ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Программный проект на тему:

Разработка симулятора вычислительного кластера

Выполнил студент:

группы БПМИ206, 4 курса Макогон Артём Аркадьевич

Принял руководитель ВКР:

Сухорослов Олег Викторович
Доцент, к.т.н.
Факультет компьютерных наук НИУ ВШЭ

Москва 2024

Содержание

Аннотация

Алгоритмы планирования задач на вычислительных кластерах являются предметом активных исследований. Проведение таких исследований на настоящем кластере долго и дорого, поэтому для сравнения различных гипотез используются симуляторы. Качество получаемых результатов во многом зависит от точности используемой модели. Большинство современных симуляторов используют простые модели, которых не всегда достаточно. Эта работа сосредоточена на разработке симулятора вычислительного кластера на базе фреймворка `DSLab`, который позволяет очень точно и гибко моделировать сценарии вычислительной нагрузки с использованием примитивов асинхронного программирования. `DSLab` предоставляет массу возможностей как для написания новых алгоритмов, так и для использования готовых модулей различных компонент распределенной системы. Разработанный симулятор получил основные преимущества `DSLab` и позволяет моделировать работу алгоритмов упаковки заданий на кластер и алгоритмов справедливого распределения ресурсов.

Ключевые слова

Распределенные системы, вычислительный кластер, алгоритмы планирования, управление ресурсами, имитационное моделирование, симулятор

Annotation

Algorithms for scheduling tasks on compute clusters are the subject of active research. It is quite expensive to conduct such studies using real cluster, so simulators are used to compare different hypotheses. The quality of the results obtained largely depends on the accuracy of the model used. Most modern simulators use simple compute models, which are not always enough. Our work is focused on developing a simulator of a compute cluster based on the `DSLab` framework, which will allow very accurate and flexible modeling of computational load scenarios using asynchronous programming primitives. `DSLab` provides a lot of opportunities both for writing new algorithms and using ready-made modules of various distributed system components. Thus, the new simulator has received the main advantages of `DSLab` and allows modeling resource packaging algorithms and fair resource sharing algorithms in complex configurations.

Keywords

Distributed systems, compute cluster, scheduling algorithms, resource management, simulation, simulator

1 Введение

Вычислительные кластеры, состоящие из набора серверов с различными ресурсами (процессор, память, диск), широко применяются для проведения сложных расчетов и обработки больших объемов данных в науке и бизнесе. На большом промышленном кластере одновременно могут выполняться тысячи заданий (batch jobs) различных пользователей, а часть заданий может находиться в очереди в ожидании выделения ресурсов. Запуском заданий и распределением ресурсов между ними управляет менеджер кластера или планировщик. В основе работы менеджера кластера лежат алгоритмы планирования задания (job scheduling), основной целью которых является максимально эффективное использование ресурсов кластера. Данные алгоритмы являются предметом активных исследований. Проводить такие исследования на реальном кластере долго и дорого, поэтому необходима компьютерная модель (симулятор), позволяющая быстро проверить гипотезу или провести сравнительное тестирование разных алгоритмов на некоторой истории нагрузки кластера. Подобный симулятор также может использоваться в учебном процессе для знакомства студентов с такими системами и возникающими в них задачами.

1.1 Актуальность и значимость

Алгоритм планирования заданий – ключевая часть архитектуры всех вычислительных кластеров. Поиск оптимальных алгоритмов очень актуален, поскольку повышение утилизации имеющихся ресурсов позволит существенно снизить затраты на оборудование кластера и ускорить его работу. Существующие симуляторы в основном предлагают простые модели задач или углубляются в конкретную область (например, глубинное обучение).

Новый симулятор, разработанный на базе фреймворка `DSLab` позволит исследователям подбирать алгоритмы под конкретные сценарии нагрузки, используемые в их кластере. Таким образом, симулятор позволит более точно предсказывать поведение настоящего кластера.

1.2 Цели и задачи ВКР

Целью ВКР является разработка симулятора вычислительного кластера на базе фреймворка `DSLab`.

Для достижения этой цели были поставлены следующие задачи:

1. Изучить литературу по теме и существующие симуляторы, подготовить обзор с анализом их преимуществ и недостатков.
2. Поэтапно реализовать компоненты симулятора, покрыть их тестами и снабдить комментариями.
3. Подготовить и провести эксперименты, демонстрирующие работоспособность симулятора и достижение всех требований.
4. Подготовить документацию пользователя.

1.3 Основные результаты работы

В результате ВКР был разработан эффективный и многофункциональный симулятор вычислительного кластера на базе фреймворка `DSLab`, были реализованны несколько популярных алгоритмов планирования задач и проведены тесты на данных трейсов различных компаний, а также синтетических данных.

1.4 Структура работы

Дальнейшая структура работы следующая:

- Глава 2 содержит обзор литературы и существующих симуляторов.
- Глава 3 описывает фреймворк `DSLab` и его асинхронное расширение, на базе которого был разработан симулятор.
- Глава 4 описывает реализацию симулятора, основные компоненты и используемые модели.
- Глава 5 содержит описание проведенных экспериментов на разработанном симуляторе и их анализ.
- Глава 6 подводит итоги работы и предлагает дальнейшие направления разработки и улучшения симулятора.

2 Обзор литературы

2.1 Вычислительная модель в симуляции

Для моделирования работы кластера и алгоритмов планирования требуется модель рабочей нагрузки. Самая распространенная модель описывает рабочую нагрузку как набор заданий (jobs), состоящих из задач (tasks), которые непосредственно исполняются на кластере и потребляют ресурсы. Структура этих заданий может быть нескольких видов:

- Фиксированное задание (rigid job) – заранее известны требования к ресурсам задания.
- Масштабируемое задание (moldable job) – существует набор конфигураций ресурсов, на которых можно запустить задание. Конфигурация выбирается на этапе планирования и не изменяется по ходу исполнения.
- Гибкое задание (malleable job) – задание, которое может адаптивно менять количество используемых ресурсов в процессе выполнения.

Требования к планированию заданий делятся на две категории:

- Раздельное планирование. Задачи (составные части задания) могут планироваться на кластере по мере освобождения ресурсов, при этом между ними могут быть зависимости. Самый распространенный пример такой модели планирования – концепция MapReduce[4].
- Комплектное планирование (gang scheduling). В такой модели задание воспринимается как параллельная программа, все задачи которой нужно одновременно запустить.

Среди распространенных форматов описания нагрузки на кластер можно выделить Standard Workload Format(SWF)[23]. Этот формат широко используется в литературе и поддерживается многими симуляторами. Однако, SWF предоставляет довольно мало информации о рабочей нагрузке. Ключевая информация задании включает время прибытия, время выполнения, приоритет и потребность в ресурсах. Единственное предположение, которое мы можем сделать о рабочей нагрузке заключается в том, что задание потребляет все выделенные ресурсы во время выполнения. Наиболее популярные трейсы рабочей нагрузки представлены в аналогичном формате (например, Google trace[25]).

Однако, существует и более сложный подход к моделированию задач на кластере, при котором время выполнения задачи рассчитывается на основе структуры рабочей нагрузки и доступных ресурсов. Основное преимущество этого подхода заключается в том, что больше деталей задачи могут быть учтены и использованы при планировании. Кроме того, появляется возможность с большей точностью рассчитать загрузку кластера и поведение его компонент. Основной недостаток – для такого моделирования довольно сложно найти данные из открытых источников, поскольку информация о детальной структуре задач обычно конфиденциальна и не публикуется.

2.2 Аналоги

Существующие симуляторы относятся к одной из двух категорий: симуляторы, созданные с нуля, и основанные на готовой платформе. Другой способ группировки симуляторов – по типу рабочей нагрузки, которую они поддерживают: симуляторы, которые работают с SWF или аналогичными форматами, и поддерживающие сложное описание рабочей нагрузки.

2.2.1 AccaSim

Характерным примером такого симулятора «с нуля» является `AccaSim`[13]. Он основан на дискретно-событийном моделировании и содержит предопределенные алгоритмы планирования. Он довольно прост в использовании, но не поддерживает сложные конфигурации. На вход принимаются только файлы в формате SWF. В отличие от `AccaSim`, наш симулятор сможет моделировать произвольные сценарии рабочей нагрузки, которые будут определены пользователями. Кроме того, наш симулятор является модульным, поскольку основан на `DSLab`. Пользователи смогут подключать модули, написанные другими исследователями в рамках платформы `DSLab`, и использовать их в своих экспериментах. Например, `DSLab` предоставляет модели сети, которые могут точно симулировать различные топологии. Среди преимуществ этого симулятора можно отметить, что он поддерживает инкрементальное чтение трейса нагрузки, что делает возможным проведение больших симуляций.

2.2.2 BatSim

Симулятор `BatSim` основан на `SimGrid`[3] и позволяет разработчикам интегрировать алгоритмы на нескольких языках на основе inter-process communication. Подробное сравнение

SimGrid и DSLab можно найти в документации по DSLab[8]. Ключевое ограничение BatSim – из возможных ресурсов кластера он поддерживает моделирование только «вычисления», «передачи данных» и «операции с распределенной файловой системой». BatSim предоставляет интерфейс для настройки пользовательских типов рабочей нагрузки, но они ограничены несколькими заданными паттернами и возможными последовательными комбинациями из них (см. рис. 2.1).

```
"jobs": [
  {"id":"simple", "subtime":1, "res": 4, "profile": "simple"},
],
"profiles": {
  /* workload profile definition */
  "simple": {
    "type": "parallel",
    "cpu": [5e6,  0,  0,  0],
    "com": [5e6,  0,  0,  0,
            5e6,5e6,  0,  0,
            5e6,5e6,  0,  0,
            5e6,5e6,5e6,  0]
  },
}
```

Рис. 2.1: Пример описания нагрузки в BatSim

Также из рис. 2.1 можно увидеть, что потребность в ресурсах задачи выражается в количестве выделенных «ресурсов» (одно число – количество серверов), и задача размещается на них целиком. Таким образом, существенно ограничивается моделирование задач с разным профилем требований к ресурсам (задача упаковки). Стоит отметить, что каждый сервер имеет свою вычислительную мощность, а также каким-то образом размещен в топологии сети, поэтому проблема размещения задач оптимальным образом на кластере не сводится к тривиальной. Необходимость учитывать топологию сети в такой модели – сложная проблема планирования. Однако, проблема упаковки задач на несколько ресурсов в такой модели не рассматривается.

Наш симулятор расширит подход BatSim для определения произвольной рабочей нагрузки, но при этом предоставит больше возможностей для управления ресурсами (процессор, память, диск), а также предоставит еще больше возможностей для описания сценария нагрузки с использованием примитивов асинхронного программирования.

2.2.3 IRMaSim

Дальнейшим развитием `BatSim` стал симулятор `IRMaSim`[16]. В нем отмечаются недостатки `BatSim` в точности моделирования вычислений и предоставляются намного более точные модели процессоров и добавляются различные модели оперативной памяти. В этом симуляторе возможно точно указать тип оперативной памяти и ее скорость, и все это будет учитываться в симуляции. Также этот симулятор поддерживает сценарии работы с задачами глубинного обучения (Deep Reinforcement Learning). Однако, ключевое ограничение остается то же – потребность задач в ресурсах указывается одним числом – количеством «ресурсов» (серверов). По аналогии с `BatSim`, этот симулятор фокусируется на потреблении энергии в процессе работы кластера и позволяет разрабатывать алгоритмы планирования, оптимизирующие энергопотребление.

2.2.4 ElastiSim

Симулятор `ElastiSim`[18] также основан на `SimGrid` и ориентирован на гибкие рабочие нагрузки (malleable workloads). Это позволяет моделировать динамические изменения количества ресурсов, выделяемых на исполнение задач. Одна из важных особенностей `ElastiSim` – высокая точность моделирования задач глубинного обучения, для которых он и предназначен. Кроме этого, в `ElastiSim` поддерживается моделирование передачи данных (I/O) по сети. Авторы обращают особое внимание на то, что операции I/O могут стать узким местом, поскольку объем передаваемых данных постоянно увеличивается. Выполнение каждой задачи состоит из нескольких этапов и может быть настроено с помощью файла JSON путем объединения предварительно определенного набора операций: CPU, GPU, операций I/O и дисковых операций. Это самый продвинутый симулятор из всех рассмотренных источников. Мы будем стремиться предоставить еще больше возможностей для подробного описания выполняемых задач, однако наш симулятор будет поддерживать более распространенную модель вычислений, при которой каждой задаче выделяется фиксированная аллокация ресурсов на время выполнения.

2.3 Выводы

Рассмотренные симуляторы решают разные задачи симуляции: симуляция упаковки задач на кластер в условиях нескольких ресурсов (`AccaSim`), и точная симуляция вычислительного

процесса (`BatSim`, `IRMaSim`, `ElastiSim`). При этом ни в одном симуляторе эти задачи не пересекаются. Наш симулятор разработан в более общей модели, что позволяет поддержать задачу упаковки и точное моделирование вычислений одновременно. Более того, одной из тем исследований планирования задач на кластерах является справедливость (в условиях, когда кластер делится между несколькими пользователями). Задача справедливости распределения ресурсов не рассматривалась в приведенных симуляторах, однако наш симулятор поддерживает и эту возможность. Наглядное сравнение симуляторов можно увидеть в таблице 2.1. Разработанный симулятор называется `DSLab-CS` (`DSLab` Cluster Simulator).

Критерий	`DSLab-CS`	`AccaSim`	`BatSim`	`IRMaSim`	`ElastiSim`
Платформа симулятора	`DSLab`	`Custom`	`SimGrid`	`SimGrid`	`SimGrid`
Язык программирования	`Rust`	`Python`	`C++`	`C++`	`C++`
Поддержка других языков программирования для алгоритмов пользователей	–	–	+	+	+
Модели заданий	rigid	rigid	rigid	rigid	rigid moldable malleable
Поддержка форматов трейсов	SWF Custom Alibaba trace Google trace	SWF	SWF Custom	SWF Custom	Custom
Поддержка итеративного чтения трейса нагрузки для длительных симуляций	+	+	–	–	–
Поддержка сложных моделей сети	+	–	+	+	+
Поддержка разных моделей оперативной памяти	–	–	–	+	–
Поддержка модели GPU и операций глубинного обучения	–	–	–	+	+
Поддержка моделирования упаковки задач на кластер с несколькими ресурсами	+	+	–	–	–
Поддержка моделирования упаковки задач на кластер с учетом топологии сети	+	–	+	+	+
Поддержка моделирования справедливого распределения ресурсов между пользователями	+	–	–	–	–
Поддержка описания сложных профилей нагрузки через файлы конфигурации	+	–	+	+	+

Таблица 2.1: Сравнение симуляторов

3 Фреймворк DSLab

3.1 Архитектура DSLab

В качестве платформы для реализации симулятора выбран `DSLab` – модульный фреймворк для моделирования и тестирования распределенных систем. Модули можно разделить на 3 группы: базовые, универсальные и специализированные (см рисунок 3.1). За основу взят подход дискретно-событийного моделирования, при котором компоненты симуляции обмениваются событиями через базовый низкоуровневый модуль `dslab-core`. Соответственно, пользовательская логика каждого компонента состоит из обработки входящих событий от других компонент и отправки новых. Универсальные модули могут использовать исследователями соответствующих предметных областей, например, разработанный симулятор вычислительного кластера использует готовые модули вычислений, сети и диска (подробнее об этом в главе 4.2).

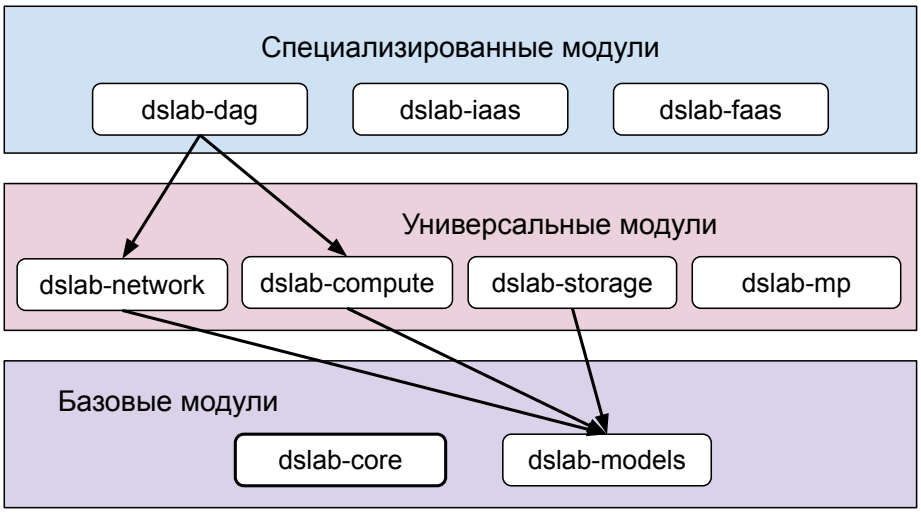


Рис. 3.1: Архитектура `DSLab`

Описание архитектуры основано на официальной документации [5]. Реализация всех модулей опубликована в GitHub-репозитории[11]. Полное сравнение с другими симуляторами можно также найти в документации[8].

3.2 Асинхронное управление событиями

В предыдущей работе была добавлена возможность управлять событиями `DSLab` асинхронно[17], что существенно расширило возможности написания компонент и сделало возможным

ключевую особенность нового симулятора – описание профиля нагрузки задач через асинхронные примитивы языка `Rust` (подробнее об этом в главе 4.3). Ключевая особенность такого расширения – возможность спрятать от пользователя явную работу с событиями, предоставив ему высокоуровневые асинхронные функции, в реализации которых зашито асинхронное ожидание внутренних событий. Например, при работе с записью на диск можно предоставить пользователю асинхронную функцию (см рисунок 3.2).

```
pub async fn write_data(&self, size: u64) -> Result<(), String> {
  let req_id = self.disk.write(size, /*requester=*/self.ctx.id());

  select! {
    _ = self.ctx.recv_event_by_key::<DataWriteCompleted>(req_id).fuse() => {
        Result::Ok(())
    }
    failed = self.ctx.recv_event_by_key::<DataWriteFailed>(req_id).fuse() => {
        Result::Err(failed.data.error)
    }
  }
}
```

Рис. 3.2: Пример асинхронной функции записи данных на диск.

В этом примере используется ожидание события по ключу: исполнение функции прерывается до того момента, пока компонент (в данном случае – вычислительный сервер, на котором запущена задача) не получит событие от собственного диска о том, что запись завершилась (либо успешно, либо с ошибкой – для альтернативного ожидания используется макрос `select!` из библиотеки `futures`[21]), после этого ядро симуляции продолжит исполнение и результат вернется пользователю.

Таким образом, пользователю не нужно будет самостоятельно следить за событиями компонент и можно будет разделить инструкции выполнения задач от конкретных компонент, предоставив лишь «сценарий работы» в виде асинхронных функций.

Точно таким же образом реализованы и другие примитивы: вычисления и работа с сетью.

4 Реализация симулятора

4.1 Архитектура симулятора

По аналогии с самим фреймворком `DSLab` симулятор модульный, его архитектура представлена на рисунке 4.1.

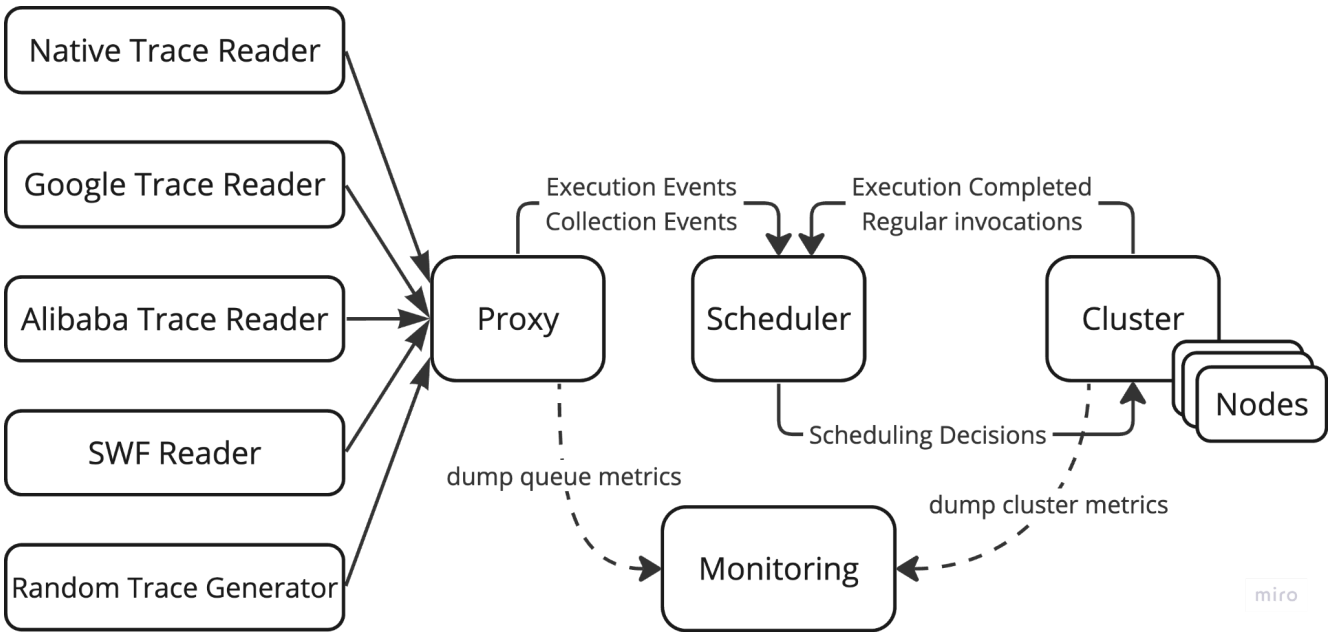


Рис. 4.1: Архитектура симулятора

Симулятор может работать в нескольких режимах, принимая на вход несколько видов входных данных. Это может быть как готовый трейс нагрузки из открытых источников (например, Google Trace), так и пользовательский ввод задач, описанный в виде `YAML`-файла (подробнее в разделе 4.5.1).

Собирая информацию с компонентов `Proxy` и `Cluster` можно получить статистику о работе кластера и алгоритмов планирования. Например, насколько загружен кластер, насколько хорошо утилизируются его ресурсы. Такие выходные данные могут быть использованы для анализа работы алгоритмов и сравнения их эффективности. За это отвечает отдельный компонент – `Monitoring`(подробнее в разделе 4.6).

4.2 Модель кластера

Вычислительный кластер – это набор отдельных серверов, которые могут быть связаны между собой моделируемой сетью (подробнее в разделе 4.2.2). В симуляторе используются готовые модули `DSLab`, из которых собирается каждый сервер (см. таблицу 4.1).

Название модуля	**Используемый модуль**	**Обязательный**
`compute`	`dslab_compute::multicore::Compute`	да
`network`	`dslab_network::Network`	нет
`disk`	`dslab_storage::disk::Disk`	нет

Таблица 4.1: Модули каждого сервера в кластере

Все эти модули конфигурируются при создании кластера. В случае, если хост запланировано выходит из строя или начинает работу после начала симуляции, его модули все равно создаются сразу как компоненты симуляции `DSLab`. Это необходимо, поскольку каждый сервер и каждый модуль сервера является обработчиком событий в симуляции и должен быть соответствующим образом зарегистрирован.

4.2.1 Модуль вычислений

Для вычислений используется модуль `Compute` с фиксированными слотами – «аллокациями», которые состоят из некоторого количества ядер процессора и объема выделенной оперативной памяти. В текущей используемой модели аллокации не могут пересекаться, однако могут быть заняты не полностью, что скажется на утилизации кластера. Т.е. на данном этапе планировщик не имеет возможность выделить больше ресурсов чем есть на сервере в надежде, что запущенные задачи используют меньше, чем просят.

Поскольку некоторые трейсы нагрузки не содержат точную информацию о конфигурации кластера, единицы измерения «ядер процессора» и размера оперативной памяти могут быть относительны. Например, в Google Trace[25] размер оперативной памяти и количества ядер процессора указаны как доля от максимального значения в кластере. Аналогичным образом нормированы и требования задач к ресурсам. Такой политики компании придерживаются с целью не раскрывать реальные конфигурации своих серверов и требования к задачам. Поскольку модуль `Compute` принимает на вход целые числа (количество ядер и объем памяти), то при создании кластера в симуляции данные в трейсе доли умножаются на большие константы таким образом, чтобы получились целые числа и погрешность была небольшой (очевидно, что при этом теряются единицы измерения). Все константы задаются пользователем при конфигурации симуляции.

Подробнее описание модуля `Compute` и примеры использования можно посмотреть в официальной документации `DSLab`[9].

4.2.2 Модуль сети

Модуль сети `Network` содержит несколько моделей на выбор, которыми можно воспользоваться в симуляции:

1. `ConstantBandwidthNetworkModel` – самая простая модель, которая предполагает, что все

сервера в кластере соединены сетью с постоянной пропускной способностью. Нагрузка сети одними компонентами никак не влияет на другие. Подходит для простой и быстрой симуляции сетевого взаимодействия с потерей в точности.

2. `SharedBandwidthNetworkModel` – модель, предполагающая наличие одного сетевого канала, общего для всех участников сети. Может быть достаточно точной в ситуации, когда все компоненты сети подключены к одному центральному коммутатору и пропускная способность сети много превосходит пропускную способность центрального коммутатора, являющегося узким местом. Однако, в реальности для такого случая можно применить модель `Star-Topology` и получить существенно большую точность с небольшим штрафом в производительности. Поэтому и эту модель используют с целью более быстрого моделирования сети в ущерб точности.

3. `TopologyAwareNetworkModel` – гибко настраиваемая и достаточно точная модель сети с возможностью задания топологии. В симуляторе пользователь может самостоятельно настроить и передать симулятору сеть, которая наиболее точно отражает желаемую ситуацию. Однако, по умолчанию предоставляется вариант `Fat-Tree Topology`, который на практике чаще всего используется в датацентрах и кластерах, поскольку является эффективным и отказоустойчивым[19]. Схематично топология сети «`Fat-Tree`» представлена на рисунке 4.2. Серверы объединяются в стойки (L1 Switch), которые связаны между собой коммутаторами (L2 Switch). Все пропускные способности и задержки между узлами, а также количество серверов в стойках и количество верхнеуровневых коммутаторов задаются пользователем при конфигурации симуляции.

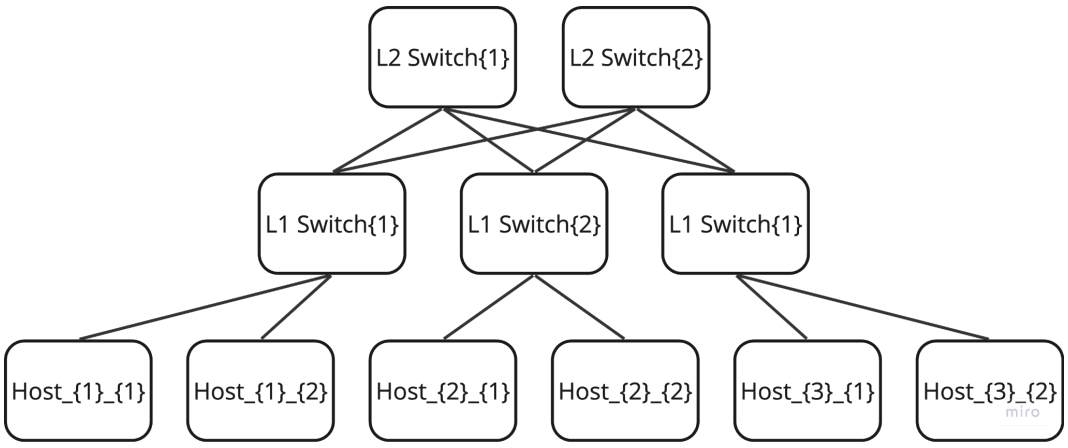


Рис. 4.2: Пример топологии сети «Fat-Tree» на 6 серверов по 2 сервера в стойке

Подробнее описание модуля `Network` и примеры использования можно посмотреть в официальной документации `DSLab`[10].

4.2.3 Модуль хранилища

Модуль диска используется для симуляции операций чтения-записи в хранилище. Модель предоставляет возможность гибко настроить деградацию производительности диска в зависимости от количества параллельных операций (особенно актуально для HDD операций). Также модель предоставляет возможность реализовать пользовательский планировщик дисковых операций, но по умолчанию используется классический `«FIFO»`-планировщик (First-In-First-Out).

Подробнее описание модуля `Storage` и примеры использования можно посмотреть в официальной документации `DSLab`[12].

4.3 Вычислительная модель

4.3.1 Деление на Collection и Execution

Поскольку симулятор может поддерживать несколько разных вычислительных моделей, для описания нагрузки был выбран обобщенный вариант: пользователь описывает задачи (`Execution`), которые непосредственно запускаются на кластере и потребляют ресурсы, и эти задачи могут быть связаны между собой заданием (`Collection`). Термин `Collection` происходит из аналогичного определения задания в последней версии Google Trace[25]. `Execution` же является минимальной единицей планирования и может из себя представлять как простые вычисления, так и сложную программу, которой для запуска нужно одновременно выделить несколько серверов. Структуры данных представлены в таблицах 4.2 и 4.3.

Описание требуемых ресурсов задачи состоит из требования к количеству хостов, а также опционально к количеству ядер процессора и памяти на хостах. Таким образом, сохраняется возможность поддержать модель планирования аналогичной `BatSim`, `IRMaSim` и `ElastiSim`, но и не упускается возможность для моделирования упаковки задач на кластерах (при указывании количества хостов больше одного, есть возможность указать требования к ресурсам каждого хоста отдельно или всем сразу).

4.3.2 ExecutionProfile

Задачи, которые непосредственно выполняются на кластере, представляются в виде объектов, для которых реализован интерфейс `ExecutionProfile` (см. рисунок 4.3).

Поле	Тип	Описание
id	`u64`	Уникальный идентификатор задачи на всю симуляцию
collection_id	`Option<u64>`	Индекс задания
execution_index	`Option<u64>`	Индекс задачи внутри задания
time	`f64`	Время, когда задача становится доступной для планирования
schedule_after	`Option<f64>`	Время, когда задача становится доступной для исполнения на кластере. Поле введено для поддержки совместимости с трейсом нагрузки из Google.
resources	`ResourceRequirements`	Структура требуемых ресурсов для исполнения задачи
profile	`Rc<dyn ExecutionProfile>`	Профиль нагрузки (см секцию 4.3.2)
wall_time_limit	`Option<f64>`	Максимальное время на выполнение задачи, после которой она автоматически будет отменена
priority	`Option<u64>`	Приоритет задачи
start_after	`Option<Vec<u64»`	Список индексов задач внутри задания, после которых можно начать выполнение этой задачи (должен быть указан `collection_id`)

Таблица 4.2: Описание структуры данных `ExecutionRequest`

Поле	Тип	Описание
id	`u64`	Уникальный идентификатор задания на всю симуляцию.
time	`f64`	Время, когда задание приходит на планировщик.
user	`Option<String>`	Пользователь, который запускает задание. Используется для тестирования алгоритмов справедливого разделения кластера.
priority	`Option<u64>`	Приоритет приоритет задания. Если у задачи указан собственный приоритет, то предполагается, что он переопределяет данный приоритет.
start_after_collection_ids	`Option<Vec<u64»`	Список идентификаторов заданий, после которых можно начать выполнение задач этого задания.

Таблица 4.3: Описание структуры данных `CollectionRequest`

```
#[async_trait(?Send)]
pub trait ExecutionProfile {
  async fn run(self: Rc<Self>, processes: &Vec<HostProcessInstance>);
}
```

Рис. 4.3: Интерфейс `ExecutionProfile`

Метод `run` может быть реализован самим пользователем, но также могут использоваться заготовки для простых сценариев работы. Такая задача получает в свое «владение» от планировщика набор выделенных процессов на узлах кластера и выполняет некоторую работу. Сама работа описывается в терминах асинхронного программирования на Rust и использует предоставленный интерфейс взаимодействия с кластером (рис. 4.4).

```
async fn sleep(&self, time: f64);
async fn run_compute_work(&self, compute_work: f64);
async fn transfer_data_process(&self, size: f64, dst_process: ProcessId);
async fn transfer_data_component(&self, size: f64, dst_component: Id);
async fn write_data(&self, size: u64);
async fn read_data(&self, size: u64);
async fn deallocate(&self);
```

Рис. 4.4: Интерфейс взаимодействия процессов с кластером

Поскольку симулятор поддерживает несколько моделей вычислений, то интерфейс `ExecutionProfile` принимает на вход несколько процессов (гарантированно хотя бы один). Количество процессов соответствует количеству серверов, на которых одновременно запущена задача (в соответствии с комплектной моделью планирования). Каждому процессу выделяется аллокация ресурсов на сервере в соответствии с требованиями, и каждый процесс получает асинхронный интерфейс 4.4 общения с кластером. Если по ходу выполнения задачи какой-то процесс заканчивает работу раньше других, его можно принудительно завершить через функцию `deallocate`, освободив ресурсы на сервере. После этого процессом пользоваться будет нельзя (вызов любой другой функции приведет к панике программы).

Для простых ситуаций (например, симуляции нагрузки по Google Trace) используются тривиальные имплементации `ExecutionProfile`, которые рассчитаны на выполнение на одном сервере (получают на вход ровно один процесс который занимает данные ресурсы на 100% и спит определенное время).

Для более сложного моделирования работы задачи можно использовать весь спектр асинхронных инструментов из библиотеки `futures`[21]. Например, паттерн работы «Master-Workers» можно реализовать как показано на примере 4.5.

Поскольку текущая версия компилятора `Rust` не поддерживает динамическую диспетчеризацию асинхронных методов, то при реализации интерфейса `ExecutionProfile` необходимо использовать стороннюю библиотеку `async-trait`[20]. Удобству использования это не мешает, т.к. все использование библиотеки сводится к добавлению соответствующего макроса к реализации интерфейса.

```rust
pub struct MasterWorkersProfile {
  pub master_compute_work: f64,
  pub worker_compute_work: f64,
  pub data_transfer_size: f64,
}

#[async_trait(?Send)]
impl ExecutionProfile for MasterWorkersProfile {
  async fn run(self: Rc<Self>, processes: &Vec<HostProcessInstance>) {
    let master = &processes[0];
    let workers = &processes[1..];

    futures::future::join_all(workers.iter().map(|p| async {
      master.transfer_data_process(self.data_transfer_size, p.id).await;
      p.run_compute(self.worker_compute_work).await;
    }))
    .await;

    master.run_compute(self.master_compute_work).await;
  }
}
```

Рис. 4.5: Пример реализации интерфейса `ExecutionProfile`

4.3.3 Описание нагрузки через входной файл

Пользователь может самостоятельно реализовывать свои паттерны нагрузки на кластер, тем самым получая высокую степень точности симуляции. Для использования этих профилей необходимо их зарегистрировать в симуляции перед ее запуском. Тогда можно будет подавать на вход задачи, использующие пользовательские профили. Например, после регистрации профиля `MasterWorkersProfile` (код 4.5) можно описать задачу (`ExecutionRequest`) следующим образом с помощью YAML-файла:

```yaml
- submit_time: 10.0
  resources:
    nodes_count: 4
    cpu_per_node: 2
    memory_per_node: 4
  profile:
    type: master-workers
    args:
      master_compute_work: 100.
      worker_compute_work: 2000.
      data_transfer_size: 100.
```

Рис. 4.6: Пример описания задачи с использованием профиля `MasterWorkersProfile`

Во время симуляции будет создан объект задачи с переданными параметрами (из секции `args`) и передан в планировщик для размещения на кластере. Как только планировщик найдет для задачи место, будет выполнен метод `run` из реализации `MasterWorkersProfile`

(см. рисунок 4.5). Поскольку задача будет запущена на 4 серверах одновременно, то в метод `run` будут переданы 4 процесса. Дальше, следуя коду реализации 4.5, первый процесс станет мастером, а остальные 3 – рабочими.

4.3.4 Декларативное создание профилей нагрузки через файл

Помимо реализации интерфейса `ExecutionProfile`, есть и другой способ описания профилей нагрузки – через YAML-файл. Такой способ описания позволяет проще создавать новые профили нагрузки без необходимости писать код, но с меньшими функциональными возможностями. Пользователю предоставляется два основных комбинатора профилей: параллельный и последовательный, которые могут быть вложены друг в друга. Пример описания профиля нагрузки представлен на рисунке 4.7. В этом профиле сначала создаются базовые профили нагрузки на вычисления, сеть и диск (рис. 4.7a). По сути это введение коротких имен для типовых сценариев нагрузки. В комбинаторах можно использовать как профили с коротким именем, так и профили с аргументами (в том числе пользовательские). Затем создается сложный профиль (рис. 4.7b) с таким сценарием работы:

1. На всех процессах происходит чтение данных с диска, затем все процессы делают простые вычисления (`compute-simple`).
2. Все процессы параллельно делают сложное вычисление и общаются по сети в соответствии с профилем `net-all-to-all-simple`.
3. Все процессы записывают результаты на диск.
4. Снова параллельные простые вычисления на всех процессах и общение по сети.
5. В конце запись данных на диск всеми процессами.

Вся последовательность повторяется дважды.

Можно видеть, что вложенные комбинаторы позволяют создавать сложные сценарии. По умолчанию предоставляются базовые профили, описанные в таблице 4.4. Эти профили покрывают функциональность профилей из `BatSim`[2] и позволяют создавать разнообразные сценарии нагрузки на кластер, а с помощью комбинаторов можно сделать более сложные вложенные профили. Более того, к такому декларативному описанию задач можно добавить и собственные профили, которые реализуют интерфейс `ExecutionProfile` (см. секцию 4.3.2). Определенные профили могут использовать в качестве зависимостей только те профили,

```yaml
compute-simple:
  type: compute-homogenous
  args:
    compute_work: 1000.0
compute-hard:
  type: compute-homogenous
  args:
    compute_work: 2000.0
net-all-to-all-simple:
  type: communication-homogenous
  args:
    size: 500.0
disk-simple-read:
  type: disk-read
  args:
    size: 1000
disk-simple-write-part:
  type: disk-write
  args:
    size: 600
```

(a) Пример описания базовых профилей

```yaml
complex_execution_profile:
  type: sequence
  args:
    repeat: 2
    profiles:
      - disk-simple-read
      - compute-simple
      - parallel
        args:
          profiles:
            - compute-hard
            - net-all-to-all-simple
      - disk-simple-write-part
      - parallel
        args:
          profiles:
            - compute-simple
            - net-all-to-all-simple
      - disk-simple-write-part
```

(b) Пример описания профиля через комбинаторы

Рис. 4.7: Пример описания профилей нагрузки через YAML-файл

которые была определены выше по файлу, это исключает возможность возникновения циклических зависимостей.

4.4 Модель планирования заданий

Симулятор поддерживает несколько основных моделей планирования:

1. Планировщик принимает решение о планировании на основе всей информации (выбирает подходящую задачу и подходящий хост). В таком случае планировщик может вызываться периодически и по событиям (например, по получению новых задач и завершению прошлых). В такой модели у планировщика есть набор задач, которые в данный момент наиболее приоритетны (обычно это некоторая очередь), и планировщик выбирает под эти задачи наиболее подходящие серверы (или части серверов) кластера. Такая модель используется в одном из наиболее популярных планировщиков для кластеров `Slurm`(Simple Linux Utility for Resource Management)[22]. В этой модели под задание (job) с помощью отдельного плагина `Node Selection` выбирается набор серверов, на которые впоследствии распределяются задачи (tasks).

2. Вычислительные серверы периодически сообщают планировщику о своих свободных ресурсах, и планировщик принимает решение о подходящих задачах (которые могут

Профиль	Аргументы	Описание
compute-homogenous	`compute_work: f64`	Выполнить `compute_work` работы на всех переданных процессах.
compute-vector	`compute_work: Vec<f64>`	Выполнить вычисления на процессах согласно вектору `compute_work`, например: [10, 20] – первый процесс выполняет работу 10, а второй – 20.
communication-homogenous	`size: f64`	Передать `size` данных между всеми переданными процессами.
communication-matrix	`matrix: Vec<Vec<f64»`	Передать данные между процессами согласно матрице `matrix`, например: [0, 20, 10, 0] – первый процесс передает второму данные размера 20, а второй первому – 10. Аналогично с определением матрицы передачи данных из `BatSim`[2].

communication-src-dst	`src: Vec<usize>`
	`dst: Vec<usize>`
	`size: f64`
		Передать `size` данных от процессов с индексами из `src` к процессам с индексами из `dst`.

communication-external	`processes: Vec<usize>`
	`input_size: f64`
	`output_size: f64`
	`component_name: String`
		Передать данные между процессами и другими компонентами сети, например, между процессами и внешним хранилищем. Для этого внешнее хранилище должно быть предварительно зарегистрировано в сети и настроено.

disk-read	`processes: Vec<usize>` `size: f64`	Прочитать `size` данных с диска у данных процессов. Если процессы не указаны, чтение происходит у всех.
disk-write	`processes: Vec<usize>` `size: f64`	Записать `size` данных на диск у данных процессов. Если процессы не указаны, запись на диск происходит у всех.

Таблица 4.4: Описание предоставляемых по умолчанию профилей нагрузки

аналогичным образом браться из очереди) конкретно под сервер, который сделал запрос. Также серверы информируют планировщик об изменении свободных ресурсов в связи с завершением текущих задач. Такая модель используется в планировщике кластера YT в Яндексе[26].

Благодаря поддержке обеих моделей симулятор способен воспроизводить работу разных планировщиков, что делает его более универсальным. Однако, из-за этого размывается четкое разделение рабочей нагрузки на задание (job) и задачи (tasks). В такой ситуации получается, что `Execution` (см. раздел 4.3.1) может выступать как простейшая задача продолжительностью несколько минут (из модели планировщика `YT`, `Google` и др.), так

и сложная много-серверная программа, внутри которой есть своя собственная логика разделения ресурсов (из модели планировщика `Slurm` и симулятора `BatSim`). В последнем случае дополнительную логику «планирования» и разделения большого объема выделенных ресурсов реализует как раз сам пользователь внутри `ExecutionProfile`.

Реализуя свой планировщик пользователь может реализовать интерфейс планировщика `Scheduler` (см. рисунок 4.8).

```
pub trait Scheduler {
  fn on_host_added(&mut self, ctx: &Context, host: HostConfig);
  fn on_execution_request(&mut self, ctx: &Context, req: ExecutionRequest);
  fn on_collection_request(&mut self, ctx: &Context, req: CollectionRequest);
  fn on_execution_finished(&mut self, ctx: &Context, execution_id: u64);
  fn on_host_notification(&mut self, ctx: &Context, host_id: Id, resources:
     ResourcesPack);
}
```

Рис. 4.8: Интерфейс планировщика `Scheduler`

С помощью контекста (который передается в каждый метод интерфейса 4.8) планировщик получает доступ к генератору псевдослучайных чисел и управляет симуляцией: ставит на исполнение задачи, выбрав для них набор серверов и частично отменяет их. Полная отмена в данный момент не полностью реализована, например невозможно отменить операцию передачи данных по сети, это связано с внутренними ограничениями модуля `dslab_network`, которые в будущем могут быть доработаны.

Для более точного управления моделью планировщика можно запустить симуляцию, передав ей отдельный компонент симуляции `DSLab`, который будет получать сырые события от компонентов `Proxy` (о новых задачах) и `Cluster` (о свободных ресурсах и завершенных задачах). Тогда у пользователя будет еще больше контроля над моделью планирования: компонент-планировщик сможет отправлять себе произвольные события, использовать асинхронные функции из асинхронного расширения ядра `DSLab` и т.д. Это качественно выделяет разработанный симулятор от аналогов, т.к. другие симуляторы предлагают только `callback` модель, где пользователь пишет обработку всех типов событий, приходящих от кластера. В новом симуляторе же есть возможность использовать асинхронное расширение ядра `DSLab` и реализовать программу-планировщик на качественно другом уровне. Подробнее о преимуществах этого функционала можно прочитать в работе о добавлении асинхронности в `DSLab`[17].

4.5 Входные данные симуляции

4.5.1 Трейс нагрузки

В качестве входных данных симуляции выступает трейс нагрузки на кластер. Трейс представляет собой набор задач, которые поступают на кластер в течение симуляции. Как описывалось в разделе 4.3.1, они могут быть связаны некоторым общим заданием. В данный момент симулятор поддерживает несколько видов входной нагрузки:

- `Native` – трейса нагрузки в формате `YAML`, описанного в секции 4.3. Состоит из нескольких файлов: файл с описанием задач (`ExecutionRequest`-ы), файл с описанием заданий (`CollectionRequest`-ы) и файл с описанием профилей нагрузки (`ExecutionProfile`).

- `Google` – трейс нагрузки кластера `Google`[25]. Версия 2019 года представлена в виде таблицы `BigQuery`, из которой достаточно сложно скачать большой объем данных (из-за ограниченных квот), однако версию 2011 года можно скачать в виде `CSV`-файла. Подробное описание и сравнение этих трейсов представлено в статье `Borg: Next Generation`[24] от Google. Для простоты в симуляторе предполагается, что все задачи потребляют все выделенные ресурсы на протяжении всего времени выполнения (заполняются тривиальными заглушками `ExecutionProfile`), однако в трейсе 2019 года присутствует подробная информация о профиле загрузки задачи, которую можно использовать для более точного моделирования в последующей работе. Сам трейс представлен в виде событий, поэтому его инкрементальное прочтение сделать сложно (для того, чтобы выяснить время выполнение задачи, необходимо прочитать трейс до события окончания задачи), из-за этого с трейсом удобнее работать, преобразовав его в формат `Native`. Поскольку данные о требованиях к ресурсам задач представлены в нормированном виде (числами с плавающей точкой от 0 до 1, где 1 – максимальное возможное значение ресурса среди всех доступных серверов), то при чтении этого трейса необходимо указать константу, на которую умножаются эти значения, чтобы получить целочисленные значения CPU и памяти.

- `Alibaba` – Трейс нагрузки кластера `Alibaba` 2018 года[1]. Состоит из двух файлов: файл `batch_tasks.csv` описывает зависимости между выполняемыми задачами (`instances`), информация о которых содержится в `batch_instances.csv`. Информация о задачах по аналогии с Google Trace переводится в тривиальные профили нагрузки. Однако, этот трейс можно читать инкрементально, что упрощает работу с ним.

- `Standard Workload Format Reader` – Трейс нагрузки в формате `SWF`[23]. Транслируется в симулятор по аналогии с Alibaba Trace, т.к. может быть в исходном виде прочитан инкрементально.

- `Random Trace Generator`. Детерминированный генератор синтетической нагрузки на кластер. С помощью встроенного в симуляцию `DSLab` генератора случайных чисел создает заданное количество задач с заданными требованиями к ресурсам. Время выполнения (либо необходимая процессорная работа) генерируется из нормального распределения с заданными параметрами среднего и дисперсии (см. пример в секции 5.2). Аналогичным образом можно генерировать задержки между задачами: либо задать интервал, из которого сгенерируется задержка, либо задать фиксированную задержку между задачами. Альтернативный режим работы генератора – бесконечное создание задач, пока симуляция не будет остановлена (например, по внутреннему времени симуляции).

Все перечисленные генераторы нагрузки реализуют интерфейс `WorkloadGenerator` (см. 4.9). Поскольку количество задач может быть очень большим (входные файлы могут иметь размеры десятки гигабайт) и не поместится в оперативную память сразу, то генераторы нагрузки должны уметь читать данные инкрементально, выдавая задачи для планировщика пачками не больше указанного лимита. Предполагается, что задачи в трейсе упорядочены по времени поступления на кластер.

```
pub trait WorkloadGenerator {
  fn get_workload(
    &mut self, ctx: &SimulationContext, limit: Option<u64>,
  ) -> Vec<ExecutionRequest>;

  fn get_collections(&self, ctx: &SimulationContext) -> Vec<CollectionRequest>;
}
```

Рис. 4.9: Интерфейс `WorkloadGenerator`

Пользователь может реализовать этот интерфейс самостоятельно и тогда у него появится возможность использовать свой собственный формат трейса нагрузки. Как альтернативный вариант, можно конвертировать имеющийся трейс в один из поддерживаемых форматов.

4.5.2 Конфигурация кластера и сети

Существует несколько способов задать конфигурацию серверов в кластере:

1. Через конфигурационный файл в формате YAML. Пример конфигурации представлен на рисунке 4.10. Есть возможность задать группы серверов с одинаковой конфигурацией, а затем получать статистику утилизации ресурсов в серверах по группам (подробнее см. раздел 4.6). Составные части каждого сервера описаны в разделе 4.2. Через конфигурацию также можно указать вид сети, используемой в кластере, и ее параметры (например, указать распределение хостов по стойкам при использовании топологии `Fat-Tree`) (описана в разделе 4.2.2).

2. Через чтение конфигурации кластера из `Google-trace`.

3. Через чтение конфигурации кластера из `Alibaba-trace`.

Стоит отметить, что использование трейса нагрузки из какого-либо источника (`Google trace` или `Alibaba trace`) не обязывает использовать конфигурацию кластера из этого же источника и наоборот. Однако, учитывая, что в обоих трейсах информация о ресурсах является нормированной (в целях скрыть данные о реальных мощностях кластера и объеме выполняемых задач), использовать их как-то иначе представляется практически невозможным.

4.6 Выходные данные симуляции

Выходными данными симуляции являются файлы с метриками загруженности кластера и очереди задач. В процессе работы отдельный компонент `Monitoring` собирает данные о мгновенной загрузке серверов и размере очереди, и агрегирует их, считая среднее значение на заданных временных промежутках. Пользователь может настроить интервал сжатия данных в конфигурационном файле симуляции (см. конфигурацию 4.10). В разделе справедливости распределения ресурсов по умолчанию собираются метрики использования доминантной доли ресурса различными пользователями[14]. Подробнее это рассмотрено в примере использования симулятора 5.2. По полученным файлам в формате `CSV` можно проводить анализ данных и строить графики, например с помощью известных библиотек `Python`: `pandas` и `matplotlib`.

4.7 Конфигурация симуляции

Вся настройка параметров симуляции происходит через конфигурационный файл в формате YAML. Пример конфигурации представлен на рисунке 4.10.

```yaml
workload:
  - type: Native
    path: /path/to/trace
    options:
      profiles_path: /path/to/profiles

hosts:
  - count: 5
    cpus: 20
    memory: 20
    name_prefix: c
  - count: 5
    cpus: 5
    memory: 60
    name_prefix: m

monitoring:
  host_load_compression_time_interval: 100
  scheduler_queue_compression_time_interval: 100
  display_host_load: false
  collect_user_queues: true
  output_dir: /path/to/output_dir

scheduler:
  hosts_invoke_interval: 10
```

Рис. 4.10: Пример конфигурации симуляции

В такой симуляции входная нагрузка будет в формате `Native` с указанными дополнительными профилями нагрузки. Используются два типа серверов: 5 серверов с 20 ядрами CPU и 20 Гб памяти и 5 серверов с 5 ядрами CPU и 60 Гб памяти. Сбор статистики загруженности кластера ведется только по группам и целиком на кластер, а также собираются метрики справедливости планирования (по пользователям). Метрики загруженности и размера очереди сжимаются на отрезках времени в 100 единиц и берется среднее на каждом отрезке. Каждый хост сообщает о свободных ресурсах планировщику каждые 10 единиц времени (единицы измерения могут быть любыми, но по умолчанию подразумеваются секунды).

4.8 Параллельный запуск нескольких симуляций

Для более быстрой параллельной проверки нескольких гипотез можно воспользоваться функциональностью параллельного запуска симуляций. Для каждой симуляции необходимо предоставить отдельный конфигурационный файл. Во избежание конфликтов важно, чтобы выходные данные записывались в разные директории (это тоже можно указать в конфигурации). Пример использования параллельного запуска симуляций представлен в репозитории с симулятором[7].

5 Эксперименты и тестирование

5.1 Производительность

В рамках эксперимента на производительность симуляции был взят `Alibaba-trace` 2018 года[1]. Трейс описывает загрузку кластера из 4000 серверов в течении 8 дней и содержит 1.5 миллиарда задач. Суммарный объем файла с задачами составляет 105GB. Для симуляции было взято подмножество задач первых двух с половиной дней (около 380 миллионов задач), входной файл занимает 25GB. Благодаря инкрементальному чтению трейса, нет необходимости загружать все данные в оперативную память сразу. Время работы симулятора в зависимости от количества входных задач представлена на графике 5.1. Видно, что рост времени работы линейный от количества задач.

Получается, что на симуляцию кластера в течение двух с половиной дней (220000 секунд) ушло 2600 секунд (около 45 минут). Таким образом получаем коэффициент ускорения в 85 раз (по сравнению с реальным временем работы кластера). При этом использовался простейший алгоритм планирования, время работы которого не зависит от количества задач в очереди, и который старается максимально загрузить все имеющиеся ресурсы. Стоит отметить, что от эффективности работы пользовательского алгоритма планирования напрямую зависит и время работы симулятора. Если задачи будут слишком быстро накапливаться и не будут планироваться на кластер (и, соответственно, уничтожаться по завершению), то это приведет к резкому росту потребления оперативной памяти программой-симулятором, т.к. инкрементальное чтение задач из трейса происходит согласно внутреннему времени симуляции.

Еще одной проблемой при запуске таких больших симуляций является необходимость подать на вход симулятору файл с задачами отсортированными по времени поступления в планировщик, однако опубликованные трейсы обычно не отсортированы. Поэтому перед запуском необходимо решить задачу сортировки большого `CSV` файла во внешней памяти.

При сравнении производительности симуляции с другими симуляторами был проведен замер сравнение времени работы на нагрузке из 200000 простых независимых задач. Симулятор `AccaSim` справился с работой за 25 секунд, а симулятор на основе `DSLab` – за 2.5 секунды. Таким образом, симулятор на основе `DSLab` работает в 10 раз быстрее, чем `AccaSim`. Это связано с тем, что `AccaSim` реализован на языке `Python`, который существенно уступает в производительности языку `Rust`. Симуляторы `BatSim`, `IRMaSim` и `ElastiSim` не поддерживают инкрементальное

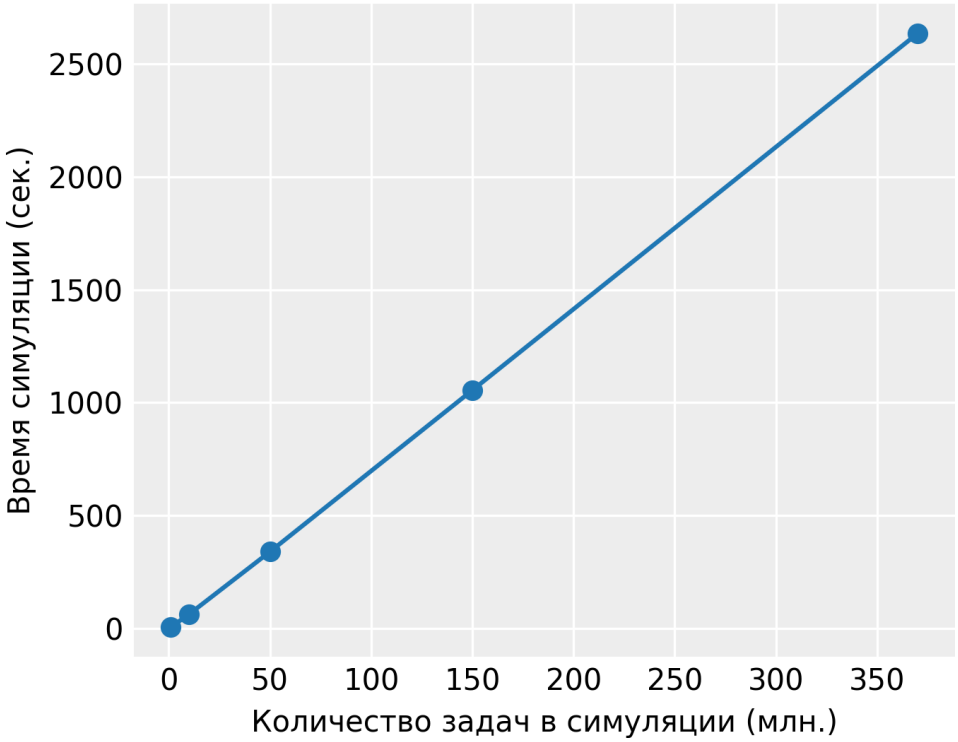


Рис. 5.1: Время работы симуляции в зависимости от количества задач

чтение трейса нагрузки, поэтому запуск таких больших симуляций в них не представляется возможным (из-за слишком большого одновременного потребления оперативной памяти). Это нельзя назвать однозначным недостатком, ведь эти симуляторы фокусируются на другой, более узкой области – точное моделирование сложных задач и сети.

5.2 Справедливость

В качестве демонстрации применения симулятора были проведены эксперименты на известный алгоритм справедливого распределения кластера между несколькими пользователями. Были реализованы алгоритмы `DRF`[14] и `Tetris`[15].

Алгоритм `DRF` (Dominant Resource Fairness) основан на понятии доминантного ресурса пользователя. Если пользователь потребляет 1/10 всей памяти в кластере, 1/15 всех ядер процессора, то для него доминантной долей будет считаться число 1/10 (в ситуации, когда распределяется два доступных ресурса – CPU и RAM). Смысл алгоритма состоит в том, чтобы каждый пользователь получал равную доминантную долю ресурсов (в случае, когда квоты пользователей равны, иначе доля считается от выделенной на кластере квоты). Поэтому из всех задач, которые готовы к исполнению планировщик в каждый момент времени выбирает

самого обделенного пользователя и запускает его задачу.

Такой подход позволяет обеспечить справедливое распределение ресурсов, однако упаковка задач на серверах может быть неоптимальной. Для борьбы с этим был предложен алгоритм `Tetris`[15]. На вход принимается произвольная функция справедливости (в нашем случае это `DRF`), а также «коэффициент справедливости» $f \in [0, 1]$. В каждый момент принятия решения, планировщик сортирует задачи по приоритету их размещения на кластере с точки зрения справедливости, а затем выбирает наилучшую задачу с точки зрения упаковки среди $(1 - f)$ доли очереди. В качестве эвристической функции упаковки `Tetris` использует скалярное произведение вектора ресурсов задачи на вектор свободных ресурсов сервера. Чем оно больше, тем лучше задача упаковывается на сервер. В таком виде приоритет явно отдается более тяжелым задачам, чтобы этого избежать, можно разделить результат на нормы векторов ресурсов.

Таким образом, конфигурируя алгоритм через параметр f, можно добиться баланса между справедливостью и плотностью упаковки задач. При $f = 0$ на каждом шаге планировщик выбирает наиболее подходящую задачу с точки зрения упаковки (т.к. выбирает среди всей доступной очереди), а при $f = 1$ выбирает наиболее обделенного пользователя (т.к. может взять только первую задачу из очереди). Стоит отметить, что при любых значениях $f < 1$ справедливость распределения ресурсов не гарантируется и существует возможность развития событий, при котором один из пользователей оказывается полностью обделен ресурсами на кластере, т.к. его задачи всегда подходят хуже других. На практике (при значении f достаточно близком к 1) такое случается крайне редко из-за очень большого количества задач с различными требованиями.

Для такой симуляции был выбран кластер с распределением ресурсов по серверам, указанными на графике 5.2.

В качестве синтетической нагрузки были сгенерированы 10 пользователей с равной долей доступа к кластеру с запросами на задачи, представленными в таблице 5.1. Каждый пользователь непрерывно поставлял задачи со своими фиксированными требованиями (CPU и RAM). Продолжительность задачи генерировалась из нормального распределения с параметрами среднего (`duration mean`) и дисперсии (`duration dev`) для каждого пользователя.

При симуляции с разными параметрами справедливости производились замеры средней утилизации кластера от момента полной загрузки кластера до момента, когда у какого-либо пользователя заканчивались задачи. Результаты представлены на графиках 5.3.

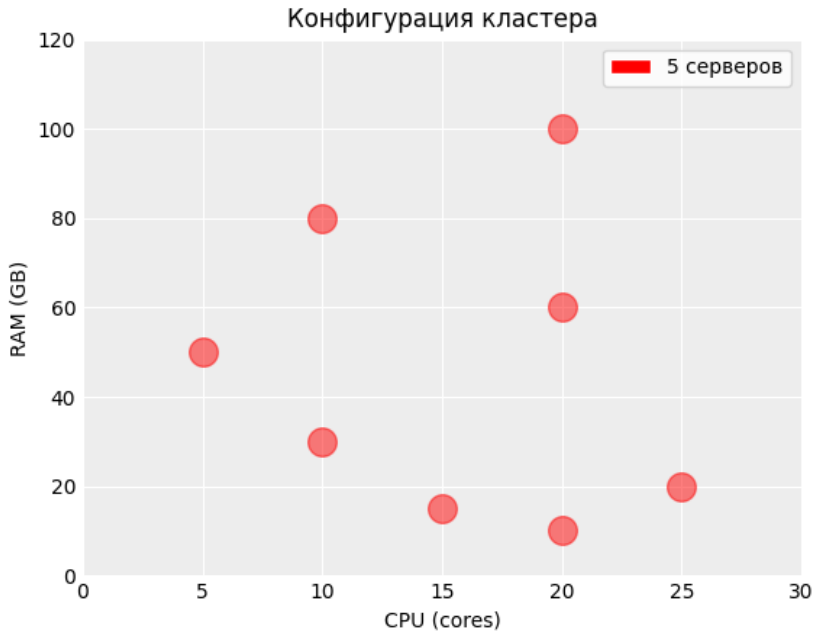


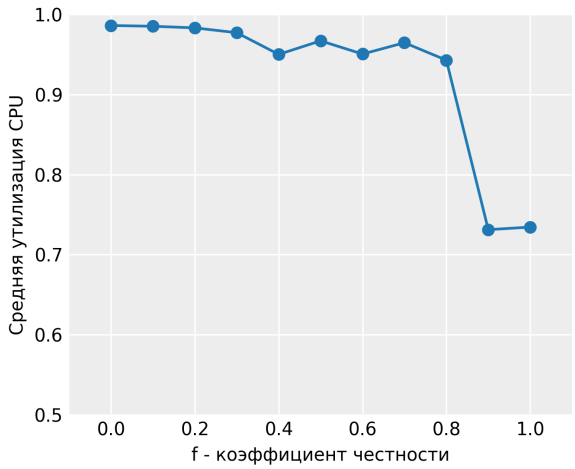
Рис. 5.2: Распределение ресурсов на серверах кластера

Пользователь	CPU	RAM	Количество задач	Duration Mean	Duration Dev
user0	1	5	1500	250	40
user1	1	10	800	400	40
user2	1	15	800	300	30
user3	2	5	1300	250	30
user4	2	6	1200	250	30
user5	2	7	800	300	40
user6	5	4	1000	180	30
user7	5	5	700	160	10
user8	7	5	200	800	200
user9	10	2	400	220	30

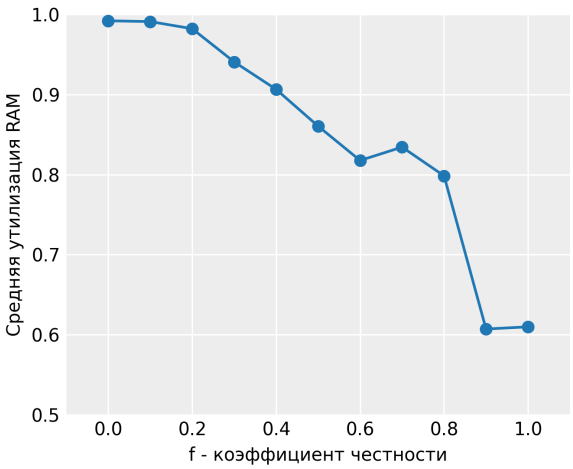
Таблица 5.1: Описание синтетической нагрузки на кластер

Из графиков видно, что качественный скачок в утилизации происходит при значении параметра справедливости $f = 0.8$. Это можно объяснить тем, что ровно при этом значении параметра, алгоритм `Tetris` получает возможность выбирать из двух задач при планировании (т.к. всего пользователей 10). Как только появляется такая возможность, качество упаковки задач на кластере резко возрастает.

Чтобы сравнить справедливость распределения, посмотрим на графики 5.4 доминантных долей пользователей в течение времени симуляции. Можно отметить, что при коэффициенте $f = 0.6$ (график 5.4d) справедливость страдает не сильно, особенно пока у всех пользователей есть задачи, а при $f = 0.5$ (график 5.4c) уже можно видеть, что два пользователя существенно обделяются (получают долю в два раза меньше, чем получает большинство), еще два других



(a) Утилизация CPU в зависимости от f



(b) Утилизация памяти в зависимости от f

Рис. 5.3: Утилизация кластера в зависимости от параметра справедливости

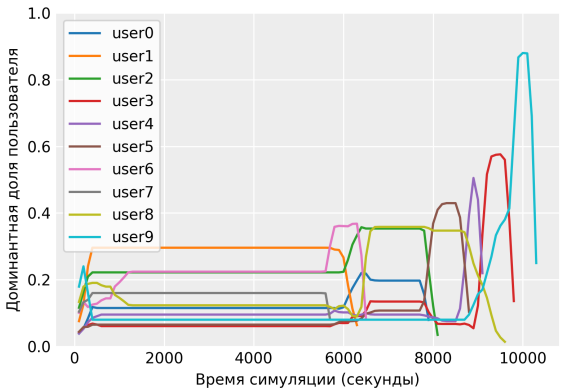
пользователя обделяются не так сильно. На обоих графиках видно, что самым обделенным пользователем является `user9` – это можно объяснить тем, что у него профиль нагрузки задач отличается больше всех от других: он потребляет больше всех `CPU` и меньше всех `RAM`, поэтому его задачи хуже всего подходят в освободившиеся слоты от других пользователей.

Ожидаемо можно отметить, что при $f = 1$ доли у всех пользователей абсолютно одинаковые. В случае, когда у какого-то пользователя заканчиваются задачи, освободившиеся ресурсы равномерно распределяются по оставшимся пользователям. Но по времени симуляции, и по утилизации (графики 5.3) видно, что это не самый лучший вариант занять кластер. А при $f = 0$ планировщик по сути сортирует пользователей по тому, насколько хорошо их задачи подходят под кластер.

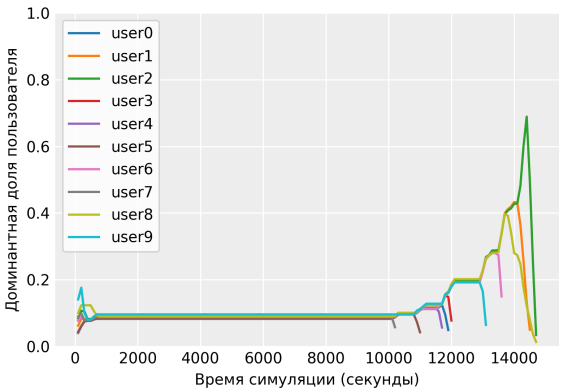
Таким образом можно сделать вывод, что для такой конфигурации кластера и такой нагрузки оптимальный параметр справедливости для классического алгоритма `Tetris` будет $f = 0.6$. Именно с целью проверки подобных гипотез можно использовать симулятор вместо того, чтобы запускать все эти эксперименты на реальном кластере, т.к. для проведения подобного эксперимента в симуляторе потребуется всего несколько секунд.

5.3 Детерминированность симуляции

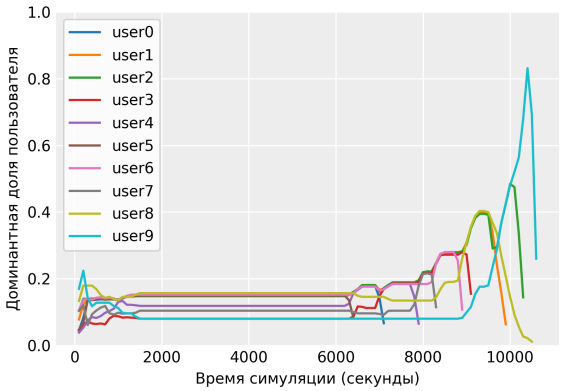
Важным аспектом любой симуляции является воспроизводимость результатов. Фреймворк `DSLab` предоставляет универсальный интерфейс доступа к генератору случайных чисел в симуляции, который зависит только от переданного в симуляцию сида. Реализованный симулятор



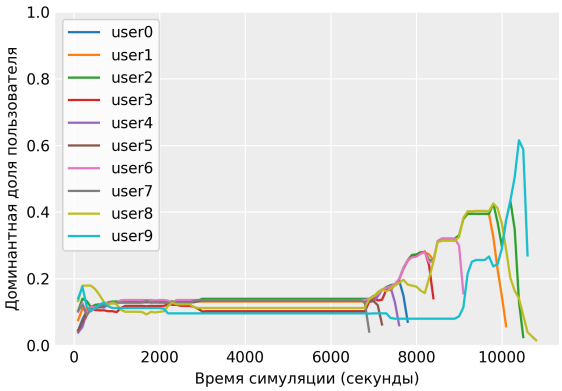
(a) $f = 0.0$



(b) $f = 1.0$



(c) $f = 0.5$



(d) $f = 0.6$

Рис. 5.4: Распределение доминантных долей использования кластера пользователями при различных коэффициентах справедливости f

полностью использует эту возможность и является детерминированным при условии детерминированности алгоритма планирования. Очень важным является использование генератора случайных чисел из переданного в планировщик контекста (см. интерфейс 4.8), однако в стандартных структурах данных языка `Rust` используется другой генератор случайных чисел, и это необходимо учитывать. Например, при итерации по ключам стандартной хэш-таблицы они будут возвращаться в разном порядке в зависимости от запуска программы.

5.4 Тестирование

Основной функционал внутренних модулей симулятора покрыт unit-тестами. На основе разобранного примера 5.2 сделано интеграционное тестирование. В репозитории проекта настроен механизм `CI`, автоматически проверяющий на корректность все поступающие изменения в коде.

6 Заключение

В рамках ВКР удалось реализовать многофункциональный и производительный симулятор вычислительного кластера. Код симулятора опубликован в открытом доступе на GitHub[6]. В этом же репозитории представлены и примеры запуска симулятора в разных конфигурациях. Таким образом главная цель работы достигнута, выполнены основные требования. Как главный результат работы можно выделить основное достижение: возможность пользователям исключительно гибко (по сравнению со всеми другими симуляциями) настраивать профиль загрузки сервера. Использование асинхронных примитивов языка `Rust` дает много возможностей для такого точного описания нагрузки. При этом, поддерживаются и более простые сценарии использования симулятора, в которых достигается высокая производительность.

С помощью симулятора проведены несколько экспериментов, которые показывают его применимость на больших симуляциях и в разных моделях планирования: справедливого распределения ресурсов между пользователями и оптимальной упаковки задач на кластере.

В качестве дальнейшего развития симулятора можно модифицировать модули `DSLab`, сделав возможной более полную отмену задач (добавить такой функционал в модулях `dslab-network` и `dslab-disk`). Другим направлением дальнейшей деятельности может стать поддержка других моделей заданий с масштабируемыми и гибкими требованиями к ресурсам.

Список литературы

[1] *Alibaba cluster-trace-v2018*. Technical Report. Posted at https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md. 2018.

[2] *BatSim profile types overview*. URL: https://batsim.readthedocs.io/en/latest/input-workload.html#profile-types-overview.

[3] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson и Frédéric Suter. "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms". в: *Journal of Parallel and Distributed Computing* 74.10 (июнь 2014), с. 2899—2917. DOI: 10.1016/j.jpdc.2014.06.008. URL: https://inria.hal.science/hal-01017319.

[4] Jeffrey Dean и Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". в: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, с. 137—150.

[5] *DSLab architecture*. URL: https://docs.google.com/document/d/12CcGpdulqMJppAYoNrOdpIWZYEa3flXn9JMX5fdyAnY/edit.

[6] *DsLab Cluster Scheduling Simulator repository*. URL: https://github.com/nogokama/dslab-cluster-simulator.

[7] *DsLab Cluster Scheduling Simulator. Example of launching parallel simulations*. URL: https://github.com/nogokama/dslab-cluster-simulator/tree/main/examples/parallel-launcher.

[8] *DsLab comparison to other simulators*. URL: https://docs.google.com/document/d/1H2711nGiS6m7QTo4pMfmrIBmg4llLKUhQzDiFNIw-UU/edit.

[9] *DSLab compute module documentation*. URL: https://osukhoroslov.github.io/dslab/docs/dslab_compute/index.html.

[10] *DSLab network module documentation*. URL: https://osukhoroslov.github.io/dslab/docs/dslab_network/index.html.

[11] *DsLab repository*. URL: https://github.com/osukhoroslov/dslab.

[12] *DSLab storage module documentation*. URL: https://osukhoroslov.github.io/dslab/docs/dslab_storage/index.html.

[13] Cristian Galleguillos, Zeynep Kiziltan, Alessio Netti и Ricardo Soto. *AccaSim: a Customizable Workload Management Simulator for Job Dispatching Research in HPC Systems*. 2018. arXiv: 1806.06728 [cs.DC].

[14] Ali Ghodsi, Matei A. Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker и Ion Stoica. “Dominant Resource Fairness: Fair Allocation of Multiple Resource Types”. в: *Symposium on Networked Systems Design and Implementation*. 2011. URL: https://api.semanticscholar.org/CorpusID:6693615.

[15] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao и Aditya Akella. “Multi-resource packing for cluster schedulers”. в: *SIGCOMM Comput. Commun. Rev.* 44.4 (авг. 2014), с. 455—466. ISSN: 0146-4833. DOI: 10.1145/2740070.2626334. URL: https://doi.org/10.1145/2740070.2626334.

[16] Adrián Herrera, Mario Ibáñez, Esteban Stafford и Jose Luis Bosque. “A Simulator for Intelligent Workload Managers in Heterogeneous Clusters”. в: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021, с. 196—205. DOI: 10.1109/CCGrid51090.2021.00029.

[17] *Implementation of Asynchronous Programming Support for the DSLab Framework*. URL: https://nogokama.github.io/ThirdYearCoursework/report.pdf.

[18] Taylan Özden, Tim Beringer, Arya Mazaheri, Hamid Mohammadi Fard и Felix Wolf. “ElastiSim: A Batch-System Simulator for Malleable Workloads”. в: *Proceedings of the 51st International Conference on Parallel Processing*. ICPP ’22. Bordeaux, France: Association for Computing Machinery, 2023. ISBN: 9781450397339. DOI: 10.1145/3545008.3545046. URL: https://doi.org/10.1145/3545008.3545046.

[19] F. Petrini и M. Vanneschi. “k-ary n-trees: high performance networks for massively parallel architectures”. в: *Proceedings 11th International Parallel Processing Symposium*. 1997, с. 87—93. DOI: 10.1109/IPPS.1997.580853.

[20] *Rust crate «async-trait»*. URL: https://crates.io/crates/async-trait.

[21] *Rust crate «futures»*. URL: https://crates.io/crates/futures.

[22] *Slurm CPU Management User and Administrator Guide*. URL: https://slurm.schedmd.com/cpu_management.html.

[23] *The Standard Workload Format Specification*. URL: https://www.cs.huji.ac.il/labs/parallel/workload/swf.html.

[24] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter и John Wilkes. “Borg: the Next Generation”. в: *EuroSys’20*. Heraklion, Crete, 2020.

[25] John Wilkes. *Google cluster-usage traces v3*. Technical Report. Posted at https://github.com/google/cluster-data/blob/master/ClusterData2019.md. Mountain View, CA, USA: Google Inc., апр. 2020.

[26] *Устройство планировщика Yandex YT. Запись с конференции HighLoad*. URL: https://www.youtube.com/watch?v=Uv-IcGZSRpk&ab_channel=HighLoadChannel.