МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
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
Образовательная программа: 09.03.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМОВ ИСПОЛНЕНИЯ ФРАГМЕНТИРОВАННЫХ ПРОГРАММ С ЗАДАННЫМ ПОВЕДЕНИЕМ

(название темы прописными буквами)

утверждена распоряжением проректора по учебной работе №___ от « __»_____ 20__г.

скорректирована распоряжением проректора по учебной работе №___ от «___»_____20__г.

Тренин Станислав Александрович, группа 15205

(Фамилия, Имя, Отчество студента, группа) (подпись студента)

«К защите допущена»
Заведующий кафедрой,
д.т.н., проф.
Малышкин В. Э./……..
(ФИО) / (подпись)
«……»……20…г.

Руководитель ВКР
д.т.н., проф.
зав. каф., каф. ПВ ФИТ
Малышкин В. Э./……...
(ФИО) / (подпись)
«……»……20…г.

Дата защиты: «……»……20…г.

Новосибирск, 2019 г.

МИНОБРНАУКИ РОССИИ
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

УТВЕРЖДАЮ
Зав. кафедрой Малашкин В. Э.
(фамилия, И., О.)
…….
(подпись)
«……»……20…г.

ЗАДАНИЕ

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Тренину Станиславу Александровичу, группы 15205
(фамилия, имя, отчество, номер группы)

Тема разработка и реализация алгоритмов исполнения фрагментированных программ с заданным поведением
(полное название темы выпускной квалификационной работы бакалавра)

утверждена распоряжением проректора по учебной работе от……..№……

Срок сдачи студентом готовой работы……20.. г.

Исходные данные (или цель работы)..……

Структурные части работы средства управления поведением, реализация и экспериментальные исследование

Консультанты по разделам ВКР (при необходимости, с указанием разделов)

……
(раздел, ФИО)

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

Задание принял к исполнению
Тренин С. А./……..
(ФИО студента) / (подпись)
«…»……20…г.

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Фрагментированная программа (ФП) – это набор <DF, CF, in, out>, где DF – это множество фрагментов данных (ФД) переменных единственного присваивания; CF – это множество фрагментов вычислений (ФВ) операции единственного срабатывания. Для каждого a CF определены набор входных ФД in(a) и набор выходных ФД out(a). Выполнение ФВ состоит в вычислении его выходных ФД из значений входных ФД. ФВ реализуется некоторой функцией без побочных эффектов. Выполнение ФП заключается в выполнении всех его ФВ. Порядок выполнения ФВ в ФП основан на информационных зависимостях между ФВ. То есть, если ФД out1(a), определённые для ФВ 1, являются подмножеством ФД in2(a), определённым для ФВ 2, то ФВ 2 не может быть исполнен раньше ФВ 2.

Вычислительный узел (узел) – элемент мультикомпьютера.

Мультикомпьютер – вычислитель с распределённой памятью.

Поведение – множество всех вариантов исполнения фрагментированной программы, возможно, отличающихся различными параметрами, такими как порядок исполнения ФВ, распределение ФВ и ФД по узлам, стратегия сборки и удаления неиспользуемых ФД, стратегия балансировки нагрузки в вычислительных узлах – то, как программа может исполняться.

Ограниченное поведение – любое подмножество поведения.

Фиксированное поведение – любой элемент поведения.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В последние годы практически невозможно представить себе эффективную научную и вычислительную деятельность без применения высокоуровневого программирования, которое, в силу своей простоты, доступно гораздо более широкой аудитории, чем программирование на низком уровне, требующее специальной квалификации. В результате, при таком подходе, значительно расширяется круг тех, кто может пользоваться всеми преимуществами таких непростых технологий, как, например, параллельное программирование в научных исследованиях или для решения расчётных задач. При этом не требуется таких больших дополнительных затрат времени на непростое изучение низкоуровневых средств программирования или средств на найм профессионального программиста (которого зачастую нужно ещё вводить в суть проблемы).

В работе рассматриваются научные и вычислительные задачи, для которых в первую очередь важна производительность. Сколь бы простым в освоении и удобным ни было программное средство, если оно не способно давать производительность, сопоставимую с уже существующими средствами, смысла в его применении для данного класса задач будет мало.

Основная и фундаментальная проблема высокоуровневого программирования, вызывающая значительное падение производительности, – множество вариантов исполнения программы (поведение). При этом все варианты исполнения отличаются друг от друга нефункциональными свойствами, такими как производительность.

Проблема разнообразия и широты поведения усугубляется его непрозрачностью. Дело в том, что разные элементы поведения априори слабо различимы. При каждом конкретном запуске программы на исполнение невозможно в общем случае сказать, какими нефункциональными свойствами она будет обладать.

Таким образом, в силу проблемы выбора и реализации выбранного поведения, в высокоуровневом программировании зачастую трудно достигнуть хорошей производительности.

Выбор поведения может быть статическим, когда поведение жёстко задаётся до исполнения программы, динамическим, когда поведение выбирается уже в процессе, основываясь на текущем состоянии, или же смешанным.

Статический выбор поведения требует больших временных затрат (необходимо самому производить анализ и выбор для каждой конкретной задачи) и не закрывает всю широту решаемых задач. И динамический, и статический выбор поведения в высокоуровневых программах осложнён фундаментальными проблемами широты и непрозрачности поведения.

Для частичного преодоления непрозрачности поведения высокоуровневых программ, как правило, используют различные подсказки, эвристики, частные решения. Такой подход позволяет сузить предметную область, значительно ограничив поведение, или же переложить управление выбором отдельными элементами поведения на человека.

Таким образом, в высокоуровневом программировании существует ряд фундаментальных проблем, мешающих его полноценному применению. Поэтому без детального исследования и проработки вопросов задания и реализации поведения дальнейшее успешное развитие высокоуровневого программирования невозможно.

Проблема задания и реализации поведения в той или иной мере решается почти во всех высокоуровневых языках и системах программирования.

В таких run-time системах, как PaRSEC [1] и Task-based BLAS (TBLAS) [2, 3], проблема частично решается путём ориентации этих систем на исполнение алгоритмов линейной алгебры в распределённых гетерогенных системах.

Для системы параллельного программирования SMP Superscalar [4] и пакета ProActive Parallel Suite [5] задание поведения возможно при помощи управления приоритетами. Но такой подход, когда изменение приоритета операции влияет на общий порядок исполнения операций, позволяет описывать лишь не очень сложное поведение и, с увеличением сложности задачи, сильно возрастает вероятность ошибки управления.

Высокоуровневый язык Julia [6], в свою очередь, обладает средствами для задания параллелизма (множество готовых примитивов), но общее управление поведением в нём практически не развито. Исключением является возможность задания сетевой топологии, поддерживающейся экспериментально.

Нередко в функциональных языках и системах параллельного программирования вводятся языковые конструкции для задания порядка срабатывания операций, похожие на конструкции традиционного программирования для обозначения последовательного цикла (итерации исполняются друг за другом) и параллельного цикла (итерации исполняются параллельно). Примерами таких функциональных языков могут служить Haskel [7] и Sisal [8] или язык LISP [9], в котором реализованы операторы циклов for и while.

Кроме того, в высокоуровневом программировании часто бывает необходимо иметь возможность задать не только порядок исполнения, но и прямое управление коммуникациями. Например, с этой целью для системы Charm++ [10] был создан язык нотаций Charisma [11].

К сожалению, все дополнительные языковые конструкции, вводимые в высокоуровневые языки программирования, понижают уровень программирования, тем самым значительно увеличивая вероятность ошибки с ростом сложности задачи.

Такие системы, как LuNA [12], использующая технологию фрагментированного программирования, и Legion [13], с одной стороны,

специализированы на решении задач численного моделирования, а с другой – используют особые конструкции языка, при помощи которых можно задать ограниченное поведение программы. Одним из основных отличий LuNA от Legion является отсутствие системных алгоритмов в последнем (пользователь всегда должен задавать низкоуровневые алгоритмы, реализующие поведение).

В результате анализа можно выделить два основных подхода к решению проблемы задания и реализации поведения в высокоуровневых языках программирования:

- Введение в язык специальных операторов и директив, задающих прямое управление. При этом уровень программирования снижается.
- Сужение предметной области, на которую ориентирована система, в том числе за счёт эвристик.
- Иногда в системах применяют оба вышеуказанных подхода.

Дальнейшее развитие специализированных операторов и директив, задающих поведение в таких языках и системах, неизбежно ведёт к автоматизации их выбора, что повышает уровень программирования по сравнению с ручным выбором. Расширение же области применения специализированных языков, как видно на примере Julia, порождает в нём новые специализированные операторы. Как видно из обзора, проблема выбора наилучшего поведения в высокоуровневом программировании полностью не решена ни в одной из существующих систем и в целом недостаточно проработана.

В рамках данной работы в качестве подхода к программированию с высоким уровнем абстракции выбрано фрагментированное программирование. Для удобства тестирования был выбран язык автоматизации генерации фрагментированных программ LuNA, весомым достоинством которого также является вынесенное отдельно поведение (изменение поведения не затрагивает

функциональные свойства программы). Возможно это благодаря модели, в которой алгоритм не зависит от ресурсов. Другой особенностью языка LuNA являются специальные конструкции языка – рекомендации, помогающие в управлении поведением (они являются необязательными и могут отсутствовать в программе).

Целью данной работы является разработка средств для управления поведением программ и их поддержка системными алгоритмами для системы LuNA.

Для достижения поставленной в данной работе цели были сформулированы следующие задачи:

1. Необходимо проанализировать реализованные в системе LuNA способы управления поведением и определить недостающие.
2. Опираясь на используемую технологию фрагментированного программирования и существующую в языке LuNA систему рекомендаций, разработать алгоритмы, реализующие выбранные методы управления поведением.
3. Интегрировать в язык LuNA модули, реализующие выбранные алгоритмы так, чтобы они не нарушали существующую логику.
4. Провести тестирование добавленных модулей, выявить класс задач, для которого выбранные алгоритмы подходят наилучшим образом.

Научной новизной обладают языковые средства задания управления, дополняющие фрагментированное программирование, а также алгоритмы трансляции и компиляции.

Практическая значимость работы состоит в том, что был расширен класс эффективно исполняемых задач на языке LuNA, а для существующего класса эффективно исполняемых задач упростилось задание поведения.

Кроме того, выбранные средства также должны обладать низкими накладными расходами при исполнении, чтобы их можно было использовать в реальных задачах.

1. Средства управления поведением

1.1. Вопросы поведения в технологии фрагментированного программирования

В данной работе роль высокоуровневого средства программирования играет фрагментированное программирование. При таком подходе программа задаётся в виде фрагментированного алгоритма, на основе которого строится фрагментированная программа, способная исполняться несколькими способами, с возможностью частичного или полного управления её поведением.

В качестве средства автоматизации генерации фрагментированных программ в данной работе рассматривается язык LuNA. Вышеупомянутая идея подсказок здесь реализована через специальные структуры языка – рекомендации, которые можно добавить на стадии трансляции в язык C из фрагментированного алгоритма. При применении рекомендаций сгенерированная программа будет обладать ограниченным поведением, но всё ещё может быть исполнена многими способами, обладающими разной эффективностью. И сказать заранее, какими именно нефункциональными свойствами будет обладать тот или иной элемент поведения в конкретном случае, на данный момент не представляется возможным.

В рамках представленной работы была выполнена разработка и реализация алгоритмов исполнения фрагментированных программ с ограниченным поведением. Выделим основные направления, по которым будет осуществляться задание поведения. Во-первых, управление распределением фрагментов вычислений по вычислительным узлам (функция распределения) – возможность включать динамическую балансировку нагрузки, при которой недогруженные узлы будут получать работу с перегруженных. Во-вторых, фиксация управления (задание частичного порядка) – возможность строго

определять порядок исполнения фрагментов. В-третьих, возможность определения стратегии сборки мусора. В-четвёртых, управление выделением, освобождением и распределением ресурсов.

1.2. Динамическая балансировка нагрузки

В системе LuNA в данный момент динамическая балансировка нагрузки осуществляется только алгоритмом Rope of Beads [14, 15], который является алгоритмом диффузионного типа с дополнительно введённой линейной структурой. Суть данного алгоритма заключается в том, что в силу линейной упорядоченности множества ФВ, для каждого вычислительного узла определены границы вычисляющихся на нём ФВ таким образом, что все ФВ распределены по узлам. Периодически или по событию каждый узел определяет свою нагрузку. Затем происходит асинхронный обмен значениями нагрузки между соседними узлами (отношение соседства определяется в зависимости от топологии). Если разница пришедшего значения нагрузки и нагрузки на узле больше порогового значения, то инициируется балансировка, которая осуществляется путём сдвига границ вычислительного узла. Именно в силу того, что балансировка по алгоритму Rope of Beads осуществляется путём сдвига границ, данный алгоритм плохо подходит для устранения точечного дисбаланса (в рамках 1–2 вычислительных узлов), но хорошо выравнивает нагрузку в случае, когда многие узлы перегружены либо же недогружены.

Универсальных методов динамической балансировки нагрузки, одинаково хорошо устраняющих и точечный, и общий дисбаланс, не существует. Поэтому среди методов динамической балансировки нагрузки [16] было решено искать такой, который бы отличался по типу от Rope of Beads и устранял в первую очередь точечный дисбаланс. Было установлено, что среди найденных алгоритм динамической балансировки нагрузки WorkStealing [17] подходит для устранения именно точечного дисбаланса на вычислительных

узлах лучше, чем существующий в системе LuNA в данный момент алгоритм Rope of Beads. Именно поэтому было решено реализовывать данный алгоритм, закрывающий существующую в LuNA проблему. Суть алгоритма WorkStealing заключается в том, что с определённой периодичностью или по событию узлы проверяют свою загруженность и, если в результате проверки какой-либо узел оказывается недогружен, он отправляет запросы на получение работы на другие узлы. Когда узел получает запрос на выделение работы, он должен принять решение о её выделении, основываясь, в свою очередь, на собственной загруженности. Тем самым достигается выравнивание вычислительной нагрузки.

Особенностью балансировки по алгоритму WorkStealing является то, что её можно тонко настраивать, изменяя основные параметры: когда будет инициироваться проверка загрузки узла (по событию или же с определённой частотой), при какой нагрузке узел будет считаться перегруженным и, следовательно, будет инициировать балансировку (это может быть как фиксированное значение, так и вычисляемое в процессе исполнения программы, например, в зависимости от средней нагрузки в системе), какое количество работы будет передаваться при балансировке (необходимое для достижения установленного ранее порога, фиксированное значение или же какой-либо другой вариант).

Но данная особенность алгоритма WorkStealing приводит к ряду проблем. При таком подходе важно соблюсти баланс количества запросов. При недостаточной частоте обмена информацией (когда вычислительные узлы редко проверяют свою нагрузку, а следовательно, и редко инициируют балансировку) дисбаланс не будет устранён, а при избыточной частоте обмена информацией между узлами (когда только что передавший работу вычислительный узел проверяет свою нагрузку) появятся большие накладные расходы по времени, нивелирующие выгоду от ускорения. Кроме того, при

неудачном выборе параметров балансировки возможна ситуация, при которой работа непрерывно переходит между узлами, не успевая исполняться, тем самым образуя цикл.

Другим важным параметром, способным привести к снижению производительности при неумелом выборе, является объём передаваемой работы. Если он будет низок, то балансировка будет незначительной и существенного прироста производительности не произойдёт (кроме того, в силу накладных расходов на коммуникации выгода от такой балансировки может вовсе нивелироваться). Если же объём передаваемой в случае балансировки нагрузки будет велик, то узел-отправитель сам может стать недогруженным, а узел-получатель – перегруженным, что может привести к описанной выше ситуации зацикливания.

В рамках технологии фрагментированного программирования, работой выберем ФВ без побочных эффектов, для которого получены все необходимые входные ФД.

Таким образом, подбор параметров необходимо производить для каждой конкретной задачи, согласуясь с поставленными требованиями к результату. Одна комбинация параметров даст лучшую производительность, другая – меньшую нагрузку на сеть, третья – меньший объём вычислений.

В соответствии с вышесказанным, во избежание чрезмерных и недостаточных обменов информацией было решено адаптировать алгоритм WorkStealing к следующему виду:

1. Периодически или по событию узел проверяет свою загруженность работой.

a. Если на узле работа отсутствует, узел инициирует балансировку нагрузки, отправляя запросы на получение работы только своим соседям (отношение соседства

определяется в зависимости от топологии вычислительных узлов).

2. При получении запроса на выделение работы узел проверяет свою загруженность.

a. Если узел перегружен (число задач на узле превышает количество потоков исполнения), он выделяет работу по запросу. При этом будет передано такое количество работы, при котором узел-отправитель останется полностью загруженным (число задач будет равно количеству потоков исполнения). При этом, так как в системе LuNA передаются только те ФВ, для которых заданы все входные ФД, дополнительные затраты времени на ожидание входных ФД, необходимых для отправки работы на другой узел, отсутствуют.

1.3. Управление распределением ресурсов

На данный момент управление распределением ресурсов в run-time системе LuNA по умолчанию устроено следующим образом:

1. Изначально на одном из узлов существует один готовый к исполнению ФВ и все необходимые ему ФД (main). Как только он исполнится, будут порождены выходные ФВ и ФД, которые будут перераспределены по узлам run-time системой.

2. Когда ФВ оказывается на вычислительном узле, он проверяет наличие всех необходимых для исполнения входных ФД и, если не обнаруживает на данном узле каких-либо из них, рассылает запросы на получение необходимых входных ФД на прочие вычислительные узлы.

3. Когда для ФВ на узле собраны все необходимые входные ФД, он исполняется, порождая новые ФВ и ФД, которые затем будут разосланы по узлам.
4. ФВ удаляется после исполнения, а ФД – в соответствии со стратегией сборки мусора.

Такой подход универсален и позволяет выстроить чёткую иерархию распределения ресурсов, но при исполнении некоторых задач может привести к падению производительности. Например, в тех случаях, когда на одном вычислительном узле в процессе работы программы будут исполняться разные ФВ с одинаковыми (или частично совпадающими) ФД. Также возможен случай, когда выходные ФД одного ФВ совпадают с входными ФД другого ФВ. В ситуациях такого рода целесообразнее было бы не передавать и удалять ФД с вычислительного узла, а сохранять на нём.

Для преодоления этой проблемы было решено использовать существующую в языке LuNA систему рекомендаций. При добавлении рекомендации cached(x), где x – это кэшируемый ФД, и управление распределение ресурсов теперь выглядит следующим образом:

1. Изначально на одном из узлов существует один готовый к исполнению ФВ и все необходимые ему ФД (main). Как только он исполнится, будут порождены выходные ФВ и ФД, которые будут перераспределены по узлам run-time системой.
2. Когда ФВ оказывается на вычислительном узле, он проверяет наличие всех необходимых для исполнения входных ФД и, если не обнаруживает на данном узле каких-либо из них, рассылает запросы на получение необходимых входных ФД на прочие вычислительные узлы.
   a. Если была применена рекомендация cached, то поиск ФД также происходит в кэше узла (изначально он пуст).

3. Когда для ФВ на узле собраны все необходимые входные ФД, он исполняется, порождая новые ФВ и ФД, которые затем будут разосланы по узлам.
   a. Если была применена рекомендация cached, то все полученные ФД перед перераспределением по прочим узлам будут сохранены в кэш.
4. ФВ удаляется после исполнения, а ФД – в соответствии со стратегией сборки мусора.

В дополнение к кэшированию, решению проблемы потери производительности на пересылку информации может способствовать механизм перевычисления ФД вместо их посылки на вычислительный узел.

Это может быть существенно, если необходимые для вычисления очередного ФВ все или некоторые входные ФД имеют большой объём и, соответственно, на их пересылку с управляющего узла на вычислительный потребуется значительное время. При этом, если ФВ, вычисляющий все или часть требуемых другому ФВ, не требует входных ФД, а также сумма времени на пересылку такого ФВ и времени на его вычисление ниже, чем на пересылку вычисляемых им ФД, для производительности выгоднее будет не высылать часть или все ФД на вычислительный узел, а выслать вычисляющий их ФВ и сделать перевычисление на узле.

Для реализации этой идеи было также решено использовать существующую в языке LuNA систему рекомендаций. При добавлении рекомендации recalc(x, a), где x – это ФД, который будет перевычисляться, а – порождающий его ФВ. Для простоты на данном этапе будем считать, что перевычисление будет применяться только к тем ФД, которые являются единственными выходными для порождающих их ФВ и этот ФВ не имеет входных ФД. При применении данной рекомендации распределение ресурсов теперь будет иметь следующий вид:

1. Изначально на одном из узлов существует один готовый к исполнению ФВ и все необходимые ему ФД (main). Как только он исполнится, будут порождены выходные ФВ и ФД, которые будут перераспределены по узлам run-time системой.
2. Когда ФВ оказывается он вычислительном узле, он проверяет наличие всех необходимых для исполнения входных ФД и, если не обнаруживает на данном узле каких-либо из них, рассылает запросы на получение необходимых входных ФД на прочие вычислительные узлы.
   a. Если к какому-либо из запрашиваемых ФД применена рекомендация recalc, то вместо него будет отправлен и вычислен порождающий его ФВ.
3. Когда для ФВ на узле собраны все необходимые входные ФД, он исполняется, порождая новые ФВ и ФД, которые затем будут разосланы по узлам в соответствии с формулой.
4. ФВ удаляется после исполнения, а ФД – в соответствии со стратегией сборки мусора.

1.4. Порядок исполнения фрагментов вычислений

В ряде вычислительных задач существенным может оказаться порядок исполнения ФВ на вычислительных узлах. На текущий момент в системе LuNA управляющий узел отправляет ФВ на свободные вычислительные узлы по готовности необходимых для них входных ФД. Но в некоторых случаях такой универсальный подход может привести к тому, что при исполнении задачи скапливается множество ФД, необходимых для дальнейших вычислений, которые, при другой последовательности исполнения ФВ, могли бы больше не понадобиться и быть утилизированы сборщиком мусора.

Наглядным примером такой задачи является умножение матриц. Как известно, умножение происходит по формуле

$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \, (i = 1, 2, ...l; j = 1, 2, ...n)$ для матриц A и B размерностей $l \times m$ и $m \times n$ соответственно. В соответствии с этим все необходимые для умножения ФВ можно разделить на две категории:

1. Порождающие ФД (те, которые находят произведение элементов исходной матрицы).
2. Поглощающие ФД (те, которые редуцируют полученные ранее произведения в элементы конечной матрицы C).

При стандартном порядке действий в программе, решающей задачу умножения матриц, сначала чаще будут исполняться порождающие ФВ (так как все необходимые им входные ФД уже есть в системе), а потом, по мере вычисления выходных ФД из этих ФВ, будут начинать исполняться поглощающие ФД. Таким образом, поглощающие ФВ будут простаивать без работы, в ожидании пока отработают все порождающие, тем самым снижая производительность.

Эту проблему можно разрешить, если ввести систему приоритетов, реализованную через существующие в языке LuNA рекомендации. При таком подходе каждому ФВ задаётся численное значение приоритета. По умолчанию это значение для каждого ФВ равно 0. Установить приоритет в любое значение можно перед генерацией программы при помощи рекомендации prior(a, n), где a – это ФВ, для которого задаётся приоритет, $n \in Z$ – его приоритет. Стоит отметить, что проверка приоритетов производится только в рамках вычислительного узла, так как при межузловой проверке приоритетов появившиеся накладные расходы на обмен информацией нивелируют прирост производительности.

Таким образом, если на вычислительном узле готовы к исполнению несколько ФВ (для них получены все необходимые входные ФД), для одного

или нескольких из которых включена рекомендация prior, то первым будет исполнен ФВ с наивысшим приоритетом.

Другим вариантом решения проблемы управления приоритетами исполнения ФВ может стать строгое задание порядка исполнения пользователем. В этом случае пользователь должен быть уверен, что для той части ФВ, где он устанавливает строгий порядок исполнения, соблюдены все информационные зависимости. В противном случае произойдёт падение производительности за счёт того, что ФВ, которые должны выполняться в соответствии с установленным пользователем порядком, не могут быть исполнены, так как необходимые им входные ФД ещё не вычислены.

Реализовать исполнение ФВ в строгом порядке было решено при помощи существующей в системе LuNA системы рекомендаций и того свойства ФП, что ФВ связаны информационными зависимостями. При таком подходе вводится рекомендация rank(a, b), где a и b – это ФВ, для которых необходимо установить строгий порядок. Данная рекомендация означает, что ФВ a будет исполнен раньше, чем ФВ b. Достигается это за счёт введения фиктивного ФД, вырабатываемого ФВ a и принимаемого ФВ b. Таким образом, ФВ b не сможет быть исполнен до ФВ a, так как они связаны информационными зависимостями.

2. Реализация и экспериментальные исследования

2.1. Устройство системы LuNA

Для лучшего понимания того, как были реализованы предложенные решения, покажем, как происходит генерация параллельной программы на языке LuNA:

1. Парсер генерирует параллельную программу в соответствии с описанием на языке LuNA, используя в качестве ФВ реализацию модулей на языке C. На этом же этапе в программу на языке LuNA можно дописать рекомендации, призванные ограничить поведение.
2. Из LuNA программы и блоков на языке C генерируется файл в формате JSON. По существующему алгоритму считываются рекомендации и их параметры.
3. Транслятор из JSON файла генеририрует байткод (byte-code).
4. Полученный байткод интерпретируется run-time системой LuNA.

2.2. Реализация предложенного решения

Для реализации балансировки динамической нагрузки по алгоритму WorkStealing в язык LuNA была введена возможность включения балансировки путём выставления рекомендации stealable в LuNA программе для тех задач, которые могут участвовать в балансировке. Ниже приведён листинг 1 на языке LuNA, где фрагменты, вычисляющие метод poi_part, помечены как stealable включенной рекомендацией stealable.

```
for i=1..$FG_COUNT-2 {
    poi_part(Ro[i], Fi[t][i], FiD[t][i-1], FiU[t][i+1], i,
                    Fi[t+1][i], FiU[t+1][i], FiD[t+1][i], lmax[t+1][i])
        -->(Fi[t][i], FiD[t][i-1], FiU[t][i+1]) @ {
        request Ro[i], Fi[t][i], FiD[t][i-1], FiU[t][i+1];
```

```
            locator_cyclic:i/11;
            //req_unlimited Fi[t+1][i], FiU[t+1][i], FiD[t+1][i], lmax[t+1][i];
        };
} @ {
        locator_cyclic:0;
        unroll_at_once;
        stealable;
}
```

Листинг 1

В код парсера был добавлен модуль, выставляющий константу (флаг stealable), отвечающую за возможность балансировки для задач, к которым на предыдущем этапе была применена рекомендация (инструмент, позволяющий определить, какие рекомендации и к чему были применены на предыдущем шаге, уже существовал в LuNA).Таким образом, в JSON файле во фрагменте, описывающем poi_part, будет выставлен соответствующий флаг, как видно на листинге 2.

```
 "flags": [
            "stealable"
            ],
            "ruletype": "flags",
            "type": "rule"
```

Листинг 2

В результате этого в исполнимый файл перед описанием каждого фрагмента, для которого выставлен флаг stealable, будут добавлена директива CHECK_STEAL.

Кроме того, были добавлены следующие модули на языках C и Python, подключающиеся при наличии вышеуказанного флага:

1. Модуль, инициирующий проверку вычислительными узлами их текущей загрузки с заданной частотой.
2. Модуль, осуществляющий проверку отношения числа задач на узле к числу его потоков исполнения. В соответствии с пунктом 2а выбранного алгоритма, если это число больше единицы и узел получил запрос на выделение работы, тогда запрошенная работа выделяется. Если это число равно нулю и узел инициировал проверку, то инициируется балансировка на данном узле.
3. Модуль, передающий запросы на получение работы в соседние узлы.
4. Добавлена возможность передавать работу по запросу к уже существующему в LuNA модулю, ответственному за распределение работы по узлам.

На рисунке 1 схематично представлено взаимодействие модулей балансировки нагрузки в окрестности одного вычислительном узле.

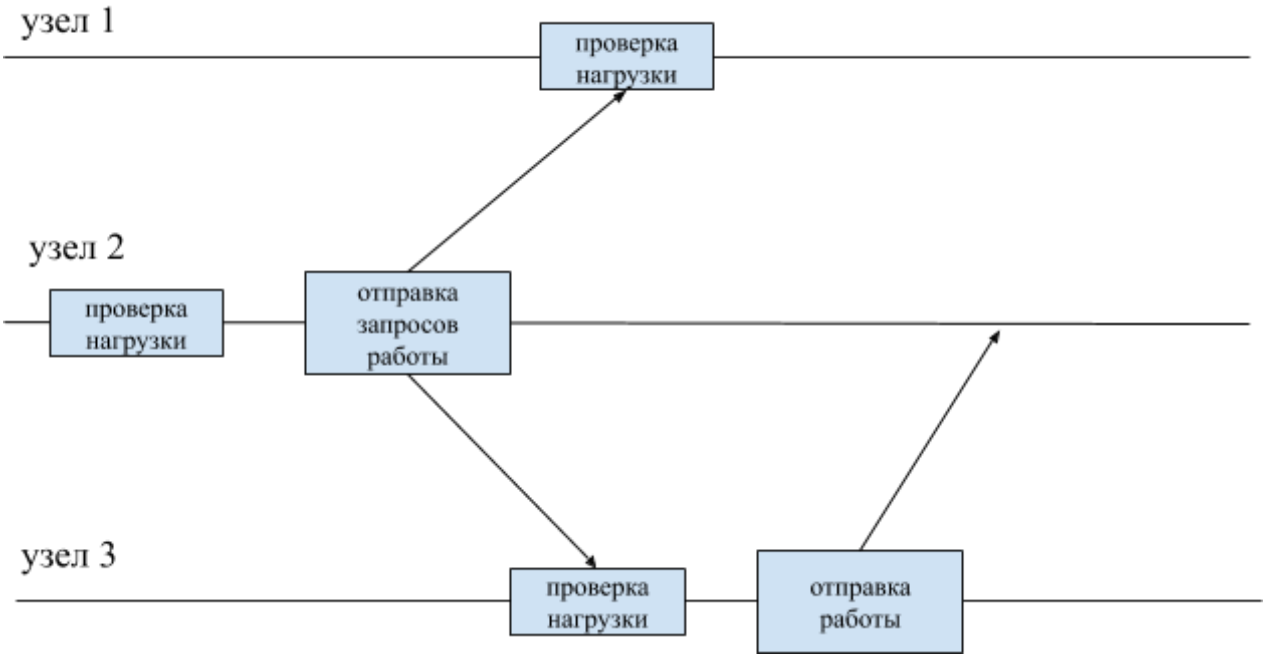

Рисунок 1 —взаимодействие модулей балансировки нагрузки в окрестности одного вычислительного узла

Для реализации кэширования была введена возможность его включения путём выставления рекомендации cached в LuNA-программе. Ниже приведён листинг 3 на языке LuNA, где все вычислительные узлы помечены как cached в момент инициализации при помощи рекомендацией cached.

```
sub f() {
        df x, y;
        // ...sub body...
} @ {
        cached x;
}
```

Листинг 3

В код парсера был добавлен модуль, выставляющий константу (флаг cached), отвечающую за возможность кэширования для ФД на узлах, к которым на предыдущем этапе была применена рекомендация (инструмент, позволяющий определить, какие рекомендации и к чему были применены на предыдущем шаге, уже существовал в LuNA). Таким образом, в JSON для узла будет выставлен соответствующий флаг, как видно на листинге 4.

```
"flags": [
        "cached"
        ],
        "ruletype": "flags",
        "type": "rule"
```

Листинг 4

В результате этого в исполнимый файл перед описанием каждого фрагмента, для которого выставлен флаг stealable, будут добавлена директива CHECK_CACHE.

Для реализации перевычисления была введена возможность его включения путём выставления рекомендации recalc в LuNA-программе. Ниже приведён листинг 5 на языке LuNA, где фрагменты вычислений помечены для перевычисления.

```
for i=1..$FG_COUNT-2 {
	poi_part(Ro[i], Fi[t][i], FiD[t][i-1], FiU[t][i+1], i,
			Fi[t+1][i], FiU[t+1][i], FiD[t+1][i], lmax[t+1][i])
		-->(Fi[t][i], FiD[t][i-1], FiU[t][i+1]) @ {
		request Ro[i], Fi[t][i], FiD[t][i-1], FiU[t][i+1];
		locator_cyclic:i/11;
		//req_unlimited Fi[t+1][i], FiU[t+1][i], FiD[t+1][i], lmax[t+1][i];
	};
} @ {
	locator_cyclic:0;
	unroll_at_once;
	recalc Ro[i];

}
```

Листинг 5

В код парсера был добавлен модуль, выставляющий константу (флаг cached), отвечающую за возможность кэширования для ФД на узлах, к которым на предыдущем этапе была применена рекомендация (инструмент, позволяющий определить, какие рекомендации и к чему были применены на предыдущем шаге, уже существовал в LuNA).Таким образом, в JSON для соответствующих ФД будет выставлен соответствующий флаг, как видно на листинге 6.

```
"flags": [
        "recalc"
        ],
        "ruletype": "flags",
        "type": "rule"
```

Листинг 6

В результате этого в исполнимый файл перед описанием каждого фрагмента, для которого выставлен флаг recalc, будет добавлена директива RECALC.

Для реализации приоритетов была введена возможность их выставления путём написания рекомендации prior в LuNA-программе. Ниже приведён листинг 7 на языке LuNA, где для части ФВ установлен приоритет 10.

```
for i=7..$FG_COUNT-7 {
    poi_part(Ro[i], Fi[t][i], FiD[t][i-1], FiU[t][i+1], i,
                Fi[t+1][i], FiU[t+1][i], FiD[t+1][i], lmax[t+1][i])
        -->(Fi[t][i], FiD[t][i-1], FiU[t][i+1]) @ {
        request Ro[i], Fi[t][i], FiD[t][i-1], FiU[t][i+1];
        locator_cyclic:i/11;
        //req_unlimited Fi[t+1][i], FiU[t+1][i], FiD[t+1][i], lmax[t+1][i];
    };
} @ {
    locator_cyclic:0;
    unroll_at_once;
    prior 10;

}
```

Листинг 7

В код парсера был добавлен модуль, выставляющий константу (флаг prior), отвечающую за назначение приоритета для ФВ, к которым на предыдущем этапе была применена рекомендация (инструмент, позволяющий определить, какие рекомендации и к чему были применены на предыдущем шаге, уже существовал в LuNA).Таким образом, в JSON для соответствующих ФВ будет выставлен соответствующий флаг, как видно на листинге 8.

```
"flags": [
    "recalc"
    ],
    "ruletype": "flags",
    "type": "rule"
```

Листинг 8

В результате этого в исполняемый файл перед описанием каждого фрагмента, для которого выставлен флаг prior, будет добавлена директива CHECK_PRIOR.

2.3. Эксперименты

Для адекватного исследования выбранной динамической балансировки нагрузки по алгоритму WorkStealing необходима репрезентативная задача с заведомым точечным дисбалансом и возможностью настройки объёма производимых вычислений. В соответствии с данными требованиями была выбрана программа на языке LuNA, решающая модельную задачу, реализующую итерационный метод с параметризованным объёмом вычислений в узлах сетки. Целью данного эксперимента является определение того, насколько хороши выбранные нами параметры балансировки и выделение классов задач, для которых производительность уменьшилась, увеличилась или осталась неизменной.

Чтобы наглядно продемонстрировать эффект динамической балансировки, один из узлов изначально должен быть не загружен работой. В серии тестов задача запускается для разного числа циклов исполнения на узле с включённой динамической балансировкой и без неё, производятся замеры времени. Полученное время сравнивается с идеальным временем исполнения такой задачи. Идеальным (теоретическим) временем работы программы в данном случае считается такое, какое было бы получено при постоянной, полной и равномерной загрузке всех вычислительных узлов работой. Если полученное время с включённой балансировкой меньше, чем без неё, значит, удалось добиться роста производительности. И чем ближе полученное время к идеальному, тем лучше прошла балансировка.

2.4. Результаты экспериментов

Для тестирования балансировки динамической нагрузки по алгоритму WorkStealing запуск выбранной тестовой задачи производился на кластере МВС-10П ОП Межведомственного Суперкомпьютерного Центра [18]. Чтобы облегчить обработку результатов и дальнейшее профилирование, было решено запускать модельную задачу на небольшом числе узлов (4 вычислительных узла с 2 потоками исполнения на каждом).

Результаты тестирования приведены в таблице 1. Как можно видеть, в данный момент реализованный алгоритм WorkStealing в рамках данной задачи лучше всего показывает себя при большом объёме вычислений в узлах сетки. При увеличении объёма вычислений в 1000 раз был получен результат, близкий к теоретическому (когда все узлы всё время загружены равномерно). Этот результат даже превзошёл теоретический, но в целом находится в пределах погрешности проводимых измерений.

Сравнение производительности — таблица 1

Использование Work Stealing	Циклов исполнения задачи	Размер одного фрагмента вычислений	Общее число фрагментов	Время работы/ идеальное время работы
нет	1	8x64×960	32	333.081413 sec/ 249.75 sec
да	1	8x64×960	32	333.081413 sec/ 249.75 sec
нет	250	8x64×960	32	104.544873 sec/ 78.00 sec
да	250	8x64×960	32	85.281108 sec/ 78.00 sec
нет	1000	16x64×960	32	310.256728 sec/ 232.50 sec
да	1000	16x64×960	32	228.624841 sec/ 232.50 sec

Результаты той же серии тестов более наглядно представлены ниже в виде графиков. На рисунке 2 можно видеть время исполнения программы для случаев с динамической балансировкой нагрузки по алгоритму WorkStealing, без балансировки и для идеального распределения задач по узлам, когда все узлы нагружены постоянно и равномерно. Время по оси ординат указано в секундах.

На рисунке 3 показано соотношение теоретического времени работы программы к реальному для случаев с динамической балансировкой нагрузки по методу WorkStealing и без неё. Чем выше значение по оси ординат, тем лучше сбалансирована нагрузка и тем выше производительность.

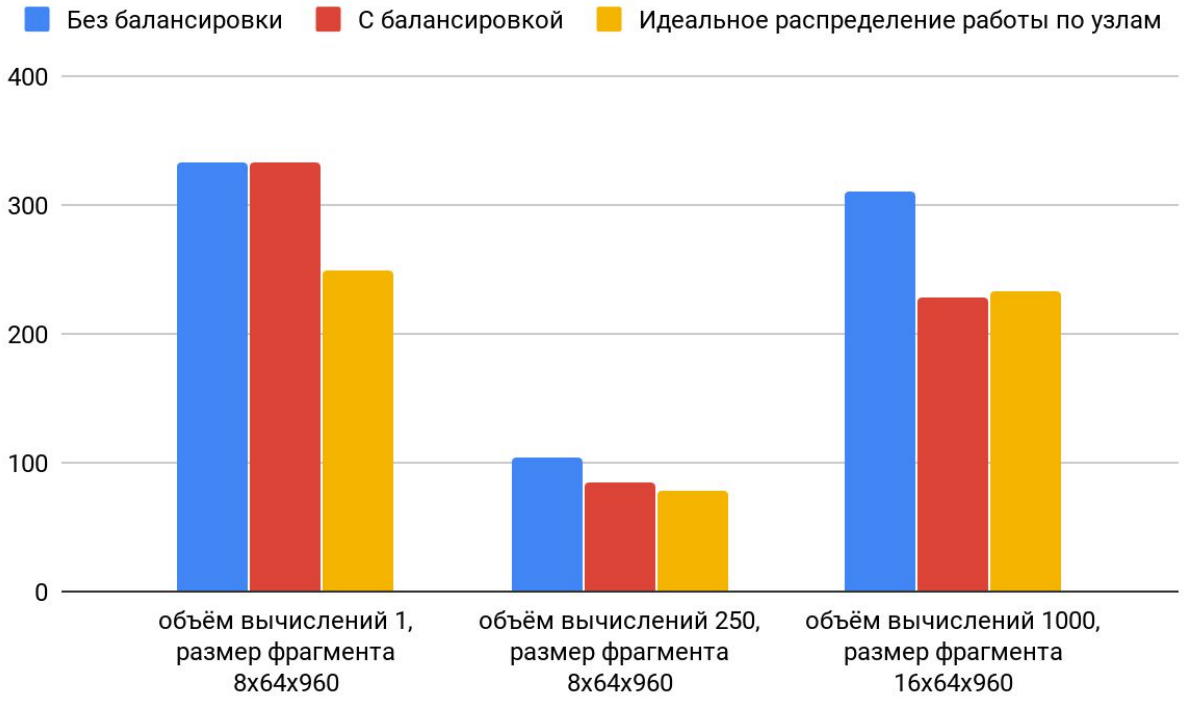

Рисунок 2 — время работы программы

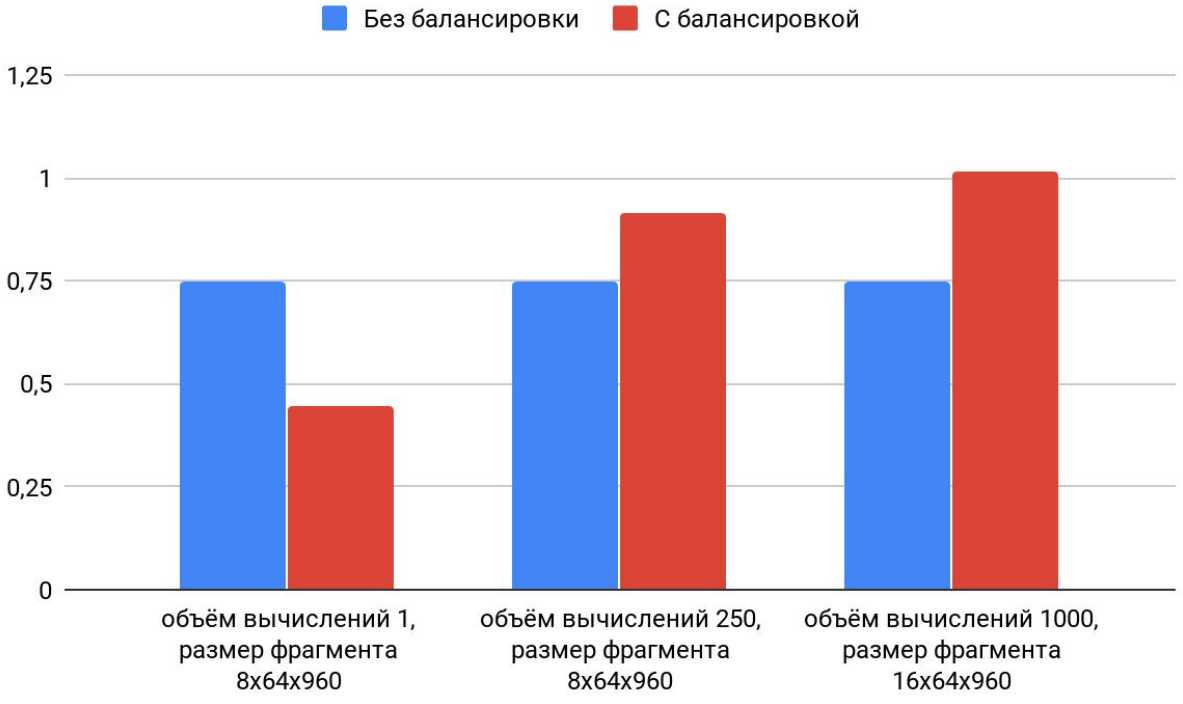

Рисунок 3 — Отношение времени работы к идеальному (теоретическому) времени работы

ЗАКЛЮЧЕНИЕ

В результате проведённой работы были предложены средства управления поведением во фрагментированных программах.

Для выбранных средств были разработаны алгоритмы. Часть алгоритмов была реализована в качестве модулей для системы LuNA. Для разработанных модулей, в свою очередь, было проведено исследование работы на модельной задаче с различными параметрами.

Защищаемые положения:

1. Разработаны средства управления поведением во фрагментированных программах.
2. Разработаны алгоритмы, реализующие выбранные средства в системе LuNA.
3. Часть алгоритмов реализована в виде модулей системы LuNA.
4. Проведено экспериментальное исследование реализованных алгоритмов.

В дальнейшем планируется программная реализация оставшихся алгоритмов. Основным же направлением дальнейшей работы является разработка новых средств управления поведением и поддерживающих их алгоритмов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J. DAGuE: A Generic Distributed DAG Engine for High Performance Computing // Proceedings of the Workshops of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2011 Workshops), IEEE, Anchorage, Alaska, USA, 1151-1158, 16-20 May, 2011.

2. Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems // InSC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. New York, NY, USA, 2009. P. 1-11.

3. Fengguang Song. Static and dynamic scheduling for effective use of multicore systems. PhD thesis, University of Tennessee, 2009.

4. Barcelona Supercomputing Center. SMP Superscalar (SMPSs) User's Manual, Version 2.2. – Режим доступа: http://www.bsc.es/media/3576.pdf(2008)

5. Caromel D., Leyton M. Proactive Parallel Suite: from active objects-skeletons-components to environment and deployment // Euro-Par 2008 Workshops –Parallel processing, 2008. P. 423-437.

6. Julia – Режим доступа: https://julialang.org/

7. Coutts D., Loeh A. Deterministic parallel programming with Haskell // Comput. Sci. Eng. 14 (6), 2012. P. 36-43.

8. Gaudiot J.-L., Deboni T., Feo J., at all. The Sisal project: real world function programming // LNCS, vol. 1808, Springer, 2001. P. 84-72.

9. Seibel P. Practical Common LISP, APRESS, 2005.

10. Miller P. Productive parallel programming with Charm++ //Proceedings of the Symposium on High Performance Computing, 2015. P. 241-242.

11. Chao Huang, Laxmikant V. Kale Charisma. Orchestrating Migratable Parallel Objects // Proceedings of the 16thInternational Symposium on High Performance Distributed Computing (HPDC), 2007. P. 75-84.

12. Malyshkin V.E., Perepelkin V.A. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem // Parallel Computing Technologies. 2011. P. 53-61.

13. Legion: Expressing Locality and Independence with Logical Region – Режим доступа: https://legion.stanford.edu/pdfs/sc2012.pdf – Загл. с экрана.

14. Malyshkin V.E., Schukin G.A. 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.

15. Malyshkin V.E., Perepelkin V.A., Schukin G.A. Scalable Distributed Data Allocation in LuNA Fragmented Programming System // Journal of Supercomputing, S.I.: Parallel Computing Technologies, Springer, 2016.

16. Иванисенко И., Кириченко Л., Радивилова Т. Методы балансировки с учётом мультифрактальных свойств нагрузки // Information Content and Processing. 2015. Vol. 2, № 4. P. 345-362.

17. Cederman D., Tsigas Ph. Dynamic Load Balancing Using Work-Stealing // GPU Computing Gems Jade Edition. 2011. P. 485–500.

18. Режим доступа: www.jscc.ru

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».

_____ _____

ФИО студента Подпись студента

« ____ »_____20 __г.