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 ЭФФЕКТИВНЫХ МОДУЛЕЙ ИСПОЛНЕНИЯ ФРГАМЕНТИРОВАННЫХ ПРОГРАММ ЧАСТНОГО ВИДА ДЛЯ RUN-TIME СИСТЕМЫ ФРАГМЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

утверждена распоряжением проректора по учебной работе № 0493 от «23» декабря 2019г.

Чмиль Александр Владимирович, группа 16203 _______________

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

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э. /…………..
(ФИО) / (подпись)
«17» июня 2020г.

Руководитель ВКР
д.т.н, профессор,
зав. каф. ПВ ФИТ НГУ
Малышкин В.Э. /………...
(ФИО) / (подпись)
«17» июня 2020г.

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

Дата защиты: «……»………………2020г.

Новосибирск, 2020

МИНОБРНАУКИ РОССИИ
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

УТВЕРЖДАЮ

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

…………………
(подпись)
«23» декабря 2019г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студент*у (ке)*……Чмиль Александру Владимировичу…………, группы…16203…………
(фамилия, имя, отчество, номер группы)

Тема…Разработка эффективных модулей исполнения фрагментированных программ частного вида для run-time системы фрагментированного программирования………………
(полное название темы выпускной квалификационной работы бакалавра)

утверждена распоряжением проректора по учебной работе от…23 декабря 2019г...№ 0493

Срок сдачи студентом готовой работы…31 мая 2020 г.

Исходные данные (или цель работы)….разработать и реализовать в виде программных модулей системы фрагментированного программирования LuNA эффективные алгоритмы компиляции и исполнения фрагментированных программ заданного класса …………………

Структурные части работы… обзор литературы, постановка задачи, разработка и реализация алгоритмов, тестирование…………………………………………………………………….

Руководитель ВКР
зав. каф. ПВ ФИТ,
д.т.н., профессор
Малышкин В.Э. /…………..
(ФИ О) / (подпись)
«23» декабря 2019г.

Задание принял к исполнению
Чмиль А.В. /…………..
(ФИО студента) / (подпись)
«23» декабря 2019г.

Соруководитель
ст. преп. каф. ПВ ФИТ
Перепёлкин В.А. /…………..
(ФИ О) / (подпись)

«23» декабря 2019г.

СОДЕРЖАНИЕ

Определения, обозначения и сокращения 4

Введение 5

1 Разработка алгоритма балансировки нагрузки 11

1.1 Постановка задачи 11

1.2 Анализ алгоритмов для использования в runtime системе LuNA 11

1.3 Описание предполагаемого решения 11

1.4 Расширение языка LuNA 12

1.5 Основная идея адаптированного work-requesting алгоритма ... 13

1.5.1 Запрос фрагментов 13

1.5.2 Отправка фрагментов 13

1.6 Характеристика предполагаемого решения 14

2 Программная реализация алгоритма 16

2.1 Технические особенности LuNA 16

2.2 Компиляция в LuNA 16

2.2.1 Коммуникации в LuNA 17

2.2.2 Вычисления в LuNA 17

2.3 Особенности реализации 17

3 Тестирование 22

3.1 Тест на большое число фрагментов маленького размера 22

3.2 Тест на малое число фрагментов большого размера 23

3.3 Тест на большое число фрагментов большого размера 25

3.4 Результаты тестирования 26

Заключение 27

Список использованных источников и литературы 29

Приложение А 31

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Фрагмент данных – это переменная алгоритма. Он представлен в виде блока памяти.

Фрагмент вычислений (ФВ) – исполняемая единица, имеющая в качестве входных и выходных значений заданные фрагменты данных.

Балансировка нагрузки - метод распределения заданий между несколькими вычислительными узлами с целью оптимизации использования ресурсов, сокращения времени работа программы, а также горизонтального масштабирования программы.

ВВЕДЕНИЕ

При исполнении параллельных программ, востребованных в современных научных высокопроизводительных вычислениях, в силу неоднородности конфигурации вычислителя, динамики моделируемого явления, неэффективности написанной программы (излишние коммуникации, неудачная декомпозиция данных и/или вычислений и т.д.) нередко возникает дисбаланс вычислительной нагрузки.

Дисбаланс вычислительной нагрузки приводит к тому, что аппаратные ресурсы начинают использоваться неэффективно или вообще перестают использоваться.

При параллельных вычислениях требуется передавать данные между узлами. Задержки на передачу данных между узлами приводят к коммуникационному дисбалансу, при котором параллельная программа может работать медленнее своего последовательного аналога. Аппаратный дисбаланс может возникнуть в случае, если вычислительная система состоит из неоднородных аппаратных или программных ресурсов.

Устранение этих проблем – нетривиальная задача, требующая индивидуального подхода. Таким образом, существует потребность в автоматизации динамической балансировки нагрузки.

Балансировка нагрузки делится на статическую и динамическую.

Статическая балансировка нагрузки предполагает распределение заданий в начале исполнения программы. Она хорошо работает в программах, у которых время выполнения заданий не имеет сильную зависимость от входных данных. Задания в этих программах можно заранее распределить таким образом, чтобы загрузка узлов вычислительной системы была одинаковой, независимо от входных данных.

Но существует множество задач, время исполнения которых может существенно зависеть от данных, подаваемых на вход. Примером такой задачи является метод частиц в ячейках [1]. При расчете частиц возникает ряд новых проблем. Во-первых, число частиц в процессе решения динамически изменяется.

Частицы порождаются и уничтожаются неравномерно в различных ячейках сетки. Во-вторых, в ходе вычислений частицы перемещаются между ячейками в соответствии с уравнениями движения, распределяя по ним также неравномерно.

Такие задачи тоже можно распределить с помощью статической балансировки, но есть большая вероятность получить дисбаланс вычислительной нагрузки. Чтобы нивелировать этот дисбаланс используют динамическую балансировку нагрузки.

Алгоритмы динамической балансировки нагрузки предполагают передачу заданий с наиболее нагруженных узлов во время исполнения программы.

Можно выделить 2 основных вида алгоритмов динамической балансировки нагрузки:

1. Алгоритмы, учитывающие динамику исполняемой программы, информационные зависимости и близость данных;
2. алгоритмы, которые не учитывают эти свойства программы.

Алгоритмы этих двух типов можно использовать по отдельности, но их совместное использование позволяет добиться наилучшего результата. Алгоритмы первого вида выполняют более глобальную балансировку, тогда как алгоритмы второго типа лучше себя показывают при точечном дисбалансе.

Проблема заключается в том, что ни один алгоритм балансировки не является универсальным. Таким образом, существует потребность в инструментарии для автоматизации динамической балансировки нагрузки, который совмещает в себе несколько подходов к балансировке и адаптируется по ходу выполнения задачи.

Чтобы понять, как должен быть устроен инструментарий, важно исследовать подходы к автоматизации параллельных вычислений.

Одним из таких подходов является система LuNA.

LuNA (Language for Numerical Algorithms) - Экспериментальная система программирования, разрабатываемая в ИВМиМГ СО РАН и поддерживающая инструментально технологию фрагментированного программирования [2].

В состав системы входит:

1. Язык LuNA описания ФВ;
2. компилятор языка LuNA;
3. исполнительные системы;
4. вспомогательные инструменты (профилирование, трассировка, визуализация)

Удобство системы заключается в наличии простой системы коммуникаций между процессами и возможности использовать готовый механизм передачи фрагментов вычислений.

В runtime системе LuNA существует алгоритм динамической балансировки Rope-of-Beads [3], который учитывает особенности расположения данных. Этот алгоритм относится к алгоритмам 1 типа. Но в системе нет алгоритма, который позволяет реагировать на точечный дисбаланс. Поэтому для обзора существующих решений я взял за основу алгоритмы второго типа.

Основным отличием различных алгоритмов этого типа между собой является способ коммуникации между узлами для передачи задач.

Существует 3 основных алгоритма:

1. Work requesting [4]. Суть данного метода заключается в том, что узел, у которого нет задач запрашивает их у других процессов.
2. Work dealing [5]. В этом алгоритме узел, у которого много задач выполняет их перераспределение между другими узлами.
3. Work stealling [6]. В алгоритме этого метода узел, у которого нет задач, крадет их у другого узла.

Стратегия work-stealing дает распределение задач наиболее близкое к оптимальному. Эта стратегия очень популярна. Её можно встретить в планировщике языка программирования Cilk [7], Java Fork/Join Framework [8], .NET TPL [9] и так далее. Основным преимуществом этой стратегии является отсутствие прямых коммуникаций между узлами. Узлы могут взаимодействовать друг с другом только на этапе работы с разделяемой памятью, в которой находятся задачи.

Но, в отличие от стратегии work-dealing, в ней не учитывается близость данных, что негативно сказывается на работе алгоритме в системах, которые предназначены для работы на высокопроизводительных кластерах.

Еще одна проблема work-stealing планировщиков - при исполнении задач небольших размеров увеличивается количество краж. При кражах узел работает с разделяемой памятью и это приводит к большому числу межпроцессорных синхронизаций.

Эта проблема в work-requesting стратегии решается перемещением по запросу несколько имеющихся задач.

В таблице 1 перечислены основные преимущества и недостатки существующих алгоритмов.

Алгоритм балансировки	+	-
Work requesting	Возможность передачи задач группами, коммуникация происходит только когда случается дисбаланс.	Нельзя учитывать близость данных.
Work dealing	Возможность учитывать близость данных при балансировке, возможность передачи задач группами.	В зависимости от политики коммуникации загрузка на узлы либо не учитывается, либо для того, чтобы актуализировать состояние загруженности узлов, требуются постоянные фоновые коммуникация.
Work stealing	Использование разделяемой памяти.	Большое количество синхронизаций при выполнении небольших задач

Таблица 1 – обзор преимуществ и недостатков существующих решений

Цель работы заключается в разработке и реализации эффективного алгоритма динамической балансировки нагрузки для runtime системы LuNA.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Исследование преимуществ и недостатков существующих алгоритмов;
2. поддержка динамической балансировки в языке LuNA;
3. разработка и реализация алгоритма балансировки в run-time системе LuNA;

4. исследование изменения скорости работы программ при применении алгоритма динамической балансировки на различных примерах с выполнением статической балансировки и без неё, а также с различным уровнем дисбаланса в фрагментах.

1 Разработка алгоритма балансировки нагрузки

1.1 Постановка задачи

В задачах балансировки нагрузка распределяется между процессорами согласно следующим критериям:

1. Равномерное распределение задач по всем процессорам;
2. высокая производительность системы для всех приложений;
3. низкое время ответа системы на каждый запрос;
4. быстрое выполнение задач согласно расписанию.

Реализуемая мной динамическая балансировка нагрузки будет использоваться для точечной балансировки, в дополнении к алгоритму балансировки Rope-of-Beads.

1.2 Анализ алгоритмов для использования в runtime системе LuNA

Программа на языке LuNA состоит из фрагментов данных и фрагментов вычислений. В ходе работы программы фрагменты динамически порождаются и мигрируют. Выравнивание нагрузки осуществляется путем перераспределения фрагментов вычислений по узлам.

Исходя из этих свойств системы мною был адаптирован алгоритм work-requesting для LuNA. Этот алгоритм был выбран по причине того, что он лучше всего адаптируется под особенности LuNA.

Если рассматривать work-stealing алгоритм, то можно столкнуться с проблемой, что в системе LuNA узел имеет доступ только к своей очереди задач. Поэтому для реализации такого алгоритма потребуется использовать дополнительные механизмы коммуникации и синхронизации, как например разделяемая память. Что может негативно сказаться на производительность системы.

Work-dealing алгоритм в зависимости от политики распределения задач либо не учитывает загруженность узлов, что, по сути, делает его аналогичным статической балансировке, либо учитывает, но для этого требуются постоянные коммуникации между процессами.

1.3 Описание предполагаемого решения

Предлагается использование следующего алгоритма:

1. За основу взять стратегию work-requesting;
2. запросы на фрагменты осуществлять к соседним узлам;
3. добавить возможность с помощью специальной аннотации самостоятельно помечать фрагменты вычислений, которые могут участвовать в динамической балансировке нагрузки;
4. обрабатывать случай, когда в момент запроса фрагментов не было, но позже они появились.

1.4 Расширение языка LuNA

Программа на языке LuNA — это описание множества фрагментов вычислений. В описании фрагмента расположены различные рекомендации по обработке фрагментов - аннотации. Пример программы на языке LuNA представлен на листинге 1. Аннотации находится на 10-11 строках.

```
 1 /*
 2  Hello world example.
 3 */
 4
 5 import c_helloworld() as hello_world;
 6
 7 sub main()
 8 {
 9         hello_world() @ {
10              locator_cyclic: 0;
11              use_for_balance;
12          };
13 };
```

Листинг 1 – пример программы на языке LuNA

Чтобы алгоритм динамической балансировки работал эффективнее, время исполнения фрагментов, участвующих в балансировке, должно превышать время на их передачу при балансировке. Но реализуемый мною алгоритм не имеет никакой информации по поводу фрагмента вычисления, который он исполняет. Для того чтобы эта информация появилась было решено расширить язык LuNA специальной аннотацией. Эта аннотация позволяет разработчикам программы на языке LuNA указывать фрагменты, которые рекомендуется задействовать в балансировке. Пример аннотации можно видеть на листинге 1, она находится на 11 строке.

1.5 Основная идея адаптированного work-requesting алгоритма

1.5.1 Запрос фрагментов

1. В случае отсутствия ФВ в очереди задач на исполнения производится запрос ФВ на некоторое количество узлов;
2. при получении ФВ происходит отправка отмены запроса.

На рисунке 21 изображен алгоритм запроса фрагмента в виде блок схемы.

Рисунок 1 – блок схема алгоритма запроса фрагмента

1.5.2 Отправка фрагментов

При исполнении ФВ проверять, помечен ли он специальной аннотацией, есть ли запросы на передачу фрагментов и наличие задач на исполнение. В случае если условия выполняются – передавать фрагмент запросившему узлу.

На рисунке 2 изображен алгоритм отправки фрагмента в виде блок схемы.

Рисунок 2 – блок схема алгоритма отправки фрагмента

1.6 Характеристика предполагаемого решения

Данное решение обнаруживает локальный дисбаланс в момент отсутствия нагрузки на узел и точечно производит выравнивание нагрузки. Это свойство позволяет его использовать для совместной работы с алгоритмом балансировки

Rope-of-Beads. Также решение использует ограниченное количество коммуникаций, связанных с динамической балансировкой, за счет использования стратегии work-requesting, что способствует сокращению издержек на коммуникации.

Оно является довольно гибким, так как позволяет разработчику самостоятельно определять фрагменты, участвующие в балансировке.

2 Программная реализация алгоритма

2.1 Технические особенности LuNA

2.2 Компиляция в LuNA

Компилятор переводит программный код с языка LuNA в код на языке C++, в котором каждый ФВ представлен в виде отдельной функции - блока. Пример скомпилированной программы представлен на листинге 2.

```
1  #include "ucenv.h"
2
3 extern "C" void c_helloworld(); // as hello_world
4
5 // MAIN
6 BlockRetStatus block_0(CF &self)
7 {
8
9         self.NextBlock=1;
10        return CONTINUE;
11 }
12
13 // STRUCT: sub main()
14 BlockRetStatus block_1(CF &self)
15 {
16
17 // GEN BODY: sub main()
18        { // FORK_BI: cf _17: hello_world();
19                CF *child=self.fork(2);
20        }
21        return EXIT;
22 }
23
24 // BI_EXEC: cf _17: hello_world();
25 BlockRetStatus block_2(CF &self)
26 {
27        if (self.migrate(CyclicLocator(0))) {
28                return MIGRATE;
29        }
30
31        {
32                // EXEC_EXTERN cf _17: hello_world();
33                c_helloworld();
34        }
35        return EXIT;
36 }
37 extern "C" void init_blocks(BlocksAppender add)
38 {
39        bool ok=true;
40        ok = ok && add(block_0)==0;
41        ok = ok && add(block_1)==1;
42        ok = ok && add(block_2)==2;
43
44        assert(ok);
45 }
```

Листинг 2 – пример скомпилированной LuNA программы

Внутри блока могут создаваться другие блоки с помощью функции *fork (int block_num)*.

Блок может возвращать следующие элементы перечисления:

1. *MIGRATE* – перед возвратом этого элемента в специальную переменную проставляется значение узла, на который нужно переслать фрагмент;
2. *CONTINUE* – продолжать исполнения с блока, который указан в переменной *NextBlock;*
3. *EXIT* – завершить исполнение этой последовательности блоков.

2.2.1 Коммуникации в LuNA

В системе LuNA коммуникации осуществляются с помощью MPI [10] и сериализации сообщений в байтовое представление.

Каждое сообщение помещается тегом. В зависимости от тега сообщения отличаются его способы обработки.

2.2.2 Вычисления в LuNA

За вычисления в системе отвечает пул потоков конфигурируемого размера. Пул потоков обрабатывает задания из очереди задач. Коммуникации между потоками во время работы с очередью осуществляются с помощью таких примитивов синхронизации, как условная переменная и мьютекс.

2.3 Особенности реализации

Реализация состоит из 2 основных частей:

1. Добавление в компилятор языка LuNA генерации кода внутри фрагмента вычислений, в котором происходит проверка наличия запросов ФВ от других узлов;
2. поддержка в runtime системе LuNA механизма для порождения и отправки запросов на балансировку.

Для поддержки рекомендаций в компиляторе было решено в начале исполнительных блоков фрагментов, которые помечены рекомендацией *balance*, добавлять код, который определяет, должен ли фрагмент подвергаться балансировке.

На листинге 3 представлен фрагмент вычислений со сгенерированным дополнительным кодом. Сгенерированный дополнительный код находится на 4-6 строке.

```
 1 // BI_EXEC: cf _l7: hello_world();
 2 BlockRetStatus block_2(CF &self)
 3 {
 4 if (self.migrate(CyclicLocator(0))) {
 5                 return MIGRATE;
 6 }
 7 if (self.need_to_balance()) {
 8                 return BALANCE;
 9 }
10 {
11                 // EXEC_EXTERN cf _l7: hello_world();
12                 c_helloworld();
13 }
14         return EXIT;
15 }
```

Листинг 3 – фрагмент вычислений со сгенерированным кодом

Этот код срабатывает в runtime системе во время исполнения фрагмента вычисления. Метод *need_to_balance()* проверяет, есть ли в очереди запросы на передачу вычислительных фрагментов от недогруженных узлов. В случае если этот запрос есть, в фрагмент вычисления устанавливается номер узла, который сделал запрос и возвращается значение типа перечисление *BALANCE*.

Runtime система анализирует значение, которое было возвращено при исполнении фрагмента. В случае если значение равно *BALANCE*, система выполняет отправку фрагмента на узел, номер которого мы указали до этого.

Для того чтобы недогруженные узлы могли исполнять запросы на другие узлы в runtime систему были добавлены специальные теги, которым помечаются сообщения: *TAG_BALANCE* и *TAG_BALANCE_REVOKE*.

Когда узлу приходит сообщений с тегом *TAG_BALANCE* он добавляет в очередь запросов на передачу вычислительных фрагментов идентификатор узла, отправившего сообщение.

Когда узлу приходит сообщений с тегом *TAG_BALANCE_REVOKE* он удаляет из очереди запросов на передачу вычислительных фрагментов идентификатор узла, отправившего сообщение.

Для определения, когда нужно отправлять запрос на балансировку в пул потоков системы был добавлены callback’и. На листинге 4 можно увидеть callback, который срабатывает при опустошении очереди задачи. На листинге 5 представлен callback, срабатывающий при появлении задач в очереди.

```
1 pool_.on_empty([this, proc_count]() {
2     if (!need_jobs_) {
3         need_jobs_ = true;
4         balance_req(TAG_BALANCE, proc_count);
5     }
6 });
```

Листинг 4 – Callback, который срабатывает при опустошении очереди задач

```
1 pool_.on_submit([this, proc_count]() {
2     if (need_jobs_) {
3             need_jobs_ = false;
4             balance_req(TAG_BALANCE_REVOKE, proc_count);
5         }
6 });
```

Листинг 5 – Callback, который срабатывает при появлении задач в очереди задач.

Callback *on_empty* проверяет значение флага *need_jobs_*. В начале работы runtime системы он имеет значение *false*. Если флаг в значении *false* он устанавливает его в значение *true* и отправляет запросы соседним узлам на получение задач с тегом *TAG_BALANCE*. Флаг требуется для того, чтобы отмена запроса не отправлялась при каждом добавлении задач в thread pool.

Callback *on_submit* проверяет значение флага *need_jobs*. Если флаг в значении *true*, он устанавливает его в значение *false* и отправляет отмену запроса на получение задач соседним узлам с тегом *TAG_BALANCE_REVOKE*.

На рисунках 3, 4, 5 демонстрируется работа алгоритма.

На рисунке 3 изображено состояние runtime системы в момент, когда произошел дисбаланс на 2 узле. На 1 узле есть фрагмент вычисления, который может участвовать в балансировке. Это фрагмент под номером 3.

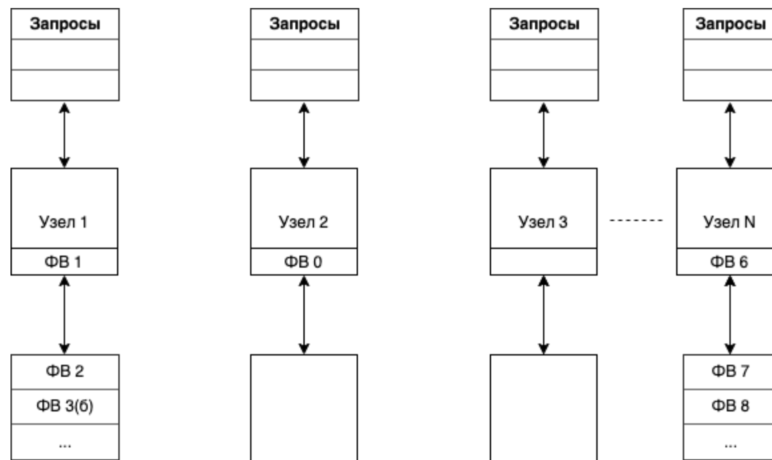

Рисунок 3 – Состояние runtime системы при появившемся дисбалансе

На рисунке 4 второй узел делает запрос на получение фрагмента вычислений на соседние узлы. Эти узлы записывают запрос в специальную очередь.

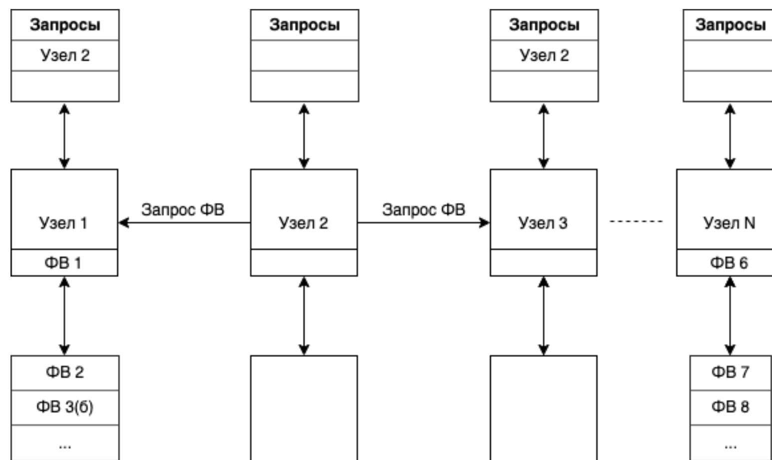

Рисунок 4 – Состояние системы в момент запроса ФВ для устранения дисбаланса

Рисунок 5 демонстрирует состояние системы в момент устранения дисбаланса. 1 узел начал исполнять фрагмент вычисления под номером 3, обнаружил что этот фрагмент участвует в балансировке и отправил его на второй узел. Второй узел после получения фрагмента отправил отмену запроса третьему узлу. Третий узел убрал запрос на получение фрагмента от 2 узла из очереди запросов.

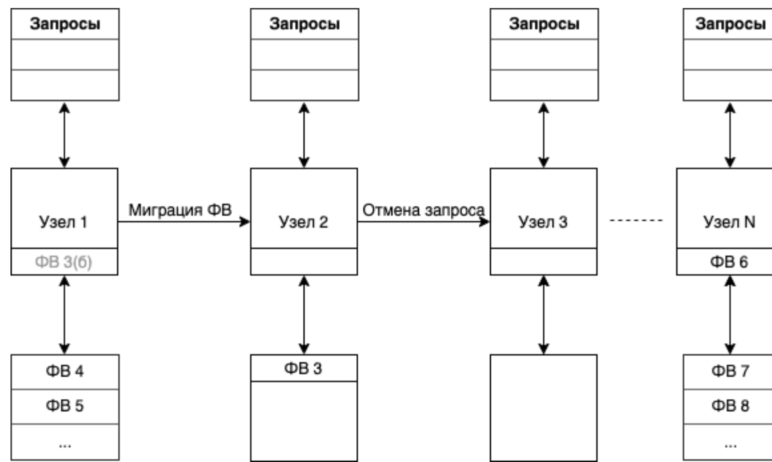

Рисунок 5 – Состояние системы после устранения дисбаланса

3 Тестирование

Тестирование проводилось на высокопроизводительном кластере МВС-10П МСЦ РАН [11]. Для тестирования была взята задача перемножения матриц.

Во время тестирования исследовалось поведение динамической балансировки при отсутствии статической балансировки, при плохой статической балансировке (нагружено только половина узлов) и при хорошей. Также исследовалась зависимость от количества фрагментов вычислений и их размера.

В результате тестирования мы должны увидеть, как начальное расположение фрагментов, их количество и размер влияют на эффективность балансировки.

Теоретически, чем больше размер фрагментов и сильнее дисбаланс - тем эффективнее будет работать динамическая балансировка нагрузки.

3.1 Тест на большое число фрагментов маленького размера

Параметры теста:

1. Размер фрагмента: 100;
2. количество фрагментов: 40.

Данный тест направлен на исследование эффективности балансировки при работе программы с большим числом фрагментов вычислений небольшого размера.

Результаты этого теста отображены в таблицах 2 и 3.

Вариант статической балансировки	С динамической балансировкой	Без динамической балансировки
Без статической балансировки	17.58 сек.	28.6 сек.
Со статической балансировкой на половину узлов	19.23 сек.	19.78 сек.
Со статической балансировкой на все узлы	20.3 сек.	16.7 сек

Таблица 2 – результаты 1 теста

Вариант статической балансировки	Ускорение
Без статической балансировки	38.5 %
Со статической балансировкой на половину узлов	2.78 %
Со статической балансировкой на все узлы	−21.56 %

Таблица 3 – ускорение в 1 тесте при использовании динамической балансировки

Видно, что динамическая балансировка дает эффект только в том случае, когда не было статической балансировки и, следовательно, дисбаланс был максимален.

При статической балансировке на половину узлов эффективность динамической балансировки становится минимальной.

При статической балансировке на все узлы видно, что происходит замедление программы при использовании динамической балансировки. Это связано с тем, что затраты на коммуникацию начинают превосходить затраты на исполнение.

3.2 Тест на малое число фрагментов большого размера

Параметры теста:

1. Размер фрагмента: 200;
2. количество фрагментов: 20.

В этом тесте был увеличен размер каждого фрагмента, но уменьшено их количество. Этот тест моделирует ситуацию, когда произошел точечный дисбаланс на небольшой группе узлов.

В таблицах 4 и 5 отмечены результаты 2 теста.

Вариант статической балансировки	С динамической балансировкой	Без динамической балансировки
Без статической балансировки	17.1 сек.	27 сек.
Со статической балансировкой на половину узлов	14.62 сек.	18.68 сек.
Со статической балансировкой на все узлы	12.57 сек	14.94 сек.

Таблица 4 – результаты 2 теста

Вариант статической балансировки	Ускорение
Без статической балансировки	36.7 %
Со статической балансировкой на половину узлов	21.7 %
Со статической балансировкой на все узлы	15.86 %

Таблица 5 – ускорение во 2 тесте при использовании динамической балансировки

В этом тесте был увеличен размер каждого фрагмента, но уменьшено их количество. Этот тест моделирует ситуацию, когда произошел точечный дисбаланс на небольшой группе узлов.

Видно, что динамическая балансировка ускоряет работу программы при всех вариантах использования статической балансировки. Это происходит,

потому что при использовании статической балансировки происходит дисбаланс из-за затрат на коммуникации при начальном распределении задач.

3.3 Тест на большое число фрагментов большого размера

Параметры теста:

1. Размер фрагмента: 200;
2. количество фрагментов: 40.

В этом тесте исследуется ситуация, когда происходит более масштабный дисбаланс. То есть дисбаланс на большое количество фрагментов, которые долго исполняются

Таблицы 6 и 7 демонстрируют результаты этого теста.

Вариант статической балансировки	С динамической балансировкой	Без динамической балансировки
Без статической балансировки	122.9 сек.	183.38 сек.
Со статической балансировкой на половину узлов	88 сек.	114.1 сек.
Со статической балансировкой на все узлы	75.65 сек.	84.3 сек.

Таблица 6 – результаты 1 теста

Вариант статической балансировки	Ускорение
Без статической балансировки	32,98 %
Со статической балансировкой на половину узлов	22,87 %
Со статической балансировкой на все узлы	10,26 %

Таблица 7 – ускорение в 3 тесте при использовании динамической балансировки

Результаты эффективности динамической балансировки в этом тесте ухудшились по сравнению с предыдущим. Это связано с тем, что помимо увеличения размера передаваемого фрагмента, также увеличились фрагменты вычислений, которые должны быть выполнены последовательно перед передаваемым фрагментом. Одним из таких фрагментов является инициализация частей массива. Из-за этого запросившему узлу приходится дольше ждать, пока выполнятся фрагменты, которые не участвуют в балансировке.

3.4 Результаты тестирования

По результатам тестов можно увидеть, что динамическая балансировка ускоряет программу даже в том случае, если она была изначально сбалансирована статической балансировкой. Причиной этого являются коммуникации, которые возникают при распределении задач во время использования статической балансировки нагрузки и вызывают дисбаланс.

Результаты тестирования соответствуют ожидаемым. Балансировка лучше работает в случаях, когда дисбаланс происходит на малом числе больших фрагментов, чего и хотелось достичь для выполнения задачи.

ЗАКЛЮЧЕНИЕ

В ходе работы алгоритм work-requesting был адаптирован под runtime систему LuNA. Адаптированный алгоритм был реализован и протестирован.

Также был расширен язык LuNA с поддержкой изменений в компиляторе и runtime системе.

Тестирование показало, что алгоритм справляется с исполнением задач, на которые он был рассчитан - динамическая балансировка больших фрагментов вычислений при точечном дисбалансе.

Данная работа была опубликована на 58-й Международной научно-студенческой конференции, г. Новосибирск, 2020 г. и была награждена дипломом второй степени [12].

Защищаемые положения:

1. Разработан алгоритм динамической балансировки нагрузки на вычислительные узлы на основе подхода work-requesting
2. реализован алгоритм динамической балансировки нагрузки в runtime системе LuNA;
3. проведено экспериментальное исследование производительности runtime системы LuNA с разработанным алгоритмом динамической балансировки нагрузки.

В дальнейшем планируется развитие динамического балансировщика нагрузки. Возможными направлениями для развития являются:

1. Настройка параметров балансировки, в том числе автоматическая;
2. реализация различных паттернов запроса ФВ.

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».

Чмиль Александр Владимирович ______________

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

«31» мая 2020 г.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Андрианов А. Н., Ефимкин К. Н. Подход к параллельной реализации метода частиц в ячейках //Препринты Института прикладной математики им. МВ Келдыша РАН. – 2009. – №. 0. – С. 9-20.
2. Малышкин В. Э. Технология фрагментированного программирования //Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. – №. 46 (305).
3. Malyshkin V. E., Perepelkin V. A., Schukin G. A. Distributed algorithm of data allocation in the fragmented programming system LuNA //International Conference on Parallel Computing Technologies. – Springer, Cham, 2015. – С. 80-85.
4. Acar U. A., Charguéraud A., Rainey M. Scheduling parallel programs by work stealing with private deques //ACM SIGPLAN Notices. – ACM, 2013. – Т. 48. – №. 8. – С. 219-228.
5. Hendler D., Shavit N. Work dealing //Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures. – ACM, 2002. – С. 164-172.
6. Cederman D., Tsigas P. Dynamic load balancing using work-stealing //GPU Computing Gems Jade Edition. – Morgan Kaufmann, 2012. – С. 485-499.
7. Blumofe R. D. et al. Cilk: An efficient multithreaded runtime system //Journal of parallel and distributed computing. – 1996. – Т. 37. – №. 1. – С. 55-69.
8. Lea D. A Java fork/join framework //Proceedings of the ACM 2000 conference on Java Grande. – 2000. – С. 36-43.
9. Leijen D., Schulte W., Burckhardt S. The design of a task parallel library //Acm Sigplan Notices. – 2009. – Т. 44. – №. 10. – С. 227-242.
10. MPI Documents. Официальная документация стандарта MPI [Электронный ресурс] / – Режим доступа: https://www.mpi-forum.org/docs/ (дата обращения 15.05.2020).
11. Межведомственный Суперкомпьютерный Центр Российской Академии Наук [Электронный ресурс] / – Режим доступа: http://www.jscc.ru/ (дата обращения 15.05.2020).

12. Чмиль А.В. Разработка динамического балансировщика нагрузки для runtime-системы LuNA // Информационные технологии : Материалы 58-й Междунар. науч. студ. конф. 10–13 апреля 2020 г. / Новосиб. гос. ун-т. — Новосибирск : ИПЦ НГУ, 2020. — 48

ПРИЛОЖЕНИЕ А

БАЛАНСИРОВЩИК НАГРУЗКИ ДЛЯ RUNTIME СИСТЕМЫ “LuNA”

РУКОВОДСТВО ПРОГРАММИСТА

Листов 7

Новосибирск 2020

СОДЕРЖАНИЕ

АННОТАЦИЯ **33**

1 НАЗНАЧЕНИЕ И УСЛОВИЯ ПРИМЕНЕНИЯ ПРОГРАММЫ **34**

1.1 НАЗНАЧЕНИЕ ПРОГРАММЫ 34

1.2 ФУНКЦИИ, ВЫПОЛНЯЕМЫЕ ПРОГРАММОЙ 34

1.3 УСЛОВИЯ, НЕОБХОДИМЫЕ ДЛЯ ВЫПОЛНЕНИЯ ПРОГРАМЫ 34

2 ХАРАКТЕРИСТИКА ПРОГРАММЫ **34**

2.1 ОПИСАНИЕ ОСНОВНЫХ ХАРАКТЕРИСТИК ПРОГРАММЫ 34

2.1.1 Режим работы программы 34

2.1.2 Средства контроля правильности выполнения программы 34

2.2 ОПИСАНИЕ ОСНОВНЫХ ОСОБЕННОСТЕЙ ПРОГРАММЫ 35

3 ОБРАЩЕНИЕ К ПРОГРАММЕ **35**

3.1 ОПИСАНИЕ ПРОЦЕДУР ВЫЗОВА ПРОГРАММЫ 35

3.1.1 Компиляция с выбором фрагментов, участвующих в балансировке 35

3.1.2 Запуск runtime системы LuNA с включенной балансировкой нагрузки ... 35

4 ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ **35**

4.1 ОРГАНИЗАЦИЯ ИСПОЛЬЗУЕМОЙ ВХОДНОЙ ИНФОРМАЦИИ 35

4.1.1 Организация используемой входной информации на этапе компиляции . 36

4.1.2 Организация используемой входной информации на этапе исполнения программы 36

4.2 ОРГАНИЗАЦИЯ ИСПОЛЬЗУЕМОЙ ВЫХОДНОЙ ИНФОРМАЦИИ 36

4.2.1 Организация используемой выходной информации на этапе компиляции 36

4.2.2 Организация используемой выходной информации на этапе исполнения 36

5 СООБЩЕНИЯ **36**

АННОТАЦИЯ

В данном программном документе приведено руководство программиста для динамического балансировщика нагрузки для runtime системы “LuNA”. Исходным языком программы является C++. Средство разработки – редактор исходного кода CLion от компании JetBrains.

Основными функциями программы является балансировка нагрузки при точечном дисбалансе.

Оформление программного документа «Руководство программиста» произведено по требованиям ГОСТ 19.504-79 «ЕСПД. Руководство программиста» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

4 Назначение и условия применения программы

4.1 Назначение программы

Балансировщик нагрузки входит в состав комплексной runtime системы LuNA. Система LuNA предназначена для исполнения фрагментированных программ на высокопроизводительных кластерах. Система LuNA позволяет исполнять фрагментированную программу на нескольких узлах вычислительного кластера. Балансировщик нагрузки используется для динамической балансировки нагрузки в случаях точечного дисбаланса.

4.2 Функции, выполняемые программой

1. Расширение компиляции языка LuNA генерацией кода;
2. отслеживание загруженности узлов;
3. коммуникация между узлами для запроса фрагментов;
4. коммуникация между узлами для получения фрагментов.

4.3 Условия, необходимые для выполнения програмы

1. Запуск runtime системы со специальным ключом “-b”;
2. использование во фрагментах вычислений специальной аннотации *use_for_balance*;
3. запуск runtime системы на двух и более MPI процессах.

5 Характеристика программы

5.1 Описание основных характеристик программы

Балансировщик нагрузки входит в состав комплексной runtime системы LuNA и обеспечивает динамическую балансировку нагрузки в случаях возникновения точечного дисбаланса.

5.1.1 Режим работы программы

Работа балансировщика нагрузки может осуществляться в двух режимах.

Режимы работы регулирует специальный ключ исполнения “-b”.

При наличии ключа динамическая балансировка срабатывает. В ином случае, при точечном дисбалансе динамическая балансировка не производится.

5.1.2 Средства контроля правильности выполнения программы

Контроль правильности работы балансировщика нагрузки осуществляется встроенными средствами runtime системы, реализованных в виде: протоколирование событий, осуществление диагностики с помощью профилировщика системы, тестирования системы на этапе компиляции.

5.2 Описание основных особенностей программы

Данный балансировщик нагрузки осуществляет локальную балансировку нагрузки путем передачи фрагментов на основе алгоритма work-requesting.

Балансировщик предоставляет возможность ручного выбора фрагментов вычислений, которые подлежат балансировке.

6 Обращение к программе

6.1 Описание процедур вызова программы

6.1.1 Компиляция с выбором фрагментов, участвующих в балансировке

Для выбора фрагментов, которые участвую в балансировке, требуется использовать специальную аннотацию *use_for_balance*. Пример использования аннотации представлен в листинге А.1 на 11 строке.

```
 1 /*
 2  Hello world example.
 3 */
 4
 5 import c_helloworld() as hello_world;
 6
 7 sub main()
 8 {
 9         hello_world() @ {
10              locator_cyclic: 0;
11              use_for_balance;
12          };
13 }
```

Листинг А.1 – пример программы с аннотацией LuNA

6.1.2 Запуск runtime системы LuNA с включенной балансировкой нагрузки

Для запуска с включенным балансировщиком нагрузки требуется использовать специальный ключ запуска “-b”.

7 Входные и выходные данные

7.1 Организация используемой входной информации

7.1.1 Организация используемой входной информации на этапе компиляции

На этапе компиляции входной информацией является наличие аннотации *use_for_balance*.

7.1.2 Организация используемой входной информации на этапе исполнения программы

На этапе исполнения программы входной информацией является: состояние очереди исполняемых фрагментов вычислений, элементы перечислений *TAG_BALANCE и TAG_BALANCE_REVOKE,* получаемые путем приема сообщений от другого узла с помощью MPI и десериализации сообщения.

7.2 Организация используемой выходной информации

7.2.1 Организация используемой выходной информации на этапе компиляции

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

7.2.2 Организация используемой выходной информации на этапе исполнения

На этапе исполнения программы выходной информацией является: элементы перечислений *TAG_BALANCE и TAG_BALANCE_REVOKE,* отправляемые другим узлам с помощью MPI и десериализации сообщения. Элемент перечисления *BALANCE,* возвращаемый при исполнении фрагмента вычислений.

8 Сообщения

В протоколе исполнения runtime системы LuNA при балансировке записываются следующие сообщения:

1. “Balancing job <ИДЕНТИФИКАТОР ФРАГМЕНТА ВЫЧИСЛЕНИЯ> to <ИДЕНТИФИКАТОР УЗЛА ПОЛУЧАТЕЛЯ>”
2. “Balance request from <ИДЕНТИФКАТОР УЗЛА ОТПРАВИТЕЛЯ>”

3. “Balance request revoke from <ИДЕНТИФКАТОР УЗЛА ОТПРАВИТЕЛЯ>”