МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Масыча Матвея Алексеевича

Тема работы:

РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМОВ ПОВЫШЕНИЯ ДОСТУПНОСТИ ДАННЫХ В СИСТЕМЕ LUNA

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В. Э./……..
(ФИО) / (подпись)
«27» мая 2021 г.

Руководитель ВКР
д.т.н., профессор
зав. каф. ПВ ФИТ НГУ
Малышкин В. Э./……...
(ФИО) / (подпись)
«27» мая 2021 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ
Перепёлкин В. А./……
(ФИО)/(подпись)
«27» мая 2021 г.

Новосибирск, 2021

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий
Кафедра параллельных вычислений
(название кафедры)

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В. Э.
(фамилия, И., О.)

…….
(подпись)

«17» декабря 2020 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Масычу Матвею Алексеевичу, группы 17203
(фамилия, имя, отчество, номер группы)

Тема: Разработка и реализация алгоритмов повышения доступности данных в системе LuNA
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 31.05.2021 № 0183
Срок сдачи студентом готовой работы «31» мая 2021 г.
Исходные данные (или цель работы): разработать и реализовать в виде программных модулей системы фрагментированного программирования LuNA эффективные алгоритмы повышения доступности фрагментированных данных
Структурные части работы: обзор, разработка алгоритмов, реализация алгоритмов, тестирование

Руководитель ВКР
зав. каф. ПВ ФИТ,
д. т. н., профессор
Малышкин В. Э./……..
(ФИО) / (подпись)

«17» декабря 2020 г..

Задание принял к исполнению
Масыч М. А./……..
(ФИО студента) / (подпись)

«17» декабря 2020 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ,
Перепелкин В. А./……..
(ФИО) / (подпись)

«17» декабря 2020 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Важной и трудоемкой частью разработки параллельных программ является управление данными. В распределенных вычислительных системах обеспечение доступности данных, поддержка сбалансированного распределения данных по процессорным элементам и приемлемой скорости доступа к ним — нетривиальные задачи.

В общем случае, для задачи управлениям распределенными данными не существует универсального алгоритма. Во многих связанных с распределенными системами сферах человеческой деятельности, начиная от высокопроизводительных вычислений и заканчивая облачными хранилищами, возникает множество различных решений, которые показывают себя лучше или хуже на тех или иных классах задач.

Разработка алгоритмов управления распределенными данными — активная сфера научной деятельности. Упомянутая ранее невозможность разработки универсального алгоритма стала почвой для разнообразных работ в данной области, которые выпускались и продолжают выпускаться в больших количествах. В своих решениях авторы используют различные подходы и касаются разнообразных сторон управления данными в распределенных системах. Положительные и отрицательные свойства различных подходов и конкретных примеров будут рассмотрены далее в обзоре.

Многие алгоритмы управления распределенными данными стремятся к хорошим средним результатам в своей области применения. Однако, неизбежно теряют эффективность на некоторых конкретных задачах.

Одним из способов обойти подобную проблему является накопление и применение частных алгоритмов. Такие алгоритмы разработаны с узкой областью

применения, в которой их использование позволило бы повысить эффективность программы относительно более универсальных решений.

Одной их актуальных подзадач в области управления данными является проблема доступности данных.

Доступность данных, которая может быть выражена, например, в количестве переходов между узлами при их запросе, в значительной степени влияет на производительность распределенной системы. Системы с высокой доступностью тратят меньше времени на поиск и переправку данных в сети, тем самым сокращая время работы программы.

Система фрагментированного программирования LuNA является удобной средой для развития и накопления подобных алгоритмов, являясь средством для решения разнообразных распределенных задач и предоставляя удобные средства разработки.

Целью работы является разработка и реализация алгоритмов, повышающих доступность данных, в системе LuNA.

Для достижения цели поставлены следующие **задачи**:

1. Провести исследование распространенных задач, которые могут встречаться при работе с системой LuNA, и уже существующих алгоритмов управления распределенными данными, которые в том или ином роде сталкиваются с подобными задачами.
2. На основе анализа, сделанного в предыдущем пункте, должны быть сформированы требования к разрабатываемым алгоритмам.
3. Разработать частные алгоритмы управления распределенными данными, основываясь на сформированных ранее требованиях.
4. Реализовать разработанные частные алгоритмы в системе LuNA.

5. Полученная реализация алгоритмов должна быть протестирована: программы использующие алгоритмы должны быть по какому-либо критерию более эффективны, чем программы без них (основным критерием является время исполнения программы).

Научная новизна работы заключается в разработке алгоритмов повышения доступности данных в системе LuNA. Эти алгоритмы позволят увеличить эффективность генерируемых системой LuNA программ, что выражает **практическую ценность** работы.

1 Обзор

Так как разработка алгоритмов управления распределенными данными активно развивающаяся область, важно обратить внимание на уже существующие решения. Анализ существующих решений позволит рассмотреть существующий опыт разработки подобных алгоритмов и незакрытые в сфере проблемы.

Одними из самых распространенных алгоритмов управления распределенными данными являются вариации диффузионных алгоритмов, основной задачей которых является динамическая балансировка нагрузки на процессорные элементы. Одной из основных причин широкой распространенности подобных алгоритмов является их простота. Суть диффузионных алгоритмов заключается в сравнении загрузки соседних процессорных элементов (или групп процессорных элементов) и перераспределении данных, если разница загрузки соседей достигает определенного уровня. Важной особенностью диффузионных алгоритмов является их децентрализованность и локальность межпроцессорных взаимодействий, что делает такие алгоритмы легко масштабируемыми, что является важной характеристикой для алгоритмов, используемых в распределенных системах. В работе [1] приводится одна из реализаций диффузионного алгоритма. Приведенный алгоритм работает быстрее стандартных реализаций, но не лишен обычных для всех диффузионных алгоритмов проблем: алгоритм не чувствует медленного нарастания нагрузки и сталкивается с проблемой “горы песка”.

Важной областью применения алгоритмов управления распределенными данными является распределенные базы данных.

В работе [2] предлагается алгоритм управления распределенными данными ASURA. Алгоритм ASURA может быть разделен на два этапа: сначала мы присваиваем процессорные элементы отрезкам на числовой прямой, а затем определяем какой из этих элементов будет хранить данные. На первом шаге происходит добавление или удаление элементов, и, как говорилось ранее, присвоение их числовой прямой с следующими обязательными условиями: длина каждого отрезка соответствует объему памяти элемента и связи уже существующих элементов и отрезков не меняются. Затем каждому фрагменту данных присваивается номер, который используется в качестве порождающего элемента в генераторе псевдослучайных чисел, который генерирует числа с распределением близким к равномерному. Используя этот генератор, мы получаем последовательность псевдослучайных чисел в заданном промежутке. Первое число, которое попадает в какой-либо из сегментов элементов определяет элемент, в который попадает фрагмент данных. Редактируя последовательность случайных чисел, можно удалять и добавлять процессорные элементы (перераспределение данных при этом будет оптимальным), увеличивать максимально допустимое количество элементов или уменьшать его, тем самым увеличивая эффективность алгоритма. Тем не менее ASURA обладает следующими минусами: в алгоритме никак не используется информация о структуре данных, не решен вопрос с поиском данных, не учитывается топология кластера и свойства связей между процессорными элементами.

В работе [3] представлены RUSH-алгоритмы. RUSH (Replication Under Scalable Hashing) — это семейство из нескольких алгоритмов, работающих с объектно-ориентированными хранилищами данных. Одной из главных возможностей алгоритмов является возможность создания и распространения множества копий объектов по многочисленным объектным хранилищам, а также

позволяет отдельным клиентам вычислять местонахождение всех копий объекта в системе, используя только список серверов хранилищ. Другой важной особенностью является поддержка масштабируемости системы. Поддержка масштабируемости реализуется через систему двойного поиска: поиск сначала ведется на уровне под-кластеров, а затем уже внутри на уровне дисков внутри под-кластера. Двухфазность этой системы позволяет совмещать различные алгоритмы отображения под-кластеров и отображения дисков. Несмотря на относительную масштабируемость алгоритма, применение алгоритма RUSH ограничено, так как в нем используются глобальные коммуникации, которых следует избегать.

Другой важной сферой, в которой используются алгоритмы управления распределенными данными, являются облачные технологии.

В работе [4] описывается алгоритм распределения ресурсов в распределенных облаках. Используя данные о сети, алгоритм пытается повысить эффективность работы с облаком. В общих чертах алгоритм выглядит так:

- Выбор дата-центра: находим подмножество дата-центров подходящих пользователю, а затем выбираем из оставшихся центры таким образом, чтобы уменьшить расстояние между ними. Важной задачей является уменьшение самого длинного пути, так как этот путь будет порождать самую большую задержку, отрицательно влияя на общую производительность системы.
- Распределение задач по дата-центрам: задачи распределяются по выбранным ранее дата-центрам таким образом, чтобы уменьшить количество межцентровых взаимодействий.

- Выбор стойки, блейд-сервера и процессора: на этом этапе мы определяем физические ресурсы дата-центра и пытаемся выбрать машины с меньшим количеством взаимодействий между стойками.
- Размещение задачи: здесь мы распределяем отдельные задачи (обычно в форме виртуальных машин) по физическим ресурсом. Здесь стараемся уменьшить взаимодействие виртуальных машин между стойками.

Хотя этот алгоритм и обладает сильными сторонами (например, учитывает строение системы для увеличения эффективности), он не лишен недостатков. Так как взаимодействий в облачных системах относительно редки перераспределения данных, да и они не обладают достаточной скоростью, такие системы используют централизованные алгоритмы, которые куда с большим трудом масштабируются на большое количество данных и процессорных элементов.

Подход к алгоритмам управления распределенными данными в области высокопроизводительных вычислений несколько отличается. Зачастую, многие задачи, с которыми мог бы в определенной степени справиться алгоритм, делаются вручную в целях оптимизации, кроме, разве что, динамических составляющих, вроде балансировки нагрузки. Подобным образом работают и упомянутые выше диффузионные алгоритмы. Тем не менее, ручной подход требует больших ресурсных затрат и требует высокой квалификации. Поэтому более широкие алгоритмы управления распределенными данными все же используются в данной области. Например, в системах конструирования параллельных программ.

Некоторые алгоритмы для решения таких задач подходят со стороны статического анализа. В работе [5] описывается алгоритм компилятора, находящий возможные варианты декомпозиции вычислений и данных.

Декомпозицией вычислений называется отображение вычислений на процессорный элемент параллельной системы, а декомпозицией данных — отображение данных в локальную память процессорного элемента. Проводя декомпозицию, мы получаем дополнительные возможности оптимизации. Ее главная суть заключается в повышении локальности данных: доступ не к локальной памяти является результатом дополнительных коммуникаций между элементами, которые могут очень сильно повлиять на производительность. Проведя же декомпозицию вычислений и данных, их соответствующее отображение на какой-либо элемент, мы можем получить прирост в скорости работы, как раз из-за уменьшения межпроцессорных взаимодействий. Тем не менее, такие алгоритмы статического анализа ограничены тем фактом, что принимать решения приходится на стадии компиляции и уже никак дальше.

Другие алгоритмы используют более универсальный подход. В работе [6] приведен алгоритм Rope-of-beads, использующийся в системе фрагментированного программирования LuNA. В данном алгоритме, каждый фрагмент данных статически отображается на отрезок [0, 1] и получает вещественное число, называемое координатой. Отрезок [0, 1] разделяется на сегменты, по одному на каждый процессорный элемент. Соседние сегменты присваиваются соседним (соединенным физически) процессорным элементам. Таким образом, Rope-of-Beads сохраняет локальность взаимодействий, обеспечивает постоянные потребление памяти и вычислительные расходы. Для балансировки нагрузки Rope-of-beads использует диффузионный алгоритм. Недостатками данного алгоритма являются линейное время поиска фрагмента данных (в худшем случае) и сложность отображения данных на отрезок [0, 1] в задачах с плохой локальностью.

На основе проведенного анализа можно сделать вывод, что многие существующие алгоритмы управления распределенными данными из области высокопроизводительных вычислений не рассматривают вопрос повышения доступности данных всеми доступными способами, ограничиваясь только возможностями успешного начального расположения данных.

2 Разработка алгоритма повышения доступности

2.1 Постановка задачи

Требуется разработать алгоритмы повышения доступности данных в системе LuNA. Использование алгоритмов должно повышать эффективность программы на соответствующих задачах. Основным критерием эффективности является время работы программы.

Система LuNA предназначена для работы на мультикомпьютере. Мультикомпьютер — это вычислительная система, в которой каждый из узлов обладает собственной памятью, а для передачи информации между узлами используется некоторая система сообщений [7]. В LuNA на каждом из узлов производится исполнение фрагментов вычислений. Фрагменты вычислений представляют собой некоторую часть программы, код, которые могут быть исполнены при наличии соответствующих им входных данных [8]. Этими данными в системе LuNA являются фрагменты данных, которые могут представлять собой обычные переменные, структуры и массивы. В результате выполнения фрагмента вычислений могут быть порождены новые фрагменты вычислений и данных. Фрагменты могут мигрировать между вычислительными узлами распределенной системы.

Для миграции фрагментов данных существует два механизма. В первом случае, данные требующиеся данные запрашиваются одним узлом у другого и отправляются после обработки запроса. В другом случае, узел отправляет данные нуждающемуся узлу данные по готовности.

Система LuNA работает с программами, описанными на одноименном фрагментированном языке программирования. Упомянутые выше механизмы миграции фрагментов данных специфицируются в тексте программы.

В контексте системы LuNA задача повышения доступности данных заключается в снижении количества и времени миграций фрагментов данных между узлами.

2.2 Описание предлагаемого решения

Для достижения высокой скорости получения данных алгоритмы управления распределенными данными из области высокопроизводительных вычислений обычно полагаются на организацию достаточно эффективного первоначального распределения данных.

Именно так, например, действуют представленный в обзоре алгоритм Rope-of-beads и родственный ему алгоритм из работы [9] — алгоритмы организуют субоптимальное распределение данных на узлах, предоставляют возможность их поиска и балансируют нагрузку. Однако данные алгоритмы теряют эффективность в ситуациях, где фрагмент данных на одном узле требуется на многих других узлах (например, фрагмент данных инициализируется на первых узлах, а потом используется в каких-либо вычислениях на других, подобные примеры можно увидеть далее в работе). Такая ситуация отрицательно сказывается на производительности системы, так как узел с фрагментом будет предоставлять запрашиваемые данные с задержкой из-за большого количества запросов. Кроме того, узел с фрагментом может быть неизвестен и на поиск фрагмента на узлах будет затрачено дополнительное время.

Предлагается предоставить альтернативу существующим механизмам распространения данных. Новое решение так же специфицируется в тексте

программы и должно поддерживаться системой LuNA (то есть, как только фрагменты данных будут выработан, над ними должны начать работу предлагаемые алгоритмы).

2.2.1 Массовая репликация

Для борьбы с перечисленными проблемами был предложен алгоритм массовой репликации. Данный алгоритм основан на стандартной операции массовой репликации, то есть репликации данных на все узлы распределенной системы.

Так как массовая репликация достаточно сильно нагружает память системы и сеть, предлагаемое решения предполагает применение массовой репликации только к фрагментам данных, которые укажет пользователь: операция массовой репликации вместе с фрагментами данных, которые желает реплицировать пользователь, указываются в тексте программы. Помеченные таким образом фрагменты рассылаются по всем узлам системы, где будут храниться на узле до удаления.

Подобное решение закрывает проблему поиска данных и обеспечения приемлемой скорости их получения. Так как фрагменту вычислений не нужно отправлять запрос на получение отсутствующего фрагмента данных другим узлам, так как, если программа корректна, фрагмент данных будет реплицирован на родной узел данного фрагмента вычислений.

В дополнение к текущему механизму удаления фрагментов данных в системе LuNA, была предложена отдельная операция удаления массово реплицированных данных. Для удаления всех реплик какого-либо фрагмента данных достаточно вызвать предлагаемую операцию только на одном узле.

2.2.2 Кэширование

В предлагаемом варианте алгоритма массовой репликации массово реплицированные фрагменты данных хранятся на узлах до удаления. Предлагается ввести более ограниченный механизм кэширования массово реплицированных данных.

В подобном варианте алгоритма при реплицирование также указывается количество запросов, которое будут храниться данные. После достижения указанного числа запросов на какой-либо из фрагментов, узел удаляет этот фрагмент данных.

Идея кэширования данных в системе LuNA не нова и уже предлагалась в работе [10], однако рассматривалась в контексте работы с единичными фрагментами данных, а не сразу со всем репликами фрагмента данных.

2.2.3 Ограниченная репликация

Также, в дополнение к алгоритму массовой репликации был предложен алгоритм ограниченной репликации. Ведь зачастую в массовой репликации может не оказаться смысла из-за более строгих ограничений на память узлов системы или достаточной высокой скорости коммуникаций между ними.

В подобных случаях выходом может оказаться ограниченная репликация, при которой реплицируемые данные распространяются не на все процессорные элементы, а только на какую-то часть.

Алгоритм, помимо информации о фрагменте данных, получает целое положительное число — степень репликации. Используя это число, алгоритм решает на какие узлы отправить реплику фрагмента данных, а на каких оставить стандартные операции запроса данных. Выбирать конкретные узлы для

ограниченной репликации предлагается взятием остатка от деления номера узла на степень репликации — если остаток оказывается нулем, то узел получает реплику (например, в системе из четырех узлов ограниченная репликация со степенью два отправит реплики на узлы под номерами ноль и два).

2.3 Характеристика предлагаемых решений

2.3.1 Характеристика массовой репликации

Благодаря массовой репликации, распространяющий фрагмент данных узел отправляет фрагменты данных массово, ему не требуется тратить дополнительное время на принятие и обработку поступающих от других процессорных элементов запросов, что вместе с относительно быстрыми коммуникациями внутри узла, позволяет повысить скорость получения нужных фрагментов данных, которая в значительной степени влияет на эффективность программы по скорости.

Алгоритм массовой репликации, тем не менее, обладает и недостатками. Массовая репликация данных произведенная в распределенной сети, где реплицируемый фрагмент запрашивает лишь малая часть, лишь засорит сеть лишними сообщениями и засорит память процессорных элементов ненужными данными. Тем не менее, как частный алгоритм управления распределенными данными массовая репликация может стать удобным и эффективным инструментом.

2.3.2 Характеристика кэширования

Кэширование позволяет использовать массовую репликацию в условиях более ограниченной памяти узлов. Благодаря кэшированию, реплики будут удалятся спустя какое-либо количество использований на узле, что позволит сэкономить время работы системы за счет отсутствия необходимости вызывать

какие-либо операции удаления фрагментов, и сделает использование памяти более эффективным.

Однако, при использовании подобного механизма нужно представлять, какое количество раз данные будут использованы на узле в программе. Если данные будут сохранены с предполагаемым количеством использований большим, чем на самом деле, то это приведет к снижению эффективности использования памяти (данные будут просто занимать память узла). Наоборот, если предполагаемое количество использований будет значительно ниже реального, то в использовании репликации может оказаться мало смысла, ведь данные узлу придется запрашивать заново.

2.3.3 Характеристика ограниченной репликации

Ограниченная репликация позволяет снизить количество коммуникаций при первоначальной рассылке желаемого фрагмента данных, что может повысить общее быстродействие системы, относительно массовой репликации.

Также сглаживается узкое место, при котором все узлы обращаются к одному за каким-либо фрагментом данных, значительно нагружая его. Однако, по сравнению с массовым вариантом, поиск нужного фрагмента данных не является тривиальной задачей и может занимать большее время, так как будет включать в себя межузловые коммуникации, которые в общем случае медленнее, чем внутриузловые, с помощью которых фрагмент вычислений получает от узла фрагмент данных в массовой репликации. Тем не менее, ограниченная репликация более щадяще нагружает память системы, и, при достаточной скорости межузловых взаимодействий, может достигать схожей с массовой репликацией эффективностью.

Ограниченная репликация, также как и массовая, дает прирост скорости в работе программы за счет снижения нагрузки на конкретный узел. В зависимости от конфигурации системы (например, скорости межузловых взаимодействий) ограниченная репликация может давать прирост скорости не хуже, или даже лучше, чем массовая.

3 Реализация алгоритмов

3.1 Описание решения

3.1.1 Описание системы LuNA

Система LuNA представляет собой набор программных модулей, взаимодействующих между собой. Данные модули реализованы на языке C++ в виде классов.

Одним из важнейших модулей является модуль RTS. Данный модуль отвечает за исполнение фрагментов вычислений на узле, их миграцию, предоставляет интерфейс для обмена и распространения фрагментов данных между фрагментами вычисления. RTS напрямую взаимодействует с модулем CF, который реализует поведение фрагмента вычислений. CF содержит в себе необходимую информацию для исполнения фрагмента, а также позволяет коду фрагмента вычислений пользоваться интерфейсом, предоставляемым RTS. На рисунке 1 приведена упрощенная схема взаимодействия данных модулей.

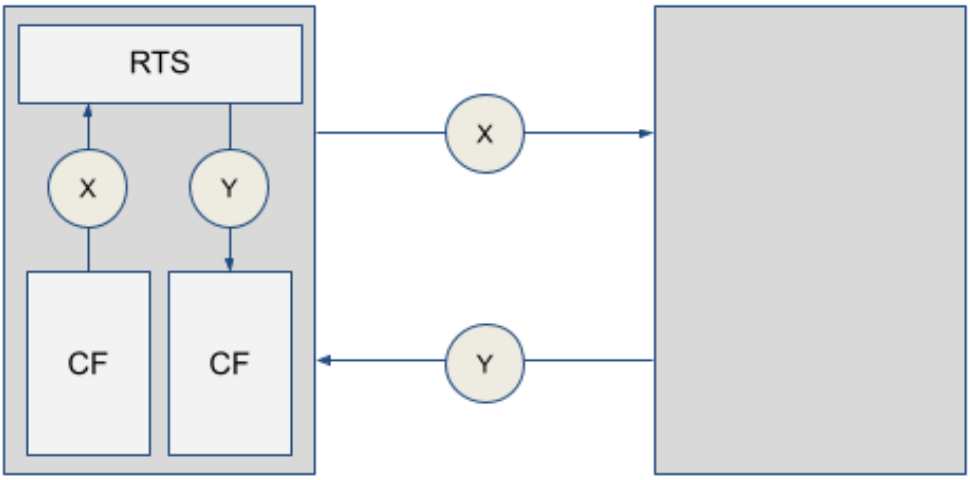

Рисунок 1 — упрощенная схема взаимодействия модулей RTS и CF

Другим важным модулем является модуль коммуникаций, представленный классом Comm. Данных модуль организует обмен сообщениями между модулями RTS на разных узлах при помощи технологии MPI.

3.1.2 Массовая репликация

Для использования массовой репликации на одном из своих фрагментов данных (рис. 2) фрагмент вычислений обращается к runtime-системе на своем узле (рис. 3), которая, в свою очередь, сохраняет в отдельном контейнере полученный фрагмент данных и рассылает его всем остальным узлам при помощи broadcast (широковещательной передачи данных).

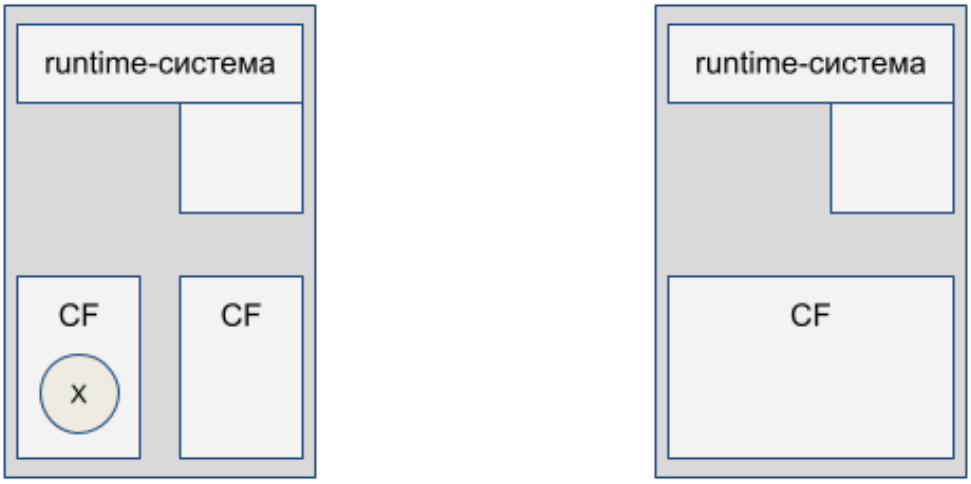

Рисунок 2 — хранение фрагментом вычислений фрагмента данных

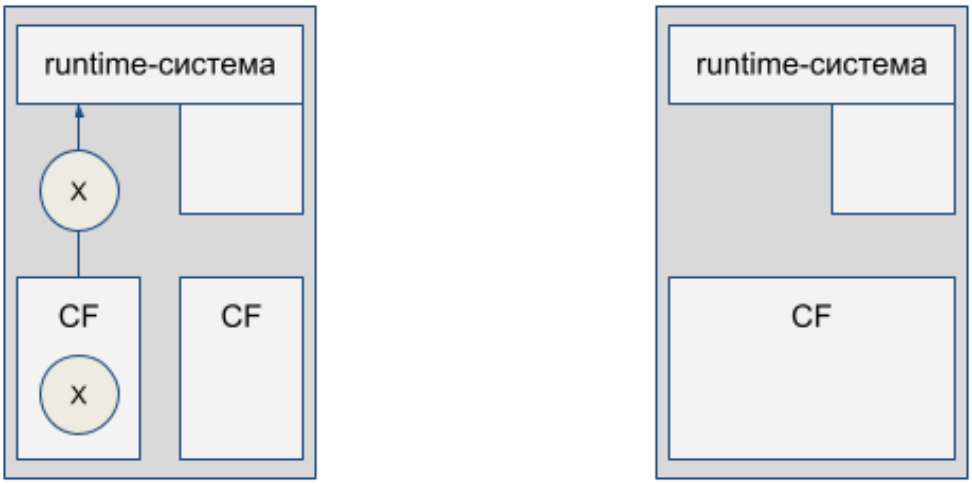

Рисунок 3 — обращение фрагмента вычислений к runtime-системе

Параллельно с этим остальные фрагменты вычислений могут запросить у своих узлов нужный фрагмент данных (рис. 4). Если фрагмент данных сохранен в контейнере узла, то он сразу же передается нуждающемуся фрагменту вычислений. В противном случае, фрагмент данных попадает в очередь (рис. 5).

Как только узел получит требующийся фрагмент данных, он передаст его в ожидающие фрагменты вычислений, после чего те могут продолжить работу (рис. 6).

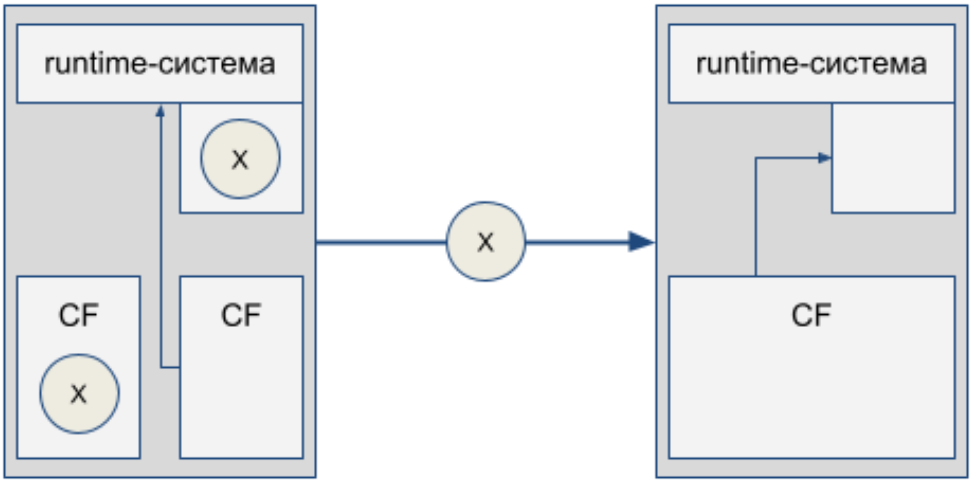

Рисунок 4 — запрос фрагментами вычислений фрагмента данных

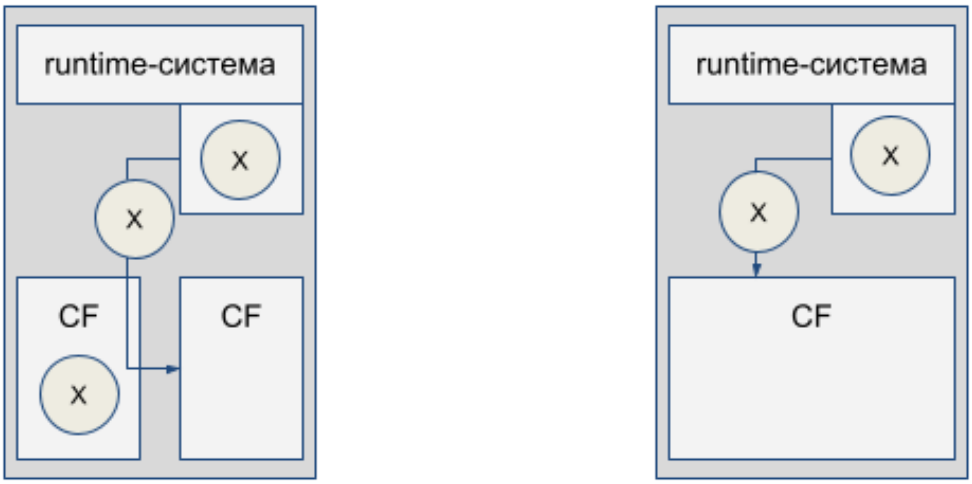

Рисунок 5 — отправка фрагментов вычислений runtime-системой

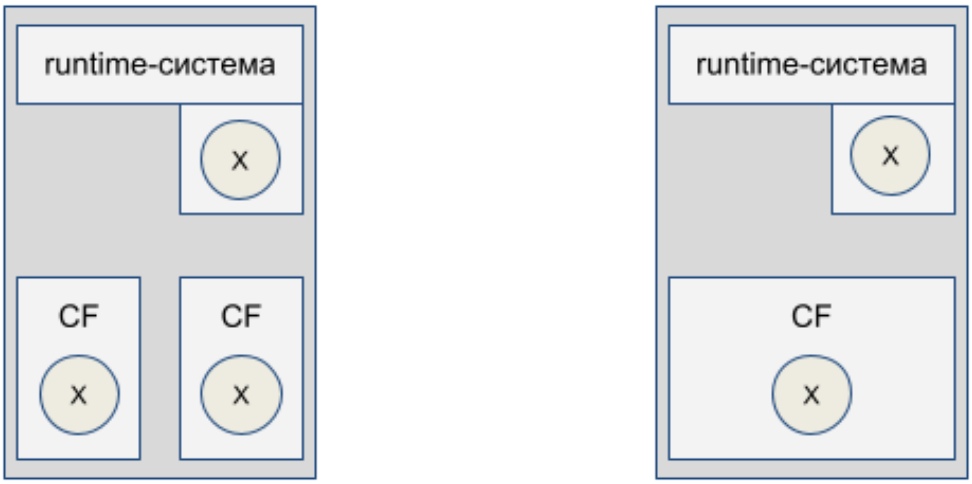

Рисунок 6 — все фрагменты вычислений получили нужные фрагменты данных

Для очищения хранилищ реплицированных данных на узлах был реализован метод массового удаления таких фрагментов. Данный метод может быть аналогично вызван фрагментом вычислений.

При вызове метод обратится к узлу с соответствующей просьбой. Если удаляемый фрагмент данных окажется в контейнере узла, то он будет удален, с проверкой, что данный фрагмент никто не ожидает.

Если же фрагмент никто не ожидает, но его нет в хранилище, то команда удаления оказывается в очереди, где ожидает появления нужного фрагмента данных на узле. В таком случае, фрагмент данных даже не будет сохранен, а команда удаления будет снята с очереди.

Для реализации массовой репликации программные модули (классы языка C++) CF и RTS были расширены новыми методами. В класс CF были добавлены методы all_push, expect_all_push, all_destroy, сигнатуры которых представлены в листинге на рисунке 7.

В класс RTS же были добавлены методы all_push, _all_push, expect_all_push, all_destroy, _all_destroy, а также поля all_pushed_, all_waiters_ и all_destroys_.

Сигнатуры представлены в листинге на рисунке 8. Псевдокод алгоритма представлен в листинге на рисунках 9 и 10.

```
void all_push(const Id &dfid, const DF &val);
void expect_all_push(const Id &dfid);
void all_destroy(const Id &dfid);
```

Рисунок 7 — сигнатуры новых методов класса cf

```
void all_push(const Id &dfid, const DF &val);
void expect_all_push(const Id &dfid,
      std::function<void (const Id &, const DF &)>);
void all_destroy(const Id &dfid);
void _all_push(const Id &dfid, const DF &val);
void _all_destroy(const Id &dfid);

std::map<Id, DF> all_pushed_;
std::multimap<Id, std::function<void (const Id &, const DF &)>> all_waiters_;
std::set<Id> all_destroys_;
```

Рисунок 8 — сигнатуры новых методов класса rts

```
PROCEDURE массовая репликация фрагмента данных DF с идентификатором ID
      разослать всем узлам фрагмент данных ID со значением DF
      IF есть запрос на удаление ID THEN
            удалить запрос
            END
      IF фрагмента данных ID нет в контейнере THEN
            сохранить фрагмент данных ID со значением DF в контейнер
      FOR каждый ожидающий фрагмент данных ID фрагмент вычислений
            отправить фрагменту вычислений значение данных DF
END
```

Рисунок 9 — алгоритм массовой репликации (рассылка)

```
PROCEDURE получить реплицированный фрагмент данных ID
      IF фрагмент данных ID есть в контейнере THEN
            получить значение данных ID из контейнера
      ELSE пометить себя как ожидающий фрагмента данных ID фрагмент
вычислений
END
```

Рисунок 10 — алгоритм массовой репликации (прием)

3.1.3 Кэширование

Механизм кэширования был реализован аналогично массовой репликации. Соответствующий массовой репликации с кэшированием метод принимает не только предполагаемый к реплицированию фрагмент данных но и максимальное количество обращений к нему. При превышении количества обращений фрагмент данных удаляется с узла, освобождая память. Для реализации механизма был использован контейнер на узле с дополнительным полем для количества использований, а также были расширены методы, созданные для массовой репликации.

Класс cf был расширен методам all_push_cached, сигнатура которого представлена в листинге на рисунке 11, а класс rts — all_push_cached, _all_push_cached, а также полем all_pushed_cached_, сигнатуры которых представлены в листинге на рисунке 12. Псевдокод алгоритма представлен в листинге на рисунке 13.

```
void all_push_cached(const Id &dfid, const DF &val, unsigned int
exp_count);
```

Рисунок 11 — сигнатуры методов кэширования в cf

```
void all_push_cached(const Id &dfid, const DF &val, unsigned int exp_count);
void _all_push_cached(const Id &dfid, const DF &val, unsigned int exp_count);

std::map<Id, std::pair<DF, unsigned int>> all_pushed_cached_;
```

Рисунок 12 — сигнатуры методов кэширования в rts

```
PROCEDURE массовая репликация фрагмента данных DF с идентификатором ID с
кэшированием на N запросов
      разослать всем узлам данные ID со значением DF на N запросов
      IF фрагмента данных ID нет в контейнере THEN
            IF фрагмента данных ID нет в кэширующем контейнере THEN
                  сохранить данные ID со значением DF в кэширующий
контейнер на N запросов
      FOR каждый ожидающий данные ID фрагмент
            IF фрагменты данных есть в контейнере THEN
                  отправить фрагменту значение данных DF
            ELSE отправить фрагменту значения данных DF и снизить счетчик
запросов на 1
            IF счетчик запросов равен 0 THEN
                  удалить фрагмент из кэширующего контейнера
END
PROCEDURE получить реплицированные фрагмент данных ID
      IF фрагмент данных ID есть в контейнере THEN
            получить значение данных ID из контейнера
      ELSE IF данные ID есть в кэширующем контейнере THEN
            получить значение фрагмента данных ID из контейнера и снизить
      счетчик запросов на 1
            IF счетчик запросов равен 0 THEN
                  удалить фрагмент из кэширующего контейнера
      ELSE пометить себя как ожидающий данных ID фрагмент
END
```

Рисунок 13 — кэширование

3.1.4 Ограниченная репликация

Реализация ограниченной репликации, обладает существенным отличием от предыдущих реализаций. В массовой репликации поиск фрагмента данных для

нуждающегося фрагмента вычислений был тривиален, в отличие ограниченного варианта. Для реализации ограниченной вариации фрагмент вычислений и runtime-система были дополнены новыми методами.

Теперь фрагмент вычислений отправляет запрос к узлу не только с нужным фрагментом, но и со степенью репликации. Степень репликации представляет собой целое положительное число. Подходящими для репликации оказываются те узлы, чей ранг дает в остатке 0 при делении на степень репликации. Фрагмент данных реплицируется на все подходящие узлы, где так они хранятся в контейнере.

Для получения доступа к желаемым данным фрагмент вычислений отправляет запрос узлу, который либо предоставляет нужный фрагмент, либо отправляет запрос соседним узлам. Соседние узлы аналогично либо отправляют запрашиваемый фрагмент, либо продолжают цепочку запросов, сохраняя пришедший запрос в очереди и указывая количество передач. Запрос продолжает двигаться по линии процессорных элементов, пока не находит нужный фрагмент, или его счетчик не достигает степени репликации.

Класс cf был расширен методом limited_push, сигнатура которого представлена в листинге на рисунке 14, класс rts — limited_push и _limited_push, сигнатуры которых представлены в листинге на рисунке 15, а класс comm был расширен методом limited_bcast, сигнатура которого представлена в листинге на рисунке 16. Псевдокод алгоритма представлен в листинге на рисунке 17.

```
void limited_push(const Id &dfid, const DF &val, unsigned int degree);
```

Рисунок 14 — сигнатуры методов ограниченной репликации в cf

```
void limited_push(const Id &dfid, const DF &val, unsigned int degree);
void _limited_push(const Id &dfid, const DF &val, unsigned int degree);
```

Рисунок 15 — сигнатуры методов ограниченной репликации в rts

```
void limited_bcast(unsigned int degree, int tag, const void* buf,
        size_t size, std::function<void()> finisher);
```

Рисунок 16 — сигнатуры методов ограниченной репликации в comm

```
PROCEDURE ограниченная рассылка фрагмента данных ID со значением DF со
степенью репликации DEGREE
      FOR все узлы
            IF остаток от деления номера узла на DEGREE равен 0 THEN
                  отправить узлу фрагмент данных ID со значением DF
END


PROCEDURE ограниченная репликация фрагмента данных ID со значением DF со
степенью репликации DEGREE
      ограниченно разослать фрагмент данных ID со значением DF со степенью
репликации DEGREE
      IF есть запрос на удаление ID THEN
            удалить запрос
            END
      IF фрагмента данных ID нет в контейнере THEN
            сохранить фрагмент данных ID со значением DF в контейнер
      FOR каждый ожидающий фрагмент данных ID фрагмент вычислений
            отправить фрагменту вычислений значение данных DF
END

// для получения фрагментов, узлы пользуются либо expect_all_push, либо
// стандартным request, в зависимости от степени репликации
```

Рисунок 17 — алгоритм ограниченной репликации

Для вызова алгоритмов в коде могут быть использованы рекомендации LuNA. Рекомендации системы LuNA — это аннотации, при помощи которых разработчики может влиять на генерацию параллельной программы, сообщая компилятору необходимую информацию. Компилятор был расширен новыми

рекомендациями, которые указывают каким образом частные алгоритмы будут использоваться в программе. В листинге на рисунке 18 показан пример использования рекомендации для массовой репликации фрагмента данных “x”.

```
#!/user/bin/luna
import c_init(int, name) as init;
import c_check(int) as check;

sub main()
{
        df x;
        cf a: init(1, x) @ {
                  locator_cyclic: 1;
                  all_push x;
        };
        for i=2..100000 {
                check(x) @ {
                        locator_cyclic: i;
                        all_wait x;
                };
        }
}
```

Рисунок 18 — пример использования рекомендации и тестовая программа

3.2 Тестирование

Основной задачей введения частных алгоритмов являлось улучшение эффективности результирующих программ относительно существующих решений.

Тестирование проводилось на персональном компьютере: процессор Intel Core i5 7300HQ.

Для тестирования алгоритма массовой репликации использовался тест, приведенный в листинге на рисунке 18. В данном тесте большое количество фрагментов вычислений используют один и тот же фрагмент данных, присутствующий изначально только на одном узле. Подобная задача является

подходящий для применения алгоритмов повышения доступности данных и призвана показать принципиальную возможность ускорения программ с их помощью. Теоретически, вариант программы с использованием массовой репликации должен быть эффективнее варианта без него на нескольких процессорных элементах, особенно если коммуникации между ними занимают большое время.

Полученные результаты отображены в таблице 1.

Таблица 1 — результаты тестирования алгоритмов на ПК

	Без алгоритма	Массовая репликация	Ограниченная репликация (половина узлов)
1 процесс	4.110 с	3.810 с	4.482 с
4 процесса	8.477 с	5.965 с	6.968 с

Ожидаемые результаты подтвердились: уже на четырех процессах на четырех ядрах процессора заметна разница во времени исполнения программы с использованием различных алгоритмов.

Также тестирование проводилось на кластере — суперкомпьютере МВС-10П . Результаты тестирования отображены в таблице 2.

Таблица 2 — результаты тестирования алгоритмов на кластере

	Без алгоритма	Массовая репликация	Ограниченная репликация (половина узлов)
1 процесс	13.028 с	11.913 с	13.208 с
8 процессов	14.460 с	12.777 с	14.125 с

Похожий результат был получен и на кластере. Здесь, как и ранее, алгоритмы репликации смогли сократить время работы программы.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы были предложены частные алгоритмы управления распределенными данными. Алгоритмы были реализованы в runtime-системе LuNA. Часть реализованных алгоритмов была протестированы.

Тестирование показало, что алгоритмы позволяют повысить эффективность генерируемых системой LuNA программ.

Защищаемые положения:

1. Разработаны частные алгоритмы и механизмы массовой репликации, кэширования и ограниченной репликации.

2. Массовая репликация, кэширование и ограниченная репликация были реализованы в runtime-системе LuNA.

3. Массовая репликация была реализована в компиляторе системы LuNA.

4. Проведено исследование эффективности программ, использующих частные алгоритмы.

В приложении А представлено руководство программиста для реализованных алгоритмов.

В дальнейшем планируется реализовать оставшиеся алгоритмы в компиляторе LuNA. Основным направлением для развития является продолжение накопления частных алгоритмов управления распределенных данных в системе LuNA.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Масыч Матвей Алексеевич _____

(фамилия, имя, отчество студента) (подпись студента)

«27» мая 2021 г.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Hu, Y.F., Blake, R.J.: An Improved Diffusion Algorithm for Dynamic Load Balancing. J. Parallel Computing, vol. 25, no. 4, pp. 417–444 (1999).
2. Ken-ichiro Ishikawa. ASURA: Scalable and Uniform Data Distribution Algorithm for Storage Clusters. Computing Research Repository, abs/1309.7720 (2013).
3. Honicky, R.J., Miller E.L.: Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution. In: 18th International Parallel and Distributed Processing Symposium (2004).
4. Alicherry, M., Lakshman, T.V.: Network Aware Resource Allocation in Distributed Clouds. In: INFOCOM 2012, pp. 963–971. IEEE (2012).
5. Anderson, J.M., Lam, M.S.: Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In: ACM-SIGPLAN PLDI'93, pp. 112–125. ACM New York, USA (1993).
6. Malyshkin V.E., Perepelkin V.A., Schukin G.A.. 2015. Distributed Algorithm of Data Allocation in the Fragmented Programming System LuNA. Parallel Computing Technologies — 13th International Conference, PaCT 2015, Petrozavodsk, Russia, August 31 — September 4, 2015, Proceedings.
7. Э. Таненбаум. Архитектура компьютера. 5-ое издание. — СПб.: Питер, 2007. — С.86 ISBN 5-469-01274-3
8. Малышкин В. Э. Технология фрагментированного программирования //Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. – №. 46 (305).
9. Malyshkin V.E., Schukin G.A.. 2017. Distributed Algorithm of Dynamic Multidimensional Data Mapping on Multidimensional Multicomputer in the LuNA Fragmented Programming System. Parallel Computing Technologies — 14th International Conference, PaCT 2017, Nizhny Novgorod, Russia, September 4-8, 2017, Proceedings.
10. Тренин С. А. Разработка и реализация алгоритмов исполнения фрагментированных программ с заданным поведением // Тезисы XX Всероссийской конференции молодых учёных по математическому моделированию и информационным технологиям. – 2019. – С. 82-82.

ПРИЛОЖЕНИЕ А

АЛГОРИТМЫ ПОВЫШЕНИЯ ДОСТУПНОСТИ ДАННЫХ ДЛЯ RUNTIME-СИСТЕМЫ LuNA

Листов 6

Новосибирск 2021

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данном программном документе приведено руководство программиста для алгоритмов повышения доступности данных для runtime-системы “LuNA”. Исходным языком программы является C++. Средство разработки – редактор исходного кода Visual Studio Code от компании Microsoft. Основными функциями программы является репликация фрагментов данных. Оформление программного документа «Руководство программиста» произведено по требованиям ГОСТ 19.504-79 «ЕСПД. Руководство программиста» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

1 Назначение и условия применения программы

1.1 Назначение программы

Алгоритмы повышения доступности данных входят в состав runtime-системы LuNA. Система LuNA предназначена для исполнения фрагментированных программ на высокопроизводительных кластерах. Алгоритмы повышения доступности предназначены для оптимизации работы программы.

1.2 Функции, выполняемые программой

1. Расширение компиляции языка LuNA генерацией кода;
2. репликация данных по узлам;
3. удаление реплик данных.

1.3 Условия, необходимые для выполнения программы

Указание в коде фрагментированной программы соответствующих алгоритмам аннотаций (*all_push*, *all_wait*) или прямым указанием нужных методов в промежуточном представлении кода системы LuNA (*all_push*, *expect_all_push*, *limited_push*, *all_push_cached*).

2 Характеристика программы

2.1 Описание основных характеристик программы

Алгоритмы повышения доступности данных входят в состав комплексной runtime-системы LuNA. Алгоритмы обеспечивают репликацию данных, удаление реплик.

2.1.1 Средства контроля правильности выполнения программы

Контроль правильности работы алгоритмов осуществляется встроенными средствами runtime-системы: протоколирование событий, осуществление диагностики с помощью профилировщика системы, тестирования системы на этапе компиляции.

2.2 Описание основных особенностей программы

Данные алгоритмы повышения доступности данных осуществляют репликацию данных по узлам распределенной системы. Репликация может быть как массовой, так и частичной. Для удаления реплицированных данных может использоваться механизм ограничения количества потреблений данных (данные удаляются после заданного числа использований) или отдельная операция удаления всех реплик.

Алгоритмы вызываются при помощи аннотаций и действуют на указанные в аннотации данные.

3 Обращение к программе

3.1 Описание процедур вызова программы

Для выбора фрагментов для репликации требуется использовать одну из предлагаемых аннотаций. Пример использования аннотации представлен в листинге на рисунке А.1.

```
#!/user/bin/luna
import c_init(int, name) as init;
import c_check(int) as check;

sub main()
{
        df x;
        cf a: init(1, x) @ {
                  locator_cyclic: 1;
                  all_push x;
        };
        for i=2..100000 {
                check(x) @ {
                        locator_cyclic: i;
                        all_wait x;
                };
        }
}
```

Рисунок А.1 — пример использования аннотации

4 Входные и выходные данные

4.1 Организация используемой входной информации

4.1.1 Организация используемой входной информации на этапе компиляции

На этапе компиляции входной информацией является наличие аннотаций all_push, all_wait а также фрагменты данных, с которыми они были вызваны.

4.1.2 Организация используемой входной информации на этапе исполнения программы

На этапе исполнения программы входной информацией являются: идентификатор реплицируемого фрагмента данных, значение этого фрагмента, число использований фрагмента, степень репликации фрагмента, элементы перечислений *TAG_ALL_PUSH*, *TAG_ALL_DESTROY* и

TAG_ALL_PUSH_CACHED, получаемые путем приема сообщений от другого узла с помощью MPI и реплики фрагментов данных.

4.2 Организация используемой выходной информации

4.2.1 Организация используемой выходной информации на этапе компиляции

На этапе компиляции выходной информацией является сгенерированный во фрагменте вычислений исполняемый код.

4.2.2 Организация используемой выходной информации на этапе исполнения

На этапе исполнения программы выходной информацией является элементы перечислений *TAG_ALL_PUSH*, *TAG_ALL_DESTROY* и *TAG_ALL_PUSH_CACHED*, отправляемые путем рассылки сообщений другим узлам с помощью MPI и реплики фрагментов данных.