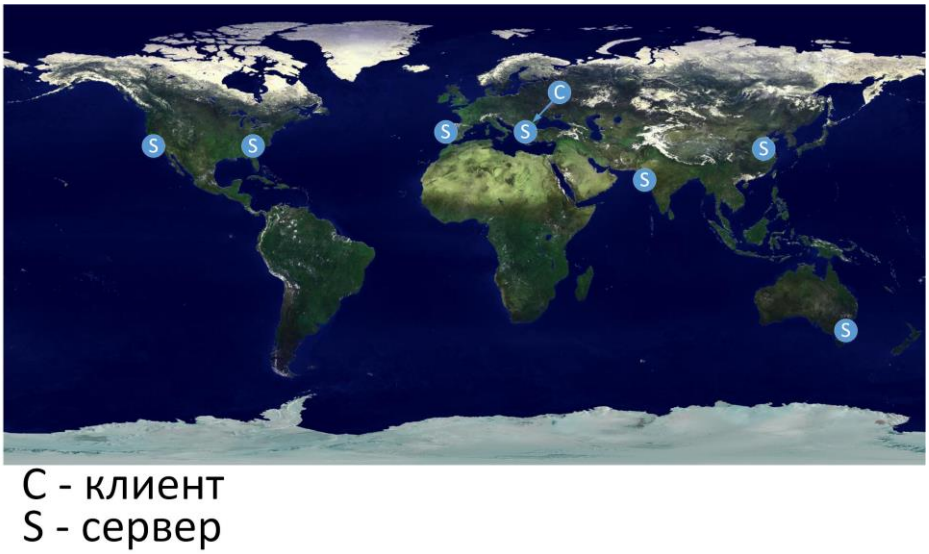

Рисунок 19 – Метод балансировки нагрузки Geo-locating load balancing

Исходя из данных рисунка 19, можно увидеть, что клиентский запрос обрабатывает ближайший к нему сервер.

Метод эффективно работает в масштабе континентов и стран, однако имеет определенные сложности в определение местоположения клиента, отправившего запрос в пределах страны. Это обусловлено отсутствием точной геобазы DNS-адресов, а также тем, что география городов и местностей зачастую не коррелирует с тем, каким образом происходит связь

между соседними местностями, т.е. соседние населенные пункты могут быть связаны через крупный центр, который находится в тысячах километрах от них.

7) Dynamic Weighted Round Robin является улучшением метода Weighted Round Robin (дополнительно используется метод Agent-Based Adaptive). Благодаря методу Agent-Based Adaptive вес динамически присваивается каждому серверу на основе данных в реальном времени о его текущей нагрузке и пропускной способности [59].

Пример использования алгоритма представлен на рисунке 20.

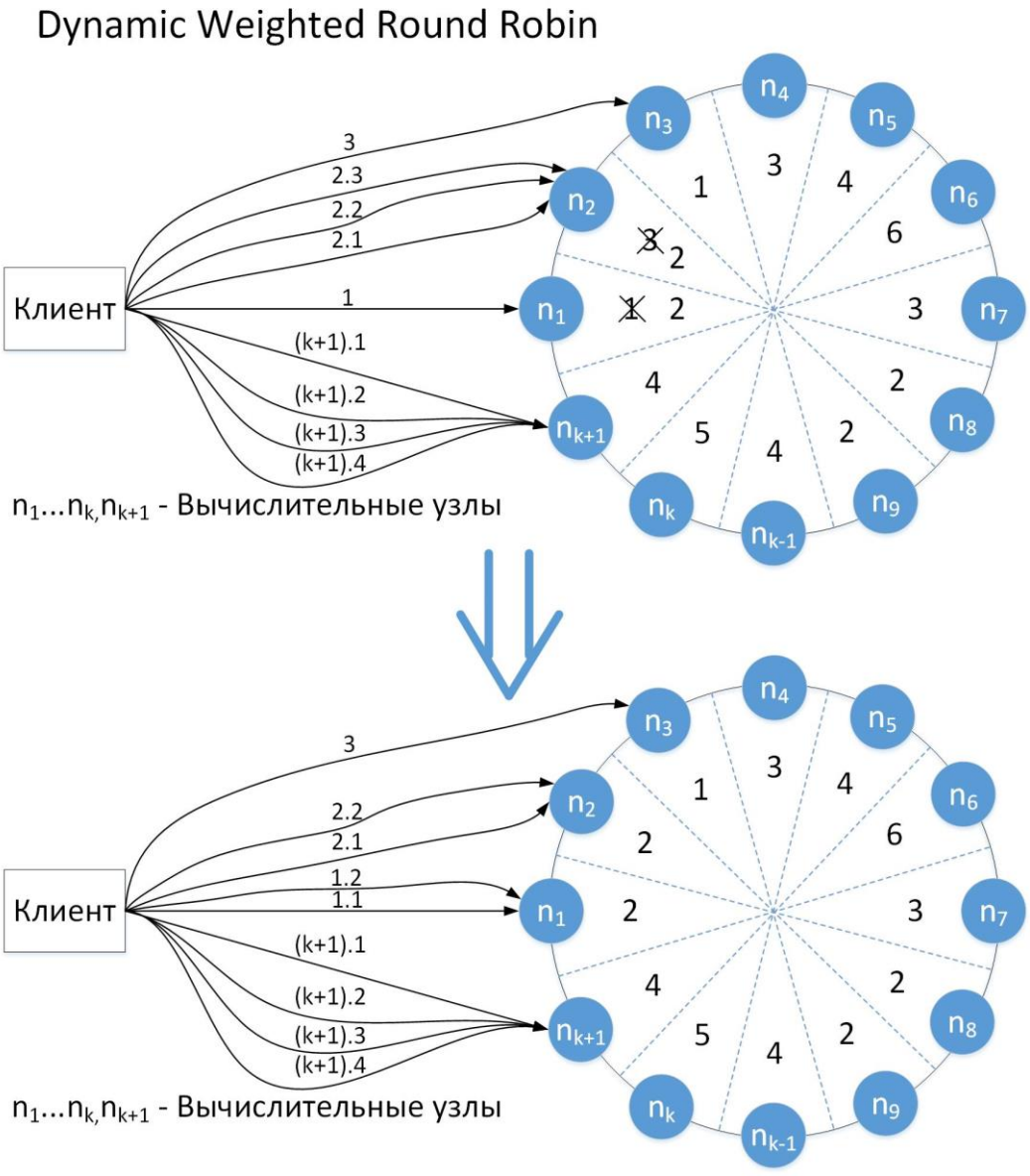

Рисунок 20 – Метод балансировки нагрузки Dynamic Weighted Round Robin

На основании данных рисунка 20 можно увидеть, что при снижении вычислительной мощности вычислительного узла n_2 диспетчер балансировки нагрузки отправляет меньше запросов на данный узел. Обратная ситуация с узлом n_1, вычислительная мощность которого увеличилась, и, следовательно, диспетчер балансировки нагрузки подает на него больше запросов на обработку.

8) Data Center Aware – метод балансировки нагрузки, который предполагает разделение вычислительных узлов на два или более кластера. Диспетчер балансировки нагрузки определяет по различным характеристикам системы (количество обрабатываемых запросов в текущий момент, загруженность кластера, количество вычислительных узлов в кластере, время обработки запросов и т.д. в зависимости от метода балансировки нагрузки) тот вычислительный кластер, куда следует отправить запрос на обработку. Каждый вычислительный кластер может обрабатывать запросы в соответствии со своим методом балансировки нагрузки.

На основе данного метода имеется возможность распределить вычислительные узлы между кластерами таким образом, чтобы, например, один вычислительный кластер обрабатывал только быстрые и легкие запросы, а другой – тяжелые, требующие высокую производительность от вычислителей. Метод позволяет распределить обязанности между вычислительными кластерами на основе различных вычислительных задач и различных функций определенного проекта.

Пример использования метода балансировки Data Center Aware представлен на рисунке 21.

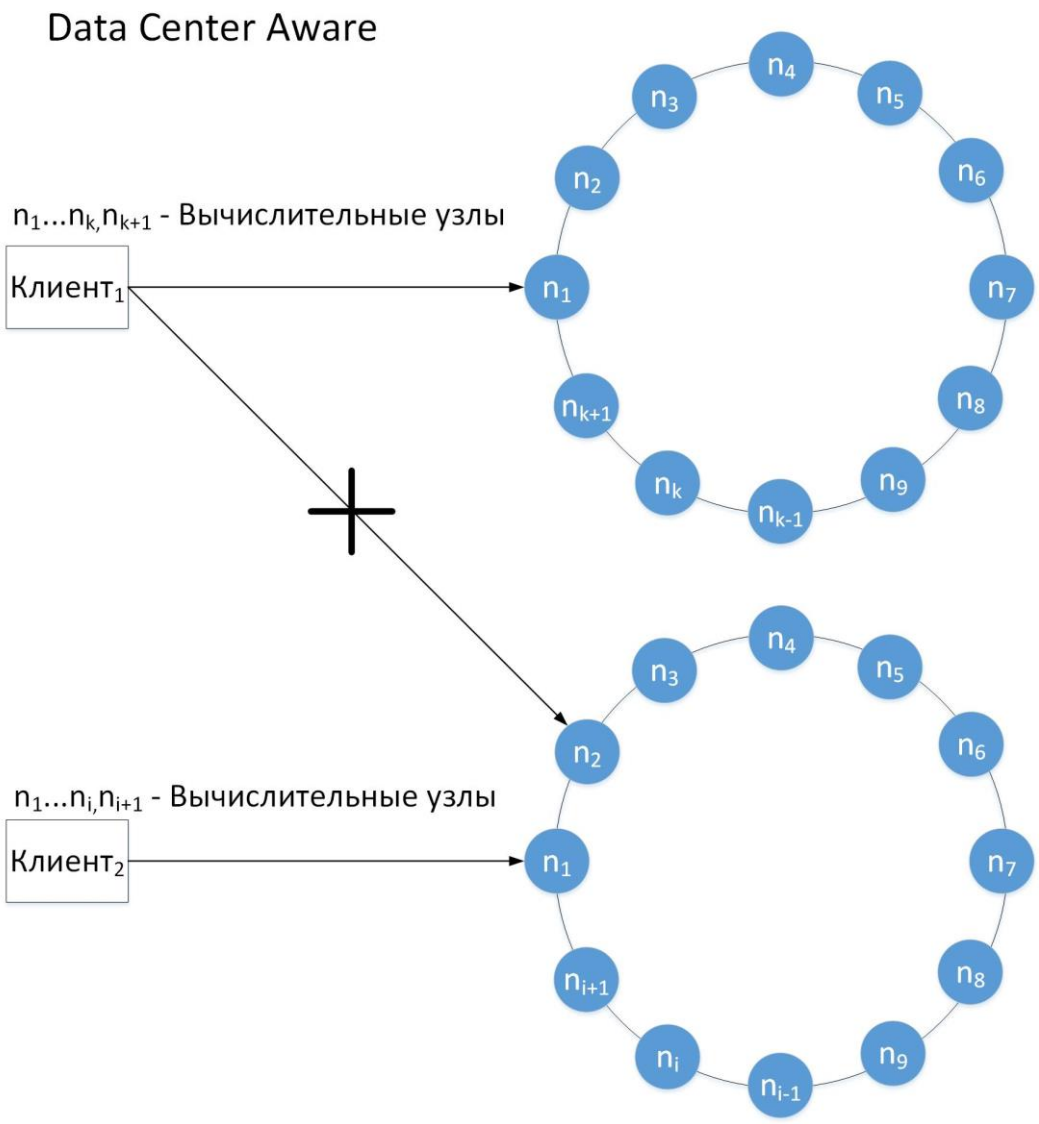

Рисунок 21 – Метод балансировки нагрузки Data Center Aware

На основе показаний рисунка 21 можно увидеть, что клиент$_1$ отправляет запросы на первый вычислительный кластер, а клиент$_2$ – на второй. Причиной может быть то, что первый кластер предназначен для только сложных типов запросов, а второй – только для легких. Алгоритм балансировки нагрузки в каждом вычислительном кластере может отличаться. Однако, при отказе первого вычислительного кластера (рисунок 22) запросы от клиента$_1$ будут перенаправлены на второй вычислительный кластер, что значительно снизит скорость обработки запросов системой, однако повышенная отказоустойчивость позволит системе остаться в работоспособном состоянии [60, 61].

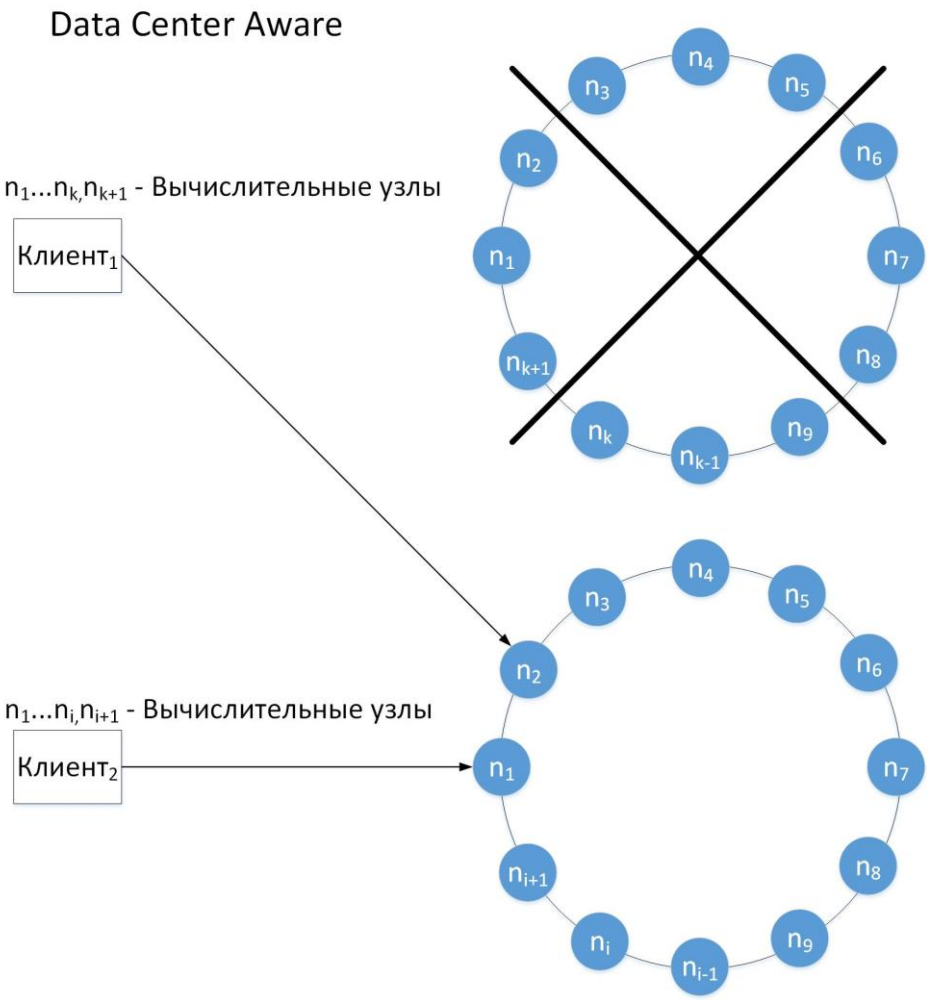

Рисунок 22 – Отказ вычислительного узла в метод балансировки нагрузки Data Center Aware

Метод эффективно применятся в высоконагруженных веб-проектах, где полный отказ вычислительной системы, даже на небольшой отрезок времени, может серьезно удалить как по имиджу компании, так и по пользователям, которые используют сервис.

9) Token Aware – метод балансировки нагрузки, позволяющий сравнить token (маркер), который извлечен из запроса, с маркером, который находится в вычислительном узле, что позволяет отправлять запросы на обработку именно на тот узел, которому они принадлежат. Метод действует как оболочка или фильтр, обертывающий другой любой алгоритм балансировки нагрузки, и позволяет сократить расстояние, которое проходят пакеты до нужного вычислительного узла [62].

Алгоритм используется решением от компании Citrix – NetScaller, которое позволяет ускорить работу приложений, а также оптимизировать сети по доставке данных. NetScaller позволяет проводить поиск маркера в первых 24-х килобайтах передачи данных по протоколу HTTP. Для передачи данных по протоколам, отличным от HTTP, производится поиска маркера в первых 16-ти пакетах или пока размер 16-ти пакетов не превысит 24 килобайта. После извлечения маркера производится поиск соответствующего вычислительного узла, который выполнит обработку запроса [63, 64].

Пример использования метода Token Aware представлен на рисунке 23.

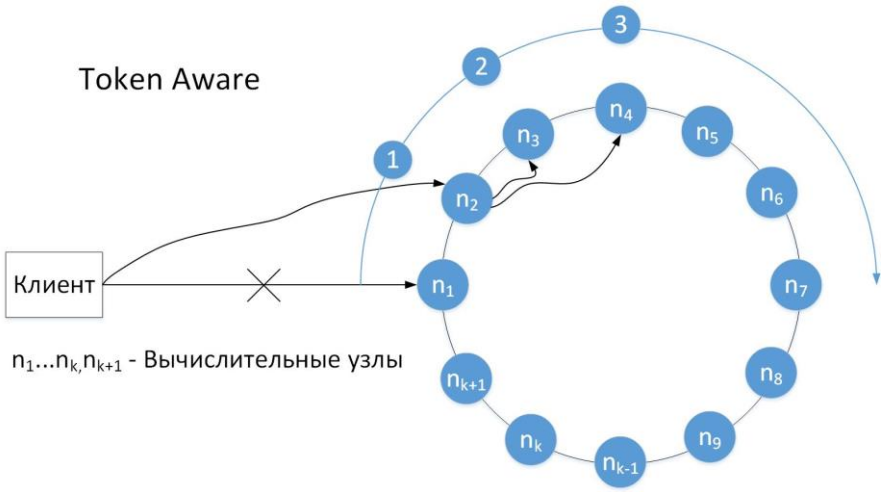

Рисунок 23 – Метод балансировки нагрузки Token Aware

Исходя из данных рисунка 23, можно увидеть, что запрос отправляется на вычислительный кластер и диспетчер балансировки нагрузки производит перенаправление запроса из вычислительного узла n_1 в вычислительный узел n_2. Далее, после обработки на вычислительном узле n_2 запрос переправляется на вычислительные узлы n_3 и n_4.

Подбор вычислительного узла для запроса по маркеру является эффективным способом повысить скорость обработки данных и сбалансировать нагрузку на вычислители.

10) Least Connections – метод балансировки нагрузки, при котором диспетчер балансировки анализирует каждый вычислительный узел на

предмет количества соединений [55]. Чем меньше соединений у вычислительного узла, тем меньше он загружен.

Применение метода Least Connections представлено на рисунке 24.

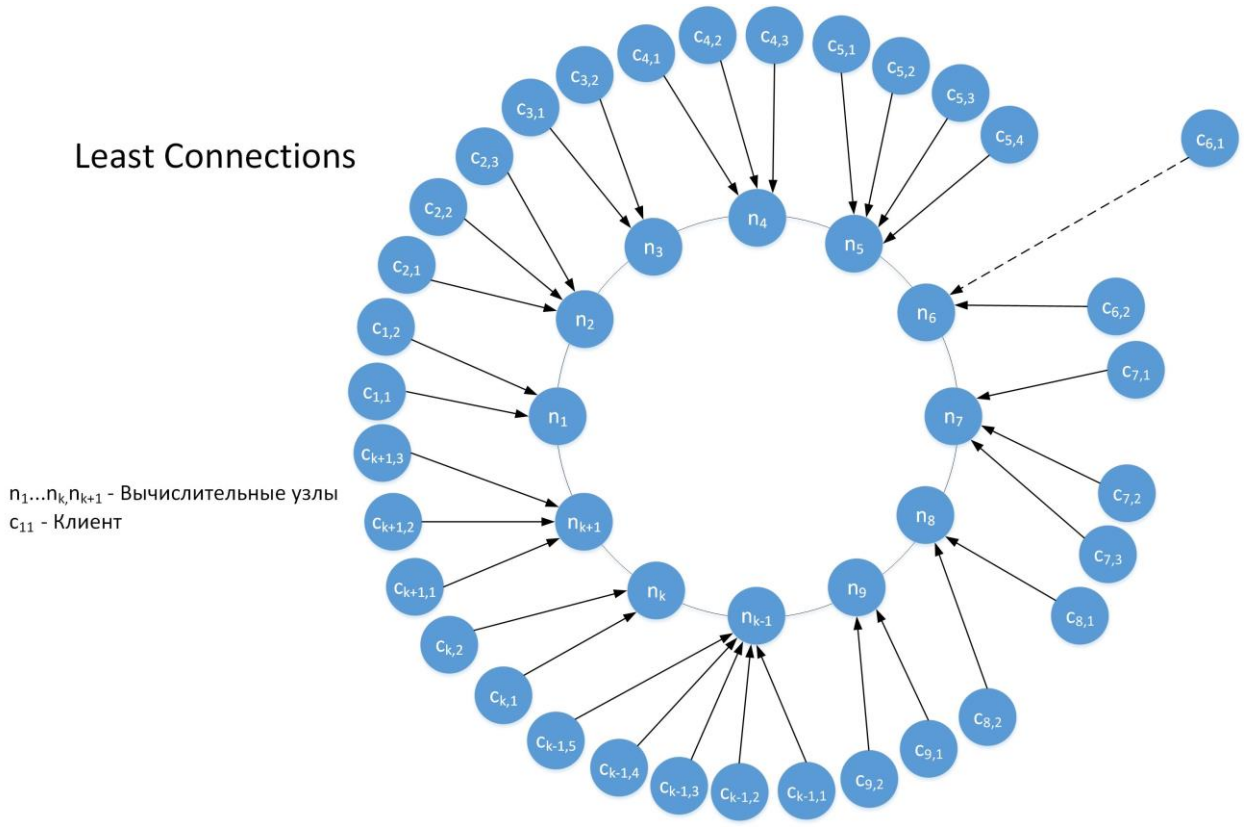

Рисунок 24 – Метод балансировки нагрузки Least Connections

Недостатком метода является то, что количество соединений не учитывает возможности вычислительного узла. Узел в системе может обладать низкой производительностью и, соответственно, малым количеством соединений. Обладая малым числом соединений, он может быть полностью нагруженным и не в состоянии принять новые запросы на обработку.

Например, вычислительный узел n_6 может обладать низкой производительностью или обрабатывать сложный запрос и в настоящий момент не в состоянии принимать больше запросов на обработку. Но вычислительный узел n_3, обрабатывая два клиентских запроса и обладая высокой производительностью, может принять гораздо больше запросов, и

было бы предпочтительней отправить новый запрос на обработку на него, а не на узел n_6.

Метод в первую очередь предназначен для однородных вычислительных сред.

11) Weighted Least Connections – алгоритм, работа которого схожа работе метода Least Connections, однако, алгоритм также позволяет учитывать вычислительные характеристики узлов системы, присваивая каждому узлу определенный вес, что аналогично методу Weighted Round Robin [65].

Работа алгоритма представлена на рисунке 25.

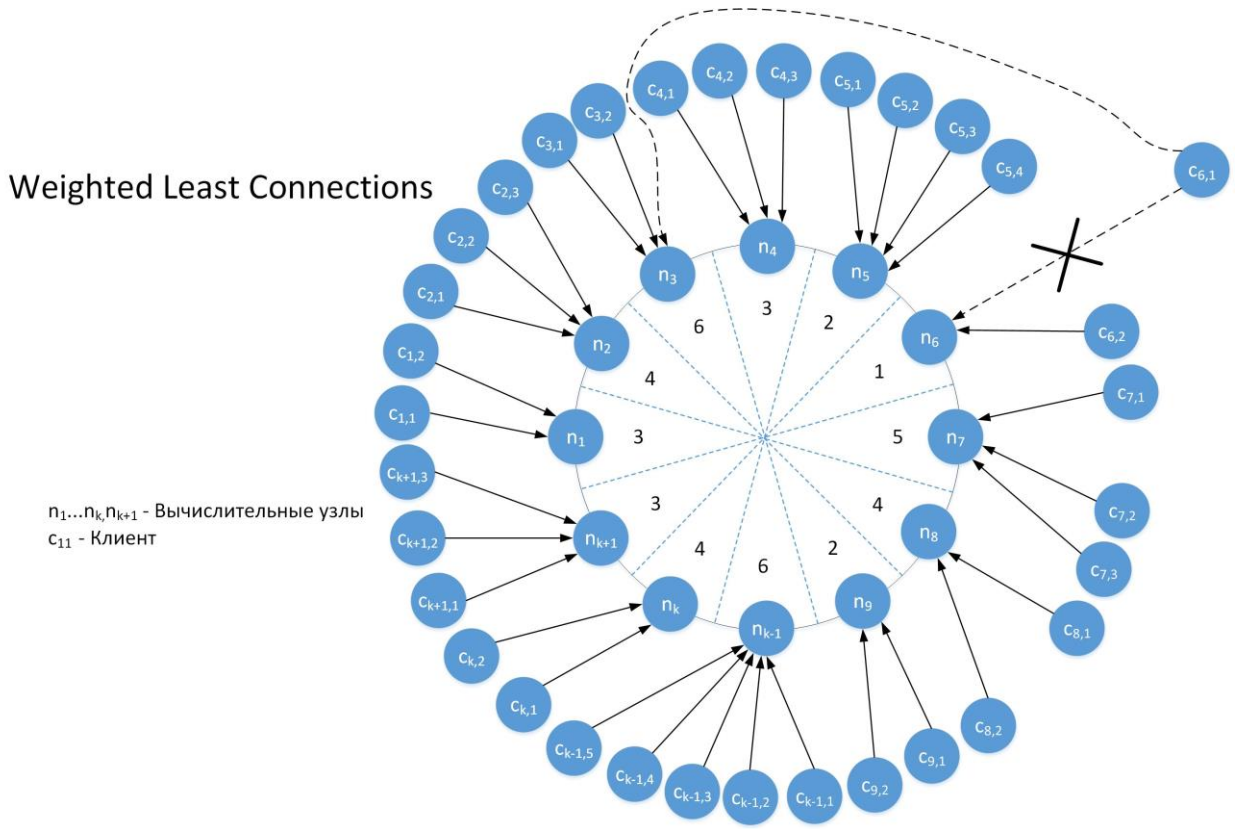

Рисунок 25 – Метод балансировки нагрузки Weighted Least Connections

В отличие от метода Least Connections, в котором диспетчер балансировки нагрузки передаст запрос на обработку вычислительному узлу n_6, обладающему низкой вычислительной мощностью, метод Weighted Least Connections передаст запрос также на основании данных о вычислительной

мощности узла. Наиболее предпочтительным вариантом является вычислитель n_3, который, обладая высокой производительностью, принимает на обработку всего 2 запроса.

Недостатком данного метода является то, что он не позволяет определить время, которое поддерживается связь, что позволило бы определить загруженность вычислительного узла. Вычислитель может обладать высокой производительностью, но в данный момент к нему подключено большое количество легких запросов и он не будет принимать другие, хотя для обработки легких запросов он может не использовать все свои возможности.

И наоборот, высокопроизводительный узел принимает всего несколько тяжелых запросов, но этого достаточно, чтобы полностью загрузить его вычислительные мощности. Алгоритм будет передавать новые запросы на обработку именно на этот, полностью загруженный, вычислитель.

12) Dynamic Weighted Least Connection – метод балансировки нагрузки совмещает в себе метод Weighted Least Connection и Agent-Based Adaptive. Принцип работы схож с методом Dynamic Weighted Round Robin – производительность каждого вычислительного узла, с помощью специального агента, в реальном времени анализируется и отправляется в диспетчер балансировки нагрузки, который на основе последних данных производит распределение запросов.

Метод значительно повышает эффективность и гибкость Weighted Least Connection ввиду того, что алгоритм подстраивается под любые изменения характеристик вычислителей.

13) Locality-Based Least Connection Scheduling – алгоритм балансировки нагрузки, основанный на методе Least Connection и предназначенный для использования в кэширующих прокси-серверах. За каждым клиентским сервером закрепляется определенное количество клиентских IP-адресов. Запросы отправляются на этот закрепленный за клиентами сервер, если он не

загружен полностью. Если он загружен, то выбирается любой другой сервер, который загружен менее чем наполовину [41, 66].

На рисунке 26 можно увидеть работу алгоритма, при котором новое подключение выполняется не к полностью загруженному вычислительному узлу, даже если клиент закреплен за ним, а выбирается другой вычислительный узел, который загружен менее чем на 50%.

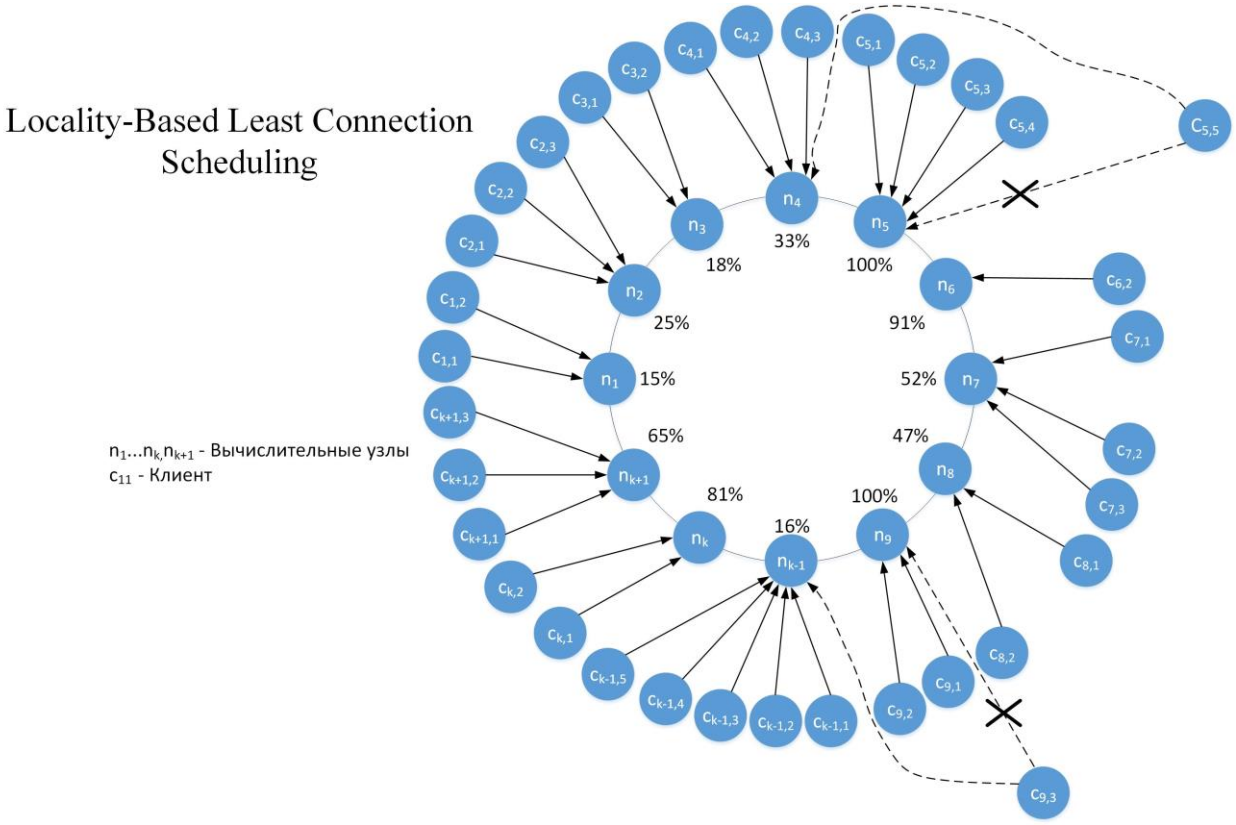

Рисунок 26 – Метод балансировки нагрузки Locality-Based Least Connection Scheduling

Алгоритм решает проблему, описанную в методе Weighted Least Connections. Теперь диспетчер нагрузки не будет отправлять новые запросы на вычислительный узел с малым количеством тяжелых запросов, который полностью загружен.

14) Locality-Based Least Connection Scheduling with Replication Scheduling – метод балансировки нагрузки, который основан на том, что каждый IP-адрес или группа адресов закрепляются за одним или группой

вычислительных узлов. Запросы передаются наименее загруженному вычислителю. При возникновении ситуации перегрузки всех вычислительных узлов в группе происходит резервирование нового вычислителя за пределами группы. Вычислитель добавляется в группу перегруженных вычислителей. Одновременно с добавлением вычислителя проводится проверка на загруженность узлов и удаление наиболее загруженного вычислителя, что позволяет избежать избыточной репликации [41].

Пример использования алгоритма представлен на рисунке 27.

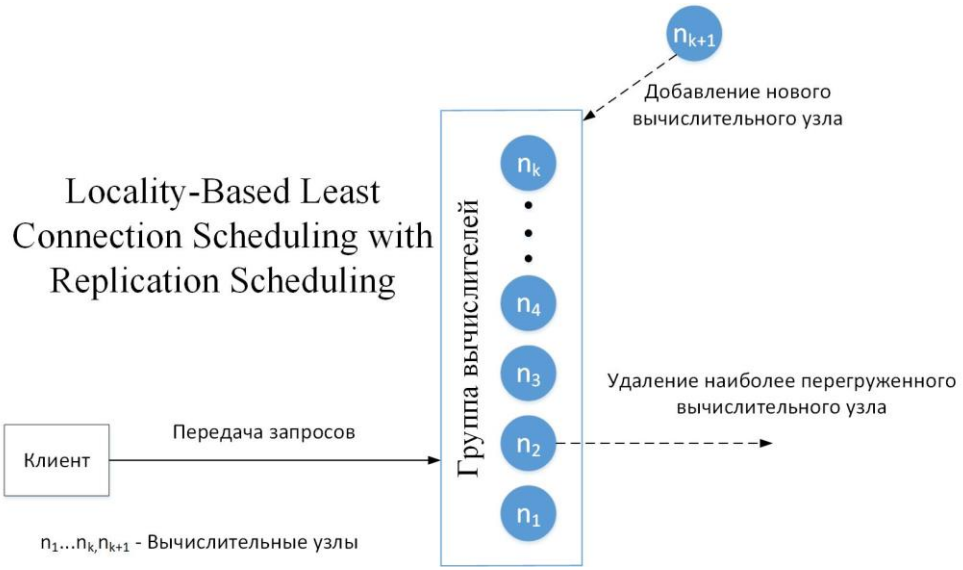

Рисунок 27 – Метод балансировки нагрузки Locality-Based Least Connection Scheduling with Replication Scheduling

15) Least Response Time – метод балансировки нагрузки, который оценивает время ответа на запрос [67]. Соответственно, чем меньше время ответа, тем производительнее вычислительный узел, т.к. обрабатывает он запросы быстрее или тем меньше соединений в данный момент у вычислительного узла и он менее загружен в настоящий момент.

Диспетчер балансировки нагрузки выбирает вычислительный узел с наименьшим временем ответа и ставит наивысший приоритет отправления

запросов данному узлу. Запросы поступают на данный узел до тех пор, пока время ответа не увеличится, т.е. до тех пор, пока узел не загрузится.

Метод Least Response Time представлен на рисунке 28.

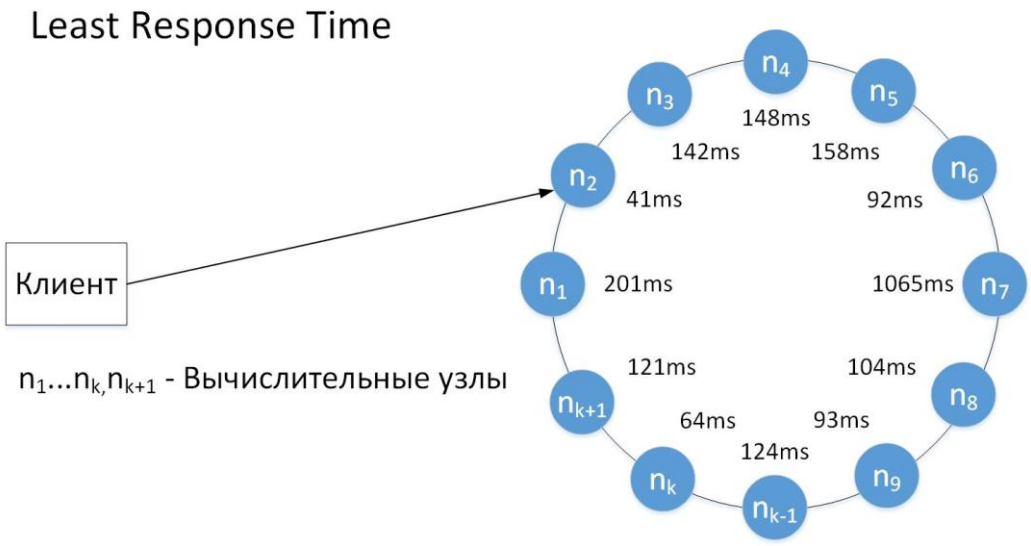

Рисунок 28 – Метод балансировки нагрузки Least Response Time

Недостатком метода является то, что диспетчер балансировки не оценивает сложность поступившего запроса. Таким образом, вычислительный узел, обладая невысокой вычислительной мощностью, с наименьшим временем обработал поступивший самый легкий запрос на обработку. Диспетчер балансировки нагрузки на основе данного метода будет считать текущий вычислитель самым высокопроизводительным и отправит на него один или несколько сложных запросов, вычисление которых узлом займет гораздо больше времени, чем, если бы эти запросы обработал наимощнейший вычислительный узел в системе. Однако самый мощный вычислительный узел в системе в данный момент обрабатывает сложные запросы и время ответа на запрос у него составляет далеко не самое лучшее время. Недостатком также будет то, что при отправке одинаковых по сложности запросов, высокопроизводительные вычислительные узлы будут принимать все запросы на себя, оставляя простаивать менее мощные вычислители.

Ввиду явного недостатка метода Least Response Time его, как и метод Least Connections, эффективно применяют для однородных вычислительных сред.

16) Load-Based – метод балансировки нагрузки, при котором диспетчер нагрузки снимает с определенные метрики и на основе их анализирует загруженность вычислительного узла.

Пример метода Load-based представлен на рисунке 29.

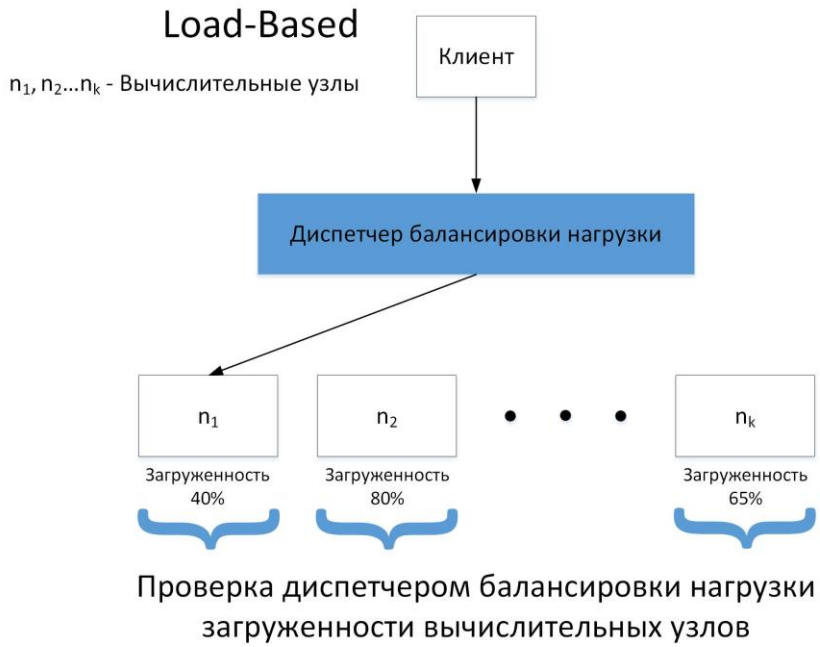

Рисунок 29 - Метод балансировки нагрузки Load-Based

Недостатком является несвоевременное получение метрик диспетчером балансировки нагрузки, которое связано как с каналами связи, так и с тем, что время, обработку запроса может быть гораздо меньше, чем период сбора метрик с вычислительных узлов. Т.е. диспетчер балансировки нагрузки будет отправлять всё новые и новые запросы на обработку на вычислительный узел, считая, что он практически не нагружен на основе устаревших метрических данных, когда как вычислительный узел давно загружен полностью и запросы копятся в очередь, что снижает скорость обработки данных для всей системы. Другие же узлы, которые были сильно или

полностью загружены на момент снятия метрических показателей системы, на текущий момент являются свободными и ненагруженными, однако диспетчер балансировки нагрузки продолжает не посылать на них запросы на обработку, т.к. на основе его устаревших метрик данные вычислительные узлы являются нагруженными.

Решение кроится в повышении частоты сбора метрических показателей системы, однако данный способ не всегда является эффективным ввиду того, что постоянным собором метрики засоряется канал связи, а также это требует дополнительного процессорного времени каждого узла, что может снизить скорость обработки запросов во всей системе и лишь усугубить проблему эффективной балансировки нагрузки.

17) Fastest – метод балансировки нагрузки, который основан на минимальном времени отклика вычислительного узла [68]. Клиент отправляет запрос на обработку и диспетчер балансировки нагрузки смотрит время отклика каждого подключенного вычислительного узла. Чем меньше время отклика, тем меньше загружен вычислительный узел.

Метод представлен на рисунке 30.

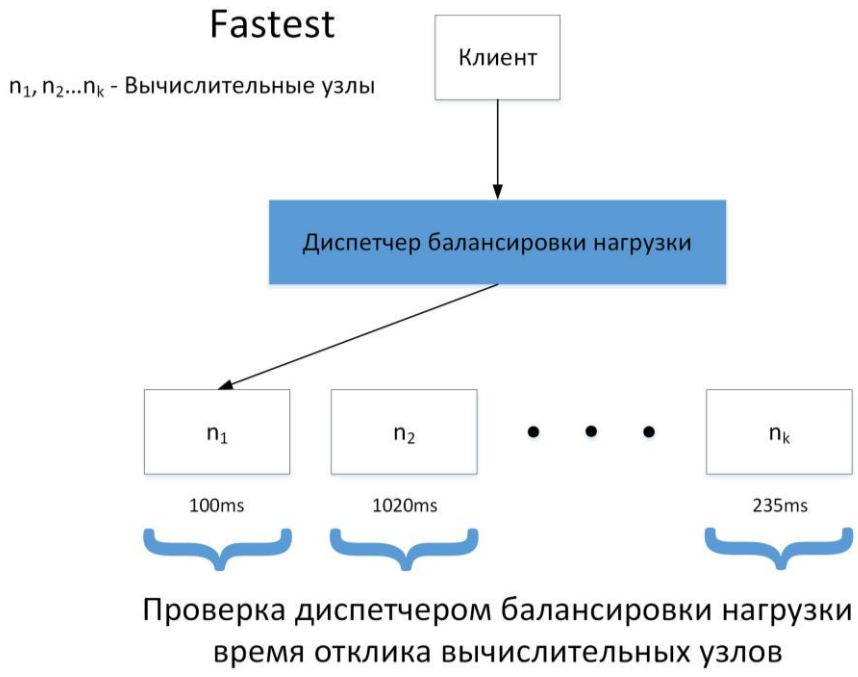

Рисунок 30 - Метод балансировки нагрузки Load-Based

18) Perceptive – метод балансировки нагрузки, который объединяет в себе методы Least Response Time и Least Connections.

Метод эффективен в гетерогенных вычислительных системах, когда узлы между собой значительно отличаются по производительности.

В методе первым срабатывает Least Connections, который позволяет равномерно (относительно количества поступающих запросов) нагрузить как высоко-, так и малопроизводительные узлы. По мере увеличения трафика, мощные вычислительные узлы будут обрабатывать запросы быстрее малопроизводительных, которые могут «увязнуть» в тяжелых запросах. Далее срабатывает метод Least Response Time, который позволяет увидеть время обработки запроса вычислительным узлом и в первую очередь нагружать те узлы, которые обрабатывают запросы быстрее, постепенно разгружая малопроизводительные вычислители.

Таким образом, метод позволяет ликвидировать недостатки каждого из двух методов: для Least Response Time – нагружать также малопроизводительные вычислительные узлы, благодаря методу Least Connections; для Least Connections – учитывать вычислительные возможности каждого узла на основе времени обработки запросов, благодаря методу Least Response Time [68, 69].

Пример использования метода Perceptive представлен на рисунке 31.

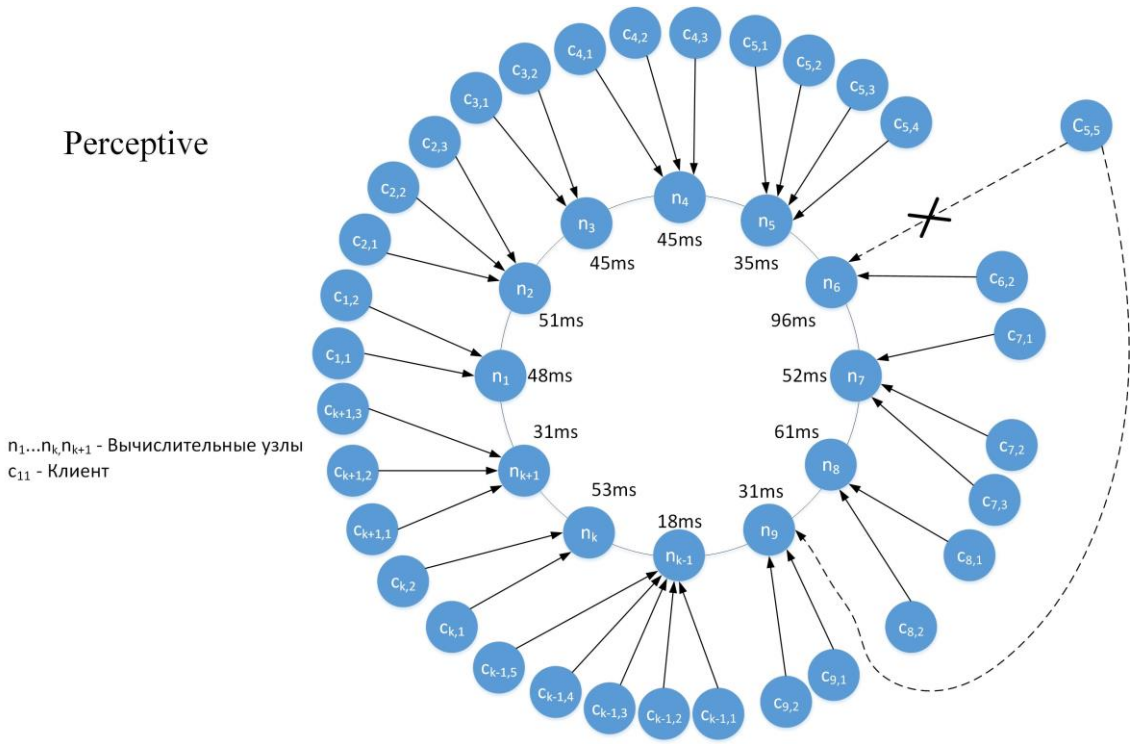

Рисунок 31 - Метод балансировки нагрузки Perceptive

Исходя из данных рисунка 31, можно увидеть, что новый запрос будет отправлен на вычислительный узел в соответствии с требованиями двух методов. Вычислительный узел n_6 является малопроизводительным и обладает большим временем обработки запросов, поэтому новый запрос будет отправлен не в узел с наименьшим количеством связей, а в узел с наименьшим временем обработки запросов относительно числа запросов. Таким является узел n_9. Среднее время обработки запроса на узле n_9 31 мс, а общее количество запросов на вычислителе – 2. Таким образом, если текущие запросы в очереди ($c_{9,1}$, $c_{9,2}$) будут выполнены за среднее время, то новый запрос ($c_{5,5}$) начнет обрабатываться через 62 мс, что является наименьшим временем среди всех запросов на текущий момент.

Если бы n_6 не обрабатывал в настоящий момент ни один запрос, то $c_{5,5}$ отправился бы на обработку на данный узел на основании метода Least Connections.

19) Observed – метод балансировки нагрузки, который основан на комбинации методов Least Connections и Fastest. Таким образом,

вычислительный узел для обработки поступающего запроса выбирается на основании комбинации количества текущих соединений и времени отклика вычислительного узла [68].

Метод схож с методом Perceptive. Совмещение двух алгоритмов позволяет улучшить степень распределения запросов на вычислителях. Постоянное сканирование времени отклика позволяет получить степень загруженности каждого вычислителя не только по характеристике количества поступающих запросов на узел, т.к. сложность запроса может значительно отличаться.

20) Chained Failover – метод балансировки нагрузки, который имеет сильное сходство с методом Weighted Round Robin тем, что происходит формирование списка вычислительных узлов. Однако в отличие от вышеупомянутого метода, запросы отправляются на обработку не в зависимости от веса каждого вычислительного узла, а полностью нагружая каждый вычислительный узел. Если вычислительный узел не может больше принять запросов, то происходит выбор следующего вычислительного узла в списке. Таким образом, для данного алгоритма нет необходимости сбора информации о весе каждого вычислителя [57].

Пример использования алгоритма Chained Failover представлен на рисунке 32.

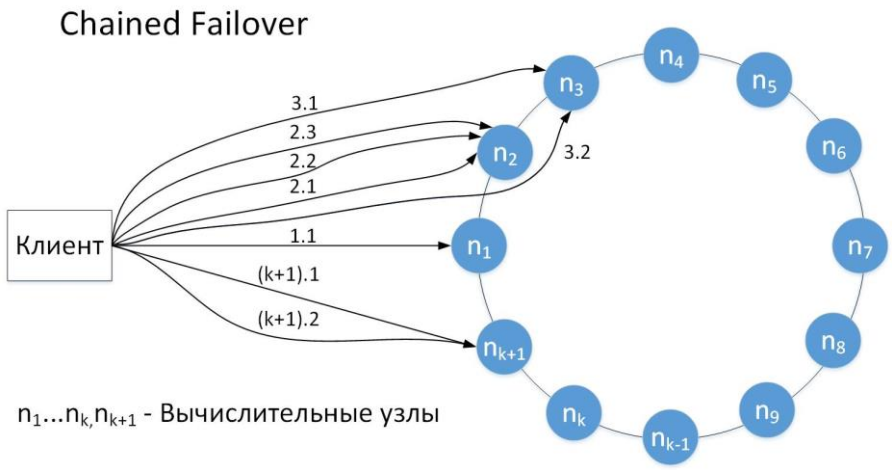

Рисунок 32 – Метод балансировки нагрузки Chained Failover

На основе данных рисунка 32 можно увидеть, что запросы отправляются на каждый вычислительный узел и после полной нагрузки вычислителя производится переход к следующему в списке (n_1, n_2, n_3…n_{k-1} …n_k…n_{k+1}). После полной нагрузки последнего в списке вычислителя происходит новый перебор всех имеющихся вычислителей с полной их загрузкой поступающими запросами на обработку.

21) Source IP Hash – метод балансировки нагрузки, который использует исходный и целевой IP-адреса клиента и сервера и производит их объединение, которое необходимо для генерации уникального ключа хеш-функции. Этот ключ используется для размещения запросов клиента на определенном вычислителе. Поскольку ключ может быть восстановлен после разрыва связи, то этот метод гарантирует то, что запрос клиента будет отправлен именно на тот же сервер, который был использован ранее [57].

Таким образом, каждый клиент подключается к определенному вычислительному узлу, что позволяет распределить нагрузку между вычислителями, однако при определенных обстоятельствах (например, большое количество тяжелых запросов) эффективность алгоритма снижается.

22) Software Defined Networking (SDN) Adaptive – метод балансировки нагрузки, который производит сбор и анализ данных о сети из верхних и нижних слоёв сетевой модели OSI. Это позволяет получать информацию о статусе вычислительных узлов в системе, о состоянии выполняемых запросов на них, состоянии сетевой инфраструктуры и уровне загруженности сети [57].

На основе всех полученных данных происходит принятие решения о распределении запросов между узлами системы.

23) Least Traffic/Weighted Least Traffic – алгоритм балансировки нагрузки, в котором диспетчер контролирует битрейт, исходящий из каждого вычислительного узла системы и отправляет запросы на вычислитель с наименьшим исходящим трафиком.

Недостатком является то, что диспетчеру балансировки нагрузки неизвестна производительность сервера. Таким образом, небольшой трафик может исходить из малопроизводительного узла системы и при его дополнительной нагрузке, запросы будут накапливаться в очереди. Поэтому, для гетерогенной вычислительной системы применяются взвешенный алгоритм.

Взвешенный алгоритм отличается от алгоритма Least Traffic лишь добавлением весов вычислительных узлов, что было описано в таких алгоритмах, как Weighted Round Robin, Weighted Random и Weighted Least Connections.

Применение алгоритма представлено на рисунке 33.

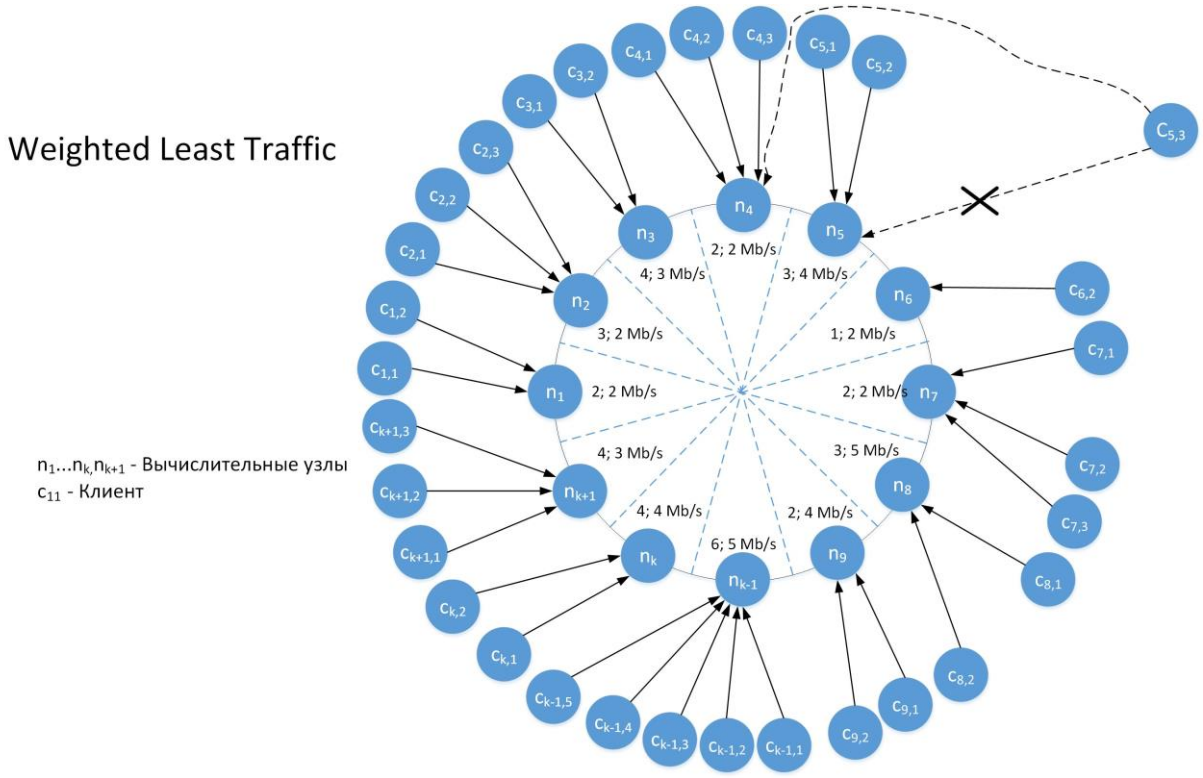

Рисунок 33 – Метод балансировки нагрузки Weighted Least Traffic

Weighted Least Traffic подает запросы на вычислительные узлы с учетом, как веса каждого узла, так и исходящего трафика, который позволяет определить их загруженность. Запрос от клиента $c_{5,3}$ будет подаваться не на

узел n_5, а на узел n_4, хотя вес узла n_5 больше n_4 и количество обрабатываемых запросов от клиентов также меньше. Однако запросы, которые подаются на вычислитель n_5 тяжелее запросов у вычислителя n_4.

Описанные в главе уровни параллелизма вычислений, примеры применения распределения нагрузки, а также способы и механизмы распараллеливания как посредством ЦП, так и посредством ГП, позволяют оценить всю важность и эффективность использования описанных средств для повышения производительности и увеличения скорости выполнения различных вычислительных задач.

Представленный в главе список методов балансировки нагрузки не является исчерпывающим, однако, он позволяет увидеть многообразие различных подходов и способов к распределению нагрузки. Сервисы, механизмы, аппаратные и программные технологии, методы и алгоритмы балансировки нагрузки позволили рассмотреть всю гибкость при построении систем и обработке запросов. Балансировка нагрузки значительно расширяет возможности и повышает эффективность от горизонтального масштабирования вычислителей. В зависимости от системы и характеристик вычислителей, поступающих запросов и требований к системе, всегда имеется возможность эффективно сбалансировать нагрузку, используя описанные методы или совмещая их в различные комбинации.

2 Разработка алгоритма балансировки нагрузки на основе динамического вычисления рангов вычислителей по заданным критериям

2.1 Проектирование вычислительных графов различных размерностей

Для сравнительного анализа алгоритмов балансировки необходимо смоделировать вычислительную задачу. Задача, создаваемая программным комплексом, представлена в виде графа вычислений, где вершины являются модулями для обработки, а связи между вершинами – связями между модулями, которые позволяют передавать данные из выхода одного модуля на вход другого.

Таким образом, каждый модуль графа может выполняться отдельно, однако, для этого ему необходимо подать соответствующие данные на вход, т.е. присутствует зависимость между составными частями графа.

Отличительными особенностями представляемого в настоящей выпускной квалификационной работе графа вычислений является единственный вход, на который подаются данные для выполнения всего алгоритма, единственный выход – откуда результат выдается пользователю, а также отсутствие циклов в графе. Отсутствие циклов в графе необходимо для того, чтобы вычисление графа не зависло на обработке цикловой последовательности модулей (на бесконечной обработке).

В качестве примера приведен вычислительный граф с 10-ю модулями (рисунок 34).

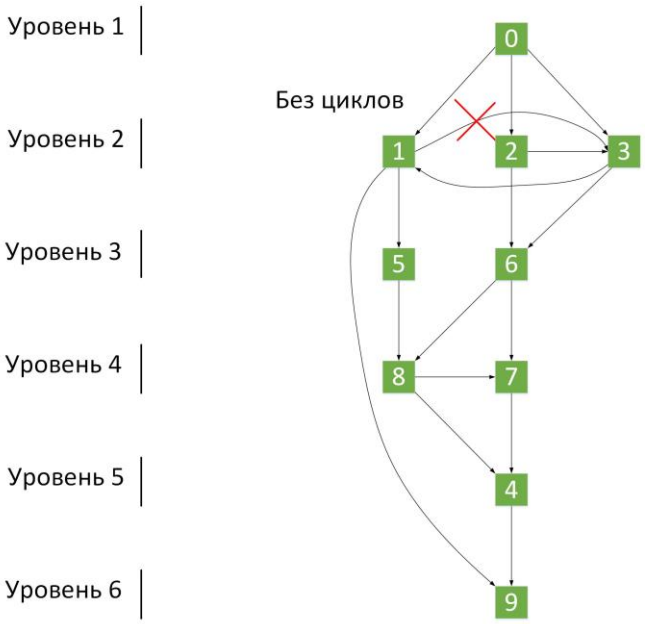

Рисунок 34 – Граф вычислений с 10-ю вершинами

В программном комплексе вычислительный граф создается отдельными уровнями на основе следующих методов: MethodForFirstLayer (создает первый уровень вычислительного графа), MethodForSecondLayer (создает второй уровень), MethodForThirdLayer (третий уровень), MethodForOtherLayer (создает все остальные уровни). Подобное разделение графа необходимо для того, чтобы исследователь имел возможность отдельно задавать количество связей для первого уровня. Такой подход позволяет быстрее создавать графы с малым количеством уровней и модулей, т.к. в методах MethodForSecondLayer и MethodForThirdLayer специально описаны условия создания для графов с числом вершин меньше или равным 5-ти, и поэтому отсутствует проверка на цикличность, что экономит время на создание.

Как можно заметить, граф состоит из шести уровней. Предлагаемый граф на рисунке выше был сгенерирован программным комплексом со следующими параметрами: количество модулей равно 10, количество связей у модуля первого уровня равно 3, количество связей у модулей последующего уровня не больше 3-х. Из рисунка видно, что граф не имеет циклов. Таким образом, связи от модуля 1, до модуля 3 не будет, потому что

это вызовет зацикливание во время обработки графа вычислений: поток из вершины 1 перейдет в вершину 3, а затем снова в вершину 1. Невозможен такой вариант генерации графа, где для модуля 1 необходимы данные из модуля 3, а для модуля 3 необходимы данные из модуля 1.

Задачи для обработки генерируются как произвольно, так и с заданием основных параметров: количества вычислительных модулей, количества связей между нулевым (начальным модулем) и дочерними элементами, максимальное количество связей между модулями ниже уровня 2.

Каждый генерируемый граф вычислений хранится в отдельном файле в виде модифицированной матрицы инцидентности, где 1 означает, что имеется связь между вычислителями, 0 – связи нет. Матрица для описываемого графа вычислений с 10-ю модулями представлена в таблице 2.

Таблица 2 – Граф вычислений в виде матрицы

	Выходные связи										
Входные связи		0	1	2	3	4	5	6	7	8	9
	0	0	1	1	1	0	0	0	0	0	0
	1	0	0	0	0	0	1	0	0	0	1
	2	0	0	0	1	0	0	1	0	0	0
	3	0	1	0	0	0	0	1	0	0	0
	4	0	0	0	0	0	0	0	0	0	1
	5	0	0	0	0	0	0	0	0	1	0
	6	0	0	0	0	0	0	0	1	1	0
	7	0	0	0	0	1	0	0	0	0	0
	8	0	0	0	0	1	0	0	1	0	0
	9	0	0	0	0	0	0	0	0	0	0

В матрице можно увидеть 10 строк и 10 столбцов. Каждая строка описывает выходные связи модуля, каждый столбец – входные связи.

Например, модуль 0 имеет только 3 выходные связи – к модулям 1, 2 и 3, что показано в строке 0, однако, модуль 0 не имеет входных связей, поэтому все элементы в столбце 0 равны нулю.

Также имеется возможность хранить граф вычислений в виде двумерного списка, где номер строки равен номеру модуля, а числа – модули, в которые передается поток вычислений после обработки текущего элемента. Например, для описанного выше графа вычислений с 10-ю вершинами: 0) [1, 2, 3]; 1) [5, 9]; 2) [3, 6]; 3) [1, 6]; 4) [9]; 5) [8]; 6) [7, 8]; 7) [4]; 8) [4, 7]; 9) [].

Предлагаемый программный комплекс имеет возможность посчитать количество уровней в генерируемых графах вычислений на основе алгоритма обхода графа в ширину. Алгоритм является более предпочтительным, чем алгоритм Дейкстры, с помощью которого также имеется возможность найти количество уровней в графе. Однако, алгоритм обхода в ширину имеет сложность не O(|V| + |E|), где V – множество вершин, E – множество ребер, а алгоритм Дейкстры – $O(n^2)$, т.е. основной цикл выполнится n раз и каждый раз для подсчета количества уровней в графе будет тратиться порядка n операций.

Программный комплекс позволяет создавать графы вычислений любой размерности и с любым количеством связей, которое ограничено только имеющимся количество модулей в графе вычислений.

На рисунке 35 представлен граф вычислений с количеством модулей, равным 40.

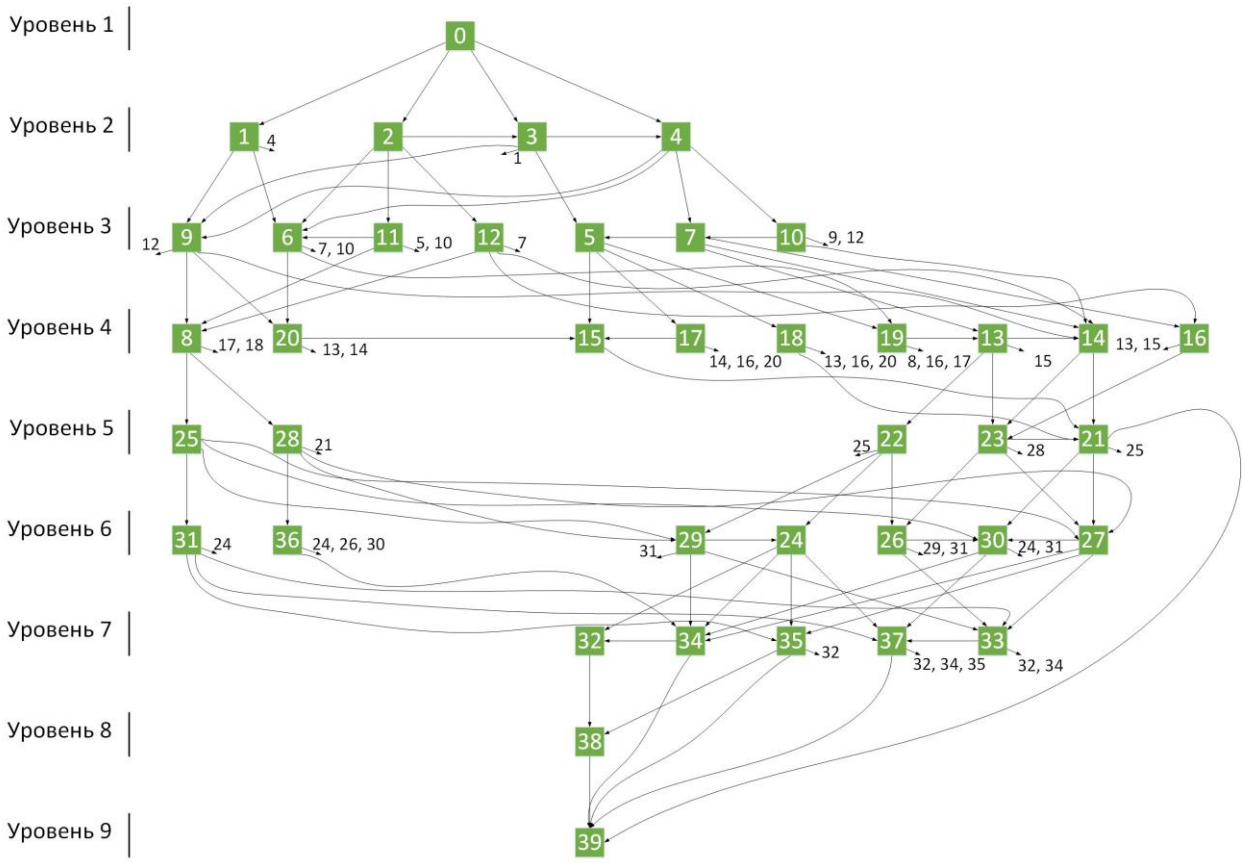

Рисунок 35 – Граф вычислений с количеством вершин, равным 40

На рисунке выше представлен другой вычислительный граф. Для того чтобы повысить ясность, связи внутри каждого уровня представлены в виде стрелки с номерами вершин, с которыми имеет связь текущий модуль. Как можно заметить, граф создан со следующими параметрами: количество модулей равно 40, количество связей у модуля первого уровня равно 4, количество связей у модулей последующего уровня не больше 4-х. Максимальное количество уровней в графе – 9, минимальное (минимальное количество уровней в графе, которое необходимо преодолеть от первой до последней вершины) – 6.

Для всестороннего анализа обработки вычислительных графов были созданы их следующие модели, которые представлены в таблице 3.

Таблица 3 – Сгенерированные графы вычислений

Количество модулей	Количество связей от модуля первого уровня			Количество связей от модулей последующего уровня			Минимальное количество уровней			Среднее время создания графа, мс
	1	2	3	1	2	3	1	2	3	
3	2	2	2	2	2	2	2	2	2	75
4	2	2	3	2	2	2	3	3	2	177
5	3	3	3	2	2	2	3	3	3	91
6	2	2	3	2	3	2	4	3	3	102
7	2	3	4	2	2	2	4	4	3	106
8	2	3	4	2	3	2	4	4	4	97
9	2	3	4	2	3	3	4	4	3	120
10	2	3	3	2	2	3	4	5	4	112
11	2	3	4	2	2	3	6	5	4	132
12	2	3	4	2	3	4	5	5	4	148
13	3	4	4	3	2	4	5	6	4	104
14	3	4	5	3	2	2	5	5	5	128
15	2	4	4	2	2	3	7	5	5	133
16	3	4	6	3	2	2	5	7	5	162
17	2	3	5	3	3	4	5	5	4	165
18	3	4	5	3	4	2	7	5	8	174
19	2	4	6	3	4	4	6	4	4	423
20	2	3	4	4	3	4	6	6	5	151
25	3	4	4	3	4	8	7	7	4	359
30	4	5	10	4	5	5	7	5	5	394
35	4	4	10	4	8	4	7	5	6	496
40	4	8	10	4	4	8	8	7	5	903

Продолжение таблицы 3

Количество модулей	Количество связей от модуля первого уровня			Количество связей от модулей последующего уровня			Минимальное количество уровней			Среднее время создания графа, мс
	1	2	3	1	2	3	1	2	3	
45	5	8	15	5	8	5	7	5	6	651
50	4	5	10	4	5	5	9	8	7	594
60	8	10	20	8	6	4	6	8	9	2144
70	4	6	9	5	6	9	10	9	7	1613
80	9	10	12	9	10	12	7	7	6	44446
90	8	12	20	8	12	5	9	6	11	11857
100	12	15	20	10	6	8	8	12	8	11574
150	5	5	25	5	5	4	19	20	21	22387
200	10	20	50	10	10	12	13	14	10	56505
250	15	25	40	15	5	10	11	30	15	2239849
300	25	50	60	10	5	10	18	32	16	346564
350	50	100	120	12	4	6	18	37	25	535426
400	25	50	80	7	10	15	33	23	15	938844
450	25	50	100	7	10	6	39	27	34	952221
500	60	80	100	8	15	10	33	19	23	15986798
2000	350			8			129			364805665

На основе данных из таблицы 3 можно построить график отображения времени, необходимого для создания графов вычислений с различным количеством модулей, а также разной степенью параллелизма. График является аппроксимацией табличных значений и показан на рисунке 36.

Рисунок 36 – График времени создания графа вычислений

На основании показаний графика можно сделать вывод, что время, отведенное на создание графа вычислений, имеет полиномиальный рост. Если граф с количеством вершин, равным 3-м, создается за 75 мс, то для графа с количеством вершин, равным 2000, время создания уже равно 364805665 мс. Подобный рост связан с тем, что при увеличении модели в 2 раза, матрица графа увеличивается в 4 раза, что усложняет проверку графа на цикличность, увеличивая время проверки. В отличие от алгоритма обработки графа вычислений, алгоритм проверки на цикличность в графе выполняется в одном потоке. Основу алгоритма составляет класс DoubleConnection со следующим кодом:

```
/* Сканирование на предмет циклов. */
for (int l = 0; l < majorList.size(); l++) {
    for (int m = 0; m < majorList.size(); m++) {
```

```
/* Если переменные совпадают (именно equals ввиду того, что происходит сравнение значений, без учета ссылок на объект, которые могут быть разными), а номера элементов массива - нет, то значит, добавлена вторая такая же вершина в список и произошло зацикливание. */
    if (Objects.equals(majorList.get(l).get(0).get(0), majorList.get(m).get(0).get(0)) && l != m) {
        int g = majorList.get(l).get(0).get(2); // Ячейка со связями вершины. l - номер вершины.
        /* Отнимаем на единицу, чтобы взять именно эту вершину (если начинается обработка вершины, то p уже увеличивается на единицу и уже готова для обработки следующей вершины). Отнимая единицу, мы возвращаемся к предыдущей связи, в которой есть вложенный цикл. Из MajorList связь, вызвавшая цикл не удаляется ввиду того, что при нахождении цикла метод DoubleConnectionTest перезапускается.*/
        g--;

        /* Удаляем соответствующую связь из цикла 1. Проходим по связям вершины и вычисляем связь, которая является причиной цикла.*/
        for (int n = 0; n < listofOne.get(majorList.get(l).get(0).get(0)).size(); n++) {
            /* Именно equals ввиду того, что происходит сравнение значений, без учета ссылок на объект, которые могут быть разными. Сравниваем каждую связь вершины со связью той вершины, которая вызвала цикл. Далее происходит удаление соответствующей вершины. */

            if(Objects.equals(listofOne.get(majorList.get(l).get(0).get(0)).get(n),
            listofFive.get(majorList.get(l).get(0).get(0)).get(g))) {
                // Удаление соответствующей связи из списка 1 для прерывания зацикливания.
```

```
                listofOne.get(majorList.get(l).get(0).get(0)).remove(n);
            }
        }

        /* Удаление соответствующей связи из списка 5 для прерывания
        зацикливания. */
        listofFive.get(majorList.get(l).get(0).get(0)).remove(g);
        /* Значит цикл был, он исправлен и необходимо выйти из метода.
        Возвращая переменную f, которая не равна нулю (т.к. у нулевой
        вершины нет внутренних циклов). f - текущая позиция перебора.
        Позволяет начать выполнение метода с предыдущей позиции
        перебора, что уменьшает выполнение задачи. */
        return f;
      }
   }
}
```

Таким образом, для программно-имитационной проверки алгоритмов балансировки комплекс позволяет создавать вычислительные графы любой размерности. Временные ограничения вступают в силу только при количестве вершин в графе свыше 2000. Ограничение связано с тем, что алгоритму необходимо проверить каждую вершину в вычислительной модели на предмет наличия циклов и удалить связи, которые их образуют.

2.2 Алгоритм балансировки нагрузки на основе динамического вычисления рангов вычислителей по заданным критериям

Иллюстрация работы алгоритма балансировки, который основан на анализе модулей и доступных вычислителей, приведена ниже. В качестве примера используется вычислительный граф с размерностью 20. Количество связей у вершины первого уровня равно 2, у вершин последующего уровня

максимальное количество связей равно 4, количество уровней в графе равно 6. Для его обработки используется количество потоков, равное 3-м. Допустим, что каждый модуль в примере выполнения графа выполняется за одинаковый промежуток времени, что имитационная модель позволяет задать. Также, в методе не используются вычислители, что значительно облегчает предоставленный анализ работы.

Для удобства рассуждений вводится вспомогательная структура – список (пул) доступных для обработки модулей, т.е. модулей, которые могут быть выполнены при наличии доступных вычислителей.

Принцип работы предлагаемого метода балансировки нагрузки (без учета характеристик вычислителей) представлен на рисунке 37.

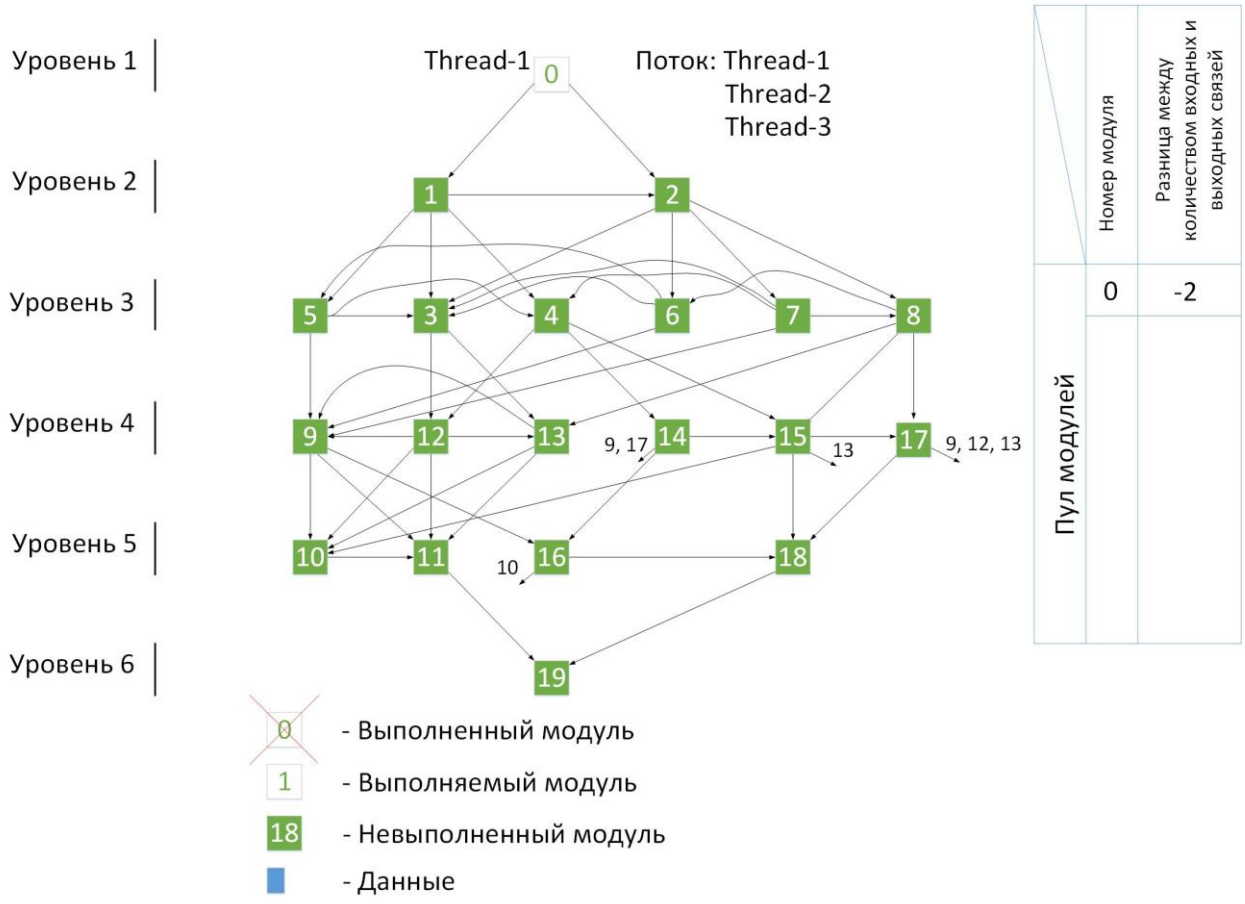

Рисунок 37 – Принцип работы предлагаемого метода балансировки нагрузки, обработка модуля-0

Выполнение вычислительного графа начинается с обработки модуля-0. Модуль заносится в специальный пул доступных модулей, который выбирает подходящий модуль на основе определенного критерия. Критерием является разница между количеством входных и выходных связей в модуле. Чем меньше значение разности, тем предпочтительнее является модуль. Это связано с тем, что каждый модуль имеет определенное количество входных и выходных связей и чем меньше входных связей у модуля, тем выше вероятность того, что он будет обработан раньше, т.к. для обработки ему требуется меньшее количество данных от меньшего количества модулей. Также, чем больше выходных связей у обрабатываемого модуля, тем шире возможности для обработки модулей, т.к. их количество в списке возрастает и тем больше шансов, что для обработки вычислительного графа будет задействовано максимальное количество потоков и вычислителей, а значит и максимальная вычислительная мощность самой системы.

Именно поэтому, на основе описанного алгоритма выбирается для обработки модуль-0, который является единственным в списке на текущий момент. Для его обработки используется поток Thread-1. После обработки модуля-0 он удаляется из списка и в список заносятся его дочерние модули: 1 и 2, что показано на рисунке 38.

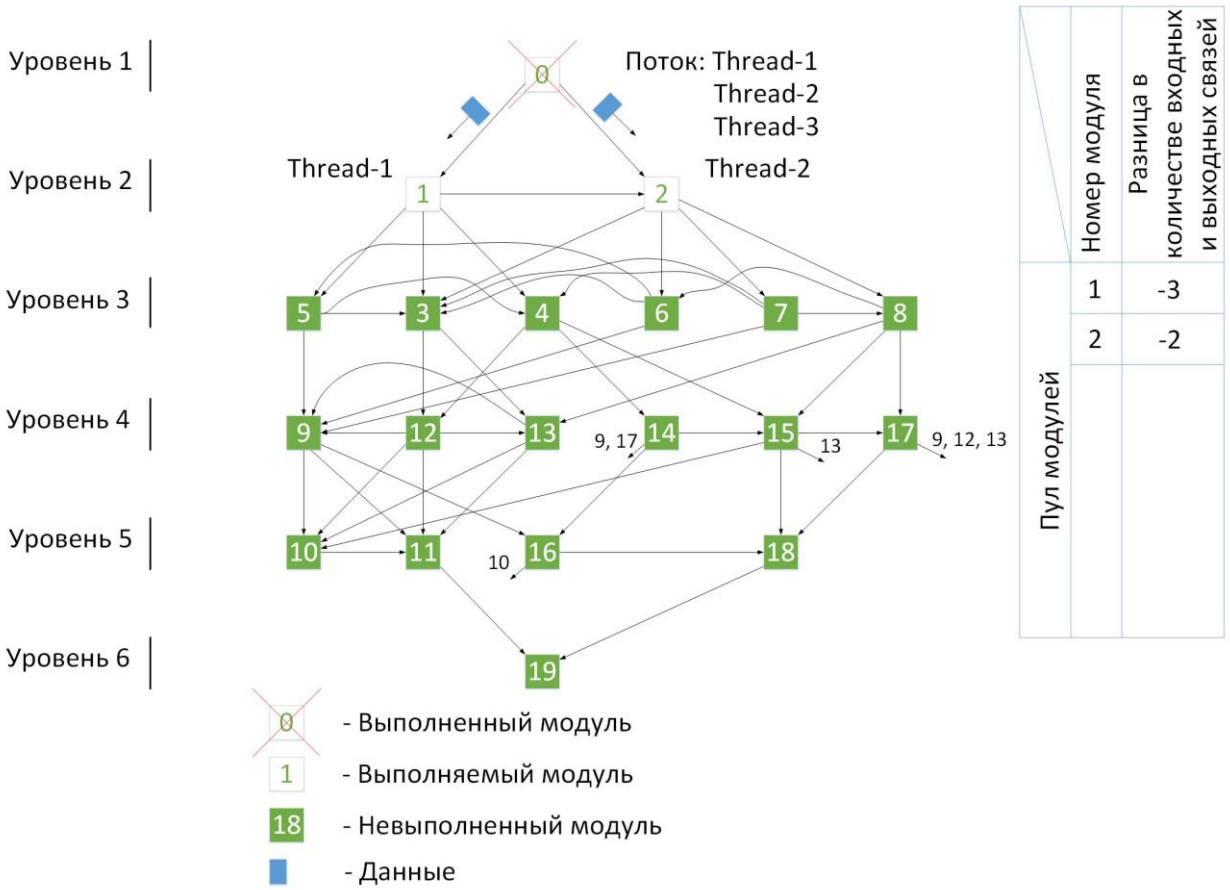

Рисунок 38 – Принцип работы предлагаемого метода балансировки нагрузки, обработка модулей 1 и 2

Количество модулей в списке доступных также меньше количества используемых потоков, что даёт возможность для вычисления всех модулей: модуль-1 потоком выполнения Thread-1, модуль-2 потоком Thread-2. На рисунке 39 показан следующий шаг работы алгоритма балансировки нагрузки. Потоки выполнения также являются и потоками данных, т.е. происходит имитация передачи данных.

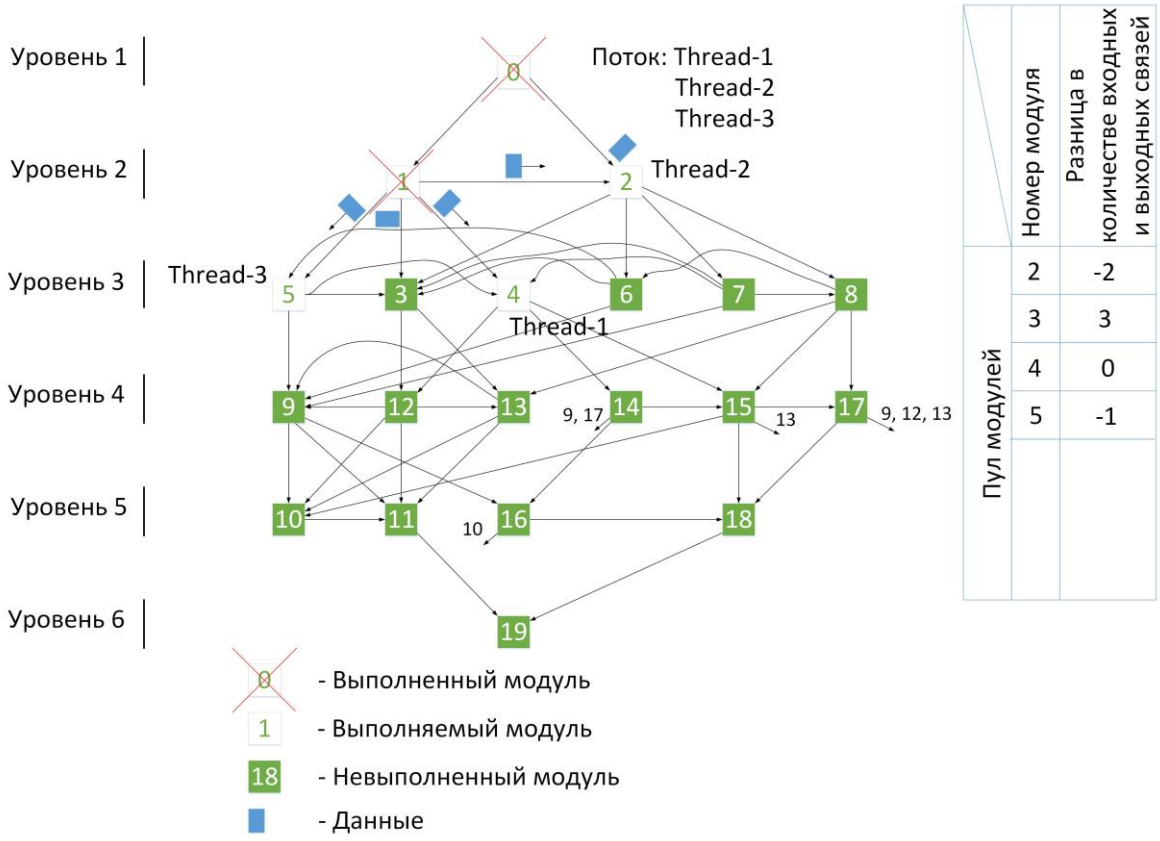

Рисунок 39 – Принцип работы предлагаемого метода балансировки нагрузки, обработка модулей 2, 4, 5

Поток выполнения Thread-1 выполнил модуль-1. В качестве входных данных для модуля используются только данные модуля-0. В свою очередь, поток выполнения Thread-2, проанализировав модуль-2, получил информацию, что для его выполнения необходимы также данные из модуля-1. Таким образом, модуль-2 после обработки модуля-1 снова заносится в список доступных для обработки. Помимо него, в список доступных модулей для обработки также заносятся следующие модуля: 3, 4, 5.

В списке доступных модулей теперь их количество больше, чем доступных потоков выполнения. Таким образом, происходит выбор модулей, которые имеют наименьшее значение разности количества входных и выходных связей. Приоритет отдается модулям, где количество выходных связей должно быть больше, чем входных. Такими модулями являются: модуль-2, в котором количество выходных связей на две больше, чем

количество входных: модуль-5, в котором количество выходных связей на одну больше, чем количество входных; модуль-4, в котором количество входных связей равно количеству выходных. Модуль-3 остается в списке и ждет своей очереди на обработку.

Результат обработки методов и следующий шаг алгоритма представлены на рисунке 40.

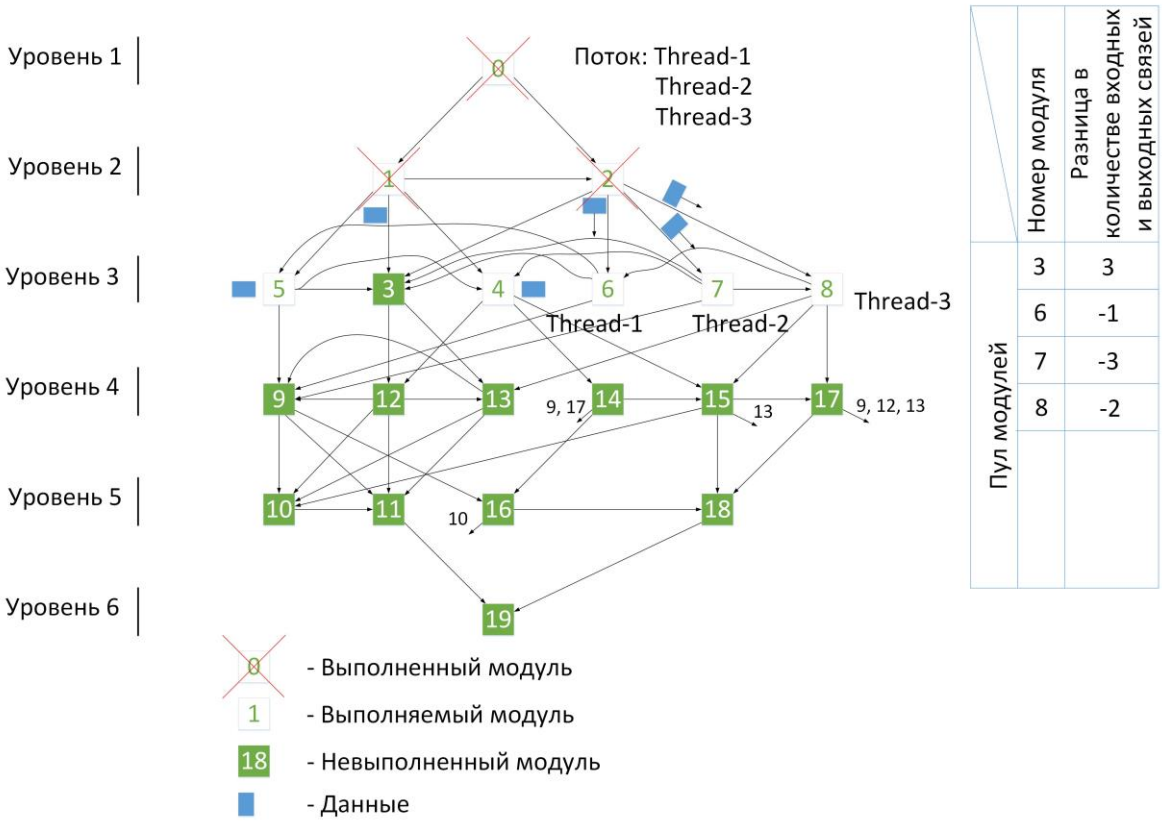

Рисунок 40 – Принцип работы предлагаемого метода балансировки нагрузки, обработка модулей 6, 7, 8

На основе рисунка, представленного выше, можно увидеть, что в пул доступных модулей после обработки модуля-2, который в отличие от модулей-4 и -5 получил все входные данные, дополнился дочерними модулями 6, 7 и 8.

Таким образом, аналогично описанному выше алгоритму, происходит выбор модулей 6, 7, 8 ввиду большей разности между количеством входных

и выходных связей. Поток выполнения Thread-1 выбирает модуль-6, поток Thread-2 – модуль-7, а поток Thread-3 – модуль-8.

Результат обработки модулей представлен на рисунке 41.

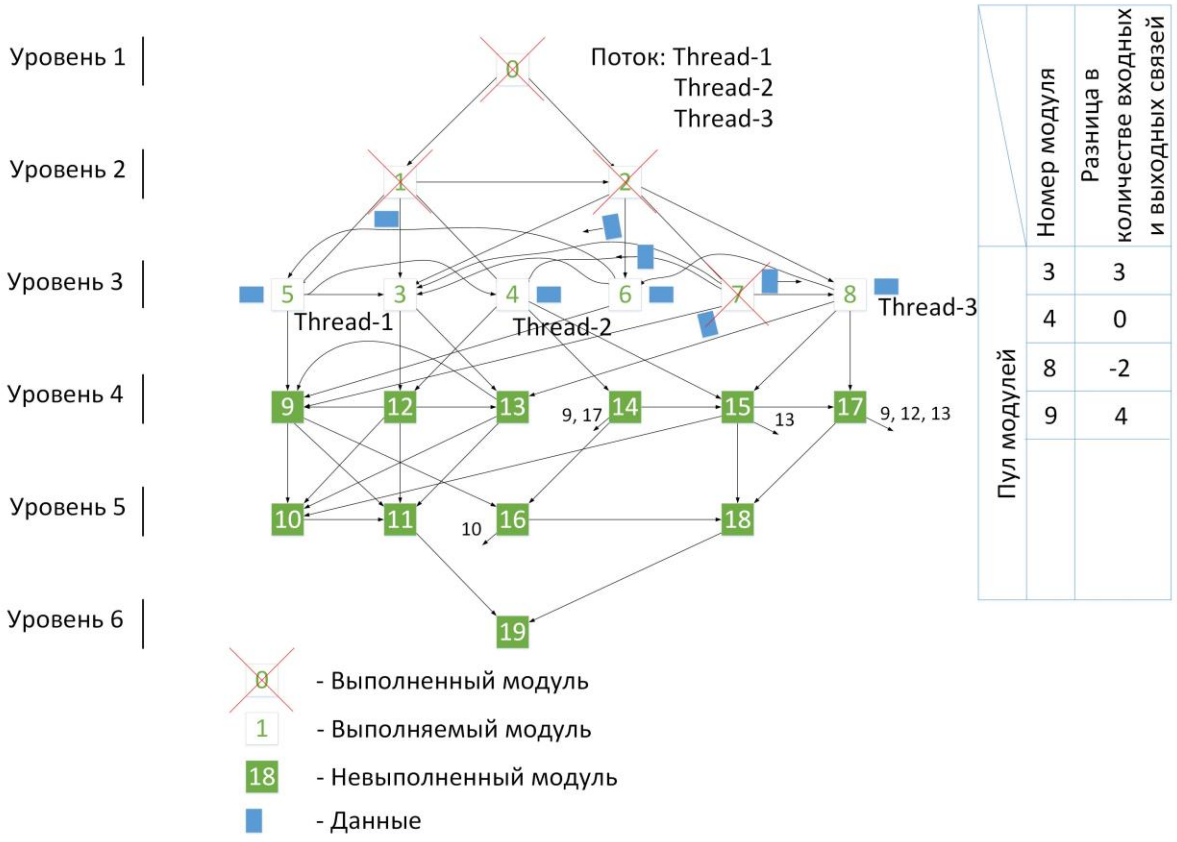

Рисунок 41 – Принцип работы предлагаемого метода балансировки нагрузки, обработка модулей 3, 4, 8

После обработки модуля-7 и его удаления из списка, в список доступных добавляются следующие модуля: 4, 8, 9. На основе предлагаемого алгоритма балансировки происходит выбор модуля-3, модуля-4 и модуля-8. Модуль-9 остается ожидать ввиду того, что количество выходных связей у него значительно меньше количества входных.

Основой вышеописанного метода балансировки нагрузки на основе анализа вычислительного графа является библиотека java.util.concurrent и семафоры, один из которых предназначен для допуска потоков к графу. В графе не может одновременно работать количество потоков, которое больше,

чем установлено в параметре семафора. Также, каждый модуль графа имеет свой семафор, который позволяет ограничивать обработку модуля всего одним потоком выполнения. Каждый модуль выполнения представлен следующим конструктором:

```
/* Создаем новый конструктор для списка, который будет содержать полную информацию о каждой вершине. Также добавляем туда количество входных связей для каждой вершины. */
class TypeForList {
    ArrayList<Integer> listForOutputAgile = new ArrayList<>(); // Количество выходных связей.
    ArrayList<Integer> listForInputAgile = new ArrayList<>(); // Количество всходных связей.

    /* Список для проверки того, потоки с каких вершин соединились с данной вершиной. Например, вершине 2 необходимы для выполнения данные из вершин 1 и 3. После подключения каждой вершины происходит проверка, все ли вершины подключились (совпадают ли значение вершин со всеми вершинами из списка входных связей) - listForInputAgile. Если совпадают, то происходит выполнение вершины. Иначе - ожидание момента поступления потоков из оставшихся вершин.*/
    ArrayList<Integer> listForCheck = new ArrayList<>();

    /* Основная информация о вершине (порядковый номер, общее количество входных связей, количество обработанных связей, время выполнения каждого модуля, количество потоков, которое необходимо, чтобы модуль начал выполняться (т.е. количество входных связей)).  */
    ArrayList<Integer> listForData = new ArrayList<>();

    Semaphore sem; // Семафор.
```

```
    TypeForList(ArrayList<Integer> listForOutputAgile, ArrayList<Integer>
    listForInputAgile, ArrayList<Integer> listForCheck, ArrayList<Integer>
    listForData, Semaphore sem) {
        this.listForOutputAgile = listForOutputAgile;
        this.listForInputAgile = listForInputAgile;
        this.listForCheck = listForCheck;
        this.listForData = listForData;
        this.sem = sem;
    }
}
```

Алгоритм заканчивает своё выполнение обработкой последней вершины, после которой данные могут быть предоставлены пользователю. Описанный метод обработки повышает эффективность выполнения задачи, что показано далее в результатах проведенных экспериментов и сравнении текущего метода балансировки нагрузки с методами Dynamic Round Robin и Dynamic Random.

2.3 Задание характеристик вычислителей

Предлагаемый алгоритм балансировки нагрузки, помимо анализа вычислительного графа, производит также анализ предоставленного пула вычислителей, доступны ли вычислители для обработки. Если вычислитель доступен, то он вызывается для обработки модуля из списка доступных модулей, принцип действия которого была описан ранее. Диспетчер балансировки нагрузки постоянно анализирует и обновляет список, что является эффективным способом подавать запросы только на вычислители, которые имеют возможность в текущий момент времени проводить обработку модуле. Таким образом, достигается динамичность в предлагаемом методе балансировки нагрузки.

Применяемый алгоритм балансировки нагрузки представлен на рисунке 42.

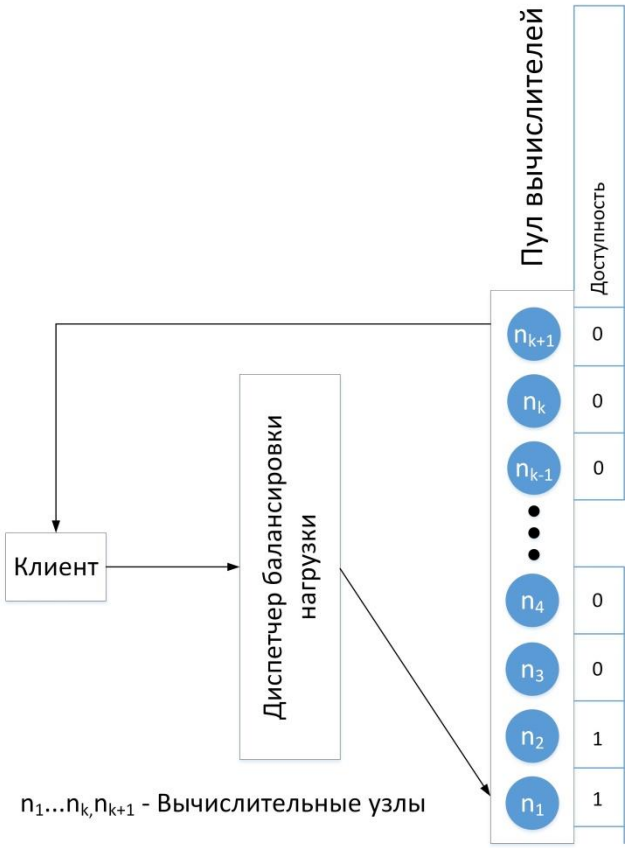

Рисунок 42 – Предлагаемый метод балансировки нагрузки в однородной вычислительной системе

На рисунке выше показано применение предлагаемого метода балансировки нагрузки в однородной вычислительной системе, в которой каждый вычислитель обладает одинаковой производительностью. Однако, программный комплекс позволяет задавать различные вычислители, которые могут отличаться между собой по производительности процессора, объему оперативной памяти, времени доступа к вычислительному узлу.

Каждая предлагаемая характеристика изменяет время задержки для потока при обработке модуля. Алгоритм на основе определенных вычислений позволяет генерировать разное время для обработки одного модуля разными вычислителями. При этом каждый вычислитель имеет не

определенное время задержки, а переменную, которая увеличивает время выполнения задачи. То есть, сложная вычислительная задача будет выполняться дольше не только на основе своей задержки, но и благодаря вычислителю, который разные вычислительные задачи будет выполнять с разным временем.

В неоднородной вычислительной системе программный комплекс различает узлы по следующим характеристикам и комбинациям характеристик:

1) Производительность процессора;
2) Объем оперативной памяти вычислительного узла;
3) Время доступа к вычислительному узлу;
4) Производительность процессора + объем оперативной памяти вычислительного узла;
5) Производительность процессора + время доступа к вычислительному узлу;
6) Объем оперативной памяти вычислительного узла + время доступа к вычислительному узлу;
7) Производительность процессора + объем оперативной памяти вычислительного узла + время доступа к вычислительному узлу.

Чем выше числовое значение производительности, объема оперативной памяти, а также времени доступа к вычислительному узлу, тем мощнее вычислительный узел и тем меньше генерируемая задержка на каждом модуле графа.

Совокупность характеристик не является среднеарифметическим значением, а высчитывается с учетом того, что время доступа к узлу влияет на вычислительную мощность сильнее, чем объем оперативной памяти и производительность процессора.

Таким образом, в неоднородной вычислительной системе на основе предлагаемого метода балансировки диспетчер нагрузки среди доступных

вычислительных узлов выбирает наиболее подходящий на основе предлагаемых характеристик.

Для простоты каждая характеристика имеет вес от 1 (наименее производительная) до 10 (наиболее производительная).

Пример выбора подходящего вычислительного узла представлен на рисунке 43.

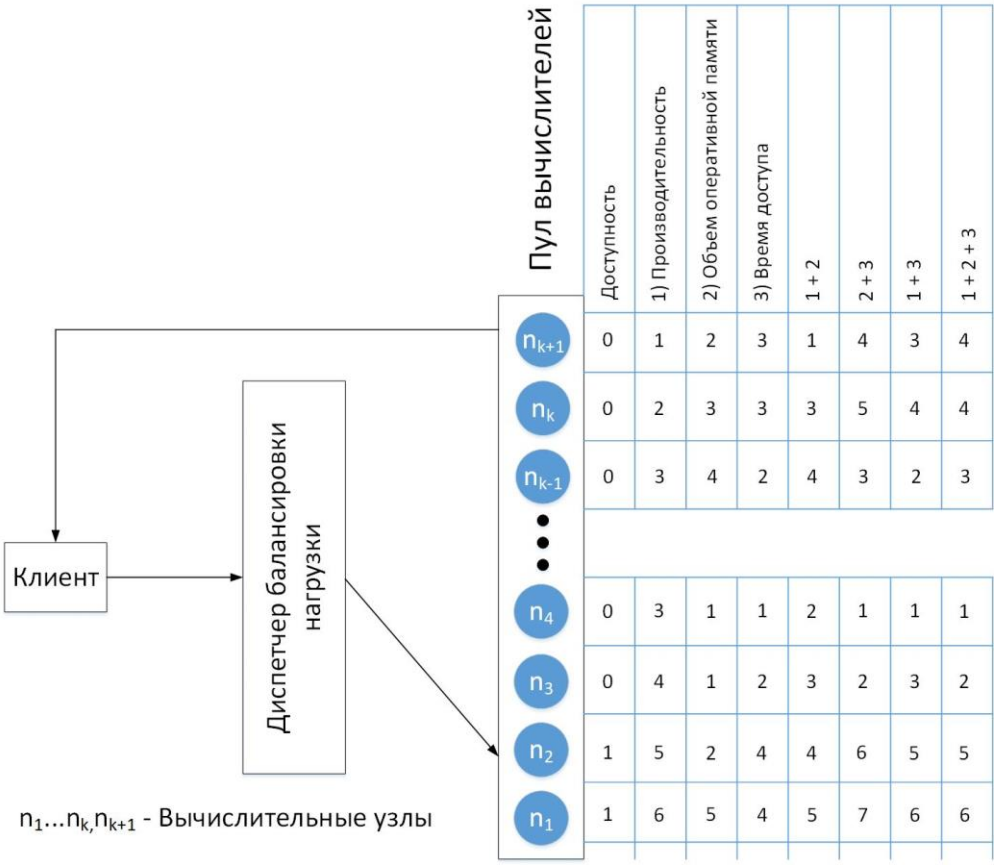

Рисунок 43 – Предлагаемый метод балансировки нагрузки в неоднородной вычислительной системе

Таким образом, основу предлагаемого алгоритма балансировки составляют анализ графа вычислений на предмет наибольшей разницы между входными и выходными связями, что позволяет эффективно обрабатывать поставленные задачи и подбор диспетчером балансировки нагрузки наиболее производительного из свободных вычислительных узлов для вычисления модуля из списка доступных.

6 Финансовый менеджмент, ресурсоэффективность и ресурсосбережение

Финансовая часть выпускной квалификационной работы позволяет определить перспективность проведения научно-исследовательских работ в области балансировки нагрузки в распределенных вычислительных системах, а также выработать план научных исследований и определить социальную, экономическую и финансовую эффективность разработки.

Описание эффективности различных методов балансировки нагрузки с применением вычислительных устройств с отличающимися характеристиками позволяет вычислить характеристики и особенности того или иного метода для выполнения широкого спектра задач. Для этих целей разработана имитационная среда моделирования, которая позволяет проводить моделирование методов балансировки нагрузки и получать статистические данные. Эти данные можно использовать для оценки эффективности и выбора наиболее подходящего метода при выполнении определенной задачи и использовать его для проектирования реальной горизонтально масштабируемой территориально-распределенной вычислительной системы.

Также был разработан собственный метод балансировки нагрузки, который основан только на программной реализации и продемонстрировал конкурентоспособные результаты при сравнении с общими и наиболее распространенными методами балансировки нагрузки.

6.1 Область практического применения балансировки нагрузки

Балансировка нагрузки встречается повсеместно. Область применения балансировки ограничивается только возможностями вычислительных кластеров, которые проводят различные научные исследования, анализ и

обработку больших объемов данных, а также поддерживают работу высоконагруженных веб-сайтов.

Любая трудоёмкая вычислительная задача требует применения различных алгоритмов балансировки. Среди описываемых задач имеются такие, как: анализ спутниковых метеоданных и прогнозирование климатических изменений, предсказание различных экстремальных процессов и явлений, моделирование кровеносной системы тела человека, анализ сигналов радиотелескопов для поиска внеземных источников жизни, моделирование городской среды, выполнение сложных математических операций и т.д. [1]

Также стоит отметить, что любой высоконагруженный сайт, с большим количеством пользователей и большим трафиком применяет балансировку нагрузки (например, Yandex, VK, Google, Facebook и т.д.).

Среди компаний, которые используют свои разработки, имеются такие, как: Oracle (Real Application Clusters, Grid Infrastructure), Google (Cloud Load Balancing), Facebook, Amazon (Elastic Load Balancing) и т.д.

Многие используют как аппаратные (Cisco – CSS L4, ACE L7; Citrix – NetScaler; F5 – BIG-IP; Radware – ADC (Application Delivery Controller) и т.д.), так и программные (Nginx; HAProxy; Perlbal и т.д.) продукты других компаний.

Часто применяемые методы балансировки нагрузки являются коммерческой тайной, однако, существует достаточное количество открытых методов, информацию о принципе работы которых можно достаточно легко найти.

6.2 Анализ конкурентных технических решений

Рассмотрим основные методы балансировки нагрузки, информацию о которых можно найти в свободном доступе. В качестве конкурентов представим следующие методы балансировки: Random; Round Robin;

Dynamic Random; Dynamic Round Robin; Dynamic Weighted Random; Dynamic Weighted Round Robin; Token Aware; Least Connections; Observed; Weighted Least Traffic.

Описанные в разделе алгоритмы будут являться конкурентами разработанному методу балансировки. Сведем характеристики каждого метода, включая разработанный метод, в оценочную карту для сравнения конкурентных технических решений (приложение А).

Анализ конкурентных технических решений определялся по формуле [72]:

$$K = \sum B_i * Б_i, \quad (1)$$

где К – конкурентоспособность научной разработки или конкурента;

B_i – вес показателя (в долях единицы);

$Б_i$ – балл i-го показателя.

Исходя из полученных данных о конкурентоспособности, можно разделить конкурентов на две части. Конкуренты, показатели которых лучше, чем показатели разработанного алгоритма: Dynamic Weighted Random, Dynamic Weighted Round Robin, Token Aware, Observed, Weighted Least Traffic.

Конкуренты, показатели которых хуже, чем показатели разработанного алгоритма: Random, Round Robin, Dynamic Random, Dynamic Round Robin, Least Connections.

По данным таблицы можно понять, чем обусловлено то, что некоторые алгоритмы превосходят спроектированный: затраты на разработку, распространенность, улучшенная поддержка неоднородной вычислительной системы, повышенная надежность и удобство эксплуатации.

Недостатками алгоритмов, которые показали себя хуже, чем спроектированный алгоритм балансировки нагрузки, являются следующие параметры: повышение скорости обработки данных, надежность, возможность работы с неоднородной вычислительной системой, цена.

Таким образом, можно выделить следующие конкурентные преимущества разработанного алгоритма балансировки нагрузки: низкая стоимость при повышенной скорости обработки данных, а также возможность работать с неоднородной вычислительной системой. Преимуществом также является то, что разработанный алгоритм написан на языке программирования Java, отличительными особенностями которого являются повышенная отказоустойчивость, безопасность, а также архитектурная независимость. Стоит отметить, что разработанный алгоритм не требует применения специальных аппаратных платформ.

6.3 Определение возможных альтернатив проведения научных исследований

Ввиду того, что разработка представляет собой имитационную среду моделирования для проведения экспериментов по балансировке нагрузки и дальнейшего анализа полученных результатов, то для проведения научного исследования необходимо воспользоваться морфологическим подходом.

Морфологический подход основан на систематическом исследовании всех теоретически возможных вариантов, которые вытекают из закономерностей строения (морфологии) объекта исследования (распределения вычислительной нагрузки в многоядерных системах).

Таким образом, синтез охватывает как известные (примененные в ходе создания имитационной среды моделирования), так и новые, необычные варианты, которые при простом переборе могли быть упущены [72]. Благодаря комбинированию различных вариантов можно получить большое количество решений.

При реализации метода для начала необходимо точно сформулировать проблему исследования. Проблема исследования состоит в нахождении статистической зависимости между размером вычислительного кластера, методом балансировки нагрузки, характеристиками каждого вычислителя, а

также моделью выполняемой задачи и выбором наиболее оптимального варианта посредством спроектированной имитационной среды моделирования, что непосредственно необходимо для создания реальной распределенной вычислительной системы.

Разработанная программная имитационная среда моделирования позволит оптимальным способом спроектировать реальную систему с учетом имеющихся финансовых ресурсов.

Далее необходимо раскрыть все важные морфологические характеристики объекта исследования. Для программной имитационной среды моделирования балансировки нагрузки в распределенных вычислительных системах можно выделить следующие характеристики:

а) механизм создания модели вычислительной задачи;

б) проверка на правильность сгенерированной модели вычислительной задачи;

в) метод сохранения спроектированной модели вычислительной задачи;

г) алгоритм подсчета количества уровней графа подзадач в модели вычислительной задачи (минимальное количество подзадач, которое необходимо выполнить для достижения последней подзадачи, после которой следует результат).

д) выбор метода балансировки нагрузки;

е) выбор количества одновременно используемых потоков обработки задачи (т.е. вычислителю) при выполнении метода балансировки нагрузки;

ж) задание характеристик каждого вычислителя;

з) механизм получения статистической информации об обработке спроектированной модели вычислительной задачи;

и) используемое вычислительное устройство для моделирования;

к) язык программирования, с помощью которого написана программная имитационная среда моделирования.

После раскрытия важных морфологических характеристик необходимо составить морфологическую матрицу, которая представлена в приложении Б.

После составления морфологического плана необходимо выбрать наиболее желательные функционально конкретные решения (варианты исполнения). В качестве решений были выбраны три варианта исходя из данных приложения Б:

1) I2; II2; III2; IV1; V2; VI2; VII2; VIII3; IX2; X3.
2) I1; II1; III1; IV2; V3; VI1; VII1; VIII2; IX1; X1.
3) I3; II3; III3; IV3; V1; VI3; VII4; VIII1; IX3; X4.

Опишем преимущества и недостатки каждого варианта относительно каждой морфологической характеристики:

1) Механизм создания модели вычислительной задачи. Для данной характеристики наиболее сложным в исполнении является вариант 3 ввиду того, что пользователь строго задает как количество уровней в графе подзадач, так и количество вершин, в отличие от вариантов 1 и 2, где пользователь строго задаёт только одну определенную характеристику, на основе которой будет происходить генерация графа. Другая характеристика является произвольной, что значительно расширяет возможности генерации вычислительных графов.

2) Проверка на правильность сгенерированной модели вычислительной задачи. В данном случае проверка на правильность сгенерированной модели равнозначна по сложности для варианта 1 и варианта 2. Не имеет значения очередность проверки. Проверка вручную варианта 3 значительно проигрывает в плане затрачиваемого времени написанному алгоритму.

3) Метод сохранения спроектированной модели вычислительной задачи. Наиболее выигрышным является вариант 1, т.к. автоматическое сохранение значительно облегчает задачу генерации вычислительного графа. Вариант 2 также является довольно простым ввиду того, что нет необходимости писать алгоритм сохранения графа, однако пользователю самому приходится копировать информацию в файл из стандартного потока

вывода, на что тратится время. Вариант 3 является наименее ресурсоэффективным среди выбранных вариантов ввиду траты большого количества времени.

4) Алгоритм подсчета количества уровней в модели вычислительной задачи. Наиболее подходящим для данной задачи является алгоритм на основе обхода графа в ширину, предложенный в варианте 1. Алгоритм Дейкстры варианта 2 является также действенным, однако ввиду того, что в вычислительном графе отсутствуют веса рёбер, алгоритм является менее подходящим и более сложным, чем алгоритм варианта 1. Сложность алгоритма Дейкстры – $O(n^2)$, т.е. основной цикл выполняется n раз и в каждом из них для подсчета количества уровней в модели тратится порядка n операций. Сложность алгоритма на основе обхода графа в ширину – $O(|V| + |E|)$, где V – множество вершин, E – множество ребер, что значительно меньше сложности алгоритма Дейкстры. Вариант 3 является наиболее затратным по времени ввиду того, что сначала необходимо отобразить граф на бумаге удобным для пользователя образом, что значительно усложняет поиск количества уровней.

5) Выбор метода балансировки нагрузки. Автоматический выбор метода (вариант 3) на основе предоставленного вычислительного графа является сложным и ресурсозатратным способом выбора метода балансировки нагрузки. Наименее затратным является вариант 2, который не позволяет выбрать метод балансировки нагрузки ввиду наличия только одного метода, однако данный способ не отображает полной картины и не является вполне подходящим для научного исследования.

6) Выбор количества одновременно используемых потоков обработки задачи. Вариант 1 обладает высокой гибкостью ввиду того, что пользователь сам имеет возможность настроить количество одновременно используемых потоков. Однако, в свою очередь, вариант 2 позволяет пользователю не вмешиваться в процесс обработки задачи, т.к. программа сама запустит требуемое число потоков на основе предоставляемой вычислительной

мощности устройства. Этот вариант обладает меньшей гибкостью, но позволяет повысить удобство использования имитационной среды моделирования . Вариант 3 является очень простым в разработке ввиду отсутствия возможности генерации потоков. Программа работает в одном потоке, и все подзадачи выполняются последовательно.

7) Задание характеристик каждого вычислителя. Наиболее подходящим является вариант 1, который позволяет вручную задавать характеристики каждого узла во время имитации вычислительного процессора системой. Также удобен вариант 2 (автоматическое задание характеристик каждого узла). Вариант 3 является наименее выгодным, однако обладает наименьшим временем создания ввиду того, что он позволяет не учитывать вычислительных характеристики узлов системы.

8) Механизм получения статистической информации об обработке спроектированной модели вычислительной задачи. В отличие от варианта 2 и 3, которые позволяют работать только с определенными сервисами (это могут быть средства языка программирования или средства сторонних приложений), то вариант 1 позволяет совместить в себе варианты 2 и 3. Однако это усложняет проектирование данного варианта.

9) Используемое вычислительное устройство для моделирования. Выбор устройства зависит как от предпочтений инженера (дипломника), так и от его профессиональных навыков проектирования систем для различных вычислительных устройств и различных операционных систем.

10) Язык программирования, с помощью которого написана программная имитационная среда моделирования. Каждый язык программирования имеет свои плюсы и минусы, однако стоит выделить язык программирования Java, в котором достаточно просто производить работу с проектированием многопоточных приложений. Также данный язык программирования является наиболее гибким при запуске на различных платформах, безопасным ввиду наличия виртуальной машины Java,

высокопроизводительным и надежным. Данный язык программирования выбран в варианте 1.

6.4 Планирование научно-исследовательских работ

6.4.1 Структура работы в рамках научного исследования

Планирование текущей научно-исследовательской работы состоит из следующих этапов:

1. Определение структуры и состава научно-исследовательской работы в рамках исследования;

2. Определение участников каждой работы;

3. Установление продолжительности выполнения каждой работы;

4. Построение графика проведения научных исследований.

Для выполнения научных исследований была создана рабочая группа, которая состоит из руководителя темы работы и инженера (дипломника). Далее необходимо составить перечень различных этапов выполнения научно-исследовательской работы и провести распределение исполнителей по их видам.

Порядок составления этапов выполнения научно-исследовательской работы, а также распределение исполнителей между этапами выполнения работ представлено в приложении В.

6.4.2 Определение трудоёмкости выполнения работ

После составления и описания перечня этапов работ и распределения исполнителей необходимо определить трудоёмкость выполнения работ. Трудовые затраты – это основная составляющая стоимости разработки. Ввиду этого необходимо определить трудоёмкость работы каждого из участников научно-исследовательской работы.

Трудоёмкость выполнения исследовательской работы оценивается экспертным путём и измеряется в человеко-днях. Данная величина носит вероятностный характер, т.к. зависит от множества трудно учитываемых факторов [72].

Ожидаемое (среднее) значение трудоёмкости ($t_{ожі}$) вычисляется по следующей формуле:

$$t_{ожi} = \frac{3t_{mini} + 2t_{maxi}}{5}, \quad (2)$$

где $t_{ожі}$ – ожидаемая трудоёмкость выполнения і-ой работы, человеко-день;

t_{mini} – минимально возможная трудоёмкость выполнения заданной і-ой работы (оптимистическая оценка: в предложении наиболее благоприятного стечения обстоятельств), человеко-день;

t_{maxi} – максимально возможная трудоёмкость выполнения заданной і-ой работы (оптимистическая оценка: в предложении наиболее благоприятного стечения обстоятельств), человеко-день [72].

Произведем расчет ожидаемого значения трудоёмкости. Для этого необходимо внести в приложение Г (таблицу Г.1) минимально и максимально возможную трудоёмкость выполнения заданной і-й работы (t_{mini}, t_{maxi}) и на основе этих значений высчитать требуемую величину.

На основе ожидаемой трудоёмкости имеется возможность определить продолжительность каждой работы в рабочих днях $T_р$ с учетом одновременного выполнения работ несколькими исполнителями (3).

$$T_{p_i} = \frac{t_{ожi}}{Ч_i}, \quad (3)$$

где $T_{рі}$ – продолжительность одной работы, рабочий день;

$t_{ожі}$ – ожидаемая трудоёмкость выполнения одной работы, человеко-день;

$Ч_і$ – численность исполнителей, выполняющих одновременно одну и ту же работу на данном этапе, человек.

Результаты нахождения продолжительности каждой работы в рабочих днях ($T_р$) представлены в таблице 5. Информация о количестве исполнителей была взята на основе данных из приложения В.

Таблица 5 – Продолжительность каждой работы

1. № работы	2. Содержание работы	3. Количество исполнителей	4. Продолжительность одной работы ($T_р$), рабочий день
1	Составление и утверждение технического задания	1	1,4
2	Подбор и изучение различных материалов по теме	1	14
3	Проведение исследований	1	7,2
4	Выбор дальнейшего направления исследований	2	0,7
5	Календарное планирование работ по теме	2	0,5
6	Проведение теоретических расчетов и обоснований	1	4,6
7	Проектирование программы для создания имитационных моделей	1	16,8
8	Построение имитационных моделей	1	12,8
9	Проектирование методов балансировки нагрузки и вычислительных узлов	1	12,6

Продолжение таблицы 5

1.	2.	3.	4.
10	Проведение экспериментов на основе созданных имитационных моделей и методов балансировки	1	16,8
11	Сбор и обработка статистических данных на основе проведенных экспериментов	1	5,6
12	Сопоставление результатов экспериментов с теоретическими исследованиями	2	1,2
13	Оценка эффективности полученных результатов	2	1,2
14	Формулировка выводов выпускной квалификационной работы	1	3,8

После определения продолжительности каждой работы с учетом одновременного выполнения работ несколькими исполнителями необходимо провести разработку графика проведения научного исследования. Наиболее подходящим и наглядным для этих целей является построение ленточного графика для проведения научно-исследовательских работ в виде диаграммы Ганта.

Диаграмма Ганта – горизонтальный ленточный график, на котором работы по теме представляются протяженными во времени отрезками, характеризующимися датами начала и окончания выполнения данных работ [72].

Для начала длительность каждого из этапов работ необходимо перевести в календарные дни. Это необходимо для удобства построения диаграммы. Для перевода необходимо воспользоваться следующей формулой:

$$T_{k_i} = T_{p_i} * k_{\text{кал}}, \quad (4)$$

где T_{ki} – продолжительность выполнения i-й работы в календарных днях;

T_{pi} – продолжительность выполнения i-й работы в рабочих днях;

$k_{кал}$ – коэффициент календарности [72].

Для вычисления коэффициента календарности используется следующая формула:

$$k_{\text{кал}} = \frac{T_{\text{кал}}}{T_{\text{кал}} - T_{\text{вых}} - T_{\text{пр}}}, \quad (5)$$

где $k_{кал}$ – коэффициент календарности [72],

$T_{кал}$ – количество календарных дней в году;

$T_{вых}$ – количество выходных дней в году;

$T_{пр}$ – количество праздничных дней в году.

Рассчитанные значения в календарных днях по каждой работе T_{ki} необходимо округлить до целого числа [72].

Произведем расчет коэффициента календарности (5). $T_{кал}$ = 365 дней, $T_{вых}$ = 104 дня, $T_{пр}$ = 14 дней.

$$k_{\text{кал}} = \frac{365}{365 - 104 - 14} = 1{,}478$$

Коэффициент календарности неизменен для всех последующих вычислений.

Все рассчитанные значения по вышеописанным формулам (4, 5) необходимо свести в приложение Д.

В приложении Д можно увидеть, что количество исполнителей не меняется, что говорит о том, что в вариантах исполнения из раздела 1 не

учитывалось изменение числа и добавление новых исполнителей. Состав исполнителей не изменился и состоит из руководителя темы, а также инженера (дипломника). При этом длительность работ в рабочих и календарных днях для разных вариантов исполнения всё равно претерпевает изменения ввиду того, что меняется трудоёмкость работ.

Разница в сложности между вариантами исполнения описана в разделе 6.3. Однако стоит отметить, что наиболее подходящим по соотношению сложности и потраченного времени является вариант 1. Это связано с со следующими условиями, которые позволяет наиболее гибко производить исследования в имитационном эксперименте: созданием вычислительного графа путём задания только количества подзадач; проверкой на правильность генерации графа после создания каждого уровня; автоматическим сохранением графа в выбранный файл; использованием алгоритма обхода графа в ширину для подсчета количества уровней; ручным выбором метода балансировки нагрузки; ручным выбором характеристик производительности (частоты процессорного устройства, количества оперативной памяти, а также качества передачи по каналам связи) и количества одновременно используемых потоков обработки задачи (вычислителей); использованием в качестве механизма получения статистической информации об обработке, как сторонних приложений, так и различных средств языка программирования.

Длительность работ в календарных днях округлена до целого значения.

На основе данных, полученных в ходе составления таблицы 5, строится календарный план-график. График необходимо построить для максимальной по длительности исполнения работы (т.е. варианта 3) в рамках научно-исследовательского проекта (210 дней), с разбивкой по месяцам и декадам (10 дней) за период времени дипломирования. Результаты представлены в приложении Е.

Календарный план-график на основе диаграммы Ганта позволяет наглядно увидеть длительность исполнения, а также распределение ролей

для каждой из работ, что позволяет строго придерживаться намеченного пути в научных исследованиях.

Финансовая часть выпускной квалификационной работы позволила определить возможные альтернативы проведения научных исследований, которые могут быть полезны при разработке имитационной среды моделирования балансировки нагрузки.

Для выявления альтернатив была спроектирована морфологическая матрица, на основе которой было выбрано три варианта исследований. Наиболее подходящим является первый вариант исследований, однако, для построения плана-графика диаграммы Ганта был выбран вариант 3 ввиду того, что данный вариант является наиболее длительным в календарных днях. Это было доказано путем вычислений ожидаемой трудоёмкости работ, коэффициента календарности и длительности работ в рабочих днях.

Третий вариант является наиболее затратным по времени и отображает максимальный срок проведения научно-исследовательских работ.

Таким образом, предлагаемый алгоритм балансировки нагрузки, при дальнейшей модификации и развитии способен стать конкурентоспособным решением в среде горизонтально масштабируемых распределенных вычислений.

Список публикаций студента

1. «Model experiment for load balancing in a distributed computer system» // 2015 International Conference on Mechanical Engineering, Automation and Control Systems (MEACS). – Tomsk, Russia, дата публикации – 01.12.2015. – pp. 1-4.;

2. «Description and development of the means of a model experiment for load balancing in distributed computing systems» // IOP Conference Series: Materials Science and Engineering, Volume 135, Number 1. – Tomsk, Russia. – 1-3 June 2016. – pp. 1-6.;

3. «Scheduling based on a dynamic resource connection» // IOP Conference Series: Materials Science and Engineering, Volume 177, conference 1. – Tomsk, Russia. – 7-1 November 2016. – pp. 1-6.;

4. «Программный эксперимент по балансировке нагрузки в распределенной вычислительной системе» // Восьмая Сибирская конференция по параллельным и высокопроизводительным вычислениям, Томск, 28-30 Октября 2015. - Томск: ТГУ, 2015 - с. 127-131;

5. «Программный эксперимент по балансировке нагрузки в распределенной вычислительной системе» // Восьмая Сибирская конференция по параллельным и высокопроизводительные вычислениям: Программа и тезисы докладов – Томск: Изд-во Том. ун-та, 2015. – с. 44-45;

6. Оценка эффективности распределенной системы управления заявками на основе программного эксперимента // Молодежь и современные информационные технологии: сборник трудов XIII Международной научно-практической конференции студентов, аспирантов и молодых ученых, Томск, 9-13 Ноября 2015. - Томск: ТПУ, 2016 - Т. 1 - с. 38-39;

7. Распределение нагрузки в территориально разнесенном кластере персональных компьютеров // Технологии Microsoft в теории и практике программирования: сборник трудов XIII Всероссийской научно-

практической конференции студентов, аспирантов и молодых ученых, Томск, 22-23 Апреля 2016. - Томск: ТПУ, 2016 - с. 170-172;

8. «Описание и разработка средств модельного эксперимента по балансировке нагрузки в распределенных вычислительных системах» // Физико-технические проблемы в науке, промышленности и медицине: сборник научных трудов VIII Международной научно-практической конференции. – Томск, 2016. – с. 249-250.

9. Разработка метода динамического подключения ресурсов // Информационные технологии в науке, управлении, социальной сфере и медицине: сборник научных трудов III Международной конференции, Томск, 23-26 Мая 2016. - Томск: ТПУ, 2016 - Т. 1 - с. 726-728.

10. «Диспетчеризация на основе динамического подключения ресурсов», Сборник трудов XIV Международной научно-практической конференции студентов, аспирантов и молодых ученых «Молодежь и современные информационные технологии». – Томск: ТПУ, 2016 - Т. 1 - с. 74-76.