МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Кафедра Параллельных Вычислений

Анна Ильинична Черникова

ФРАГМЕНТАЦИЯ АЛГОРИТМОВ РЕАЛИЗАЦИИ СИМПЛЕКС-МЕТОДА И РАЗРАБОТКА ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
по направлению высшего профессионального образования

230100.68 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Тема диссертации утверждена распоряжением по НГУ № 1 от «11» января 2012г.
Тема диссертации скорректирована распоряжением по НГУ № 533 от «14» декабря 2012г.

Руководитель

Малышкин В.Э

д.т.н., проф.

Новосибирск, 2013г.

Содержание

ВВЕДЕНИЕ 5

1 Представление алгоритмов для параллельного программирования 7

1.1 Проблемы параллельного программирования 7

1.2 Требования к представлению алгоритмов и программ 7

1.3 Обзор средств параллельного программирования и библиотек 10

1.3.1 Charm++ 10

1.3.2 SMP Superscalar 10

1.3.3 Библиотеки PLASMA и DPLASMA 11

1.3.4 Технология фрагментированного программирования 12

1.4 Постановка задачи 13

2 Разработка фрагментированного алгоритма симплекс-метода 15

2.1 Задача линейного программирования (ЛП) 15

2.2 Симплекс-метод 16

2.3 Фрагментация наивного алгоритма симплекс-метода 18

2.3.1 Описание алгоритма 18

2.3.2 Анализ алгоритма 19

2.3.3 Фрагментация алгоритма 21

2.4 Фрагментация модифицированного алгоритма симплекс-метода 23

2.4.1 Описание алгоритма 23

2.4.2 Анализ алгоритма 24

2.4.3 Фрагментация алгоритма 25

3 Разработка фрагментированных программ 33

3.1 Модуль Reading mps 33

3.2 Модуль Preprocessing 34

3.3 Модуль LU 35

3.4 Модуль GE 36

3.5 Модуль Naive 36

3.6 Модуль Modified 37

3.7 Модуль Constants 38

4 Тестирование фрагментированного алгоритма 40

ЗАКЛЮЧЕНИЕ 41

Литература 42

Приложение А Графическое представление фрагментированного наивного алгоритма симплекс-метода 44

ВВЕДЕНИЕ

Создание параллельных программ (ПП) – трудоемкий процесс и требует от программиста определенных навыков. Однако с каждым днем потребность в параллельных вычислениях растет, о чем свидетельствует рост числа вычислительных элементов в современных процессорах. Наращивать далее производительность процессоров бессмысленно, ведь предел тактовой частоты, диктуемый законами физики, практически достигнут. Благодаря наличию массового параллелизма в задачах, которые ставят современные физика, химия, математика, и другие отрасли, параллельное программирование становится основным инструментом решения этих задач.

В процессе разработки ПП необходимо решить ряд проблем, которые не относятся непосредственно к решению целевой задачи, а скорее представляют собой сервисную часть, без которой, однако, программа не сможет работать эффективно. Под эффективностью ПП подразумевается сокращение времени счета, увеличение точности полученного решения, экономное использование памяти и др. К упомянутым выше проблемам относятся отображение переменных и операций на ресурсы вычислителя, динамическая и статическая балансировка нагрузки, обмен данными между узлами вычислителя и другие. Для решения таких задач необходимы специальная подготовка и время. Весьма привлекательна возможность иметь простой инструмент, для конструирования ПП, который бы позволил автоматизировать сервисную часть и сосредоточиться на программировании непосредственно алгоритма.

Технология фрагментированного программирования (ТФП) предоставляет большие возможности в области программирования численных алгоритмов [1]. Библиотеки фрагментированных подпрограмм для задач прикладной математики являются удобным инструментарием для создания эффективных параллельных программ, в том числе пользователями, неспециализирующимся в параллельном программировании.

ТФП решает проблему распределения ресурсов: фрагментированная программа исполняется максимально параллельно на всех доступных ресурсах, динамическая балансировка нагрузки обеспечивается системой фрагментированного программирования (СФП). В настоящий момент на базе Института вычислительной математики и математической геофизики в лаборатории Синтеза параллельных программ ведется разработка СФП LuNA.

Для того, чтобы воспользоваться преимуществами ТФП, необходимо записать алгоритм в специальном виде. Фрагментированный алгоритм состоит из фрагментов

данных и фрагментов вычислений, при этом размер фрагментов – настраиваемый параметр. Фрагменты данных должны быть примерно одного размера, чтобы упростить задачу их эффективного распределения по ресурсам. Алгоритм представляется в СФП математической записью для обеспечения переносимости программы на любое оборудование.

Для увеличения эффективности использования ТФП необходимо дополнить систему библиотекой фрагментированных подпрограмм. Имеет смысл включить в библиотеку распространенные численные алгоритмы, например, алгоритмы на регулярных и нерегулярных сетках, методы частиц и другие.

Целью данной работы является разработка фрагментированного алгоритма и фрагментированной программы решения задачи линейного программирования симплекс-методом [2]. Разработка фрагментированного алгоритма симплекс-метода ведется в рамках наполнения библиотеки подпрограмм СФП LuNA. Симплекс-метод широко распространен на практике для решения оптимизационных задач, чем обусловлена целесообразность разработки фрагментированных программ для данного алгоритма.

1 Представление алгоритмов для параллельного программирования

1.1 Проблемы параллельного программирования

Реалии современного мира таковы, что нужно решать задачи на большом числе вычислительных узлов (ВУ). Размеры задач и вычислителей неуклонно растут. Многие задачи имеют хорошую масштабируемость и их можно эффективно решать на многопроцессорных вычислителях [3]. Казалось бы, что мешает, ведь давно уже существуют средства для разработки параллельных программ?

Особенность большинства существующих программных средств – это статическое распределение ресурсов, что значительно снижает гибкость управления. Многие решения по управлению вычислениями выгоднее принимать динамически. Современные задачи и вычислители требуют динамического подхода к программированию: для них нужны программы, в которых распределение ресурсов и управление недоопределены на стадии компиляции и окончательное решение об управлении принимается во время исполнения программы.

Также существенным ограничением при проектировании программ является зависимость от вычислительной среды. В настоящий момент активно развиваются средства программирования, основанные на платформонезависимом представлении алгоритма. Примеры систем: Charm++, ProActive, SMP Superscalar. Активно развиваются технологии, использующие недоопредленное представление программ с динамическим управлением ресурсами и вычислениями. Данная технология используется, например, в параллельных программных пакетах PLASMA, Uintah.

Решая проблему зависимости программы от вычислительной среды, появляется возможность также решить проблемы, не связанные с функциональностью программы, имеющие системный характер. К таким задачам можно отнести выделение памяти под данные, обеспечение коммуникаций между ВУ, синхронизацию вычислений и другие. Несмотря на то, что непосредственно к алгоритму данные задачи не имеют отношения, однако, без их решения программа не будет исполняться эффективно, либо вообще не будет исполняться.

1.2 Требования к представлению алгоритмов и программ

Процесс распараллеливания программы можно представить в виде совокупности двух задач управления – распределение вычислений в пространстве и во времени. Плохое управление может привести к низкой производительности программы из-за простоев ресурсов по причине глобальных синхронизаций и дисбаланса загрузки, а также к

ошибкам типа deadlock, и др. Таким образом, в основе современного параллельного программирования должна лежать идея о разделении ответственностей на задачи программирования алгоритмов и задачи управления. Подобное разделение вполне разумно и допустимо, т.к. это две существенно разные задачи. Разделив задачу параллельного программирования на два независимых подмножества задач, можно автоматизировать некоторые типичные подзадачи из каждого подмножества. Так, алгоритмы многих широко применяемых численных методов имеют значительные сходства, что позволяет создать специальные шаблоны для конструирования программ по этим алгоритмам. Задачу статического планирования вычислений можно вынести в отдельный планировщик. А поддержку динамических свойств программы, таких как начальная настройка на ресурсы, динамическая балансировка загрузки, организация асинхронных вычислений и передач данных, можно вынести в специальную runtime-систему .

Чтобы автоматически преобразовать алгоритм в параллельную программу, необходимо получить такое его представление, которое не будет зависеть от вычислительной среды, и в то же время полученное представление должно обеспечивать эффективное отображение алгоритма на конкретную вычислительную систему. Для того, чтобы получить желаемое представление, необходимо выделить определенные критерии, котором оно должно соответствовать.

Задача эффективного, т.е. оптимального, отображения алгоритма (т.е. переменных и операций) на ресурсы вычислительной системы в общем случае является NP-полной, т.е. уже на этапе компиляции планирование ресурсов может занять больше времени, нежели непосредственно вычисления. Динамика и обстановка в вычислительной среде значительно усугубят данную ситуацию, и может оказаться, что данная задача не решаема в разумно обозримых сроках. Однако, если число объектов алгоритма уменьшить, то сложность задачи удастся снизить. Один из способов упростить задачу планирования – объединить операции и данные в более крупные, для системы неделимые, объекты, т.е. выполнить агрегацию. В таком представлении алгоритма принимается всего одно решение о планировании множества агрегированных переменных (фрагментов данных) и операций (фрагментов вычислений). При этом размер и число фрагментов необходимо параметризовать для обеспечения переносимости алгоритма. Это будет первым требованием к представлению алгоритма.

Описанное выше, т.е. фрагментированное, представление алгоритма позволяет контролировать сложность задачи планирования вычислений. Однако, экспоненциальная сложность алгоритма сохраняется пусть и на меньшем числе объектов. Необходимо предпринять дополнительные меры, чтобы решение задачи планирования получить в обозримое время. Известно, что многие численные алгоритмы обладают регулярной структурой данных и вычислений. Это свойство можно использовать для уменьшения сложности задачи распределения ресурсов: одно решение по управлению может быть принято для множества однородных частей алгоритма. Такой подход кроме того позволяет провести декомпозицию задачи управления на множество более простых подзадач. Регулярность будет вторым требованием к представлению алгоритма.

Естественно полагать, что дополнительные зависимости (порядок исполнения операций), навязанные синтаксисом и семантикой языков программирования, усложняют планирование ресурсов. Таким образом, следующее требование к представлению алгоритма – отсутствие неалгоритмических зависимостей.

Перечисленные выше требования к представлению алгоритма позволяют значительно уменьшить сложность задачи планирования и сократить время, затрачиваемое на решение этой задачи. И все же, не всегда возможно делегировать решение задачи планирования ЭВМ, и будет необходимо участие программиста при распределении вычислений и данных, т.е. задача планирования должна решаться полуавтоматически. В таком случае, возникает необходимость разработки средств «ручного» управления ресурсами. При этом крайне желательно, чтобы это были средства высокого уровня, т.е. позволяли управлять фрагментами данных и вычислений.

Перечисленные выше требования к фрагментированной программе позволяют абстрагироваться от конкретной вычислительной среды. Однако, необходимо обеспечить поддержку со стороны исполнительной системы для того, чтобы максимально загрузить вычислительные мощности. Во-первых, чтобы исключить простои ресурсов, как вычислительных, так и коммуникационных, исполнение фрагментов вычислений и обмены данными должны выполняться асинхронно. Во-вторых, в процессе исполнения программа должна подстраиваться под динамику задачи и меняющуюся обстановку в вычислительной среде, т.е. должна выполняться динамическая балансировка загрузки. В-третьих, необходимо реализовать сервисные функции, такие как сбор статистики исполнения, управление контрольными точками, защита от сбоев. Все это – задачи системного программного обеспечения[3].

1.3 Обзор средств параллельного программирования и библиотек

Ниже следует обзор программных систем для суперкомпьютеров, в которых частично используются перечисленные принципы построения параллельной программы из исходного алгоритма. Все перечисленные системы используют фрагментированное представление алгоритма, операции над объектами алгоритма выполняются асинхронно, в некоторых обеспечена динамическая балансировка нагрузки.

1.3.1 Charm++

Charm++ - параллельный объектно-ориентированный язык на базе c++. Язык поддерживает высокую производительность ПП на многих аппаратных платформах. Язык разработан с целью увеличить эффективность программирования за счет использования высокоуровневых абстракций. Система программирования Charm++ включает в себя язык Charm++, компилятор, runtime-систему, а также множество подключаемых библиотек, реализующих системные функции. Система предназначена для создания ПП в системах с распределенной и общей памятью, также Charm++ поддерживает исполнение программ на архитектурах Cell и Cuda.

Единицами планирования в Charm++ являются взаимодействующие объекты, называемые чарами (chare). Чары взаимодействуют друг с другом посредством передачи сообщений, обмен сообщениями обеспечивает исполнительная система. Сообщение вызывает синхронное исполнение метода внутри чара.

В основе Charm++ лежат многие из сформулированных выше принципов: разделение задач программирования алгоритма и отображение его объектов на ресурсы вычислительной системы, фрагментированное представление алгоритма, автоматизированная поддержка эффективного исполнения на различных аппаратных платформах и некоторых сервисных функций с помощью runtime-системы. Но использование этой системы не очень удобно, т.к. используемый формат представления алгоритма является довольно низкоуровневым и требует от разработчика сложной записи в языке C++ даже сравнительно простых параллельных алгоритмов. Кроме того, в Charm++ отсутствует статическое планирование вычислений. Поэтому способы организации управления, не предусмотренные runtime-системой и библиотеками, должны быть зафиксированы в программе, что уменьшает ее переносимость [4].

1.3.2 SMP Superscalar

SMP Superscalar – программная среда, сосредоточенная на легкости создания ПП, компактности и гибкости. Система создана на базе Cell superscalar и предназначена для

программирования систем с общей памятью. SMP Superscalar состоит из расширения языка Си, компилятора и runtime-системы.

ПП в системе записывается на языке Си и аннотируется специальными директивами для компилятора. Распараллеливание в системе происходит за счет естественного параллелизма в задачах. Программа представляет собой множество заданий, в основе используемой модели программирования лежит представление программы в виде ориентированного ациклического графа (DAG – directed acyclic graph). Узлы графа представляют собой фрагменты вычислений (задания), дуги – информационные зависимости (входы и выходы), каждой из которых соответствует фрагмент данных (область памяти).

Преимуществами системы являются простота программирования, поддержка динамических свойств программы со стороны системы, фрагментированное представление алгоритма, асинхронное исполнение заданий. Однако, планирование заданий происходит во время исполнения программы прозрачно для программиста, такой подход исключает возможность статического планирования вычислений и оставляет пользователю мало средств управления. Программист имеет возможность только задавать повышенный приоритет некоторым заданиям [5].

1.3.3 Библиотеки PLASMA и DPLASMA

Библиотека PLASMA – блочная асинхронная реализация некоторых операций линейной алгебры (подмножество операций библиотеки LAPACK) для систем с общей памятью [4]. В ней используются блочные (фрагментированные) версии алгоритмов, представленные в виде ориентированного ациклического графа задач. Структура графа и размеры фрагментов данных для каждого алгоритма зафиксированы в библиотеке. В качестве фрагментов кода используются процедуры из последовательных реализаций библиотек BLAS и LAPACK.

Библиотека DPLASMA – аналог библиотеки PLASMA для распределенной памяти [5]. Алгоритмы в этой библиотеке представлены в виде распределенного ориентированного ациклического графа, который исполняется с помощью специальной runtime-системы. Runtime-система обеспечивает эффективное асинхронное исполнение заданной графом программы со статическим распределением фрагментов вычислений по узлам. На текущий момент проект находится в начальной стадии, и в системе отсутствует автоматическое распределение ресурсов. Тем не менее, даже при неоптимальном способе

распределения ресурсов при использовании 5 тыс. ядер библиотека показывает хорошую эффективность (на уровне ScaLapack) [6].

1.3.4 Технология фрагментированного программирования

Во всех системах рассмотренных выше системах параллельного программирования и библиотеках используется фрагментированное представление алгоритма. Разделение данных и вычислений алгоритма на фрагменты требуется для двух целей:

- Эффективное использование кэш-памяти (PLASMA, SMP Superscalar).
- Выделение неделимых блоков данных и работ для распределения по ресурсам, передачи между узлами в процессе счета и динамической балансировки (PLASMA, Charm++, SMP Superscalar).

Таким образом, фрагментированное представление алгоритма является необходимым для его реализации на современных параллельных вычислительных системах. В ИВМиМГ в Лаборатории синтеза параллельных программ ведется разработка технологии фрагментированного программирования, в которой принцип высокоуровневого фрагментированного представления алгоритма взят за основу [7-9].

Программа в рамках ТФП – это представление алгоритма, удовлетворяющее перечисленным выше требованиям. Фрагментированная программа в ТФП является совокупностью множеств:

- множества фрагментов данных, определенных как агрегированные переменные единственного присваивания,
- множества фрагментов вычислений, определенных как агрегированные операции единственного срабатывания,
- отношения частичного порядка на множестве фрагментов вычислений.

Алгоритм в таком виде является переносимым. Его применение на различных вычислительных системах будет отличаться только исполняемым кодом фрагментов вычислений, размером фрагментов и способом распределения ресурсов. Причем, распределение ресурсов, как уже говорилось, должно выполняться полуавтоматически.

В рамках развития ТФП также ведется разработка экспериментальной системы фрагментированного программирования LuNA. Сейчас она включает следующие компоненты:

1. язык фрагментированного программирования LuNA,
2. компилятор,
3. планировщик,

4. runtime-система.

Алгоритм в СФП LuNA имеет следующее представление. Алгоритм – это четверка < X, O, In, Out >, где:

- X – множество переменных,
- O – множество операций,
- In: X → O – отношение «вход»,
- Out: X → O – отношение «выход».

Алгоритм – двудольный ориентированный ациклический граф с вершинами двух видов: переменные единственного присваивания и операции единственного срабатывания (тождественно определению вычислимой функции Клини). Дуги графа обозначают отношения «вход-выход».

В системе LuNA используется следующая модель исполнения алгоритма. Каждая переменная модели может находиться в двух состояниях: означенном (ей сопоставлено конкретное значение) и неозначенном (конкретное значение не сопоставлено). Перед началом работы алгоритма некоторые переменные получают значения. В процессе работы алгоритма каждая операция может сработать независимо от других, если все ее входные переменные получили значения. В результате срабатывания операции значения получают ее выходные переменные. Исполнение завершается, если ни одна операция больше не может сработать. Чтобы избежать неоднозначности при означивании переменных, введем следующее «правило уникальности»: только одна операция может иметь какую-либо переменную в качестве выходной.

Для представления более сложных численных алгоритмов, базовая модель представления алгоритма расширяется набором средств:

- условные операции – необходимы для представления алгоритмов с ветвлениями,
- структурированные операции (аналог подпрограмм в процедурных языках программирования) – необходимы для иерархического описания алгоритма и повторного использования фрагментов алгоритма.
- массивы переменных и операций – необходимы для компактного представления алгоритмов большого и потенциально бесконечного размера.

1.4 Постановка задачи

Целью работы является:

- разработка фрагментированного алгоритма симплекс-метода в представлении для СФП LuNA,
- разработка фрагментированной программы на основе разработанного алгоритма.

Для достижения поставленных целей были сформулированы следующие задачи:

- выбрать алгоритм симплекс-метода,
- выделить в алгоритме фрагменты данных и вычислений,
- записать алгоритм в представлении для СФП LuNA,
- реализовать фрагменты данных в виде отдельных блоков, а фрагменты вычислений – в виде отдельных подпрограмм,
- организовать запуск фрагментов вычислений над фрагментами данных в порядке, определяемым соответствии разработанным алгоритмом.

2 Разработка фрагментированного алгоритма симплекс-метода

2.1 Задача линейного программирования (ЛП)

Линейное программирование (ЛП) относится к классу экстремальных задач с линейными ограничениями и линейной целевой функцией. Существует три постановки задачи линейного программирования, между которыми существует взаимно однозначное соответствие: каноническая форма, стандартная форма и обобщенная. Т.к. решение задачи ЛП использует каноническую постановку задачи для решения симплекс-методом, рассмотрим ее ниже.

$$\omega(x) = (c, x) \to \max \tag{1}$$

$$Ax = b \tag{2}$$

$$x \geq 0 \tag{3}$$

где $c = \{c_j\} \in R^n$, $x = \{x_j\} \in R^n$, $A = \{a_{i,j}\} \in R^{m \times n}$, $b = \{b_j\} \in R^m$, $m < n$, $rang(A) = m$.

Основное отличие в разных формах постановки задачи заключается в типе ограничений. Стандартная форма предусматривает ограничения-неравенства, а обобщённая содержит как ограничения-равенства, так и ограничения-неравенства. Предполагается, что система ограничений совместна и неизбыточна. Переход от одной формы представления, к другой рассмотрен в [2].

Естественно, не все задачи ЛП являются задачами на максимум, однако, несложно свести задачу нахождения минимума целевой функции к задаче поиска максимума и наоборот. Для этого необходимо всего лишь умножить коэффициенты при переменных в целевой функции на -1.

Рассмотрим критерий разрешимости задачи линейного программирования. Прежде, чем перейти к непосредственному обсуждению данного критерия, введем определения.

Определение 1. Множество вещественных чисел является множеством допустимых решений, если оно удовлетворяет условиям (2), (3).

Следствие 1.1. Если система (2) несовместна, то множество допустимых решений пусто.

Определение 2. Если целевая функция в задаче ЛП ограничена снизу (сверху) на множестве допустимых решений, то она принимает минимальное (максимальное) значение на данном множестве.

Следствие 2.1. Если целевая функция не ограничена снизу (сверху) на множестве допустимых решений, то задача минимизации (максимизации) целевой функции неразрешима.

Критерий разрешимости задачи ЛП. Задача ЛП разрешима, если множество допустимых решений не пусто и целевая функция ограничена на нем.

2.2 Симплекс-метод

Предложенный подход решения задачи ЛП симплекс-методом был разработан Джорджем Данцигом в 1947 году. Симплекс-метод представляет собой совокупность алгоритмов, которые можно объединить по следующему признаку. В основу метода положен целенаправленный перебор значений из области допустимых значений. Было показано [2], что оптимальное решение лежит в угловых точках многогранника области допустимых значений (2), (3). Каждой угловой точке многогранника (2), (3) соответствует базисное допустимое решение (БДР). Перебор всех БДР является алгоритмом с экспоненциальной оценкой. Однако, если осуществлять перебор базисных решений в сторону неувеличения (неуменьшения) значения целевой функции, то можно значительно сократить число операций.

Понятие БДР требует пояснения. Компоненты БДР делятся на базисные и небазисные. Базис составляют линейно независимые столбцы матрицы ограничений и размер базиса определяется по рангу матрицы условий, соответственно. В БДР небазисные компоненты являются нулями, а на местах базисных компонент стоят неотрицательные значения. Если среди базисных компонент встречаются нули, то такое БДР называется вырожденным, одному и тому вырожденному решению может соответствовать несколько различных базисов [2].

Симплекс–метод состоит из трех основных этапов:

- определения некоторого первоначального допустимого базисного решения задачи;
- проверки оптимальности найденного решения;
- правила перехода к следующему не худшему допустимому базисному решению, если текущее БДР неоптимально.

Чтобы решать задачу ЛП алгоритмами симплекс-метода, необходимо выбрать базис и поставить ему в соответствие БДР. Как говорилось ранее, базис – множество

линейно независимых столбцов, которые, по сути, являются квадратной невырожденной матрицей. Тогда все множество столбцов исходной матрицы A можно разделить на два непересекающихся подмножества базисных – B и небазисных столбцов N: $A=[B|N]$. Таким же образом можно выделить базисные и небазисные компоненты вектора коэффициентов целевой функции: $c=[c_B|c_N]$. Как уже было сказано, БДР на местах небазисных компонент имеет нулевые значения, а на местах базисных – неотрицательные: $x=[x_B|x_N]$, $x_N=0$, $x_i\geq 0$, $i\in I_B$. Тогда текущее БДР можно найти из равенства: $B\cdot x_B=b$.

Прежде, чем перейти к обсуждению критерия оптимальности, необходимо ввести некоторые понятие. Вектор относительных оценок выражается следующим соотношением. Базисные компоненты вектора нулевые, небазисные вычисляются по формуле: $d=c_B\cdot B^{-1}\cdot N-c_N$. Тогда, критерием оптимальности является отсутствие отрицательных компонент вектора относительных оценок. С другой стороны, наличие отрицательных компонент указывает на то, что значение целевой функции можно улучшить, а номера отрицательных компонент соответствуют столбцам, которые можно ввести в базис, чтобы уменьшить (увеличить) значение целевой функции.

Переход к новому базису осуществляется заменой некоторого столбца из базисного множества на выбранный ранее столбец из небазисного множества. Столбец, который необходимо вывести из базиса, чтобы улучшить значение целевой функции, определяется следующим образом. Сначала необходимо получить представление вводимого столбца, пусть его номер q, в текущем базисе: $\alpha_q=B^{-1}\cdot a_q$. Если среди компонентов вектора α_q нет положительных, это свидетельствует о том, что задача ЛП неразрешима из-за неограниченности целевой функции. Если же среди компонент находятся положительные, то номер p выводимого из базиса столбца определяется соответствующей компонентой, которая дает минимальное отношение $\frac{\alpha_{q,i}}{x_{B,i}}, i=1..m$. После перехода к новому базису повторяются указанные процедуры до тех пор, пока не будет найдено оптимальное решение или установлено, что целевая функция не ограничена на данной системе ограничений.

2.3 Фрагментация наивного алгоритма симплекс-метода

2.3.1 Описание алгоритма

Наивный алгоритм симплекс-метода появился в 1947. С тех пор алгоритм претерпел модификации и дал начало другим алгоритмам симплекс-метода. Наивный алгоритм изучается в университетах в рамках дисциплин, посвященных оптимизации, а также используется в ПО для решения задач линейной оптимизации, например, в пакете glpk (GNU Linear Programming Kit).

Для решения задачи ЛП наивным алгоритмом симплекс-метода исходные данные задачи приводятся к специальному виду – симплекс-таблице. Все операции производятся над симплекс-таблицей, в ходе решения задачи используются понятия ведущей строки, ведущего столбца и ведущего элемента, которые будут пояснены непосредственно в описании алгоритма.

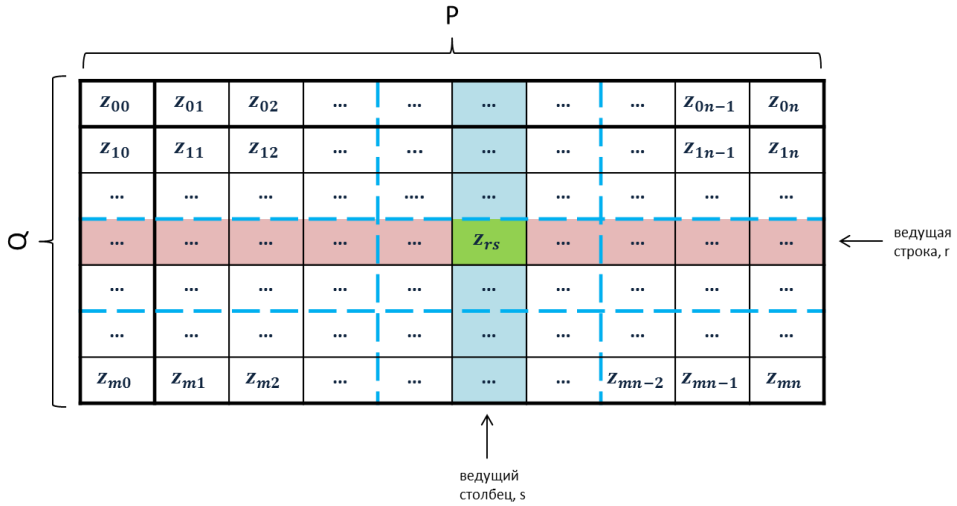

Рис.1. Симплекс-таблица

Зависимость между элементами симплекс-таблицы и исходными данными выражается следующими соотношениями:

$$z_{0,0} = -c_B B^{-1} b$$

$$(z_{1,0},\ldots,z_{m,0})^T = B^{-1} b$$

$$z_{0,j} = c_j - c_B B^{-1} A_j,\ j = 1..n$$

$$(z_{i,j},\ldots,z_{m,j})^T = B^{-1} A_j,\ j = 1..n$$

где c_B - вектор, состоящий из базисных компонент вектора c, B – матрица, составленная из базисных столбцов матрицы условий A.

Ограничения на приведение исходных данных к требуемому виду не накладываются. Однако, зачастую для этих целей решается дополнительная задача минимизации суммы искусственных переменных; при этом способ приведения данных дополнительной задачи к виду симплекс-таблицы достаточно прост [10].

Наивный алгоритм симплекс-метода приведен ниже.

1. Найти ведущий столбец:
   a. Если невозможно найти $d_j < 0$, то конец (оптимальное решение найдено).
   b. Иначе найти номер ведущего столбца s, такого что $z_{0,s} = \min z_{0,j} < 0, j = 1..n$
2. Найти ведущую строку:
   a. Если невозможно найти ни одного неотрицательного элемента ведущего столбца, то конец (целевая функция не ограничена на данной системе ограничений.
   b. Иначе найти номер ведущей строки r, такой что

   $$\frac{z_{r,s}}{x_r} = \min \frac{z_{i,j}}{x_i} > 0, i = 1..m, j = 1..n.$$

3. Преобразовать симплекс-таблицу по следующему правилу:

$$\begin{cases} z'_{i,j} = z_{i,j} - \dfrac{z_{i,s} z_{r,j}}{z_{r,s}}, i \neq r, i = 0..m-1, j = 0..n-1 \\ z'_{r,j} = \dfrac{z_{r,j}}{z_{r,s}}, j = 0..n-1 \end{cases}$$

Перейти к шагу 1.

2.3.2 Анализ алгоритма

Данный алгоритм содержит большое число независимых однотипных операций, а также редукционные операции, которые можно эффективно исполнить параллельно.

Для того, чтобы сделать явным параллелизм в операциях выбора ведущей строки и ведущего столбца, которые представляют собой редукционные операции, представим их в

следующем виде. Поделим область входных данных операции на части, а задачу поиска ведущей строки (столбца) представим в виде 2 последовательных подзадач:

1. выбор претендента на ведущий столбец (строку) локально внутри каждой части;
2. выбор среди претендентов.

Теперь очевидно, что выбор претендентов внутри отдельных частей – множество независимых операций, и можно данные операции исполнить параллельно.

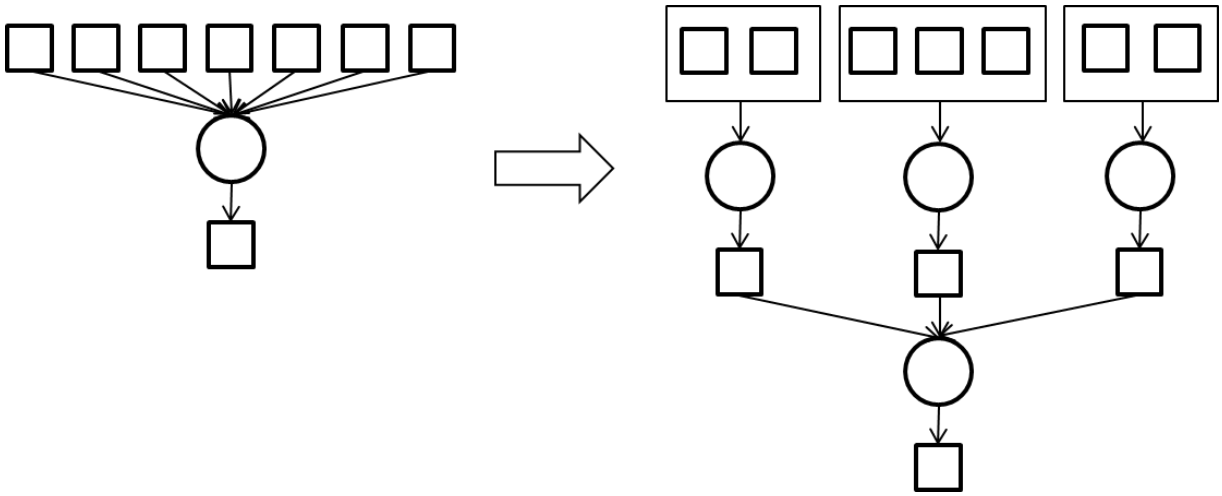

Рис.2 Фрагментация редукционных операций

Самый ресурсоемкий этап наивного алгоритма – пересчет симплекс-таблицы – представляет собой множество однотипных независимых операций над множеством элементов симплекс-таблицы, которые могут быть исполнены параллельно.

Наивный алгоритм симплекс-метода представляет собой точный метод решения задачи ЛП. Однако, во время работы алгоритма на ЭВМ накапливается ошибка округления..

Один из способов борьбы с накоплением ошибки округления повышение точности вычислений – это использование типа данных, позволяющего представлять вещественные числа с большей точностью. Однако, точность вычислений не может расти сколь угодно долго и имеет предел, определяемый производительностью. Следовательно, существует задача оптимизации такой размерности, для которой предельной точности окажется недостаточно. Другой способ борьбы с накоплением ошибки округления – периодический пересчет симплекс-таблицы из исходных данных. Оба предложенных способа достаточно ресурсоемки, поэтому было принято решение не использовать наивный алгоритм для решения оптимизационных задач большой размерности.

2.3.3 Фрагментация алгоритма

На основании анализа алгоритма симплекс-метода были выделены следующие операции, требующие фрагментации:

1. выбор ведущего столбца;
2. выбор ведущей строки;
3. пересчет симплекс-таблицы.

Для фрагментации базового алгоритма был выбран следующий способ фрагментации: разделение симплекс-таблицы и связанного с ней множества операций по вертикали и горизонтали, а также выделение ведущих столбца, строки и элемента в отдельные фрагменты. Предложенный способ организации данных позволяет провести агрегацию операций, ассоциированных с фрагментом.

Недостатком наивного алгоритма симплекс-метода является наличие нелокальных операций, однако, они представляют собой операции редукции и коллективные взаимодействия, для которых существует эффективная реализация, например, в библиотеке MPI. При агрегации переменных соответствующие им операции тоже агрегируются и фрагментированный алгоритм сохраняет структуру исходного алгоритма, но с меньшим количеством фрагментов вычислений и фрагментов данных.

Ввиду того, что предложенный фрагментированный алгоритм учитывает особенности вычислителя через параметры, фрагментированная программа может быть эффективно исполнена при увеличении размера задачи, т.е. обладает свойством масштабируемости.

Таким образом, полученный фрагментированный наивный алгоритм симплекс-метода сохраняет структуру базового алгоритма и позволяет эффективно реализовать его на мультикомпьютерах.

Приведем фрагментированный алгоритм симплекс-метода в терминах фрагментированного программирования, т.е. опишем фрагменты данных, фрагменты вычислений и определим между ними отношения «вход-выход».

Фрагменты вычислений:

1. Имя: Duality[k], k = 0,1,2,…

   Вход: $z0_j[k]$ (j=0…Q-1, k = 0,1,2,…)

   Выход: Dual[k] , k = 0,1,2,…

   Каждый фрагмент вычислений Duality[k] принимает на вход совокупность

фрагментов нулевой строки симплекс-таблицы $z0_j[k]$ и порождает булев фрагмент вычислений Dual[k].

2. Имя: ExitDual[k]

   Вход: Dual[k]

   Выход: пустой фрагмент

   Фрагмент вычислений ExitDual[k] принимает на вход булев фрагмент вычислений Dual[k]. Если значение Dual[k] == true, то операция инициирует завершение программы.

3. Имя: LeadCol[k]

   Вход: Dual[k], $z_{i,j}[k]$ (i=0…P-1, j=0…Q-1)

   Выход: s[k], $S_i[k]$ (i=0…P-1)

   Фрагмент вычислений Lead Col[k] принимает на вход булев фрагмент вычислений Dual[k] и совокупность фрагментов симплекс-таблицы $z_{i,j}[k]$, на выходе – фрагменты вычислений s[k] (структура с информацией о ведущем столбце: глобальный индекс, номер фрагмента, содержащего ведущий столбец, локальный индекс внутри фрагмента) и совокупность фрагментов $S_i[k]$, содержащих ведущий столбец.

4. Имя: Limit[k]

   Вход: $S_i[k]$ (i=0…P-1)

   Выход: Lim[k]

   Фрагмент вычислений Limit[k] принимает на вход совокупность фрагментов $S_i[k]$, содержащих ведущий столбец, и порождает булев фрагмент вычислений Lim[k].

5. Имя: ExitLim[k]

   Вход: Lim[k]

   Выход: пустой фрагмент

   Фрагмент вычислений ExitLim[k] принимает на вход булев фрагмент вычислений Lim[k]. Если значение Lim[k] == false, то операция инициирует завершение программы.

6. Имя: LeadRow[k]

   Вход: Lim[k], $z_{i,j}[k]$, $x_i[k]$ (i=0…P-1, j=0…Q-1)

   Выход: r[k], $R_j[k]$, Zrs[k] (j=0…Q-1)

   Фрагмент вычислений Lead Row[k] принимает на вход булев фрагмент вычислений Lim[k], совокупность фрагментов симплекс-таблицы $z_{i,j}[k]$ и совокупность

фрагментов, содержащих нулевой столбец, $x_i[k]$, на выходе – фрагменты вычислений r[k] (структура с информацией о ведущей строке: глобальный индекс, номер фрагмента, содержащего ведущую строку, локальный индекс внутри фрагмента), совокупность фрагментов $R_i[k]$, содержащих ведущую строку, и фрагмент Zrs[k] содержащий значение ведущего элемента симплекс-таблицы.

7. Имя: Transform[k]

   Вход: $z_{i,j}[k]$, $z0_j[k]$, $x_i[k]$, r[k], s[k], R_j [k], $S_i[k]$, Zrs[k] (i=0…P-1, j=0…Q-1)

   Выход: $z_{i,j}[k+1]$, $z0_j[k+1]$, $x_i[k+1]$ (i=0…P-1, j=0…Q-1)

   Фрагмент вычислений выполняет переход от текущей итерации k к следующей k+1-ой, на данном этапе происходит пересчет симплекс-таблицы в зависимости от ведущих строки, столбца и элемента.

Подробная схема алгоритма приведена в Приложении А.

2.4 Фрагментация модифицированного алгоритма симплекс-метода

2.4.1 Описание алгоритма

Рассмотрим также модифицированный алгоритм симплекс-метода, который чаще применяется на практике для решения задач ЛП. Для описания алгоритма дополнительных определений вводить не нужно. Исходными данными для алгоритма служат матрица системы ограничений и вектор-столбец правых частей (2), а также коэффициенты при целевой функции (1). Предполагается, что до начала работы алгоритма множество переменных поделено на базисные и небазисные, при этом базисное решение является допустимым [11].

1. Вычислить вектор относительных оценок для небазисных столбцов:
   a. Найти π: $\pi \cdot B = cB$
   b. Вычислить d: $d = \pi^T \cdot NB - cN$
2. Выбрать столбец, вводимый в базис:
   a. Если нет ни одной отрицательной относительной оценки, то оптимальное решение получено. Конец.
   b. Иначе найти номер q вводимого в базис столбца, такого что: $d_q = \min d_j < 0, j = 0..M-1.$
3. Найти вектор xB: $B \cdot xB = b$.

4. Вычислить представление столбца, вводимого в базис, в текущем базисе: $\alpha_q = B^{-1}a_q$.
5. Найти столбец, выводимый из базиса:
    a. Если невозможно найти $B_{i,p} > 0$, то конец (целевая функция не ограничена).
    b. Иначе найти номер p выводимого из базиса столбца, такого что
    $$\frac{B_{p,j}}{xB_p} = \min \frac{B_{i,j}}{xB_i} > 0, i = 0..M-1$$
6. Операция исключения: на место выводимого из базиса столбца подставляется вводимый.

2.4.2 Анализ алгоритма

Модифицированный алгоритм имеет сходство с наивным алгоритмом: операции выбора вводимого и выводимого из базиса элемента являются полными аналогами выбора ведущих строки и столбца соответственно. Выше уже были проанализированы данные операции, и более останавливаться на них не будем.

Однако, имеющиеся в алгоритмах различия существенны для способа фрагментации. В модифицированном алгоритме присутствует операция решения систем линейных алгебраических уравнений (СЛАУ), а также матричные операции умножения и алгебраической суммы.

Обсудим подробнее матричные операции. Операция суммирования матриц представляет собой совокупность независимых операций, отношения «вход-выход» для каждой из которых жестко фиксированы. В результате разделение на фрагменты данных и операций производится взаимно однозначно.

Операция матричного умножения представляет собой множество независимых операций умножения и редукционных операций суммирования. Способ фрагментации редукционных операций был продемонстрирован выше и содержит два этапа. В данном случае первый этап – вычисление частичных сумм, второй этап – их суммирование.

В ходе анализа современных библиотек, реализующих симплекс-метод, было решено использовать LU-разложение для решения СЛАУ. Пусть СЛАУ записана в матричном виде: Fx = g.Тогда необходимо выполнить следующие шаги, чтобы найти решение СЛАУ x.

1. Найти LU-разложение основной матрицы системы: F = LU;
2. Найти решение системы уравнений Ly = g;

3. Найти решение исходной задачи из системы уравнений Ux = y.

Подходы к распараллеливанию LU-разложения и решения СЛАУ с треугольной матрицей существуют уже давно [12-14]. Это дает возможность использовать существующие методы для фрагментации данного вычислительного этапа.

Необходимо учесть, что LU-разложение выполняется с точностью до перестановки: F = P·L·U·Q. Тогда решение СЛАУ Fx = g с помощью LU-разложения с учетом перестановок имеет следующий вид:

$$x = Q^{-1}\cdot(U^{-1}\cdot y)$$

$$y = L^{-1}\cdot(P^{-1}\cdot g)$$

Здесь умножение вектора на перестановочную матрицу P^{-1} или Q^{-1} означает выполнение соответствующей перестановки элементов вектора, а умножение на треугольную матрицу U^{-1} или L^{-1} – решение соответствующей системы уравнений. Таким образом, получается следующий алгоритм решения СЛАУ с произвольной матрицей:

1. Перестановка элементов вектора g в соответствии с перестановочной матрицей P^{-1}. В результате получаем вектор g'.
2. Решение СЛАУ с нижней треугольной матрицей Ly=g’.
3. Решение СЛАУ с верхней треугольной матрицей Ux’=y.

Перестановка элементов вектора x’ в соответствии с перестановочной матрицей Q^{-1}. В результате получаем вектор x.

В отличие от наивного алгоритма, модифицированный алгоритм принимает на вход систему ограничений в каноническом виде, т.е. дополнительных преобразований исходных данных относительно вида (2) не требуется.

Модифицированный алгоритм симплекс-метода является точным методом. При исполнении алгоритма на ЭВМ ошибка округления накапливается, но существует возможность на любой итерации ее устранить, если выбрать базис из исходных данных и заново вычислить для него LU-разложение. Накопление ошибки округления зависит от способа перехода к новому базису, а если быть точнее, к его LU-разложению (т.к. при вычислениях фактически используется LU-разложение).

2.4.3 Фрагментация алгоритма

Рассмотрим фрагментацию некоторых операций отдельно.

Фрагментация матричных операций

Суммирование. Исходная операция: C = A + B, где A, B, C – матрицы размера M*N. Необходимо выполнить фрагментацию. Матрицы фрагментируются разрезанием по

горизонтали и вертикали. Пусть P и Q – параметры, реализующие число фрагментов по горизонтали и вертикали соответственно. Тогда фрагменты данных можно описать следующим образом: $\{A_{x,y}\}$, $\{B_{x,y}\}$, $\{C_{x,y}\}$ – множества фрагментов матриц A, B, C соответственно, x=0..P-1, y=Q-1.

Назовем фрагментированные операции суммирования Sum, которая представляет собой множество фрагментов вычислений суммирования.

Sum (in: $\{A_{x,y}\}$, $\{B_{x,y}\}$; out: $\{C_{x,y}\}$; x=0..P-1, y=0..Q-1)

{

$Sum_{x,y}$ (in: $A_{x,y}$, $B_{x,y}$; out: $C_{x,y}$) | x=0..P-1, y=0..Q-1

}

Фрагменты вычислений $Sum_{x,y}$ реализуют функцию матричного суммирования, принимая на вход фрагменты данных $A_{x,y}$, $B_{x,y}$ и выдавая на выходе фрагмент $C_{x,y}$.

Произведение. Исходная операция: C = A·B, где A, B, C матрицы размера M*N, N*L, M*L соответственно. Пусть параметры P, Q и R реализуют число фрагментов следующим образом: матрица A делится на P фрагментов по горизонтали и Q фрагментов по вертикали, матрица B делится на Q фрагментов по горизонтали и R фрагментов по вертикали и матрица C делится на P фрагментов по горизонтали и R фрагментов по вертикали. Тогда фрагменты данных можно описать следующим образом: $\{A_{x,y}\}$, $\{B_{y,z}\}$, $\{C_{x,z}\}$ – множества фрагментов матриц A, B, C соответственно, x=0..P-1, y=0..Q-1, z=0..R-1.

Mult (in: $\{A_{x,y}\}$, $\{B_{y,z}\}$; out: $\{C_{x,z}\}$; x=0..P-1, y=0..Q-1, z=0..R-1)

{

$pmul_{x,y,z}$ (in: $A_{x,y}$, $B_{y,z}$; out: $D_{x,y,z}$) | x=0..P-1, y=0..Q-1, z=0..R-1;

$psum_{x,z}$ (in: { $D_{x,y,z}$ | y=0..Q-1 }; out: $C_{x,z}$) | x=0..P-1, z=0..R-1;

}

Фрагменты вычислений $pmul_{x,y,z}$ реализуют операцию матричного умножения входных фрагментов $A_{x,y}$, $B_{y,z}$ в выходной фрагмент $D_{x,y,z}$. Затем $psum_{x,z}$ редуцирует (суммирует) фрагменты $D_{x,y,z}$ по координате y в входной фрагмент $C_{x,z}$.

Фрагментация LU-разложения

Необходимо фрагментировать алгоритм разложения произвольной матрицы на произведение нижней и верхней треугольных матриц.

Для того, чтобы определить способ фрагментации, рассмотрим зависимости между исходной матрицей и ее LU-разложением. Пусть матрицы A, L и U фрагментированы

согласованным образом (рис. 3). Тогда зависимости между их фрагментами без учета перестановок выражаются соотношением (4).

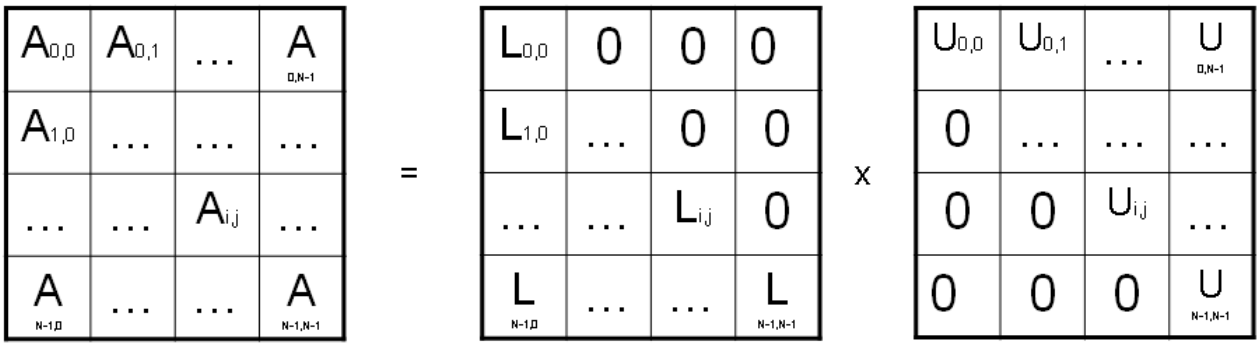

Рис.3 Фрагментированное LU-разложение

$$A_{i,j} = \begin{cases} \sum_{k=0}^{i} L_{i,k} \cdot U_{k,j}, i < j \\ \sum_{k=0}^{j} L_{i,k} \cdot U_{k,j}, i \geq j \end{cases} \quad i,j = \overline{0, N-1} \tag{4}$$

На основании формулы (4) с учетом перестановок можно предложить следующий фрагментированный алгоритм LU-разложения квадратной матрицы:

LU (in: {$A_{i,j}$ | i=0..N-1, j=0..N-1}; out: {$L_{i,j}$ | i=0..N-1, j=0..N-1}, {$U_{i,j}$ | i=0..N-1, j=0..N-1}, {P_i | i=0..N-1}, {Q_j | j=0..N-1})

{

assign(in: $A_{0,0}$; out: $A'_{0,0}$);

assignL_i(in: $A_{i,0}$; out: $A'_{i,0}$) | i=1..N-1;

assignU_i(in: $A_{0,i}$; out: $A'_{0,i}$) | i=1..N-1;

diag$_i$(in: $A'_{i,i}$; out: $L_{i,i}$, $U_{i,i}$, P_i, Q_i) | i=0..N-1;

solve$L_{i,j}$(in: $A'_{i,i}$, P_i, $A_{j,i}$; out: $L'_{j,i}$) | i=0..N-1, j=i+1..N-1;

solve$U_{i,j}$(in: $A'_{i,i}$, Q_i, $A_{i,j}$; out: $U'_{i,j}$) | i=0..N-1, j=i+1..N-1;

permut$L_{i,j}$(in: $L'_{i,j}$, P_i; out: $L_{i,j}$) | i=1..N-1, j=0..i-1;

permut$U_{i,j}$(in: $U'_{j,i}$, Q_i; out: $U_{j,i}$) | i=1..N-1, j=0..i-1;

mul$D_{i,j}$(in: $L'_{i,j}$, $U'_{j,i}$; out: $MD_{i,j}$) | i=1..N-1, j=0..i-1;

sumD_i(in: { $MD_{i,j}$ | j=0..i-1 }; out: SD_i) | i=1..N-1;

subD_i(in: $A_{i,i}$, SD_i; out: $A'_{i,i}$) | i=1..N-1;

$mulL_{i,j,k}$(in: $L'_{i,k}$, $U'_{k,j}$; out: $ML_{i,j,k}$) | i=1..N-1, j=1..i-1, k=0..j-1;

$sumL_{i,j}$(in: { $ML_{i,j,k}$ | k=0..j-1 } ; out: $SL_{i,j}$) | i=1..N-1, j=1..i-1;

$subL_{i,j}$(in: $A_{i,j}$, $SL_{i,j}$; out: $A'_{i,j}$) | i=1..N-1, j=1..i-1;

$mulU_{j,i,k}$(in: $L'_{j,k}$, $U'_{k,i}$; out: $MU_{j,i,k}$) | i=1..N-1, j=1..i-1, k=0..i-1;

$sumU_{j,i}$(in: {$MU_{j,i,k}$ | k=0..i-1} ; out: $SU_{j,i}$) | i=1..N-1, j=1..i-1;

$subU_{j,i}$(in: $A_{i,j}$, $SU_{j,i}$; out: $A'_{j,i}$) | i=1..N-1, j=1..i-1;

}

Данный алгоритм содержит следующие фрагменты вычислений:

- assign выполняет операцию присвоения фрагменту $A'_{0,0}$ элементов фрагмента $A_{0,0}$;
- $assignL_i$ и $assignU_i$ выполняют операцию присвоения фрагментам $A'_{i,0}$ элементов фрагмента $A_{i,0}$ и фрагментам $A'_{0,i}$ элементов фрагмента $A_{0,i}$ соответственно;
- $diag_i$ выполняет LU-разложение диагонального фрагмента $A'_{i,i}$;
- $solveL_{i,j}$ и $solveU_{i,j}$ – поиск $L'_{j,i}$ и $U'_{i,j}$ с учетом перестановок соответственно;
- $permutL_{i,j}$ и $permutU_{i,j}$ выполняют перестановки P_i и Q_i внутри фрагментов $L'_{i,j}$ и $U'_{j,i}$ соответственно;
- $mulD_{i,j}$, $mulL_{i,j,k}$, $mulU_{j,i,k}$ выполняют умножение (слева направо) входных фрагментов .
- $sumD_i$, $sumL_{i,j}$, $sumU_{i,j}$ выполняют редукцию (суммирование) входных фрагментов
- $subD_i$, $subL_{i,j}$, $subU_{j,i}$ реализует поиск фрагментов $A'_{i,i}$, $A'_{i,j}$, $A'_{j,i}$ соответственно.

Фрагментация решения СЛАУ с треугольной матрицей

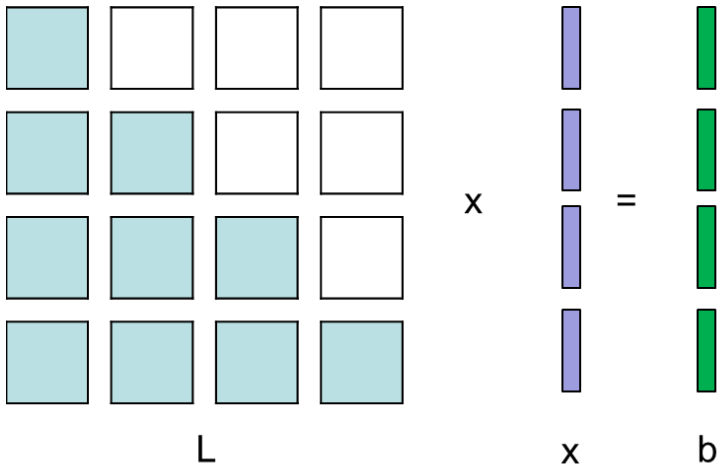

Рис.4 Фрагментированная СЛАУ Lx = b

Исходная система уравнений для фрагментации Lx = b, где L – нижняя треугольная матрица. Можно предложить следующий фрагментированный алгоритм решения СЛАУ с нижней треугольной квадратной матрицей (матрица фрагментирована на N фрагментов и по вертикали, и по горизонтали, векторы x, b также фрагментированы на N фрагментов):

$L\_GE(in: \{L_{i,j} \mid i=0..N-1, j=0..N-1\}, \{b_i \mid i=0..N-1\}; out: \{x_j \mid j=0..N-1\})$

{

$assignb(in: b_0; out: b'_0)$

$fr\_L\_GE_i(in: L_{i,i}, b'_i; out: x_i) \mid i = 0..N-1;$

$mult_{i,j}(in: L_{i,j}, x_j; out: Lx_{i,j}) \mid i=0..N-1, j=0..i-1;$

$sum_i(in: \{Lx_{i,j} \mid j=0..i-1 ; out: SumLx_i) \mid i=1..N-1;$

$subb_i(in: b_i, SumLx_i; out: b'_i) \mid i=1..N-1;$

}

Данный алгоритм содержит следующие фрагменты вычислений:

- assign выполняет операцию присвоения фрагменту b'_0 элементов фрагмента b_0
- fr_L_GE выполняет решение СЛАУ вида $L_{i,i}x_i = b'_i$
- $mult_{i,j}$ выполняет операцию $Lx_{i,j} = L_{i,j}x_j$
- sum_i выполняет редукцию (суммирование) фрагментов $Lx_{i,j}$ по координате j во фрагменты $SumLx_i$
- $subb_i$ выполняет операцию $b'_i = b_i - SumLx_i$

Подобные рассуждения можно применить и для построения алгоритма решения СЛАУ с верхней треугольной матрицей.

Фрагментация решения СЛАУ

Исходная СЛАУ для фрагментации Fx = g. Исходная матрица F делится на N*N фрагментов, вектора x и g также поделены на N фрагментов каждый.

1. Получить фрагментированное LU-разложение матрицы системы F, т.е. получить фрагментированные матрицы L, U, а также совокупность фрагментов векторов перестановки P* и Q*.
2. Получить новый вектор правой части g' посредством перестановки элементов вектора g в соответствии с оператором перестановки P^{-1*}.
3. Решить фрагментировано СЛАУ с треугольной матрицей Ly = g', получить вектор y.

4. Решить фрагментировано СЛАУ с треугольной матрицей Ux’ = y, получить вектор x’.
5. Получить вектор решения СЛАУ x посредством перестановки элементов вектора x’ в соответствии с оператором перестановки Q^{-1*}

*Понятия умножить на перестановочную матрицу и выполнить перестановку в соответствии с оператором перестановки тождественны и обозначают перестановку элементов (строк и столбцов) матриц и векторов.

Тогда фрагментированный алгоритм решения СЛАУ можно записать:

solver_eq_GE(in: $\{F_{i,j} \mid i=0..N-1, j=0..N-1\}$, $\{g_i \mid i=0..N-1\}$; out: $\{x_j \mid j=0..N-1\}$)
{
LU (in: $\{F_{i,j} \mid i=0..N-1, j=0..N-1\}$: out: $\{LF_{i,j} \mid i=0..N-1, j=0..N-1\}$,
$\{UF_{i,j} \mid i=0..N-1, j=0..N-1\}$, $\{PF_i \mid i=0..N-1\}$, $\{QF_j \mid j=0..N-1\}$);
reshuffleP_i(in: g_i, PF_i; out: g'_i) | i=0..N-1;
L_GE(in: $\{LF_{i,j} \mid i=0..N-1, j=0..N-1\}$, $\{g'_i \mid i=0..N-1\}$; out: $\{y_j \mid j=0..N-1\}$);
U_GE(in: $\{UF_{i,j} \mid i=0..N-1, j=0..N-1\}$, $\{y_i \mid i=0..N-1\}$; out: $\{x'_j \mid j=0..N-1\}$);
reshuffleQ_j (in: x'_j, PQ_j; out: x_j) | j=0..N-1;
}

Фрагментация модифицированного алгоритма симплекс-метода

На вход алгоритма подается совокупность K*K фрагментов базиса B, совокупность K*J фрагментов небазисной компоненты NB, K фрагментов вектора cB и J фрагментов вектора cN.

modified_solver (in: $\{B_{i,j} \mid i=0..K-1, j=0..K-1\}$, $\{NB_{i,s} \mid i=0..K-1, s=0..J-1\}$, $\{cB_j \mid j=0..K-1\}$,
$\{cN_s \mid s=0..J-1\}$, $\{b_i \mid i=0..K-1\}$; out: $\{xB_j \mid j=0..K-1\}$)

{

Assign(in: $\{B_{i,j} \mid i=0..K-1, j=0..K-1\}$, $\{NB_{i,s} \mid i=0..K-1, s=0..J-1\}$, $\{cB_j \mid j=0..K-1\}$,
$\{cN_s \mid s=0..J-1\}$, $\{b_i \mid i=0..K-1\}$; out: $\{B^0_{i,j} \mid i=0..K-1, j=0..K-1\}$,
$\{NB^0_{i,s} \mid i=0..K-1, s=0..J-1\}$, $\{cB^0_j \mid j=0..K-1\}$, $\{cN^0_s \mid s=0..J-1\}$, $\{b^0_i \mid i=0..K-1\}$);

Duality^k(in: $\{cN^k_s \mid s=0..J-1\}$, $\{cB^k_j \mid j=0..k-1\}$, $\{B^k_{i,j} \mid i=0..K-1, j=0..K-1\}$,
$\{NB^k_{i,s} \mid i=0..K-1, s=0..J-1\}$; out: Dual^k, q^k) | k = 0,1,2…;

{

solve_eq_GE(in: $\{B^k_{i,j} \mid i=0..K-1, j=0..K-1, k = 0,1,2…\}$,
$\{cB^k_i \mid i=0..K-1, k = 0,1,2…\}$; out: $\{pi^k_j \mid j=0..K-1, k = 0,1,2…\}$);

transpose(in: $\{pi^k_j \mid j=0..K-1, k = 0,1,2…\}$; out: $\{pi\_tr^k_j \mid j=0..K-1, k = 0,1,2…\}$);

transpose(in: $\{cN^k_s \mid s=0..J-1, k = 0,1,2…\}$; out: $\{cN\_tr^k_s \mid s=0..J-1, k = 0,1,2…\}$);

Mult(in: $\{pi\_tr^k_j \mid j=0..K-1, k=0,1,2…\}$, $\{NB^k_{i,s} \mid i=0..K-1, s=0..J-1, k=0,1,2…\}$;
out: $\{temp\_d^k_s \mid s=0..J-1, k=0,1,2…\}$);

Sum(in: $\{temp\_d^k_s \mid s=0..J-1, k=0,1,2…\}$, $\{cN\_tr^k_s \mid s=0..J-1, k = 0,1,2…\}$;
out: $\{ d^k_s \mid s=0..J-1, k=0,1,2…\}$);

Dua^k(in: $\{ d^k_s \mid s=0..J-1\}$; out: Dual^k, q^k) | k = 0,1,2…;

}

Find_xB^k (in: $\{B^k_{i,j} \mid i=0..K-1, j=0..K-1\}$, $\{b^k_i \mid i=0..K-1\}$; out: $\{xB^k_j \mid j=0..K-1\}$) |
k = 0,1,2…

{

solve_eq_GE (in: $\{B^k_{i,j} \mid i=0..K-1, j=0..K-1, k = 0,1,2…\}$,
$\{b^k_i \mid i=0..K-1, k = 0,1,2…\}$; out: $\{xB^k_j \mid j=0..K-1, k = 0,1,2…\}$);

}

Limitation^k(in: {$B^k_{i,j}$ | i=0..K-1, j=0..K-1}, q^k, {xB^k_j | j=0..K-1}; out: Lim^k, p^k) |
k=0,1,2…
{
take_q_col^k (in: {$B^k_{i,j}$ | i=0..K-1, j=0..K-1}, q^k; out: {$a\_q^k_j$ | j=0..K-1}) |
k=0,1,2…;
solve_eq_GE (in: {$B^k_{i,j}$ | i=0..K-1, j=0..K-1, k = 0,1,2…},
{$a\_q^k_i$ | i=0..K-1, k = 0,1,2…}; out: { $\alpha\_q^k_j$ | j=0..K-1, k = 0,1,2…});
limit^k (in: {$\alpha\_q^k_j$ | j=0..K-1}, {xB^k_j | j=0..K-1}; out: Lim^k, p^k) | k = 0,1,2…;
}

Rearrange_Cols^k (in: {$B^k_{i,j}$ | i=0..K-1, j=0..K-1}, {$NB^k_{i,s}$ | i=0..K-1, s=0..J-1},
{cB^k_j | j=0..K-1}, {cN^k_s | s=0..J-1}, {b^k_i | i=0..K-1};
out: {$B^{k+1}_{i,j}$ | i=0..K-1, j=0..K-1}, {$NB^{k+1}_{i,s}$ | i=0..K-1, s=0..J-1},
{cB^{k+1}_j | j=0..K-1}, {cN^{k+1}_s | s=0..J-1}, {b^{k+1}_i | i=0..K-1});

}

- Assign – инициирование переменных
- Duality^k
  - solve_eq_GE – решение СЛАУ
  - transpose – транспонирование
  - Mult – матричное умножение
  - Sum – матричная сумма
  - Dua^k – поиск номера вводимого в базис столбца по минимальному элементу входного вектора
- Find_xB^k – поиск xB^k
- Limitation^k
  - take_q_col^k – взять столбец q
  - solve_eq_GE – решение СЛАУ
  - limit^k – поиск номера выводимого из базиса столбца
- Rearrange_Cols^k – операция перехода к новому базису

3 Разработка фрагментированных программ

На основе разработанных алгоритмов были спроектирован и реализован комплекс фрагментированных программ на языке C++ с использованием библиотеки Eigen. Комплекс программ был распараллелен с использованием технологии OpenMP. Комплекс состоит из семи модулей, выделенных в соответствии с функциональным наполнением. Ниже приведен перечень модулей с кратким описанием. Далее модули будут рассмотрены более подробно.

- Constants

  Модуль содержит описания и реализацию глобальных констант, структур данных и пользовательских типов.

- Reading mps

  Модуль содержит процедуры, осуществляющие чтение файлов типа *.mps в двух вариантах: в файл и во внутреннее представление программы.

- Preprocessing

  Модуль осуществляет препроцессинг произвольных входных данных к виду, требуемому для решения задачи ЛП симплекс-методом.

- LU

  В модуле реализованы основные и вспомогательные функции для фрагментированного LU-разложения.

- GE

  В модуле реализованы основные и вспомогательные функции для решения произвольной СЛАУ на основании LU-разложения основной матрицы системы.

- Naive

  В модуле реализованы основные и вспомогательные функции для решения задачи ЛП в каноническом виде фрагментированным наивным алгоритмом симплекс-метода.

- Modified

  В модуле реализованы основные и вспомогательные функции для решения задачи ЛП в каноническом виде фрагментированным модифицированным алгоритмом симплекс-метода.

3.1 Модуль Reading mps

```
void reading_mps_in_matrices(char * filename, MatrixXd & A,
```

```
                                VectorXd & b, VectorXd & c,
                                vector<int> & type);
```

Процедура производит чтение файла *.mps во внутреннее представление программы.

```
void reading_mps_in_file(char * mps_filename, char * output);
```

Процедура производит чтение файла *.mps из файла `mps_filename` в файл `output`, данные записываются в выходной файл в формате, соответствующем внутреннему представлению программы. Выходной файл составляется по следующему шаблону:

- Число строк матрицы ограничений
- Число столбцов матрицы ограничений
- Матрица ограничений
- Вектор правой части системы ограничений
- Вектор коэффициентов при целевой функции
- Тип ограничений

3.2 Модуль Preprocessing

Масштабирование:

```
void Scaling(MatrixXd & Matr, VectorXd & Right,
             VectorXd & OF_coeff, MatrixXd & sca_Matr,
             VectorXd & sca_Right, VectorXd & sca_OF_coeff);
```

Процедура выполняет масштабирование входных данных. Цель работы процедуры – привести данные задачи к одному порядку, практически – добиться, чтобы разброс не превышал 2 порядков при условии, что в каждой строке матрицы ограничений разброс значений не превышает 2 порядков.

Канонизация системы ограничений:

```
void Canonize(MatrixXd & Matr, VectorXd & c, vector<int> & type,
              MatrixXd & extend_Matr, vectorXd & extend_c);
```

Процедура приводит систему ограничений общего вида к каноническому виду. На вход подается матрица системы ограничений и вектор коэффициентов при целевой функции, а также вектор с информацией о типе ограничений; на выходе получается расширенные

матрица системы ограничений и вектор коэффициентов при целевой функции.

Проверка на разрешимость и неизбыточность системы ограничений:

```
bool Permissible_irredandant(MatrixXd & Matr, VectorXd & Right,
                             MatrixXd & red_Matr,
                             VectorXd & red_Right,
                             vector<int> & Base,
                             vector<int> & NBase);
```

Функция возвращает значение true, если система ограничений имеет хотя бы одно решения, и false иначе. Также функция редуцирует систему ограничений до множества линейно независимых строк и выдает рекомендации по назначению базиса.

Запись результатов препроцессинга в файл:

```
void Preproc_result(MatrixXd & Matr, VectorXd & Right,
                    VectorXd & OF_coeff, vector<int> & Base,
                    vector<int> & NBase, char * filename);
```

Выходной файл составляется по следующему шаблону:

- Число строк матрицы ограничений
- Число столбцов матрицы ограничений
- Матрица ограничений
- Вектор правой части системы ограничений
- Вектор коэффициентов при целевой функции
- Вектор номеров базисных столбцов
- Вектор номеров небазисных столбцов

3.3 Модуль LU

```
template <typename _Scalar, int _Rows, int _Cols>
void fr_LU(Matrix<_Scalar, _Rows, _Cols> & matr,
           Matrix<_Scalar, _Rows, _Cols> & lu_matr,
           Matrix<int, _Rows, 1> & P,
           Matrix<int, _Rows, 1> & Q);
```

Процедура реализует LU-разложение матрицы размером _Rows*_Cols . Результат работы процедуры – LU-разложение в компактной форме lu_matr, вектора перестановок строк P и столбцов Q.

```
void LU(matrix_of_MM_fr & A, matrix_of_MM_fr & LU_Matr,
        vector_of_perm_fr & P, vector_of_perm_fr & Q);
```

Процедура реализует фрагментированное LU-разложение. Входом является матрица фрагментов A, выход – матрица фрагментов LU-разложения в компактной форме LU_Matr, фрагментированные вектора перестановок строк P и столбцов Q.

3.4 Модуль GE

```
template <typename _Scalar, int _Rows, int _Cols>
void fr_solving_eq_GE(Matrix<_Scalar, _Rows, _Cols> & lu_matr,
                      Matrix<int, _Rows, 1> & permP,
                      Matrix<int, _Cols, 1> & permQ,
                      Matrix<_Scalar, _Cols, 1> & sol_vec,
                      Matrix<_Scalar, _Rows, 1> & right_vec);
```

Процедура производит решение СЛАУ на основе LU-разложения основной матрицы системы. На вход подается lu_matr – матрица LU-разложения, перестановочные вектора permP и permQ, вектор правой части right_vec.

```
void solving_eq_GE(matrix_of_MM_fr & LU_Matr,
                   vector_of_perm_fr & P, vector_of_perm_fr & Q,
                   vector_of_M_fr & Sol, vector_of_M_fr & Right);
```

Процедура производит фрагментированное решение СЛАУ на основе LU-разложения.

3.5 Модуль Naive

```
naive_preprocessing(MatrixXd & A, VectorXd & b, VectorXd & c,
                    double ** z, double * z0, double * x,
                    vector<int> & Base, vector<int> & Nonbase);
```

Процедура приводит исходные данные задачи к виду, требуемому для решения задачи ЛП

наивным алгоритмом симплекс-метода.

```
void stage_2(double** z, double* z0, double* x, int* Base,
            int* Nonbase, int num_row, int num_col, int along_x,
            int along_y);
```

Процедура-решатель задачи ЛП фрагментированным наивным алгоритмом симплекс-метода. Параметры along_x и along_y содержат информацию о количестве фрагментов по горизонтали и вертикали соответственно.

```
bool red_duality(bool *fr_dual, N_LC **s, N_LC *NLC, int n,
                 int fr_n, fr_type_z0** Z0);
```

Функция возвращает ответ на вопрос о двойственной допустимости симплекс-таблицы, а также производит поиск ведущего столбца.

```
bool red_limit(bool *fr_lim, int m, int fr_m, fr_type_S** SS);
```

Функция возвращает ответ на вопрос об ограниченности целевой функции на данной системе ограничений.

```
void lead_row(N_LR **fr_r, N_LR *NLR, fr_type_x **xx,
              fr_type_S **SS, int fr_m, int m);
```

Процедура поиска ведущей строки.

3.6 Модуль Modified

```
bool duality(matrix_of_MM_fr & B_LU, vector_of_perm_fr & B_P,
            vector_of_perm_fr & B_Q, vector_of_M_fr & cB,
            vector_of_N_M_fr & cN, vector_of_MN_M_fr & NB,
            LC & q)
```

Функция возвращает ответ на вопрос о двойственной допустимости симплекс-таблицы, а также производит поиск вводимого в базис элемента.

```
bool limitation(matrix_of_MM_fr & B_LU, vector_of_perm_fr & B_P,
                vector_of_perm_fr & B_Q, vector_of_M_fr & a_q,
                vector_of_M_fr & xB, LR & p);
```

Функция возвращает ответ на вопрос об ограниченности целевой функции на данной системе ограничений, а также производит поиск выводимого из базиса элемента.

```
void rearrange_cols(matrix_of_MM_fr & B, matrix_of_MN_M & NB,
                    vector_of_M_fr & cB, vector_of_N_M_fr & cN,
                    vector<int> & Base, vector<int> & NBase,
                    LC & q, LR & p)
```

Процедура производит пристраивание базиса и небазиса путем обмена выбранных ранее столбцов.

```
void modified_solver(matrix_of_MM_fr & B, matrix_of_MN_M_fr & NB,
                     vector_of_M_fr & cB, vector_of_N_M_fr & cN,
                     vector<int> & Base, vector<int> & NBase,
                     vector_of_M_fr & b, vector_of_M_fr & xB)
```

Процедура-решатель задачи ЛП фрагментированным модифицированным алгоритмом симплекс-метода.

3.7 Модуль Constants

```
struct LC
{
  int No_fr;
  int No_in_fr;
  int G_No;
  float value;
};

struct LR
{
  int No_fr;
  int No_in_fr;
```

```
  int G_No;
  float value;
};
```

Структуры LC и LR хранят информацию о ведущих столбце и строке соответственно. Поле No_fr хранит номер фрагмента, в котором был найден объект; No_in_fr указывает позицию, на которой объект расположен внутри фрагмента No_fr; G_No – глобальный номер объекта; поле value содержит служебную информацию.

Фрагменты данных и их объединения моделируются с помощью встроенных средств c++, а также средствами библиотеки Eigen. Непосредственно фрагменты представлены в программе в виде объектов класса Matrix<double, _Rows, _Cols> библиотеки Eigen. Для представления векторов и матриц фрагментов используется stl контейнер vector.

4 Тестирование фрагментированного алгоритма

Тестирование проводилось в два этапа. Первый этап заключался в независимом тестировании отдельных модулей, его целью было выявление уязвимостей в каждой отдельно взятой функции. Целью второго этапа было тестирование системы в целом и соответствие работы программы логике алгоритма.

Первый этап тестирования показал работоспособность отдельно взятых модулей и корректность взаимодействия процедур в пределах одного модуля. Для поведения тестирования были синтезированы данные, соответствующие целям исследования. Например, для проверки работоспособности модуля решения СЛАУ GE были синтезированы случайные матрицы системы и векторы правых частей. Корректность работы процедур, сопряженных с алгебраическими вычислениями, проверялась сравнением результатов работы процедуры с результатами работы сторонних приложений (MathCad 14) со схожей функциональностью.

Проверка функциональности всего комплекса программ сначала проводилась на синтетических задачах линейного программирования, а затем на задачах, взятых из библиотеки NETLIB [15]. Для тестирования были выбраны задачи AFIRO.mps и BLEND.mps из библиотеки NETLIB. На них был получен ожидаемый результат с точностью 10^{-3}. На тестах большого размера результат не был достигнут из-за накопления ошибок округления, т.к. в алгоритм не была включена процедура перепостроения обратной матрицы.

Тестирование показало, что фрагментация алгоритмов выполнена корректно. В результате распараллеливания на многоядерной системе была достигнута эффективность распараллеливания 80% на 8 потоках.

ЗАКЛЮЧЕНИЕ

В работе была рассмотрена проблема автоматизации создания эффективных параллельных программ. Выполнен обзор средств параллельного программирования и библиотек. Исследования базировались на технологии фрагментированного программирования как средстве автоматического конструирования параллельных программ численного моделирования.

Основные результаты работы:

1. В рамках работ по созданию библиотеки фрагментированных численных подпрограмм были изучены и фрагментированы два алгоритма симплекс метода: наивный и модифицированный.
2. В ходе работы были отработаны приемы фрагментации численных алгоритмов, которые в дальнейшем могут быть применены для фрагментации других численных алгоритмов с целью включения в библиотеку фрагментированных численных подпрограмм.
3. Разработанные фрагментированные алгоритмы были реализованы в виде комплекса фрагментированных параллельных программ на языке C++. Программы были протестированы на ряде модельных задач и показали свою работоспособность. Эффективность распараллеливания составила 80%.

Дальнейшая работа по теме может быть расширена по следующим направлениям:

- Разработка алгоритмов отображения фрагментированных алгоритмов симплекс-метода на ресурсы мультикомпьютера.
- Создание библиотеки фрагментированных подпрограмм для решения задач линейного программирования на мультикомпьютерах.
- Разработка общей методики фрагментации численных алгоритмов.

Литература

1. S.Kireev, V.Malyshkin Fragmentation of Numerical Algorithms for Parallel Subroutines Library // The Journal of Supercomputing. Vol. 57. Number 2. 2011. pp. 161-171.
2. Victor Malyshkin, V. Fragmented Programming of Library Parallel Numerical Subroitines – In the Proceedings of the 7-th Int. conference on New Trends in Software Methodologies, Tools and Techniques, IOS Press, Vol. 193, pp. 425-430. 28-30 September, 2007, Dubai
3. В.А. Вальковский, В.Э. Малышкин. Синтез параллельных программ и систем на вычислительных моделях. – Наука, Новосибирск, 1988, 128 стр.
4. Charm++, http://charm.cs.uiuc.edu/ [Электронный ресурс]
5. SMP Superscalar, http://www.bsc.es/smpsuperscalar [Электронный ресурс]
6. PLASMA Library, http://icl.cs.utk.edu/plasma/ [Электронный ресурс]
7. Kalgin K.V., Malyskin V.E., Nechaev S.P., Tschukin G.A.: Runtime System for Parallel Execution of Fragmented Subroutines. – In Proceedings of the 9th International conference on Parallel Computing Technologies (PaCT-2007), LNCS, Vol. 4671, Springer, pp 544-552 (2007)
8. Grimshaw A.S., Weissman J.B., Strayer W.T.: Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing. – ACM Transactions on Computer Systems (TOCS), Volume 14, Issue 2, pp 139-170 (1996)
9. Malyshkin V.E., Perepelkin V.A. Optimization of Parallel Execution of Numerical Programs in LuNA Fragmented Programming System // MTPP-2010 revised selected papers, Springer, LNCS 6083 (2010), pp. 1-10.
10. Данциг, Дж. Линейное программирование, его применения и обобщения. – М.: «Прогресс», 1966. – 600с.
11. Муртаф, Б. Современное линейное программирование / Теория и практика – М.: «Мир», 1984. - 224 с.
12. Buttari, A., Langou, J., Kurzak, J., Dongarra, J. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures // Parallel Computing, Volume 35, pp. 38-53, 2009.
13. Фадеев, Д.К., Фадеева, В.Н. Вычислительные методы линейной алгебры. – М.: Наука, 1967. – 658 с.
14. Gene H. Golub, Charles F. Van Loan: Matrix Computations. John Hopkins University Press, 3rd edition (1996)

15. http://www.netlib.org/lp/data/ [Электронный ресурс]

Приложение А

(обязательное)

Графическое представление фрагментированного наивного алгоритма симплекс-метода

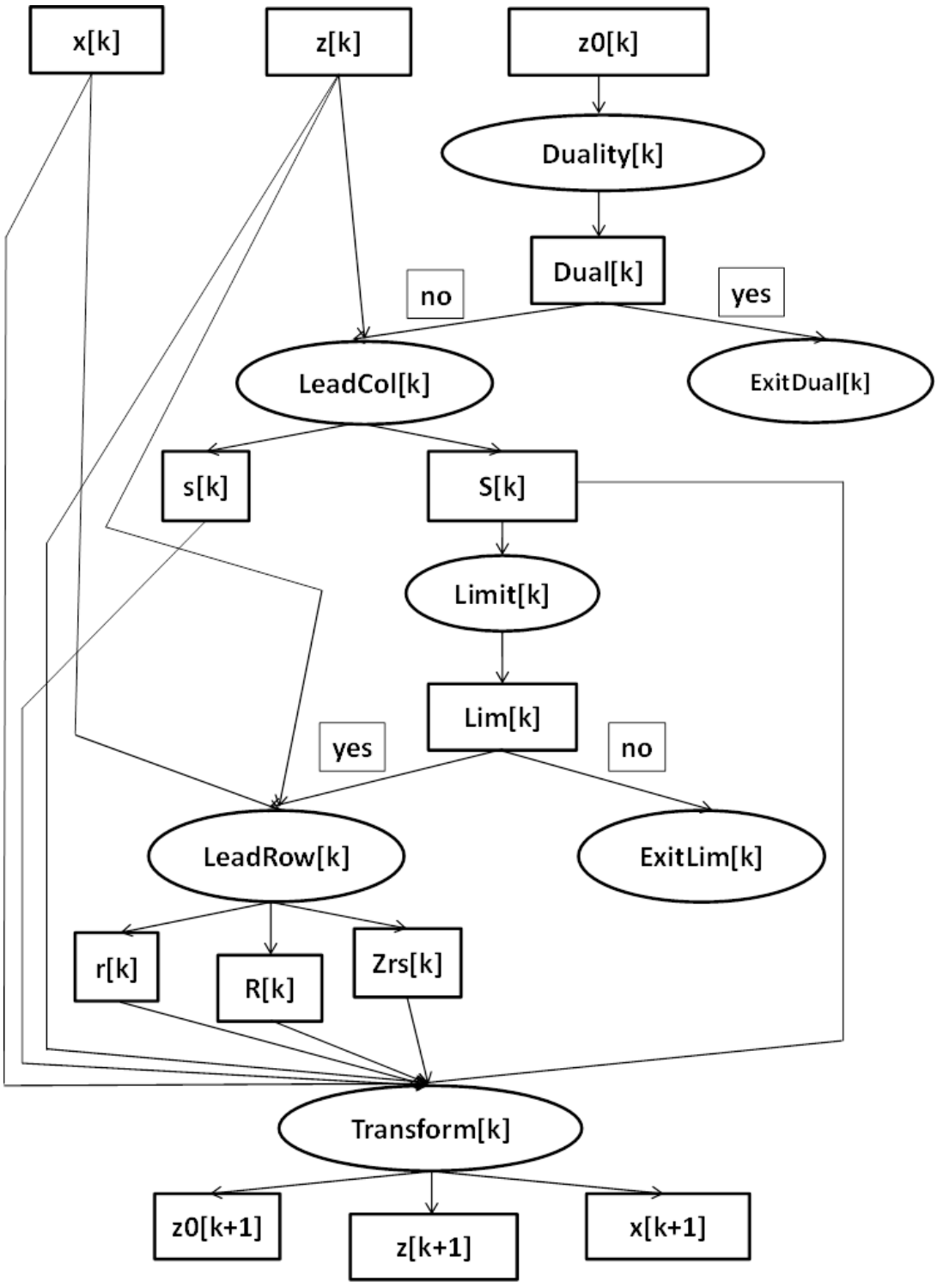

Рис.5 Фрагментированный наивный алгоритм симплекс-метода

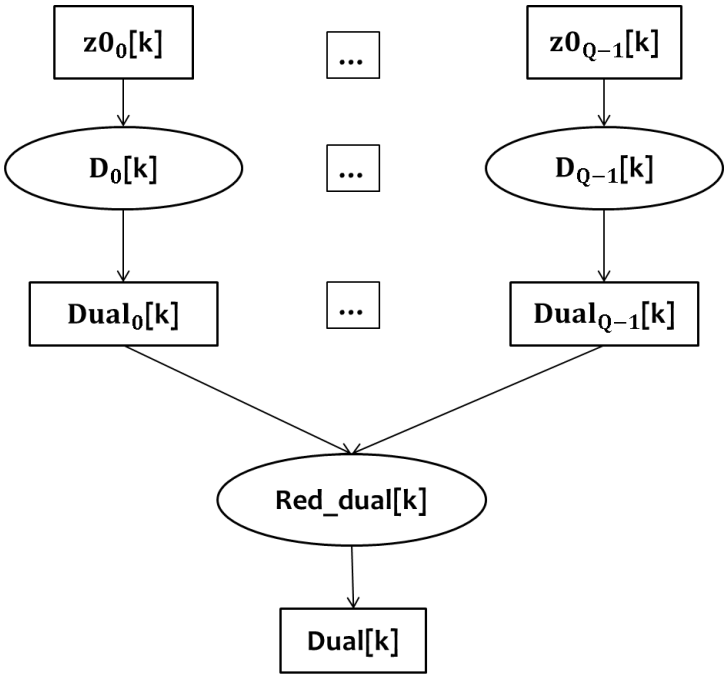

Рис.6 Фрагментированная операция Duality наивного алгоритма симплекс-метода

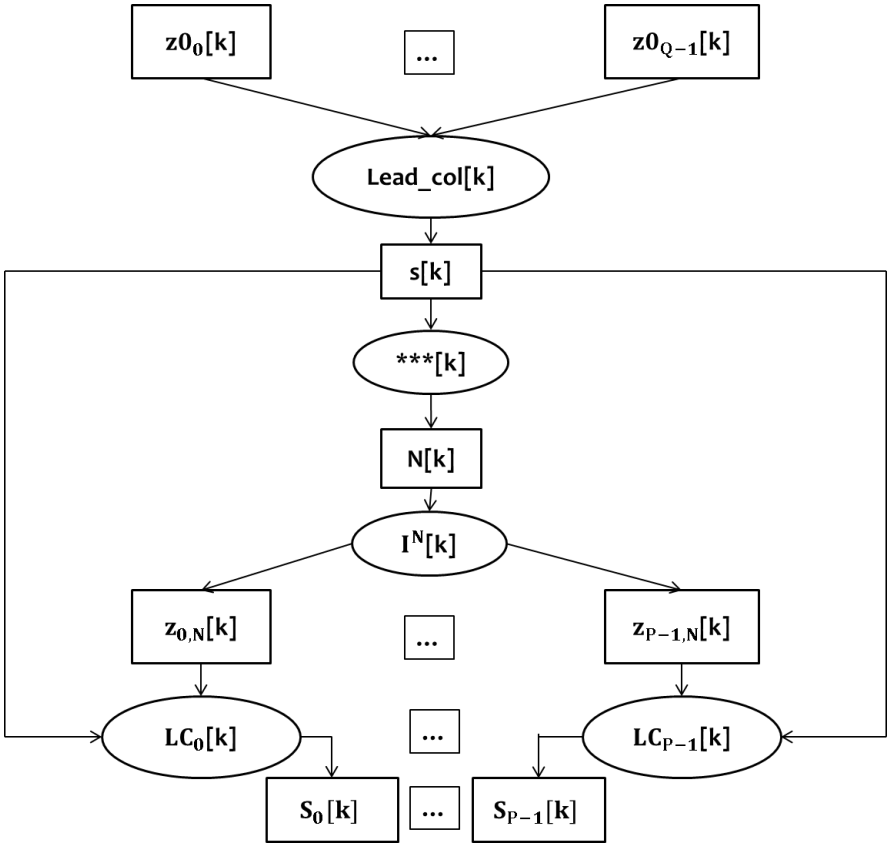

Рис.7 Фрагментированная операция LeadCol наивного алгоритма симплекс-метода

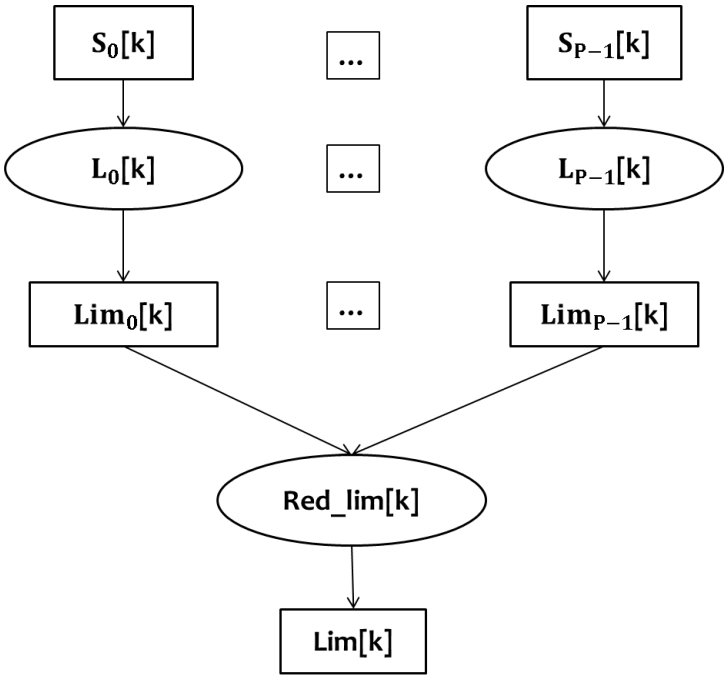

Рис.8 Фрагментированная операция Limit наивного алгоритма симплекс-метода

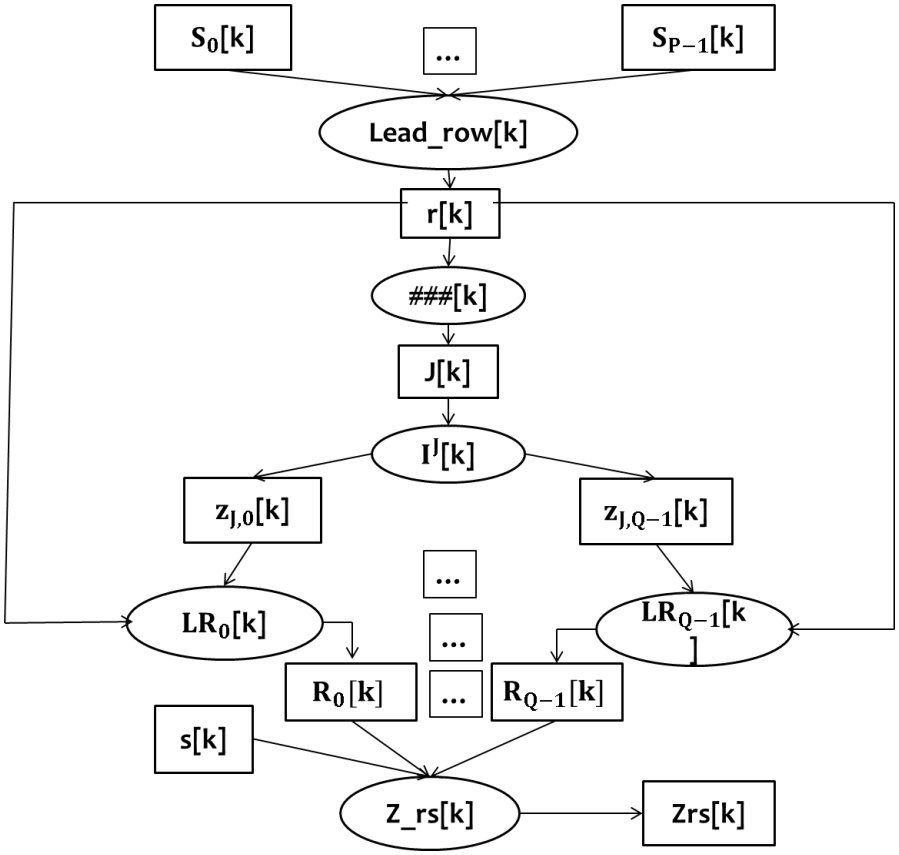

Рис.9 Фрагментированная операция LeadRow наивного алгоритма симплекс-метода

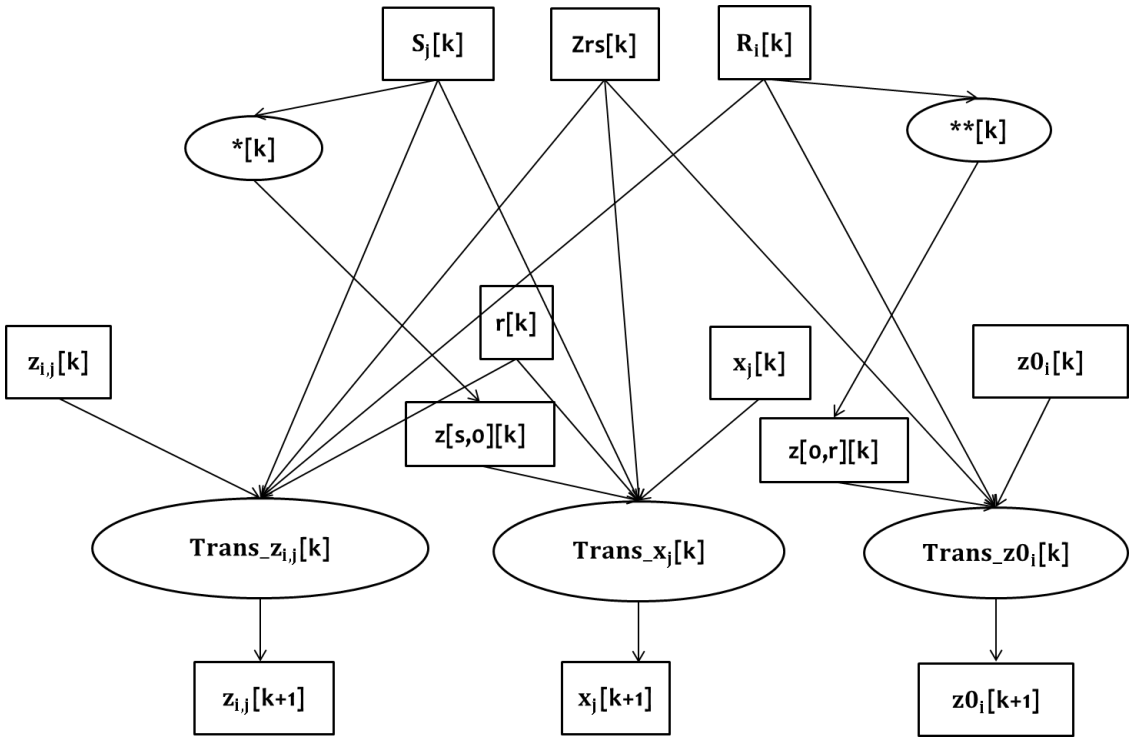

Рис.10 Фрагментированная операция Transform наивного алгоритма симплекс-метода

Условные обозначения

– фрагменты данных

– фрагменты вычислений

yes – переходы по условию: булева переменная принимает значение true

no – переходы по условию: булева переменная принимает значение false