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 КВАЛИФИКЦИОННАЯ РАБОТА БАКАЛАВРА

Лямина Артёма Сергеевича

Тема работы:

РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМА РАСПРЕДЕЛЕНИЯ РЕСУРСОВ ФРАГМЕНТИРОВАННЫХ ПРОГРАММ НА ОСНОВЕ ПРОФИЛИРОВАНИЯ

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

Руководитель ВКР
к.т.н., доцент
доцент каф. ПВ ФИТ НГУ
Маркова В.П. /……...
(ФИО)/ (подпись)
«26»…мая……2021г.

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

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…2020г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студент*у(ке)*……Лямину Артёму Сергеевичу…, группы…17202……
(фамилия, имя, отчество, номер группы)

Тема… Разработка и реализация алгоритма распределения ресурсов фрагментированных программ на основе профилирования…
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 17.12.2020 № 0451

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 алгоритм распределения ресурсов фрагментированных программ на основе профилирования

Структурные части работы:

обзор литературы, постановка задачи, разработка и реализация алгоритма, тестирование

Руководитель ВКР
доцент. каф. ПВ ФИТ НГУ,
к.т.н, доцент
Маркова В.П. /……..
(ФИО)/(подпись)
«17»…декабря…2020г.

Задание принял к исполнению
Лямин А.С. /……..
(ФИО)/(подпись)
«17»…декабря…2020г.

Соруководитель
ст. преп. каф. ПВ ФИТ
Перепёлкин В.А. / ……..
(ФИО)/(подпись)
«17»…декабря…2020г.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Величина квантования – размер отрезка, на которые разбивается всё время работы программы.

Вычислительный узел – узел мультикомпьютера.

Дисбаланс нагрузки – состояние вычислительной системы в конкретный момент времени, при котором отдельные вычислительные узлы более нагружены, чем другие.

Квантовый промежуток – полуинтервал, содержащий в себе отметки времени, относящиеся к кванту.

Квантованный вес фрагмента вычислений – длина пересечения отрезка исполнения фрагмента вычислений и квантового промежутка.

Квантованная нагрузка на узел – суммарный вес квантованных весов фрагментов вычислений в заданный квант.

Профилирование – форма динамического анализа программы, при которой фиксируются некоторые важные особенности поведения программы, такие как продолжительность операций и коммуникаций, дисбаланс загрузки процессоров и др.

Профиль программы – статистика, собранная при профилировании.

Фрагмент вычислений – независимая единица программы, содержащая описание входных/выходных фрагментов данных и кода (модуля, процедуры) фрагмента [2, с. 46].

Фрагмент данных – агрегат из переменных [10, с. 58].

Фрагментированная программа – рекурсивно перечислимое множество фрагментов вычислений и их входных/выходных фрагментов данных [2, с. 46].

ВВЕДЕНИЕ

В современном мире существует тенденция повышения уровня языков программирования. На ряду с этим возникают системы автоматического конструирования программ, которые позволяют создавать программы по некоторому высокоуровневому описанию, не вдаваясь в детали низкоуровневого программирования.

Преимуществом таких систем является уменьшение трудозатрат на написание программы, так как в процессе традиционного программирования необходимо не только написать алгоритм, но и проверить правильность взаимодействий между отдельными частями программы, а это, порой, занимает больше времени, чем сам процесс написания.

Особенно сложным является автоматическое конструирование параллельных программ, так как необходимо обеспечить взаимодействие между отдельными процессами.

Важную роль в конструировании программ играют алгоритмы балансировки нагрузки, сборки мусора и др. Поэтому эффективность таких систем напрямую зависит от эффективности данных алгоритмов, в частности, от алгоритмов балансировки нагрузки. Под эффективностью подразумевается эффективность по времени, то есть чем эффективнее алгоритм, тем меньшее время исполнения программы достигается.

Основной проблемой балансировки является то, что не существует универсального алгоритма, дающего максимальную производительность для всех программ. Поэтому алгоритмы автоматической балансировки нагрузки обычно дают результат хуже, чем при ручном программировании. Но использование алгоритмов автоматической балансировки нагрузки оправдано, так как в настоящее время их эффективность для многих классов задач не сильно уступает ручной балансировке, но сильно экономит время программиста.

Алгоритмы балансировки нагрузки можно условно разделить на статические, динамические и на основе профилирования.

Статические алгоритмы балансировки задают распределение на этапе подготовки к исполнению. Такой подход не нагружает систему во время исполнения, но спланировать нагрузку не всегда возможно, так как она может зависеть от входных параметров, а также от хода вычислений, и, как следствие, не может гарантировать, что распределение является наилучшим.

Динамические алгоритмы работают во время исполнения. Зная нагрузку в определённый момент времени, алгоритм перераспределяет задачи между узлами так, чтобы она была более равномерной. Но недостатком таких алгоритмов является наличие дополнительных коммуникаций между узлами, что уменьшает производительность вычислительной системы.

Алгоритмы на основе профилирования используют преимущества двух предыдущих подходов – работа по созданию распределения происходит на подготовительном этапе, но использует опыт предыдущих запусков программы с такими же параметрами, что позволяет узнать сложность всех задач заранее и распределить их так, чтобы нагрузка на узлы была равномерной. Но существует и ряд минусов. Профиль программы связан с входными данными. При запуске программы с другими входными данными характеристики могут значительно поменяться. С другой стороны, профиль также зависит и от конфигурации вычислительной системы. Это означает, что профиль, полученный на одной вычислительной системе непригоден для другой. Несмотря на все изложенные недостатки, балансировка на основе профилирования является востребованной, так как в задачах вычислительной математики необходимо неоднократно запускать одну и ту же программу, при этом изменение входных параметров не всегда сильно сказывается на профиле.

Таким образом, существует потенциал в разработке и реализации алгоритмов автоматической балансировки нагрузки, использующих профиль программы.

Цель работы – разработка и реализация алгоритмов автоматической балансировки нагрузки на основе профилирования.

Задачи:

1. Анализ систем автоматического конструирования параллельных программ и их средств профилировки, возможности применения алгоритма;
2. Разработка модели профиля программы;
3. Разработка алгоритма автоматической балансировки нагрузки;
4. Программная реализация;
5. Экспериментальное исследование алгоритма;

Научная новизна состоит в том, что разработан алгоритм автоматической балансировки нагрузки на основе профилирования, а также решены необходимые научные задачи для исполнения фрагментированных программ с использованием разработанного алгоритма.

Практическая ценность заключается в том, что реализованный алгоритм балансировки нагрузки на основе профилирования повысил эффективность исполнения фрагментированных программ.

Настоящая работа состоит из введения, четырёх глав и заключения. Во введении описывается проблема автоматического распределения нагрузки, актуальность проблемы, цель работы, практическая значимость и научная новизна. В первой главе рассматриваются необходимые требования систем для использования в них балансировки на основе профилирования. Во второй главе предлагается алгоритм автоматического балансирования нагрузки на основе профилирования. В третьей главе описываются детали реализации. В четвёртой главе приведены результаты тестирования.

1 Анализ предметной области

Чтобы в систему было возможно встроить балансировщик нагрузки на основе профилирования, необходимо:

- Наличие средств профилирования или возможность их добавления;
- Возможность распределять задачи, не меняя логику программы, используя при этом результат работы алгоритма автоматической балансировки нагрузки на основе профилирования;
- Возможность анализировать производительность программы.

На основе вышеизложенных критериев произведён обзор систем автоматического конструирования параллельных программ.

1.1 Обзор систем автоматического конструирования параллельных программ

Charm++ – task based система [8]. Charm++ использует в качестве основы язык C++. Программа в Charm++ представлена множеством структурных единиц, называемых чарами (chare). Исполнение программы Charm++ происходит в runtime системе [9]. Runtime-система управляет порядком исполнения чаров, а также использует динамический балансировщик нагрузки при распределении чаров между вычислительными узлами мультикомпьютера. Для итеративных алгоритмов (например серия временных шагов, исполняющаяся до сходимости) динамический балансировщик в Charm++ использует «принцип постоянства» («principle of persistence»). Он описывается в п. 2.2.6 «Load balancing» в официальной документации к Charm++ [13]. Его суть заключается в том, что нагрузка на узлы и взаимодействия между чарами имеет тенденцию сохраняться. Поэтому динамический балансировщик, в некотором роде, использует профилирование, чтобы предсказывать будущие нагрузки.

Система обладает средствами сбора информации о загруженности процессоров и о коммуникациях между чарами, п. 2.4.1 «Tuning and Developing Load Balancers» в официальной документации к Charm++ [13]. Также она

предоставляет функционал по подбору стратегии балансировки нагрузки по профилировочной информации исполнявшейся программы.

Система PaRSEC [7] берёт за основу язык C. PaRSEC анализирует последовательный алгоритм и строит для него бесконтурный граф вычисления. Исполнение происходит в runtime-системе. Балансировка нагрузки происходит статически по узлам мультикомпьютера, а во время исполнения runtime-система позволяет динамически распределять нагрузку [7, 11]. Основным недостатком этой системы является ограниченность применимости классом задач типа линейной алгебры. В PaRSEC реализован функционал по получению профиля программы и его визуализации [14].

Система Legion [6] использует в качестве основы язык C++. В ней вводится понятие регион (region) – совокупность данных, которые являются локальными и независимыми от других регионов. Для каждого региона существует набор задач, которые его используют. Таким образом, система Legion автоматически извлекает параллелизм из программы, обнаруживая независимые между собой задачи, а также находя возможность распараллеливания внутри регионов. Такой подход даёт возможность разделять описание вычислительной части алгоритма от того, каким образом он будет исполняться.

Балансировка нагрузки в Legion происходит с помощью мапперов (mappers) [6]. Для более точного распределения задач между узлами мультикомпьютера мапперы можно переопределять, задавая их поведение от полностью статического распределения до полностью динамического.

Система Legion обладает средствами профилирования на уровне задач [15]. Они позволяют анализировать нагрузку на узлы, а также время исполнения задач на узле и информационные зависимости между задачами. Но в системе нет средств автоматического анализа профиля.

LuNA (Language for Numerical Algorithms) [2] – система фрагментированного программирования, разрабатываемая в ИВМиМГ СО РАН. Язык LuNA создан для описания сложных вычисленных моделей. Программа в LuNA представляется множеством фрагментов вычислений и фрагментов

данных. В LuNA используется похожий подход, что и в системе Legion [6]: описание алгоритма разделено с управлением исполнением программы. Но в LuNA управляющий код генерируется автоматически, а влиять на него на него можно с помощью рекомендаций в коде программы [4].

В LuNA присутствует статический балансировщик нагрузки, который использует рекомендации для распределения фрагментов вычислений по заданным программистом узлам. В предыдущих версиях LuNA были успешные попытки встроить балансировщик на основе профилирования [3]. Алгоритм балансировки определял по профилю простаивающие вычислительные узлы и перераспределял нагрузку на них. Но в предыдущих версиях LuNA язык описания программ был неуниверсальным. Поэтому для современной версии системы алгоритм требует доработки для его адаптации к текущим условиям.

1.2 Итог

Из всех рассмотренных систем LuNA больше всего подходит для реализации в ней алгоритма балансировки нагрузки на основе профилирования, так как программа представляется множеством фрагментов вычислений и фрагментов данных. Фрагментированная структура программы позволяет производить профилирование над отдельными фрагментами вычислений и перемещать их, не меняя логику программы. Кроме того, система LuNA удобна для расширения, так как благодаря её модульной структуре в неё легко встраивать новые системные алгоритмы.

2 Алгоритм автоматической балансировки нагрузки на основе профилирования

2.1 Постановка задачи

Система фрагментированного программирования LuNA – задаче-ориентированная (task-based) система программирования, в которой для распараллеливания применяется крупноблочный (крупнозернистый) параллелизм. Это означает, что распараллеливание происходит путём распределения крупных задач по вычислительным узлам мультикомпьютера.

Задачами в системе LuNA являются фрагменты вычислений. Фрагмент вычислений – независимая единица программы, которая содержит описание входных/выходных переменных и кода фрагмента [2, с. 46]. Фрагменты вычислений принимают на вход и вырабатывают фрагменты данных. Фрагмент данных – агрегат переменных [10, с. 58].

Фрагментированная программа – рекурсивно перечислимое множество фрагментов вычислений и фрагментов данных [2, с. 46].

За исполнение фрагментированных программ в LuNA отвечает RunTime-система. RunTime-система выполняет операции по созданию, перемещению фрагментов вычислений и фрагментов данных.

Исполнение фрагментов вычислений происходит по готовности входных фрагментов данных. После исполнения фрагмента вычислений все его выходные фрагменты данных получают свои значения. Порядок исполнения фрагментов вычислений определён лишь информационными зависимостями между ними: если фрагмент вычислений cf1 потребляет фрагмент данных, который вырабатывается после исполнения фрагмента вычислений cf2, то исполнение cf1 не начнётся, пока не исполнится cf2.

Фрагмент вычислений может мигрировать с узла, на котором был порождён. При этом все входные фрагменты данных должны быть копированы на тот узел, на котором фрагмент вычислений будет исполняться, так как LuNA исполняется на вычислительной системе с распределённой памятью. RunTime-

система организует перемещение необходимых фрагментов данных к фрагменту вычислений. Поэтому при написании фрагментированной программы не нужно заботиться о расположении фрагментов данных и фрагментов вычислений (разве что для увеличения её эффективности, так как пересылки данных замедляют программу).

При исполнении фрагментированной программы RunTime-система может логировать все события, связанные с созданием, перемещением, завершением фрагментов вычислений, а также и события перемещения фрагментов данных.

Таким образом, задача автоматического распределения фрагментов вычислений на основе профилирования рассматривается как считывание логированной RunTime-системой информации (профилированию), получение профиля программы и нахождение корректирующего распределения фрагментов вычислений на вычислительные узлы системы, при котором эффективность программы повышается.

2.2 Описание предлагаемого решения

Основными частями предлагаемого алгоритма являются:

1. Сбор информации о исполнении программы;
2. Квантование всего времени работы программы;
3. Определение квантованной нагрузки на вычислительные узлы системы;
4. Определение дисбаланса нагрузки вычислительных узлов системы;
5. Определение перемещаемых фрагментов вычислений.

2.2.1 Сбор информации о исполнении программы

Для определения нагрузки на узлы необходимо получить информацию о том, в какой момент времени и на каком узле фрагмент вычислений был создан, начал и закончил исполнение, а также информация о перемещениях фрагмента вычислений между узлами. Используя эти данные, можно восстановить ход исполнения каждого отдельного фрагмента вычислений.

2.2.2 Квантование

Из полученной информации – серии событий изменения нагрузки, сложно сделать вывод о том, есть ли дисбаланс и насколько он долог по времени. Поэтому в алгоритме применяется квантование – разбиение времени работы программы на равные между собой промежутки времени. Величина квантования – размер такого промежутка.

После получения информации о работе программы, можно получить время последнего события, связанного с фрагментом вычислений. Тогда, число квантов для заданной величины квантования определяется как:

$$N_q = \left\lceil \frac{t_{last}}{q} \right\rceil, \tag{1}$$

где q – величина квантования, t_{last} – время последнего события, связанного с фрагментом вычислений.

Таким образом всё время исполнения программы представляется как набор квантовых промежутков:

$$x_i = \left[q * i\ ;\ q * (i+1)\right), \text{где } 0 \leq i < N_q \tag{2}$$

Далее весь анализ исполнения программы на предмет дисбаланса нагрузки сводится к определению дисбаланса в каждом из квантов.

Важным является выбор величины квантования. Она должна быть не слишком большой, так как при этом невозможно определить дисбаланс нагрузки точечно. Но и малые размеры величины квантования ведут к резким скачкам нагрузки, связанных с влиянием легковесный фрагментов вычислений, что усложняет анализ.

2.2.3 Определение квантованной нагрузки на вычислительные узлы системы

Весом фрагмента вычислений принимается суммарное время его исполнения. Нагрузка фрагмента вычислений в заданный квант называется квантованным весом фрагмента вычислений и определяется как длина

пересечения квантового промежутка и отрезка исполнения фрагмента вычислений:

$$w_{cf,i} = \|x_i \cap [t_b\ ;\ t_e]\|, \tag{3}$$

где i – номер кванта, а t_b, t_e – время начала и конца исполнения фрагмента вычислений соответственно.

Таким образом, квантованная нагрузка на узел в каждый заданный квант времени определяется как:

$$W_{i,j} = \sum_{cf \in CF_i} w_{cf,j}\ , \tag{4}$$

где CF_i – множество фрагментов вычислений, исполняемых на i узле, $w_{cf,j}$ – квантованный вес фрагмента вычислений на i узел в j квант.

Пример такого преобразования представлен на рисунках 1 – 3. Изначально имеется информация о промежутках времени, в которые каждый фрагмент вычислений исполнялся, на рисунке 1 они представлены в виде отрезков. Далее, происходит подсчёт квантованных весов фрагментов вычислений на узле с величиной квантования 0,5. Пример для cf1 представлен на рисунке 2. После чего происходит подсчёт квантованной нагрузки на узел с величиной квантования 0,5 на рисунке 3.

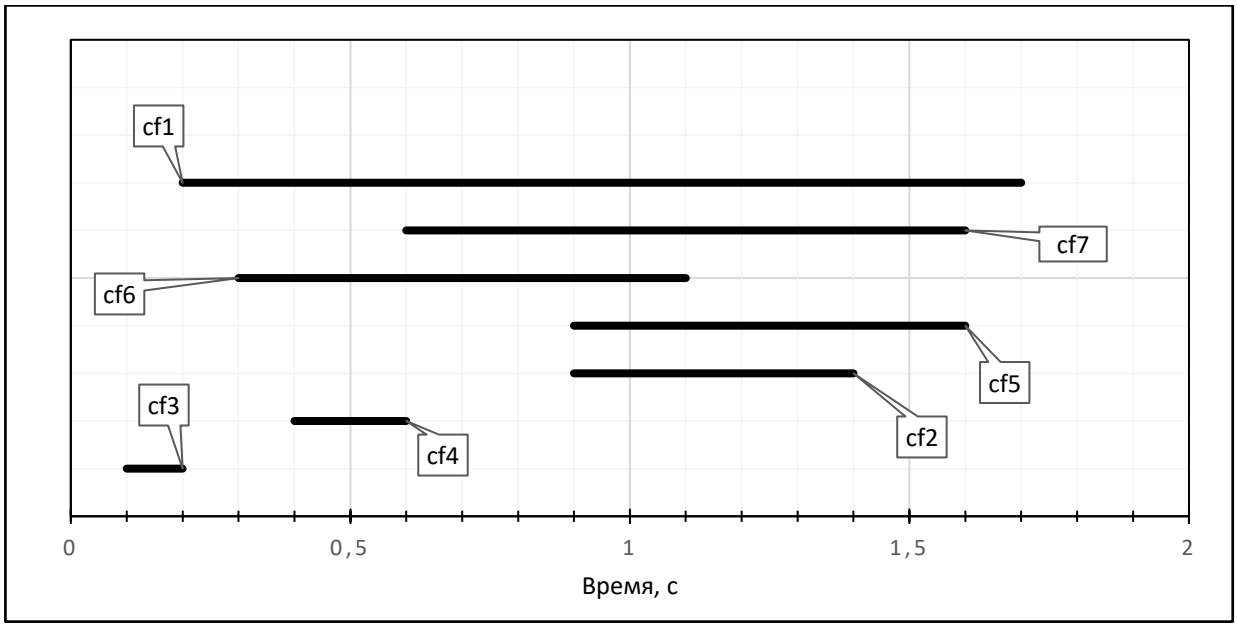

Рисунок 1 – Представление фрагментов вычислений в виде отрезков исполнения

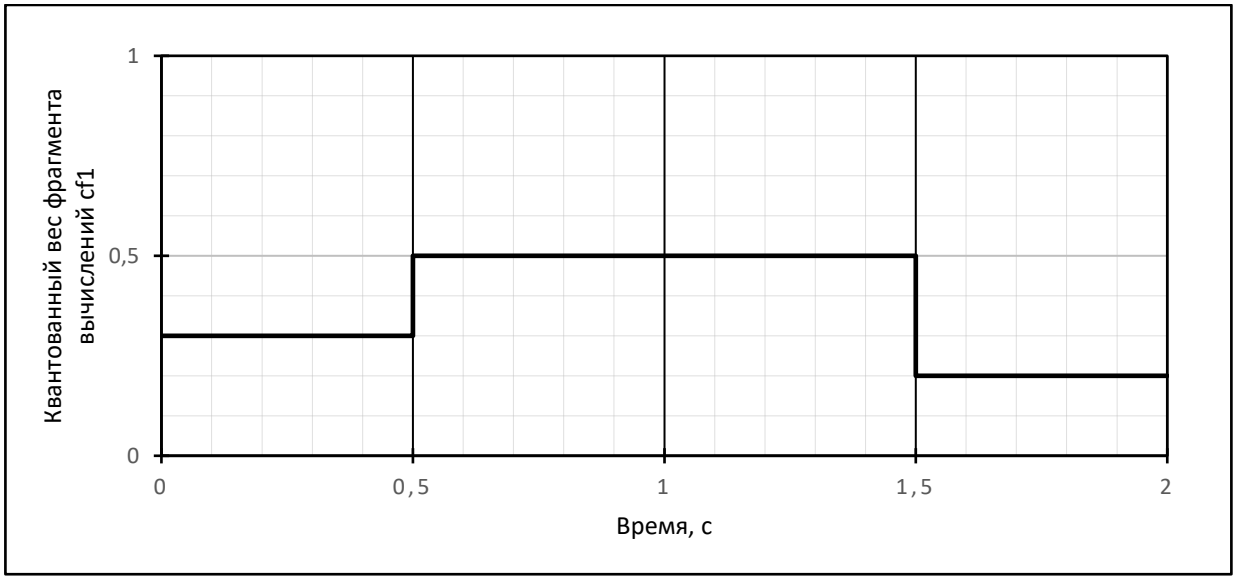

Рисунок 2 – Подсчёт квантованного веса фрагмента вычислений cf1 с величиной квантования 0,5

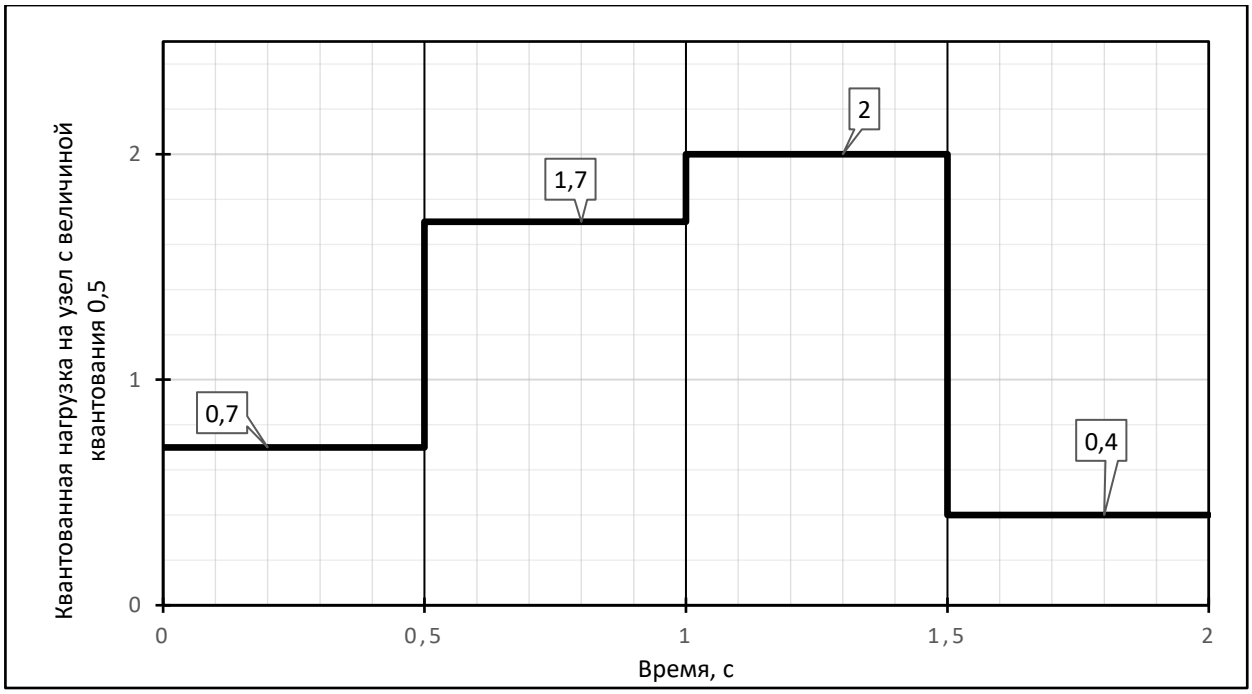

Рисунок 3 – Подсчёт квантованной нагрузки на вычислительный узел с величиной квантования 0,5

2.2.4 Определение дисбаланса нагрузки вычислительных узлов системы

В данной постановке задачи дисбалансом будем называть состояние системы, при котором квантованная нагрузка на одном вычислительном узле больше, чем на других в заданный квант.

Поэтому для определения дисбаланса считается средняя квантованная нагрузка в каждый квант:

$$W_{avg,j} = \frac{\sum_{i \epsilon N} W_{i,j}}{\|N\|}, \tag{5}$$

где N – множество вычислительной узлов системы, $\|N\|$ – мощность множества вычислительных узлов системы.

В зависимости от того, больше или меньше нагрузка, чем средняя, можно говорить о существовании дисбаланса и более или менее нагруженных узлов соответственно. Пример приведён на рисунке 4.

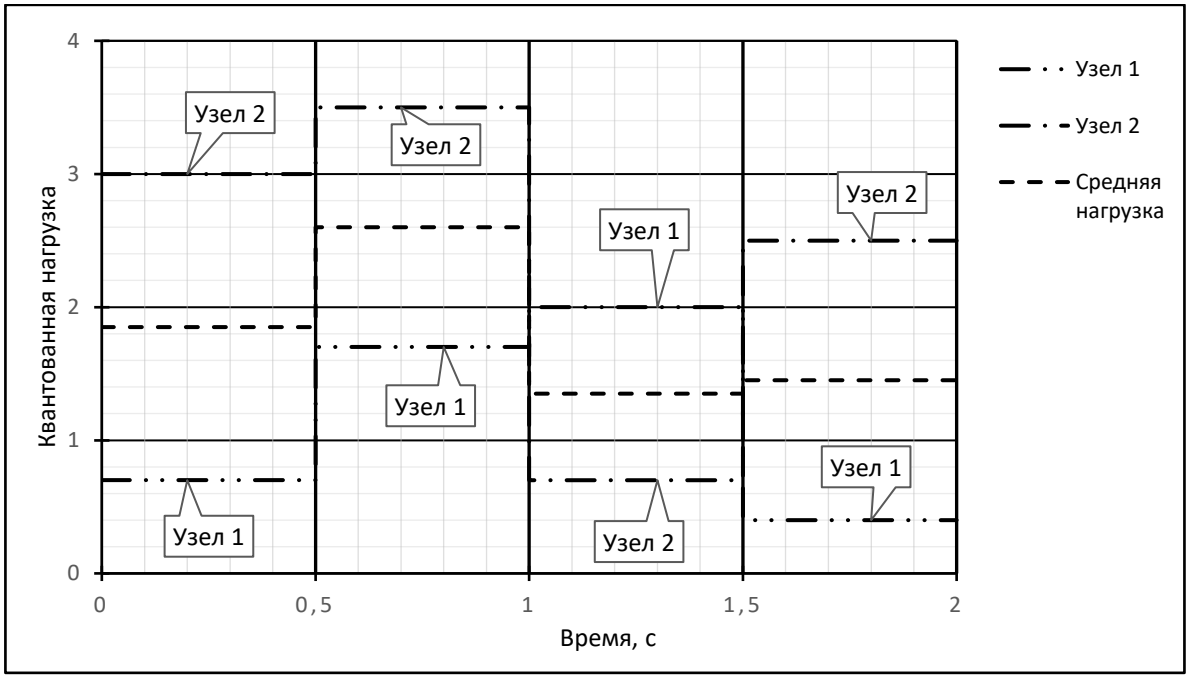

Рисунок 4 – Определение дисбаланса для величины квантования 0,5

2.2.5 Определение перемещаемых фрагментов вычислений

Получив информацию о дисбалансе в каждый квант времени, необходимо выбрать, распределение каких фрагментов вычислений нужно изменить. Чтобы уменьшить число перемещений, сначала корректируется распределение фрагментов вычислений с большим весом, если это возможно. Шаги работы алгоритма балансировки:

1. Кванты сортируются по убыванию суммарной квантованной нагрузки;

2. В каждом кванте определяется самый нагруженный узел и самый мало загруженный узел;

3. Для самого нагруженного узла сортируются фрагменты вычислений по убыванию нагрузки;

4. Для перемещения фрагмента вычислений с самого нагруженного узла на самый мало загруженный необходимо выполнение условия:

$$W_{max,j} - w_{cf,j} > W_{min,j} + w_{cf,j}\,, \qquad (6)$$

где j – заданный квант, max и min – узел с наибольшей и наименьшей квантованной нагрузкой в j квант соответственно, $w_{cf,j}$ – квантованный вес фрагмента вычислений в j квант.

То есть, фрагмент вычислений переносится, если после перемещения квантованная нагрузка на узел max всё ещё больше, чем на узел min;

5. При выполнении условия фрагмент перемещается с max узла на min, после чего квантованная нагрузка пересчитывается;

6. Далее повторяются шаги 2 – 5, пока условие шага 4 выполняется;

7. Процесс повторяется для следующего кванта.

2.3 Характеристика предложенного алгоритма

Разработанный алгоритм обнаруживает дисбаланс в каждый квант времени и производит перемещение фрагментов вычислений до тех пор, пока выполняется условие шага 4.

Алгоритм не гарантирует, что производительность увеличится или уменьшится. Во многих случаях алгоритм будет устранять крупные дисбалансы нагрузки вычислительных узлов, но в меньших масштабах может давать ухудшения. При повторных запусках, алгоритм будет также обнаруживать дисбалансы нагрузки и корректировать созданное на предыдущем шаге распределение, постепенно приводя распределение к некоторой окрестности локального оптимума.

Также, возможно использование этого алгоритма совместно с другими алгоритмами статической балансировки нагрузки, которые можно будет корректировать посредством балансировки на основе профиля программы.

3 Программная реализация

Алгоритм был реализован на языке Python. Реализация проводилась на базе системы фрагментированного программирования LuNA. Для реализованной программы в приложении А приведено руководство программиста, а в приложении Б – описание программы.

3.1 Особенности порождения фрагментов вычислений в LuNA

Фрагменты вычислений можно условно разделить на атомарные и порождающие. Фрагмент вычислений, порождённый другим фрагментом вычислений, будет инициализирован на том же узле, что его родитель.

При запуске программы создаётся один фрагмент вычисления, соответствующий подпрограмме main LuNA программы. Он является порождающим фрагментом вычисления и не имеет родителя.

3.2 Средства профилирования LuNA

Основным средством профилирования в LuNA являются лог-файлы. Лог-файлы появляются во время исполнения программы. Они хранят информацию о фрагментах вычислений и фрагментах данных. Важной для задачи была информация о фрагментах данных. Пример приведён на рисунке 5.

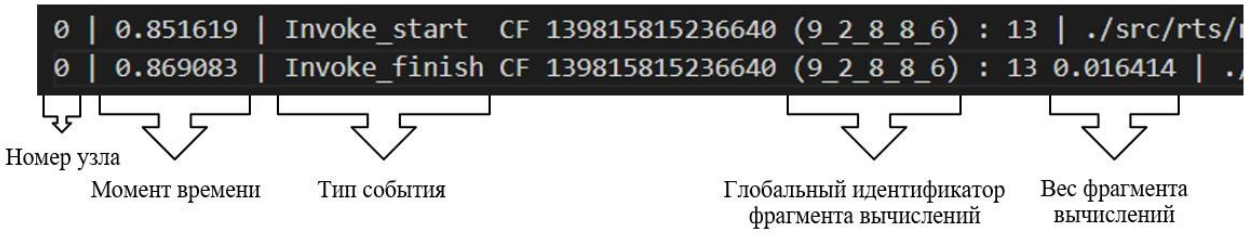

Рисунок 5 – Пример событий Invoke_start и Invoke_finish для фрагмента вычислений

Лог-файлы предоставляют информацию о:

- Номере узла, на котором произошло событие;
- Момент времени от начала исполнения программы;
- Тип события:
  - Create – создание фрагмента вычислений;
  - Invoke_start – начало исполнения фрагмента вычислений;
  - Invoke_finish – конец исполнения фрагмента вычислений;

- Finish – завершения исполнения фрагмента вычислений;
- Migrate – миграция фрагмента вычислений на другой узел;
- Receive – приём фрагмента вычислений с другого узла.

- Локальный идентификатор фрагмента вычислений, данный ему при инициализации во время создания или получения от другого узла;
- Глобальный идентификатор – уникальный идентификатор фрагмента вычислений;
- Номер блока, в котором произошло событие.

3.2.1 Добавление глобального идентификатора

Ранее в LuNA единственным идентификатором фрагмента вычислений был его локальный идентификатор, который представлял из себя адрес ячейки памяти, в которой он был инициализирован.

Проблема локального идентификатора состоит в том, что он не является идемпотентным – при повторном запуске программы не гарантируется, что локальный идентификатор будет таким же, как в предыдущий запуск, для заданного фрагмента вычислений. Также при перемещении фрагмента вычислений на другой узел он получает новый идентификатор. Соответственно, невозможно отследить всю историю исполнений конкретного фрагмента вычислений.

Поэтому был разработан и реализован способ идентификации фрагментов вычислений через глобальный идентификатор. Из себя он представляет вектор целых чисел. Он учитывает особенности порождения фрагментов вычислений и соответствует стеку вызовов (порождений).

Все фрагменты вычислений, кроме головного, порождаются другими фрагментами вычислений. Поэтому, во всех местах фрагментированной программы, где происходит порождение нового фрагмента вычислений, было решено присваивать уникальный номер порождения на этапе компиляции. Тогда при порождении фрагмента вычислений его глобальным идентификатором назначается последовательность номеров, которая складывается из глобального

идентификатора родительского фрагмента вычислений и уникального номера порождения.

Процесс отличается для таких операторов как for и while. В LuNA они являются порождающими фрагментами вычислений, повторяющими своё тело заданное число раз. Поэтому недостаточно добавить только уникальный номер порождения. Для них в идентификатор порождённого фрагмента вычислений добавляется сначала номер итерации, а после него – уникальный номер порождения. Пример присваивания глобальных идентификаторов для программы из рисунка 6 представлен в таблице 1.

```
1:      #!/usr/bin/luna
2:
3:      import c_init(int, name) as init;
4:
5:      sub main()
6:      {
7:         df x;
8:
9:         cf init0: init(8, x[0]);  // сгенерированный номер: 1
10:
11:        for i=1..2  //  сгенерированный номер: 3
12:        {
13:           cf init[i]: init(i, x[i]);  // сгенерированный номер: 2
14:        }
15:     }
```

Рисунок 6 – Пример присвоения глобальных идентификаторов

Таблица 1 – Глобальные идентификаторы фрагментов вычислений

Название фрагмента вычислений в коде программы	Глобальный идентификатор фрагмента вычислений
main	пустой
init0	1
for i=1..2	3
init[1]	3_1_2
init[2]	3_2_2

Так как задание уникальных номеров для порождений фрагментов вычислений происходит на этапе компиляции, то от запуска к запуску фрагменты вычислений будут иметь одинаковые глобальные идентификаторы.

К тому же, глобальный идентификатор сохраняется при миграции фрагмента вычислений на другой узел, что позволяет отслеживать весь ход исполнения конкретного фрагмента вычислений, а также создавать указания по его новому распределению при последующих запусках.

На рисунке 7 представлен пример дерева порождений, включающий глобальный идентификатор в круглых скобках. Те фрагменты вычислений, чьи индексы заключены в штрихованный прямоугольник, на самом деле не существуют. Они нужны, чтобы показать, как операторы for и while добавляют номер итерации к глобальному идентификатору.

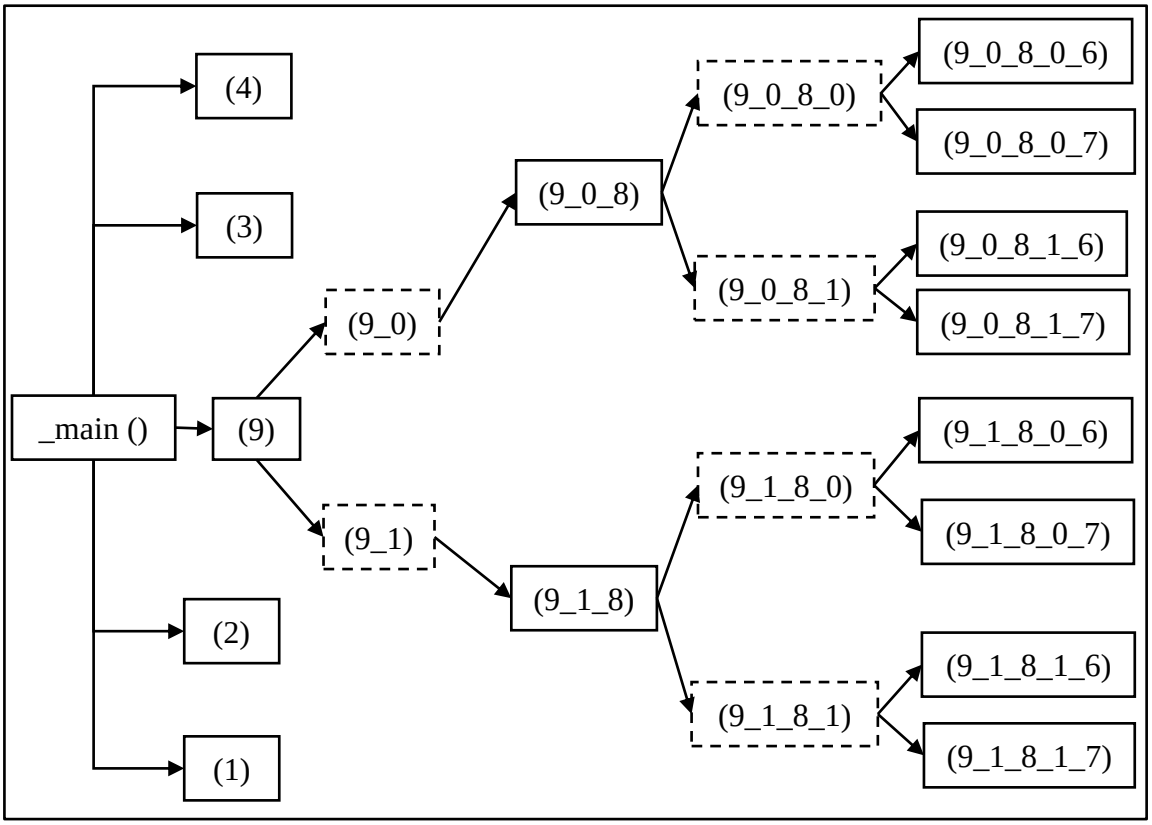

Рисунок 7 – Пример дерева порождений фрагментов вычислений

3.2.2 Изменение в системе логирования

Вес фрагмента вычислений определялся как разность между временными метками для событий Invoke_start и Invoke_finish фрагмента вычислений. Так как вывод в лог-файл может происходить с задержкой, есть вероятность неточно определить вес фрагмента. Поэтому во время исполнения фрагмента вычислений происходит замер времени исполнения, которое выводится при событии

Invoke_finish в лог-файл. Это позволяет более точно определить вес фрагмента вычислений.

3.3 Парсер лог-файлов

Парсер был написан на языке Python. Алгоритм считывает из лог-файлов информацию, связанную с фрагментами вычислений. В результате работы парсера для каждого фрагмента вычислений собираются события с ним связанные. Далее эта информация используется для определения квантованного веса фрагментов вычислений.

3.4 Определение нагрузки на вычислительные узлы системы

Фрагмент вычислений нагружает узел между событиями Invoke_start и Invoke_finish. Так как событие Invoke_Finish содержит вес фрагмента вычислений, то есть время его исполнения, то отрезок времени, в который фрагмент вычислений исполнялся, можно определить как:

$$[IF.time - IF.weight; IF.time], \quad (7)$$

где IF – событие Invoke_finish фрагмента вычислений, $IF.time$ – время события Invoke_finish, $IF.weight$ – вес промежутка исполнения фрагмента вычислений.

Получив для каждого фрагмента вычислений его промежутки исполнения, можно приступить к вычислению квантованной нагрузки на узлы.

3.5 Создание нового распределения

Алгоритм балансировки порождает новое распределение в виде карты ключ – значение, где ключом является глобальный идентификатор фрагмента вычислений, а значение – номер узла, на который он будет перемещён после создания.

3.6 Поддержка созданного распределения в RunTime-системе

Данные о новом распределении собираются в файл libucodes.so.profile, располагающийся в папке с файлом libucodes.so. libucodes.so – это файл, в который компилируется программа, написанная на языке LuNA.

Код RunTime-системы был доработан таким образом, что при чтении файла скомпилированной программы, считывается файл libucodes.so.profile, если он есть. RunTime-система парсит файл с новым распределением, получая при этом указанию по размещению конкретных фрагментов вычислений.

В RunTime-системе есть код, который отвечает за миграцию фрагментов вычислений. Этот код был модифицирован так: если для фрагмента вычислений есть коррекция распределения, созданная алгоритмом балансировки на основе профилирования, то ей отдаётся приоритет перед основным механизмом миграции.

3.7 Дополнительный функционал

Собранная информация позволяет визуализировать квантованную нагрузку вычислительных узлов, что позволяет пользователю увидеть дисбаланс нагрузки вычислительных узлов.

Был написан скрипт для утилиты gnuplot [16], который позволяет на основе файла с данными о квантованной нагрузке на вычислительные узлы строить график квантованной нагрузки. Для генерации файла с данными о квантованной нагрузке был написан python скрипт, который использует собранную алгоритмом информацию. Пример представлен на рисунке 8.

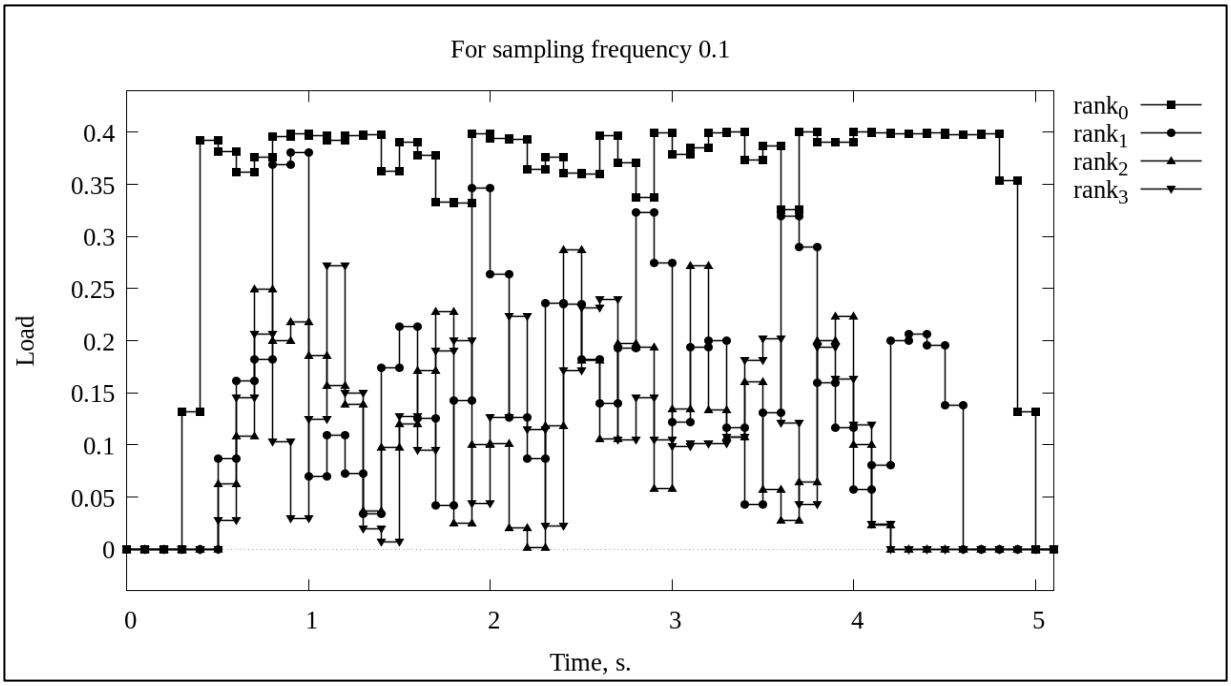

Рисунок 8 – График квантованной нагрузки

4 Тестирование

Для проверки эффективности работы алгоритма балансировки были проведены тесты на программах на языке LuNA.

Тестирование проводилось на кластере MVS-10p МСЦ РАН [12].

4.1 Тестирование на программе без статической балансировки фрагментов вычислений

Идея теста состоит в том, чтобы применить алгоритм балансировки к программе с заведомым дисбалансом и посмотреть, какой результат получится.

Тестовая программа (test_1) декомпозирует матрицу по двум измерениям и производит прибавление единицы к каждому элементу. Листинг программы приведён на рисунке 9. Изначально, все фрагменты вычислений распределены на один вычислительный узел. Далее проводится запуск с распределением, созданным алгоритмом балансировки. Программа запускалась на 4 узлах. Параметры запуска программы: размерность матрицы фрагментов данных 48x48, каждый фрагмент данных является подматрицей размера 48x48. Число повторных прибавлений 1000000. Добавление этого параметра связано с увеличением веса фрагмента вычислений с помощью увеличения количества операции прибавления единицы. Величина квантования 0,1. Результаты приведены в таблице 2.

Таблица 2 – Результаты тестирования на программе без статической балансировки

Имя программы	До использования балансировки, с.	Первое применение балансировки, с.	Второе применение балансировки, с.
test_1	73,401	25,941	26,521

Из результатов теста можно сделать вывод, что алгоритм работает, а также даёт значительное ускорение. Второе применение балансировки привело к незначительному увеличению времени работы программы. То, как изменялась квантованная нагрузка на узлы, можно увидеть на рисунках 10, 11 и 12. Из графиков видно, что квантованная нагрузка не является одинаковой на всех узлах в каждый квант времени. Это связано с тем, что алгоритм не может сделать

нагрузку одинаковой, так как распределяет нагрузку на вычислительные узлы с точностью до фрагмента вычислений.

```
1:    #!/usr/bin/luna
2:
3:    import c_init(int, name) as init;
4:    import c_init_submatrix(int `height, int `width, name `dest) as init_mat;
5:    import c_print_mat(value `a) as print_mat;
6:    import c_add_one(value `a, int `repeat, name `b) as add_one;
7:
8:    // N – размер фрагмента матрицы, FgSize – число фрагментов матрицы,
9:    // Repeat – число повторных прибавлений единицы к фрагменту матрицы
10:   sub main(int N, int FgSize, int Repeat)
11:   {
12:      df A, B, FgMxWidth, FgMxHeight, MxHeight, MxWidth, NumOfRepeats;
13:
14:      // Инициализация размеров матрицы
15:      init(N, MxWidth);
16:      init(N, MxHeight);
17:
18:      // Инициализация размеров фрагмента матрицы
19:      init(FgSize, FgMxWidth);
20:      init(FgSize, FgMxHeight);
21:
22:      // Инициализация числа повторных прибавлений единицы
23:      init(Repeat, NumOfRepeats);
24:
25:      for i=0..FgMxHeight-1
26:      {
27:         for j=0..FgMxWidth-1
28:         {
29:            // Инициализация фрагмента матрицы начальными значениями
30:            cf initA[i][j]: init_mat(MxWidth, MxHeight, A[i][j]);
31:
32:            // Добавление к каждому элементу фрагмента матрицы единицы
33:            // NumOfRepeats раз
34:            cf addOne[i][j]: add_one(A[i][j], NumOfRepeats, B[i][j]);
35:         }
36:      }
37:   }
```

Рисунок 9 – Листинг программы test_1

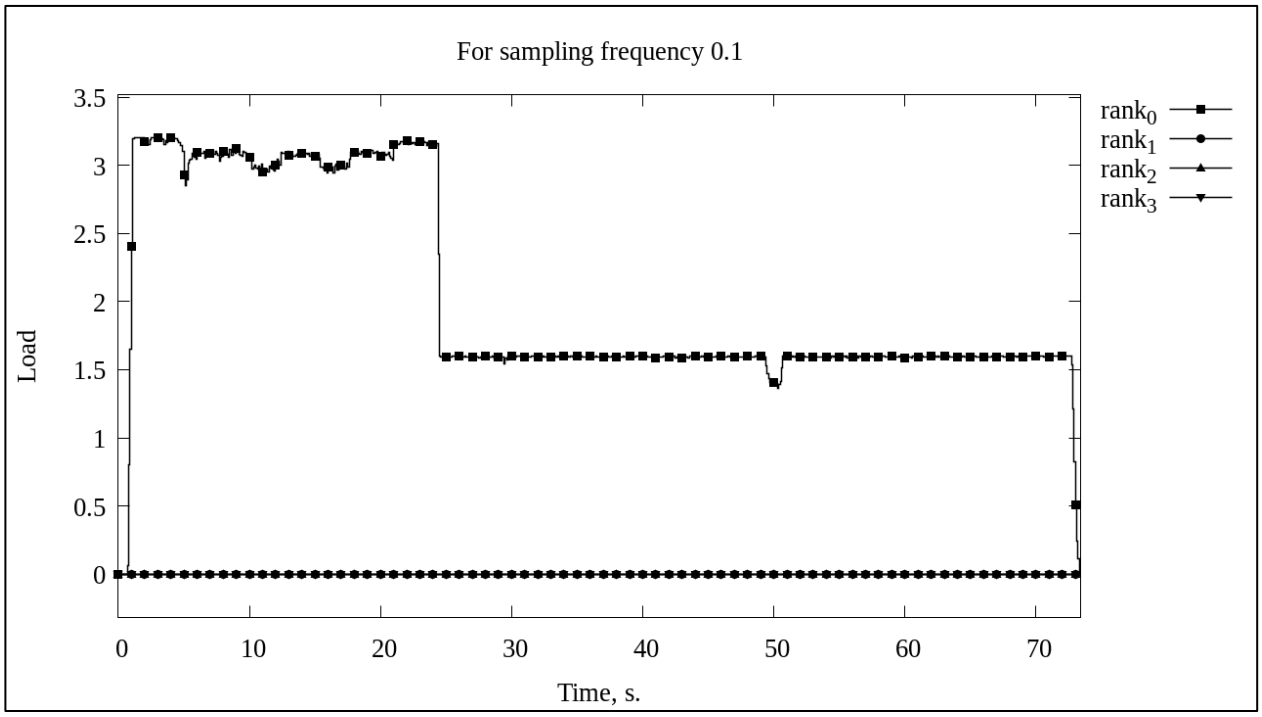

Рисунок 10 – График квантованной нагрузки программы (test_1) с величиной квантования 0,1 до применения алгоритма балансировки

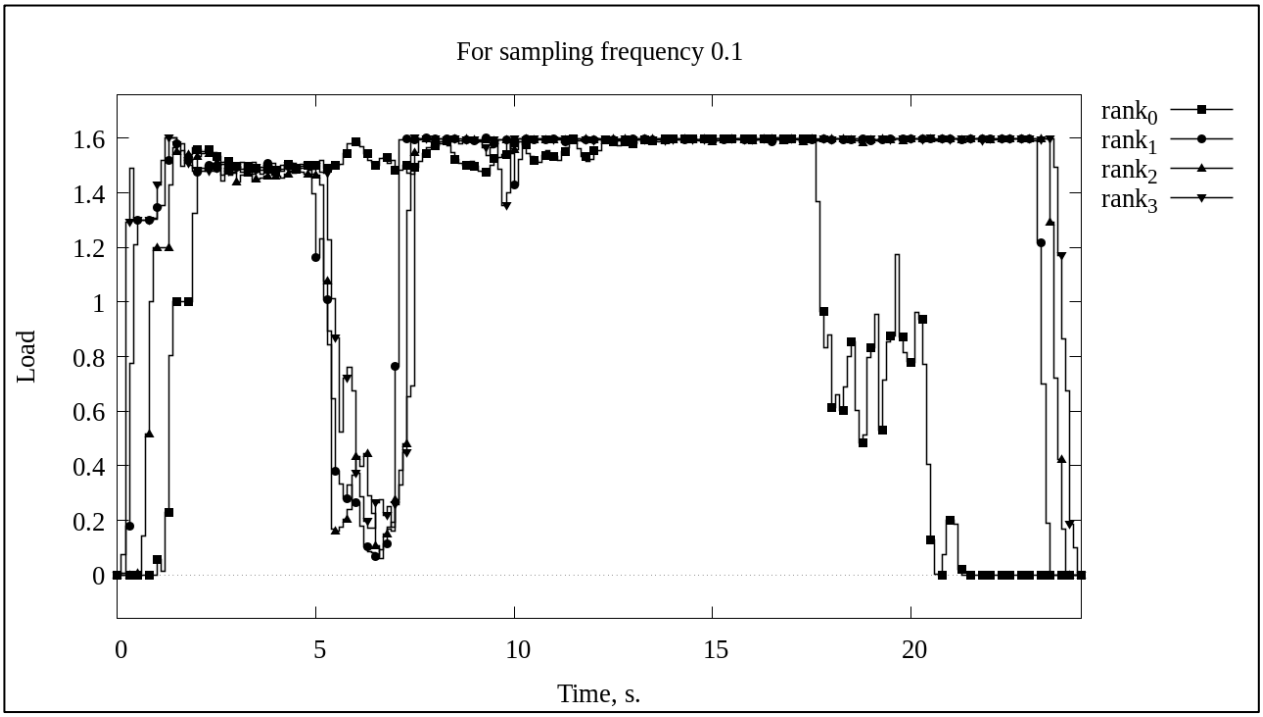

Рисунок 11 – График квантованной нагрузки программы (test_1) с величиной квантования 0,1 после первого применения алгоритма балансировки

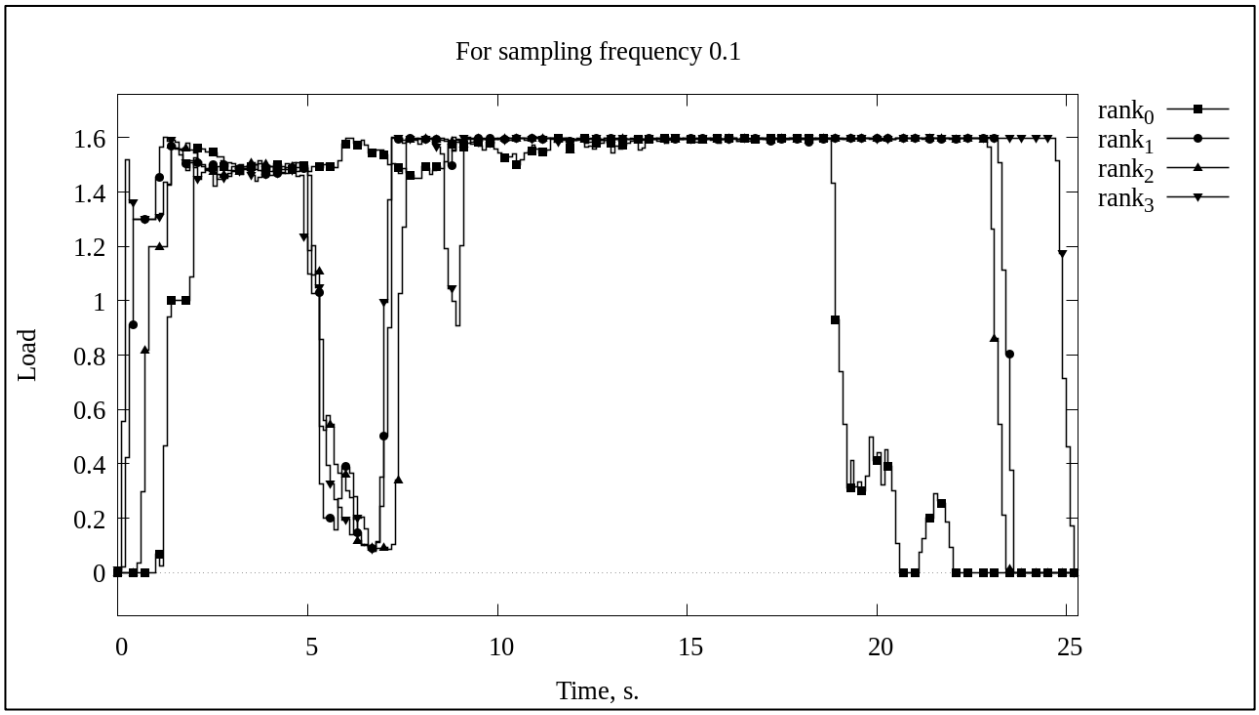

Рисунок 12 – График квантованной нагрузки на узлы программы (test_1) с величиной квантования 0,1 после второго применения алгоритма балансировки

Таким образом, тест показал, что алгоритм успешно работает, а количественные показатели вписываются в ожидание – время работы программы уменьшилось из-за перераспределения нагрузки с одного вычислительного узла на все.

4.2 Тестирование на программе со статической балансировкой фрагментов вычислений на все узлы

Идея теста – применить алгоритм балансировки к программе test_2 со статической балансировкой фрагментов вычислений на все узлы и определить, как алгоритм сработает в данной ситуации.

Для этого теста программа test_1 была изменена. В неё добавлены указания по равномерному распределению по вычислительным узлам фрагментов вычислений, которые инициализируют фрагмент матрицы и добавляют к каждому элементу матрицы единицу. Листинг программы приведён на рисунке 13. В строках 30 и 37 задано статическое распределение фрагментов вычислений по вычислительным узлам. Параметры запуска программы: размерность матрицы фрагментов данных 48x48, каждый фрагмент данных является

подматрицей размера 48x48. Число повторных прибавлений 1000000. Величина квантования 0,1. Результаты тестирования приведены в таблице 3.

Таблица 3 – Результаты тестирования алгоритма балансировки с величиной квантования 0,1 на программе со статической балансировкой на все узлы

Имя программы	До использования балансировки, с.	Первое применение балансировки, с.	Второе применение балансировки, с.
test_2	25,381	34,617	27,710

Таким образом, алгоритм показал свою применимость к программам, уже имеющим некоторое статическое распределение фрагментов вычислений. По результатам теста видно, что производительность программы ухудшается при первом применении алгоритма. Это ожидаемое поведение, так как алгоритм применялся к программе с заведомо равномерным распределением. Поэтому предложенная коррекция оказалась хуже. Но при втором применении алгоритм обнаружил и устранил дисбаланс.

По сути, программа test_2 является той же программой, что и в test_1, но с равномерным распределением фрагментов вычислений. При этом, время работы программы test_1 с применением алгоритма балансировки близко к времени test_2 без применения алгоритма балансировки. Это означает, что алгоритм балансировки изменил распределение фрагментов вычислений в программе test_1 так, что оно стало близким к оптимальному распределению.

Таким образом, алгоритм, как и ожидалось, обнаруживает и устраняет дисбаланс нагрузки вычислительных узлов.

4.3 Тестирование алгоритма балансировки с различной величиной квантования

Идея теста заключается в проверке, как влияет величина квантования на эффективность получаемого распределения.

В этом тесте используется программа test_1 из пункта 4.1. Были проведены тесты для числа узлов 2 и 4, а также для различных величин квантования. Параметры запуска программы: размерность матрицы фрагментов данных

48x48, каждый фрагмент данных является подматрицей размера 48x48. Число повторных прибавлений 1000000. Результаты представлены в таблице 4.

Таблица 4 – Результаты тестирования программы test_1 с различными значениями величины квантования

Число вычислительных узлов	До применения алгоритма балансировки	Величина квантования 0,01	Величина квантования 0,1	Величина квантования 1	Величина квантования 10
2	72,08	41,046	41,058	43,857	44,217
4	73,401	24,256	25,9410	25,291	24,189

Из результатов видно, что для данной программы величина квантования играет небольшую роль – при её изменении время исполнения изменяется незначительно. Также, при использовании алгоритма балансировки, время работы программы на 2-х узлах уменьшилось примерно в 1,7 раза, а на 4-х узлах – примерно в 3 раза. Это хороший показатель для алгоритма автоматической балансировки.

```
1:    #!/usr/bin/luna
2:    import c_init(int, name) as init;
3:    import c_init_submatrix(int `height, int `width, name `dest) as init_mat;
4:    import c_print_mat(value `a) as print_mat;
5:    import c_add_one(value `a, int `repeat, name `b) as add_one;
6:
7:    // N – размер фрагмента матрицы, FgSize – число фрагментов матрицы,
8:    // Repeat – число повторных прибавлений единицы к фрагменту матрицы
9:    sub main(int N, int FgSize, int Repeat)
10:   {
11:      df A, B, FgMxWidth, FgMxHeight, MxHeight, MxWidth, NumOfRepeats;
12:      // Инициализация размеров матрицы
13:      init(N, MxWidth);
14:      init(N, MxHeight);
15:      // Инициализация размеров фрагмента матрицы
16:      init(FgSize, FgMxWidth);
17:      init(FgSize, FgMxHeight);
18:
19:      // Инициализация числа повторных прибавлений единицы
20:      init(Repeat, NumOfRepeats);
21:
22:      for i=0..FgMxHeight-1
23:      {
24:         for j=0..FgMxWidth-1
25:         {
26:            // Инициализация фрагмента матрицы начальными значениями
27:            cf initA[i][j]: init_mat(MxWidth, MxHeight, A[i][j]) @ {
28:               // Рекомендация по распределению фрагмента вычислений initA[i][j]
29:               // в соответствии с i индексом
30:               locator_cyclic: i;
31:            };
32:            // Добавление к каждому элементу фрагмента матрицы единицы
33:            // NumOfRepeats раз
34:            cf addOne[i][j]: add_one(A[i][j], NumOfRepeats, B[i][j]) @ {
35:               // Рекомендация по распределению фрагмента вычислений add_one[i][j]
36:               // в соответствии с i индексом
37:               locator_cyclic: i;
38:            };
39:         }
40:      }
41:   }
```

Рисунок 13 – Листинг программы test_2

4.4 Тестирование на реальной задаче вычислительной математики

Идея теста – применить алгоритм балансировки к реальной задаче вычислительной математики. В данном примере тестовая программа (test_3)

решает трёхмерное уравнение теплопроводности методом конвейерной прогонки [5].

Программа test_3 содержит в статическую балансировку, заданную программистом из соображений соседства фрагментов вычислений. Были применены несколько величин квантования. Параметры запуска: размер сетки 15x15x15, число потоков 3x3x3, шаг по времени 0,001, максимальная отметка времени 0,01. Запуск осуществлялся на двух узлах. Результаты приведены в таблице 5.

Таблица 5 – Результаты тестирования на программе test_3 с величинами квантования 1 и 10

	До применения алгоритма балансировки, с.	Величина квантования 1, с.	Величина квантования 10, с.
test_3	226,856	218,182	177,653

Из результатов видно влияние величины квантования: для величины квантования 1 результат немногим лучше, чем до применения алгоритма. Но для величины квантования 10 время исполнения программы уменьшилось примерно в 1,3 раза. Это связано с тем, что величина квантования 1 слишком мала для данной задачи.

Таким образом, алгоритм показал ускорение на реальной задаче, в которой программист вручную распределил ресурсы. Это означает, что практическая ценность алгоритма подтверждается тестированием.

4.5 Итоги

Таким образом, реализованное решение показало свою работоспособность. Так для случая с программой без статической балансировки (test_1) алгоритм скорректировал распределение, которое значительно уменьшило время исполнения программы. Повторное применение алгоритма балансировки не дало ускорения. В случаях, когда статическое распределение задано, алгоритм балансировки показывал как ухудшение (test_2), так и улучшение эффективности программы (test_3). Ухудшение можно объяснить тем, что алгоритм эвристический: он устраняет крупные дисбалансы, но в

меньших масштабах может давать ухудшение, что подтверждается результатами тестов.

Также важную роль играет величина квантования и от её выбора зависит эффективность корректирующего распределения, что видно в test_3.

Кроме того, практическая ценность алгоритма подтверждена результатами теста на реальной задаче, в которой было вручную задано распределение (test_3).

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы был разработан и реализован алгоритм автоматического распределения ресурсов фрагментированных программ на основе профилирования и встроен в систему фрагментированного программирования LuNA.

Данная работы была опубликована на 59-й Международной научно-студенческой конференции, г. Новосибирск, 2021 г. и была награждена дипломом второй степени [1].

Защищаемые положения:

1. Разработан и реализован алгоритм автоматической балансировки нагрузки на вычислительные узлы системы на основе профилирования в системе LuNA;
2. Проведено экспериментальное исследование работоспособности алгоритма балансировки нагрузки на основе профилирования в системе LuNA.

Разработка и реализация алгоритма автоматического распределения ресурсов на основе профилирования в системе LuNA улучшило производительность программ. Кроме того, практическая ценность работы подтверждается результатами тестов.

В дальнейшем планируется развивать алгоритмы автоматической балансировки нагрузки на основе профилирования. Возможные направления развития:

1. Разработка новых алгоритмов устранения дисбаланса;
2. Разработка рекомендаций по автоматической балансировке на основе профилирования с возможностью указывать величину квантования;
3. Разработка эвристик на основе профиля для оптимального выбора величины квантования.

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».

Лямин Артём Сергеевич _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Лямин, А. С. Разработка и реализация алгоритма распределения ресурсов фрагментированных программ на основе профилирования / А. С. Лямин // Информационные технологии : Материалы 59-й Междунар. науч. студ. конф. 12–23 апреля 2021 г. / Новосиб. гос. ун-т. — Новосибирск : ИПЦ НГУ, 2021. — С. 114.
2. Малышкин, В. Э. Технология фрагментированного программирования / В. Э. Малышкин // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. – 2012. – N 46(305). – С. 45 – 55.
3. Перепёлкин, В. А. Оптимизация исполнения фрагментированных программ на основе профилирования / В. А. Перепёлкин // Шестая Сибирская конференция по параллельным и высокопроизводительным вычислениям: Программа и тезисы докладов. Томск: Изд-во Том. ун-та., 2011. – С. 117 – 122.
4. Перепелкин, В. А. Проблема распределения ресурсов мультикомпьютера в технологии фрагментированного программирования // Научный сервис в сети Интернет: поиск новых решений: Труды Международной суперкомпьютерной конференции (17-22 сентября 2012 г., г. Новороссийск). – М.: Изд-во МГУ, 2012. – С. 398 – 401.
5. Akhmed-Zaki, D., Lebedev, D., Perepelkin, V. Implementation of a 3D model heat equation using fragmented programming technology // The Journal of Supercomputing. – 2019. – Vol. 75, Issue 12. – P. 7827 – 7832.
6. Bauer, M. Legion: Expressing Locality and Independence with Logical Regions / M. Bauer, S. Treichler, E. Slaughter, A. Aiken // SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis – 2012. – Article N 66 – P. 1 – 11.
7. Danalis, A. PaRSEC in Practice: Optimizing a legacy Chemistry application through distributed task-based execution / A. Danalis, H. Jagode, G. Bosilca, J.

Dongarra // 2015 IEEE International Conference on Cluster Computing. – 2015. – P. 304 – 313.

8. Kale, L. V. Parallel Science and Engineering Applications: The Charm++ Approach / L. V. Kale , A. Bhatele. – CRC Press, 2013. – 314 P.
9. Kale, L. V., Krishnan, S. Charm++: a portable concurrent object oriented system based on c++ // Proccedings of OOPSLA'93. – 1993. – Vol. 28. – P. 91 – 108.
10. Malyshkin, V. E. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem / V. E. Malyshkin, V. A. Perepelkin // Proceedings of the 11th Conference on Parallel Computing Technologis. – 2011. – Vol. 6873. – P. 53–61.
11. Wu, W. Hierarchical DAG Scheduling for Hybrid Distributed Systems / W. Wu, A. Bouteiller, G. Bosilca, M. Faverge [et al] // 2015 IEEE International Parallel and Distributed Processing Symposium. – 2015. – P. 156 – 165.
12. Межведомственный Суперкомпьютерный Центр Российской Академии Наук [Электронный ресурс]. – Режим доступа: http://www.jscc.ru/ (дата обращения 10.05.2021).
13. The Charm++ Parallel Programming System. Официальная документация Charm++ [Электронный ресурс]. – Режим доступа: https://charm.readthedocs.io/en/latest/charm++/manual.html (дата обращения 09.05.2021).
14. PaRSEC. Официальный репозиторий bitbucket [Электронный ресурс]. – Режим доступа: https://bitbucket.org/icldistcomp/parsec/wiki/Profiling%20PaRSEC-based%20applications (дата обращения 09.05.2021)
15. Performance Profiling and Tuning. Официальный сайт Legion [Электронный ресурс]. – Режим доступа: https://legion.stanford.edu/profiling/ (дата обращения 09.05.2021).
16. Official gnuplot documentation. Официальная документация . Официальная документация gnuplot [Электронный ресурс]. – Режим доступа: http://www.gnuplot.info/documentation.html (дата обращения 12.05.2021)

ПРИЛОЖЕНИЕ А

БАЛАНСИРОВЩИК НАГРУЗКИ НА ОСНОВЕ ПРОФИЛИРОВАНИЯ ДЛЯ СИСТЕМЫ «LuNA»

РУКОВОДСТВО ПРОГРАММИСТА

Листов 9

Новосибирск 2021

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данном документе приведено руководство программиста для балансировщика нагрузки на основе профилирования в системе LuNA. Исходным языком программы является Python. Средство разработки – редактор исходного кода PyCharm от компании JetBrains.

Основной функцией программы является построение корректирующего распределение фрагментов вычислений на вычислительные узлы мультикомпьютера.

Оформление программного документа «Руководство оператора» 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 программы

1.1 Назначение программы

Балансировщик нагрузки на основе профилирования предназначен для работы с программами, которые запускаются на более чем одном вычислительном узле.

1.2 Функции, выполняемые программой

Программа строит корректирующее распределение для LuNA-программы, используя лог-файлы её исполнения.

Также программа позволяет строить графики квантованной нагрузки на вычислительные узлы.

1.3 Условия, необходимые для выполнения программы

Чтобы программа могла исполняться, необходимо наличие у программиста установленной утилиты gnuplot, интерпретатора Python версии 3.8 и выше, а также пакеты typing, subprocess, glob.

2 Характеристика программы

2.1 Описание основных характеристик программы

Программа имеет несколько режимов работы. Первый – построение графика квантованной нагрузки на вычислительные узлы для исполненной программы. Второй – построение корректирующего распределения фрагментов вычислений на вычислительные узлы для исполненной программы.

2.2 Описание основных особенностей программы

Программа сохраняет корректирующие распределение фрагментов вычислений на вычислительные узлы в виде файла.

3 Обращение к программе

3.1 Описание процедур вызова программы

Для обработки информации лог-файлов исполненной LuNA-программы используется класс Parser из parser.py. Конструктор данного класса принимает число узлов, на которых исполнялась LuNA-программа.

Для указания лог-файлов исполненной LuNA-программы используется метод parse_file класса Parser. Его входные параметры – путь до директории с лог-файлами, а также маска файла, например «log.*».

Далее собранную парсером информацию необходимо преобразовать в класс LoadData из load_data.py. Для этого у класса Parser вызывается метод fill_load_data, который принимает на вход величину квантования.

Получив данные о загрузке (LoadData), можно построить график квантованной нагрузки посредством вызова метода build_plot, аргументами которого является директория, в которой файл графика будет создан, а также задаваемый префикс файла.

Для получения корректирующего распределение, необходимо вызвать метод print_cf_move_to_file у класса LoadData. Метод принимает на вход имя файла назначения.

4 Входные и выходные данные

4.1 Организация используемой входной информации

Входная информация – лог-файлы исполненной LuNA-программы, число узлов вычислительной системы, название файла для корректирующего распределения, директория и префикс для файлов-графиков квантованной нагрузки на вычислительные узлы.

4.2 Организация используемой выходной информации

Выходной информацией программы является файл с корректирующим распределением фрагментов вычислений на узлы вычислительной системы, представленным в виде карты, где ключом является идентификатор фрагмента вычислений, а значением – номер узла, на который фрагмент вычислений распределён.

5 Сообщения

В случае, если в лог-файле встречено событие, которое не указано среди поддерживаемых, программа выбрасывает исключение с текстом: «Unrecognizable event type *type_name*». В случае возникновения данного сообщения следует убедиться, что лог-файлы исполненной LuNA-программы содержат только поддерживаемые события, связанные с фрагментами вычислений.

6 Лист регистрации изменений

Таблица 6 – Лист регистрации изменении руководства программиста

Лист регистрации изменений									
Номера листов (страниц)					**Всего листов (страниц) в докум.**	**№ докумен -та**	**Входящий № сопроводит. докум. и дата**	**Подп.**	**Дата**
Изм.	**изменен-ных**	**заменен-ных**	**новых**	**аннули-рованных**					

ПРИЛОЖЕНИЕ Б

БАЛАНСИРОВЩИК НАГРУЗКИ НА ОСНОВЕ ПРОФИЛИРОВАНИЯ ДЛЯ СИСТЕМЫ «LuNA»

ОПИСАНИЕ ПРОГРАММЫ

Листов 14

Новосибирск 2021

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данном документе приведено описание балансировщика на основе профилирования в системе LuNA. Пользователь имеет возможность запускать построение корректирующего распределение фрагментов вычислений исполненной LuNA-программы на вычислительные узлы мультикомпьютера, а также строить графики квантованной нагрузки на вычислительные узлы.

Чтобы программа могла исполняться, необходимо наличие установленного интерпретатора Python версии 3.8 и выше, а также пакеты typing, subprocess, glob.

Оформление программного документа «Описание программы» произведено по требованиям ГОСТ 19.402-78 «ЕСПД. Описание программы» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

1 Общие сведения

1.1 Обозначение и наименование программы

Полное наименование программы – Балансировщик нагрузки на основе профилирования для системы «LuNA»

1.2 Программное обеспечение, необходимое для функционирования программы

Для функционирования программы необходимо наличие установленного интерпретатора Python версии 3.8 и выше, а также пакеты typing, subprocess, glob и утилита gnuplot.

1.3 Языки программирования

Исходным языком программы является Python версии 3.8. Средство разработки – редактор исходного кода PyCharm от компании JetBrains.

2 Функциональное назначение

2.1 Назначение программы

Программа предназначена для корректировки распределения фрагментов вычислений на вычислительные узлы мультикомпьютера на основе профилирования исполненных в системе LuNA-программ.

2.2 Сведения о функциональных ограничениях на применение

Программа не предназначена для запуска на платформах, не поддерживающих gnuplot, Python 3.8, а также пакеты typing, subprocess, glob.

3 Описание логической структуры

3.1 Алгоритм программы

Программа парсит лог-файлы исполненной LuNA-программы и собирает данные о исполнении фрагментов вычислений. Если формат лог-файлов не соответствует заявленному, пользователь получит сообщение об ошибке.

Если предыдущий этап завершился успехом, то программа применяет алгоритм балансировки к собранным данным о нагрузке фрагментов вычислений на вычислительные узлы мультикомпьютера, строит корректирующие распределение фрагментов вычислений и выводит его в файл.

3.2 Используемые методы

Алгоритм балансировки осуществляет квантование всего времени программы, подсчёт квантованного веса фрагментов вычислений на вычислительные узлы, подсчёт квантованной нагрузки на вычислительные узлы, после чего осуществляет перераспределение нагрузки и вычисляет, каким фрагментам вычислений необходимо скорректировать распределение.

Также на основе данных о квантованной нагрузке на узлы программа может строить графики, используя утилиту gnuplot с вызовом её из кода программа с помощью пакета subprocess.

3.3 Структура программы

Программа состоит из набора модулей, реализованных на языке Python. Полный список приведён в таблице 7.

Таблица 7 – Описание модулей программы

Модуль	**Описание**
parser	Содержит файлы parser.py, storage.py, node_load.py и cfevent_parser.py. В файле parser.py реализован класс Parser, в котором реализовано считывание лог-файлов исполненной LuNA-программы.

	В файле storage.py реализован класс Storage, который хранит считанные парсером данных о событиях, связанных с фрагментами вычислений в виде экземпляров класса NodeLoad. В файле node_load.py реализован класс NodeLoad, который хранит считанные парсером данные о событиях, связанных с фрагментами вычислений, исполненных на заданном вычислительном узле. хранениесчитывание лог-файлов исполненной LuNA-программы. В файле cfevent_parser.py реализован класс CfEventParser, реализующий преобразование данных, считанных парсером, во внутреннее представление событий, связанных с фрагментом вычислений (класс CfEvent).
cf	Содержит файлы cf.py, event.py. В файле cf.py реализован класс Cf, который хранит данные о фрагменте вычислений: узел, на котором он исполнялся, глобальный идентификатор фрагмента вычислений. В файле event.py реализован класс-перечислений Event, который необходим для определения поддерживаемых событий, связанных с фрагментом вычислений. В файле cf_tree реализован класс CfTree, реализующий хранение дерева порождений фрагментов вычислений.
cf_event	Содержит файлы cf_event.py и event.py.

	В файле event.py реализован класс-перечислений Event, который необходим для определения поддерживаемых событий, связанных с фрагментом вычислений. В файле cf_event.py реализован класс CfEvent, который хранит информацию о событии, связанном с фрагментом вычислений: Cf, тип событя, время события, вычислительный узел, на котором произошло событие.
Movement_recomendation	Содержит файлы cf_movet.py, load_data.py и recommendation_merger.py. В файле cf_move.py реализован класс CfMovement, который хранит информацию о скорректированном распределении фрагмента вычислений: Cf, узел, с которого происходит перемещение, узел, на который происходит перемещение. В файле load_data.py реализован класс LoadData, который осуществляет квантование нагрузки, а также производит перераспределение нагрузки по узлам и строит корректирующие распределение в виде набора элементов класса CfMovement. В файле recommendation_merger.py реализован класс RecommendationMerger, который производит запись корректирующего распределения в файл, а также производит слияние рекомендаций с рекомендациями, полученными на предыдущих этапах балансировки.
plot_builder	Содержит файл plot_builder.py.

	В файле plot_builder.py реализован класс PlotBuilder, который реализует построение графиков квантованной нагрузки на вычислительные узлы, используя подготовленный скрипт для gnuplot и пакет subprocess.

3.4 Связи между составными частями программы

Связь между модулями программы осуществляется при помощи вызова функций и методов объектов.

3.5 Связи программы с другими программами

Для запуска программы необходимо наличие установленного интерпретатора Python версии 3.8 и пакетов typing, subprocess, glob, а также утилиты gnuplot.

4 Используемые технические средства

Режим работы – консольное приложение.

Для работы программы необходимо устройство со следующими требованиями:

1. Оперативную память объемом 2 ГБ и выше.

5 Вызов и загрузка

Запуск программы осуществляется при помощи терминала. Необходимо ввести команду: python3 balancer.py.

6 Входные данные

Входные данные программы – число узлов, на которых исполнялась LuNA-программа, путь до директории с лог-файлами исполненной LuNA-программы, маска лог-файла и величина квантования.

7 Выходные данные

Выходными файлами являются файл с корректирующим распределением, представленный в виде карты, где ключом является идентификатор фрагмента вычислений, а значением – номер узла, на который фрагмент вычислений распределён, и файлы-графики квантованной нагрузки на вычислительные узлы.

8 Лист регистрации изменений

Таблица 8 – Лист регистрации изменении описания программы

Лист регистрации изменений									
Номера листов (страниц)					**Всего листов (страниц) в докум.**	**№ докумен -та**	**Входящий № сопроводит. докум. и дата**	**Подп.**	**Дата**
Изм.	**изменен-ных**	**заменен-ных**	**новых**	**аннули-рованных**					