МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Нуштаева Юрия Юрьевича

Тема работы:

ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ КРАЕВОЙ ЗАДАЧИ ФИЛЬТРАЦИИ ДВУХФАЗНОЙ ЖИДКОСТИ В СИСТЕМЕ LUNA

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э /……..
(ФИО) / (подпись)
«31» …мая…… 2023г.

Руководитель ВКР
к.т.н., доцент
доцент каф. ПВ ФИТ НГУ
Маркова В.П./……...
(ФИО) / (подпись)
«20»…мая……2023г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Перепёлкин В.А./……..
(ФИ О) / (подпись)
«20»…мая……2023г.

Новосибирск, 2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий

Кафедра параллельных вычислений
(название кафедры)

Направление подготовки 09.03.01 Информатика и вычислительная техника

Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ

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

…….
(подпись)

«7» ноября 2022г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студент*у(ке)* Нуштаеву Юрию Юрьевичу, группы 19207
(фамилия, имя, отчество, номер группы)

Тема Эффективная реализация краевой задачи фильтрации двухфазной жидкости в системе LuNA
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 07.11.2022 №0330

Срок сдачи студентом готовой работы 20 мая 2023г.

Исходные данные (или цель работы):

анализ нефункциональных свойств программы, конструируемой системой LuNA, на примере задачи фильтрации

Структурные части работы: обзор, теоретическая часть, LuNA—реализация, тестирование

Руководитель ВКР
к.т.н., доцент
доцент каф. ПВ ФИТ НГУ
Маркова В.П. /……..
(ФИ О) / (подпись)

«7» ноября 2022г.

Задание принял к исполнению
Нуштаев Ю.Ю./……..
(ФИО студента) / (подпись)

«7» ноября 2022г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ,

Перепёлкин В.А. /……..
(ФИ О) / (подпись)

«7» ноября 2022г.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 4
ВВЕДЕНИЕ 5
1 Обзор 8
1.1 DVM (Distributed virtual memory) 8
1.2 Норма 9
1.3 CAF (Coarray Fortran) 9
1.4 Charm++ 10
1.5 KeLP (Kernel Lattice Parallelism) 11
1.6 T-система 11
1.7 Результаты анализа 12
2 Фрагментированная программа задачи фильтрации 14
2.1 Анализ фрагментированной программы 14
2.2 Построение фрагментированной программы задачи фильтрации 18
2.2.1 Описание фрагментированной программы 18
2.2.2 Описание поведения 20
2.3 Анализ системы 22
2.4 Выводы 23
3 Реализация фрагментированной программы 24
3.1 Реализация алгоритма в системе LuNA 25
3.2 Настройка исполнения, использование основных директив 27
3.3 Дополнительная настройка исполнения, использование дополнительных директив 30
3.4 MPI процедура 32
3.5 Итоги 35
4 Экспериментальное исследование 36
4.1 Тестирование LuNA-программы этапа 2 36
4.2 Тестирование LuNA-программы этапа 3 37
4.3 Тестирование LuNA-программы 38
4.3.1 Тестирование на кластере НГУ 39
4.3.2 Тестирование на кластере МВС 41
4.4 Итоги тестирования 44
ЗАКЛЮЧЕНИЕ 45
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ 46
ПРИЛОЖЕНИЕ А 49
ПРИЛОЖЕНИЕ Б 57

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Фрагмент данных (ФД) — переменные единственного присваивания;

Фрагмент вычислений (ФВ) — операции над фрагментами данных;

Локатор — модуль, вычисляющий локационную функцию;

Фрагментированный алгоритм (ФА) — множество фрагментов данных и вычислений (ФД и ФВ), связанных отношениями in и out;

Фрагментированная программа (ФП) — параллельная программа, реализующая ФА;

Директивы – обязательные предписания для выполнения системой (Системой LuNA).

ВВЕДЕНИЕ

В области системного параллельного программирования актуальной проблемой является автоматическое конструирование эффективной параллельной программы. Под эффективностью понимается время выполнения программы, расход памяти и т.п. Автоматическое конструирование эффективной программы является алгоритмически труднорешаемой задачей в общем случае, но решается в частных случаях. Вектором развития языков и систем автоматического конструирования параллельных программ является накопление частных решений и эвристик. Поэтому необходимо исследование и создание эффективных параллельных программ, реализующих конкретные задачи. Под реализацией задачи понимается разработка программы, решающей данную задачу. Опыт, накопленный при их создании, позволит создать новые методы и алгоритмы автоматического построения эффективных параллельных программ.

Одной из систем автоматического конструирования параллельных программ является система LuNA **[1]** — система автоматического конструирования параллельных программ, предназначенная для поддержки параллельной реализации больших численных моделей для суперкомпьютеров. Система основана на теории синтеза параллельных программ на вычислительных моделях. Данная работа посвящена эффективной реализации численного алгоритма решения краевой задачи фильтрации двухфазной жидкости в трёхмерной области при наличии скважин **[2,3]** в системе LuNA и сравнению с MPI реализацией этой же задачи.

Теоретическая значимость: частное решение данной задачи может случить основой для развития системных алгоритмов. Эвристики, полученные для данной задачи, могут помочь системе LuNA эффективно автоматически конструировать программы, реализующие данный класс задач.

Практическая значимость: разработанная параллельная эффективная LuNA программа может быть использована при моделировании на практике.

Научная новизна: Разработанная фрагментированная программа впервые позволила обеспечить эффективную реализацию алгоритма фильтрации на базе исполнительной системы LuNA.

Цель и задачи работы: разработать эффективную LuNA программу задачи фильтрации двухфазной жидкости в трёхмерной области при наличии скважин и проанализировать её нефункциональные свойства.

Достижение поставленной цели требует решения следующих задач:

1) разработать эффективную программу, реализующую краевую задачу фильтрации двухфазной жидкости в системе LuNA;

2) провести сравнение нефункциональных свойств программы LuNA и программы MPI, а именно оценить эффективность;

Объект исследования: реализация краевой задачи фильтрации двухфазной жидкости в системе LuNA.

Предмет исследования: нефункциональные свойства LuNA программы.

Методы исследования: методы разработки распределенных параллельных программ, технология фрагментированного программирования **[4]**, теория синтеза параллельных программ **[5]**, анализ результатов сравнения нефункциональных свойств программ в системе LuNA и MPI, анализ алгоритмов автоматизации конструирования параллельных программ в системе LuNA.

Личный вклад автора: Некоторые результаты были опубликованы **[6]**. Анализ краевой задачи фильтрации был проведён совместно с соавторами. Автору принадлежит раздел статьи “LuNA-реализация”.

Структура работы:

Первая глава посвящена обзору систем автоматического конструирования параллельных программ с точки зрения эффективности конструируемых программ.

Во второй главе представлены некоторые теоретические результаты, полученные при разработке LuNA программы.

Третья глава посвящена разработке программы в системе LuNA, её настройке исполнения и оптимизации.

В четвёртой главе представлено сравнительное тестирование промежуточных и итоговой версии программы LuNA и MPI реализации.

В заключении представлены выводы по итогам работы.

Приложение А является руководством программиста программы, решающей краевую задачу фильтрации двухфазной жидкости в системе LuNA

Приложение Б является описанием программы, решающей краевую задачу фильтрации двухфазной жидкости в системе LuNA

1 Обзор

Рассмотрим распространенные системы автоматического конструирования программ с позиций эффективности автоматически конструируемых программ. В первую очередь интерес представляет время работы программ, решающих различные классы задач на данных системах.

1.1 DVM (Distributed virtual memory)

DVM-система **[7,8]**, созданная в ИПМ им. М.В. Келдыша РАН при активном участии аспирантов и студентов факультета ВМК МГУ им. М.В. Ломоносова, предназначена для разработки параллельных программ научно-технических расчетов на языках C-DVMH и Fortran-DVMH, являющихся расширениями языка C и Fortran соответственно. Система использует DVMH-модель, которая подразумевает несколько уровней параллелизма: параллельное выполнение вычислений, параллельное выполнение распределённых циклов, параллельное выполнение циклов в общей памяти. Система поддерживает распараллеливание итераций цикла, распределение элементов массива между процессорами, выполнение операций редукции. В состав системы входят:

1) Компилятор Fortran-DVMH;

2) компилятор C-DVMH;

3) библиотека LIB-DVMH;

4) средства функциональной отладки DVMH-программ;

5) средства отладки эффективности DVMH-программ.

В своей основе система использует OpenMP, MPI и CUDA.

Цели, которые ставят разработчики системы — простота разработки параллельных программ и высокая эффективность выполнения программы на ЭВМ различной архитектуры

На официальном сайте DVM **[9]** представлены тесты для разных задач, таких как параллельная сортировка, система уравнений Навье-Стокс, метод переменных направлений, быстрое преобразование Фурье, спектральный метод в трёхмерной области и т.д. Сравнение происходит с последовательной программой, параллельной MPI и DVM. По производительности DVM

программы в ограниченном множестве задач сопоставимы с MPI, но есть ряд задач, в котором DVM отстаёт от MPI на 40-50%. В целом система хорошо показывает себя во многих классах задач, демонстрируя удовлетворительную производительность.

1.2 Норма

Язык программирования Норма **[10]** — декларативный язык для описания программ, предназначениях для выполнения на вычислительных системах. Язык НОРМА был разработан в ИПМ им. М.В. Келдыша РАН, создана система параллельного программирования НОРМА; язык применялся для решения задач математической физики. В программе на языке НОРМА не требуется информация о порядке счета, способах организации вычислительных (параллельных) процессов. Порядок предложений языка может быть произвольным, а информационные взаимосвязи выявляются компилятором и учитываются им при организации вычислительного процесса. Такой подход позволяет трансляторам осуществлять эффективное распараллеливание вычислений, а также избавляет язык от побочных эффектов.

На языке Норма можно реализовать множество различных математических задач. К сожалению, прямого сравнения производительности Норма и других систем автоматического конструирования параллельных программ или MPI нет, но в литературе **[11]** для задач на вложенных сетках можно найти графики ускорения Норма программы и линейного ускорения. По графику можно заключить, что в зависимости от параметров задачи ускорение Норма программы сопоставимо с линейным до 64-128 процессоров.

Можно сделать вывод, что система Норма удовлетворительно решает многие сеточные задачи.

1.3 CAF (Coarray Fortran)

Язык Coarray Fortran пришёл на смену HPF **[12,13]**, ранее известный как **F--**. Язык F-- начинался как расширение Fortran 95/2003, включён в стандарт Fortran 2008 **[14]**, в литературе представлено полное описание языка **[15].** Особенностью CAF является то, что приложение представляется как множество копий

программы, все копии программы выполняются асинхронно. Каждая копия программы имеет свой собственный набор данных. В терминологии CAF такой набор данных называется изображением.

В статье **[16]** сравнивается Coarray Fortran и MPI с точки зрения производительности, гибкости и простоты программирования. В качестве задачи взят численный алгоритм метода Больцмана (LBM). Используются разные подходы к построению CAF программы, сравнивается коммуникация CAF и MPI. В результате тестов наилучшее время выполнения показывает MPI, также её преимущество в лучшем масштабировании. Если сравнить CAF с другими системами в обзоре, например, с DVM, то с точки зрения производительности CAF показывает себя лучше. Это связано, с тем, что CAF предоставляет более низкоуровневые методы настройки исполнения и не имеет среды выполнения.

1.4 Charm++

Система Charm++ — это платформа параллельного программирования на C ++, поддерживаемая адаптивной системой выполнения **[17]**.

Программы на языке Charm++ разбиваются на объекты называемые “чары”, взаимодействующие посредством сообщений. Система распределяет “чары” по узлам сети. Используется объектно-ориентированный дизайн программы. Есть глобально адресуемые объекты и глобально вызываемые методы. Более подробно о Charm++ можно найти в **[18]**.

В состав Charm++ входят:

1. язык (расширение C++);
2. множество подключаемых библиотек, реализующих различные функции;
3. среда выполнения.

С помощью Charm++ было реализовано множество проектов, описание которых можно найти в **[19]**. Самая известная из них — NAMD — программа для молекулярной динамики. Её особенность заключается в оптимизации всех возможностей Charm++ под эту задачу — итогом стало одно из самых больших на сегодняшний день моделирование для данной задачи — более 100 000 000

атомов на 300 000 ядрах. Используется и балансировка нагрузки на основе измерений для масштабирования до тысяч процессоров.

Использование Charm++ в реальных проектах показывает, что удовлетворительная производительность достигается в некоторых задачах. Например, в статье **[20]** сравнивается MPI и Charm++ в задаче построения минимального остовного дерева в графе. В результате оптимизированная реализация на Charm++ показывает практически сопоставимую производительность с MPI реализацией, отставание несущественно. Можно заключить, что Charm++ является неплохой системой автоматического конструирования параллельных программ с точки зрения производительности, но для многих задач для достижения удовлетворительной производительности требуется настройка исполнения.

1.5 KeLP (Kernel Lattice Parallelism)

Система KeLP является платформой для реализации научных приложений на параллельных системах с распределенной памятью. Полное описание самой системы можно найти в **[21,22]**. В системе есть поддержка динамической балансировки нагрузки, автоматическое распределение данных по узлам на основе графа зависимостей. Есть возможность подключать модули, написанные на других языках программирования. Основные области задач, на которых специализируется данная система — моделирование земных недр, исследования турбулентности и другие сеточные задачи.

Статья **[23]** посвящена Data Mover — абстракции коммуникационной подсистемы, позволяющей добиться хорошей производительности на KeLP. Приводится множество результатов сравнения на различных задачах — тестах, таких как Redblack3D, NAS-MG, NAS-FT и т.д. Тесты показали, что отставание KeLP от MPI может достигать 30%, что является неплохим результатом. Таких результатов удалось достичь благодаря тонкой настройке исполнения.

1.6 T-система

Система основана на собственной модели организации вычислительного процесса **[24, 25]**. Основные принципы:

1. основная парадигма — функциональное программирование;

2. каждый вызов функции сопровождается созданием процесса с количеством входов, соответствующих количеству аргументов функции, и количеством выходов, соответствующих количеству результатов функции;

3. все процессы вычисления функций порождаются в состоянии “готов к вычислению” (dataflow);

4. процесс теряет “готовность” в случае попытки выполнить операцию с неготовым значением;

5. вычислитель представляется в виде сети, узлами которой являются запущенные процессы и структуры данных, а дугами — отношения поставщик—потребитель.

Более подробно о системе и принципах системы в **[26]**.

Система сама организовывает синхронизацию, распределение по узлам мультикомпьютера.

Ввиду того, что автоматическое распараллеливание программ происходит на базе графа функций, удовлетворительные нефункциональные свойства параллельных программ обеспечиваются только на ограниченном классе задач, так как отображение графа функций на ресурсы мультикомпьютера во времени в общем случае является труднорешаемой задачей.

1.7 Результаты анализа

Различные системы автоматического конструирования параллельных распределённых программ имеют свои особенности, эти особенности по-разному могут влиять на производительность системы для различных классов задач. Некоторые системы, такие как CAF, имеют более низкоуровневые методы настройки исполнения программы, что даёт значительный выигрыш в производительности. Другие системы, такие как Charm++, могут быть лучше оптимизированы для конкретных классов задач (проект NAMD).

Анализ средств параллельного программирования показал, что нефункциональные свойства программ достигаются только на определённом множестве задач, это множество задач может отличаться от системы к системе.

Можно сделать вывод, что развитие систем автоматического конструирования параллельных — актуальная задача. Важно продолжать исследование в этом направлении, развивать частные подходы к решению различных классов задач.

2 Фрагментированная программа задачи фильтрации

Для разработки фрагментированной программы задачи фильтрации необходимо провести анализ фрагментированной задачи фильтрации, анализ представлен в 2.1, а сама фрагментированная программа в 2.2.

2.1 Анализ фрагментированной программы

Рассмотрим краевую задачу фильтрации с точки зрения фрагментированного программирования и проведём анализ, результаты данного анализа представлены в статье **[6]**. Анализ необходим для разработки LuNA программы, реализующей данный алгоритм.

Рассмотрим схему метода сопряжённых градиентов (Рисунок 1). Данная схема отражает зависимости между данными и операциями. Каждая операция обладает свойствами. Анализируя свойства операций, можно создать фрагментированную программу.

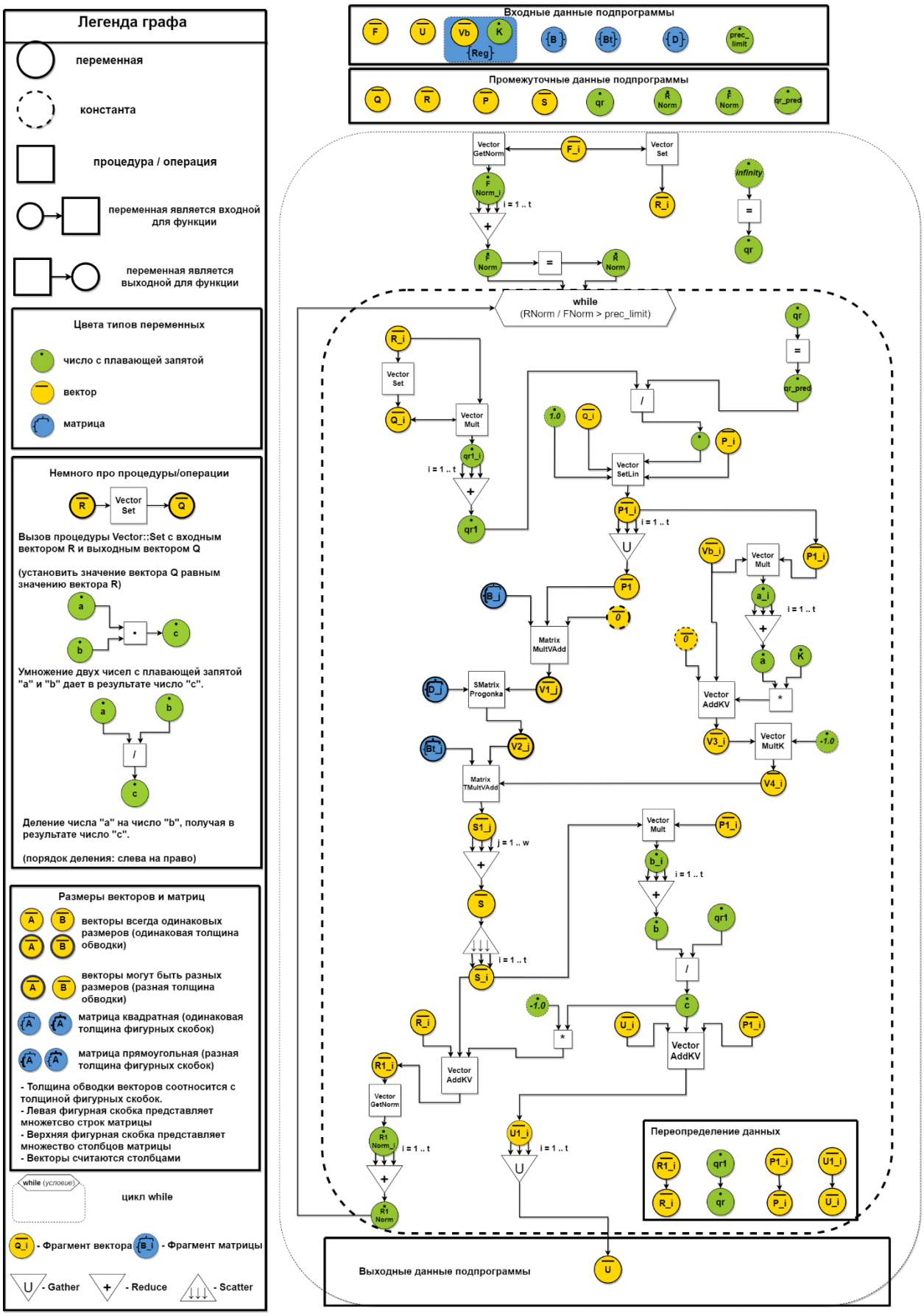

Рисунок 1 — Схема метода сопряжённых градиентов

Операции можно разделить на два типа: векторные и матричные.

Векторные операции — операции, в которых входными данными являются только векторы.

Матричные операции — операции, у которых во входных данных есть матрицы.

Выделим все векторные операции:

1.VectorAddKV — умножение вектора на коэффициент и добавление его к другому вектору;

2. VectorMultK — умножение вектора на коэффициент;

3. VectorSet — копирование одного вектора в другой;

4. VectorSetLin — установление вектора как линейной комбинации двух других векторов;

5. VectorMult — скалярное произведение векторов;

6. VectorGetNorm — взятие евклидовой нормы вектора.

Первые 4 операции: (1) VectorAddKV, (2) VectorMultK, (3) VectorSet и (4) VectorSetLin — обладают свойством покомпонентной независимости. Можно разбить входные данные (вектор) на фрагменты и выполнить соответствующую операцию над каждым фрагментом вектора независимо.

Последние 2 операции (5) VectorMult и (6) VectorGetNorm не обладают свойством покомпонентной независимости в общем, но возможно разбить эти операции на две части. В данном случае при вычислении, например, скалярного произведения векторов сначала производится покомпонентное умножение векторов; если выделить это в отдельную операцию, то она будет обладать свойством покомпонентной независимости, то же можно сказать и про вычисление евклидовой нормы вектора. Второй частью данных операций будут редукция и объединение всех частичных результатов. Редукцию можно выделить в отдельную операцию. В итоге эти две операции сводятся к двух типовым реализациям, первая из которых ввиду свойства покомпонентной независимости может быть распараллелена.

Рассмотрим матричные операции:

1. MatrixMultVAdd — умножение матрицы на вектор и добавление результата к другому вектору;

2. MatrixTMultVAdd — умножение транспонированной матрицы на вектор и добавление результата к другому вектору;

3. SMatrixProgonka — умножение обратной матрицы на вектор как решение СЛАУ методом прогонки.

Операцию (1) MatrixMultVAdd можно выполнить по-разному, матрицу можно разбить на фрагменты, однако в данной реализации каждая матрица хранится целиком на своём узле. Вектор, на который умножается матрица, можно разбить на фрагменты, тогда после умножения матрицы на вектор получим вектор, элементы которого будут только частичными суммами элементов итогового вектора. Для выигрыша по времени было принято решение собирать вектор для получения нефрагментированного вектора. Под сборкой вектора подразумевается получение нефрагментированного вектора из фрагментированных. Также в пользу сборки вектора говорит то, что операция сложения двух векторов будет обладать свойством покомпонентной зависимости. Эту операцию можно произвести над фрагментами векторов.

Данная операция наглядно показывает, что несмотря на то, что главная цель — это эффективность, в которую входит не только время итерации, но и потребление памяти, приходится выбирать, чем пожертвовать в пользу другого показателя. В данном случае для минимизации количеств пересылок между узлами вектор будет собираться перед операцией MatrixMultVAdd.

В операции (2) MatrixTMultVAdd будут получены вектора, элементы которых будут частичными суммами элементов итогового вектора. Для получения итогового вектора будет использована операция суммы всех векторов с частичными суммами, далее итоговый вектор будет снова разбит на фрагменты.

При операции (3) SMatrixProgonka матрица разбивается на диагональные блоки — это одно из условий выполнения этой операции параллельно.

На основе выполненного анализа становиться ясно, как должна быть построена фрагментированная программа для достижения эффективности. Далее будет описана фрагментированная программа задачи фильтрации и её поведение, что позволит реализовать её в системе автоматического конструирования параллельных программ.

2.2 Построение фрагментированной программы задачи фильтрации

В данном подразделе в 2.2.1 описана фрагментированная программа.

Поведение — это множество реализаций. В 2.2.2 описано поведение фрагментированной программы в 2 этапа, каждый этап уточняет поведение фрагментированной программы.

2.2.1 Описание фрагментированной программы

Опишем множество ФД.

Пусть FG — количество ФД, на которое разбиты вектора, тогда опишем множество ФД. Большинство названий ФД соответствует названиям структур данных из анализа 2.1. Множество G представляет собой множество индексов по пространственной координате, индекс — номер итерации в цикле. Множество F — множество индексов от 0..FG-1, индекс — номер фрагмента вектора. |F|=FG.

Рассмотрим множество всех ФД D = CF ∪ MF ∪ IF, где CF — фрагменты векторов, MF — фрагменты матриц, IF — фрагменты с промежуточными значениями.

Множество фрагментов векторов CF = V1 ∪ V2 ∪ pRes ∪ R ∪ pU ∪ pVb ∪ P ∪ new_pRes, где:

1. $V1 = \{ V1_{i,j} | \forall i \in G, \forall j \in F \}$
2. $V2 = \{ V2_{i,j} | \forall i \in G, \forall j \in F \}$
3. $pRes = \{ pRes_{i,j} | \forall i \in G, \forall j \in F \}$
4. $R = \{ R_{i,j} | \forall i \in G, \forall j \in F \}$
5. $pU = \{ pU_{i,j} | \forall i \in G, \forall j \in F \}$
6. $pVb = \{ pVb_{j} | \forall j \in F \}$
7. $P = \{ P_{i,j} | \forall i \in G, \forall j \in F \}$
8. $new_pRes = \{ new_pRes_{i,j} | \forall i \in G, \forall j \in F \}$

Множество фрагментов матриц MF = m_pB ∪ m_pD ∪ m_pBt, где:

1. m_pB
2. m_pD
3. m_pBt

Множество ФД с промежуточными значениями IF = qr ∪ ps ∪ a ∪ minus_a ∪ mult_v_a ∪ k1 ∪ k2 ∪ r ∪ Rnorm ∪ Iters ∪ m_K, где:

1. qr = { qr_i| ∀ i ∈ G}
2. ps = { ps_i| ∀ i ∈ G}
3. a = { a_i| ∀ i ∈ G}
4. minus_a = { $minus_a_i$| ∀ i ∈ G}
5. mult_v_a = { $mult_v_a_i$| ∀ i ∈ G}
6. k1
7. k2 = { $k2_i$| ∀ i ∈ G}
8. r = { r_i| ∀ i ∈ G}
9. Rnorm = { $Rnorm_i$| ∀ i ∈ G}
10. Iters
11. m_K

Рассмотрим множество ФВ.

Разобьём данное множество ФВ на три подмножества: в первое подмножество войдут ФВ, соответствующие векторным операциям; во второе — соответствующие матричным операциям; в третье — соответствующие остальным операциям.

ФВ представлены в виде триплетов <in, mod, out>, где in — множество входных ФД, mod —программный модуль, out — множество выходных ФД.

Векторные ФВ:

1. <{R_{i+1}}, $Norm_i$, {$Rnorm_i$}>
2. <{a_i, P_{i+1}, pU_i, }, $AddKV_r_i$, {pU_{i+1}}>
3. <{$minus_a_i$, new_pRes_i, R_i, }, $AddKV_r_i$, {R_{i+1}}>
4. <{$mult_v_a_i$, pVb}, $AddKV_pRes_i$, {$pRes_i$}>
5. <{R_i, R_i}, $vector_mult_i$, {qr_i}>

6. <{R_i}, $vector_set_init_i$, {P_i}>

7. <{k1, R_i, $k2_i$, P_i}, set_lin_i, {P_{i+1}}>

8. <{pVb, P_{i+1}}, $vector_mult_i$, {tmp_a_i}>

9. <{new_pRes_i, P_{i+1}}, $vector_mult_i$, {ps_i}>

Матричные ФВ:

1. <{P_{i+1}, m_pB}, $mult_v_add_i$, {$V1_i$}>

2. <{$V1_i$, m_pD}, $progonka_i$, {$V2_i$}>

3. <{$pRes_i$, m_pBt, $V2_i$}, $t_mult_v_add_i$, {new_pRes_i}>

Остальные ФВ:

1. <{qr_i, qr_{i-1}}, set_div_i, {$k2_i$}>

2. <{tmp_a_i, m_K}, set_mult_i, {$mult_v_a_i$}>

3. <{qr_i, ps_{i-1}}, set_div_i, {a_i}>

4. <{a_i}, set_minus_i, {$minus_a_i$}>

2.2.2 Описание поведения

В данном разделе будет определено поведение путём определения функций отображения, отношения эквивалентности ФД и т.п. Описание поведения будет проведено в два этапа.

Этап 1:

На данном этапе определим функции отображения на узлы для ФД и ФВ, удаление ФД. Все узлы пронумерованы неотрицательными целыми числами по порядку, ввиду этого отображение происходит на это множество номеров.

Опишем функцию отображения векторных ФД на узлы мультикомпьютера. Пусть NP — количество узлов мультикомпьютера, FG — количество ФД, j — индекс ФД. Тогда F(j) = {j * NP / FG} — множество (номер) узлов, на котором находится ФД с индексом j.

Необходимо, чтобы ФВ с индексом j, обрабатывающий ФД с индексом j, выполнялся на узле, где уже есть все необходимые данные, следовательно, функция отображения ФД на узлы мультикомпьютера и функция отображения ФВ на узлы мультикомпьютера имеют одинаковую зависимость от j.

Так как все узлы пронумерованы, то достаточно отображать на целое число, тогда можно задать отображение на множество узлов. Для матричных ФД и ФД с промежуточными значениями опишем функцию как F(j) = {0} — то есть все ФД отображаются на основной (нулевой узел).

Так как ФВ, на выход которым подаётся матричные ФД или ФД с промежуточными значениями, существуют не только на основном узле, то отображение матричных ФД и ФД с промежуточными значениями на основной узел — это ещё не оптимальное решение. Оптимальное решение данной проблемы будет описано на этапе 2.

Для определения поведения в полной мере необходимо задать их удаление. В основном это относится к ФД, подвергающимся изменениям, например, к векторным ФД и ФД с промежуточным значениями. Зададим следующие поведение для таких ФД: у нас есть информация, что определённый ФД будет потреблен N число раз, тогда будем считать, что после N потреблений данный ФД будет удалён из памяти.

Этап 2:

На данном этапе уточним поведение, задав репликацию некоторых ФД на множество узлов и отношение эквивалентности ФД.

Для матричных ФД и ФД с промежуточными значениями функция отображения F(I) = {0}; для работы ФВ необходимо, чтобы эти ФД были доступны на каждом узле. Используем другую функцию отображения. Проведём репликацию для каждого ФД R {0} → {0..NP-1}, данная функция производит репликацию для каждого матричного ФД и ФД с промежуточными значениями на всё множество доступных узлов.

Важно переиспользовать память, а именно чтобы все ФД из D, имеющие пространственную координату i — номер итерации цикла, отображались на один и тот же буфер в памяти. Для этого заведём отношение эквивалентности на множестве ФД. Два ФД эквивалентны, если они имеют один и тот же буфер в памяти. Например, фрагмент $R_{1,j} \equiv R_{2,j}$, то есть эти два ФД эквивалентны.

2.3 Анализ системы

Проанализируем, что необходимо иметь в системе, чтобы она эффективно выполнила фрагментированную программу, описанную в подразделе 2.2.

1. *Эффективное распределение по узлам*. Если системе доступно множество узлов и множество ФВ, их надо как-то отобразить. Данная задача не является тривиальной в общем случае, так как различные узлы могут обладать разными характеристиками, как и ФВ могут иметь разное время выполнения. В различных системах для этого есть свои подсистемы балансировки нагрузки, алгоритмы. Система должна быстро реагировать и на ситуацию в процессе работы программы.

В данной задаче фрагменты векторов делятся примерно поровну на все доступные узлы, поэтому свойства ФВ примерно одинаковые, но точно необходимо оперативное назначение ФВ на освободившийся узел в случае, если количество фрагментов не равно количеству узлов и потоков, а также если узлы имеют разные характеристики.

2. *Эффективное управление памятью*. ФД не должны оставаться в памяти во время всего времени работы программы. Система должна различать, какие ФД ещё будут использоваться, а какие уже можно освободить.

В данной задаче на каждой итерации каждый ФД будет потреблен константное число раз; это наиболее простой случай для автоматической сборки мусора.

3. *Операция редукции*. Большая часть коммуникаций в данной программе приходится на операцию редукции, наиболее критическое место для производительности всей программы. Система должна иметь методы оптимизации данной операции и выполнения её эффективно. Ввиду различных коммуникационных подсистем с разными характеристиками, такими как пропускная способность и задержка, система должна минимизировать количество коммуникаций между узлами в случае медленной коммуникационной подсистемы.

2.4 Выводы

Описана фрагментированная программа, задано её поведение в 2 этапа, каждый этап уточняет поведение фрагментированной программы. Проведён анализ минимального набора компонент системы автоматического конструирования параллельных программ, необходимых ей для успешного выполнения описанной фрагментированной программы.

3 Реализация фрагментированной программы

Система LuNA (Language for Numerical Algorithms) — это система автоматического конструирования параллельных программ для мультикомпьютеров, поддерживающая технологию фрагментированного программирования. Разрабатывается в ИВМиМГ СО РАН в Лаборатории синтеза параллельных программ. В состав системы ходят следующие компоненты:

1) язык фрагментированного программирования LuNA;
2) компилятор;
3) планировщик;
4) runtime-система LuNA.

Программа описывается на языке фрагментированного программирования LuNA как множество фрагментов. Для описания используются атомарные и структурированные фрагменты кода. Структурированный фрагмент кода — это процедура, реализованная на языке LuNA, а атомарный фрагмент кода — процедура, реализованная на языке C++. Компилятор транслирует LuNA-программы во внутреннее представление. Планировщик выполняет частичное статическое планирование вычислений на основе имеющейся информации о программе, о вычислительной системе. Исполнение производит runtime-система LuNA, кроме этого, в зону ответственности системы исполнения входит:

1) динамическое распределение ресурсов во время исполнения;
2) балансировка нагрузки;
3) организация асинхронных обменов данными.

Данный раздел посвящён разработке программы и её оптимизации, также анализу нефункциональных свойств программ, конструируемых системой LuNA для фрагментированных программ. Наибольший интерес представляет текущее состояние системы LuNA, её возможности и результаты, которые могут быть получены с помощью данной системы. Достижение удовлетворительной эффективности конструируемых программ в таких системах, как LuNA, является проблемой; потому интерес представляет то, какую производительность на текущий момент способна выдавать система на данном классе задач несмотря на

то, что системы автоматического конструирования дают результаты хуже, чем ручное программирование, по крайней мере на начальном этапе развития системных алгоритмов. Конструирование программы с 2-3-кратным отставанием является удовлетворительным. Основной вопрос, на который отвечает данный раздел:

- какие методы надо использовать для достижения эффективности конструируемой LuNA программы?

В рамках данного раздела была реализована и оптимизирована LuNA программа, реализующая алгоритм фильтрации. Ниже рассмотрим эту реализацию в 4 этапа, которые соответствуют хронологическим этапам разработки и могут быть использованы для решения других похожих задач. Каждый этап приближает показатели эффективности реализации задачи в системе LuNA к MPI реализации алгоритма.

3.1 Реализация алгоритма в системе LuNA

Реализация в системе LuNA была создана на основе параллельной программы **[2,3]** и фрагментированной программы, описанной в разделе 2. Подробно задача фильтрации описывается в разделе 2.1. Использовалась схема параллельного метода сопряжённых градиентов (Рисунок 1). Для каждой ФВ из описания 2.2 создан фрагмент кода с такими же входными и выходными данными. Аналогично были определены ФД.

Как пример можно рассмотреть ФВ Norm из описания фрагментированной программы 2.2. Есть входные и выходные данные, тип зависимости по данным и его особенности. Благодаря этой информации можно создать соответствующий ФВ в программе LuNA (листинг 1).

Листинг 1 — Подпрограмма Norm

```
01: sub Norm(name FNorm, name F)
02: {
03:     df tmp;
04:     for i=0..$FG-1
05:         norm(tmp[i], F[i]);
06:     ReduceSum(FNorm, tmp, $FG);
```

```
07: }
```

В листинге 1 представлена фрагментированная подпрограмма Norm. Выходными данными являются норма вектора F. Сам вектор F представлен в фрагментированном виде, каждому фрагменту соответствует свой ФД в программе LuNA. Фрагмент кода состоит из ФВ norm (строка 05), который производит покомпонентное умножение соответствующего ФД и ФВ ReduceSum (строка 06), который производит суммирование всех значений, полученных на предыдущем этапе. Константа $FG — значение общего количества ФД, на которое разбит вектор F. Данная фрагментированная подпрограмма соответствует векторной операции *VectorGetNorm* описанной в подразделе 2.1. Как можно увидеть, она была поделена на две части, первая часть представлена в стоках 04, 05, вторая — в строке 06.

Рассмотрим, как реализован атомарный фрагмент кода norm использующейся в листинге 1 строке 05.

Листинг 2 — Атомарный фрагмент кода norm

```
01: void norm(DF &norm, const DF &arr) {
02:    vector_mult(norm, arr, arr);
03: }
```

В листинге 2 представлен атомарный фрагмент кода norm. Входные данные — параметр arr, являющийся ФД, выходные данные — ФД norm, который вычисляется посредством подпрограммы vector_mult; реализация данной подпрограммы представлена в листинге 3.

Листинг 3 — Подпрограмма vector_mult

```
01: void vector_mult(DF &res, const DF &v1, const DF &v2) {
02:    assert(v1.getSize() == v2.getSize());
03:    const double *d1 = v1.getData<double>(),
04:          *d2 = v2.getData<double>();
05:
06:    double r = 0;
07:
```

```
08:     for (auto i = 0u; i < v1.getSize() / sizeof(double); i++)
09:         r += d1[i] * d2[i];
10:
11:     res.setValue(r);
12: }
```

В листинге 3 представлена подпрограмма vector_mult. Данная подпрограмма производит покомпонентное умножение фрагментов векторов, представленных в виде фрагментов данных v1, v2.

Структурированный фрагмент кода ReduceSum подробно рассмотрен в подразделе 3.3 в пункте 3.

Основная цель данного этапа — написание LuNA программы. Производилась проверка соответствия выходных данных: выходные данные программ, реализованных на MPI и на LuNA, должны совпадать.

На данный момент развития системы LuNA алгоритмы автоматического управления памятью и распределением по узлам в системе LuNA не могут обеспечить удовлетворительную эффективность в данной задаче, поэтому необходима настройка исполнения.

Основные проблемы:

1) чрезвычайно большой расход памяти из-за плохого автоматического управления памятью;

2) алгоритмы автоматического распределения по узлам в сети производят неэффективное распределение ФД и ФВ.

3.2 Настройка исполнения, использование основных директив

Цель данного этапа — базовая настройка исполнения LuNA-программы посредством директив.

Этот этап устраняет некоторые проблемы первого этапа.

Основные изменения представлены ниже:

1) В 2.2 описан подход для работы с памятью. Такого поведения можно добиться средствами системы LuNA, используя директиву req_count. С помощью данной директивы можно задать количество потреблений для ФД как условие его удаления.

Пример в листинге 4.

Листинг 4 — Подпрограмма set_lin

```
01: sub set_lin(name res, name k1, name v1, name k2, name v2)
02: {
03:     for i=0..$FG-1
04:         cf setlin_mat[i]: setlin_mat(res[i], k1, v1[i], k2, v2[i]) @{
05:             locator_cyclic:$LOC1(i);
06:             req_count res[i] = 5;
07:         };
08: } @ {
09:     locator_cyclic res[n] => $LOC1(n);
10:     locator_cyclic v1[n] => $LOC1(n);
11:     locator_cyclic v2[n] => $LOC1(n);
12: }
```

В листинге 4 представлена фрагментированная подпрограмма set_lin, необходимая для реализации алгоритма. Используется директива req_count “имя ФД”[индекс] = целое число. В данном фрагменте кода указывается, что ФД res[i] будет потреблён 5 раз, после чего удалён автоматически средой выполнения.

Есть ещё один подход для решения этой проблемы — использование директивы delete. Данная директива указывает системе, что конкретный ФД нужно удалить.

2) В 2.2 были определены функции отображения для разных классов ФД и ФВ. В системе LuNA есть подходящая директива для задания таких отображений, которые называются локаторами. Локаторы необходимы для более эффективного управления распределением по узлам. Локаторы реализуют выбор узлов в распределённой системе. Директива locator_cyclic — одна из директив локаторов. Эта директива работает как кольцо вычетов по модулю N, где N — количество узлов, доступное системе. Данный локатор указывает, на каком узле будет выполняться ФВ и на каком узле хранить ФД (листинг 5);

Листинг 5 — Подпрограмма vector_mult

```
01: sub vector_mult(name res, name X, name Y) {
02:     df tmp;
```

```
03:     for i=0..$FG-1
04:         vector_mult_mat(tmp[i], X[i], Y[i]) @ {
05:             locator_cyclic:$LOC1(i);
06:             req_count tmp[i] = 1;
07:         };
08:     reduce_sum(res, tmp, tmp, $FG);
09: } @ {
10:     locator_cyclic Y[n] => $LOC1(n);
11:     locator_cyclic X[n] => $LOC1(n);
12:     locator_cyclic tmp[n] => $LOC1(n);
13: }
```

В листинге 5 представлена фрагментированная подпрограмма реализации vector_mult. Данный фрагмент кода состоит из ФВ vector_mult_mat и reduce_sum. Произведение фрагментов векторов производит vector_mult_mat, а объединение результатов — reduce_sum. В строке 05 указана директива locator_cyclic, которая указывает на узел, на котором будет выполнен соответствующий ФВ. Директивы, указанные в строках 10, 11, 12 указывают, где находятся соответствующие ФД.

3) Построение программы в системе LuNA на основе уже имеющейся параллельной реализации накладывает некоторые ограничения. Одно из таких ограничений — это структуры данных, которые используются в оригинальной программе. В данном случае говорится про структуру Matrix, которая не лежит непрерывно в памяти, в отличии от ФД системы LuNA. Некоторые ФВ используют в процессе своей работы именно структуры Matrix, для использования этой структуры в ФВ необходима сериализация и десериализация в ФД. Причём для экономии памяти нужно отображение Matrix в ФД. То есть фактически Matrix превращается в интерфейс для доступа к различным областям соответствующего ФД.

После данных манипуляций над программой LuNA было проведено сравнительное тестирование, которое показало отставание реализации LuNA от MPI реализации в 5 раз на общей памяти на одном узле, и в 30-40 раз на распределенной памяти. Такие показатели являются неприемлемыми, в связи с

этим целесообразна дальнейшая настройка исполнения. Тестирование данной программы можно найти в подразделе 4.1.

3.3 Дополнительная настройка исполнения, использование дополнительных директив

Цель данного этапа — более тонкая настройка исполнения LuNA-программы, затрагивающая более сложные механизмы исполнения.

Изменения на данном этапе:

1) На этапе 2 подраздела 2.2.2 определена функция реплицирования для подмножества ФД. В системе LuNA есть возможность реплицирования ФД посредством директивы locator_replicating. Она указывает на то, что на определённом множестве узлов нужно создать копии ФД (листинг 6).

Листинг 6 — Подпрограмма init_args

```
init_args(m_R_init_full, m_pU_init_full,
    m_pVb_init_full, m_K, m_pB , m_pD, m_pBt) @ {
    req_count m_R_init_full = $FG/$NP + 1;
    req_count m_pU_init_full = $FG/$NP;
    req_count m_pVb_init_full = $FG/$NP;
    locator_replicating m_pB => all;
    locator_replicating m_pD => all;
    locator_replicating m_pBt => all;
    locator_replicating m_R_init_full => all;
    locator_replicating m_pU_init_full => all;
    locator_replicating m_pVb_init_full => all;
};
```

В листинге 6 представлена фрагментированная подпрограмма, инициализирующая данные, такие как матрицы m_pB, m_pBt, m_pBt, locator _replicating “имя ФД” => множество узлов. В данном листинге директива locator_replicating указывает скопировать ФД с этими матрицами на все доступные узлы. Если ФВ запросит любой из этих ФД, то в результате запроса будут получены данные, находящиеся в локальной памяти соответствующего узла.

2) На 2 этапе подраздела 2.2.2 было задано отношение эквивалентности для ФД, для отображения подмножеств ФД на один и тот же буфер в памяти. Для

реализации этого поведения в системе LuNA был использован механизм уничтожающего потребления. Этот механизм позволяет использовать и изменять буфер предшествующего ФД в текущем ФД.

3) В алгоритме существуют состояния, в которых необходимо иметь нефрагментированный вектор для дальнейших вычислений. Для объединения фрагментов векторов в нефрагментированный вектор разработана фрагментированная подпрограмма, которой на вход подаются фрагменты векторов; на выходе же выдаётся нефрагментированный вектор. Фрагментированная подпрограмма рекурсивно собирает нефрагментированный вектор. Для эффективности данная подпрограмма сначала собирает все ФД внутри узла, объединяя их в единый ФД, после данные ФД с разных узлов объединяет в единый ФД, представляющий нефрагментированный вектор.

Листинг 7 — Подпрограмма reduce_sum

```
01: sub reduce_sum(name res, name arr, name arr1,
                   int count, int deep, int req_count)
01: {
02:     df tmp;
03:     if count == 1 {
04:         set(res, arr[0]) @ {
05:             locator_cyclic:$LOC1(0);
06:             req_count res = req_count;
07:         };
08:     }
09:     if count != 1 {
10:         for i=0..count / 2 - 1
11:             cf set[i]: set_add_real(tmp[i], arr[i * 2], arr1[i * 2 + 1]) @ {
12:                 locator_cyclic:$LOC2(i,deep);
13:                 req_count tmp[i]=1;
14:             };
15:
16:         if count % 2 == 0
17:             reduce_sum(res, tmp, tmp, count/2, deep * 2, req_count);
18:         if count % 2 == 1 {
19:             set(tmp[count / 2], arr[count - 1]) @ {
```

```
20:                 locator_cyclic:$LOC2(count / 2, deep);
21:                 req_count tmp[count / 2]=1;
22:             };
23:             reduce_sum(res, tmp, tmp, count / 2 + 1,deep * 2 ,req_count);
24:         }
25:     }
26: } @ {
27:     locator_cyclic arr[n] => $LOC2(n,deep/2);
28:     locator_cyclic arr1[m] => $LOC2(m,deep/2);
29:     locator_cyclic tmp[h] => $LOC2(h,deep);
30:     locator_replicating res => all;
31: }
```

В листинге 7 представлена подпрограмма reduce_sum. В строках 02—08 база рекурсии. В строках 10—14 сложение значений ФД, обработка двух случаев для чётного количества ФД в строке 16 и нечетного в строке 18. Целочисленная переменная deep — глубина рекурсии. Она необходима для вычисления номера узла, на котором находятся необходимые ФД. Объединение с разных узлов происходит на нулевом узле, после посредством директивы locator_replicating реплицируется на все узлы, доступные системе.

Тестирование показало, что отставание реализации LuNA к MPI реализации удалось сократить. В общей памяти показатели почти приблизились к удовлетворительным. На распределенной памяти отставание в 6—10 раз в зависимости от количества узлов. Результаты тестов можно найти в подразделе 4.2.

3.4 MPI процедура

Как показало профилирование, узким местом в программе является операции gather, reduce, scatter. Из-за особенностей данных операций здесь происходит наибольшее количество рассылок по сети.

К сожалению, средствами LuNA данные операции реализовать эффективно на данный момент невозможно, но в системе LuNA есть возможность описывать подпрограммы на MPI. Именно эти возможности можно использовать, чтобы эффективно реализовать операции gather, reduce, scatter.

Данную подпрограмму можно логически поделить на 4 части:

1) запрос всех необходимых ФД;

2) сборка фрагментов вектора в рамках одного узла (в общей памяти) или суммирование частичных сумм одного узла;

3) объединение фрагментов векторов с каждого узла в один вектор посредством MPI_Allgatherv или суммирование частичных сумм всех узлов;

4) помещение собранного вектора или итоговой суммы на все доступные узлы.

Для пользователя MPI подпрограмма выглядит как обычный фрагмент вычислений со своими входными, выходными данными и директивами.

В программе LuNA это выглядит так, как показано в листинге 8.

Листинг 8 — Вызов reduce_sum_mpi

```
01: reduce_sum_mpi(tmp, res) @ {
02:     number_dfs: $FG;
03:     number_dfs: 1;
04: };
```

В листинге 8 представлен вызов ФВ reduce_sum_mpi. Входные данные — tmp, выходные — res. В строке 02 указано количество входных ФД, в строке 03 — выходных.

Листинг 9 — Реализация reduce_sum_mpi

```
01: class ReduceSum : public SubroutineInstance {
02:    using SubroutineInstance::SubroutineInstance;
03:     double result = 0;
04:     double local_result = 0;
05:     std::mutex mutex;

06:     void request_dfs() override {
07:         int idsize = id_info.size() - 1;
08:         for (int sz = 0; sz < idsize; sz++) {
09:             int n = id_info[sz].number[0];
10:             int size = mpiSize;
11:             int threads = n / mpiSize;
12:             for (int i = mpiRank * threads;
```

```
        i < threads * (mpiRank + 1);
13:         i++) {
14:           request(id_info[sz].id[i],
          CyclicLocator(mpiRank),
          [sz, 15: this](const DF &df) {
16:             on_receive_df(sz, df);
17:           });
18:         }
19:       }
20:     }
21:
22:     void on_receive_df(int subId, const DF &df) {
23:       if (subId == 0) {
24          std::lock_guard <std::mutex> lock(mutex);
25:         local_result += df.get_real();
26          mutex.unlock();
27:       }
28:
29:       std::atomic_fetch_add(&received_count[subId], 1);
30:
31:       if (received_count[subId]
       == id_info[0].number[0] / mpiSize) {
32:         submit_job([this]() {
33:           MPI_Allreduce(&local_result, &result, 1,
       MPI_DOUBLE,  MPI_SUM, MPI_COMM_WORLD);
34:           DF out;
35:           out.setValue(result);
36:           post(id_info[1].id, out,
       CyclicLocator(mpiRank),
       id_info[0].number[0]*mpiSize + 2);
37:           destroy_callback();
38:         });
39:       }
40:     }
41: };
```

В листинге 9 представлена реализация ФВ reduce_sum_mpi. В соответствии с логическими частями, изложенными выше, она поделена на 4 части. В строке 06 метод, который выполняет запрос всех необходимых ФД. В

строке 25 суммирование частичных сумм одного узла. В строке 33 происходит суммирование частичных сумм всех узлов посредством MPI_Allreduce. В строке 36 посредством post итоговая сумма помещается на соответствующий узел.

3.5 Итоги

В результате была реализована и оптимизирована LuNA-программа, реализующая алгоритм фильтрации. Получены несколько реализаций, каждая из которых имеет свои нефункциональные свойства. Особенностями этих реализаций является то, что каждая следующая реализация показывает улучшение показателей эффективности по сравнению с предшественником.

4 Экспериментальное исследование

Проведём тестирование полученных программ для оценки эффективности программы LuNA по сравнению с реализацией MPI. Удовлетворительным результатом тестирования считали отставание LuNA программы в 2-3 раза по сравнению с MPI реализацией.

Тестирование проводилось на информационно-вычислительном центре Новосибирского государственного университета (Кластер НГУ) **[27]** и на Межведомственным Суперкомпьютерном Центром Российской Академии Наук (Кластер МВС) **[28]**. На кластере МВС использовались две коммуникационные подсистемы: Ethernet и InfiniBand. Были выбраны следующие параметры задачи (данные параметры использовались для всех тестов):

- размер сетки: 100x100x100
- количество итераций: 100
- количество шагов по времени: 1
- размер временного шага: 2000

Структура тестирования:

Тестирование 1, 2 посвящено промежуточным реализациям LuNA программы, описанным в подразделах 3.2 и 3.3 раздела 3 соответственно. Тестирование проводилось только на кластере НГУ.

Тестирование 3 посвящено итоговой реализации LuNA программы. Тестирование 3 разбито на два подраздела — тестирование на кластере НГУ и кластере МВС.

4.1 Тестирование LuNA-программы этапа 2

Тестирование реализации, описанной в 3.2, проводилось на общей памяти на одном узле, на котором использовалось 4 потока, и на 2 и 4 узлах в распределенной памяти, по 4 потока на каждый узел. Тестирование проводилось на кластере НГУ.

Результаты представлены на рисунке 2.

Рисунок 2 — Тестирование 1

На графике представлены результаты тестирования, описанного в 3.2. По вертикали откладывается время в секундах, по горизонтали — количество узлов. Количество потоков на каждый узел — 4 для MPI реализации и LuNA реализации. По графику можно сделать вывод, что результаты неудовлетворительные, на распределенной памяти отставание от MPI реализации в 40 раз, а на общей памяти в 5 раз.

4.2 Тестирование LuNA-программы этапа 3

Тестирование реализации, описанной в 3.3. Использовались аналогичные параметры тестирования, описанные в 4.4.1. Результаты представлены на рисунке 3.

Рисунок 3 — Тестирование 2

На графике представлены результаты тестирования реализации, описанной в 3.3. По вертикали время откладывается в секундах, по горизонтали — количество узлов. Количество потоков на каждый узел — 4 для MPI реализации и LuNA реализации. По графику можно сделать вывод, что результаты неудовлетворительные, на распределенной памяти отставание от MPI реализации примерно в 10 раз и на общей памяти примерно в 5 раз. Но результаты улучшились по сравнению с реализацией, описанной в разделе 3.2.

4.3 Тестирование LuNA-программы

Тестирование итоговой реализации, описанной в разделе 3.4. Было проведено тестирование на двух кластерах: НГУ и МВС. Причём на МВС тестирование проводилась на двух коммуникационных подсистемах: Ethernet и InfiniBand.

Для тестирования было выбрано оптимальное количество потоков для каждого кластера и каждой реализации. Выбор оптимального количества потоков осуществлялся с помощью запуска программы с изменением количества потоков от 1 до количества потоков, при котором результаты по времени становились хуже по отношению к предыдущему запуску программы. Было взято количество потоков, при котором программа отрабатывала за наименьшее время. Результаты представлены в таблице 1.

Таблица 1 — Оптимальное количество потоков на разном количестве узлов

Количество потоков					
	Кластер НГУ		Кластер МВС		
Кол—во узлов	MPI	LuNA	MPI	LuNA	LuNA IB
1	8	12	8	18	21
2	4	15	8	18	21
4	4	15	8	18	21
6	4	15	—	—	—

8	—	—	8	18	21
12	—	—	8	18	21
16	—	—	8	18	21

4.3.1 Тестирование на кластере НГУ

Результаты представлены на рисунках 4,5,6.

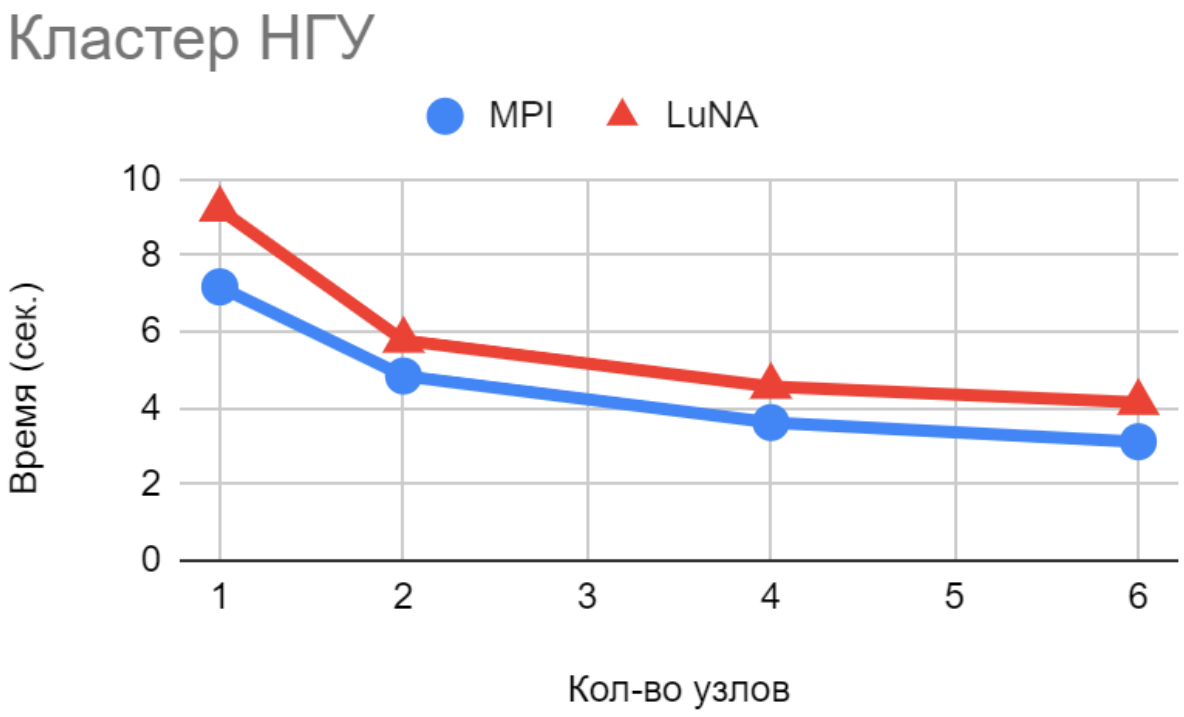

Рисунок 4 — Тестирование на кластере НГУ

На графике представлены результаты тестирования реализации алгоритма фильтрации в системе LuNA. По вертикали время откладывается в секундах, по горизонтали — количество узлов. По графику можно сделать вывод, что отставание реализации LuNA можно оценить в 30% от MPI реализации, что оценивается как удовлетворительный результат.

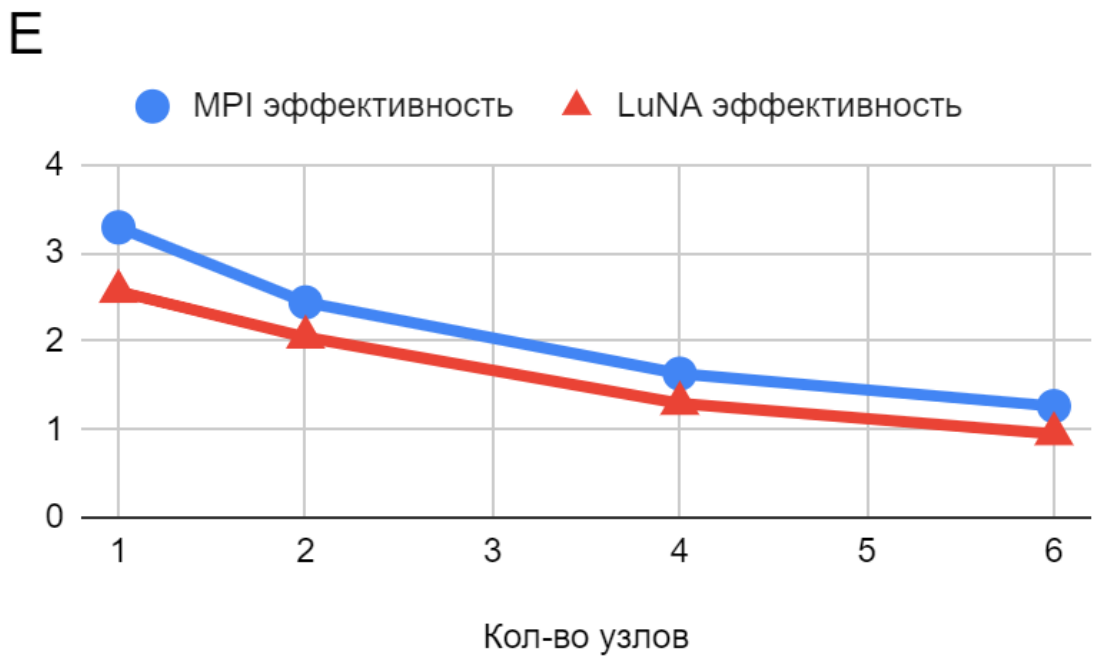

Рисунок 5 — Эффективность на кластере НГУ

На рисунке 6 представлена эффективность двух реализаций. На данном графике отображена эффективность на узел относительно последовательной программы. По графику можно сделать вывод, что по эффективности реализация LuNA отстаёт от MPI реализации, но не критично. Результаты удовлетворительны.

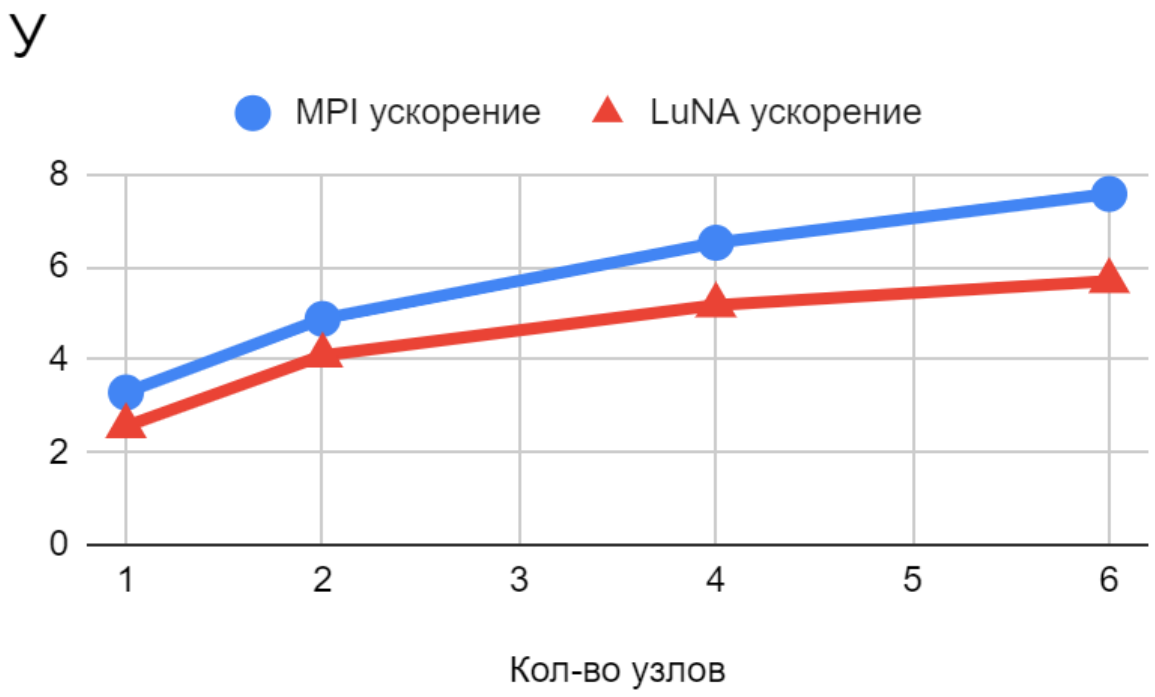

Рисунок 6 — Ускорение на кластере НГУ

На данном графике представлено ускорение двух реализаций. По вертикали откладывается ускорение, по горизонтали — количество узлов. Показатели ускорения LuNA-программы отстают от MPI реализации.

4.3.2 Тестирование на кластере МВС

Результаты на коммуникационной подсистеме Ethernet представлены на рисунке 7.

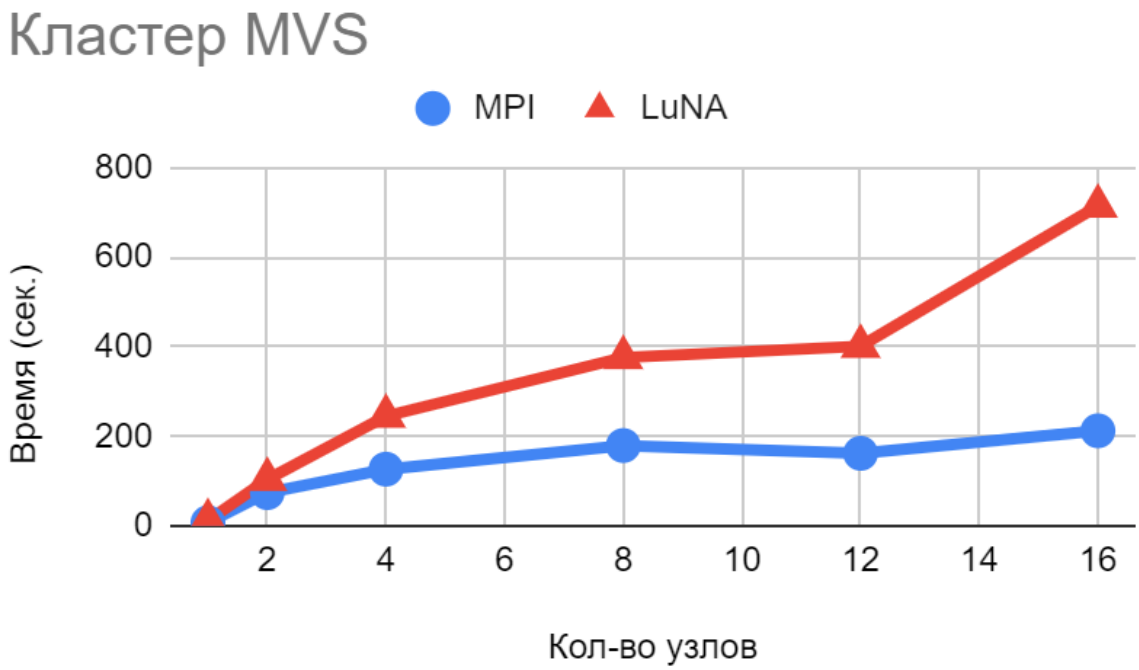

Рисунок 7 — Тестирование на кластере МВС на коммуникационной подсистеме Ethernet

На графике представлены результаты тестирования реализации алгоритма фильтрации в системе LuNA на коммуникационной подсистеме Ethernet.По вертикали время откладывается в секундах, по горизонтали — количество узлов. По графику можно сделать вывод, что отставание реализации LuNA можно оценить в 3-3,5 раза от MPI реализации, что является пограничным результатом. Тут можно объяснить этот результат тем, что Ethernet является наиболее медленной коммуникационной подсистемой из всех доступных для тестирования. Система LuNA, к сожалению, из-за своих особенностей является более чувствительной к скорости и задержке коммуникаций между узлами, чем MPI.

Результаты на коммуникационной подсистеме InfiniBand представлены на рисунке 8.

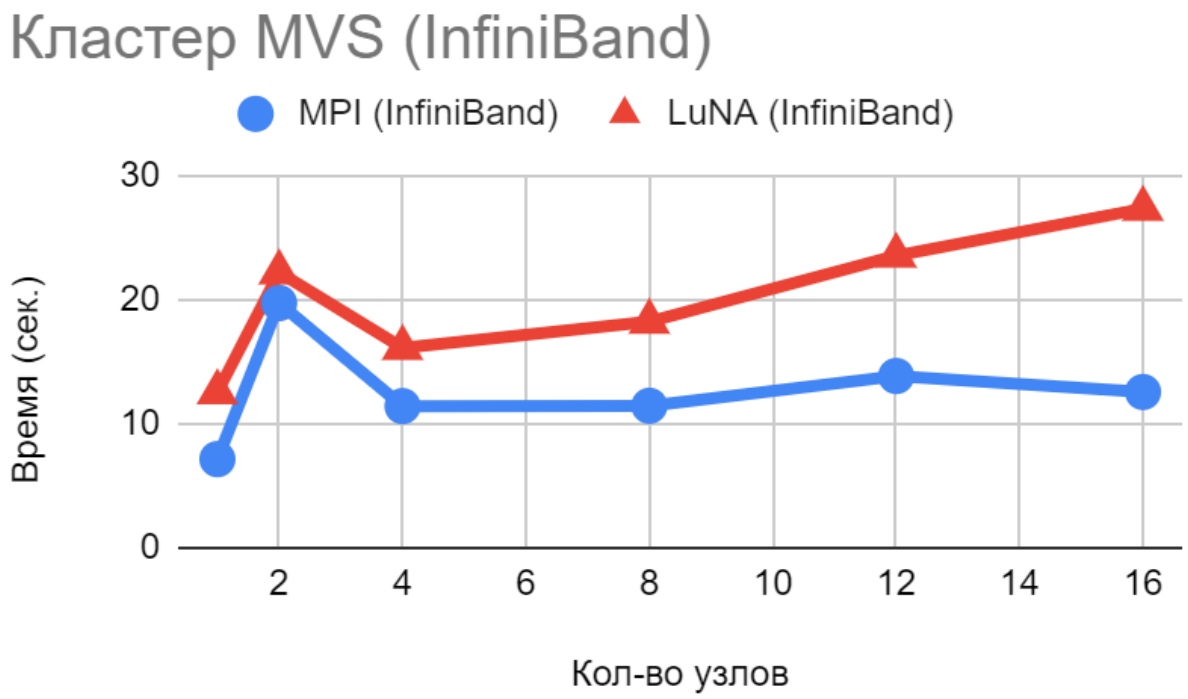

Рисунок 8 — Тестирование на кластере МВС на коммуникационной подсистеме InfiniBand

На графике представлены результаты тестирования реализации алгоритма фильтрации в системе LuNA на коммуникационной подсистеме InfiniBand.По вертикали откладывается время в секундах, по горизонтали — количество узлов. По графику можно сделать вывод, что отставание реализации LuNA можно оценить в 2-2,5 раза от MPI реализации, что является удовлетворительным результатом. Результат лучше, чем с коммуникационной подсистемой Ethernet, так как InfiniBand является более быстрой коммуникационной подсистемой.

Отметим, что на кластере МВС при увеличении количества узлов время увеличивается — это связано в первую очередь с тем, что много времени занимают коммуникации между узлами. Ввиду того, что коммуникационные подсистемы, доступные на кластере МВС имеют меньшую пропускную способность и большую задержку, чем на кластере НГУ, были получены такие результаты.

4.4 Итоги тестирования

Тестирование показало рост эффективности программы LuNA с каждым этапом настройки исполнения. На последнем этапе тестирования удалось добиться отставания LuNA программы от MPI реализации в 2-3 раза, что является удовлетворительным результатом.

ЗАКЛЮЧЕНИЕ

Разработана эффективная программа задачи фильтрации и проанализированы её нефункциональные свойства. А именно разработана программа, реализующая краевую задачу фильтрации двухфазной жидкости в системе LuNA в 4 этапа, проведено экспериментальное исследование программы LuNA и программы MPI, которое показало, что LuNA-программа отстаёт от MPI реализации в 2-3 раза, что является удовлетворительным результатом.

Защищаемые положения:

1.разработана и настроена LuNA программа, решающая краевую задачу фильтрации;

2. проведены экспериментальные исследования производительности LuNA реализации, сравнение с MPI реализацией.

В дальнейшем планируется улучшение различных подсистем LuNA с целью сокращения отставания системы LuNA от MPI на данном классе задач, а также проведение аналогичные исследований для других классов задач.

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».

Нуштаев Юрий Юрьевич _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Malyshkin V.E., Perepelkin V.A. LuNA Fragmented programming system, Main functions and peculiarities of run—time subsystem // Parallel Computing Technologies. PaCT 2011. Lecture Notes in Computer Science. — V. 6873. — Berlin; Heidelberg: Springer, 2011. — https://doi.org/10.1007/978—3—642—23178—0_5.

2. Ivanov M.I., Kremer I.A., and Laevsky Yu.M. On the streamline upwind scheme of solution to the filtration problem // Siberian Electronic Mathematical Reports. ——— 2019. ——— V. 16. ——— P. 757—776.

3. Ivanov M.I., Kremer I.A., and Laevsky Yu.M. On wells modeling in filtration problems // Siberian Electronic Mathematical Reports. ——— 2019. ——— V. 16. ——— P. 1868—1884.

4. В.Э.Малышкин, “Технология фрагментированного программирования”, Вестн. ЮУрГУ. Сер. Выч. матем. информ., 2012, № 1, 45—55

5. Вальковский В.А., Малышкин В.Э. Синтез параллельных программ и систем на вычислительных моделях. — Издательство "Наука", Сибирское отд—е, Новосибирск, 1988г., 129 стр.

6. Эффективная фрагментированная реализация краевой задачи фильтрации двухфазной жидкости А.А. Кудрявцев, В.Э. Малышкин, Ю.Ю. Нуштаев, В.А. Перепёлкин, В.А. Спирин //Проблемы информатики, Новосибирск

7. DVM — система разработки параллельных программ [Электронный ресурс]. URL: http://dvm—system.org/ru/about/ (дата обращения 01.05.23)

8. DVM — система [Электронный ресурс]. URL: https://www.keldysh.ru/dvm/dvmhtm1107/rus/index.html (дата обращения 01.05.23)

9. Эффективность выполнения DVM-программ [Электронный ресурс].URL: https://www.keldysh.ru/dvm/dvmhtm1107/rus/eff0302r.html (дата обращения 01.05.23)

10. Язык НОРМА / А.Н.Андрианов [и др.] //Препринты ИПМ им. М.В.Келдыша. 2019. № 132. 48 с. http://doi.org/10.20948/prepr—2019—132 URL: http://library.keldysh.ru/preprint.asp?id=2019—132

11. Андрианов А.Н. Применение языка Норма для решения задач на вложенных сетках // Вычислительные методы и программирование, №1, Т.3, 2002, с.103—112

12. Язык параллельного программирования HPF [Электронный ресурс]. URL: http://hpc.sfedu.ru/tutor/high_performance_computing/chapter1/page15.html (дата обращения 01.05.23)

13. Ken Kennedy, Charles Koelbel, Hans Zima, The Rise and Fall of High Performance Fortran: An Historical Object Lesson, Proceedings of the third ACM SIGPLAN conference on History of programming languages 2007, San Diego, California, 2007

14. Coarray Fortran [Электронный ресурс]. URL: http://caf.rice.edu (дата обращения 01.05.23)

15. J.Reid. Coarrays in the next Fortran Standard. ISO/IEC JTC1/SC22/WG5 №1702—2

16. Manuel Hasert, Harald Klimach, and Sabine Roller “CAF versus MPI — Applicability of Coarray Fortran to a Flow Solver” German Research School for Simulation Sciences GmbH, 52062 Aachen, Germany 2 RWTH Aachen University, 52062 Aachen, Germany

17. Laxmikant V. Kale, Sanjeev Krishnan, CHARM++: A portable concurrent object oriented system based on C++, University of Illinois at Urbana—Champaign, Champaign, IL, 1993

18. Charm++. Parallel Computer Network. [Электронный ресурс] URL: http://charmplusplus.org/ (дата обращения 01.05.23)

19. Charm++. Applications. [Электронный ресурс] URL: http://charmplusplus.org/applications/ (дата обращения 01.05.23)

20. А. В. Мазеев, А. С. Семенов, А. С. Фролов, “Сравнение технологий параллельного программирования MPI и Charm++ на примере задачи построения минимального остовного дерева в графе”, Comp. nanotechnol., 2015, № 4, 18—25

21. The KeLP Programming System // [Электронный ресурс]. Режим доступа: http://cseweb.ucsd.edu/groups/hpcl/scg/kelp/, свободный.

22. S.B. Baden, P. Colella , D. Shalit, B. Van Straalen, "Abstract KeLP", 10th SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, March 2001.

23. S. B. Baden and S. J. Fink The Data Mover: A Machine—Independent Abstraction for Managing Customized Data Motion, LCPC '99, La Jolla, CA, 1999.

24. Abramov S.M., Adamowitch A.I., Nesterov I.A., Pimenov S.P., Shevchuck Yu.V. Autotransformation of evaluation network as a basis for automatic dynamic parallelizing//Proceedings of NATUG 1993 Meeting “Transputer: Research and Application”, May 10—11, 1993.

25. Abramov S.M., Nesterov I.A., Shevchuk Yu.V. T—language. Preliminary description//RCMS Tech. Report #09/18/1994. Доступен в Интернет: ftp://pub/local/RCMS/SSPA/tlp.ps (а также как tlp.ps.Z, tlp—ps.zip).

26. С.М. Абрамов, А.И. Адамович Т—система—среда программирования с поддержкой автоматического динамического распараллеливания программ // Институт программных систем РАН 1998 Переславль—Залесский

27. Информационно—вычислительный центр Новосибирского государственного университета [Электронный ресурс] / — Режим доступа: http://nusc.nsu.ru/wiki/doku.php/doc/index

28. Межведомственный Суперкомпьютерный Центр Российской Академии Наук [Электронный ресурс] / — Режим доступа: http://www.jscc.ru/

ПРИЛОЖЕНИЕ А

Программа, решающая краевую задачу фильтрации двухфазной жидкости в системе LuNA

Руководство программиста

Листов 8

Новосибирск 2023

Аннотация

В данном документе приведено руководство программы, решающей краевую задачу фильтрации двухфазной жидкости в системе LuNA. Исходным языком программы является язык LuNA. Средство разработки – редактор исходного кода Notepad++ и Clion от компании JetBrains.

Основной функцией программы является моделирование задачи на практике.

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

Содержание

Содержание

1. Назначение и условия применения программы 51
1.1 Назначение программы 51
1.2 Функции, выполняемые программой 51
1.3 Условия, необходимые для выполнения программы 51
2. Характеристика программы 52
2.1 Описание основных характеристик программы 52
3. Обращение к программе 53
3.1 Конфигурирование и запуск программы 53
4. Входные и выходные данные 54
4.1 Организация используемой входной информации 54
4.2 Организация используемой выходной информации 54

1. Назначение и условия применения программы

1.1 Назначение программы

Программа, решающая краевую задачу фильтрации двухфазной жидкости в системе LuNA используется для моделировании на практике.

1.2 Функции, выполняемые программой

1) Решение краевой задачи фильтрации двухфазной жидкости в трёхмерной области при наличии скважин

1.3 Условия, необходимые для выполнения программы

Наличие входных данных программы, вычислительной системы с операционной системой Ubuntu 20.04.x LTS и системой LuNA.

2. Характеристика программы

2.1 Описание основных характеристик программы

Программа, решающая краевую задачу фильтрации двухфазной жидкости в системе LuNA используется для моделировании на практике.

2.2 Описание основных особенностей программы

Входные файлы являются текстовые файлы, которые представляют собой сериализованные данные различных структур данных.

3. Обращение к программе

3.1 Конфигурирование и запуск программы

Запуск программы происходит посредством mpirun.

Пример: mpirun —report—bindings —np 4 ~/luna/bin/rts ——worker—threads—count=6 ~/luna_project3/build/libucodes.so —u

В данном случае программа запускается на 4 процессах, каждому из которых доступно максимум 6 потоков.

Скрип для запуска на кластере НГУ представлен в листинге 1.

Листинг 10 — Скрип для запуска на кластере НГУ

```
#!/bin/bash

#PBS -l walltime=00:50:00
#PBS -l select=4:ncpus=4:mem=1500m,place=scatter
#PBS -m n

cd $PBS_O_WORKDIR

MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')
echo "Number of MPI process: $MPI_NP"

echo 'File $PBS_NODEFILE:'
cat  $PBS_NODEFILE
echo

module load luna-mpi-subroutine

CXX=mpic++ luna --worker-threads-count=6 --allow-growing-thread-pool --compile-only --build-dir=build
--verbose f3d-mpi-subroutine.fa
mpic++ -o build/libucodes.so build/ucodes.o build/foreign_blocks.o build/test.o -ldl -L ~/mpi-
subroutine/lib -lrts -shared -lsolver -fPIC -L ~/luna_project3 -lfiltration_base -lshared -lsolver
time mpirun --bind-to socket -report-bindings -hostfile $PBS_NODEFILE -np $MPI_NP ~/mpi-
subroutine/bin/rts --worker-threads-count=6 --allow-growing-thread-pool
~/luna_project3/build/libucodes.so -u
module unload luna-mpi-subroutine
```

4. Входные и выходные данные

4.1 Организация используемой входной информации

Входная информация, представляет собой файлы формата .txt m_pB, m_pBt, m_pD, pU, pVb, R являющееся соответствующими сеарилизоваными структурами данных. Они представляют начальные данные. Их генерация и сериализация выполняется отдельной специальной программой.

4.2 Организация используемой выходной информации

Выходная информация может быть двух видов, в виде файла формата .txt в котором расположены значения всех Rnorm или в виде итогового Rnorm на последней итерации.

Лист регистрации изменений

Таблица 2 – Лист регистрации изменении руководство программиста

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц) в докум.	№ докумен -та	Входящий № сопроводит. докум. и дата	Подп.	Дата
Изм.	измененных	замененных	новых	Аннулированных					

ПРИЛОЖЕНИЕ Б

Программа, решающая краевую задачу фильтрации двухфазной жидкости в системе LuNA

Описание программы

Листов 10

Новосибирск 2023

Аннотация

В данном документе приведено описание программы, решающей краевую задачу фильтрации двухфазной жидкости в системе LuNA. Пользователь имеет возможность моделировать на практике задачу фильтрации.

Для функционирования программы необходимо наличие установленной системы LuNA с поддержкой MPI процедуры.

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

Содержание

Содержание

1. Общие сведения 59

1.1 Обозначение и наименование программы 59

1.2 Программное обеспечение, необходимое для функционирования программы 59

1.3 Языки программирования 59

2. Функциональное назначение 60

2.1 Назначение программы 60

2.2 Сведения о функциональных ограничениях на применение 60

3. Описание логической структуры 61

3.1 Алгоритм программы 61

4. Используемые технические средства 62

5. Вызов и загрузка 63

6. Входные данные 64

7. Выходные данные 65

1. Общие сведения

1.1 Обозначение и наименование программы

Полное наименование программы — Программа, решающая краевую задачу фильтрации двухфазной жидкости в системе LuNA

1.2 Программное обеспечение, необходимое для функционирования программы

Для функционирования программы необходимо наличие установленной системы LuNA с поддержкой MPI процедуры.

1.3 Языки программирования

Исходными языками программы являются язык LuNA и C++. Средство разработки Notepad++ и Clion.

2. Функциональное назначение

2.1 Назначение программы

Программа, решающая краевую задачу фильтрации двухфазной жидкости в системе LuNA используется для моделировании на практике.

2.2 Сведения о функциональных ограничениях на применение

Программа не предназначена для запуска на платформах, не поддерживающие систему LuNA или её компоненты.

3. Описание логической структуры

3.1 Алгоритм программы

Программа решает краевую задачу фильтрации двухфазной жидкости в трёхмерной области при наличии скважин.

4. Используемые технические средства

Режим работы — консольное приложение.

Для работы необходима систем поддерживающая систему LuNA и все её компоненты.

5. Вызов и загрузка

Запуск программы происходит посредством терминала командой mpirun.

Пример: mpirun —report—bindings —np 4 ~/luna/bin/rts ——worker—threads—count=6 ~/luna_project3/build/libucodes.so —u

В данном случае программа запускается на 4 процессах, каждому из которых доступно максимум 6 потоков.

6. Входные данные

Входные данные, представляет собой файлы формата .txt m_pB, m_pBt, m_pD, pU, pVb, R являющееся соответствующими сеарилизоваными структурами данных. Они представляют начальные данные. Их генерация и сериализация выполняется отдельной специальной программой.

7. Выходные данные

Выходные данные могут быть двух видов, в виде файла формата .txt в котором расположены значения всех Rnorm или в виде итогового Rnorm на последней итерации.

Лист регистрации изменений

Таблица 3 – Лист регистрации изменении описания программы

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц) в докум.	№ докумен -та	Входящий № сопроводит. докум. и дата	Подп.	Дата
Изм.	измененных	замененных	новых	аннулированных					