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 09.04.01 Информатика и вычислительная техника
Направленность (профиль): Технология разработки программных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Чмиль Александра Владимировича

Тема работы:

РАЗРАБОТКА ПОДСИСТЕМЫ АВТОМАТИЗИРОВАННОГО ПРИМЕНЕНИЯ АЛГОРИТМОВ ДИНАМИЧЕСКОЙ БАЛАНСИРОВКИ НАГРУЗКИ ДЛЯ СИСТЕМЫ LUNA

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В. Э. /……..
(ФИО) / (подпись)
«31» мая 2022 г.

Руководитель ВКР
д.т.н, профессор,
зав. каф. ПВ ФИТ НГУ
Малышкин В.Э. /……...
(ФИО) / (подпись)
«31» мая 2022г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Перепёлкин В.А. /……...
(ФИО) / (подпись)
«31» мая 2022г.

Новосибирск, 2022

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: 09.04.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА
Направленность (профиль): Технология разработки программных систем

УТВЕРЖДАЮ

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

…….
(подпись)
«20» января 2022 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ МАГИСТРА

Студент*у(ке)*……Чмиль Александру Владимировичу……, группы…20221……
(фамилия, имя, отчество, номер группы)

Тема Разработка подсистемы автоматизированного применения алгоритмов динамической балансировки нагрузки для системы LuNA
(полное название темы выпускной квалификационной работы магистра)

утверждена распоряжением проректора по учебной работе от 20 января 2022г. № 0011
Срок сдачи студентом готовой работы 20 мая 2022 г.
Исходные данные (или цель работы) …разработать подсистему автоматизированного применения алгоритмов динамической балансировки нагрузки для системы LuNA
Структурные части работы …обзор литературы, постановка задачи, разработка и реализация подсистемы, тестирование

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

«20» января 2022 г.

Задание принял к исполнению
Чмиль А.В./……..
(ФИО студента) / (подпись)

«20» января 2022 г.

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

«20» января 2022 г.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Фрагмент данных (ФД) – 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

Дисбаланс вычислительной нагрузки является одной из ключевых проблем параллельного программирования. Дисбаланс вычислительной нагрузки может происходить из-за различных факторов. Причины возникновения дисбаланса могут носить как аппаратный, так и программный характер. Аппаратный дисбаланс возникает вследствие неоднородности ресурсов вычислительной системы. Программный дисбаланс связан с такими факторами как динамика моделируемого явления и неэффективно распараллеленная программа, содержащая излишние коммуникации, неудачно распределение вычислений между узлами и так далее.

Устранение дисбаланса является нетривиальной задачей, для решения которой не существует единого подхода. Существует множество алгоритмов, направленных на устранение дисбаланса, но ни один из них не является универсальным.

Возникает проблема того, как разработчикам распределенных программ осуществлять подбор подходящего для задачи алгоритма балансировки или реализовать свой собственный алгоритм. Поэтому разработчикам распределенных программ важно иметь широкий выбор алгоритмов балансировки и инструментарий, позволяющий автоматизировано выбирать подходящий для конкретной задачи алгоритм, а также механизм добавления в систему пользовательских балансировщиков.

Решением является автоматизация обеспечения динамической балансировки вычислительной нагрузки. Под автоматизацией в данном случае понимается ситуация, когда различные методы, алгоритмы и программы, осуществляющие динамическую балансировку нагрузки, накапливаются в некоторой базе или библиотеке в виде, допускающем их автоматическое применение, а пользователь представляет свою программу в таком виде, при котором соответствующие методы, алгоритмы и программы применяются автоматически, без необходимости пользователя глубоко понимать проблематику динамической балансировки

нагрузки. Вырожденным случаем является поддержка динамической балансировки вычислительной нагрузки в ПО, таком как Charm++ [1] или PICADOR [2]. Не вырожденным случаем являлась бы ситуация, когда система программирования не является специализированной, а в библиотеке доступны и автоматически применяются накопленные человечеством знания о динамической балансировке нагрузки.

Одной из систем, ориентированных на подобный инструментарий и накопление алгоритмов балансировки является система LuNA [3]. В системе присутствуют различные алгоритмы балансировки, но нет возможности автоматизировано управлять ими и отсутствует стандартизированный механизм добавления алгоритмов балансировки.

Целью магистерской работы является автоматизация применения алгоритмов динамической балансировки нагрузки для системы LuNA.

Для достижения этой цели требуется разработать инструментарий, позволяющий автоматизировать процесс выбора алгоритма и параметров балансировки нагрузки и разработать механизм добавления в систему пользовательских балансировок.

Научной новизной данной работы является исследование существующих механизмов автоматизации выбора алгоритма и параметров балансировки нагрузки и реализация на основе исследования аналогичного механизма для системы фрагментированного программирования LuNA.

1 Обзор средств автоматизации и добавления пользовательских балансировок в существующих системах параллельного программирования

Существующие распределенные системы, имеющие несколько алгоритмов балансировки решают проблему автоматизированного управления балансировки с помощью сущности, управляющей ходом балансировки.

1.1 Charm++

В Charm++ за принятие решение о том о том, когда запускать балансировку нагрузки отвечает метабалансировщик. Принятие решение происходит на основе модели линейного предсказания.

Так же в Charm++ присутствует класс метабалансировщик – LBManager. Он отвечает за инициализацию и управление алгоритмами балансировки, исходя из настроек системы.

Выбор и настройка алгоритмов балансировки происходит с помощью ключей запуска программ. Система расширяема пользовательскими алгоритмами балансировки с помощью статической и динамической линковки.

Интерфейс, который реализуют пользовательские балансировки, содержит:

1. Статистику по времени работы CPU на узле;
2. статистику по ходу исполнения, такую как количество мигрировавших объектов, количество объектов, находящихся на узле;
3. функцию миграции для объектов;

Коммуникация между узлами происходит в фоне, в результате коммуникации у каждого узла есть информация о нагрузке других узлов.

1.2 StarPU

В StarPU [4] системе настройки алгоритмов хранятся в Context классах. Есть возможность изменять параметры контекста балансировки с помощью Hypervisor’a, который управляется пользовательским вводом и вызовом методов из кода. Выбор

и настройка алгоритмов балансировки происходит с помощью переменных окружения. Добавление пользовательских алгоритмов балансировки возможно с помощью модификации исходного кода runtime системы.

Балансировщики в StarPU во время балансировки получают информацию о нагруженности системы, представленную в формате, указанном в листинге 1

```
1 /** interface for load_data */
2 struct load_data_interface
3 {
4         /** Starting time of the execution */
5         double start;
6         /** Elapsed time until the start time and the time when event "launch a
7          * load balancing phase" is triggered */
8         double elapsed_time;
9         /** Current submission phase, i.e how many balanced steps have already
10         * happened so far. */
11        int phase;
12        /** Number of currently submitted tasks */
13        int nsubmitted_tasks;
14        /** Number of currently finished tasks */
15        int nfinished_tasks;
16 };
```

Листинг 1 – формат информации нагруженности системы в StarPU

1.3 Фреймворки, использующие параллелизм на уровне потоков

В таких фреймворках как Cilk [5], Java fork join framework [6], .NET TPL [7] присутствует 1 алгоритм балансировки - work stealing [8] и нет возможности расширения. Связано это с тем, что параллелизм в этих фреймворках происходит на уровне потоков и они работают с общей памятью без персональных очередей задач. Поэтому потребность в перемещении задач между очередями с целью устранения дисбаланса отсутствуют.

1.4 Результаты исследования

Исходя из исследованных систем можно выделить общий подход к решению проблемы:

1. В системе существует сущность, которая управляет ходом балансировки;
2. системы имеют интерфейс балансировщика, реализация которого позволяет использовать пользовательские алгоритмы балансировки нагрузки.

В системах параллельного программирование автоматизированное применение алгоритмов динамической балансировки нагрузки производится с помощью метабалансировщика – сущности, которая отвечает за накопление и выбор алгоритма балансировки нагрузки. Требуется реализовать подобную сущность в runtime системе LuNA.

Для накопления алгоритмов в системе LuNA требуется разработать интерфейс, который должны будут реализовывать встраиваемые в систему алгоритмы. Этот интерфейс должен иметь достаточный для алгоритмов балансировки набор методов.

Так же требуется адаптировать под разрабатываемый интерфейс существующие в системе алгоритмы балансировки нагрузки.

Для того чтобы провести тестирование механизма автоматизации и интерфейса балансировщика требуется реализовать еще один метод балансировки в runtime системе LuNA, так как в системе на данный момент есть статическая балансировка и реализован лишь 1 алгоритм динамической балансировки.

2 Теоретическая часть

2.1 Технические особенности LuNA

2.1.1 Исполнение фрагментированной программы в системе LuNA

Фрагментированная программа в LuNA представляет собой набор фрагментов вычислений и фрагментов данных.

Фрагменты вычислений в LuNA разделяются на 2 типа:

1. Атомарные – принимают на вход фрагменты данных и вычисляют выходные фрагменты данных;
2. Структурированные – являются множеством атомарных и других структурированных фрагментов.

Вычисления в LuNA происходят посредством исполнения фрагментов вычислений с заданными фрагментами данных. Система выбирает фрагменты вычисления, готовые к исполнению. Если фрагмент является структурированным – результатом его выполнения являются его дочерние фрагменты вычисления. В случае если фрагмент вычисления является атомарным – результатом являются фрагменты данных.
В языке LuNA существует система рекомендаций – указаний того, как системе стоит обрабатывать фрагмент данных или вычислений. Одной из таких рекомендаций является локатор. Локатор позволяет определить алгоритм, с помощью которого должен балансироваться этот узел и идентификатор фрагмента. Этот идентификатор используется некоторыми алгоритмами балансировки.

В листинге 2 представлен пример локатора, перемещающего фрагменты вычислений на узел под номером i%size, где size – количество узлов в системе.

```
1 sub main() {
2         for i=1..100 {
3                 show("i=", i) @ { locator_cyclic: i; };
4         }
5 }
```

Листинг 2 – пример локатора для фрагмента вычислений

На листинге 3 представлен пример локатора, перемещающего фрагменты данных двумерного массива z.

```
 1 sub main()
 2 {
 3        df z;
 4        cf a: initialize(z[10], 5)@{locator_cyclic: 0;};
 5        cf b: display(z, z[10][6]) @ {
 6                request z[10][6];
 7                rope_cyclic: 0;
 8        };
 9 } @ {
10        locator_cyclic z[i][j]=>i;
11 }
```

Листинг 3 – пример локатора для фрагмента данных

В системе LuNA на данный момент присутствует 2 варианта балансировки: статический и методом work requesting [9]. Так же для системы разработан метод Rope of Beads [10], но он не реализован в текущей версии системы.

2.1.2 Компиляция

Компилятор переводит код с языка LuNA в код на языке C++, в котором каждый ФВ представлен в виде отдельной функции - блока.

Пример фрагмента скомпилированной программы представлен на листинге 4.

```
 1 // IF: preparations: if r548
 2 BlockRetStatus block_306(CF &self)
 3 {
 4        if (self.migrate(CyclicLocator(0))) {
 5                return MIGRATE;
 6        }
 7
 8        // request r548
 9        self.request(self.id(0), CyclicLocator(0));
10
11        self.NextBlock=307;
12        return CONTINUE;
13 }
```

Листинг 4 – пример фрагмента 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.

2.1.3 Система коммуникаций

В системе LuNA компиляции реализованы с помощью MPI [11]. Каждый объект, который можно отправить между узлами, имеет метод *serialize*, переводящий объект в байтовое представление, и *deserialize*, создающий объект из массива байт.

Так же при отправке указывается тип сообщения. Сообщения различного типа обрабатываются системой по-разному.

Сообщение можно отправить как на конкретный узел, так и всем узлам системы.

2.1.4 Вычисления

Вычисления в системе происходят с помощью выделенного потоков, который исполняет задачи из очереди. Размер пула можно конфигурировать. Если у фрагмента вычислений подготовлены не все фрагменты данных – задача возвращается в конец очереди. При использовании переменных между потоками синхронизации между ними происходят с помощью таких примитивов синхронизации, как условная переменная и мьютекс.

2.1.5 Хранение фрагментов данных

В системе LuNA хранятся следующие ассоциативные массивы, связанные с фрагментами данных:

1. Фрагменты данных;
2. функции обратного вызова от фрагментов вычислений, которые должны исполниться при получении фрагментов данных;
3. функции обратного вызова запросов от других узлов, которые должны исполниться при получении фрагментов данных.

Ключом элементов ассоциативного массива является идентификатор фрагмента вычисления.

2.1.6 Балансировка нагрузки

Балансировка нагрузки происходит с помощью локаторов – рекомендаций, указанных у блока фрагмента вычислений или фрагмента данных. В случае фрагмента вычислений при компиляции LuNA программы локатор превращается в дополнительный код. Этот код проверяет, нужно ли мигрировать фрагмент. Если требуется миграция – в фрагменте вычислений выставляется идентификатор узла, на который нужно мигрировать фрагмент и возвращается элемент перечисления *MIGRATE,* который инициирует отправку фрагмента. Пример кода представлен на листинге 4 на 4–6 строках.

Для фрагмента данных локатор превращается в дополнительный код, который передает объект, определяющий идентификатор узла на который следует отправить фрагмент. Но этот объект не сохраняется в ассоциативном массиве вместе с фрагментом данных, поэтому их динамическая балансировка невозможна. Пример кода с отправкой фрагмента данных представлен на листинге 4 на 9 строке.

2.1.7 Rope of Beads

В алгоритме Rope of Beads фрагментам сопоставляется координата на отрезке [0,1].

Отображение фрагментов на диапазон делается таким образом, чтобы соседние фрагменты данных отображались на одну и ту же координату или на близкие координаты в диапазоне. Отображение фиксируется перед началом исполнения фрагментированной программы и не меняется в ходе ее исполнения.

Отрезок [0,1] разбивается на сегменты по числу вычислительных узлов. Таким образом каждому фрагменту соответствует вычислительный узел.

Для каждого фрагмента вычислительный узел может вычислить координату и, если он принадлежит своему сегменту, получить доступ к требуемому фрагменту

или, в зависимости от значения координаты, перенаправить запрос на следующий или предыдущий вычислительный узел.

В зависимости от загруженности узлов границы сегментов могут изменяться.

2.1.8 Work requesting

Суть данного алгоритма заключается в том, что узел, у которого отсутствует нагрузка, отправляет запрос на получение задач соседним узлам. В системе LuNA можно варьировать количество соседей, которым узел отправит запрос.

Особенностью этого алгоритма является то, что в ответ на запрос узел получает фрагмент вычислений вместе с уже подготовленными фрагментами данных, тогда как в таком алгоритме как Rope of Beads узел получает фрагмент вычислений и самостоятельно запрашивает фрагменты данных. Это было сделано по причине того, что work requesting считается вспомогательным алгоритмом, который должен выравнивать точечный дисбаланс на вычислительных узлах.

2.2 Интерфейс для алгоритмов балансировки

Одним из недостатков системы LuNA является отсутствие единого механизма для добавления в систему новых алгоритмов балансировки. Для того чтобы решить проблему расширяемости системы LuNA алгоритмами балансировки требуется разработать интерфейс, который будет закрывать большую часть потребностей алгоритмов балансировки. По сути, алгоритмы балансировки должны стать плагинами для runtime системы, и у пользователя должна быть возможность их переключать.

Основные требования к интерфейсу:

1. Механизм коммуникации между узлами;
2. возможность определения нагрузки узла;
3. получение параметров для балансировки из параметров запуска.

В runtime системе LuNA фрагмент вычислений может принимать участие в балансировки в 2 состояниях: без входных фрагментов данных и уже с собранными

фрагментами данных. Важно дать возможность определять алгоритмам балансировки в каком состоянии находится фрагмент.

Еще одной особенностью системы является подход к балансировке фрагментов данных. При использовании динамических балансировщиков фрагменты данных требуется переносить между узлами в соответствии с динамикой балансировки. Поэтому для таких алгоритм балансировки требуется механизм массового переноса данных при изменении динамики.

2.3 Метабалансировщик

Метабалансировщик должен являться хранилищем и центром управления алгоритмов балансировки. Чтобы использование метабалансировщика не отличалось от использования других алгоритмов балансировки требуется, чтобы он реализовывал разрабатываемый интерфейс и мог использоваться как обычный алгоритм балансировки нагрузки.

Все вызовы, которые система будет осуществлять через разрабатываемый интерфейс метебалансировщик должен перенаправлять в активные алгоритмы балансировки нагрузки.

Метабалансировщик должен иметь доступ к параметрам, которые используют алгоритмы балансировки, а также возможность включать и выключать алгоритмы. Это требуется для того, чтобы в будущем расширить метабалансировщик модулем принятия решений, который будет управлять алгоритмами балансировки во время исполнения LuNA-программы.

2.4 Описание предлагаемого решения

Для реализации механизма автоматизированного применения алгоритмов динамической балансировки нагрузки для runtime системы LuNA предлагаются следующие шаги:

1. Разработка интерфейса для алгоритмов балансировки – это позволит решить проблему расширяемости системы LuNA алгоритмами балансировки;

2. адаптация существующего алгоритма балансировки work requesting под разработанный интерфейс –данный шаг нужен для того, чтобы протестировать удобство разработанного интерфейса;
3. реализация еще одного метода балансировки на разработанном интерфейсе – требуется для экспериментального исследования характеристик производительности программ на системы LuNA с применением различных алгоритмов динамической балансировки нагрузки. Для реализации был выбран алгоритм Rope of Beads;
4. реализация метабалансировщика – этот шаг направлен на автоматизирование применения алгоритмов. На текущем этапе он позволит выбирать алгоритм балансировки в момент запуска программы. Механизм выбора алгоритма балансировки должен быть расширяем, для того чтобы иметь возможность в дальнейшем встраивать алгоритмы, осуществляющие принятие решение об использовании алгоритмов балансировки и настройку их параметров.

2.5 Характеристика предлагаемого решения

Данное решение облегчит разработчикам накопление информации о методах балансировки нагрузки.

Также решение позволит разработчикам автоматизировать процесс выбора алгоритма балансировки нагрузки для конкретной задачи.

Расширяемость метабалансировщика позволит облегчит дальнейшие исследования автоматизации применения алгоритмов динамической балансировки нагрузки.

3 Реализация

3.1 Механизм передачи фрагментов данных

При исследовании функциональности runtime системы LuNA, связанной с балансировкой, выяснилось, что в системе отсутствует возможность динамической балансировки фрагментов данных. После исполнения фрагментов вычислений фрагменты данных отправляются на указанный узел, но при изменении отображения фрагмента данных на узел не происходит балансировка фрагмента в соответствии с новым отображением.

Для того чтобы добавить в систему возможность динамической балансировки фрагментов вычислений было сделано следующее:

1. Добавлено сохранение локаторов вместе с фрагментами данных;
2. реализован метод, которых проходит по всем сохраненным фрагментам данных и отправляет на новый узел фрагмент данных, если его отображение изменилось. Предполагается, что алгоритмы балансировки будут вызывать этот метод самостоятельно при надобности;
3. при получении фрагментов данных перед сохранением была добавлена проверка на то, является ли узел конечным. Это нужно для реализации алгоритмов балансировок, в которых отправка фрагментов происходит не на конкретный конечный узел, а через соседние узлы.

3.2 Разработка интерфейса для алгоритмов балансировки

Исходя из требований, описанных в разделе 2.1.9. был разработан интерфейс для runtime системы LuNA.

Методы разработанного интерфейса балансировщика можно разделить на 3 группы:

1. Абстрактный метод, реализация обязательна:
   1.1.int calculate_rank(bool with_data, int pos=0).
2. Абстрактный метод, реализация не обязательна:
   2.1.std::set<int> get_msg_acceptable_tags();

2.2. void accept_msg(int src, int tag, void *buf, size_t size);

2.3. void notify_pool_empty();

2.4. void notify_pool_submitted().

3. Метод реализован в абстрактном варианте балансировщика. Может использоваться внутри методов балансировщиков:

3.1. void send(int dest, int tag, const void *buf, size_t size, std::function<void()> finisher=nullptr);

3.2. int size();

3.3. int rank();

3.4. unsigned int get_queue_size();

3.5. void relocate_DFs().

Основным методом интерфейса балансировки является метод, рассчитывающий на какой узел следует отправить фрагмент: *int calculate_rank(bool with_data, int pos=0)*. Аргумент *with_data* сообщает о том, когда происходит балансировка – до или после получения всех входных фрагментов данных. *pos* является опциональным параметром. Для некоторых алгоритмов балансировки он служит для того, чтобы сопоставить порядковый номер с фрагментом. Например, для статической балансировки узел, на который должен отправиться, фрагмент определяется как *pos%size*, где *size* – это количество узлов в runtime системе.

Также интерфейс балансировки должен позволять обмениваться сообщениями между балансировщиками на разных узлах. Для этого в интерфейс было введено 5 следующих методов:

1. *std::set<int> get_msg_acceptable_tags()* – метод, возвращающий список идентификаторов сообщений, которые принимает балансировщик. Если узел runtime системы получается сообщение с данными идентификатором – он перенаправляет его балансировщик;

2. *void accept_msg(int src, int tag, void *buf, size_t size)* – метод, в который runtime система подаёт полученные сообщения, если *tag* этого сообщения входит в набор, возвращаемый в *get_msg_acceptable_tags()*.
3. void send(int dest, int tag, const void *buf, size_t size, std::function<void()> finisher=nullptr) – метод, с помощью которого балансировщик может отправлять сообщения на другие узлы.
4. *int size()* – получить количество узлов в runtime системе.
5. *int rank()* – получить идентификатор текущего узла.

Для некоторых алгоритмов балансировки требуется отслеживать состояние очереди задач, чтобы при отсутствии нагрузки на узле производить перебалансировку фрагментов между узлами. Было реализовано 3 метода, позволяющих следить за состоянием очереди задач:

1. *void notify_pool_empty()* – метод оповещает балансировщик когда задачи в очереди кончились.
2. *void notify_pool_submitted()* – метод оповещает балансировщик когда задача была добавлена в очередь.
3. *unsigned int get_queue_size()* – метод возвращает текущее количество задач в очереди.

Чтобы дать балансировщикам инструмент для мониторинга за нагруженностью системы в интерфейс были добавлены следующие методы:

1. *unsigned int get_exec_time_ms()* – метод возвращает сколько суммарно миллисекунд система исполняет фрагменты вычислений с начала работы программы;
2. *unsigned int get_finished_tasks_cnt()* – метод возвращает количество исполненных фрагментов вычислений
3. *unsigned int get_active_executors_cnt()* – метод сообщает, сколько потоков активны на момент вызова метода

Для динамических балансировок был добавлен метод *void relocate_DFs()*. Данный метод выполняет перебалансировку всех фрагментов данных, имеющихся на узле на требуемый узел. Этот метод требуется вызывать после изменения конфигурации динамического балансировщика, чтобы фрагменты находились на актуальном для них узле.

Для получения настроек балансировки используется конфигурационный класс, который хранит настройки на основе ключей запуска с префиксом “-B”.

В результате существующие в LuNA алгоритмы балансировки были адаптированы под общий интерфейс.

3.3 Адаптация Work requesting балансировщика

Для адаптации work requesting балансировщика к новому интерфейсу было сделано следующее:

1. В интерфейс были вынесены коммуникации между узлами с запросами на получение задач;
2. Методы нотификации интерфейса о том, что у узла закончились или добавились задачи в очередь были использованы в балансировщике для того, чтобы отправлять запросы на получение задач;
3. существующая реализация work requesting алгоритма выполняла балансировку фрагментов вычислений до получения фрагментов данных, а не после. Это было исправлено, так как work requesting позиционируется как вспомогательный алгоритм балансировки, решающий проблему точечного дисбаланса.

3.4 Реализация алгоритма Rope Of Beads

Реализацию алгоритма можно разбить на несколько этапов:

1. Отображение фрагментов на отрезок;
2. разбиение на подотрезки;
3. динамическое изменение длины подотрезков;

Отображение фрагментов на отрезок происходит в методе calculate_rank. Код метода представлен на листинге 5.

```
1 int RopeBalancer::calculate_rank(int pos) {
2         int rope_pos = pos % conf_->get_rope_length();
3         if (rope_pos < rope_borders_.first) {
4                 return balancer_-> prev_rank();
5         } else if (rope_pos > rope_borders_.second) {
6                 return balancer_-> next_rank();
7         } else {
8                 return balancer_->rank();
9         }
10 }
```

Листинг 5 – отображение фрагментов на отрезок в алгоритме Rope Of Beads

Разбиение на подотрезки происходит при инициализации балансировщика. Левая граница считается по формуле slice_size * rank, где slice_size равен rope_length() / size, balancer_rank – порядковый номер узла, size – количество узлов в системе. Правая граница считается по формуле slice_size * (rank + 1) - 1

Динамическое изменение длины подотрезков происходит в отдельном потоке раз в 100 мс. Изменение длины подотрезков происходит в несколько этапов:

1. Узел, имеющий нулевой идентификатор, отправляет следующему узлу сообщение с идентификатором *TAG_ROPE_REBALANCE*, содержащее текущие границы узла и информацию о нагрузке на данном узле – количество задач в очереди + количество активных потоков на момент отправки сообщения.
2. Узел, который получает сообщение *TAG_ROPE_REBALANCE*, на основе своей нагрузки и нагрузки соседнего узла определяет сдвигать ли левую границу на единицу или оставить текущую границу по следующей логике:
3. После определения левой границы узел отправляет назад сообщение с идентификатором *TAG_ROPE_REBALANCE_ANSWER* с новым значением границы. Так же, если у узла не нулевой идентификатор, он отправляет следующему узлу сообщение с идентификатором *TAG_ROPE_REBALANCE*.

4. Если значение границы изменилось – узел вызывает метод *relocate_DFs(),* который перераспределяет фрагменты данных в соответствии с новыми границами.

3.5 Реализация метабалансировщика

Метабалансировщик хранит в себе все активные балансировки в runtime системе. Метабалансировщик, как и другие алгоритмы балансировки реализует общий интерфейс. Методы, которые он реализует:

1. *int calculate_rank(bool with_data, int pos=0).* Перенаправляет вызов во включенный алгоритм балансировки;
2. *std::set<int> get_msg_acceptable_tags().* Возвращает набор тегов всех хранящихся балансировщиков;
3. void accept_msg(int src, int tag, void *buf, size_t size). Перенаправляет сообщение в требуемый балансировщик;
4. *void notify_pool_empty().* Нотифицирует балансировщики о том, что очередь задач пуста;
5. *void notify_pool_submitted().* Нотифицирует балансировщики о том, что в очередь задач была добавлена задача.

В LuNA программе алгоритм балансировки определяется с помощью рекомендаций *use_for_balance* – это алгоритм work requesting, *locator_cyclic* – это статическая балансировка и *rope_cyclic* — это алгоритм rope of beads. Эта рекомендация преобразуется в код по балансировке на этапе компиляции LuNA программы. Для того чтобы обойти это ограничение была добавлена рекомендация *abstract_cyclic*, которая компилируется в код, использующий метабалансировщик для выбора алгоритма балансировки фрагмента. Таким образом метабалансировщик решает потребность перекомпиляции LuNA программы при изменении алгоритма балансировки.

4 Тестирование

Цель тестирования - убедиться в потребности выбора алгоритма балансировки и настройки параметров для алгоритмов балансировки. Для тестирования были выбраны задачи разного типа и исследована зависимость скорости выполнения задач от типа балансировщика и его параметров. Результатом тестирования должно служить демонстрация превосходства различных алгоритмах при выполнении разных задач.

Тестирование производилось на кластере Информационно-вычислительного центра Новосибирского Государственного Университета [12]. Характеристика каждого узла: 12 ядер и 24 ГБ ОЗУ.

Для тестирования была взята задача Метод частиц в ячейках. Эта задача хорошо подходит для тестирования балансировки нагрузки из-за неравномерности нагрузки, требуемой для исполнения фрагментов.

В задаче вычислительная сетка разбивается по осям с помощью параметров NX, NY и NZ. Количество фрагментов в задаче, которые исполняются на каждом шаге, равняется NX*NY*NZ.

Тестирование проводится на следующих типах балансировки: Без балансировки – все задачи исполняются на одном узле;

1. Статическая балансировка – балансировка выполняется при написании программы с помощью разработчика. Разработчик задает фрагментам номер узла, на котором они должны исполняться;
2. Work-requesting – алгоритм балансировки, направленный на устранение точечного дисбаланса, служит вспомогательным алгоритмом к основному балансировщику;
3. Rope of beads – динамически перебалансирует фрагменты вычислений и данных, исходя из собственной нагрузки и нагрузки на соседний узлы, разработчик присваивает фрагментам вычислений и данных координату на

отрезке, на основе которой определяется на каком узле должен исполняться фрагмент.

4.1 Тест 1

Параметры задачи: NX = 8, NY = 8, NZ = 2, Eps = 0.01.

Таблица 1 демонстрирует результаты теста.

Таблица 1 – Результаты первого теста

Тип балансировки	Количество узлов	Параметры балансировки	Результат(сек)
Без балансировки	1		558.5
Work requesting	2	Количество соседей: 2	504.48
Work requesting	4	Количество соседей: 2	353.99
Work requesting	4	Количество соседей: 4	233.22
Статическая	2		471.41
Статическая	4		180.54
Статическая + work requesting	2	Количество соседей: 2	471.27
Статическая + work requesting	4	Количество соседей: 2	151.92
Статическая + work requesting	4	Количество соседей: 4	174.38
Rope of Beads	2	Длина веревки: 20	658.89
Rope of Beads	2	Длина веревки: 60	684.67
Rope of Beads	4	Длина веревки: 20	123.25
Rope of Beads	4	Длина веревки: 60	148.97
Rope of Beads + work requesting	2	Длина веревки: 20; Количество соседей: 2	1248.49
Rope of Beads + work requesting	2	Длина веревки: 60; Количество соседей: 2	1229.89
Rope of Beads + work requesting	4	Длина веревки: 20; Количество соседей: 2	133.38
Rope of Beads + work requesting	4	Длина веревки: 20; Количество соседей: 4	137.23
Rope of Beads + work requesting	4	Длина веревки: 60; Количество соседей: 2	152.76
Rope of Beads + work requesting	4	Длина веревки: 60; Количество соседей: 4	149.23

Как видно из результатов лучше всего в данном тесте себя показал алгоритм Rope of Beads с длиной веревки 20. Это связано с тем, что при статической балансировке нагрузка между узлами разделена довольно неравномерно и использование work requesting для решения точечного дисбаланса не полностью устраняеют дисбаланс. Так же можно выделить то, что использование work requesting ускоряет работу программы почти во всех случаях. Так как work requesting является вспомогательным балансировщиком, который используется для решения точечного дисбаланса, его результаты отдельно от других алгоритмов оказались хуже, чем у других балансировщиков.

4.2 Тест 2

Параметры задачи: NX = 8, NY = 8, NZ = 1, Eps = 0.001.

В данном тесте было уменьшено количество тяжелых фрагментов, но увеличена точность Eps.

Таблица 2 демонстрирует результаты теста.

Таблица 2 – Результаты второго теста

Тип балансировки	Количество узлов	Параметры балансировки	Результат(сек)
Без балансировки	1		128.88
Work requesting	2	Количество соседей: 2	129.02
Work requesting	4	Количество соседей: 2	97.94
Work requesting	4	Количество соседей: 4	93.54
Статическая	2		129.92
Статическая	4		88.47
Статическая + work requesting	2	Количество соседей: 2	130.19
Статическая + work requesting	4	Количество соседей: 2	89.32
Статическая + work requesting	4	Количество соседей: 4	117.53
Rope of Beads	2	Длина веревки: 20	130.59
Rope of Beads	2	Длина веревки: 60	137.63
Rope of Beads	4	Длина веревки: 20	91.57
Rope of Beads	4	Длина веревки: 60	93.40
Rope of Beads + work requesting	2	Длина веревки: 20; Количество соседей: 2	128.76
Rope of Beads + work requesting	2	Длина веревки: 60; Количество соседей: 2	130.23
Rope of Beads + work requesting	4	Длина веревки: 20; Количество соседей: 2	92.86
Rope of Beads + work requesting	4	Длина веревки: 20; Количество соседей: 4	95.27
Rope of Beads + work requesting	4	Длина веревки: 60; Количество соседей: 2	93.17
Rope of Beads + work requesting	4	Длина веревки: 60; Количество соседей: 4	97.73

В данном тесте лучше всего показала себя статическая балансировка. Причина заключается в том, что из-за меньшего количества тяжелых частиц исполнение программы происходит более сбалансировано и динамическая балансировка не требуется, так как она содержит излишние коммуникации и поэтому проигрывает статической.

4.3 Результаты тестирования

По результатам тестов можно увидеть, что для различных параметров задач лучше всего показывают себя разные алгоритмы балансировки.

Это означает, что существует потребность в широком арсенале алгоритмов балансировки и в механизме для автоматизирования подбора алгоритма балансировки и его параметров, что соответствует ожиданиям.

ЗАКЛЮЧЕНИЕ

В ходе работы был разработан интерфейс для алгоритмов балансировки, который позволяет расширять runtime систему специализированными алгоритмами.

На основе этого интерфейса был реализован алгоритм Rope Of Beads.

Так же под данный интерфейс был адаптирован алгоритм Work Requesting.

Для автоматизирования подбора алгоритмов и их параметров был разработан метабалансировщик, который унифицировал механизм управления алгоритмами балансировками нагрузки. В перспективе метабалансировщик может быть расширен модулем принятия решений.

Тестирование показало, что существует потребность в широком наборе алгоритмов балансировки и в механизме для автоматизирования подбора алгоритма балансировки и его параметров.

Данная работа была представлена на 60-й Международной научно-студенческой конференции, г. Новосибирск, 2022 г.

Защищаемые положения:

1. Разработан интерфейс для алгоритмов балансировки нагрузки в системе LuNA;
2. реализован алгоритм динамической балансировки Rope Of Beads на основе разработанного интерфейса;
3. алгоритм Work Requesting был адаптирован под разработанный интерфейс;
4. статическая балансировка нагрузки была адаптирована под разработанный интерфейс;
5. разработан метабалансировщик, управляющий имеющимися в LuNA балансировщиками нагрузки;

6. Проведено экспериментальное исследование производительности runtime системы LuNA с различными балансировщиками и параметрами балансировщиков.

Дальнейшим направлением развития по данной теме является разработка алгоритма выбора алгоритма балансировки и подбора параметров на основе данных о вычислительной нагрузки.

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

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

_____ _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Kale L. V., Krishnan S. Charm++ A portable concurrent object oriented system based on C++ //Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. – 1993. – С. 91-108.
2. Bastrakov S. et al. Particle-in-cell plasma simulation on heterogeneous cluster systems //Journal of Computational Science. – 2012. – Т. 3. – №. 6. – С. 474-479.
3. Малышкин В. Э. Технология фрагментированного программирования //Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. – №. 46 (305).
4. Augonnet C. et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures //Concurrency and Computation: Practice and Experience. – 2011. – Т. 23. – №. 2. – С. 187-198.
5. Blumofe R. D. et al. Cilk: An efficient multithreaded runtime system //Journal of parallel and distributed computing. – 1996. – Т. 37. – №. 1. – С. 55-69.
6. Lea D. A Java fork/join framework //Proceedings of the ACM 2000 conference on Java Grande. – 2000. – С. 36-43.
7. Leijen D., Schulte W., Burckhardt S. The design of a task parallel library //Acm Sigplan Notices. – 2009. – Т. 44. – №. 10. – С. 227-242.
8. Cederman D., Tsigas P. Dynamic load balancing using work-stealing //GPU Computing Gems Jade Edition. – Morgan Kaufmann, 2012. – С. 485-499.
9. Acar U. A., Chargu.raud A., Rainey M. Scheduling parallel programs by work stealing with private deques //ACM SIGPLAN Notices. – ACM, 2013. – Т. 48. – №. 8. – С. 219-228.
10. 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.

11. MPI Documents. Официальная документация стандарта MPI [Электронный ресурс] / – Режим доступа: https://www.mpi-forum.org/docs/ (дата обращения 17.05.2022).
12. Информационно-вычислительный центр Новосибирского государственного университета [Электронный ресурс] / – Режим доступа: http://nusc.nsu.ru/ (дата обращения 17.05.2022).