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
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Мустафина Дамира Эркиновича
(Фамилия, Имя, Отчество автора)

Тема работы:

РЕАЛИЗАЦИЯ ЦЕНТРАЛИЗОВАННОГО ПОДХОДА К ДИНАМИЧЕСКОЙ БАЛАНСИРОВКЕ ВЫЧИСЛИТЕЛЬНОЙ НАГРУЗКИ В СИСТЕМЕ ФРАГМЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ LUNA И ЕГО СРАВНЕНИЕ С ДЕЦЕНТРАЛИЗОВАННЫМ ПОДХОДОМ

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

Руководитель ВКР
к.т.н,
доц. каф. ПВ ФИТ НГУ
Власенко А. Ю./……
(ФИО) / (подпись)
«20» мая 2022 г.

Новосибирск, 2022

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
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ
Зав. кафедрой Малышкин В.Э.
(фамилия, И., О.)
…….
(подпись)
«21» января 2022 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Мустафину Дамиру Эркиновичу, группы 18201
(фамилия, имя, отчество, номер группы)

Тема работы: Реализация централизованного подхода к динамической балансировке вычислительной нагрузки в системе фрагментированного программирования LuNA и его сравнение с децентрализованным подходом.
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 20 янв 2022 № 0012

Срок сдачи студентом готовой работы 20 мая 2022 г.

Исходные данные (или цель работы):

Реализация модуля централизованной динамической балансировки вычислительной нагрузки в системе LuNA и экспериментальное сравнение с модулем, использующим распределенный подход.

Структурные части работы:

Обзор существующих средств оптимизации параллельных программ, постановка задачи, формулировка требований к разрабатываемому модулю, описание реализации, тестирование

Руководитель ВКР
канд. техн. наук,
доц. каф. ПВ ФИТ НГУ
Власенко А. Ю./……
(ФИО) / (подпись)
«20» января 2022 г.

Задание принял к исполнению
Мустафин Д. Э./……..
(ФИО студента) / (подпись)
«20» января 2022 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Равномерное распределение нагрузки при вычислениях на системах с распределенной памятью (мультикомпьютерах) — нетривиальная задача, эффективное решение которой требует специальной квалификации и больших трудозатрат. Для улучшения производительности параллельных программ нередко прибегают к использованию алгоритмов балансировки вычислительной нагрузки. Балансировка нагрузки [1] предполагает решение задачи максимально равномерного распределения вычислений между узлами.

В некоторых случаях время выполнения задач, из которых состоит программа, известно до их выполнения (например, если задачи не имеют сильную зависимость от входных данных). В таких программах задачи можно распределить по узлам **статически**, до начала вычислений, таким образом, чтобы нагрузка каждого из узлов была равномерной.

Однако для большинства программ невозможно адекватно оценить вычислительный вес задач до начала исполнения, а узлы вычислительной системы могут быть заняты другими вычислениями, не относящимися к задаче, что может значительно повлиять на эффективность балансировки. Более того, может измениться сама вычислительная система: например, вычислительный узел может выйти из строя. Для таких программ гораздо лучше проявит себя распределение задач **динамически**, во время исполнения программы. Однако недостатком динамических алгоритмов являются дополнительные накладные расходы, связанные с коммуникациями между узлами.

Существуют две основные стратегии динамической балансировки нагрузки: **централизованная** [2] и **распределенная** [3].

В централизованной динамической балансировке имеется выделенный узел, который собирает информацию с других узлов и в соответствии с ней осуществляет перераспределение нагрузки. В распределенном (децентрализованном) алгоритме узлы хранят информацию о некотором подмножестве узлов и в соответствии с ней осуществляют балансировку. Распределенный подход считается более масштабируемым, однако узлы не

имеют полной информации о нагруженности всей системы, в связи с чем достигнуть равномерного распределения нагрузки существенно сложнее.

Технология фрагментированного программирования [4] представляется перспективной и актуальной для высокопроизводительных вычислений. Одна из немногих разработок в этом направлении — система LuNA (Language for Numeric Algorithms), развиваемая на кафедре параллельных вычислений ФИТ НГУ.

Система LuNA [5] представляет собой систему параллельного программирования, ориентированную на решение больших задач численного моделирования и вычислительной математики.

Одним из главных преимуществ системы LuNA является автоматическое конструирование параллельной программы и наличие балансировки вычислительной нагрузки, благодаря чему программист может сосредоточиться лишь на реализации численных алгоритмов.

В системе LuNA до последнего времени имелся только децентрализованный динамический балансировщик нагрузки, однако для ряда задач централизованная балансировка может оказаться более эффективной.

Цель работы – реализация модуля централизованной динамической балансировки вычислительной нагрузки в системе LuNA и экспериментальное сравнение с модулем, использующим распределенный подход.

Для достижения этой цели были поставлены следующие задачи:

1) Обзор средств оптимизации параллельных программ.
2) Формулировка требований и проектирование модуля динамической балансировки нагрузки, использующего централизованный подход.
3) Реализация централизованного подхода к динамической балансировке нагрузки в системе LuNA.
4) Тестирование и экспериментальное сравнение двух подходов к динамической балансировке нагрузки.

Научная новизна состоит в том, что разработан алгоритм централизованной динамической балансировки нагрузки для фрагментированных программ.

Практическая ценность работы обусловлена тем, что использование модуля централизованной балансировки может в несколько раз сократить время работы LuNA-программ.

Работа изложена в четырех главах. Первая глава содержит обзор средств оптимизации параллельных программ. Вторая глава содержит описание основных компонентов языка и системы LuNA, а также требования к разрабатываемому модулю. Третья глава описывает реализацию алгоритма централизованной динамической балансировки нагрузки в системе LuNA. В четвертой главе приводятся результаты тестирования, а также сравнение централизованного и распределенного балансировщика нагрузки в системе LuNA.

Глава 1. Обзор средств оптимизации параллельных программ

Для того, чтобы реализовать централизованный подход к динамической балансировке в системе LuNA, необходимо ознакомиться с существующими средствами оптимизации параллельных программ, изучить их преимущества, а также использовать или адаптировать некоторые из подходов и концепций при реализации централизованного балансировщика в системе LuNA.

1.1 MPI Tuning

MPI Tuning [6] — это утилита, которая позволяет автоматически настраивать параметры библиотеки Intel MPI Library (например, алгоритмы коллективных операций) в соответствии с конфигурацией кластера [7] или приложения. Утилита итеративно запускает приложение для тестирования производительности с различными конфигурациями для измерения производительности и сохраняет результаты каждого запуска. На основе этих результатов утилита генерирует оптимальные значения для настраиваемых параметров.

1.2 Intel Trace Analyzer and Collector

Intel Trace Analyzer and Collector [8] профилирует и анализирует приложения MPI [9], помогая сосредоточить усилия на оптимизацию программ.

Данный инструмент позволяет:

- найти узкие места в коде, замедляющие работу всей программы;
- визуализировать поведение параллельной программы по профилю программы;
- оценить эффективность балансировки нагрузки;
- анализировать производительность подпрограмм и блоков кода;
- определить накладные расходы на коммуникации;
- сократить время обнаружения ошибок и увеличить производительность приложения.

Во время работы параллельной программы генерируется трассировочный файл, собирающий подробную информацию во время работы программы. После окончания генерации файла, его можно проанализировать с помощью графической утилиты traceanalyzer.

На рисунке 1 представлена одна из возможностей утилиты: таймлайн событий параллельной программы. Синим обозначены промежутки времени, затраченные на вычисления, красным — промежутки времени, затраченные на коммуникации между процессами.

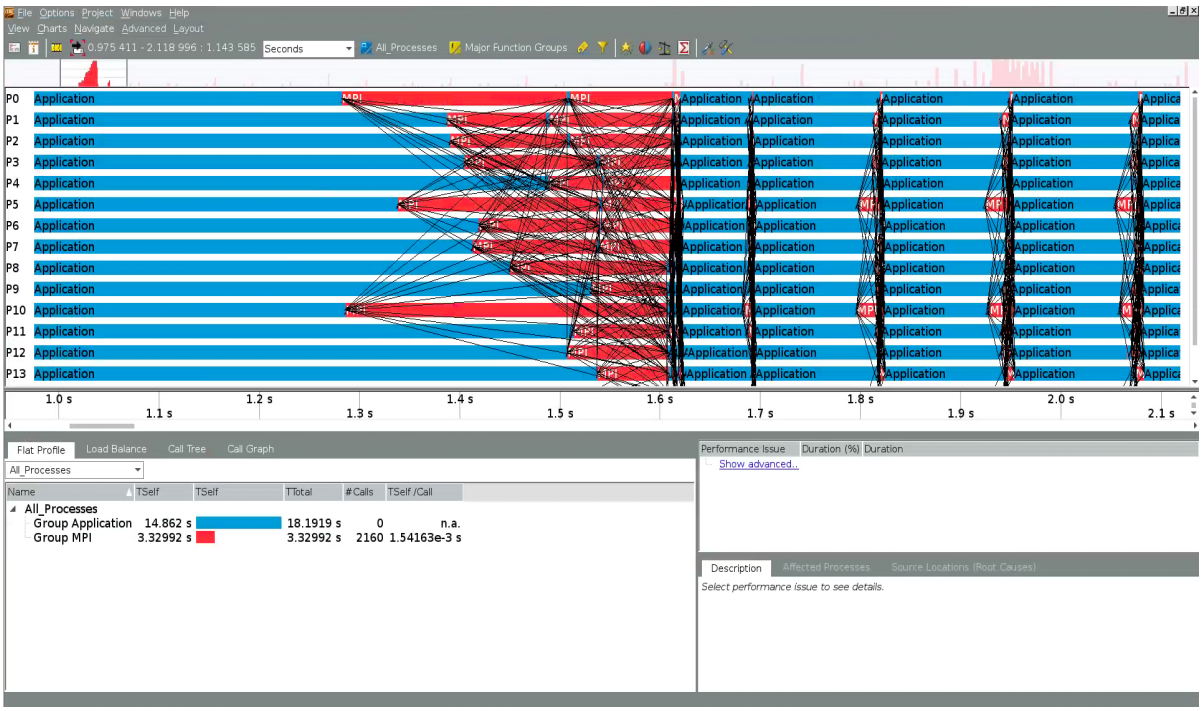

Рисунок 1 — Таймлайн событий

1.3 Charm++

Charm++ [10] — это система параллельного программирования, основанная на языке C++ [11]. В Charm++ используется модель миграции объектов, которая поддерживается мощной адаптивной runtime-системой. Charm-программа состоит из чар (chares), распределенных между процессами. Чары взаимодействуют между собой с помощью вызовов асинхронных методов или же сообщений. Runtime-система оперирует пулом сообщений, по очереди вызывая методы, указанные в сообщении. Балансировка нагрузки в Charm++ обеспечивается благодаря возможности размещать и перемещать чары.

Charm++ основан на парадигме параллельного программирования, управляемого сообщениями (message-driven). В отличие от многих других подходов, программисты указывают точки входа (вызовы методов) в свои параллельные чары, но не ожидают явным образом их выполнения. Таким образом, общий поток управления чаром может быть фрагментирован на несколько отдельных методов, скрывая явную последовательность их выполнения.

Для приложений, использующих итеративные алгоритмы, как правило, действует эвристический принцип, который называется «принципом постоянства» [12]: вычислительный вес задач и взаимодействие между чарами имеют тенденцию сохраняться. Это позволяет эффективно распределять чары между процессами.

1.4 DVM

DVM [13] — это система, предназначенная для создания переносимых и эффективных параллельных программ вычислительного характера на языках C-DVM (расширение языка C) и Fortran-DVM (расширение языка Fortran) для многопроцессорных компьютеров с общей и распределенной памятью. Аббревиатура DVM соответствует двум понятиям: Distributed Virtual Memory и Distributed Virtual Machine. В системе имеется компилятор, который переводит программу на языке C-DVM (Fortran-DVM) в программу на стандартном языке C (Фортран), расширенную функциями системы поддержки выполнения DVM-программ, которая для организации межпроцессорного взаимодействия использует стандартные коммуникационные библиотеки (MPI, PVM [14], Router). Моделью выполнения программы в системе DVM является модель SPMD [15]: одна программа — множество потоков данных. На все процессоры загружается одна и та же программа, но каждый процессор в соответствии с правилом собственных вычислений выполняет только те операторы присваивания, которые изменяют значения переменных, размещенных на данном процессоре (собственных переменных).

В системе имеются средства для анализа и отладки эффективности выполнения DVM-программ. Система во время выполнения программы накапливает информацию с временными характеристиками в оперативной памяти процессоров, а после завершения выполнения записывает информацию в файл, для анализа которого имеется визуализатор производительности.

1.5 Выводы по обзору

Таким образом, существует большое количество различных средств оптимизации параллельных программ. Каждое из средств имеет свои особенности, которые подходят для разных задач. Некоторые из особенностей можно учесть при реализации централизованного подхода к динамической балансировке нагрузки в системе LuNA. Особо интересными представляются сохранение результатов запуска программы для последующей генерации оптимальных значений параметров (MPI Tuning), а также использование эвристического «принципа постоянства» (Charm++).

Использование информации о предыдущих запусках LuNA-программы позволит значительно повысить эффективность работы централизованного балансировщика при последующих запусках программы. Балансировщик, зная вычислительный вес подзадач, сможет более равномерно перераспределить нагрузку, а также проигнорировать перераспределение легких подзадач на другие узлы, передача которых по сети лишь замедлит время работы программы из-за дополнительных накладных расходов.

Использование эвристического принципа постоянства в централизованном балансировщике позволит оценить вычислительный вес еще не выполненных задач, что позволит оценить текущую нагрузку на узле.

Глава 2. Разработка алгоритма централизованной динамической балансировки нагрузки

2.1 Балансировка нагрузки

Под **балансировкой нагрузки** понимается разделение вычислительной нагрузки между процессорами распределенной вычислительной системы с целью оптимизации использования ресурсов и сокращения времени работы программы.

Задачу балансировки нагрузки можно сформулировать следующим образом: дан набор задач и набор вычислительных устройств (процессорных ядер / процессоров / вычислительных узлов), на которых эти задачи могут быть выполнены. Необходимо распределить задачи таким образом, чтобы каждый из узлов имел примерно одинаковый объем работы.

Для больших задач численного моделирования и вычислительной математики сбалансированное распределение задач обычно повышает эффективность вычислений, то есть сокращает время работы программы — это является важнейшей целью балансировки нагрузки.

Балансировка нагрузки может осуществляться как *статически*, так и *динамически*. Статические алгоритмы балансировки задают распределение до начала выполнения основных вычислений программы. Динамические алгоритмы балансировки перераспределяют задачи между узлами во время исполнения программы.

Динамическая балансировка подходит для программ, в которых:

1) заранее неизвестно количество задач;
2) количество задач изменяется во время исполнения программы;
3) заранее неизвестен вес задач.

Решение проблемы динамической балансировки может быть разделено на несколько этапов [16]:

1) Оценка “*вычислительных весов*” задач, находящихся на каждом из узлов. Этот этап необходим для обнаружения ситуации дисбаланса. Зная

нагрузку узлов, можно определить, какие задачи следует передать другим узлам для достижения равномерного распределения.

2) Определение *накладных расходов* на передачу задач. Если время на перераспределение задач и дальнейшее их выполнение превышает время работы системы при отсутствии динамической балансировки, то в данный момент времени нет необходимости в балансировке нагрузки.
3) Определение, кто из узлов будет отправителем, а кто — получателем.
4) Для каждого узла-отправителя определение наиболее подходящих для передачи задач для обеспечения наилучшей балансировки нагрузки.
5) Распределение задач по узлам в соответствии с предыдущими этапами.

Декомпозиция процесса динамической балансировки на отдельные этапы позволит в дальнейшем сравнивать различные стратегии для каждого из этапов.

Методология балансировки нагрузки представлена на рисунке 2.

```
    балансировка_нагрузки(...)
    оценить нагрузку каждого задания и общую нагрузку каждого из
узлов
        if балансировка нагрузки выгодна then
        определить узлы, участвующие в балансировке
        выбрать наиболее подходящие для передачи задачи
        распределить задачи по узлам
    end if
end балансировка_нагрузки
```

Рисунок 2 — Абстрактный алгоритм балансировки нагрузки

2.1.1 Оценка нагрузки

Эффективность балансировки напрямую зависит от качества оценки нагрузки. Определить нагрузку можно аналитически, эмпирически или с помощью комбинации этих двух методов.

Аналитический метод основан на знании временной сложности алгоритма. Например, алгоритм сортировки массива из N = 64 элементов методом выбора [17] имеет вычислительный вес (N — 1) * (N / 2 * C + S), где C — вес операции сравнения, а S — вес операции обмена.

В эмпирическом методе нагрузка определяется через измерение времени работы, времени простоя и времени на коммуникации между узлами.

2.1.2 Инициация балансировки нагрузки

Перед началом балансировки необходимо определить ее выгодность. Для этого необходимо обнаружить ситуацию дисбаланса (например, сравнив нагрузку на узлах или сообщив другим узлам об отсутствии нагрузки на своем) и оценить накладные расходы на передачу задач.

2.1.3 Выбор задач

Качество выбора задач, подлежащих передаче, напрямую влияет на эффективность балансировки. Проблема выбора задач является NP-полной, так как она сводится к задаче о сумме подмножеств [18].

2.2 Технология фрагментированного программирования

Для достижения высокой производительности исполнения параллельных программ численного моделирования от программиста требуются знания специфики задачи, а также знания системного параллельного программирования.

Чтобы автоматизировать процесс реализации задач численного моделирования на суперкомпьютере, а также обеспечить динамические свойства параллельной программы, такие как динамическая балансировка загрузки, настройка на все доступные ресурсы, реализация коммуникаций на фоне вычислений, в лаборатории синтеза параллельных программ ИВМиМГ СО РАН разрабатывается система LuNA, поддерживающая технологию фрагментированного программирования (ТФП).

В основе ТФП лежит представление прикладного алгоритма в виде не более чем счетного множества вычислительных задач, называемых **фрагментами вычислений** (ФВ). Каждый ФВ состоит из конечного множества

входных и выходных объектов данных, называемых **фрагментами данных** (ФД) и вычисляет значения выходных ФД из входных ФД. Такое представление алгоритма называется **фрагментированным алгоритмом** (ФА). Отличительными особенностями ФА являются иммутабельность (единственное присваивание) ФД и отсутствие побочных эффектов у ФВ, что позволяет выполнять ФВ на любом вычислительном узле по выбору системы, в том числе динамически распределять ФВ по узлам **мультикомпьютера** и обеспечивать динамическую балансировку нагрузки.

Фрагмент вычислений считается **готовым к исполнению**, если он не зависит от фрагментов данных, либо все фрагменты данных, от которых он зависит, получили свои значения.

Исполнение программы заключается в исполнении всех его ФВ. Порядок исполнения ФВ определяется информационными зависимостями между ФВ. Программист не может знать, в каком порядке выполняются готовые ФВ.

В ТФП предусмотрена возможность влияния на ход исполнения программы путем определения "**рекомендаций**" — дополнительной информации об алгоритме и способе его отображения на ресурсы мультикомпьютера во времени.

Исполнение ТФП позволяет при разработке параллельных программ ограничиться разработкой последовательного кода.

2.3 Система фрагментированного программирования LuNA

LuNA (Language for Numerical Algorithms) — язык программирования и исполнительная система для параллельной реализации больших задач численного моделирования и вычислительной математики. Она может исполнять ФП параллельно в многопоточном режиме в случае вычислителя с общей памятью и в многопроцессорном и многопоточном режимах совместно в случае вычислителя с распределенной памятью.

Язык LuNA позволяет описывать ФА как множество ФВ и ФД, а также использовать рекомендации для контроля над нефункциональными свойствами программы [19], такими как:

1) миграция [20] ФВ и ФД на узлы мультикомпьютера;
2) порядок выполнения ФВ;
3) задействование ФВ при балансировке нагрузки;
4) сборка мусора [21].

В системе LuNA фрагменты вычислений делятся на *атомарные* и *структурированные* [22]. Атомарные фрагменты представляют собой подпрограмму (функцию) на языке C++. Структурированные фрагменты вычислений соответствуют управляющим конструкциям языка LuNA (циклы, условные операторы, вызовы подпрограмм). В частности, цикл for из N итераций, при его исполнении, порождает N новых фрагментов вычислений.

Схему исполнения ФВ можно представить следующим образом (рисунок 3):

Рисунок 3 — Схема исполнения ФВ

Пример программы на языке LuNA представлен на рисунке 4. Она печатает числа от 1 до 5, возможно в другом порядке из-за особенностей системы.

```
1   C++ sub set(name N, int val) ${{ N = val; $}}
2
3   C++ sub print_number(int n) ${{ std::printf("%d\n", n); $}}
4
5   sub print_all_numbers(name N) {
6       for i = 1..N {
7           print_number(i);
8       }
9   }
10
11  sub main() {
12      df N;
13      set(N, 5);
14      print_all_numbers(N);
15  }
```

Рисунок 4 — пример программы на языке LuNA

Здесь атомарными ФВ являются *set* и *print_number*, а структурированными — *main* и *print_all_numbers*.

В системе LuNA распараллеливание происходит путем распределения задач (фрагментов вычислений) как по узлам мультикомпьютера, так и по потокам в рамках каждого узла.

Система обеспечивает трансляцию LuNA-программ во внутреннее представление, которое затем исполняется исполнительной runtime-системой на мультикомпютере, а также выполняет операции по порождению и перемещению ФВ и ФД.

Исполнительная система реализована на языке C++ и состоит из трех модулей:

1) *Менеджер фрагментов вычислений.* Отвечает за исполнение ФВ, распределение их по узлам и определение порядка вычислений. Благодаря менеджеру фрагментов вычислений, программисту нет необходимости реализовывать собственный механизм балансировки нагрузки.

2) *Менеджер фрагментов данных.* Осуществляет распределение фрагментов данных по узлам, их поиск и доставку с других узлов по запросу, слежение за их готовностью и удаление фрагментов данных, не являющихся входными ни для какого из еще не выполненных ФВ.

3) *Коммуникационная подсистема.* Необходима для асинхронной передачи сообщений между другими подсистемами, расположенными на разных узлах. Примерами сообщений являются отправка фрагментов вычислений во время динамической балансировки нагрузки на узел, запросы на отправку и сама отправка фрагментов данных. В системе LuNA в данный момент коммуникации осуществляются с помощью MPI.

Благодаря модульной структуре, система LuNA удобна для расширения. Например, можно расширить возможности системы, реализовав централизованный модуль балансировки нагрузки или коммуникатор, не основанный на MPI.

2.3 Распределенная динамическая балансировка нагрузки в системе LuNA

Для перераспределения фрагментов вычислений по узлам, в системе LuNA имеется модуль распределенной динамической балансировки нагрузки [23].

Для его использования необходимо с помощью рекомендации *stealable* указать, какие ФВ следует задействовать в балансировке. На рисунке 5 представлен пример использования рекомендации.

```
1   C++ sub print_number(int n) ${{ std::printf("%d\n", n); $}}
2
3   sub main() {
4       for i = 1..5 {
5           print_number(i) @{
6               stealable;
7           };
8       }
9   }
```

Рисунок 5 — пример программы на языке LuNA с использованием рекомендации stealable

В этой программе имеется 5 ФВ *print_number*, которые порождены циклом for и могут быть по запросу отправлены на свободные от задач узлы.

Алгоритм распределенной динамической балансировки нагрузки в системе LuNA заключается в следующем:

1) узел, обнаружив, что у него не осталось ФВ в очереди задач, производит запрос новых ФВ на некоторое количество узлов;
2) при исполнении ФВ проверяется, помечен ли он специальной аннотацией и получал ли текущий узел запросы на передачу ФВ. При выполнении обоих условий этот ФВ передается узлу, выполнившему запрос;
3) при получении ФВ, узел отправляет отмену запроса.

Данный модуль динамической балансировки нагрузки способен сократить время работы LuNA-программ, однако он обладает рядом недостатков, которые значительно снижают эффективность балансировщика:

1) Балансировщик не полностью автоматический:
    a) необходимо явно расставлять рекомендации;
    b) необходимо понимать, в каких местах кода указание рекомендации поможет сократить время работы программы.
2) Балансировщик не имеет полной информации о загруженности всех узлов.
3) Фрагменты вычислений передаются по одному, даже если имеются сильно нагруженные узлы.
4) Передаваемые ФВ проверяются на готовность к исполнению, а значит, в некоторых случаях не могут сразу выполниться при наличии хотя бы одного невычисленного ФД.
5) Простаивающий узел после отправки запроса должен ожидать получения ФВ, и, возможно, готовности его исполнения, то есть по-прежнему простаивать.

2.4 Описание предлагаемого алгоритма централизованной динамической балансировки нагрузки

В качестве решения проблемы балансировки нагрузки и устранения недостатков децентрализованного подхода предлагается задать выделенный (*центральный*) узел, который:

1) Получает информацию от остальных узлов о готовности к выполнению фрагментов вычислений и об окончании их выполнения путем отправки сообщений.
2) Получает дополнительную информацию, позволяющую оценить веса ФВ, а также вычислить текущую нагрузку на каждом из узлов.
3) Проверяет необходимость в перераспределении нагрузки на основе полученной информации;
4) Составляет план балансировки, согласно которому фрагменты вычислений отправляются с одних узлов на другие.

В качестве модели взаимодействия узлов с центральным используется модель *ведущий-ведомый* [24], в которой центральный узел осуществляет однонаправленное управление рабочими узлами.

На рисунке 6 представлен пример взаимодействия узлов. Здесь все ФВ считаются одинаковыми по весу. Рабочий узел *У1* отправляет балансировщику сообщение о том, что его ФВ готов к исполнению. Рабочий узел *У2* сообщает о завершении ФВ.

Балансировщик обнаружил ситуацию дисбаланса между двумя рабочими процессами и составил план балансировки. Согласно плану, узел *У2* должен отправить 2 ФВ узлу *У1*.

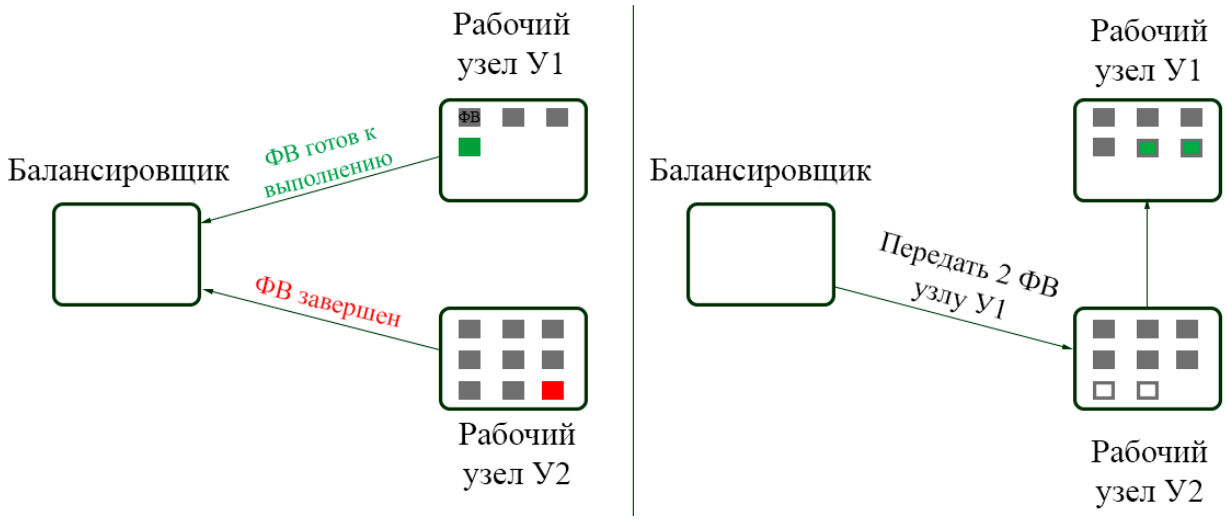

Рисунок 6 — Пример взаимодействия узлов

2.4.1 План балансировки

План балансировки — схема передач ФВ между узлами в процессе перераспределения нагрузки. При составлении плана балансировки следует учитывать следующие критерии:

1) максимальной равномерности распределения нагрузки между узлами после балансировки;
2) минимальных накладных расходов, связанных с передачей ФВ.

На рисунке 7 представлена математическая запись данных двух критериев. Критерий максимальной равномерности заключается в минимизации разности нагрузки между самым нагруженным и самым ненагруженным узлом. Критерий минимальных накладных расходов заключается в минимизации суммарного времени передачи фрагментов вычислений от одних узлов к другим.

фв_b^a – ФВ на узле a с весом ВЕС_b^a

T_b^a – время передачи фрагментов вычислений от узла a к узлу b в процессе данной балансировки

Узел 0: $\text{фв}_0^0,\ \text{фв}_1^0, \ldots, \text{фв}_{n_0}^0$;

Узел 1: $\text{фв}_0^1,\ \text{фв}_1^1, \ldots, \text{фв}_{n_1}^1$;

. . . . .

Узел k: $\text{фв}_0^k,\ \text{фв}_1^k, \ldots, \text{фв}_{n_k}^k$.

$$\begin{cases} \text{Критерий максимальной равномерности:} \\ \max_{p,q}\left(\left|\sum_{i=0}^{n_p}\text{ВЕС}_i^p - \sum_{j=0}^{n_q}\text{ВЕС}_j^q\right|\right) \to \min \\ \text{Критерий минимальных накладных расходов:} \\ \min \sum_a^k \sum_b^k T_b^a \to \min \end{cases}$$

Рисунок 7 — Математическая запись критериев

2.4.2 Оценка весов фрагментов вычислений

В качестве *веса* выполненного ФВ будет использоваться время его выполнения. Для того, чтобы оценить веса еще не выполненных ФВ, они будут *группироваться*. Невыполненные ФВ будут оцениваться средним арифметическим весом выполненных ФВ из одной группы. В одну группу попадут:

1) повторные вызовы одного и того же ФВ;
2) ФВ, порожденные в цикле.

2.4.3 Перераспределение нагрузки

Необходимость в перераспределении нагрузки предлагается определять с помощью двух параметров: *JOBS_LEFT_THRESHOLD* и *JOBS_DIFFERENCE_RATIO*. По умолчанию их значения равны 1 и 0.5 соответственно.

Первый параметр (*JOBS_LEFT_THRESHOLD*) определяет минимальную суммарную нагрузку узлов (в секундах), при которой необходимо осуществить балансировку нагрузки. LuNA-программы имеют разное количество фрагментов вычислений. В некоторых программах их количество составляет около тысячи, а в некоторых может запросто превосходить сотни тысяч. Параметр позволяет задавать момент, после которого балансировку нет смысла применять из-за малого количества оставшихся готовых к исполнению ФВ, так как часто от готовых ФВ зависит начало выполнения других зависящих от них ФВ.

Второй параметр (*JOBS_DIFFERENCE_RATIO*) определяет минимальную разность нагрузки в долях. Если разность нагрузки между двумя узлами, равная (H1-H2)/H1, превышает данный параметр, то будет совершено перераспределение нагрузки между ними. В противном случае балансировщик ожидает, пока не возникнет новая ситуация дисбаланса. Этот параметр необходим, так как в большинстве программ слишком частая балансировка способствует увеличению времени выполнения программы в связи с накладными расходами на коммуникацию между процессами. В каком-то

смысле параметр задает погрешность, при которой процессы считаются одинаково нагруженными.

Чтобы сократить объем хранимой и получаемой информации на узле, отвечающем за балансировку нагрузки, пригодится иметь возможность:

1) Сохранять информацию о времени выполнения ФВ для каждой из групп в файл. За это будет отвечать параметр *save_groups*.
2) Составлять список групп, которые могут участвовать в балансировке нагрузки. За это будет отвечать параметр *load_groups*. Составление будет происходить при запуске программы с параметром *load_groups* и наличии специального файла, генерируемого при запуске программы с параметром *save_groups*.

Балансировщик может работать без использования *save_groups* и *load_groups* программистом. Однако если балансировщик не демонстрирует необходимого ускорения, стоит иметь эти параметры в виду.

Таким образом, алгоритм, отвечающий за уравнивание нагрузки, можно описать следующим образом:

1) Если суммарная нагрузка на всех узлах меньше, чем *JOBS_LEFT_THRESHOLD*, то алгоритм заканчивает свою работу.
2) Составляется план балансировки.
3) Если разность нагрузки между каждой парой узлов из плана балансировки меньше, чем *JOBS_DIFFERENCE_RATIO*, то нет необходимости в перераспределении нагрузки, алгоритм заканчивает свою работу.
4) Отправляются сообщения о перераспределении нагрузки в соответствии с планом балансировки.

2.5 Преимущества централизованного подхода

Централизованный балансировщик имеет ряд преимуществ по сравнению с децентрализованным:

1) Балансировщик работает полностью автоматически, программисту нет необходимости расставлять рекомендации в коде.

2) Балансировщик имеет полную и актуальную информацию о загруженности всех узлов.
3) При перераспределении нагрузки:
    a) фрагменты вычислений передаются одним сообщением;
    b) учитываются только готовые к исполнению фрагменты вычислений.
4) Передаваемые ФВ могут быть выполнены без каких-либо ожиданий.
5) Балансировщик способен передать фрагменты вычислений на ненагруженный узел до того, как он начнет простаивать.

2.6 Требования, предъявляемые к алгоритму

1) Допускается расширение компиляции языка LuNA дополнительной генерацией кода, а также расширение языка LuNA необходимыми модулями.
2) Центральный узел должен отслеживать информацию о загруженности рабочих узлов.
3) Перераспределение нагрузки от нагруженного узла к ненагруженному должно происходить в соответствии с планом балансировки.
4) Центральный узел при обнаружении ситуации дисбаланса должен незамедлительно принять меры по ее устранению.
5) Не допускается аварийное завершение программы из-за ошибок в работе балансировщика нагрузки.

Глава 3. Реализация централизованного подхода динамической балансировки нагрузки в системе LuNA

3.1 Технические особенности системы LuNA

3.1.1 Выполнение фрагментов вычислений

За выполнение ФВ в системе LuNA в рамках одного узла отвечает пул потоков, размер которого можно изменить. Потоки выполняют ФВ из очереди задач.

3.1.2 Сообщения в системе LuNA

В системе LuNA взаимодействие между узлами происходит с помощью MPI, каждое сообщение помечается тегом. Это дает возможность определить тип сообщения и выполнить необходимые действия.

3.1.3 Трансляция кода из LuNA в C++

Перед компиляцией кода, система LuNA транслирует программный код с языка LuNA в код на языке C++, в котором ФВ представляет из себя набор функций, называемых блоками. На рисунке 8 представлен пример программы, печатающей несколько чисел.

```
import c_init(int, name) as init;
import c_iprint(int) as print;

sub print_all(int N) {
    for i = 1..N {
        print(i);
    }
}

sub main()
{
    df N;
    init(5, N) @ {
        locator_cyclic: 1;
    };

    print(N)@{
            stealable;
        };
    print_all(N);

}
```

Рисунок 8 — программа на языке LuNA

В программе один из ФВ инициирует ФД x значением 5, а другой печатает значение x.

Рекомендация “*locator_cyclic: number*” говорит о том, что ФВ следует выполнить на узле с номером number. Номера узлов начинаются с 0. Если *number* больше, чем количество узлов, то сначала считается остаток от деления number на количество узлов, а затем выбирается соответствующий узел.

Рекомендация *stealable* означает то, что ФВ может быть задействован децентрализованным балансировщиком.

Теперь рассмотрим, как выглядят фрагменты вычислений после трансляции.

На рисунке 9 показан код ФВ *init* после трансляции.

```
// BI_EXEC: cf _l15: init(5, N);
BlockRetStatus block_2(CF &self)
{
    if (self.migrate(CyclicLocator(1))) {
        return MIGRATE;
    }

    {
        DF _out_0;
        // EXEC_EXTERN cf _l15: init(5, N);
        c_init(
            // int 5
            5,
            // name N
            _out_0);

        {
            DF stored=_out_0;
            self.store(self.id(0), stored);
        }
    }

    // req_unlimited: N
    {       DF posted=self.wait(self.id(0));
    self.post(self.id(0), posted, CyclicLocator(0), -1);
    }
    return EXIT;
}
```

Рисунок 9 — Код фрагмента вычислений init после трансляции

ФВ имеет 1 входной ФД N. Сначала происходит проверка номера текущего узла, если он совпадает с 1, то выполнения ФВ продолжается, в противном случае ФВ передается (*мигрирует*) на другой узел.

Затем происходит инициация ФД N, который хранится в структуре самого ФВ, а также отправляется на узел с номером 0 с помощью метода *post*. С помощью рекомендаций можно задать другой получающий узел.

На рисунке 10 показан код ФВ print после трансляции.

```
// BI_EXEC: cf _l17: print(N);
BlockRetStatus block_3(CF &self)
{
    if (self.migrate(CyclicLocator(0))) {
        return MIGRATE;
    }

    if (self.check_steal()) {
        return STEAL;
    }

    // request N
    self.request(self.id(0), CyclicLocator(0));

    self.NextBlock=4;
    return CONTINUE;
}

// Request : cf _l17: print(N);
BlockRetStatus block_4(CF &self)
{
    // wait N
    if (self.wait(self.id(0)).is_unset()) {
        return WAIT;
    }


    self.NextBlock=5;
    return CONTINUE;
}

// After requests: cf _l17: print(N);
BlockRetStatus block_5(CF &self)
{
    {
        // EXEC_EXTERN cf _l17: print(N);
        c_iprint(
            // int N
            (self.wait(self.id(0))).get_int());

    }

    return EXIT;
}
```

Рисунок 10 — Код фрагмента вычислений print после трансляции

ФВ состоит из блоков 3, 4, 5 и имеет 1 входной ФД N. Стоит отметить, что print может участвовать в процессе балансировки нагрузки, такие ФВ не могут мигрировать, в отличие от ФВ *init*, поэтому первое условие в блоке 3 всегда ложно. Условие *check_steal* истинно, если имеются простаивающие узлы, запросившие ФВ, и в этом случае, ФВ отправляется первому из них. В противном случае происходит запрос ФД N (в данном случае запрос отправится узлу 0) и переход в к блоку 4. Если ФД инициирован, то начнет выполняться блок 5, в противном случае ФВ еще не готов к выполнению и кладется в конец очереди задач. Он будет снова выполнен, когда окажется в начале очереди.

В блоке 5 вызывается *c_iprint*, печатающая результат.

ФВ *print_all* состоит из блоков 6, 7 и 8. Последний блок (8) выполняется после инициализации N и порождает ФВ, начинающийся с блока 9 (рисунок 11). Порожденный ФВ кладется в очередь задач.

```
// After requests: cf _l20: print_all(N);
BlockRetStatus block_8(CF &self)
{
    { // EXEC_STRUCT: cf _l20: print_all(N);
        CF *child=self.fork(9);
        child->arg(0)=(self.wait(self.id(0))).get_int();
    }

    return EXIT;
}
```

Рисунок 11 — Код блока фрагмента вычислений print_all

Таким образом, фрагменты вычислений:

1) Могут состоять из одного или нескольких блоков.
2) Могут мигрировать или отправиться на свободный узел в ходе децентрализованной балансировки.
3) Отправляют вычисленные выходные ФД на другой узел, определяющийся “локатором” (по умолчанию узел с номером 0).
4) Запрашивают входные ФД у узлов.
5) Могут порождать другие ФВ.

6) Порожденные ФВ добавляются в очередь задач и выполняются пулом потоков.

Блок может возвращать следующие элементы перечисления:

1) MIGRATE, STEAL — необходимо передать ФВ на другой узел;
2) EXIT — ФВ завершен;
3) CONTINUE — необходимо начать выполнение ФВ со следующего блока.

3.2 Модификации в системе LuNA

Для того чтобы реализовать эффективную централизованную динамическую балансировку нагрузки в системе LuNA, необходимо добавить некоторые дополнительные возможности в систему.

3.2.1 Определение готовых к исполнению ФВ

В систему LuNA была добавлена возможность определять готовые к исполнению фрагменты вычислений. Для этого блок, отвечающий за ожидание данных (содержащий вызов метода *wait*), возвращает новый элемент перечисления *STEALABLE*. Пример блока 4 фрагмента вычислений print после трансляции представлен на рисунке 12. В случае, если блок возвращает *STEALABLE*, система устанавливает флаг того, что ФВ готов к исполнению и добавляет его в очередь задач для отложенного выполнения. Если в ходе балансировки нагрузки такой ФВ будет передан, то он может быть выполнен без ожидания входных ФД.

```
// Request : cf _l17: print(N);
BlockRetStatus block_4(CF &self)
{
    // wait N
    if (self.wait(self.id(0)).is_unset()) {
        return WAIT;
    }


    self.NextBlock=5;
    return STEALABLE;
}
```

Рисунок 12 — Код блока 4 фрагмента вычислений print

Если же фрагмент вычислений не имеет зависимостей от фрагментов данных, то он состоит из одного блока и об этом нельзя узнать до начала его выполнения. Поэтому в начале блока фрагмента вычислений, не имеющего зависимостей, дополнительно генерируется следующий код:

```
if(!self.isStealable()) {
   return STEALABLE;
}
```

Метод *isStealable* возвращает значение истина, если ФВ может быть передан на другой узел при перераспределении нагрузки (то есть ФВ готов к исполнению).

Как и в случае с ФВ, имеющем зависимости, при возвращении STEALABLE ФВ помечается готовым к исполнению. При следующем выполнении фрагмента вычислений условие **if**(!self.isStealable()) будет ложно, и фрагмент вычислений начнет исполняться.

3.2.2 Определение первого номера блока ФВ и вычисление веса ФВ

Теперь в системе LuNA хранится информация о номере первого блока, с которого начинается выполнение фрагмента вычислений. Это пригодится для группировки ФВ и оценки их весов. В структуру ФВ добавлено поле *FirstBlock*, которое в конструкторе инициируется значением поля *NextBlock*, которое совпадает с номером первого блока.

В структуру ФВ добавлено поле *weight*, в которую после завершения ФВ записывается его общее время выполнения. Это позволит оценивать

3.2.3 Извлечение готового ФВ из очереди задач

Для того чтобы иметь возможность передавать готовые ФВ, нужна возможность их извлечения из очереди задач. ФВ добавляются в очередь в виде функций, поэтому было решено модифицировать очередь так, чтобы в них, помимо функции, хранился указатель на ФВ. Это позволит итерироваться по очереди и извлекать из нее подходящий ФВ. У класса, отвечающего за пул потоков и выполняющего ФВ из очереди задач, был добавлен метод извлечения ФВ по номеру его первого блока (рисунок 13):

```
1   CF *ThreadPool::steal(int block) {
2       std::unique_lock<std::mutex> lk(m_);
3       for (auto it = jobs_.begin(); it != jobs_.end(); it++) {
4           CF *cf = (*it).first;
5           if (cf != nullptr && cf->isStealable() && cf->FirstBlock == block) {
6               jobs_.erase(it);
7               return cf;
8           }
9       }
10      return nullptr;
11  }
12
```

Рисунок 13 — Метод извлечения ФВ

3.2.4 Передача нескольких ФВ одним сообщением

В системе LuNA имеется код, отвечающий за передачу одного ФВ, однако при необходимости передать большое число ФВ это решение не является эффективным.

Для поддержки передачи нескольких ФВ был создан класс *CFArray*, который может сериализовать и десериализовать массив фрагментов вычислений. Также был добавлен специальный тег *TAG_MANY_CF* для передачи объекта этого класса.

Также добавлена возможность отключить *миграцию* фрагмента вычислений, в противном случае передача ФВ на другие узлы не имела бы смысла, так как миграция строго определяет номер узла, на котором ФВ может выполниться.

3.2.5 Оценка пропускной способности сети

Для того, чтобы в системе LuNA имелась возможность оценивать и учитывать время передачи ФВ в процессе балансировки, необходимо знать пропускную сети. При запуске программы происходит отправка нескольких сообщений разных размеров каждому из узлов для оценки *латентности* и *пропускной способности сети*. Эта информация хранится на центральном узле.

3.3 Выделенный узел для централизованной динамической балансировки

Для реализации модуля централизованной балансировки нагрузки, необходимо задать для выделенный (*центральный*) узел. Это можно сделать, уменьшив число узлов на 1 и задав центральным самый последний узел. Например, если всего узлов N, то считается, что узлы с номерами 0, 1, … N-2 являются рабочими, то есть могут выполнять ФВ, а узел с номером N-1 является центральным и вся важная информация о ФВ должна отправляться именно ему.

3.4 Получение и отправка сообщений центральным узлом

Выделенному узлу необходимо получать сообщения от рабочих узлов, для этого в систему LuNA были добавлены специальные теги: *TAG_NEW_JOB*, *TAG_COMPLETED_JOB*, *TAG_BALANCE* и *TAG_SENT_JOBS*.

Сообщения с тегом *TAG_NEW_JOB* и *TAG_COMPLETED_JOB* отправляются центральному узлу при появлении готового к исполнению ФВ и при завершении выполнения ФВ соответственно.

В сообщении с первым тегом дополнительно отправляется размер ФВ и номер первого блока фрагмента вычислений, а в сообщении со вторым тегом — вес ФВ.

Сообщения с тегом *TAG_BALANCE* отвечают за балансировку нагрузки и отправляются нагруженному узлу с указанием узла-получателя. Сообщение содержит сериализованный объект класса *RebalanceMessage*, в котором хранится узел-получатель (*dst*), узел-отправитель (*src*) и множество пар группа-количество (*group_counts*), означающее количество ФВ из каждой группы, которое необходимо передать узлу *dst*. Код класса представлен на рисунке 14.

```
1   class RebalanceMessage : Serializable {
2       int src;
3       int dst;
4       std::vector<std::pair<int, int>> group_counts;
5   public:
6       RebalanceMessage(int src, int dst,
7                        std::vector<std::pair<int, int>> &group_counts);
8
9       int get_src() const;
10
11      int get_dst() const;
12
13      std::vector<std::pair<int, int>> get_group_counts() const;
14
15      size_t get_serialization_size() const override;
16
17      size_t serialize(void *buf, size_t buf_size) const override;
18
19      size_t deserialize(const void *buf, size_t buf_size) override;
20  };
```

Рисунок 14 — Код класса RebalanceMessage

Узел-отправитель обязан сообщить выделенному узлу об успешной (или неуспешной) передаче указанных фрагментов вычислений, используя тег *TAG_SENT_JOBS*. В сообщении отправляется объект класса *RebalanceMessage*, с реально отправленным количеством переданных ФВ из каждой группы. Балансировка считается завершенной, когда все Узлы-отправители сообщили об окончании передачи фрагментов вычислений.

3.4 Модуль централизованной динамической балансировки нагрузки

Для реализации модуля централизованной динамической балансировки нагрузки был написан класс *Balancer* с методами *on_new_cf*, *on_complete_cf*, *rebalance* и *update*.

Метод *on_new_cf* вызывается при получении сообщения от рабочего узла о готовности к исполнению его ФВ. В сообщении имеется информация о номере узла, размере ФВ (в байтах) и номере первого блока фрагмента вычислений. *Balancer* для каждого узла по отдельности *группирует* информацию о фрагментах вычислений по номеру первого блока. В частности,

фрагменты вычислений, порожденные в цикле, попадут в одну группу. Информация хранится в объекте класса *BlockWeights*, содержащего:

1) веса всех выполненных ФВ в группе (поле *completed_weights*);
2) средний вес выполненных ФВ (поле *average_weight*);
3) количество невыполненных ФВ (*cf_count*);
4) размер всех ФВ (*sizes*).

Метод *on_complete_cf* вызывается при получении сообщения от рабочего узла о завершении ФВ. В сообщении также имеется вес ФВ — время его выполнения. В *BlockWeights* значение *cf_count* уменьшается на 1 и добавляется вес в массив *completed_weights*, после этого вызывается метод *rebalance*, отвечающий за перераспределение нагрузки между узлами. Если с момента прошлого вызова метода *rebalance* не все ФВ успели перераспределиться (поле *rebalance_count* класса *Balancer* не равно 0), метод ничего не делает, в противном случае:

1) оценивается нагрузка каждого из узлов, а также общая нагрузка;
2) проверяется необходимость в перераспределении нагрузки;
3) при необходимости, составляется план балансировки;
4) фрагменты вычислений перераспределяются в соответствии с планом балансировки.
5) в поле *rebalance_count* фиксируется количество перераспределений между узлами.

Метод *update* вызывается при получении сообщения с тегом *TAG_SENT_JOBS*. В *BlockWeights* обновляется информация, а значение *rebalance_count* уменьшается на 1.

3.4.1 Оценка весов фрагментов вычислений на основе времени их выполнения

Благодаря классу *BlockWeights*, имеется возможность оценить веса еще не выполненных фрагментов вычислений:

- веса еще не выполненных ФВ из группы корректируются, используя экстраполяцию средним арифметическим весов выполненных ФВ из той

же группы;

- если выполненных ФВ из группы нет, то веса устанавливаются равными средним арифметическим весов всех выполненных ФВ на узле.

Пример оценки весов фрагментов вычислений представлен на рисунке 15.

Рисунок 15 — Оценка весов ФВ

После выполнения ФВ, начинающегося с первого блока и имеющего вес 2, количество выполненных ФВ в группе 1 увеличивается на 1, а количество невыполненных уменьшается на 1. Теперь средний вес выполненных ФВ оказался равен 2.5. Соответственно, оценка весов невыполненных ФВ также равна 2.5.

В группе 3 отсутствуют выполненные ФВ, однако имеется 3 невыполненных ФВ. Посчитав средний арифметический вес всех выполненных ФВ, получаем оценку, равную 1.9.

3.4.2 Составление плана балансировки нагрузки

Балансировщик при необходимости в выравнивании нагрузки составляет план балансировки.

Текущий план балансировки составляется итерационно и учитывает только критерий максимальной равномерности: в приоритетном порядке перемещаются самые тяжелые ФВ, затем менее тяжелые. Нагрузка на процессе, получающем фрагменты вычислений, не должна превышать среднюю арифметическую нагрузку по всем процессам.

На рисунке 16 представлен пример плана балансировки.

Сперва *П1* должен отправить *П4* один ФВ из группы 1, один ФВ из группы 2 и два ФВ из группы 3. В результате, нагрузка перераспределится следующим образом:

- *П1*: 50 -> 36;
- *П4*: 10 -> 24.

На следующем этапе распределения нагрузки *П1* должен *П2* отправить один ФВ из группы 2. Итоговая нагрузка на каждом из процессов:

- *П1*: 32;
- *П2*: 24:
- *П3*: 20;
- *П4*: 24.

Процессы *П1* и *П3* по-прежнему недостаточно равномерно нагружены, но передав любой ФВ (каждый из них имеет вес 8), нагрузка на на *П3* превысит среднюю арифметическую нагрузку по всем процессам. Для исключения ситуации бесконечного перемещения фрагментов вычислений, алгоритм перераспределения нагрузки заканчивает свою работу.

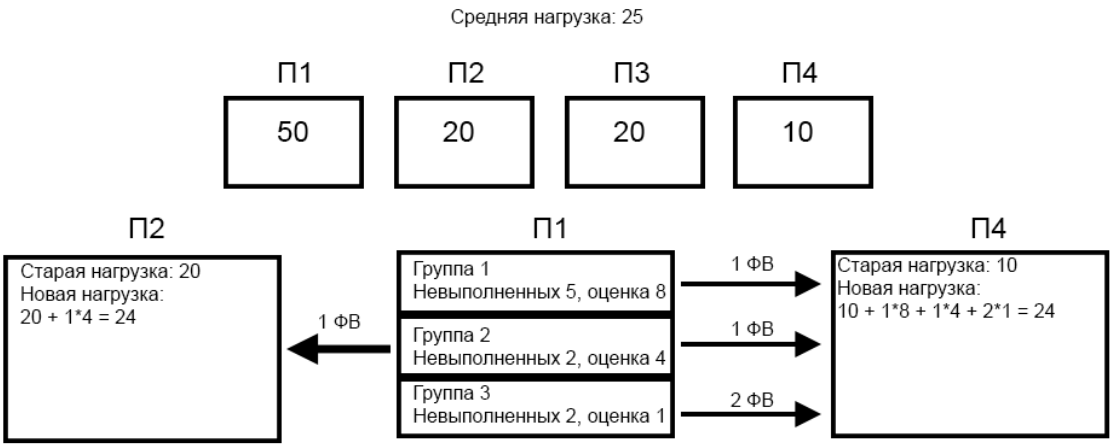

Рисунок 16 — План балансировки при распределении нагрузки

3.4.3 Параметры модуля

Балансировщик имеет 2 параметра, влияющие на эффективность балансировки и погрешность при перераспределении нагрузки: *JOBS_LEFT_THRESHOLD* и *JOBS_DIFFERENCE_RATIO*. Для них заданы значения по умолчанию 1 и 0.5 соответственно. Также имеются 2 параметра,

сокращающие объем хранимой и получаемой информации на центральном узле: *save_groups* и *load_groups*.

3.4.4 Проверка на необходимость в перераспределении нагрузки

За счет оценки латентности и пропускной способности сети, а также наличии информации о размере вычислительном весе ФВ, имеется возможность оценить время передачи и проверить наличие выигрыша при передаче фрагментов вычислений на другой узел.

Проверка на необходимость в перераспределении нагрузки происходит следующим образом:

1) Если значение поля *rebalance_count* не равно 0, то прошлый процесс балансировки все еще не закончен и нет необходимости совершать следующий;
2) Если общая нагрузка меньше, чем *JOBS_LEFT_THRESHOLD*, то в данный момент нет необходимости в перераспределении нагрузки.

После составления плана балансировки для каждой пары узлов, указанных в плане, выполняются следующие проверки:

1) Если разность нагрузки между двумя узлами меньше, чем *JOBS_DIFFERENCE_RATIO*, то считается, что ситуация дисбаланса не обнаружена.
2) Происходит подсчет времени передачи фрагментов вычислений от одного узла к другому. Если это время превышает суммарный вес фрагментов вычислений, то в дальнейшем передача таких ФВ не совершается.

Глава 4. Тестирование

Тестирование эффективности модуля централизованной динамической балансировки нагрузки проводилось на высокопроизводительном кластере НГУ (HP BL2x220c G7, 12 ядер, 24 ГБ ОЗУ на каждом узле). У каждого узла имеется только один поток, выполняющий фрагменты вычислений.

4.1 Тестирование на задаче блочного умножения матриц

Работа алгоритма централизованной балансировки нагрузки была протестирована на задаче блочного умножения матриц, реализованной в системе фрагментированного программирования LuNA. Код программы указан в Приложении А.

Стоит отметить, что в случае централизованного балансировщика 1 из узлов является выделенным для балансировщика, поэтому узлов, выполняющих вычисления, на 1 меньше, чем их общее количество.

Изначально, все фрагменты вычислений распределены на один вычислительный узел.

Результаты тестирования ускорения и эффективности двух балансировщиков представлены в таблице 1, а также на рисунках 17 и 18.

Таблица 1 — Сравнение эффективности двух балансировщиков на задаче блочного умножения матриц.

Количество узлов	Централизованная балансировка нагрузки					Распределенная балансировка нагрузки		
	Время, сек	Ускорение	Эффективность	JOBS_LEFT_THRESHOLD	JOBS_DIFFERENCE_RATIO	Время, сек	Ускорение	Эффективность
1	-	-	-	-	-	60.43	1.00	100.00
2	63.50	0.95	47.58	1.00	0.50	38.08	1.59	79.35
3	31.18	1.94	64.60	0.50	0.10	29.27	2.06	68.82
3	31.70	1.91	63.54	1.00	0.50	29.27	2.06	68.82
4	21.25	2.84	71.09	0.50	0.10	39.93	1.51	37.83

4	21.81	2.77	69.27	1.00	0.50	39.93	1.51	37.83
8	15.16	3.99	49.83	0.50	0.10	40.72	1.48	18.55
8	15.71	3.85	48.08	1.00	0.50	40.72	1.48	18.55
12	10.12	5.97	49.76	0.50	0.10	39.66	1.52	12.70
12	11.71	5.16	43.00	1.00	0.50	39.66	1.52	12.70
16	6.72	8.99	56.20	0.50	0.10	40.69	1.49	9.28
16	7.73	7.82	48.86	1.00	0.50	40.69	1.49	9.28

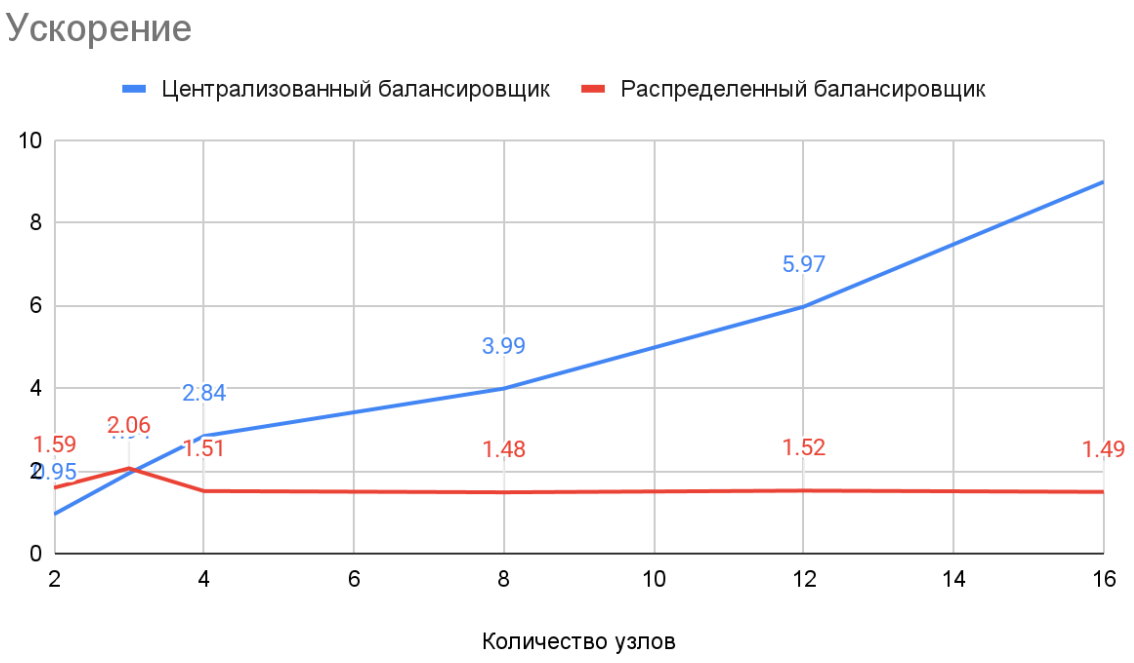

Рисунок 17 — Ускорение в зависимости от количества узлов

Как видно из таблицы, время работы программы при использовании централизованного балансировщика значительно уменьшается. Но при росте количества узлов растут и накладные расходы на коммуникации [25]. В дальнейшем планируется усовершенствовать алгоритм и максимально приблизиться к оптимальному алгоритму балансировки.

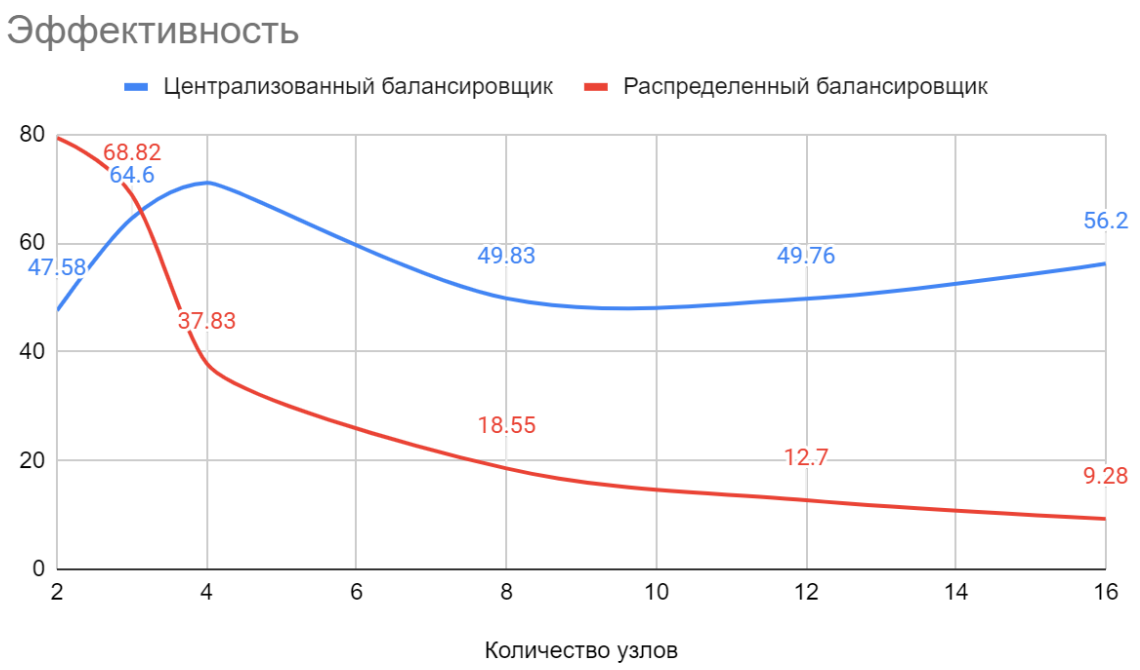

Рисунок 18 — Эффективность в зависимости от количества процессов

Малая эффективность при небольшом количестве узлов в централизованном балансировщике связана с тем, что один из процессов является выделенным (центральным), который и распределяет фрагменты вычислений между остальными процессами, а не выполняет их. При большем же количестве узлов растут накладные расходы на взаимодействие между узлами.

Таблица времени выполнения ФВ, в зависимости от группы, представлена ниже (таблица 2).

Таблица 2 — Информация о весах ФВ.

Группа	Вес	Количество	Суммарный вес
0	0.000028	1	0.000028
2	0.000023	1	0.000023
3	0.000002	1	0.000002
4	0.000064	1	0.000064
7	0.000047	10	0.000465
10	0.002642	100	0.26423

13	0.002653	100	0.265327
16	0.000006	100	0.000633
19	0.000007	100	0.000662
20	0.00002	100	0.001962
21	0.060854	1000	60.854218
24	0.000001	100	0.000098
25	0.000863	100	0.086303
28	0	100	0.000027
32	0.000008	100	0.000786
33	0.000731	800	0.584862
36	0.000005	100	0.000503

В данной программе централизованный балансировщик игнорирует все группы ФВ, кроме 21, 10 и 13 (к ним относятся ФВ, являющиеся вызовами функций *mult_mat*, *init_mat* для первой матрицы, *init_mat* для второй матрицы соответственно, см. приложение А).

4.2 Тестирование на задаче приведения матрицы к диагональному виду

Работа централизованного балансировщика нагрузки также была протестирована на задаче приведения матрицы к диагональному виду. Код программы приведен в приложении Б.

Особенность этой программы в том, что балансировщик не должен перераспределять нагрузку на другие узлы, в противном случае время работы программы может только увеличиться (см. таблицу 3):

Таблица 3 — Информация о весах ФВ.

Группа	Вес	Количество	Суммарный вес
0	0.00001	1	0.00001
1	0.000002	50	0.000109
2	0.000125	50	0.006256
3	0.000001	2500	0.002333

4	0.001189	2450	2.912519
7	0	2500	0.000577
11	0.000002	50	0.000085
12	0.00001	50	0.000494
16	0.000002	1	0.000002
17	0.000011	1	0.000011
18	0.000045	1	0.000045
21	0.005978	50	0.298899
24	0.000089	1	0.000089
27	0.000003	50	0.00015
30	0.000002	50	0.000096
31	0.000006	50	0.000323
45	0.000043	50	0.002157
46	0.000003	2500	0.008746
49	0.000003	50	0.00013
50	0.000078	50	0.003895
54	0	1225	0.000183

1) При передаче фрагментов вычислений, выполняющихся за миллисекунды (колонка *Weight*), время работы программы не уменьшится (как минимум, за счет латентности, имеющей схожий порядок).
2) При передаче более тяжелых ФВ (например, из группы 4 — вызов функций *row_reduction*, см. приложение Б), время работы программы увеличится из-за большого размера фрагментов данных (и, соответственно, долгой передачи ФВ по сети), от которых зависят ФВ.
3) Увеличение числа строк в матрице приведет к увеличению количества всех ФВ, а значит не повлияет на их вес.
4) Увеличение числа столбцов прямо пропорционально увеличит размеры тяжелых ФВ, их передача по сети по-прежнему будет слишком долгой.

Таким образом, данная программа в силу своих особенностей плохо

поддается балансировке.

Таблица 4 — Время работы программы в зависимости от количества узлов.

Количество процессов	Централизованная балансировка нагрузки			Распределенная балансировка нагрузки		
	Время, сек	Ускорение	Эффективность	Время, сек	Ускорение	Эффективность
1	-	-	-	6.19	1.00	100.00
2	6.32	0.98	48.97	6.59	0.94	46.97
3	6.41	0.97	32.19	8.01	0.77	25.76
4	6.36	0.97	24.33	8.05	0.77	19.22
8	6.69	0.93	11.57	6.23	0.99	12.42
12	6.73	0.92	7.66	6.68	0.93	7.72
16	6.76	0.92	5.72	6.85	0.90	5.65

Централизованный балансировщик, благодаря параметрам save_groups, load_groups и оценке времени на передачу ФВ сделал вывод, что выполнять балансировку нагрузку для данной программы нецелесообразно, поэтому все вычисления были выполнены на одном узле.

4.3 Результаты тестирования

По результатам тестирования можно увидеть, что централизованная балансировка нагрузки может значительно ускорить время работы LuNA-программ. Также удачный подбор параметров балансировщика может положительно повлиять на ускорение программ.

После профилирования обеих тестовых программ можно сделать вывод: использование динамического балансировщика нагрузки имеет смысл для программ, имеющих хотя бы одну группу фрагментов вычислений с большим объемом вычислений и относительно небольшим размером входных данных.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был проведен обзор средств оптимизации параллельных программ. Для системы LuNA был разработан и реализован модуль централизованной динамической балансировки нагрузки. Проведено экспериментальное исследование эффективности централизованного подхода к динамической балансировке нагрузки и сравнение с децентрализованным подходом. Модуль централизованной динамической балансировки нагрузки показал, что для программ, имеющих большие по вычислительному весу фрагменты вычислений, время их выполнения может значительно сократиться.

Также система LuNA была дополнена рядом новых возможностей:

1) определение готовых к исполнению ФВ;
2) определение блока, с которого начинается выполнение ФВ;
3) вычисление времени выполнения ФВ;
4) извлечение готового к исполнению ФВ из очереди задач;
5) передача нескольких ФВ одним сообщением;
6) оценка пропускной способности сети.

В дальнейшем планируется продолжить работу над модулем централизованной динамической балансировки нагрузки системы фрагментированного программирования LuNA. Возможными направлениями для развития являются:

1) усовершенствование составления плана балансировки для лучшего удовлетворения приведенных в работе критериев;
2) теоретическая оценка накладных расходов на балансировку нагрузки с целью оптимизации параметров, влияющих на общее время работы фрагментированной программы;
3) теоретическое сравнение времени исполнения программы с балансировкой централизованной, децентрализованной и без балансировки.

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».

Мустафин Дамир Эркинович _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1) Cybenko G., Dynamic Load Balancing for Distributed Memory Multiprocessors // J. Parallel and Distributed Comp. — 1989.
2) Khan S., Nazir B., Khan I.A., Shamshirband S., Chronopoulos A.T. Load balancing in grid computing: Taxonomy, trends and opportunities / Journal of Network and Computer Applications. — 2011. — Vol. 88.
3) Cao J., Bennett G., Zhang K. Direct execution simulation of load balancing algorithms with real workload distribution // The Journal of Systems and Software 54. — 2000.
4) Малышкин В. Э. Технология фрагментированного программирования // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. — № 46 (305).
5) Perepelkin V. A. LuNA system for automatic construction of numerical parallel programs for multicomputers // Проблемы информатики. 2020. № 1 (46).
6) Intel® MPI Library Developer Guide for Windows* OS [Электронный ресурс]. URL: https://www.intel.com/content/www/us/en/develop/documentation/mpi-developer-guide-windows/top/analysis-and-tuning/mpi-tuning.html (дата обращения 11.05.2022).
7) Foster, I., Zhao, Y., Raicu, I. & Lu, S. Cloud computing and grid computing 360-degree compared // IEEE Grid Computing Environment Workshops. — 2008.
8) Intel Trace Analyzer and Collector [Электронный ресурс]. URL: https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-itac/top.html (дата обращения: 11.05.2022).
9) MPI Documents. Официальная документация стандарта MPI [Электронный ресурс]. URL: https://www.mpi-forum.org/docs/ (дата обращения 11.05.2022).

10) Kale L. V., Krishnan S. Charm++: a portable concurrent object oriented system based on c++ // Proceedings of OOPSLA'93. — 1993. — Vol. 28.
11) Stroustrup B. C++ Programming Language // Pearson Education. — 2013.
12) The Charm++ Parallel Programming System. Официальная документация Charm++ [Электронный ресурс]. – URL: https://charm.readthedocs.io/en/latest/charm++/manual.html#load-balancing (дата обращения 11.05.2022).
13) DVM-система разработки параллельных программ [Электронный ресурс]. – URL: https://www.keldysh.ru/dvm/dvmhtm1107/rus/dvmINTRr.html (дата обращения 11.05.2022).
14) Коммуникационный интерфейс PVM [Электронный ресурс]. – URL: http://rsuib.cc.rsu.ru/tutor/high_performance_computing/chapter1/page12.html (дата обращения 11.05.2022).
15) Darema F. The SPMD model : Past, present and future // EuroPVM/MPI Conference. — 2001.
16) Willebeek-LeMair M., Reeves A. Strategies for Dynamic Load Balancing on Highly Parallel Computers // IEEE Transactions on Parallel and Distributed Systems. — 1993.
17) Levitin, A. Introduction to the Design and Analysis of Algorithms // VU: Pearson. — 2011.
18) Cormen T., Leiserson C., Rivest R., Introduction to Algorithms // MIT Press/MacGrawHill. — 1990.
19) Ахмед-Заки Д. Ж. и др. Автоматизация конструирования распределенных программ численного моделирования в системе LuNA на примере модельной задачи // Проблемы информатики. – 2019. – №. 4 (45).
20) Ahmad K., Maurya S. Load Balancing in Distributed System through Task Migration // International Journal of Engineering and Technology. — 2013.
21) Плешков А. В. Разработка алгоритмов распределенной сборки мусора в системе фрагментированного программирования LuNA //

Информационные технологии : Материалы 59-й Междунар. науч. студ.конф. 12–23 апреля 2021 г. / Новосиб. гос. ун-т. — Новосибирск : ИПЦ НГУ, 2021. — С. 118.

22) Описание языка LuNA [Электронный ресурс] . URL: https://gitlab.ssd.sscc.ru/luna/luna5/wikis/luna_lang_v01

23) 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. — С. 48.

24) Baldo L., Brenner L., Fernandes L., Fernandes P., Sales A. Performance Models ForMaster/Slave Parallel Programs // Electronic Notes in Theoretical Computer Science. — 2005.

25) El-Zoghdy S. F. A Load Balancing Policy for Heterogeneous Computational Grids // International Journal of Advanced Computer Science and Applicationsю -2011. — Vol. 2.

ПРИЛОЖЕНИЕ А

LuNA-программа блочного умножения матриц

```
import c_init(int, name) as init;
import c_init_submatrix(int `file, int `row, int `col, int `height, int `width, name `dest) as init_mat;
import c_mult_matrix(value `a, value `b, name `c) as mult_mat;
import c_sum_matrix(value `a, value `b, name `c) as sum_mat;
import c_save_submatrix(value `a, int `row, int `col) as save_mat;
import c_copy_matrix(value `a, name `b) as copy_mat;

#define FG_SIZE 360
#define FG_COUNT 10

sub calc_mat(name A, name B, name C, int i, int j, int N)
{
   df Ctmp, Csum;

   for k=0..N-1
   {
      cf f[i][j][k]: mult_mat(A[i][k], B[k][j], Ctmp[k]);
   }

   if N>1
      sum_mat(Ctmp[0], Ctmp[1], Csum[1]);
   if N==1
      copy_mat(Ctmp[0], Csum[0]);

   for k=2..N-1
      cf sum[k]: sum_mat(Ctmp[k], Csum[k-1], Csum[k]);

   copy_mat(Csum[N-1], C);
}

sub main()
```

```
{
   df A, B, C, N, M;

   init($FG_COUNT, N);
   init($FG_SIZE, M);

   for i=0..N-1
      for j=0..N-1
      {
         cf initA[i][j]: init_mat(0, i*M, j*M, M, M, A[i][j]);
         cf initB[i][j]: init_mat(1, i*M, j*M, M, M, B[i][j]);
         cf calc[i][j]: calc_mat(A, B, C[i][j], i, j, N);
      }
}
```

ПРИЛОЖЕНИЕ Б

LuNA-программа приведения матрицы к диагональному виду

```
import c_init(int, name) as init;
import c_print_row(value,value,name) as print_row;
import c_row_reduction(value,value,name,int) as row_reduction;
import c_check_row(value,int,name,int) as check_row;
import c_init_row(int `width, name `dest) as init_row;
import c_copy_matrix(value `a, name `b) as copy_mat;

#define MATRIX_ROWS 50
#define MATRIX_COLUMNS 300000

C++ sub empty() ${<END}
{
   // do nothing
}END

sub set_index(int i, name index, name Ai, int M)
{
   df is_null;
   for j=0..M-1
   {
      check_row(Ai[j], i, is_null[j], j);
   }

   init(0,is_null[M]);

   while(is_null[j]),j=0..out index
   {
      empty();
   }
}

sub all_rows_reduction(name A, int it, int index, name B, int M){
```

```
    for j=0..M-1
    {
        if j!=index && index<M {
            row_reduction(A[it][j],A[it][index], B[j], index);
        }

        if j!=index && index==M
        {
            copy_mat(A[it][j], B[j]);
        }
    }

    if index < M {
        copy_mat(A[it][index], B[index]);
    }
}

sub print_matrix(name A, int M){
    df mutex;
    init(1, mutex[0]);
    for k=0..M-1
    {
        print_row(A[k], mutex[k], mutex[k+1]);
    }
}

sub main()
{
    df A, B, C, N, M, ind;

    init($MATRIX_COLUMNS, N);
    init($MATRIX_ROWS, M);

    for j=0..M-1
    {
```

```
        init_row(N, A[0][j]);
    }

    for i=0..M-1
    {
        set_index(i, ind[i], A[i], M);
        all_rows_reduction(A, i, ind[i], A[i+1], M);
    }
}
```

ПРИЛОЖЕНИЕ В

Централизованный динамический балансировщик нагрузки для системы “LuNA”

Руководство оператора

Листов 6

Новосибирск 2022

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению централизованного динамического балансировщика нагрузки в системе LuNA.

В данном программном документе, в разделе «Назначение программы» указаны сведения о назначении программы и информация, достаточная для понимания функций программы и ее эксплуатации.

В разделе «Условия выполнения программы» указаны требования, необходимые для выполнения программы.

В разделе «Выполнение программы» указана последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы.

Оформление программного документа «Руководство оператора» произведено по требованиям ЕСПД: 19.101-77, 19.105-78, ГОСТ 19.505-79.

1 Назначение программы

1.1 Функциональное назначение программы

Разработанный централизованный динамический балансировщик нагрузки предназначен для работы с программами, которые запускаются на более чем одном вычислительном узле. Он используется для равномерного перераспределения нагрузки при возникновении дисбаланса в системе.

1.2 Эксплуатационное назначение программы

Модуль должен использоваться разработчиками фрагментированных программ для решения больших задач численного моделирования и вычислительной математики в системе LuNA.

1.3 Состав функций

Программа обеспечивает возможность выполнения перечисленных ниже функций:

1) отслеживание центральным узлом информации о загруженности остальных узлов;
2) отправка центральным узлом сообщения о перераспределении нагрузки между узлами;
3) передача фрагментов вычислений от нагруженного узла к ненагруженному в соответствии с полученным сообщением о перераспределении нагрузки.

2 Условия выполнения программы

2.1 Минимальный состав аппаратных средств

Программа предназначена для использования на персональном компьютере или вычислительном кластере. Минимальный перечень технических средств, обеспечивающих работу программы:

1) Оперативная память объемом не менее 4 Гб;
2) Жёсткий диск объёмом не менее 64 Гб;
3) Монитор;
4) Клавиатура.

2.2 Требование к персоналу

Конечный пользователь программы (оператор) должен обладать практическими навыками использования интерфейса командной строки, а также уметь запускать LuNA-программу на более чем одном MPI-процессе.

3 Выполнение программы

3.1 Загрузка и запуск программы

Запуск программы осуществляется при выполнении LuNA-программы с флагом “-c”.

Пользователь может указать следующие опциональные параметры:

1) --jobs_left_threshold <number> — определяет минимальную суммарную нагрузку узлов (в секундах), при которой необходимо осуществлять балансировку нагрузки. По умолчанию значение параметра равно 1;
2) --jobs_difference_ratio <number> — определяет минимальную разность нагрузки в долях, при которой считается, что возник дисбаланс между двумя узлами. По умолчанию значение параметра равно 0.5;
3) --save_groups — сохранить в файл информацию о времени выполнения ФВ для каждой из групп;
4) --load_groups — загрузить из файла, полученного после запуска программы с параметром save_groups, информацию о списках групп, которые могут участвовать в балансировке нагрузки.

3.2 Выполнение программы

После запуска LuNA-программы с использованием централизованного балансировщика центральный узел при обнаружении ситуации дисбаланса осуществляет отправку сообщений остальным узлам о необходимости перераспределить нагрузку.

3.3 Завершение работы программы

Балансировщик завершается автоматически после завершения выполнения LuNA-программы.