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

Спирина Виталия Андреевича

Тема работы:

РАЗРАБОТКА ЭФФЕКТИВНОЙ ФРАГМЕНТИРОВАННОЙ ПРОГРАММЫ РЕШЕНИЯ ЗАДАЧИ ФИЛЬТРАЦИИ ДВУХФАЗНОЙ ЖИДКОСТИ

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

Руководитель ВКР
д.т.н., профессор
зав. каф. ПВ ФИТ НГУ
Малышкин В. Э./…….
(ФИО) / (подпись)
«20» мая 2023 г.

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

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

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)

«16» января 2023 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Спирину Виталию Андреевичу, группы 19203
(фамилия, имя, отчество, номер группы)

Тема Разработка эффективной фрагментированной программы решения задачи фильтрации двухфазной жидкости
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 7 ноября 2022г. №0330, скорректирована распоряжением проректора по учебной работе от 16 января 2023г. №0006

Срок сдачи студентом готовой работы: 20 мая 2023 г.

Исходные данные (или цель работы):

разработка эффективной фрагментированной программы решения задачи фильтрации двухфазной жидкости

Структурные части работы:

содержание, введение, обзор родственных работ, анализ решения задачи, проектирование программы, реализация программы, тестирование

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

«16» января 2023 г.

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

«16» января 2023 г.

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

«16» января 2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В области численного моделирования на мультикомпьютерах / суперЭВМ стоит проблема разработки эффективных параллельных программ. Тут и далее эффективность понимается с точки зрения времени выполнения программы, расхода памяти, нагрузки на сеть и других аналогичных свойств. Такие связанные с эффективностью свойства далее будут также называться нефункциональными.

Для обеспечения требуемых нефункциональных свойств программист берет на себя ответственность за: настройку параллельной программы на доступные ресурсы, экономное использование памяти, равномерное распределение вычислительной нагрузки, и т. п. В таком случае эффективность обеспечена (с точки зрения аппаратных характеристик) только для определенных классов вычислительных систем, а именно тех, под которые программист адаптирует программы. Но может возникнуть потребность в использовании других вычислительных систем, либо тех же, но с другим количеством узлов. И в них эффективность работы программы может быть непредсказуемой. Поэтому для каждого нового рассматриваемого вычислителя программисту потребуется написать вручную новые программы. Они будут решать те же задачи, но, при этом, подстраиваться на эффективное исполнение уже на других вычислителях. Повторять такой процесс для каждого нового вычислителя будет трудоемко. Поэтому уместен вопрос обеспечения переносимости параллельных программ, решение которого позволит учитывать различные особенности вычислительных систем и подстраивать под них эффективное выполнение параллельной программы. Обозначенная проблема входит в множество проблем, которые решаются при помощи автоматизации конструирования параллельных программ [1].

Системы, ориентированные на такую автоматизацию, выполняют конструирование эффективных программ по их высокоуровневым спецификациям. От программиста в данном случае не требуется вручную обеспечивать нефункциональные свойства. Но задача такого конструирования, в общем случае, является алгоритмически труднорешаемой, то есть отсутствует удовлетворительное универсальное решение. Поэтому различные системы автоматического конструирования параллельных программ рассматривают лишь частные подходы к обеспечению эффективности. Развитие таких систем с этой точки зрения зачастую происходит при помощи исследования ручной разработки эффективных параллельных программ для частных случаев среди задач численного моделирования. Это позволяет накапливать и систематизировать опыт создания программ и улучшать системные алгоритмы. Также такая разработанная вручную программа может служить эталоном эффективности, с

которым должна будет сравниваться система при конструировании аналогичной программы на вычислительные системы.

Данная работа посвящена такому исследованию для задачи моделирования процесса фильтрации двухфазной жидкости на основе законов сохранения в интегральной форме [2-6]. В качестве системы автоматизации конструирования параллельных программ рассматривается система фрагментированного программирования LuNA [5, 7-8], разрабатываемая в Институте вычислительной математики и математической геофизики (ИВМиМГ) СО РАН. Она представляет интерес с точки зрения последующего улучшения системных алгоритмов. Чтобы иметь возможность произвести такие улучшения, требуется разработать вручную такую эффективную параллельную программу, которая была бы совместима с этой системой. Под совместимостью здесь понимается соответствие программы модели вычислений, которая используется в системе LuNA. Это означает, что высокоуровневая спецификация разработанной вручную программы должна быть общей с той, которая будет использоваться в системе LuNA. Отличие будет состоять в том, что система LuNA будет конструировать программу по спецификации автоматически, а вручную разработанная программа будет обеспечивать нефункциональные свойства непосредственно. Такая высокоуровневая спецификация в системе LuNA называется фрагментированным алгоритмом. А реализующая такой алгоритм программа, сконструированная автоматически или разработанная вручную, называется фрагментированной программой.

Объект исследования – множество параллельных программ решения задачи фильтрации двухфазной жидкости.

Предмет исследования – совместимость программы с системой фрагментированного программирования LuNA.

Целью данной работы ставится разработка эффективной фрагментированной программы решения задачи фильтрации двухфазной жидкости.

Для достижения поставленной цели решаются следующие **задачи**:

1. выполнение анализа решения задачи фильтрации двухфазной жидкости с точки зрения параллельной реализации;
2. выработка требований к фрагментированной программе;
3. разработка схемы эффективной фрагментированной программы;
4. реализация полученной схемы и проведение тестирования.

Методы исследования – анализ родственных работ и существующих решений задачи фильтрации двухфазной жидкости; формирование схемы программы и реализация этой схемы методами параллельного программирования; сравнительный экспериментальный анализ нефункциональных характеристик реализаций.

Научная новизна состоит в разработке схемы эффективной фрагментированной программы решения задачи фильтрации двухфазной жидкости, описывающей взаимосвязь используемых в задаче данных и вызовов процедур и уменьшающей объем коммуникаций.

Практическая ценность заключается в том, что разработанная программа работает достаточно эффективно. Результаты данной работы предполагается использовать в следующих лабораториях ИВМиМГ СО РАН:

1. математических задач химии – для решения задачи фильтрации;
2. синтеза параллельных программ – для усовершенствования системы LuNA.

Теоретическая ценность заключается в исследовании особенностей эффективной параллельной реализации одного из частных случаев среди задач численного моделирования с точки зрения автоматизации конструирования параллельных программ.

Работа состоит из введения, пяти глав, заключения, списка используемой литературы и приложений, среди которых присутствуют руководство программиста и описание программы. В первой главе представлен обзор родственных работ в сфере адаптации программ решения задач численного моделирования для систем с автоматизацией конструирования параллельных программ. Вторая глава содержит анализ решения задачи фильтрации с точки зрения параллельной реализации. Третья глава посвящена описанию технологии фрагментированного программирования, формированию требований и разработке схемы программы. Четвертая глава описывает процесс реализации полученной схемы в виде фрагментированной программы. Пятая глава посвящена тестированию разработанной программы.

1 Обзор родственных работ

Существует множество систем, выполняющих автоматическое конструирование параллельных программ. Эти системы во многом схожи: разработчики при решении проблемы автоматического конструирования выбирают аналогичные подходы, так как они в процессе поиска решения приходят к выводу, что эти подходы являются удовлетворительными. И целью этой главы является рассмотрение этих систем на предмет подходов, которые используют они, а также программисты, для обеспечения эффективности.

1.1 Обзор библиотек и систем автоматизации

Проведем обзор инструментов, с помощью которых может выполняться автоматизация конструирования параллельных программ.

LuNA (Language for Numerical Algorithms) [5, 7-8] – система фрагментированного программирования, разрабатываемая в ИВМиМГ СО РАН. Язык, использующийся в системе, имеет одноименное название и создан для описания сложных численных алгоритмов. Фрагментированный алгоритм (ФА) на языке LuNA представляется множеством фрагментов вычислений (ФВ) и фрагментов данных (ФД), которые можно сравнить с вызовами процедур и переменными. Описание алгоритма разделено с управлением исполнением программы, но есть возможность влиять на исполнение при помощи “рекомендаций” в коде – информации о желаемой схеме исполнения ФА [9].

Эффективность исполнения программ в системе достигается за счет:

- Отсутствия побочных эффектов у ФВ. Это позволяет выполнять их независимо.
- Регуляции загрузки процессоров за счет того, что система видит, какие ФД уже вычислены, а какие нет.
- Оценки объема коммуникаций, а также моментов времени, в которые их нужно произвести, имея информацию о зависимостях ФД и ФВ.

В работе [10] рассматривается фрагментация метода IADE-RB-CG на примере решения одномерного уравнения теплопроводности. Декомпозированные на фрагменты данные соответствуют разбиению всей области моделирования на подобласти одинакового размера. Для сравнения производительностей использовались следующие реализации: MPI программа и две фрагментированные программы на LuNA, одна из которых использует рекомендации, а другая – нет. MPI программа заняла на исполнение меньше всего времени, а LuNA программа без рекомендаций – больше всего времени. Все три реализации сопоставимы при использовании не более 16 узлов (разница составляет где-то 0,01 секунды при общем времени работы меньше 0,1 секунд). При использовании более 16 узлов LuNA программа без

рекомендаций начинает отставать сильнее, а LuNA программа с рекомендациями также остается сопоставимой с MPI реализацией.

Таким образом, рассмотрено, как происходит обеспечение эффективности как со стороны системы, так и со стороны программиста. Система обеспечивает все необходимые нефункциональные характеристики, и на примере приведенной выше работы, это позволяет приблизиться к эффективности MPI программы. Со стороны программиста можно использовать рекомендации, которые позволяют произвести больше оптимизаций.

PLASMA (Parallel Linear Algebra Software for Multi-core Architectures) [11] – библиотека для параллельных реализаций операций линейной алгебры. Она использует PaRSEC в качестве исполнительной системы. В библиотеке вводится понятие tile, что переводится как “плитка”. Оно используется для обозначения части от каких-то данных (например, для матрицы плиткой будет какая-либо подматрица). Основная идея данной библиотеки заключается в трех свойствах:

- Плиточные алгоритмы. Такие алгоритмы работают с матрицами, которые декомпозированы на плитки, каждая из которых помещается в один из уровней кэш-памяти.
- Расположение плиточных данных в памяти. Каждая плитка матрицы или вектора расположена в памяти непрерывно. Это позволяет добиваться оптимизаций программ с учетом работы кэша.
- Динамическое планирование. Идея динамического планирования основана на планировании по готовности данных.

Алгоритмы линейной алгебры представляются в виде ориентированного бесконтурного графа (directed acyclic graph, DAG), где вершины – это задачи, а дуги – это зависимости между задачами.

Приводится сравнение работы данной библиотеки с другими на примере трех алгоритмов: разложение Холецкого, QR- и LU-разложение [12]. Показано, как при использовании библиотеки PLASMA производительность (в Гфлопс/с) выше, чем у других библиотек на всех трех алгоритмах.

В данной библиотеке было рассмотрено, что операции линейной алгебры реализованы достаточно эффективно (по сравнению с другими библиотеками), что позволяет добиваться хорошей производительности с минимальными усилиями.

Charm++ – это task-based система, использующая в качестве основы язык C++ [13]. В ней вводится понятие chare – структурная единица, которая используется для описания программ. Исполнением программы занимается исполнительная система, где происходит автоматическое отображение задач на ресурсы компьютера, благодаря чему достигается

эффективность [14]. Система решает следующие задачи: балансировка нагрузки, обеспечение отказоустойчивости, автоматическая установка контрольных точек и возможность увеличивать или уменьшать количество вычислителей, используемых параллельной программой.

В работе [15] рассматривается, как с использованием Charm++ достигается эффективность для задач молекулярной динамики. Указываются отмеченные выше балансировка нагрузки, отображение на ресурсы с учетом топологии вычислителей, и различные оптимизации. Также Charm++ предоставляет параллельный ввод/вывод. В результате использования различных кластеров отмечается, что удалось добиться эффективности выше при использовании графических ускорителей, чем при использовании новейших (на момент написания указанной статьи) процессоров.

Данная система позволяет добиваться достаточной эффективности с минимальными затратами со стороны программиста.

PaRSEC (Parallel Runtime Scheduling and Execution Control) [16] – система, которая берёт за основу язык C. Также является task-based системой, с вычислениями, происходящими по готовности данных. Она строит DAG путем анализа последовательного алгоритма. Исполнение происходит в исполнительной системе.

В работе [17] рассматривается решение задачи геостатистического моделирования с использованием PaRSEC. Отмечается, что в дополнение к DAG, строится параметризованный граф задач (Parameterized Task Graph, PTG). Исполнительная система отображает задачи на доступные вычислители, учитывая зависимости, представленные в PTG. Отмечается, что системе удалось добиться эффективности выше, чем у реализаций с использованием системы StarPU. Причиной является тот факт, что в системе StarPU нет естественной поддержки вычислений смешанной точности (в отличие от PaRSEC), что влечет увеличение объема коммуникаций.

Как и в уже рассмотренных системах, эффективность в этой библиотеке достигается системой автоматически, имея в виду различные зависимости в программах.

Legion [18] – система, использующая в качестве основы язык C++. В ней вводится понятие region (регион) – совокупность данных. Для него справедливы два свойства:

- данные в регионе являются локальными и независимыми от других регионов;
- для каждого региона существует набор задач, которые его используют.

Эффективность в системе достигается путем автоматического обнаружения независимых задач, а также выявления возможности параллельных вычислений внутри регионов.

В модели, используемой в системе, отмечаются следующие факторы:

- Пользовательская спецификация свойств данных – система предоставляет абстракции для программистов, чтобы учесть различные свойства данных: организацию, разделение на части, привилегии и согласованность.
- Автоматические механизмы – система позволяет, как отмечалось выше, извлекать параллелизм из программ и обеспечивать необходимую передачу данных.
- Отображение, контролируемое пользователем – система позволяет легко описывать отображения приложений на разные типы архитектур с использованием абстракций для описания задач и данных. У программиста есть возможность прямого управления всех деталей того, как приложения отображаются и исполняются.

В работе [19] отмечаются все указанные выше свойства системы. Здесь также сравнивается работа программ с использованием Fortran и Julia, а также программы с использованием системы Legion. Отмечается, что производительность Legion-реализации получилась хуже, чем у Fortran-реализации, но лучше, чем у Julia-реализации.

Система позволяет добиваться достаточной эффективности с минимальными усилиями.

1.2 Итог обзора

Во всех рассмотренных системах выявлены общие подходы. Например, данные, с которыми оперируют системы, структурированы особенным образом, и у каждой системы для них есть собственные названия: фрагменты, плитки, chare, регионы. Также в некоторых системах есть разделение между описанием алгоритма и его исполнением. Это говорит о том, что от программиста требуется написать только высокоуровневую спецификацию, а полноценная программа с нефункциональными свойствами строится автоматически.

Обзор показал, что универсальных решений проблемы нет, поэтому нужно исследовать разные частные подходы.

В ходе обзора были рассмотрены все ключевые системы автоматизации, в том числе система LuNA. Это помогло рассмотреть основные подходы, которые они используют, что дает возможность, с учетом опыта, реализовать совместимую с системой LuNA программу решения задачи фильтрации.

2 Анализ решения задачи фильтрации двухфазной жидкости

Важность решения задачи фильтрации обусловлена тем, что в современных условиях совместная фильтрация несмешивающихся жидкостей является одним из ключевых разделов в подземной гидродинамике, а для получения новых знаний используется математическое моделирование [20].

Под фильтрацией понимают движение жидкости в пористой среде. Пористой называется среда, в которой содержится значительное число пустот, причем размеры пустот малы по сравнению с размерами среды [20]. В рамках данной работы рассматривается задача фильтрации двухфазной (нефть-вода) жидкости в трехмерном пространстве.

2.1 Описание структуры решения задачи

Для решения задачи используется алгоритм, основанный на решении сеточных уравнений для скорости и давления на одном шаге интегрирования по времени. Используемые уравнения представляют собой линейную алгебраическую систему седлового типа [20]. В формуле 1 представлена матрично-векторная система, полученная в статьях [3-4].

$$[B^T D^{-1} B + \varepsilon C] \cdot \bar{u} = \bar{f} \quad (1)$$

Здесь матрица D – это симметричная блочно-диагональная матрица с трехдиагональными блоками, $C = (\bar{e} \cdot \bar{e}^T)$ – симметричная квадратная плотная матрица, матрица B – прямоугольная (количество строк примерно в 3 раза превышает количество столбцов) и разреженная (в каждой строке матрицы по 1-2 элемента и в каждом столбце по 3-6 элементов) [5].

В статье [3] доказывается положительная определенность матрицы $[B^T D^{-1} B + \varepsilon C]$ для любого $\varepsilon > 0$, что дает возможность решать данную систему с использованием итерационного метода сопряженных градиентов. В статье [4] показано, как искать ε, чтобы количество итераций было близко к минимальному. Условием окончания итераций является достижение требуемой точности. На уровне программы, решающей задачу, также важно иметь верхний предел допустимого количества итераций, при превышении которого заканчивается выполнение метода.

Определив структуру метода сопряженных градиентов, перейдем к его реализации в существующей многопоточной реализации. Для реализации параллелизма она использует только библиотеку OpenMP [21]. Далее в этой главе представлены результаты анализа упомянутой реализации.

Отметим, что, на основе тестов и анализа кода, выполнение именно метода сопряженных градиентов занимает большую часть времени. Одна итерация метода

сопряженных градиентов состоит из комбинации векторных и матрично-векторных операций. Среди векторных операций присутствуют: скалярное произведение, вычисление евклидовой нормы, составление линейной комбинации векторов, копирование векторов. Матрично-векторными операциями являются: умножение матрицы на вектор, умножение транспонированной матрицы на вектор и умножение обратной матрицы на вектор.

При увеличении размеров сетки, увеличиваются размеры векторов и матриц. Поэтому важно иметь параллельную реализацию, которая позволит масштабировать вычисления на разное количество вычислительных ресурсов.

2.2 Формирование и анализ графа зависимостей данных

Метод сопряженных градиентов, представленный в многопоточной реализации решения задачи, будет удобно (для дальнейшего анализа с точки зрения параллельной реализации) представить в виде ориентированного графа зависимостей данных и вызовов процедур. Вершинами будут выступать данные и вызовы процедур, а дугами – связь данных с процедурами (являются они входными или выходными).

Но прежде, чем работать с графом, нужно выработать условные обозначения.

2.2.1 Создание условных обозначений

Графическая нотация была выработана в процессе работы, и служит для отображения на графе существенных свойств с точки зрения параллельной реализации. Она также представляет практический интерес для использования в других задачах.

Основу графа составляют данные и вызовы процедур. Данные будут представляться в виде кругов, а вызовы процедур – в виде прямоугольников. Также имеет смысл ввести обозначение для констант; они будут представляться кругами с пунктирной обводкой. Входные данные будут представляться как связь стрелкой от данных к вызову процедуры, а выходные данные – стрелкой от вызова процедуры к данным.

Дополнительно для удобства была введена цветовая гамма для отличия типов данных. Для черно-белой печати также добавлены определенные обозначения. В итоге получено следующее:

- числа с плавающей запятой обозначаются зеленым цветом, а также точкой над названием переменной / значением числа;
- вектора обозначаются желтым цветом и чертой над названием переменной / значением вектора;
- матрицы обозначаются голубым цветом и дополнительно обрамляются фигурными скобками.

Также рассмотрена возможность отличать размеры векторов и матриц: используется разная толщины обводки у векторов, а также разная толщина фигурных скобок у матриц.

Результат выработанных условных обозначений представлен на рисунке 1 ниже:

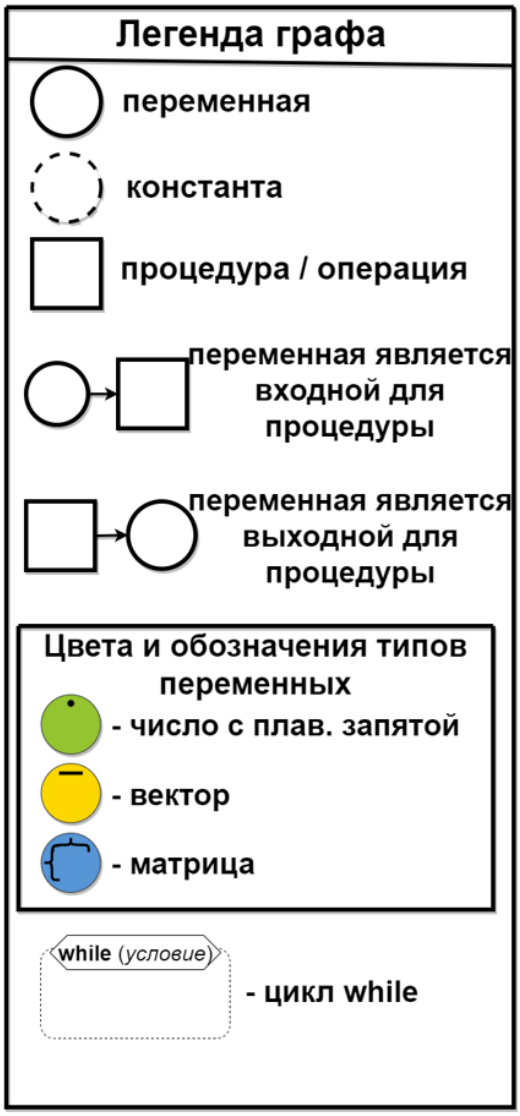

Рисунок 1 – Легенда графа

2.2.2 Построение графа

На основе проведенного анализа метода сопряженных градиентов в многопоточной реализации решения задачи было разработано представление графа зависимостей между переменными и вызовами процедур с использованием условных обозначений, приведенных в предыдущем пункте. Часть полученного графа (представляющая вычисление формулы 1) показана на рисунке 2 ниже. Полный граф представлен на рисунке А.1 в приложении А.

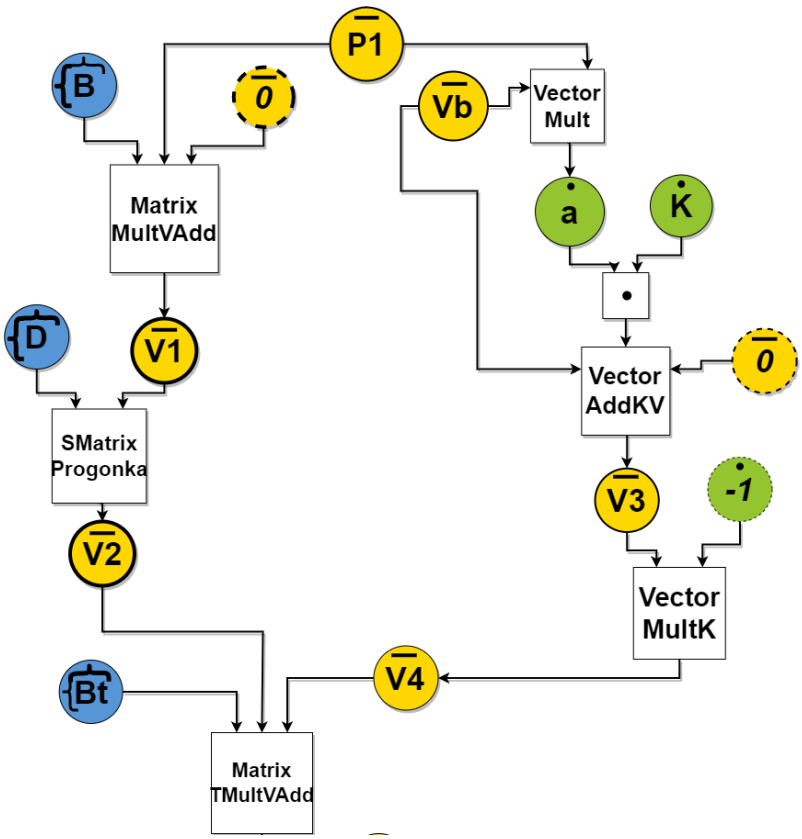

Рисунок 2 – Часть графа метода сопряженных градиентов

Проведем сравнение части графа, представленной на рисунке выше, с кодом в многопоточной реализации.

Сначала рассмотрим правую ветку представленного графа. Первым идет вызов процедуры VectorMult (скалярное произведение) с входными векторами *Vb* и *P1*. Далее, полученное в результате число умножается на *K*, и результат идет в вызов процедуры VectorAddKV (умножение вектора на число и добавление результата к другому вектору). В коде это реализовано так, как представлено в листинге 1 ниже. Содержимое этого и всех последующих листингов, представляющих код, используют синтаксис языка C++.

Листинг 1 – Код, соответствующий правой ветви рисунка 2

```
1  pRes->SetNull();
2  double a = m_K * Basics::Vector::Mult(bParallel, m_pVb, pV);
3  pRes->AddKV(bParallel, a, m, m_pVb);
4  pRes->MultK(bParallel, -1.0);
```

Стоит отметить, что в листинге 1 вектор `pV` представляет вектор *P1* на графе. Также в начале кода вектор `pRes` инициализируется нулями, а на графе это отмечено нулевым вектором при вызове метода `VectorAddKV`.

Для левой ветви графа на рисунке 2 код выглядит так, как представлено в листинге 2 ниже.

Листинг 2 – Код, соответствующий левой ветви рисунка 2

```
1  V1.SetNull(bParallel);
2  m_pB->MultVAdd(bParallel, pV, &V1, Progress);
3  m_pD->Invert(bParallel, &V1, &V2);
4  m_pBt->MultVAdd(bParallel, &V2, pRes, Progress);
```

Также, как и для предыдущего листинга, имеет место следующее: нулевой вектор, который является входным для вызова MatrixMultVAdd на графе, соответствует инициализации нулями вектора `V1` в листинге 2 (строка 1).

Полученный граф теперь требуется проанализировать с точки зрения его использования в параллельной программе.

2.2.3 Анализ графа с точки зрения параллельной реализации

Для каждой процедуры на графе был выполнен анализ и составлены их подробные описания. Прежде, чем перейти к их описаниям, требуется рассмотреть некоторые определения.

Будем говорить, что входные векторы процедуры *покомпонентно независимы*, если для получения i-го элемента выходного вектора требуется произвести операцию только над i-мы элементами входных векторов. В англоязычной литературе такие процедуры называются процедурами типа map [22]. Для примера рассмотрим операцию умножения вектора на скаляр: $\bar{w} = k * \bar{v}$. Если представить эту формулу в виде $w_i = k * v_i \quad \forall i \in \{0, \dots, n-1\}$, то отсюда явно видно выполнение указанного определения.

Также рассмотрим понятие процедуры *редукционного типа* [22]. В качестве примера рассмотрим скалярное произведение векторов: $a = \bar{v} \cdot \bar{w}$. Из представления этой операции в виде $a = \sum_{i=0}^{n-1} v_i w_i$ видно, что a получается в результате применения редукционной операции к множеству частичных результатов $\{v_i w_i\}_{i=0}^{n-1}$.

Теперь перейдем к описанию процедур. Каждое название состоит из двух частей. Первая часть составляет класс (тип) данных – вектор (Vector), обычная матрица (Matrix) или симметричная матрица (SMatrix). Вторая часть – наименование непосредственно вычисляемой процедуры. Для каждой процедуры на графе, работающей только с векторами, выявлен тип зависимости входных данных. Эта информация будет полезна далее при составлении схемы разрабатываемой параллельной программы.

Таблица 1 – Описание векторных процедур на графе

Название процедуры на графе	Описание процедуры	Математическая формула	Тип процедуры
Vector Mult	Скалярное произведение двух векторов	$\bar{v} \cdot \bar{w} = \sum_{i=0}^{n-1} v_i w_i$	Редукционный тип
Vector AddKV	Добавление одного вектора, умноженного на коэффициент, к другому	$\overline{v_2} := \overline{v_2} + k * \overline{v_1}$	Покомпонентно независимые входные векторы
Vector MultK	Умножение вектора на константу	$\overline{v_1} := k * \overline{v_1}$	Покомпонентно независимые входные векторы
Vector Set	Копирование данных из одного вектора в другой	$\overline{v_2} := \overline{v_1}$	Покомпонентно независимые входные векторы
Vector GetNorm	Вычислить евклидову норму вектора	$\|\|v\|\|_2 = \left(\sum_{i=0}^{n-1} {v_i}^2\right)^{\frac{1}{2}}$	Редукционный тип
Vector SetLin	Создать линейную комбинацию двух векторов	$w := \alpha_1 v_1 + \alpha_2 v_2$	Покомпонентно независимые входные векторы

Таблица 2 – Описание матрично-векторных процедур на графе

Название процедуры на графе	Описание процедуры	Математическая формула
Matrix MultVAdd	Умножение матрицы на вектор, и добавление результата к другому вектору	$\overline{v_2} := \overline{v_2} + B\overline{v_1}$

Окончание таблицы 2

Название процедуры на графе	Описание процедуры	Математическая формула
Matrix TMultVAdd	Умножение транспонированной матрицы на вектор, и добавление результата к другому вектору	$\overline{v_2} := \overline{v_2} + B^T\overline{v_1}$
SMatrix Progonka	Умножение обратной матрицы на вектор (при помощи решения СЛАУ), и добавление результата к другому вектору	$\overline{v_2} := \overline{v_2} + \overline{w}$ $\overline{w} = D^{-1}\overline{v_1} \Leftrightarrow D\overline{w} = \overline{v_1}$

Полученные результаты помогут далее определить, как можно распределить вычисления. Это рассматривается в следующей главе.

2.3 Итог анализа

Главными результатами главы стали создание графической нотации, представление графа с ее использованием, анализ графа с точки зрения параллельной реализации и выявление основных свойств каждой процедуры на графе.

Таким образом, требуемый анализ решения задачи с точки зрения параллельной реализации был выполнен. С его помощью теперь может строится эффективная параллельная программа.

3 Проектирование фрагментированной программы

3.1 Технология фрагментированного программирования

Прежде, чем приступить к проектированию программы, есть потребность в введении терминов, связанных с фрагментированным программированием. Данные термины соотносятся с системой LuNA, а для других систем автоматизации конструирования параллельных программ существуют аналогичные термины, что было рассмотрено в первой главе.

В этой системе определяются такие понятия как фрагмент данных, фрагмент кода и фрагмент вычислений. *Фрагмент кода* представляет собой некоторый модуль, который реализует вычисления (процедура). *Фрагмент вычислений* – это непосредственное использование фрагмента кода в ходе работы программы (вызов процедуры). *Фрагменты данных* являются единицами, с которыми работают фрагменты вычислений (данные).

В данной системе пользователь реализует алгоритм в виде информационно- зависимых фрагментов данных и фрагментов вычислений. Как говорилось ранее, такое описание алгоритма является высокоуровневой спецификацией системы, и называется *фрагментированным алгоритмом*. В нем отсутствуют: отображения на ресурсы, императивное управление и т. п. *Фрагментированная программа* реализует фрагментированный алгоритм, но уже имея в виду обозначенные выше свойства.

3.2 Требования к фрагментированной программе

Разработка фрагментированной программы может производиться как автоматизировано, так и вручную. Во втором случае на программиста ложится работа по обеспечению нефункциональных свойств, а фрагментированный алгоритм, в котором содержится множество фрагментов данных и вычислений, является общей для обоих способов конструирования фрагментированных программ. Поэтому важно обеспечить совместимый фрагментированный алгоритм, то есть такой, который можно также использовать в системе LuNA. Это, в том числе, возложит определенные требования на фрагментированную программу. На основе анализа работ, проведенного в первой главе, а также анализа, проведенного во второй главе, были составлены приведенные ниже требования к фрагментированной программе. Они разделены на две группы.

Первой группой являются требования к фрагментированной программе с прицелом на LuNA:

1. Процедуры, выполняющие вычисления над фрагментами данных, не должны иметь побочных эффектов: обращаться к глобальным переменным, изменять входные фрагменты.
2. Использовать примитивные типы данных языка программирования C++.

Второй группой являются требования, связанные с нефункциональными характеристиками:

1. Программа должна решать задачу на разном количестве процессов и потоков так же, как и другие реализации.
2. Размеры фрагментов для одного и того же вектора или матрицы на разных процессах должны отличаться не более чем на 1.
3. Программа не должна делать лишних коммуникаций при передаче фрагментов между процессами. Например, если в конце одной операции фрагменты вектора собираются вместе, а потом в начале следующей операции вектор опять фрагментируется, то эти две коммуникации можно считать лишними.
4. Полученная программа должна работать достаточно эффективно.
5. Эффективность распараллеливания должна быть удовлетворительной. Это рассматривается в главе 5.

После предъявления указанных требований перейдем к процессу проектирования программы.

3.3 Разработка схемы фрагментированной программы

Аналогично тому, как во второй главе был создан граф метода сопряженных градиентов, стоит также разработать схему фрагментированного метода сопряженных градиентов. Проведем дополнительный анализ, чтобы выявить ключевые моменты, которые должны быть представлены на схеме.

3.3.1 Рассмотрение фрагментации каждой процедуры

В пункте 2.3 был проведен анализ процедур с точки зрения их типа (покомпонентно независимые входные векторы или редукционный тип). Теперь рассмотрим, как эти результаты соотносятся с фрагментацией данных.

Входные векторы процедур VectorAddKV, VectorMultK, VectorSet и VectorSetLin являются покомпонентно независимыми, то есть, если разделить векторы на фрагменты, то эти процедуры можно применить к фрагментам независимо на каждом процессе, без коммуникаций.

Процедуры VectorMult и VectorGetNorm имеют редукционный тип, то есть, если входные данные разделены на фрагменты, то для получения полного выходного результата этих процедур, требуется выполнить редукционные операции с использованием коммуникаций.

Теперь рассмотрим матрично-векторные операции.

Операция MatrixMultVAdd умножения матрицы на вектор математически представляется следующей формулой: $f = Ax \Leftrightarrow f_j = \sum_{k=1}^{n} A_{jk} x_k$. Также на рисунке 3 показано ее визуальное представление с акцентированием внимания (при помощи цвета) на зависимость между строками матрицы и компонентами выходного вектора.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n-1} & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \dots \\ f_m \end{bmatrix} = \begin{bmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,n}x_n \\ a_{2,1}x_1 + a_{2,2}x_2 + \dots + a_{2,n}x_n \\ \dots \\ a_{m,1}x_1 + a_{m,2}x_2 + \dots + a_{m,n}x_n \end{bmatrix}$$

Рисунок 3 – Наглядное представление умножения матрицы на вектор

Для получения *j*-ого элемента результирующего вектора достаточно рассмотреть *j*-ую строку матрицы *A*. Тогда матрицу можно разбить на фрагменты, каждый из которых будет содержать какое-то множество строк матрицы *A*. Но входной вектор *x* придется хранить полностью, как один фрагмент. Это означает, что перед выполнением умножения придется собирать входной вектор с помощью коммуникаций, если он был разбит на фрагменты.

Операция MatrixTMultVAdd умножения транспонированной матрицы на вектор во многом схожа с предыдущей операцией, но есть важное отличие в обходе матрицы. Так как матрица разреженная, то удобно будет ее хранить в памяти в формате CSR или CSC [23], и тут важно, является ли матрица транспонированной или нет. Математически умножение представляется следующей формулой: $f = A^T x \Leftrightarrow f_j = \sum_{k=1}^{n} A_{kj} x_k$. А ее визуальное представление с аналогичным акцентированием внимания на зависимость между столбцами матрицы и компонентами выходного вектора содержится в рисунке 4.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n-1} & a_{m,n} \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \dots \\ f_{n-1} \\ f_n \end{bmatrix} = \begin{bmatrix} a_{1,1}x_1 + a_{2,1}x_2 + \dots + a_{m,1}x_m \\ a_{1,2}x_1 + a_{2,2}x_2 + \dots + a_{m,2}x_m \\ \dots \\ a_{1,n-1}x_1 + a_{2,n-1}x_2 + \dots + a_{m,n-1}x_m \\ a_{1,n}x_1 + a_{2,n}x_2 + \dots + a_{m,n}x_m \end{bmatrix}$$

Рисунок 4 – Наглядное представление умножения транспонированной матрицы на вектор

Теперь *j*-ый элемент результирующего вектора зависит не от *j*-ой строки, а от *j*-ого столбца матрицы. То есть, один из способов разбиения матрицы на фрагменты – это хранить в каждом фрагменте какое-то множество столбцов матрицы. Но, как и в предыдущем случае, входной вектор придется хранить полностью, что также потребует коммуникаций для сбора фрагментов.

Другой подход, которой можно использовать — это разбить матрицу по строкам. Это позволит также разбить входной вектор на фрагменты с соответствующими размерами. Но в этом случае выполнение операции над каждым фрагментом в качестве результата даст частичный вектор, и для получения полного результата требуется выполнить их редукцию. Схематично это представлено на рисунке 5 ниже.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} f_{1,1} \\ f_{2,1} \\ \dots \\ f_{n-1,1} \\ f_{n,1} \end{bmatrix} = \begin{bmatrix} a_{1,1}x_1 + a_{2,1}x_2 \\ a_{1,2}x_1 + a_{2,2}x_2 \\ \dots \\ a_{1,n-1}x_1 + a_{2,n-1}x_2 \\ a_{1,n}x_1 + a_{2,n}x_2 \end{bmatrix}$$

$$\dots \qquad \dots \qquad \dots$$

$$\begin{bmatrix} a_{m-1,1} & a_{m-1,2} & \dots & a_{m-1,n-1} & a_{m-1,n} \\ a_{m,1} & a_{m,2} & \dots & a_{m,n-1} & a_{m,n} \end{bmatrix}^T \begin{bmatrix} x_{m-1} \\ x_m \end{bmatrix} = \begin{bmatrix} f_{1,k} \\ f_{2,k} \\ \dots \\ f_{n-1,k} \\ f_{n,k} \end{bmatrix} = \begin{bmatrix} a_{m-1,1}x_1 + a_{m,1}x_2 \\ a_{m-1,2}x_1 + a_{m,2}x_2 \\ \dots \\ a_{m-1,n-1}x_1 + a_{m,n-1}x_2 \\ a_{m-1,n}x_1 + a_{m,n}x_2 \end{bmatrix}$$

$$\begin{bmatrix} f_1 \\ f_2 \\ \dots \\ f_{n-1} \\ f_n \end{bmatrix} = \begin{bmatrix} f_{1,1} \\ f_{2,1} \\ \dots \\ f_{n-1,1} \\ f_{n,1} \end{bmatrix} + \dots + \begin{bmatrix} f_{1,k} \\ f_{2,k} \\ \dots \\ f_{n-1,k} \\ f_{n,k} \end{bmatrix}$$

Рисунок 5 – Наглядное представление фрагментации операции умножения транспонированной матрицы на вектор при разбиении матрицы по строкам

Какой из способов выбрать зависит от способа расположения данных в памяти, что рассматривается далее в пункте 3.4.

Последняя матричная операция SMatrixProgonka выполняет умножение обратной матрицы на вектор. В данной задаче матрица является блочно-трехдиагональной, что позволяет выполнять умножение при помощи решения СЛАУ методом прогонки [24]. Его можно применить к каждому трехдиагональному блоку матрицы независимо. Это

единственный способ произвести вычисления параллельно, так как сам метод прогонки является последовательным.

В результате рассмотрены способы распределить вычисления, и теперь перейдем к построению схемы фрагментированной программы.

3.3.2 Итоговая схема, представленная в виде графа

Полученная схема фрагментированной программы представлена на рисунке 6 ниже.

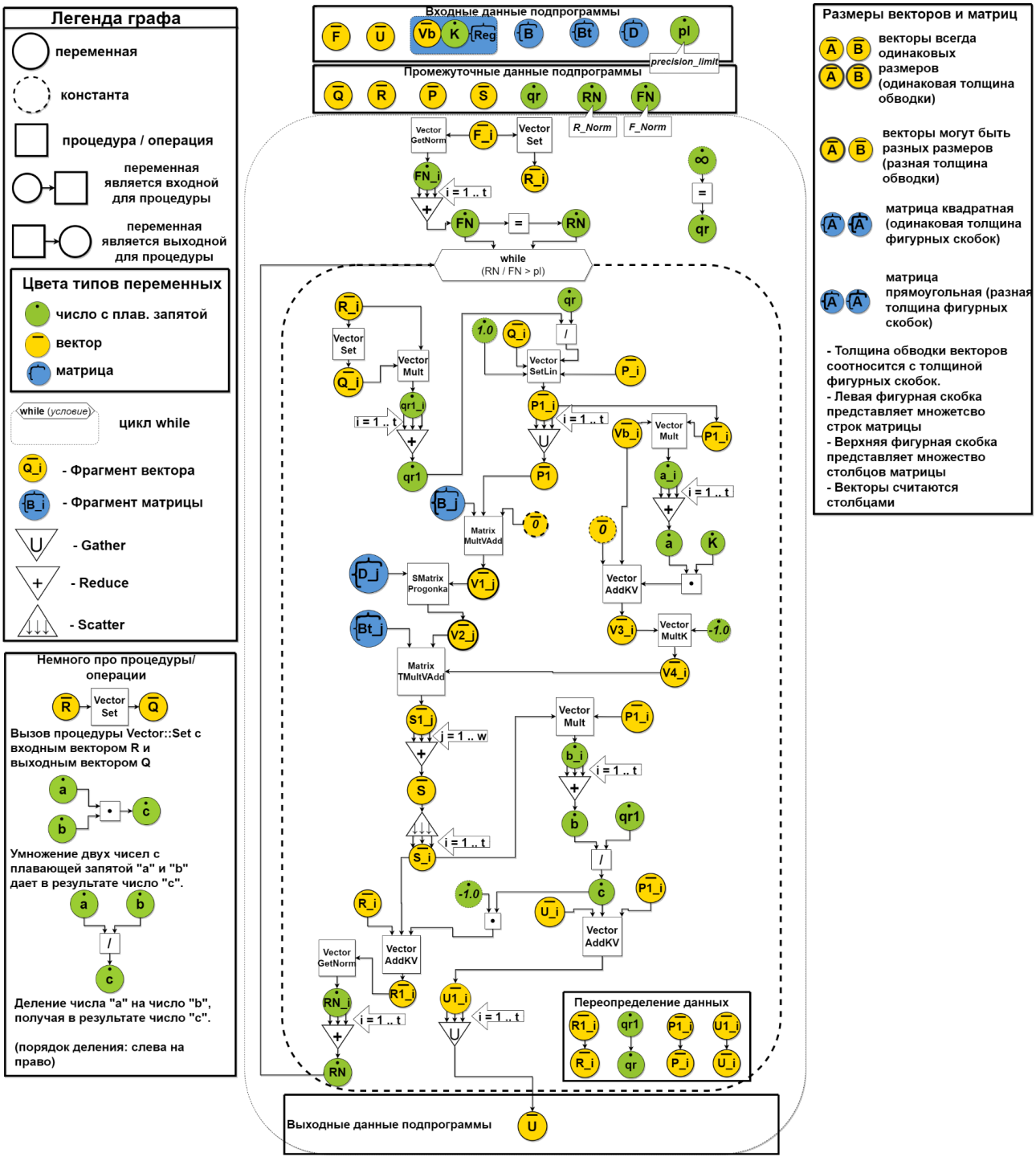

Рисунок 6 – Схема фрагментированного метода сопряженных градиентов

Рассмотрим основные моменты на этой схеме. По сравнению с графом во 2-й главе, здесь все данные были декомпозированы на фрагменты. Это подчеркивается наличием у данных индекса *i* или *j*, причем, в общем случае, индекс *i* используется для фрагментов одного размера, а индекс *j* – для фрагментов другого размера.

Операции над данными выполняются следующим образом. Пусть, например, задан некоторый вектор *R* из задачи. Он представляется множеством фрагментов R_1, R_2, …, R_m. Если на графе в операции участвует "*R_i*", то это означает, что эта операция выполняется над всеми фрагментами вектора R.

Также, так как появилось разделение данных на фрагменты, должна быть возможность производить обмен и редукцию. Для этого существуют операции типа gather, scatter и reduce [22, 25].

В итоге была составлена схема фрагментированной программы. Далее, перед тем как реализовывать полученную схему, нужно рассмотреть структуру программы, то есть ее ключевые моменты, входные и выходные данные, и т. д.

3.4 Структура программы

3.4.1 Архитектура

За основу архитектуры программы взята архитектура многопоточной реализации. Как говорилось ранее, для хранения матриц удобно использовать разреженный тип. В многопоточной реализации используется тип CSR [23], который также будет использоваться и в фрагментированной программе.

3.4.2 Входные и выходные данные программы

На вход программе будет подаваться файл с описанием структуры задачи, а также описанием того, на скольких потоках программа будет запускаться. Пример описания файла представлен в листинге 3 ниже.

Листинг 3 – Входные данные задачи фильтрации

```
 1  [Geom]
 2  Limits XMin 0.0 XMax 50.0 YMin 0.0 YMax 50.0 ZMin 0.0 ZMax 10.0
 3  Por m 0.375 k0 3.06e-12
 4  Fracture m 0.05 k0 1.53e-11
 5  Oil mu 9.28e-4 rho 0.8e+3
 6  Water mu 1.15e-4 rho 1.0e+3
 7  GeomData sigmacosteta 0.0 gamma 1e-10 //1e-8 // 0.2?
 8
 9  [Mesh]
10  Regular nX 200 nY 200 nZ 25
11
12  [InOut]
13  Well Q 5e-4 X1 10.0 Y1 25.0 Z1 5.01 X2 10.0 Y2 25.0 Z2 10.0 Q_ToFracture
```

```
    0.001
14  Well Q -1 X1 40.0 Y1 25.0 Z1 0 X2 40.0 Y2 25.0 Z2 4.99 Q_ToFracture -1
15
16  [Run]
17  Params nTimeSteps 1 TimeStep 2000 nMaxIters 100 Prec 1e-3 nInnerSteps 1
18  Parallel bParallel true nThreads 1
19  Options bNeedGravity false Type Pors //AndFractures
20
21  [Analyse]
22  FlatXY Value S_Por FileName s_flatxy_t=1e4.dat Time 1e4 Z 4.99
23  FlatXY Value Psi_Por FileName psi_flatxy_t=1e4.dat Time 1e4 Z 4.99
24  FlatXY Value S_Fracture FileName s_f_flatxy_t=1e4.dat Time 1e4 Z 4.99
25  FlatXY Value Psi_Fracture FileName psi_f_flatxy_t=1e4.dat Time 1e4 Z 4.99
26  FlatXY Value S_PorAndFracture FileName s_p+f_flatxy_t=1e4.dat Time 1e4 Z
    4.99
27  FlatXY Value Q_PorAndFracture FileName q_flatxy_t=1e4.dat Time 1e4 Z 4.99
28  FlatXY Value D_Por FileName d_flatxy_t=1e4.dat Time 1e4 Z 4.99
29  FlatXY Value MB_Por FileName mb_flatxy_t=1e4.dat Time 1e4 Z 4.99
30  FlatXY Value MB_Fracture FileName mb_f_flatxy_t=1e4.dat Time 1e4 Z 4.99
31  FlatXY Value K_Water_Por FileName kw_flatxy_t=1e4.dat Time 1e4 Z 4.99
32  FlatXY Value K_Oil_Por FileName ko_flatxy_t=1e4.dat Time 1e4 Z 4.99
33  FlatXY Value K_Por FileName k_flatxy_t=1e4.dat Time 1e4 Z 4.99
34  FlatXY Value Vx_Por FileName vx_flatxy_t=1e4.dat Time 1e4 Z 4.99
35  FlatXY Value Vy_Por FileName vy_flatxy_t=1e4.dat Time 1e4 Z 4.99
```

В качестве выходных данных программа будет генерировать данные, связанные с задачей, на каждой временной итерации, что представлено в листинге 4 ниже.

Листинг 4 – Выходные данные задачи на каждой временной итерации

```
 1  Инициализация...
 2  Решение...
 3  Основной цикл:
 4  0 0
 5  1 2000
 6    QCell_Por=-2.94264e-21
 7    QWell_Por=-4.06576e-20
 8    QCell_Water_Por=0.000477273
 9    QWell_Water_Por=-0.000477273
10    Q_Por=-4.36002e-20
11    Q_Water_Por=-5.42101e-19
```

Также в результате работы программы будут сгенерированы .dat файлы, содержащие данные о моделировании. Список этих файлов совпадает с тем, что представлен в секции “Analyse” листинга 3 (начинается со строки 21). Содержимое файлов состоит из 3-х столбцов: значения по осям y, x и значение функции в этих точках. Эти .dat файлы можно визуализировать. Пример визуализации представлен на рисунке 7 ниже.

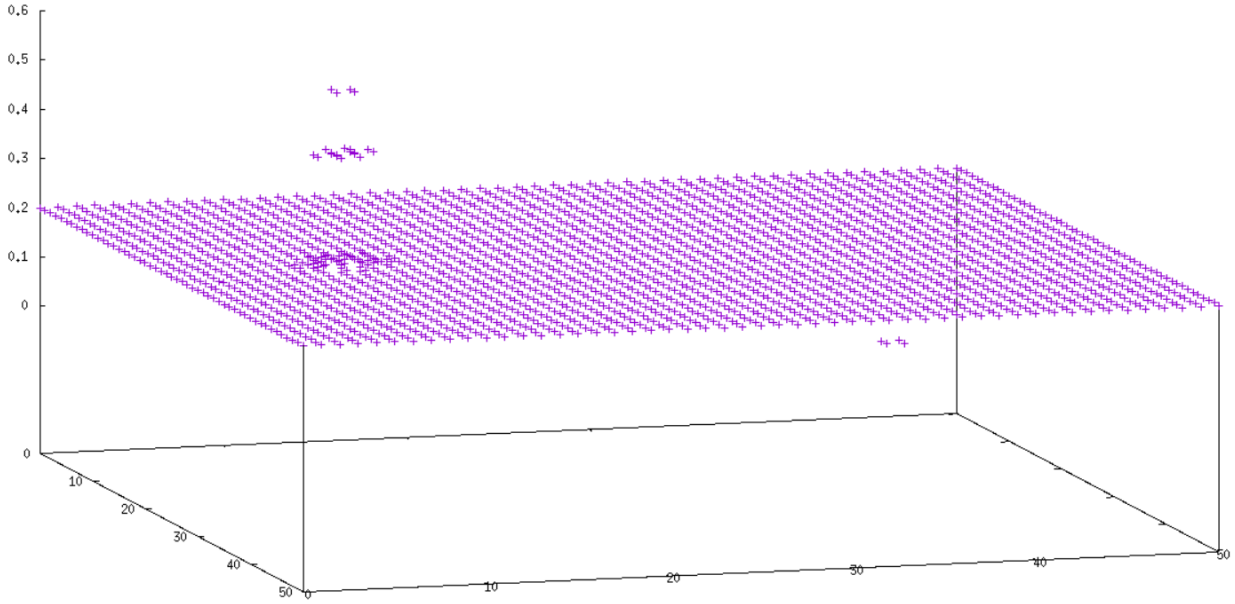

Рисунок 7 – Пример визуализации .dat файла при помощи утилиты gnuplot

3.5 Итог проектирования

В результате была разработана схема фрагментированного метода сопряженных градиентов, рассмотрены способы составления параллельных версий различных процедур и структура программы. Этого достаточно, чтобы приступить к реализации схемы.

4 Реализация фрагментированной программы

Реализация производилась на языке C++ с использованием MPI [26] для запуска и коммуникаций процессов, и библиотеки OpenMP для управления параллелизмом на уровне потоков.

4.1 Реализация метода сопряженных градиентов

4.1.1 Реализация схемы

При реализации схемы все векторы, представленные в виде класса C++ в многопоточной реализации, были преобразованы в массивы, использующие примитивные типы данных.

Далее, векторы были разрезаны на фрагменты и распределены между процессами. Разрезание производилось следующим образом: каждый процесс получал количество элементов, равное частному от деления размера всего вектора на количество процессов. При наличии остатка, он распределяется между процессами (первым k процессам отдается ещё по 1 элементу, где k — величина остатка). Стоит отметить, что содержимое всех фрагментов в процессах расположено в памяти непрерывно.

Матрицы было решено не разрезать в силу следующего: к тому моменту, когда начинает работать параллельный метод сопряженных градиентов, все матрицы уже сформированы и находятся на каждом процессе. Обход матриц, тем не менее, производится только по нужной определенному процессу части.

Операции типа gather, scatter и reduce уже эффективно реализованы в библиотеке MPI в виде процедур MPI_Allgatherv, MPI_Allreduce и MPI_Scatterv.

Структурно далее содержимое пункта 4.1 строится следующим образом: в названии подпункта находится название матрично-векторной операции, а внутри подпункта находятся детали их реализаций.

4.1.2 Операция умножения матрицы на вектор

Данная операция является одной из простых с точки зрения ее параллельной реализации. Ее код представлен в листинге 5 ниже.

Листинг 5 – Код процедуры умножения матрицы на вектор

```
1  #pragma omp parallel for schedule(static)
2  for (intptr_t iRow = iRowStart; iRow < iRowEnd; iRow++) {
3     intptr_t iElemStart = Struct.GetFirstRowElem(iRow);
4     intptr_t iElemEnd = Struct.GetFirstRowElem(iRow + 1);
5     for (intptr_t iElem = iElemStart; iElem < iElemEnd; iElem++) {
```

```
6           intptr_t iCol = Struct.GetElemCol(iElem);
7           pRes[iRow - iRowStart] += pB_array[iElem] * pV[iCol];
8     }
9  }
```

Внешний цикл for (строка 2) служит для обхода строк матрицы. Поскольку матрицы должны быть разрезаны на несколько фрагментов, но в программе так такового разрезания нет, обход происходит только по нужной процессу части (в диапазоне [*iRowStart, iRowEnd - 1*]). Также, директива "*#pragma omp*" (строка 1) говорит о том, что итерации этого цикла будут распределены между несколькими потоками.

4.1.3 Операция умножения транспонированной матрицы на вектор

Для реализации операции умножения транспонированной матриц на вектор был использован подход разбиения матрицы по строкам, что было описано в предыдущей главе. Сначала, как представлено в листинге 6 ниже, вычисляется частичный результирующий вектор (строка 8).

Листинг 6 – Код процедуры умножения транспонированной матрицы на вектор

```
1  intptr_t iRowStart = displs_n[mpi_rank];
2  intptr_t iRowEnd = displs_n[mpi_rank] + counts_n[mpi_rank];
3  intptr_t iElem = Struct.GetFirstRowElem(iRowStart);
4  for (intptr_t iRow = iRowStart; iRow < iRowEnd; iRow++) {
5    intptr_t iElemEnd = Struct.GetFirstRowElem(iRow + 1);
6    for (; iElem < iElemEnd; iElem++) {
7          intptr_t iCol = Struct.GetElemCol(iElem);
8          pRes_full[iCol] += pBt_array[iElem] * pV[iRow - iRowStart];
9    }
10 }
11 double *pRes1 = new double[counts_m[mpi_rank]];
12 MPI_Reduce_scatter(pRes_full, pRes1, counts_m, MPI_DOUBLE, MPI_SUM,
   MPI_COMM_WORLD);
```

Затем, используя процедуру MPI_Reduce_scatter библиотеки MPI, все частичные векторы суммируются (reduce), затем полученный результат распределяется между процессами (scatter).

В отличие от операции умножения обычной матрицы на вектор, здесь нет распределения итераций по потокам. Если попытаться добавить, например, директиву "*#pragma omp*" перед внешним циклом (строка 4), как это было сделано для предыдущей операции, то тогда потоки могут писать в одну и ту же память: для разных строк *iRow* (строка 4) значение *iCol* (строка 7) может совпасть, и возникает проблема в строке 8. Это можно решить добавлением взаимного исключения, но это сделает производительность данной процедуры только хуже.

Несмотря на описанную проблему, есть другой подход распределения вычислений по потокам. Его реализация представлена в листинге 7 ниже.

Листинг 7 – Код процедуры умножения транспонированной матрицы на вектор с использованием потоков

```
 1  double *pRes_full = new double[size_m]();
 2  double *localPart = new double[nThreads * size_m]();
 3  intptr_t iRowStart = displs_n[mpi_rank];
 4  intptr_t iRowEnd = displs_n[mpi_rank] + counts_n[mpi_rank];
 5  #pragma omp parallel
 6 {
 7    int iThread = omp_get_thread_num();
 8    int offset = iThread * size_m;
 9    #pragma omp for schedule(static)
10    for (intptr_t iRow = iRowStart; iRow < iRowEnd; iRow++) {
11          double vectorElem = pV[iRow - iRowStart];
12          intptr_t iElemStart = Struct.GetFirstRowElem(iRow);
13          intptr_t iElemEnd = Struct.GetFirstRowElem(iRow + 1);
14          for (intptr_t iElem = iElemStart; iElem < iElemEnd; iElem++) {
15                intptr_t iCol = Struct.GetElemCol(iElem);
16                localPart[offset + iCol] += pBt_array[iElem] * vectorElem;
17          }
18    }
19    #pragma omp for schedule(static)
20    for (intptr_t i = 0; i < size_m; i++) {
21          for (int t = 0; t < nThreads; t++) {
22                pRes_full[i] += localPart[t * size_m + i];
23          }
24    }
25  }
26  double *pRes1 = new double[counts_m[mpi_rank]];
27  MPI_Reduce_scatter(pRes_full, pRes1, counts_m, MPI_DOUBLE, MPI_SUM,
    MPI_COMM_WORLD);
```

Решение состоит в следующем: требуется создать несколько копий выходного вектора. На уровне программы это реализовано как матрица (*localPart*, строка 2) размера количество потоков (*nThreads*) на размер выходного вектора (*size_m*). Каждый поток пишет только в определенную строку матрицы, которая соответствует его идентификатору. Тогда потоки могут независимо друг от друга модифицировать данную матрицу, то есть, это можно сделать параллельно.

Следующий шаг – просуммировать строки полученной матрицы. Это также можно сделать параллельно: столбцы полученной матрицы распределяются между потоками, и каждый поток суммирует вдоль столбцов.

Схематично работа алгоритма представлена на рисунке 8 ниже.

Размеры матрицы: mxn, где m - количество потоков. n -размер выходного вектора

Шаг 1: вычисление частичных векторов:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n-1} & a_{m,n} \end{bmatrix}$$

← сюда пишет поток 0
← сюда пишет поток 1
← сюда пишет поток m-1

Шаг 2: суммирование частичных векторов:

столбцы суммирует поток 0 ↓ … столбцы суммирует поток m-1 ↓

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n-1} & a_{m,n} \end{bmatrix}$$

$$\Downarrow \quad \Downarrow \quad \dots \quad \Downarrow \quad \Downarrow$$

$$\begin{bmatrix} f_1 & f_2 & \dots & f_{n-1} & f_n \end{bmatrix}$$

Рисунок 8 – Схема работы умножения транспонированной матрицы на вектор с использованием потоков

4.1.4 Операция умножения обратной матрицы на вектор

Для выполнения операции умножения обратной матрицы на вектор используется метод прогонки [24]. Как описано в пункте 3.3.1, для параллельного выполнения данной операции требуется разбить матрицу на несколько диагональных блоков. Выполняется это вызовом процедуры, представленным в листинге 8 ниже.

Листинг 8 – Разбиение матрицы на диагональные блоки для метода прогонки

```
1  SizedArray<intptr_t, int> *pD_iStartIndexes =
2           pD->GetMatrix()->InitForParallel(nThreads * mpi_size);
```

Полученный объект является массивом индексов, где каждый индекс говорит, где начинается тот или иной диагональный блок. Таких блоков много, но достаточно выбрать количество, равное умножению количества потоков на количество процессов. Каждый процесс, обладая каким-то множеством блоков, распределяет их между своими потоками.

Далее, применяется метод прогонки к каждому блоку, что представлено в листинге 9 ниже.

Листинг 9 – Распределение вычислений для метода прогонки

```
1  #pragma omp parallel for schedule(static)
2  for (int i = mpi_rank * nThreads; i < (mpi_rank + 1) * nThreads; i++) {
3     Progonka_ForDiap_PartIn_PartOut(pD, pF,
4                  (*pD_iStartIndexes)[mpi_rank * nThreads],
5                  (*pD_iStartIndexes)[i], (*pD_iStartIndexes)[i + 1],
6                  pSol);
7  }
```

Цикл for (строка 2) обходит все индексы, соответствующие началам блоков, принадлежащих текущему процессу. Директива “*pragma omp*” позволяет распределить эти итерации цикла между потоками.

4.2 Проверка функциональных требований программы

Полученную программу требуется проверить на удовлетворение функциональным требованиям. Для сравнения будут использоваться выходные данные, генерируемые многопоточной реализацией.

4.2.1 Компиляция программы

Для компиляции программы используются утилиты cmake и make.

Для запуска программы требуется использовать команду mpirun библиотеки MPI. Она принимает на вход, среди основных параметров, количество процессов, на которых требуется запустить программу. Количество запускаемых потоков, как рассматривалось ранее, расположено во входном файле, имя которого передается в качестве аргумента при запуске программы.

4.2.2 Запуск программы

Проведено несколько запусков с разными входными данными и разным количеством узлов и потоков, на которых происходило распараллеливание.

Написана утилита, которая сравнивает выходные данные. Данная утилита берет в учет точность, с которой требуется производить сравнение.

В результате сравнения результатов фрагментированной программы и многопоточной программы с использованием написанной утилиты, получено, что содержимое .dat файлов (про которые описано в пункте 3.4.2) совпадают с точностью 10^{-5}. Данные, которые выводятся в конце каждой временной итерации, совпадают с точностью 10^{-19}. Полученные точности являются достаточными [4].

4.3 Проверка требований фрагментированной программы

Помимо функциональных требований, также предъявлялись специальные требования, описанные в пункте 3.2. Они были разделены на две группы.

Рассмотрим требования, предъявляемые для фрагментированного программы с прицелом на LuNA.

1. Все процедуры в коде были написаны, имея в виду следующее: никакие входные данные не модифицируются и не используются глобальные переменные.
2. Были использованы примитивные типы языка C++.

Теперь рассмотрим требования, связанные с нефункциональными характеристиками. Выполнение части этих требований также рассматривается в следующей главе.

1. Было проверено, что программа решает задачу на разном количестве процессов и потоков правильно.
2. Разбиение векторов на фрагменты производится следующим образом: общий размер вектора N делится на количество запущенных процессов P. Полученное число – это размеры фрагментов на каждом процессе. В случае, если N не делится на P, получаемый остаток R распределяется между первыми R процессами (все процессы пронумерованы от 0 до P-1). Такой подход гарантирует то, что размеры фрагментов для одного и того же вектора на разных процессах отличаются не более чем на 1. Матрицы, фактически, не разбиваются, но обход производится так, будто они разбиты на фрагменты. И с учетом этого, используется такой же подход, как и для векторов.
3. Для того, чтобы проверить, что программа не выполняет лишних коммуникаций, нужно посмотреть на два свойства: как программа будет работать на тестовых данных, что описывается в следующей главе, и с точки зрения анализа схемы, описанной в пункте 3.2. По результату анализа, на схеме все коммуникации обоснованы, а так как схема полностью отображалась в код, то и в программе все коммуникации также обоснованы.
4. Достижение программой достаточной эффективности, а также эффективности распараллеливания рассматривается в следующей главе.

4.4 Итог разработки

В результате была разработана фрагментированная программа с учетом модели фрагментированного программирования и схемы, разработанной в предыдущей главе. Результаты, связанные с разработкой, также описаны в приложениях В и Г, в которых представлены руководство программиста и описание программы соответственно.

5 Тестирование программы

Данная глава посвящена тестированию разработанной программы для рассмотрения нефункциональных характеристик программы, а именно времени работы. Для сравнения, помимо многопоточной реализации, также присутствуют две MPI реализации, одна из которых не реализует параллелизм на уровне потоков, а вторая была разработана без учета теории фрагментированного программирования и с использованием другого подхода к распределению вычислений. Итого, среди сравниваемых реализаций присутствуют:

1. реализация с использованием только OpenMP (многопоточная);
2. реализация с использованием только MPI;
3. реализация с использованием MPI и OpenMP;
4. фрагментированная (написанная программа);

Для исследования нефункциональных свойств разумно проверить работу программ на разном оборудовании. В доступе было два кластера: кластер MVS [27] и кластер НГУ [28]. Тестирование с их использованием описано в пунктах 5.1 и 5.2 соответственно.

Структура тестов строится из поиска зависимости времени работы программ от количества используемых узлов. Так как многопоточная реализация не имеет возможности запускаться на нескольких узлах, ее график является горизонтальной прямой при зависимости времени от количества узлов и представляет собой продленную линию для удобства восприятия. Для всех графиков присутствуют табличные представления, приведенные в приложении Б.

Для всех тестов были использованы следующие основные данные задачи, которые могут использоваться для реальных экспериментов:

- Размер сетки: 200x200x25.
- Количество временных итераций: 1.
- Шаг временных итераций: 2000.
- Максимальное количество итераций для сходимости: 100.

Полные данные представлены в листинге 3, в пункте 3.4.2.

5.1 Тестирование программы на кластере MVS

В данном пункте будут описаны тесты, проведенные на кластере MVS. Использовался узел “МВС-10П ТОРНАДО” со следующими характеристиками:

- Процессор: Intel Xeon E5-2690 (Sandy Bridge).
- Частота процессора: 2.9 ГГц.
- Количество ядер процессора: 8.

- Количество виртуальных процессоров: 16.
- Количество процессоров в узле: 2.
- Количество ядер в узле: 16.
- Количество виртуальных процессоров в узле: 32.
- Объем оперативной памяти в узле: 64 ГБ.

На кластере MVS на аппаратном уровне существует две сети: Ethernet и InfiniBand. Использование каждого из них описано в пунктах 5.1.2 и 5.1.3 соответственно. А в пункте 5.1.1 рассматривается поиск оптимального количества потоков для всех реализаций.

5.1.1 Определение оптимального количества потоков

На рассматриваемом кластере было найдено оптимальное количество потоков для всех реализаций. Результаты представлены на рисунке 9 ниже. Использованные данные для построения графика представлены в таблице 3 приложения Б. Реализация “MPI” не реализует параллельные вычисления на уровне потоков, поэтому является горизонтальным графиком и получена в результате продления линии для удобства восприятия.

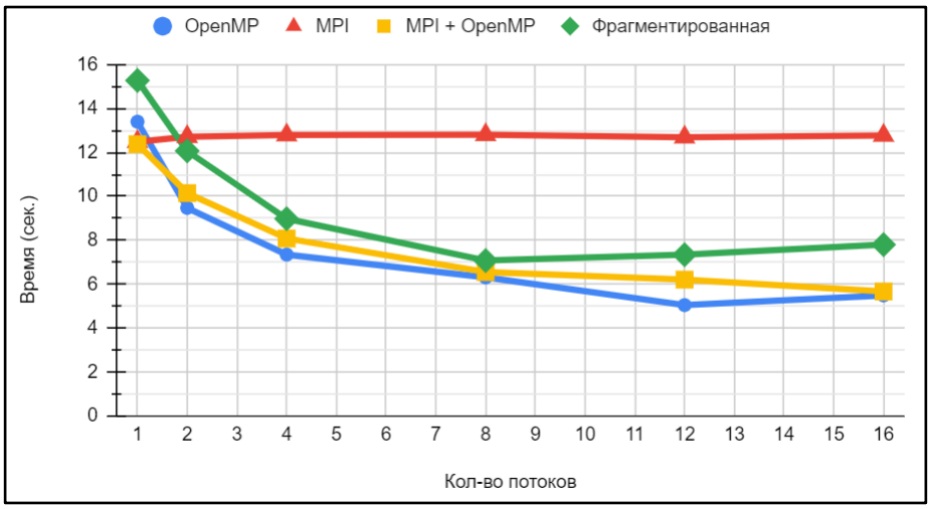

Рисунок 9 – Зависимость времени от количества потоков на 1 узле

Для OpenMP реализации оптимальным получилось 12 потоков; для MPI+OpenMP реализации – 16 потоков; для фрагментированной реализации – 8 потоков.

5.1.2 Выполнение запусков с использованием стандартного сетевого интерфейса Ethernet

С используемыми оптимальными количествами потоков были проведены запуски на нескольких узлах. Получены результаты, представленные на рисунке 10. Также эти данные представлены в таблице 4 приложения Б.

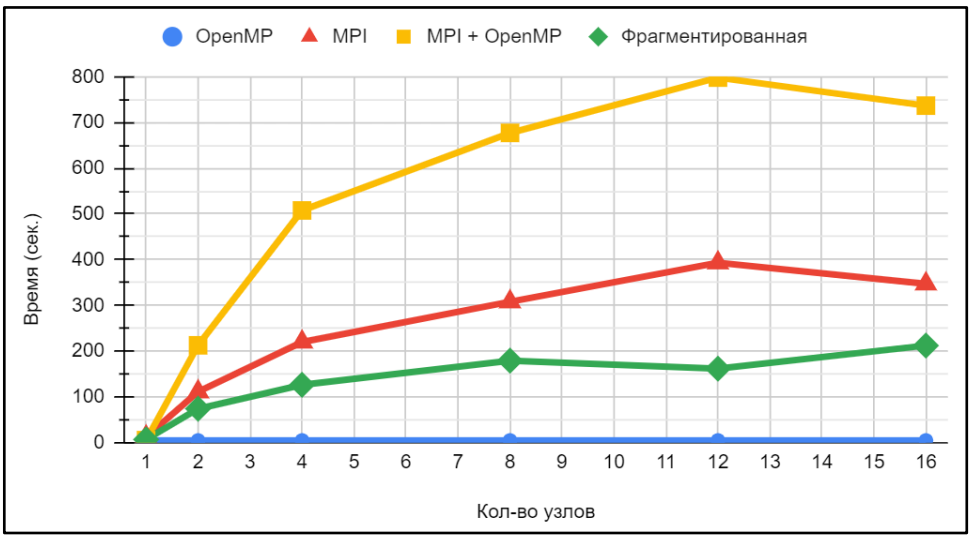

Рисунок 10 – Зависимость времени от количества узлов на оптимальном количестве потоков на кластере MVS

Из графика видно, что все MPI реализации работают дольше, чем OpenMP реализация. Но если не учитывать время, потраченное на коммуникации, то получим результаты, представленные на рисунке 11. Также эти данные представлены в таблице 4 приложения Б.

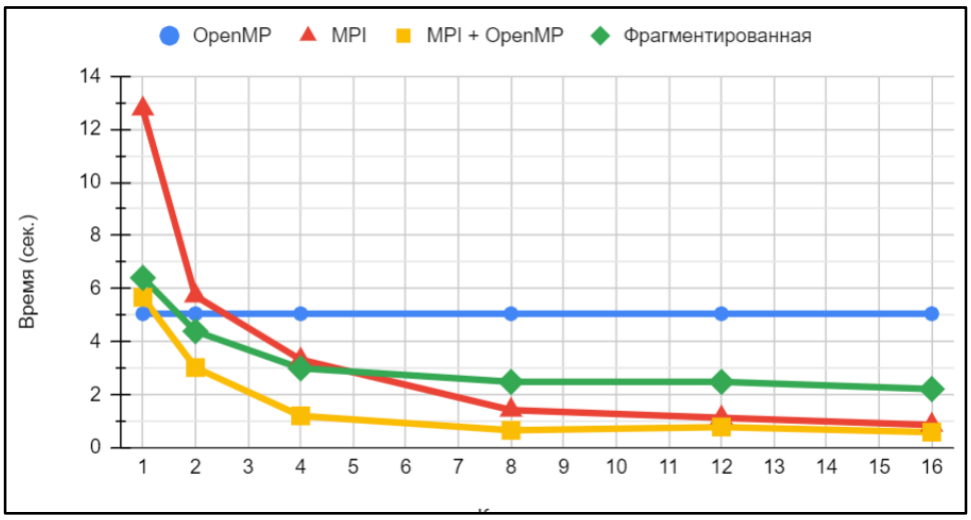

Рисунок 11 – Зависимость времени от количества узлов на оптимальном количестве потоков без коммуникаций на кластере MVS

Время на полезные вычисления у MPI реализаций тратится заметно меньше. Таким образом, большая часть времени работы программ тратится на коммуникации процессов.

5.1.3 Выполнение запусков с использованием сетевого интерфейса InfiniBand

Причина, по которой в предыдущем пункте реализации тратили много времени на коммуникации заключается в том, что при запусках использовался сетевой интерфейс по

умолчанию, который был медленнее, чем интерфейс InfiniBand. Поэтому рассмотрим также запуск программ с использованием быстрого сетевого интерфейса.

Был проведен запуск программ на тех же данных и с теми же оптимальными количествами потоков, что и в предыдущем пункте. Результат представлен ниже на рисунке 12. Эти же данные представлены в таблице 4 приложения Б.

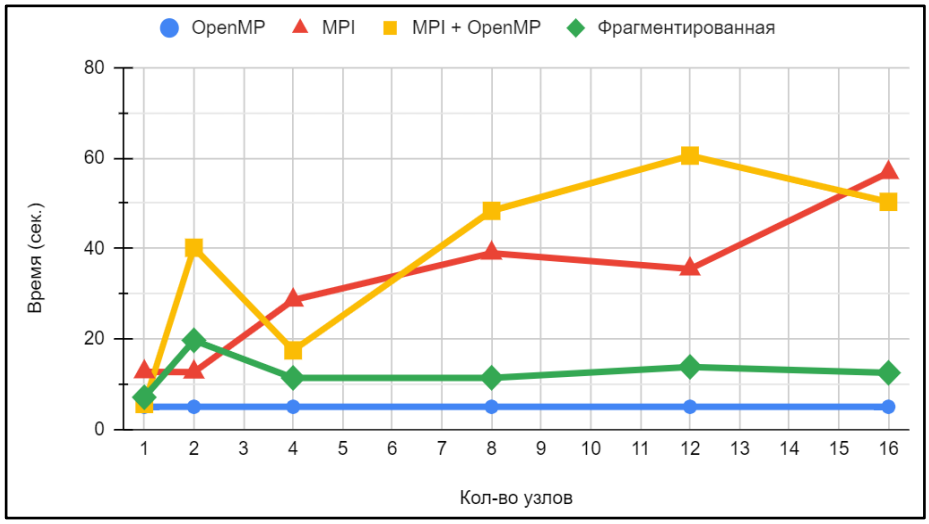

Рисунок 12 – Зависимость времени от количества узлов на оптимальном количестве потоков на кластере MVS с использованием InfiniBand

Время снизилось значительно, но все равно недостаточно, чтобы сравниться с OpenMP реализацией. Время работы без учета коммуникаций совпадает с тем, что представлено на рисунке 11.

В итоге отметим, что на кластере MVS реализациям не удалось получить выигрыш от распределенных вычислений.

5.2 Тестирование программы на кластере НГУ

Для выполнения тестов на кластере НГУ использовался узел “HP BL2x220c G7” со следующими характеристиками:

- Процессор: Intel Xeon X5670 (Westmere).
- Частота процессора: 2.932 ГГц.
- Количество ядер процессора: 6
- Количество виртуальных процессоров: 12.
- Количество процессоров в узле: 2.
- Количество ядер в узле: 12.
- Количество виртуальных процессоров в узле: 24.
- Объем оперативной памяти в узле: 24 ГБ.

5.2.1 Определение оптимального количества потоков

Как и в пункте 5.1.1, сначала было определено оптимальное количество потоков для каждой реализации, так как применить предыдущие данные невозможно в силу использования другого оборудования. Получены результаты, представленные на рисунке 13. Эти же данные также представлены в таблице 5 приложения Б.

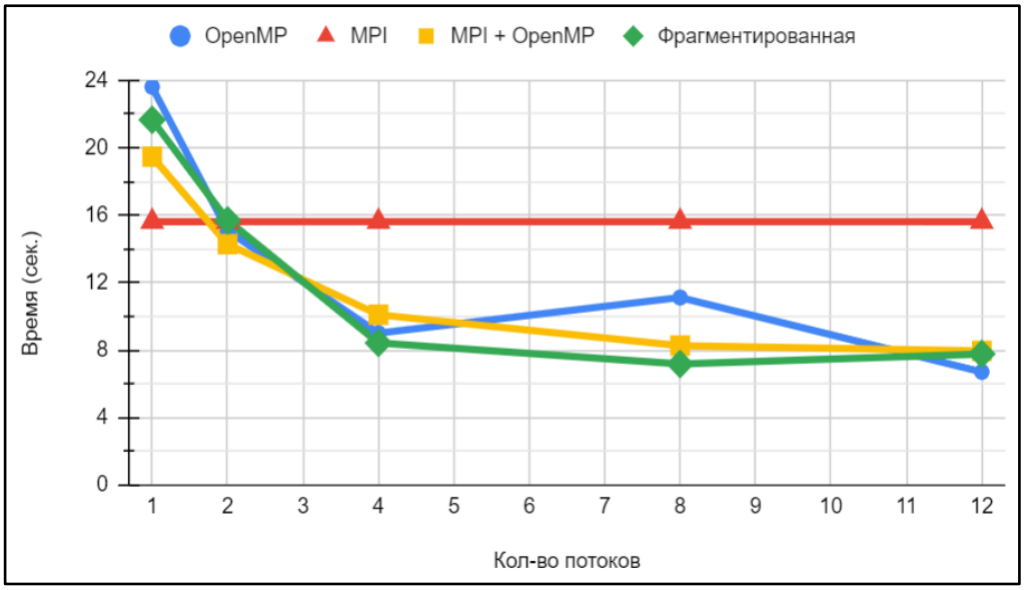

Рисунок 13 – Зависимость времени от количества потоков на 1 узле на кластере НГУ

Для OpenMP реализации оптимальным получилось 12 потоков; для MPI+OpenMP реализации оптимальным получилось 12 потоков; для фрагментированной реализации – 8 потоков.

5.2.2 Выполнение запусков

На найденных оптимальных количествах потоков был выполнен запуск всех реализаций на нескольких узлах. Результаты представлены ниже на рисунке 14. Эти же данные также представлены в таблице 6 приложения Б.

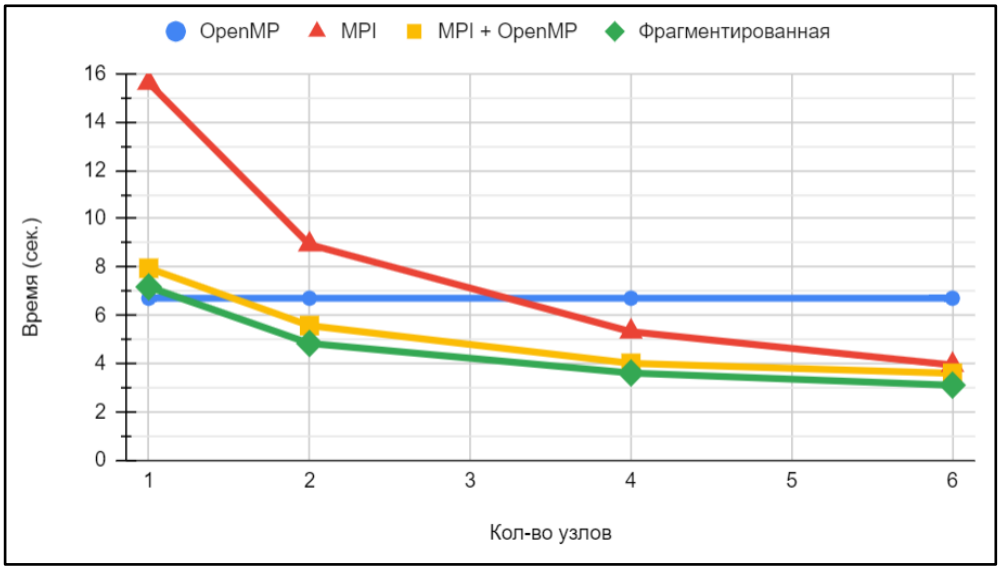

Рисунок 14 – Зависимость времени от количества узлов на оптимальном количестве потоков на кластере НГУ

Теперь все реализации сравнимы с OpenMP реализацией, так как время на коммуникации достаточно небольшое. В данном случае MPI реализации на нескольких узлах дают выигрыш по сравнению с OpenMP реализацией. Более того, фрагментированная реализация получилась быстрее всех. Если же опять взглянуть на зависимости без учета коммуникаций, то получится результат, который представлен на рисунке 15. Эти же данные также представлены в таблице 6 приложения Б.

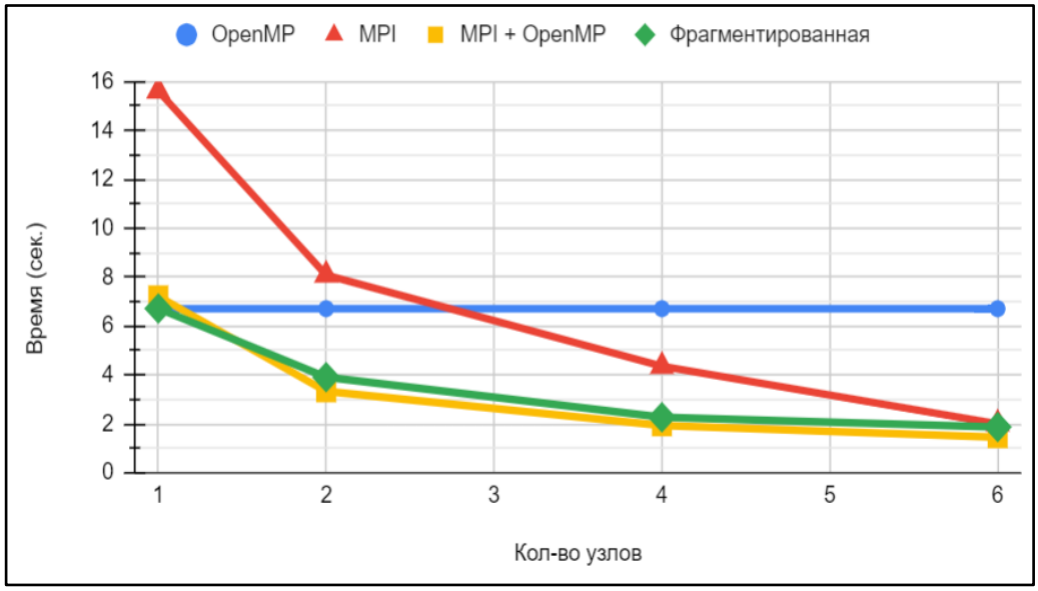

Рисунок 15 – Зависимость времени от количества узлов на оптимальном количестве потоков без коммуникаций на кластере НГУ

Эти результаты сопоставимы с результатами без коммуникаций, полученные для кластера MVS. Тем самым, подтверждается, что действительно фрагментированной реализации удалось сэкономить время работы путем уменьшения объема коммуникаций.

5.3 Анализ результатов тестирования

Рассмотрим эффективность распараллеливания реализаций на кластере НГУ. На кластере MVS этого смысла делать нет, так как все MPI реализации работают дольше, чем многопоточная.

Для начала рассмотрим понятия ускорения и эффективности распараллеливания [25, 29]. Ускорение определяется следующей величиной: $S_p = T_1/T_p$, где T_1 – время работы последовательной программы, а T_p – время работы параллельной программы на p процессорах. В качестве T_1 выбирается время работы многопоточной реализации с использованием 1-го потока. Далее, с использованием этой величины, определяется *эффективность* распараллеливания: $E_p = S_p/N$, где S_p – ускорение параллельной программы с использованием p процессоров. В результате получены графики, представленные ниже. На рисунке 16 показана зависимость ускорения от количества узлов, а на рисунке 17 – зависимость эффективности распараллеливания от количества узлов. Также эти данные представлены в таблице 7 Приложения Б.

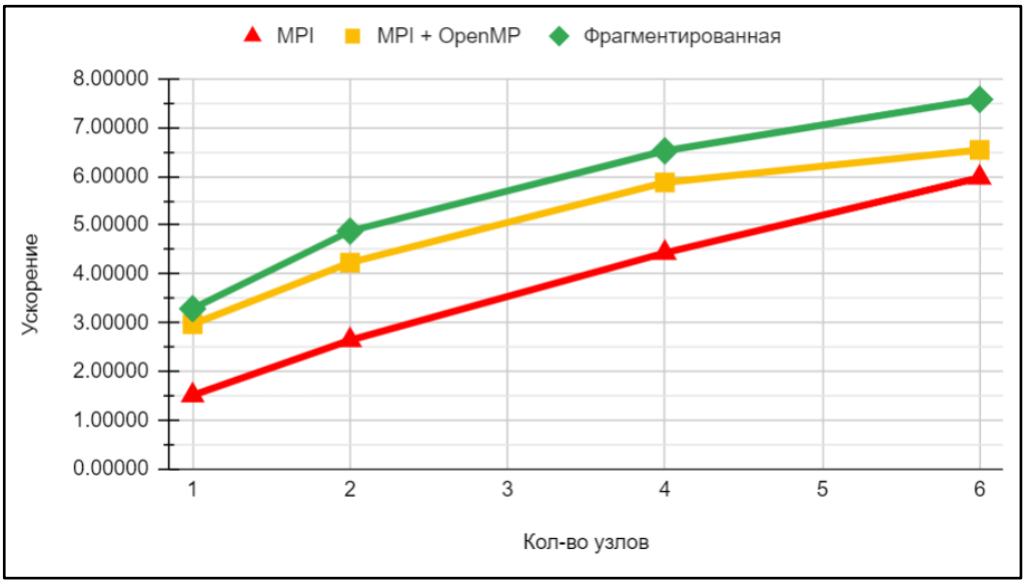

Рисунок 16 – Зависимость ускорения от количества узлов на кластере НГУ

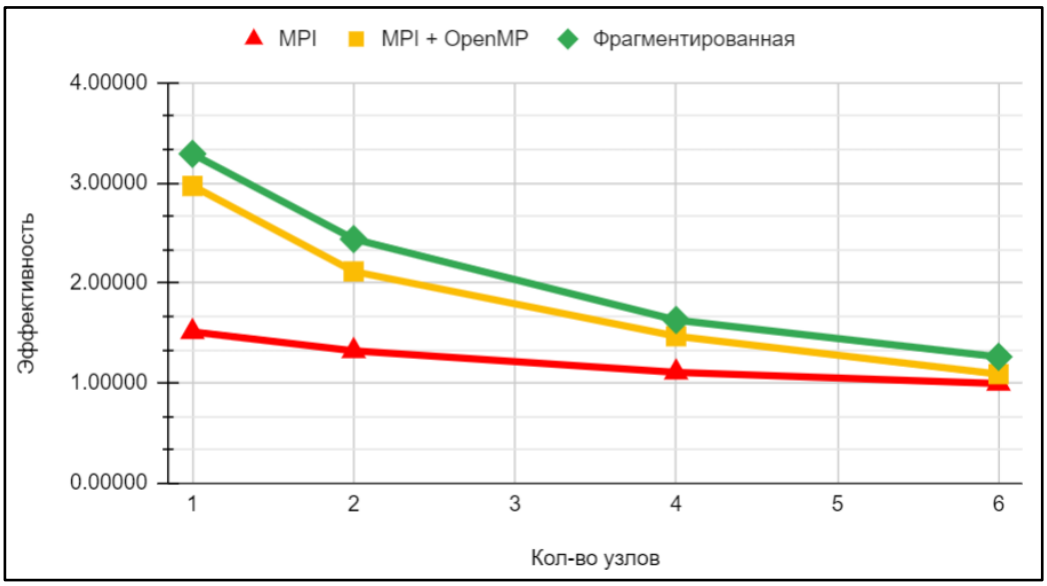

Рисунок 17 – Зависимость эффективности распараллеливания от количества узлов на кластере НГУ

Для созданной фрагментированной реализации полученная эффективность распараллеливания является удовлетворительной для решаемого класса задач [4].

На основе полученных данных можно увидеть, что фрагментированная программа работает достаточно эффективно. Достаточность определяется, во-первых, из того, как производился анализ и строилась схема фрагментированной программы: были рассмотрены все необходимые способы извлечения параллелизма из алгоритмов и оценено, как они должны быть реализованы в программе. Во-вторых, производительность получилась сопоставима с альтернативными реализациями, представленными в статьях [3-4].

ЗАКЛЮЧЕНИЕ

В результате работы разработана и исследована эффективная фрагментированная программа для задачи фильтрации двухфазной жидкости. В ходе работы был составлен подробный анализ решения задачи с точки зрения ее эффективной параллельной реализации, составлен граф зависимостей, предъявлены требования к программе. На основе требований составлена схема программы и выполнена ее реализация. Полученная программа исследована на эффективность посредством сравнения с другими реализациями.

Результаты работы также представлены в статье [30], где внесенным вкладом является описание реализации MPI программы. Совместный вклад с другими авторами, работавшими над этой задачей, внесен в описание анализа решения задачи и тестирование.

На защиту выносятся:

1. Требования, предъявляемые к фрагментированной программе.
2. Реализация фрагментированной программы.
3. Экспериментальное исследование эффективности фрагментированной программы.

В дальнейшем возможны следующие направления развития данной работы:

1. Реализация полученной схемы в системе LuNA (уже реализуется в другой работе на основе написанной в этой работе программы).
2. Улучшение системных алгоритмов в системе LuNA.
3. Возможное улучшение параллелизма в алгоритмах, решающих задачу.

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».

Спирин Виталий Андреевич _____

ФИО студента *Подпись студента*

« ____ »_____ 20__ г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Malyshkin V.E. Active Knowledge, LuNA and Literacy for Oncoming Centuries. // Springer, LNCS, Vol. 9465, pp. 292-303.
2. Иванов М.И., Кремер И.А., Лаевский Ю.М. Моделирование процесса фильтрации двухфазной жидкости на основе законов сохранения в интегральной форме, 2020. - 35 с.
3. Ivanov M. I., Kremer I. A., Laevsky Y. M. On the streamline upwind scheme of solution to the filtration problem //Sib. Electronic Math. Reports. – 2019. – Т. 16. – С. 757-776.
4. Ivanov M. I., Kremer I. A., Laevsky Y. M. On wells modeling in filtration problems //Sib. Electronic Math. Reports. – 2019. – Т. 16. – С. 1868-1884.
5. Перепёлкин В.А., Иванов М.И. Повышение производительности LuNA-программ на основе воспроизведения трасс // В сборнике: Десятая Сибирская конференция по параллельным и высокопроизводительным вычислениям. Сборник статей. Под редакцией А.В. Старченко. Томск, 2021. С. 29-36.
6. Спирин В. А. Разработка эффективной фрагментированной программы решения задачи фильтрации двухфазной жидкости / В. А. Спирин // Информационные технологии. Цифровые технологии нефтегазовой индустрии : Материалы 61-й Междунар. науч. студ. конф. 17-26 апреля 2023 г. / Новосиб. гос. ун-т. — Новосибирск : ИПЦ НГУ, 2023
7. V.E. Malyshkin, V.A.Perepelkin. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem. // Proceedings of the 11th International Conference on Parallel Computing Technologies (PaCT-2011), LNCS, Vol. 6873, 2011, P. 53-61. DOI: 10.1007/978-3-642-23178-0_5
8. Перепелкин В.А., Софронов И.В., Ткачёва А.А. Автоматизация конструирования численных параллельных программ с заданными нефункциональными свойствами на базе вычислительных моделей //журнал Проблемы информатики, 2017, № 4. С.47-60
9. Перепелкин, В. А. Проблема распределения ресурсов мультикомпьютера в технологии фрагментированного программирования // Научный сервис в сети Интернет: поиск новых решений: Труды Международной суперкомпьютерной конференции (17-22 сентября 2012 г., г. Новороссийск). – М.: Изд-во МГУ, 2012. – С. 398 – 401.
10. Киреев С.Е., Перепелкин В.А., Ткачёва А.А. Реализация метода IADE_RB_CG в системе фрагментированного программирования LuNA // VIII Сибирская конференция по параллельным и высокопроизводительным вычислениям: Программа и тезисы докладов (28-30 октября 2015 года). – Томск: Изд-во Том. Ун-та, 2015, с. 25.
11. PLASMA Library [Электронный ресурс] / – Режим доступа: http://icl.cs.utk.edu/plasma/ (дата обращения: 03.05.2023)

12. Plasma User Guide [Электронный ресурс] / – Режим доступа: https://icl.utk.edu/projectsfiles/plasma/pdf/users_guide.pdf (дата обращения: 03.05.2023)
13. Kale, L. V. Parallel Science and Engineering Applications: The Charm++ Approach / L. V. Kale , A. Bhatele. – CRC Press, 2013. – 314 P. 9.
14. Kale, L. V., Krishnan, S. Charm++: a portable concurrent object oriented system based on c++ // Proceedings of OOPSLA'93. – 1993. – Vol. 28. – P. 91 – 108.
15. Acun B. et al. NAMD: Scalable molecular dynamics based on the Charm++ parallel runtime system //Exascale Scientific Applications: Scalability and Performance Portability. – 2017. – С. 119-143.
16. Danalis, A. PaRSEC in Practice: Optimizing a legacy Chemistry application through distributed task-based execution / A. Danalis, H. Jagode, G. Bosilca, J. 37 Dongarra // 2015 IEEE International Conference on Cluster Computing. – 2015. – P. 304 – 313.
17. Abdulah S. et al. Accelerating geostatistical modeling and prediction with mixed-precision computations: A high-productivity approach with parsec //IEEE Transactions on Parallel and Distributed Systems. – 2021. – Т. 33. – №. 4. – С. 964-976.
18. Bauer, M. Legion: Expressing Locality and Independence with Logical Regions / M. Bauer, S. Treichler, E. Slaughter, A. Aiken // SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis – 2012. – Article N 66 – P. 1 – 11.
19. Soi R. et al. An Implicitly Parallel Meshfree Solver in Regent //2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+ X (PAW-ATM). – IEEE, 2020. – С. 40-54.
20. Попов П. Е. Математическое моделирование процессов фильтрации двухфазной несжимаемой жидкости на параллельных вычислительных системах : дис. – Институт вычислительной математики и математической геофизики, 2011.
21. The OpenMP API specification for parallel programming [Электронный ресурс] / – Режим доступа: https://www.openmp.org (дата обращения: 03.05.2023)
22. Samadi M. et al. Paraprox: Pattern-based approximation for data parallel applications //Proceedings of the 19th international conference on Architectural support for programming languages and operating systems. – 2014. – С. 35-50.
23. Buluç A. et al. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks //Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. – 2009. – С. 233-244.
24. Годунов С. К., Рябенький В. С. Разностные схемы (введение в теорию). – 1977. – С. 51-53.
25. Sanders P. et al. Sequential and Parallel Algorithms and Data Structures. – Springer, 2019.

26. Message Passing Interface (MPI) Forum [Электронный ресурс] / – Режим доступа: https://www.mpi-forum.org (дата обращения: 03.05.2023)
27. Межведомственный Суперкомпьютерный Центр Российской Академии Наук [Электронный ресурс] / – Режим доступа: https://www.jscc.ru/ (дата обращения: 03.05.2023)
28. Информационно-вычислительный центр Новосибирского государственного университета [Электронный ресурс] / – Режим доступа: http://nusc.nsu.ru/wiki/doku.php/doc/index (дата обращения: 03.05.2023)
29. Eager D. L., Zahorjan J., Lazowska E. D. Speedup versus efficiency in parallel systems //IEEE transactions on computers. – 1989. – Т. 38. – №. 3. – С. 408-423.
30. Кудрявцев А.А., Малышкин В.Э., Нуштаев Ю.Ю., Перепёлкин В.А., Спирин В.А. Эффективная фрагментированная реализация краевой задачи фильтрации двухфазной жидкости // журнал "Проблемы информатики", 2023 (принято в печать).

ПРИЛОЖЕНИЕ А

Полный граф зависимостей для метода сопряженных градиентов

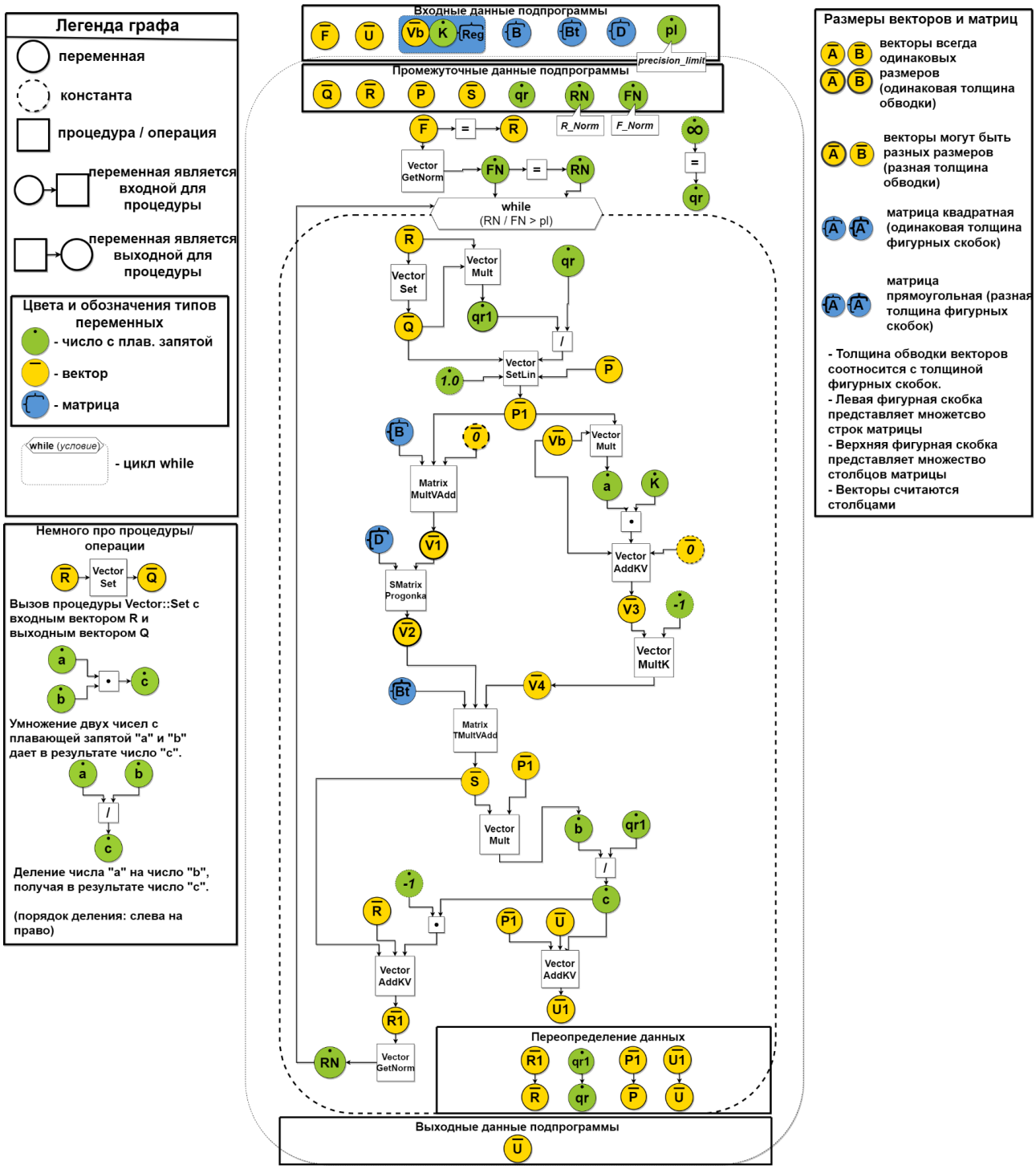

Рисунок А.1 – Полный граф зависимостей для метода сопряженных градиентов задачи фильтрации

ПРИЛОЖЕНИЕ Б

Табличное представление результатов тестирования

Таблица 3 – Время работы (сек.) программ на кластере MVS на 1 узле

Кол-во потоков	OpenMP	MPI	MPI + OpenMP	Фрагмент-ная
1	13.398	12.4887	12.3749	15.2769
2	9.47449	12.7149	10.1465	12.0671
4	7.34727	12.7897	8.08046	8.98081
8	6.31089	12.8013	6.55378	7.07646
12	5.0497	12.6929	6.20829	7.34056
16	5.49236	12.7722	5.66782	7.80016

Таблица 4 – Время работы (сек.) программ на кластере MVS

Кол-во узлов	OpenMP Ethernet / InfiniBand / Без коммуникаций	MPI Ethernet / InfiniBand / Без коммуникаций	MPI + OpenMP Ethernet / InfiniBand / Без коммуникаций	Фрагмент-ная Ethernet / InfiniBand / Без коммуникаций
1	5.04908 / 5.04908 / 5.04908	12.7722 / 12.7722 / 12.7722	5.66782 / 5.66782 / 5.66782	7.07646 / 7.14995 / 6.4
2	5.04908 / 5.04908 / 5.04908	111.282 / 12.7094 / 5.724	212.915 / 40.1807 / 3.02	74.3729 / 19.7217 / 4.39
4	5.04908 / 5.04908 / 5.04908	220.155 / 28.6087 / 3.316	507.623 / 17.507 / 1.2	126.619 / 11.4196 / 2.99
8	5.04908 / 5.04908 / 5.04908	308.047 / 38.9884 / 1.419	677.097 / 48.3358 / 0.6526	179.469 / 11.4458 / 2.48
12	5.04908 / 5.04908 / 5.04908	393.17 / 35.4873 / 1.131	797.926 / 60.526 / 0.781	162.033 / 13.8441 / 2.48

Окончание таблицы 4

Кол-во узлов	OpenMP Ethernet / InfiniBand / Без коммуникаций	MPI Ethernet / InfiniBand / Без коммуникаций	MPI + OpenMP Ethernet / InfiniBand / Без коммуникаций	Фрагмент-ная Ethernet / InfiniBand / Без коммуникаций
16	5.04908 / 5.04908 / 5.04908	347.21 / 56.7864 / 0.856	736.688 / 50.3151 / 0.58577	212.376 / 12.5571 / 2.21

Таблица 5 – Время работы (сек.) программ на кластере НГУ на 1 узле

Кол-во потоков	OpenMP	MPI	MPI + OpenMP	Фрагмент-ная
1	23.6106	15.6164	19.4681	21.6499
2	15.0897	15.6164	14.2832	15.6893
4	9.00891	15.6164	10.098	8.42741
8	11.1185	15.6164	8.26984	7.17733
12	6.71136	15.6164	7.95094	7.77334

Таблица 6 – Время работы (сек.) программ на кластере НГУ

Кол-во узлов	OpenMP С коммуникациями / без коммуникаций	MPI С коммуникациями / без коммуникаций	MPI + OpenMP С коммуникациями / без коммуникаций	Фрагмент-ная С коммуникациями / без коммуникаций
1	6.71136 / 6.71136	15.6164 / 15.6164	7.95094 / 7.22	7.17733 / 6.71733
2	6.71136 / 6.71136	8.93176 / 8.089	5.58007 / 3.32654	4.84004 / 3.91
4	6.71136 / 6.71136	5.32752 / 4.35659	4.01865 / 1.93565	3.61975 / 2.27
6	6.71136 / 6.71136	3.95258 / 2.0	3.60899 / 1.453	3.11598 / 1.87

Таблица 7 – Ускорение и эффективность программ на кластере НГУ

Кол-во узлов	MPI Ускорение / Эффективность	MPI + OpenMP Ускорение / Эффективность	Фрагмент-ная Ускорение / Эффективность
1	1.51191 / 1.51191	2.96954 / 2.96954	3.28961 / 3.28961
2	2.64344 / 1.32172	4.23124 / 2.11562	4.87818 / 2.43909
4	4.43182 / 1.10795	5.87526 / 1.46881	6.52272 / 1.63068
6	5.97347 / 0.99558	6.54216 / 1.09036	7.57726 / 1.26288

ПРИЛОЖЕНИЕ В

ФРАГМЕНТИРОВАННАЯ ПРОГРАММА РЕШЕНИЯ ЗАДАЧИ ФИЛЬТРАЦИИ ДВУХФАЗНОЙ ЖИДКОСТИ

РУКОВОДСТВО ПРОГРАММИСТА

Листов 8

Новосибирск 2023

АННОТАЦИЯ

В данном документе приведено руководство программиста для фрагментированной программы решения задачи фильтрации двухфазной жидкости. Исходным языком программы является C++.

Основной функцией программы является моделирование процесса фильтрации двухфазной жидкости.

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

СОДЕРЖАНИЕ

1 Назначения и условия применения программы

1.1 Назначение программы

Фрагментированная программа предназначена для решения задачи моделирования процесса фильтрации двухфазной жидкости на основе законов сохранения в интегральной форме с использованием возможностей распределенных вычислений. Фрагментированность программы означает связь с системой LuNA.

1.2 Функции, выполняемые программой

Инициализация и анализ данных выполняется также, как и в других реализациях программы.

Основу фрагментированной программы составляет выполнение метода сопряженных градиентов с использованием распределенных вычислений.

1.3 Условия, необходимые для выполнения программы

Чтобы программа могла запускаться, необходимо наличие у программиста:

1. компилятора gcc;
2. утилит cmake и make;
3. библиотеки MPI;
4. библиотеки OpenMP;
5. утилиты gnuplot для визуализации выходных данных.

2 Характеристика программы

2.1 Описание основных характеристик программы

Программа имеет один режим работы: считывание входных данных, моделирование процесса фильтрации, формирование выходных данных.

Время работы программы зависит от:

1. размеров сетки;
2. количества временных итераций;
3. верхнего предела количества итераций для сходимости метода сопряженных градиентов внутри одной временной итерации;
4. скорости передачи данных между узлами.

2.2 Описание основных особенностей программы

Программа генерирует выходные данные в виде:

1. Описания данных задачи на каждом временном шаге.
2. Данных моделирования, расположенных в файлах типа .dat, которые можно визуализировать с помощью утилиты gnuplot (или аналогичных).

Также программа имеет возможность записывать данные (векторы и матрицы) в файлы и считывать эти данные из файлов. При такой потребности требуется вручную указать в программе, в нужном месте, что тот или иной вектор или матрица должны быть записаны в файл.

3 Обращение к программе

3.1 Описание процедур вызова программы

Основу программы составляет метод сопряженных градиентов `MPI_Solve_SLAE2`, расположенный в файле `Filtration_Base/MPISolver/MPI_Shur_Method_V2.cpp`.

При запуске всей программы достаточно использовать входные данные задачи. Программа использует точку входа, расположенную в файле `Filtration3D/Filtration3D.cpp`, и запускается с использованием `mpirun`. Для запуска программы требуются следующие ключи:

1. `-run`;
2. `-data "data3d.txt"` - файл с описанием входных данных задачи;
3. `-dir "Out"` - имя директории, в которую будут записаны выходные данные.

Если есть потребность в использовании только метода сопряженных градиентов, то можно сохранить данные (векторы и матрицы) в файл вызовом у их классов процедуры LoadStore. Этой же процедурой можно считывать данные из файлов.

4 Входные и выходные данные

4.1 Организация используемой входной информации

Входные данные - данные задачи. Они представлены в листинге 3.

4.2 Организация используемой выходной информации

В качестве выходных данных программа будет, во-первых, выводить данные, пример которых представлен в листинге 4, в консоль. Во-вторых, будут сгенерированы файлы моделирования. Список этих файлов совпадает с тем, что представлен в секции “Analyse” листинга 3. Содержимое файлов состоит из 3-х столбцов: значения по осям y, x и значение функции в этих точках. Эти .dat файлы можно визуализировать, например, при помощи утилиты gnuplot.

5 Сообщения

Если программа работает на нескольких потоках и какие-то из них работают на одном ядре, то будет выведено соответствующее сообщение. Это не останавливает ход работы программы. Пример сообщения представлен в листинге В.1 ниже.

Листинг В.1 – Сообщение при работе нескольких потоков на одном ядре

```
1  Some threads are running on the same core:
2  Thread 0 is running on cpu 0
3  Thread 1 is running on cpu 11
4  Thread 2 is running on cpu 7
5  Thread 3 is running on cpu 1
6  Thread 4 is running on cpu 3
7  Thread 5 is running on cpu 7
8  Thread 6 is running on cpu 0
9  Thread 7 is running on cpu 5
```

Также возможен вывод сообщения, представленный в листинге В.2.

Листинг В.2 – Сообщение об инициализации MPI с потоками

```
1  MPI_Init_thread: provided=2, desired=3
```

В начале любой MPI программы требуется вызвать MPI_Init, а в случае, если будут использоваться потоки, нужно вызывать MPI_Init_thread. В качестве параметров MPI_Init_thread, помимо тех, что есть и у MPI_Init, принимает данные о том, как предполагается использовать потоки:

0. **MPI_THREAD_SINGLE –** будет запущен только 1 поток.
1. **MPI_THREAD_FUNNELED** – может быть запущено несколько потоков, но только основной поток будет делать MPI вызовы.
2. **MPI_THREAD_SERIALIZED** – может быть запущено несколько потоков, и несколько потоков могут делать MPI вызовы, но не одновременно.
3. **MPI_THREAD_MULTIPLE** – несколько потоков могут делать MPI вызовы без ограничений.

В программе запрашивается константа MPI_THREAD_MULTIPLE (3), о чем говорит надпись “*desired=3*” в листинге В.2. Надпись “*provided=2*” говорит о том, что применилась константа MPI_THREAD_SERIALIZED (2), потому что MPI_THREAD_MULTIPLE (3) применить не удалось.

В программе никогда не происходит того, что два или более потоков одновременно делают MPI вызовы (более того, только основной поток делает MPI вызовы), поэтому, из описания выше, достаточно использовать константу MPI_THREAD_FUNNELED (1). Поэтому если вы видите сообщение, как в листинге В.2, убедитесь, что значение *provided* равно как минимум 1.

ПРИЛОЖЕНИЕ Г

ФРАГМЕНТИРОВАННАЯ ПРОГРАММА РЕШЕНИЯ ЗАДАЧИ ФИЛЬТРАЦИИ ДВУХФАЗНОЙ ЖИДКОСТИ

ОПИСАНИЕ ПРОГРАММЫ

Листов 10

Новосибирск 2023

АННОТАЦИЯ

В данном документе приведено описание фрагментированной программы решения задачи фильтрации двухфазной жидкости.

Чтобы программа могла исполняться, необходимо наличие компилятора gcc, библиотек MPI и OpenMP, а также утилит cmake и make. Для визуализации результатов работы программы требуется утилита plot.

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

1 Общие сведения

1.1 Обозначение и наименование программы

Полное название – фрагментированная программа решения задачи фильтрации двухфазной жидкости.

Обозначение – Filtration 3D 2022-2023.

1.2 Программное обеспечение, необходимое для функционирования программы

Необходимое ПО – gcc, cmake, make, MPI, OpenMP, gnuplot.

1.3 Языки программирования

Язык программирования – C++.

2 Функциональное назначение

2.1 Назначение программы

Программа предназначена для моделирования процесса фильтрации двухфазной жидкости с использованием распределенных вычислений, использующая модель фрагментированного программирования.

2.2 Сведения о функциональных ограничениях на применение

Программа должна запускаться через `mpirun`.

3 Описание логической структуры

3.1 Алгоритм программы

Алгоритмы формирования данных (векторы и матрицы) и формирования выходных данных совпадают с существующими реализациями. Были внесены модификации в метод сопряженных градиентов, которые позволяют запускать программу на нескольких процессах.

3.2 Используемые методы

Работа программы на нескольких процессах достигается за счет разбиения данных между процессами и своевременных коммуникаций для обмена и сбора данных.

3.3 Структура программы

Программа состоит из набора классов, представляющих информацию о моделировании задачи, и классов, представляющих математические структуры (векторы, матрицы), а также классов, реализующих разные способы вычислений.

3.4 Связи между составными частями программы

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

3.5 Связи программы с другими

Для запуска программы необходимо наличие компилятора gcc, библиотек MPI и OpenMP, утилит cmake и make. Для визуализации данных моделирования требуется утилита gnuplot, или аналогичные.

4 Используемые технические средства

Режим работы – консольное приложение.

Для работы программы необходимо устройство со следующими требованиями:

1. Количество ядер в процессоре – столько, сколько требуется потоков для запуска.
2. Объем оперативной памяти – зависит от размера используемой модели (например, для модели 100x100x100, достаточно 4 ГБ памяти).
3. Объем дисковой памяти – 2 МБ для исходных файлов, и дополнительный объем, который зависит от размеров модели (для модели 100x100x100 достаточно дополнительно 4 МБ памяти).

5 Вызов и загрузка

Запуск программы осуществляется с использованием `mpirun`. Синтаксис команды должен быть таким:

```
mpirun -np <кол-во процессов> ./filtration3d -run -data “data3d.txt” -dir “Out”,
```

где “`data3d.txt`” – название входного файла, “`Out`” – название директории, в которую будут сохраняться выходные данные.

6 Входные данные

Входные данные - данные задачи. Пример их описания представлен в листинге 3.

7 Выходные данные

В качестве выходных данных программа будет, во-первых, выводить данные, пример которых представлен в листинге 4, в консоль. Во-вторых, будут сгенерированы файлы моделирования. Список этих файлов совпадает с тем, что представлен в секции “Analyse” листинга 3. Содержимое файлов состоит из 3-х столбцов: значения по осям y, x и значение функции в этих точках. Эти .dat файлы можно визуализировать, например, при помощи утилиты gnuplot.