Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Институт естественных и точных наук
Факультет математики, механики и компьютерных технологий
Кафедра математического и компьютерного моделирования

РАБОТА ПРОВЕРЕНА	ДОПУСТИТЬ К ЗАЩИТЕ
Рецензент, доцент кафедры прикладной математики, к.п.н.	Заведующий кафедрой, д.ф.-м.н., доцент
____________ Эвнин А.Ю.	____________ Загребина С.А.
« » ________ 2017 г.	« » ________ 2017 г.

Параллельная реализация алгоритма симплекс-метода для задач оптимизации большой размерности

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУрГУ-010400.68.2017.049.001 КР

Руководитель проекта, д.ф.-м.н., профессор
____________ А.В. Панюков
«   » ________ 2017 г.

Автор проекта
студент группы ЕТ-224
____________ В.А. Безбородов
«   » ________ 2017 г.

Нормоконтролер, к.ф.-м.н., доцент
____________ Т.А. Макаровских
«   » ________ 2017 г.

Челябинск, 2017

Дипломная работа выполнена мной совершенно самостоятельно. Все использованные в работе материалы и концепции из опубликованной научной литературы и других источников имеют ссылки на них.

_______________ В.А. Безбородов

«   » _______________ 2017 г.

Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Институт естественных и точных наук
Факультет математики, механики и компьютерных технологий
Кафедра математического и компьютерного моделирования

УТВЕРЖДАЮ
Заведующий кафедрой, д.ф.-м.н.,
доцент
____________ Загребина С.А.
«   » ________ 2017 г.

З А Д А Н И Е

на выпускную квалификационную работу студента

Безбородова Вячеслава Александровича

Группа ЕТ-224

1. Тема работы: Параллельная реализация алгоритма симплекс-метода для задач оптимизации большой размерности.
   Утверждена приказом по университету от «   » ________ 2017 г.
   №____________
2. Срок сдачи студентом законченной работы «   » ________ 2017 г.
3. Исходные данные к работе
   3.1. Модельные данные;
   3.2. Типовые примеры;
   3.3. Самостоятельно сконструированные тестовые примеры.
4. Перечень вопросов, подлежащих разработке
   4.1. Изучение общей схемы работы алгоритма обратного симплекс-метода;

4.2. Изучение приемов параллельной обработки данных;
4.3. Изучение приемов программирования на языке для технических расчетов Julia;
4.4. Разработка класса (типа данных) для ЗЛП;
4.5. Разработка решателя, инкапсулирующего работу алгоритма параллельного обратного симплекс-метода;
4.6. Тестирование разработанного ПО;
4.7. Проверка разработанного ПО на модельных данных;
4.8. Разработка отчетной документации (отчета по преддипломной практике), в которой отражены основные этапы работы.

5. Перечень графического материала

5.1. Итерация обратного симплекс-метода – 1 л.
5.2. Прототип параллельной реализации обратного симплекс-метода с субоптимизацией – 1 л.
5.3. Итерация обратного симплекс-метода с применением субоптимизации и метода наиболее крутого ребра – 1 л.
5.4. Обзор существующих пакетов для математической оптимизации в Julia – 1 л.

6. Календарный план

Наименование этапов работы	Срок выполнения этапов	Отметка о выполнении
1. Сбор материалов и литературы по теме работы	30.01.2017 г.	
2. Исследование математической модели алгоритма обратного симплекс-метода	15.02.2017 г.	
3. Реализация параллельного алгоритма	10.03.2017 г.	
4. Проведение вычислительного эксперимента	15.03.2017 г.	
5. Подготовка пояснительной записки	16.03.2017 г.	
6. Написание главы 1	18.03.2017 г.	
7. Написание главы 2	20.03.2017 г.	
8. Написание главы 3	21.03.2017 г.	
9. Оформление пояснительной записки	22.03.2017 г.	
10. Получение отзыва руководителя	24.03.2017 г.	
11. Проверка пояснительной записки руководителем, исправление замечаний	24.03.2017 г.	
12. Подготовка графического материала и доклада	25.03.2017 г.	
13. Нормоконтроль	25.03.2017 г.	
14. Рецензирование, представление зав. кафедрой	26.03.2017 г.	

7. Дата выдачи задания «___» __________ 2017 г.

Заведующий кафедрой_______________________________/Загребина С.А./

Руководитель ______________________________________/А.В. Панюков/

Студент____/В.А. Безбородов/

Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Институт естественных и точных наук
Факультет математики, механики и компьютерных технологий
Кафедра математического и компьютерного моделирования

АННОТАЦИЯ

Безбородов, В.А. Параллельная реализация алгоритма симплекс-метода для задач оптимизации большой размерности / В.А. Безбородов – Челябинск: ЮУрГУ, Факультет математики, механики и компьютерных технологий, 2017 – 44 с., 3 ил., 3 табл., 1 прил., библиогр. список – 32 названий.

В работе произведен анализ вычислительной сложности выполняемых операций алгоритма метода эллипсоидов. На основе результатов анализа разработана параллельная реализация метода эллипсоидов, адаптированная для решения задач оптимизации большой размерности на многопроцессорных и/или многоядерных вычислительных системах с общей разделяемой памятью.

Приведены результаты вычислительных экспериментов, продемонстрирован пример решения задачи оптимизации большой размерности с применением разработанной программной реализации.

ОГЛАВЛЕНИЕ

Введение

Линейное программирование (ЛП) является широко распространенной техникой решения оптимизационных задач различных областей науки. Сегодня используются два основных подхода к решению задач линейного программирования (ЗЛП) – симплекс-метод и методы внутренней точки. В случаях, когда необходимо решать семейства взаимосвязанных ЗЛП (целочисленное программирование, методы разложения, некоторые классы задач ЛП), симплекс-метод обычно более эффективен [14].

Возможность распараллелить симплекс-метод для решения ЗЛП рассматривалась с 1970-х гг., хотя первые попытки разработать практические реализации предпринимались только с начала 1980-х гг. Наиболее плодотворным для решения этой проблемы оказался период с конца 1980-х до конца 1990-х гг. Также было несколько экспериментов использования векторной обработки и ЭВМ с общей разделяемой памятью; подавляющее большинство реализаций строилось на мультипроцессорах с распределенной памятью и сетевых кластерах [14].

Есть два главных мотивирующих фактора разработать параллельную реализацию модифицированного симплекс-метода для стандартных настольных архитектур. Во-первых, существующие параллельные реализации используют дорогие высокопроизводительные кластеры для достижения лучшей производительности. Сегодня широко распространены настольные компьютеры, почти каждый из которых является многоядерным, и любое ускорение желательно в смысле уменьшения стоимости и времени получения решения при ежедневном использовании. Во-вторых, практически любая прикладная задача имеет внушительные размеры, а это диктует необходимость использовать арифметику произвольной точности для получения корректного результата.

Актуальность работы состоит в том, что на сегодняшний момент не существует параллельной реализации симплекс-метода, предоставляющей значительный прирост производительности по сравнению с эффективной последовательной реализацией.

Цели работы:

- разработка параллельной реализации алгоритма симплекс-метода, поддерживающей арифметику произвольной точности;
- тестирование производительности полученной программной реализации на ЗЛП большой размерности.

Задачи работы:

- изучение математической модели алгоритма параллельного симплекс-метода;
- разработка программной реализации алгоритма, поддерживающей арифметику произвольной точности;
- unit-тестирование корректности разработанной программной реализации;
- тестирование производительности разработанной программной реализации на ЗЛП большой размерности.

Работа состоит из введения, 4 глав, заключения, 1 приложения и списка литературы. Объем работы составляет 44 страницы. Список литературы содержит 32 наименований.

В первой главе делается обзор существующей литературы по проблеме и дается краткая классификация алгоритмов симплекс-метода.

Во второй главе обсуждается параллелизм и способы его внедрения в симплекс-метод.

В третьей главе описываются особенности программной реализации.

В заключении перечислены основные результаты работы.

1 Симплекс-метод

Линейное программирование (ЛП) широко и успешно использовалось в различных прикладных областях с момента представления симплекс-метода Дж. Б. Данцигом в конце 40-х годов для численного решения основной задачи ЛП [6]. Позднее был разработан метод внутренней точки, ставший конкурентоспособным и популярным с 1980-х, но зачастую двойственный симплекс-метод оказывался предпочтительнее, в частности при решении семейств взаимосвязанных задач линейного программирования (ЗЛП).

Стандартный симплекс-метод реализует алгоритм симплекса с помощью прямоугольной таблицы, что неэффективно при решении разреженных ЗЛП. Для таких задач предпочтительнее оказывается модифицированный симплекс-метод, поскольку в методе применяются техники факторизации разреженных матриц и решения сильно разреженных систем линейных алгебраических уравнений (СЛАУ). Различные алгоритмические варианты и техники (DSE, BFRT), предложенные в 1990-х, привели к значительному приросту производительности симплекс-метода, что стало ключевой причиной его популярности.

Многочисленные попытки распараллелить стандартный симплекс-метод давали хороший прирост производительности (от десятков до сотен раз). Хотя без использования большого количества мощных (и дорогих) ресурсов аппаратного параллелизма, производительность на разреженных ЗЛП оказывалась даже хуже эффективной последовательной реализации. Стандартный симплекс-метод также численно не стабилен. Попыток создать параллельную реализацию модифицированного симплекс-метода предпринималось относительно меньше, и успех в смысле производительности был гораздо скромнее. Действительно, поскольку масштабируемое ускорение для больших разреженных ЗЛП выглядит недостижимым, модифицированный симплекс-метод рассматривался как не подходящий для распараллеливания. Хотя если это относится к эффективной последовательной реализации, любое улучшение производительности благодаря применению параллелизма в модифицированном симплекс-методе является хорошей целью.

Вычислительные требования алгоритма удобнее обсуждать в контексте стандартной формы ЗЛП

$$
\begin{aligned}
&c^T x \to \min \\
&Ax = b \\
&x \geq 0,
\end{aligned}
\tag{1.1}
$$

где $x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$. Матрица A в (1.1) обычно содержит столбцы с единицами, соответствующими фиктивным переменным, возникающим при переводе ограничений-неравенств в равенства. Оставшиеся столбцы A соответствуют обычным переменным.

В симплекс–методе индексы переменных подразделяются на два подмножества: подмножество $\mathcal{B}$, соответствующее m базисным переменным x_B, и подмножество $\mathcal{N}$, соответствующее $n - m$ небазисным переменным x_N. При этом базисная матрица B, составленная из соответствующих $\mathcal{B}$ столбцов A, является невырожденной. Множество $\mathcal{B}$ условно называют базисом. Соответствующие $\mathcal{N}$ столбцы A формируют матрицу N. Компоненты c, соответствующие $\mathcal{B}$ и $\mathcal{N}$, называют базисными c_B и небазисными c_N издержками соответственно [14].

Когда небазисные переменные нулевые, значения $\hat{b} = B^{-1}b$ базисных переменных соответствуют вершинам допустимого региона при условии, что они неотрицательны. Выражение $x_B + B^{-1}N = \hat{b}$, следующее из (1.1), позволяет убрать базисные переменные из целевой функции, которая становится равной $(c_N^T - c_B^T B^{-1} N)x_N + c_B^T \hat{b}$. Если все компоненты вектора альтернативных издержек $\hat{c}_N = c_N^T - c_B^T B^{-1} N$ неотрицательны, тогда текущий базис оптимален.

Если текущий базис неоптимален, на каждой итерации симплекс-метода для ввода в базис выбирается имеющая отрицательное значение альтернативной издержки небазисная переменная x_q. Увеличение этой переменной от нуля при выполнении условий (1.1) соответствует перемещению вдоль ребра допустимого региона в направлении уменьшения значения целевой функции. Направление этого ребра определяется соответствующим x_q столбцом $\hat{a}_q$ при $\hat{N} = B^{-1}N$. При просмотре отношений компонентов вектора $\hat{b}$ к соответствующим положительным компонентам $\hat{a}_q$ находится первая базисная переменная, которая обнулится при росте x_q и, следовательно, шаг к следующей точке допустимого региона вдоль этого ребра.

Существует много стратегий выбора переменной x_q для ввода в базис. Первоначальное правило выбора переменной с наименьшей альтернативной издержкой известно как критерий Данцига [5]. Хотя, если компоненты $\hat{a}_j$ намного превосходят компоненты $\hat{c}_j$, то только небольшое увеличение x_j возможно до обращения одной из базисных переменных в ноль. Альтернативные ценовые стратегии взвешивают альтернативную издержку путем деления на длину $\hat{a}_j$. Точная стратегия наиболее крутого ребра [13] вводит понятие весов $s_j = 1 + ||\hat{\alpha}_j||^2$, соответствующих длине шага при единичном изменении x_j. Практический (приближенный) метод наиболее крутого ребра [10] и стратегия Devex [16] вычисляют приближенное значение весов. При использовании этих подходов количество итераций, необходимых для решения ЗЛП на практике может быть оценено как $O(m+n)$, и теоретически нет препятствий для достижения сложности $O(2^n)$.

Популярной техникой выбора выводимой из базиса переменной является процедура EXPAND [22]. Посредством небольшого расширения ограничений, эта стратегия часто позволяет выбрать выводимую переменную из числа возможных на основании численной стабильности.

Два главных варианта симплекс-метода соответствуют различным пониманиям того, какие данные требуются для определения шага к новой точке. Первый вариант – *обычный симплекс-метод*, в котором альтернативные издержки и направления всех ребер в текущей точке определяются прямоугольной таблицей. В *обратном симплекс-методе* альтернативные издержки и направление выбранного ребра определяются путем решения систем с базисной матрицей B.

1.1 Обычный симплекс-метод

В обычном симплекс-методе матрица $\hat{N}$, правый вектор-столбец $\hat{b}$, альтернативные издержки $\hat{c}_N$ и текущее значение целевой функции $f = c_B^T\hat{b}$ располагаются в виде таблицы следующей формы.

	$\mathcal{N}$	RHS
$\mathcal{B}$	$\hat{N}$	$\hat{b}$
	$\hat{c}_N^T$	$-\hat{f}$

На каждой итерации обычного симплекс-метода для перехода к новому базису к колонкам этой таблицы применяется процедура преобразований Жордана-Гаусса.

Выполнение симплекс-метода начинается с базиса $B = E$, в то время как в таблице симплекса записана матрица N. Это означает, что таблица является разреженной. Принято считать, что степень заполненности матрицы в процессе выполняемых преобразований такова, что нет необходимости использовать разреженные структуры данных, поэтому обычный симплекс-метод часто реализуют без их использования.

Обычный симплекс-метод по природе своей численно нестабилен, поскольку использует длинную цепочку операций последовательного исключения переменных, выбирая колонку поворота согласно алгоритму, а не из соображений стабильности вычислений. Если алгоритм получает плохо обусловленные базисные матрицы, любая подпоследовательность таблицы, соответствующая хорошо обусловленному базису, скорее всего будет содержать численные ошибки, вызванные плохой обусловленностью на ранних этапах. Это может привести к такому выбору вводимой или выводимой переменной, что при точных вычислениях целевая функция не будет убывать монотонно, ограничения будут нарушены либо базисная матрица станет вырожденной. Для большей надежности необходимо отслеживать возникающие в таблице ошибки и, при необходимости, выполнять полный ее пересчет численно стабильным способом. Проверка ошибок может осуществляться сравнением обновленных альтернативных издержек со значением, полученным напрямую с использованием колонки поворота и базисных издержек. Поскольку операции с матрицей, обратной базисной, могут быть выполнены посредством использования соответствующих ячеек таблицы, вычисление колонки поворота напрямую и сравнение ее с ячейками таблицы может предоставить более надежный (но и более затратный) механизм проверки ошибок.

CHUZC: Выбрать из $\hat{c}_N$ хорошего кандидата q для ввода в базис.
FTRAN: Сформировать колонку поворота $\hat{a}_q = B^{-1}a_q$, где a_q – колонка q матрицы A.
CHUZR: Из отношений $\hat{b}_i/\hat{a}_{iq}$ определить номер p строки хорошего кандидата для вывода из базиса.
Положить $\alpha = \hat{b}_p/\hat{a}_{pq}$.
Обновить $\hat{b} := \hat{b} - \alpha\hat{a}_q$.
BTRAN: Сформировать $\pi_p^T = e_p^T B^{-1}$.
PRICE: Сформировать строку поворота $\hat{a}_p^T = \pi_p^T N$.
Обновить альтернативные издержки $\hat{c}_N^T := \hat{c}_N^T - \hat{c}_q\hat{\alpha}_p^T$.
Если {рост в представлении B^{-1}}, тогда
INVERT: Сформировать новое представление B^{-1}.
иначе
UPDATE: Обновить представление B^{-1} в соответствии с изменением базиса.
конец если

Таблица 1 — Итерация обратного симплекс-метода

1.2 Модифицированный симплекс-метод

Вычислительные этапы модифицированного симплекс-метода представлены в таблице 1. Вначале каждой итерации полагается, что вектор альтернативных издержек $\hat{c}_N$ и вектор $\hat{b}$ текущих значений базисных переменных известны, и представление B^{-1} доступно. Первым шагом алгоритма является CHUZC, который ищет хорошего кандидата q для ввода в базис среди (взвешенных) альтернативных издержек. *Колонка поворота* $\hat{a}_q$ формируется на шаге FTRAN, используя представление B^{-1}.

На шаге CHUZR определяется выводимая из базиса переменная. Индекс p показывает, в какой строке расположена выводимая переменная, а сама строка именуется *строкой поворота*. Индекс самой переменной обозначается как p'. Как только индексы q и p' меняются местами между множествами $\mathcal{B}$ и $\mathcal{N}$, говорят, что произошло *изменение базиса*. После этого, правый вектор-столбец $\hat{b}$ обновляется в соответствии с увеличением $\alpha = \hat{b}_p/\hat{a}_{pq}$ в x_q.

Перед выполнением следующей операции необходимо получить значения альтернативных издержек и представление новой матрицы B^{-1}. Хотя альтернативные издержки могут быть вычислены и напрямую, используя выражения

$$\pi_B^T = c_B^T B^{-1};\ \hat{c}_N^T = c_N^T - \pi_B^T N,$$

в вычислительном смысле гораздо эффективнее обновлять их с помощью строки поворота $\hat{a}_p^T = e_p^T B^{-1} N$ из таблицы стандартного симплекса. Это выполняется в два шага. Сначала, используя представление B^{-1}, на шаге `BTRAN` формируется вектор $\pi_p^T = e_p^T B^{-1}$, а затем строится вектор $\hat{a}_p^T = \pi_p^T N$ значений строки поворота (шаг `PRICE`). Как только были получены значения альтернативных издержек, шак `UPDATE` изменяет представление B^{-1} в соответствии с изменением базиса. Необходимо отметить, что из соображений эффективности и численной стабильности, периодически необходимо находить новое представление B^{-1} с помощью операции `INVERT`.

При применении стратегии Devex [16], строка поворота, вычисленная для обновления альтернативных издержек, используется также для обновления Devex весов при незначительных вычислительных затратах. Для обновления точных весов в методе наиболее крутого ребра в дополнение к строке поворота требуется дополнительный шаг `BTRAN` для вычисления $\hat{a}_q^T B^{-1}$ и шаг `PRICE` для получения результата матричного умножения этого вектора и матрицы N. Также вычислительно неэффективно инициализировать значения весов наиболее крутого ребра, если начальная базисная матрица не единичная. Как следствие этих дополнительных затрат и поскольку стратегия Devex работает хорошо в плане уменьшения количества итераций, необходимых для решения ЗЛП, эта стратегия обычно используется в эффективных последовательных реализациях обратного симплекс-метода.

Выводы по главе один

Симплекс-метод – алгоритм решения оптимизационной задачи линейного программирования путём перебора вершин выпуклого многогранника в многомерном пространстве.

Сущность метода заключается в построении базисных решений, на которых монотонно убывает линейный функционал, до ситуации, когда выполняются необходимые условия локальной оптимальности.

Симплекс-метод имеет среднюю полиномиальную сходимость при широком выборе распределения значений в случайных матрицах [18, 24].

2 Параллельный симплекс-метод

В этой главе представлен прототип схемы распараллеливания обратного симплекс-метода с применением субоптимизации и метода наиболее крутого ребра.

2.1 Параллельные вычисления

Прежде чем переходить к вопросу распараллеливания симплекс-метода, необходимо рассмотреть некоторые термины и концепции из области параллельного программирования. В этом разделе представлен краткий обзор необходимых понятий. Полное и более общее введение в параллельные вычисления можно найти в [17].

Классифицируя архитектуры параллельных мультипроцессоров, необходимо понимать важное отличие между *распределенной памятью*, когда каждый процессор имеет свою собственную локальную память, и *общей памятью*, когда все процессоры имеют доступ к общей разделяемой памяти. Современные мощные ЭВМ могут состоять из множества распределенных кластеров, каждый из которых может иметь множество процессоров с общей памятью. На более простых мультипроцессорах память может быть либо общей, либо разделяемой.

Обычно успешность распараллеливания измеряется в терминах *ускорения* – отношения времени, необходимого для решения задачи с использованием более одного процессора, ко времени решения задачи на одном процессоре. Традиционной является цель достичь фактор ускорения, равный количеству подключаемых процессоров. Такой фактор называется *линейным ускорением* и соответствует 100% *параллельной эффективности*. Увеличение доступной кэш-памяти и оперативной памяти одновременно с количеством процессоров иногда приводит к феномену *сверхлинейного ускорения*. Схемы распараллеливания, для которых (по крайней мере в теории) производительность растет линейно без ограничений с ростом количества подключаемых процессоров, называются *масштабируемыми* схемами. Если параллелизм не используется во всех главных операциях алгоритма, то ускорение, в соответствии с законом Амдала [1], ограничено долей времени выполнения непараллельных операций.

Существуют две основных парадигмы параллельного программирования. Если работа большинства операций алгоритма может быть распределена среди множества процессоров, тогда говорят о *параллелизме по данным*. В противоположность этому, если возможно выполнять несколько главных операций алгоритма одновременно, тогда имеет место *параллелизм по задачам*. На практике возможно применять одновременно оба подхода для определенного набора главных операций алгоритма.

Есть два фундаментальных способа реализации алгоритмов на параллельных ЭВМ. На машинах с распределенной памятью передача данных между процессорами осуществляется посредством инструкций, порожденных явными вызовами методов *передачи сообщений*. На машинах с общей разделяемой памятью применяется *параллелизм по данным*, когда инструкции записываются как для последовательного исполнения, но транслируются специальным компилятором в параллельный код. Большинство протоколов передачи сообщений также поддерживаются на ЭВМ с общей памятью, равно как и распараллеливание по данным возможно на ЭВМ с распределенной памятью.

На машинах с распределенной памятью, накладные расходы на передачу сообщений между процессорами определяются *задержкой* и *пропускной способностью канала*. Первое – это время передачи, не зависящее от размера сообщения, а второе – это скорость связи. Для общих протоколов передачи сообщений задержка и пропускная способность на определенной архитектуре может быть значительно выше, чем в независящей от архитектуры среде, что обычно регулируется и настраивается поставщиком. Если алгоритму для вычислений необходим интенсивный обмен информацией, растущие накладные расходы на связь могут перевесить любые улучшения от использования дополнительных процессоров.

2.2 Распараллеливание симплекс-метода

Существующие подходы к распараллеливанию симплекс-метода и ему подобных удобно классифицировать по виду симплекс-метода и по использованию разреженных типов данных. Такая классификация позитивно коррелирует с практической ценностью реализации в контексте решения ЗЛП и негативно с успешностью этих подходов в достигнутом ускорении.

Некоторые из рассматриваемых ниже схем предлагают неплохое ускорение относительно эффективных последовательных решателей своего времени. Другие только кажутся неэффективными в свете последовательного обратного симплекса, который к тому моменту либо был малоизвестен, либо был разработан впоследствии. Такие случаи определяются ниже, чтобы подчеркнуть, что в результате огромного увеличения эффективности последовательного обратного симплекс-метода (как во время исследований в области распараллеливания симплекса, так и после) проблема разработки практического параллельного симплекс-решателя стала очень актуальной.

2.2.1 Параллельный симплекс-метод с использованием алгебры плотных матриц

Стандартный и обратный симплекс-методы с использованием алгебры плотных матриц реализовывались неоднократно. Простота обычных структур данных и потенциал достичь линейного ускорения делают их привлекательными для применения в параллельных вычислениях. Хотя при решении общих разреженных ЗЛП больших размерностей такие реализации малоэффективны, поскольку они могут соперничать с эффективными последовательными реализациями обратного симплекс-метода, использующими разреженные структуры, только при подключении значительного числа процессоров.

Первые работы в этом направлении ограничиваются обсуждением схем распределения данных и коммуникации; реализации ограничиваются небольшим числом процессов на ЭВМ с распределенной памятью (краткие обзоры даются в [28,31], примеры других ранних работ можно найти в [3,9,11,32]). В одной из относительно ранних работ [27], в которой были реализованы обычный и обратный симплекс-методы на 16-процессорном Intel hypercube, достигнутое ускорение варьируется от 8 до 12 для небольших задач из библиотеки Netlib. В [4] сообщается о 12-кратном ускорении при решении двух небольших ЗЛП с использованием обычного симлекс-метода на 16-процессорной ЭВМ с *общей разделяемой памятью*. Также были случаи получения более чем 12-кратного ускорения [21].

В [7] разработаны параллельный обычный и обратный симплекс-методы с применением метода наиболее крутого ребра [13] и протестированы на машинах Connection Machine CM-2 и CM-5 с массовым параллелизмом. Решая некоторые ЗЛП средней размерности из Netlib и очень плотные задачи машинного обучения, ускорение между 1.6 и 1.8 было достигнуто только при удвоении числа процессоров. В [29] также используется метод наиболее крутого ребра и обычный симплекс-метод на ЭВМ MasPar MP-1 и MP-2. Решая в основном случайно сгенерированные ЗЛП большой размерности, авторы достигали практически троекратного ускорения. Одной из более поздних работ по реализации параллельного обычного симплекс-метода с запуском на небольшом количестве процессоров является [30].

Работы по созданию параллельных реализаций обычного симплекс-метода с использованием алгебры плотных матриц для ЗЛП небольших размерностей продолжаются. Были представлены результаты реализации на 8 процессорах с 5-кратным ускорением при решении небольших случайных ЗЛП [26].

2.2.2 Параллельный симплекс-метод с использованием алгебры разреженных матриц

Особой сложностью в разработке действительно хорошего в практическом смысле параллельного симплекс-метода является применение эффективных техник работы с разреженными матрицами. Разработанный параллельный решатель будет конкурентноспособным по отношению к хорошей последовательной реализации только тогда, когда решение общих разреженных ЗЛП большой размерности будет затрагивать разумное число процессоров.

В период, когда распараллеливание симплекс-метода только начиналось и широко дискутировалось, практические параллельные методы факториза-

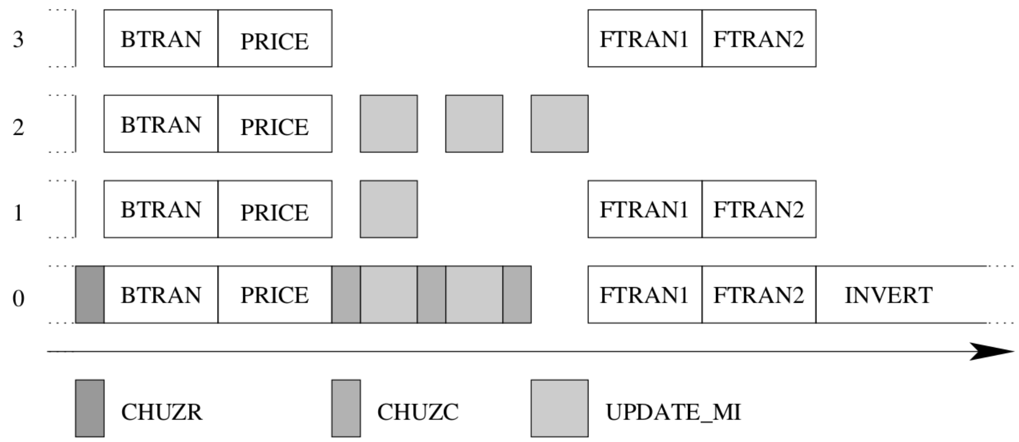

Рисунок 1 — Прототип параллельной реализации обратного симплекс-метода с субоптимизацией

ции и решения разреженных асимметричных СЛАУ были только на стадии становления. Как следствие, несмотря на то, что в симплекс-методе с плотными матрицами внедрение параллелизма проходило успешно, было распространено мнение, что использование разреженных матриц сильно ограничивает возможности распараллеливания (за исключением `PRICE`). Некоторых это наводило на мысль, что разработать хорошую параллельную реализацию невозможно в принципе. Но несмотря на преимущественно последовательную природу компонентов обратного симплекс метода, все еще существуют возможности применения параллелизма по задачам.

2.2.3 Схема распараллеливания

Рассмотрим следующий подход, использующий некоторые (но не все) возможности применения параллелизма по данным и по задачам к алгоритму обратного симплекс-метода с субоптимизацией [15]. Рисунок 1 иллюстрирует идею.

Сначала относительно дешевая операция CHUZR выбора множества $\mathcal{P}$ хороших кандидатов для вывода из базиса выполняется на одном ядре (таблица 2). Затем, несколько операций BTRAN ($\pi_p^T = e_p^T B^{-1}$) и PRICE ($\hat{a}_p^T = \pi_p^T N$) для $p \in \mathcal{P}$ распределяются между всеми ядрами. Поскольку малый цикл итераций обрабатывает только небольшую часть строк, операция CHUZR_MI

`CHUZR`: Из отношений $\hat{b}_i/s_p$ для $p = 1, \dots, m$ определить
множество $\mathcal{P}$ строк хороших кандидатов для вывода из базиса.

`BTRAN`: Сформировать $\pi_p^T = e_p^T B^{-1}$, $\forall p \in \mathcal{P}$.

`PRICE`: Сформировать строку поворота $\hat{a}_p^T = \pi_p^T N$, $\forall p \in \mathcal{P}$.

Цикл {младшие итерации}

  `CHUZR_MI`: Из $\hat{b}$ определить номер строки $p \in \mathcal{P}$ хорошего кандидата для вывода из базиса.
  Если p не определен, то **Конец цикла** {младшие итерации}

  `CHUZC`: Среди отношений $\hat{c}_j/\hat{a}_{pj}$ определить номер столбца q хорошего кандидата для ввода в базис.
  Обновить $\hat{c}_N^T := \hat{c}_N^T - \beta \hat{a}_p^T$, где $\beta = \hat{c}_q/\hat{a}_{pq}$.

  `UPDATE_MI`: Обновить $\mathcal{P} := \mathcal{P} \backslash \{p\}$ и $\hat{c}_N^T := \hat{c}_N^T - \beta \hat{a}_p^T$, где $\beta = \hat{c}_q/\hat{a}_{pq}$.
  Обновить строки $\hat{a}_P^T$ и $\hat{b}_P$.

Конец цикла {младшие итерации}

Для {каждого изменения базиса} **выполнять**

  `FTRAN1`: Сформировать $\hat{a}_q = B^{-1} a_q$, где a_q – столбец q матрицы A.
  Обновить $\hat{b} := \hat{b} - \alpha \hat{a}_q$, где $\alpha = \hat{b}_p/\hat{a}_{pq}$.

  `FTRAN2`: Сформировать $\tau = B^{-1} \hat{a}_q$.
  Обновить s_p для $p = 1, \dots, m$.

  Если {рост в представлении B^{-1}}, **тогда**

    `INVERT`: Сформировать новое представление B^{-1}.

  иначе

    `UPDATE`: Обновить представление B^{-1} в соответствии с изменением базиса.

  конец если

Конец для

Таблица 2 — Итерация обратного симплекс-метода с применением субоптимизации и метода наиболее крутого ребра выполняется на одном ядре и не показана на рисунке 1. Выбор вводимой колонки выполняется в CHUZC относительно просто, поэтому также не распределяется. Малый цикл замыкает операция UPDATE_MI, в которой параллельное обновление данных в строках таблицы обратного симплекс-метода (оставшиеся кандидаты) и альтернативных издержек $\hat{c}_N^T$ выполняется на всех доступных ядрах. Простое обновление $\hat{b}_P$ выполняется на одном ядре операцией CHUZR_MI. После завершения малого цикла итераций, операции FTRAN $\hat{a} = B^{-1}a_q$ и $\tau = B^{-1}\hat{a}_q$ для каждого изменения базиса распределяется между всеми ядрами. Если необходимо, INVERT выполняется последовательно, без перекрытия любых других вычислений.

Выводы по главе два

Попытки использования параллелизма в симплекс-методе были связаны со многими ведущими членами разработки эффективного последовательного обратного симплекс метода. То, что относительный успех в этой области оказался довольно ограниченным, объясняется трудоемкостью задачи.

Необходимо отметить, что попытки внедрения параллелизма в обычный симплекс-метод для общих разреженных ЗЛП значительно улучшили его производительность по сравнению с хорошей последовательной реализацией обратного симплекс-метода. Параллельные обычный или обратный симплекс-методы с использованием алгебры плотных матриц несостоятельны без привлечения значительного числа процессоров. Внедрение параллелизма по задачам также было ограничено численной нестабильностью и большими накладными расходами на передачу сообщений на ЭВМ с распределенной памятью.

3 Реализация алгоритма обратного симплекс-метода

В данной главе рассматриваются особенности реализации параллельного алгоритма обратного симплекс-метода на сравнительно новом языке для технических расчетов Julia.

Прежде, чем переходить к детальному описанию и обсуждению кода, рассмотрим кратко некоторые возможности этого языка.

3.1 Julia как язык разработки решателя

Язык программирования Julia – это высокоуровневый высокопроизводительный динамический язык программирования для технических вычислений [2]. Синтаксис языка очень похож на синтаксис других популярных сред для технических расчетов. В распоряжении имеются умный компилятор, распределенное параллельное исполнение, численная точность и большая библиотека математических функций. Базовая библиотека Julia, в большинстве своем написанная на Julia, также содержит лучшие открытые библиотеки C и Fortran для линейной алгебры, генерации случайных чисел, обработки сигналов и работы со строками. Сообщество разработчиков Julia поставляет большое количество внешних пакетов через встроенный пакетный менеджер. По аналогии с популярной утилитой Jupyter, существует многофункциональный браузерный графический интерфейс IJulia.

Ниже перечислены некоторые из встроенных возможностей Julia[1].

- Мультиметод: обеспечивает возможность определять поведение функции в зависимости от типа передаваемых аргументов.
- Динамическая типизация.
- Хорошая производительность, сравнимая со статичиски компилируемыми языками такими, как C.
- Встроенная система управления пакетов.
- Макросы и другие возможности метапрограммирования.
- Архитектура, специально спроектированная для параллельных и распределенных вычислений.

[1] Подробнее на http://julialang.org.

- Сопрограммы: легковесные "зеленые"потоки.
- Возможность определять пользовательские типы, не уступающие в скорости и удобстве встроенным.
- Автоматическое создание эффективного специализированного кода для различных типов аргументов.
- Элегантные и расширяемые преобразования для числовых и других типов.
- Поддержка Unicode, включая (но не ограничиваясь) UTF-8.
- Лицензия MIT: бесплатность и открытость исходного кода.

Основанный на LLVM JIT-компилятор Julia и дизайн самого языка позволяют ему по производительности приблизиться и часто сравняться с производительностью языка C [19].

Julia не заставляет пользователя следовать какому-либо определенному стилю параллелизма [25]. Вместо этого, в распоряжении имеется множество ключевых строительных блоков для распределенных вычислений, которые позволяют использовать различные стили и добавлять новые.

JuliaBox позволяет запускать ноутбуки IJulia в контейнерах-песочницах Docker. Это открывает простор для полностью облачной обработки, включая управление данными, редактирование и обмен кодом, выполнение, отладки, анализа и визуализации [23]. Цель этого проста – перестать беспокоиться об администрировании машин и управлении данными и сразу перейти к решению задачи.

3.2 Разработка решателя

Пакеты для математической оптимизации в Julia представляют собой единое пространство JuliaOpt.

Пакеты JuliaOpt построены на основе MathProgBase.jl – прослойке абстракции, предоставляющей высокоуровневые функции для линейного и целочисленного программирования, а также набор низкоуровневых функций для создания новых алгоритмов (см. рисунок 2, показана зеленым). Над уровнем абстракции расположены языки моделирования (красный), а ниже – интерфейсы внешних библиотек для решения ЗЛП (фиолетовый).

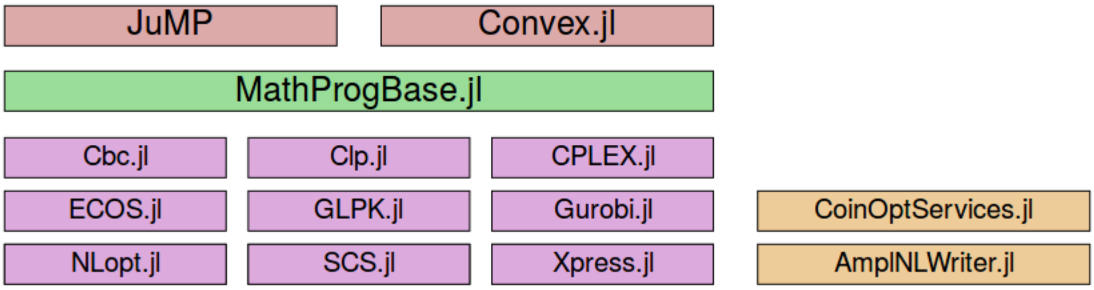

Рисунок 2 — Обзор существующих пакетов для математической оптимизации в Julia

JuliaOpt предоставляет 2 языка моделирования для решения ЗЛП:

- **JuMP** – алгебраический язык моделирования для задач оптимизации с линейными, квадратичными и нелинейными ограничениями. Генерирует модели также быстро, как аналогичные коммерческие утилиты, а также поддерживает дополнительные возможности, такие, как функции обратного вызова для решателей.
- **Convex.jl** – алгебраический язык моделирования для высокодисциплинированного выпуклого программирования.

Пакет MathProgBase.jl определяет модуль *SolverInterface*, который представляет собой абстракцию над низкоуровневыми интерфейсами, общими для большинства библиотек. Модуль *SolverInterface* определяет такие высокоуровневые функции, как `linprog`, `mixintprog` и `quadprog`, которые не зависят от используемой для решения ЗЛП библиотеки. Языки моделирования *JuMP* и *Convex.jl* используют интерфейсы этого модуля для обмена информацией с различными решателями.

Существует 3 категории решателей (некоторые решатели могут принадлежать к более чем одной категории):

- **LinearQuadratic** – решают линейные и квадратичные задачи программирования и принимают на вход данные в виде матриц, определяющих линейные и квадратичные компоненты ограничений и целевую функцию. Примерами решателей этой категории являются `Cbc`, `Clp`, `CPLEX`, `GLPK`, `Gurobi` и `Mosek`.

- **Conic** – решают конические задачи программирования. Входной формат таких решателей предсталяет собой матрицы и векторы, определяющие афинные функции и список конусов. Примерами являются `ECOS`, `Mosek` и `SCS`.
- **Nonlinear** – традиционные нелинейные решатели, которым необходим доступ к алгебраическому представлению задачи. Примеры этой категории: `AmpINLWriter`, `CoinOptServices`, `Lpopt`, `KNITRO`, `MOSEK` и `NLopt`.

Разделение решателей на небольшое число категорий позволяет легко реализовать автоматический перевод задачи между разными представлениями. Это необходимо для перевода ЗЛП из пользовательского представления в структуры данных, которые принимают на вход решатели.

Модуль *SolverInterface* разделяет понятия "решатель"и "модель". Решатель – это небольшой объект, используемый для настройки параметров, он не хранит никаких данных задачи. Решатель используется для создания объекта модели – представление задачи решателя в оперативной памяти.

В файле `DrsMathProgSolverInterface.jl` определяется модуль решателя, производится настройка уровня логирования и подключаются необходимые библиотеки.

```
module DrsMathProgSolverInterface

include("Simplex.jl")
using .Simplex

using Logging
@Logging.configure(level=DEBUG)

importall MathProgBase.SolverInterface
```

Далее следует секция экспорта, определяющая, какие методы и сущности будут видны пользователям модуля.

```
export DrsMathProgModel,
    DrsMathProgSolver,
    loadproblem!,
    optimize!,
    status,
    getreducedcosts,
```

```
    getconstrduals,
    getobjval,
    getsolution
```

Потом следуют определения решателя и модели – абстракции, используемые далее в коде для решения ЗЛП.

```
immutable DrsMathProgSolver <: AbstractMathProgSolver
    options
end
DrsMathProgSolver(; kwargs...) = DrsMathProgSolver(kwargs)

type DrsMathProgModel <: AbstractLinearQuadraticModel
    A               # constraint coefficients
    b               # RHS
    c               # objective coefficients
    basis           # basis variables
    nonbasis        # nonbasis variables
end
LinearQuadraticModel(s::DrsMathProgSolver) = DrsMathProgModel(; s.options...)
```

Решение задачи начинается с функции инициализации, в которой задаются исходные данные и производится первоначальная настройка.

```
function loadproblem!(m::DrsMathProgModel, A, l, u, c, lb, ub, sense)
    @debug("loadproblem!: A $A, l $l, u $u, c $c, lb $lb, ub $ub, sense $sense")
    m.A = A
    m.b = zeros(size(ub))
    m.c = c

    DrsTransformToStandardForm!(m, lb, ub, sense)

    r, c = size(m.A)
    m.basis = zeros(Int, r)

    DrsFindPotentialBasis!(m)
end
```

Прежде, чем переходить к решению ЗЛП, необходимо привести ее к стандартной форме (тело функции довольно объемно, полный исходный код см. в приложении А) и найти первоначальный базис.

```
function DrsFindPotentialBasis!(m::DrsMathProgModel)
    r, c = size(m.A)
    for ic in 1:c
        column = m.A[:,ic]
        if countnz(column) == 1
```

```
            ir = findfirst(x -> x == 1, column)
            if ir != 0 && m.basis[ir] == 0
                # add the column if current row has not been selected
                m.basis[ir] = ic
            end
        end
    end
    m.nonbasis = setdiff(1:c, m.basis)
end
```

Модуль также определяет различные функции опроса статуса и получения результата решения задачи.

3.3 Параллельный вызов функций решателя

Передача сообщений между процессами в Julia несколько отличается от других сред, таких как MPI. Общение зачастую "одностороннее т.е. программист явно управляет только одним главным процессом.

Параллельное программирование в Julia строится на 2 примитивах: *удаленных ссылках* и *удаленных вызовах*. Удаленная ссылка – это объект, который может быть использован любым процессом для идентификации объекта, созданного в контексте какого-либо процесса. Удаленный вызов – это запрос процесса к другому процессу выполнить определенную функцию на некотором наборе аргументов.

Удаленный вызов возвращает объект `Future` в качестве результата. Объект возвращается немедленно; процесс, сделавший вызов, продолжает выполение следующих операций, в то время как удаленный вызов выполняется в другом процессе. Результат выполнения операции будет доступен в объекте `Future`.

В алгоритме симплекс-метода параллельное выполнение приходится на шаги `BTRAN` и `PRICE`.

```
        @sync begin
            for p in P
                pi = @spawn BTRAN(invB, p)
                pivotal_row = @spawn PRICE(N, fetch(pi))
                push!(pivotal_rows, fetch(pivotal_row))
            end
        end
```

Выполнение блока начинается с удаленного вызова BTRAN для выполнения в другом процессе. Объект Future, хранящий в себе числовой результат, будет доступен сразу после выполнения операции. Результат передается функции PRICE, также запускаемой удаленно.

Для параллельной обработки данных в Julia предусмотрены специальные структуры данных. DistributedArrays предназначены для оперирования массивами, размеры который слишком велики для одной ЭВМ. Каждый процесс при этом обрабатывает только свою часть массива, которая доступна локально. Но реализация алгоритма использует SharedArray.

```
    invB = SharedArray(typeof(B[1]), size(B),
        init = S -> S[linearindices(B)] = inv(B)[linearindices(B)])
```

SharedArray позволяет нескольким процессам получить доступ к общим данным. Поскольку метод основан на вычислениях с обратной матрицей, доступ к ней должен осуществляться практически с любого шага алгоритма не зависимо от того, на каком процессе он выполняется.

Выводы по главе три

Julia – высокоуровневый высокопроизводительный свободный язык программирования с динамической типизацией, созданный для математических вычислений. Эффективен также и для написания программ общего назначения. Синтаксис языка схож с синтаксисом других математических языков (например, MATLAB и Octave), однако имеет некоторые существенные отличия. Julia написана на Си, C++ и Scheme. В стандартный комплект входит JIT-компилятор на основе LLVM, благодаря чему, по утверждению авторов языка, приложения, полностью написанные на языке, практически не уступают в производительности приложениям, написанным на статически компилируемых языках вроде Си или C++. Большая часть стандартной библиотеки языка написана на нём же. Также язык имеет встроенную поддержку большого числа команд для распределенных вычислений.

4 Результаты

Алгоритм параллельного симплекс-метода был реализован на языке для технических расчетов Julia с использованием техник, обсуждаемых в главе 2, и протестирован на четырехядерной системе AMD Opteron 2378. Эксперименты проводились с использованием задач из наборов Netlib [12] и Kennigton [8]. Результаты представлены в таблице 3.

Задачи выбирались по следующему принципу. Среди полного набора из 114 задач большинство (84) оказались слишком малы (решение получено менее, чем за 1 секунду) и не были включены в таблицу. Из оставшихся 30 задач 14 не удалось решить за один или более запусков с использованием 1, 2, 4 или 8 потоков.

				Ускорение по отношению				
				К 1 потоку			К Clp	
ЗЛП	**Строк**	**Колонок**	**Элементов**	**2 потока**	**4 потока**	**8 потоков**	**1 поток**	**8 потоков**
25fv47	822	1571	11127	1.12	1.03	0.50	0.47	0.20
80bau3b	2263	9799	29063	0.91	1.02	0.80	0.16	0.17
cre-b	9649	72447	328542	0.82	1.23	1.04	1.12	1.21
cre-d	8927	69980	312626	1.13	1.27	1.66	0.85	1.46
degen3	1504	1818	26230	1.25	1.13	1.09	0.26	0.29
fit2p	3001	13525	60784	0.94	0.90	0.94	0.39	0.40
osa-14	2338	52460	367220	0.92	0.79	1.00	0.12	0.12
osa-30	4351	100024	700160	0.99	0.93	0.84	0.14	0.16
pds-06	9882	28655	82269	1.61	2.13	3.02	0.47	1.37
pds-10	16559	48763	140063	1.27	1.81	1.91	0.48	0.96
qap8	913	1632	8304	1.35	1.18	1.70	0.30	0.51
stocfor3	16676	15695	74004	1.76	2.56	3.38	0.10	0.42
truss	1001	8806	36642	0.96	1.06	1.04	0.41	0.43
Среднее (геометрическое) ускорение				**1.13**	**1.23**	**1.26**	**0.32**	**0.43**

Таблица 3 — Полученное ускорение (до 8 потоков включительно) и производительность по отношению к Clp

Следует заметить, что при использовании 1 потока вычисляется только 1 строка симплекс-таблицы, и, таким образом, выполняется стандартная последовательная версия симплекс-метода. Время работы последовательной

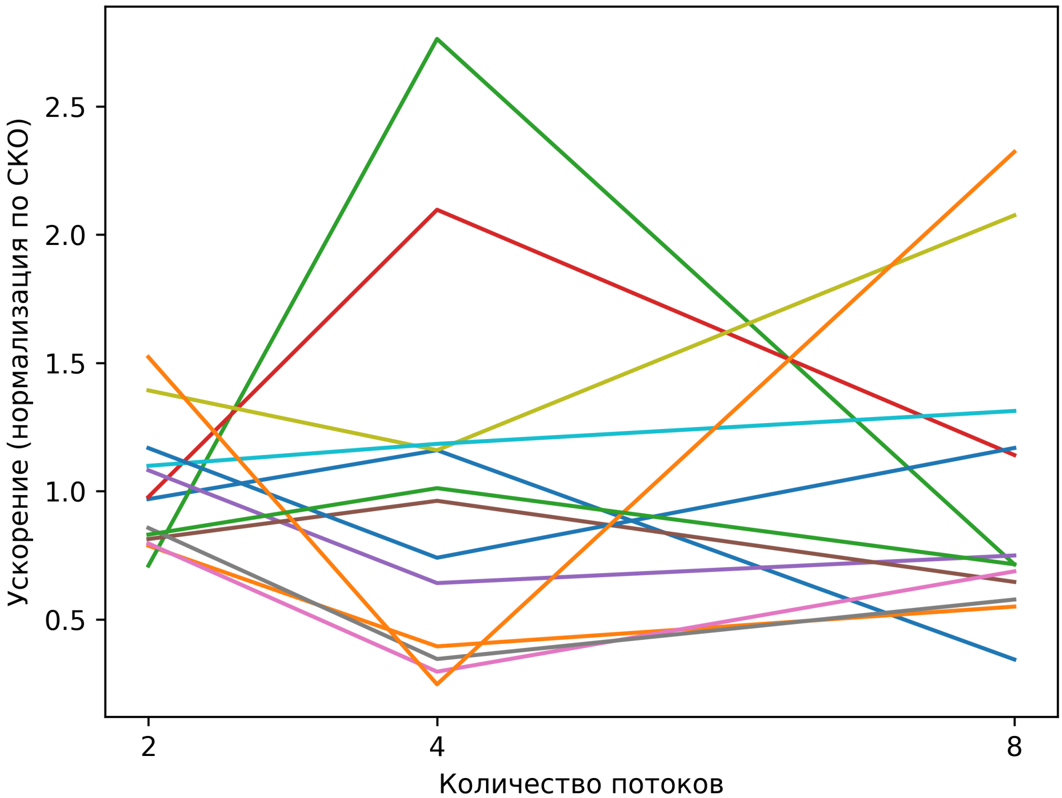

Рисунок 3 — Динамика ускорения при изменении количества потоков

версии сравнивалось со временем решения задачи при помощи последовательного COIN-OR [20] решателя Clp версии 1.06.

Решения для оставшихся 16 задач были получены примерно в десять раз медленнее, чем решения Clp, и тоже не были включены в таблицу. Таким образом, таблица 3 содержит результаты решения 13 тестовых задач, для которых решение требовало не менее 1 секунды процессорного времени на одном потоке, и решение было получено эффективно при использовании 1 потока и успешно при использовании 2, 4 и 8 потоков.

Как показывают результаты в колонках 5-7 таблицы 3, некоторое ускорение было получено для всех задач, кроме трех (fit2p, osa-14 и osa-30), и среднее (геометрическое) ускорение составило около 25% на 4 и 8 потоках (рисунок 3). Колонка 8 показывает полученное ускорение при использовании 1 потока по отношению к Clp: только 1 задача была решена быстрее (cre-b), а в остальном решение было получено в среднем в 3.1 раз медленнее. Хотя, при использовании 8 потоков, для трех задач (cre-b, cre-d и pds-06) решение было получено как минимум так же быстро, как и решение Clp. В остальном время решения в среднем было в 2.3 раза больше.

Заключение

В работе представлена параллельная реализация алгоритма обратного симплекс-метода с использованием субоптимизации и метода наиболее крутого ребра.

В работе удалось реализовать все поставленные **цели**:

- изучена общая схема работы алгоритма обратного симплекс-метода;
- изучены приемы параллельной обработки данных.

Решены следующие **задачи**:

- разработан класс (тип данных) для решения ЗЛП;
- разработан решатель, инкапсулирующий работу алгоритма параллельного обратного симплекс-метода;
- оформлена пояснительная записка, в которой отражены основные этапы работы.

В качестве направлений дальнейших исследований можно рассматривать следующие:

- повышение эффективности работы с типами данных произвольной точности;
- применение последних алгоритмических разработок в области параллельного умножения матриц;
- использование различных модификаций симплекс-метода для повышения производительности.

Разработанный класс может быть использован для решения семейств взаимосвязанных ЗЛП, для которых другие методы не дают удовлетворительный результат за приемлемое время.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

Листинг А.1 — Файл "SolverInterface.jl"

```
module DrsMathProgSolverInterface

include("Simplex.jl")
using .Simplex

using Logging
@Logging.configure(level=DEBUG)

importall MathProgBase.SolverInterface

export DrsMathProgModel,
    DrsMathProgSolver,
    loadproblem!,
    optimize!,
    status,
    getreducedcosts,
    getconstrduals,
    getobjval,
    getsolution

immutable DrsMathProgSolver <: AbstractMathProgSolver
    options
end
DrsMathProgSolver(; kwargs...) = DrsMathProgSolver(kwargs)

type DrsMathProgModel <: AbstractLinearQuadraticModel
    A               # constraint coefficients
    b               # RHS
    c               # objective coefficients
    basis           # basis variables
    nonbasis        # nonbasis variables
end
LinearQuadraticModel(s::DrsMathProgSolver) = DrsMathProgModel(; s.options...)

function setparameters!(m::Union{DrsMathProgSolver, DrsMathProgModel}; kwargs...)
    for (option, value) in kwargs
        if option == :TimeLimit
            println("WARNING: TODO: $option")
        elseif option == :Silent && value == true
```

```
            Logging.configure(level=OFF)
        elseif option == :LogLevel
            Logging.configure(level=value)
        else
            println("WARNING: $option is unsupported")
        end
    end
end

function DrsMathProgModel(; kwargs...)
    m = DrsMathProgModel(0, 0, 0, 0, 0)
    setparameters!(m; kwargs...)
    m
end

function loadproblem!(m::DrsMathProgModel, A, l, u, c, lb, ub, sense)
    @debug("loadproblem!: A $A, l $l, u $u, c $c, lb $lb, ub $ub, sense $sense")
    m.A = A
    m.b = zeros(size(ub))
    m.c = c

    DrsTransformToStandardForm!(m, lb, ub, sense)

    r, c = size(m.A)
    m.basis = zeros(Int, r)

    DrsFindPotentialBasis!(m)
end

function DrsFindPotentialBasis!(m::DrsMathProgModel)
    r, c = size(m.A)
    for ic in 1:c
        column = m.A[:,ic]
        if countnz(column) == 1
            ir = findfirst(x -> x == 1, column)
            if ir != 0 && m.basis[ir] == 0
                # add the column if current row has not been selected
                m.basis[ir] = ic
            end
        end
    end
    m.nonbasis = setdiff(1:c, m.basis)
end

function DrsTransformToStandardForm!(m::DrsMathProgModel, lb, ub, sense)
    @assert length(lb) == length(ub) """the lengths of
```

```
    lower bounds ($(length(lb))) and
    upper bounds ($(length(ub))) are different"""

    r, c = size(m.A)

    # check if b is negative

    for i in 1:length(lb)
        if lb[i] == typemin(typeof(lb[i]))
            # <
            if ub[i] < 0
                m.A[i,:] = -m.A[i,:]
                lb[i], ub[i] = -ub[i], -lb[i]
            end
        elseif ub[i] == typemax(typeof(ub[i]))
            # >
            if lb[i] < 0
                m.A[i,:] = -m.A[i,:]
                lb[i], ub[i] = -ub[i], -lb[i]
            end
        else
            # =
            if lb[i] < 0
                m.A[i,:] = -m.A[i,:]
                lb[i], ub[i] = -ub[i], -lb[i]
            end
        end
    end

    # add variables
    surplus = []

    for i in 1:length(lb)
        if lb[i] == typemin(typeof(lb[i]))
            # <, add slack
            m.A = [m.A zeros(r, 1)]
            m.A[i, end] = 1
            m.b[i] = ub[i]
            m.c = [m.c; 0]
        elseif ub[i] == typemax(typeof(ub[i]))
            # >, add surplus
            m.A = [m.A zeros(r, 1)]
            m.A[i, end] = -1
            m.b[i] = lb[i]
            m.c = [m.c; 0]
            push!(surplus, i)
```

```
        end
    end

    # =, add artificial
    for i in 1:length(lb)
        if lb[i] != typemin(typeof(lb[i])) && ub[i] != typemax(typeof(ub[i]))
            m.A = [m.A zeros(r, 1)]
            m.A[i, end] = 1
            m.b[i] = lb[i]
            m.c = [m.c; 0]
        end
    end

    # add artificial for surplus
    for i in surplus
        m.A = [m.A zeros(r, 1)]
        m.A[i, end] = 1
        m.c = [m.c; 0]
    end

    if sense == :Max
        m.c = -m.c
    end
end

function optimize!(m::DrsMathProgModel)
    @debug("A $(m.A)")
    @debug("b $(m.b)")
    @debug("c $(m.c)")
    @debug("basis $(m.basis)")
    @debug("nonbasis $(m.nonbasis)")

    invB = SharedArray(typeof(B[1]), size(B),
        init = S -> S[linearindices(B)] = inv(B)[linearindices(B)])
    @debug("B $B")

    N = m.A[:,m.nonbasis]
    @debug("N $N")

    invB = inv(B)
    @debug("invB $invB")

    basic_vars = invB * m.b
    @debug("basic_vars $basic_vars")

    terminate = false
```

```
    while !terminate
        P = CHUZR(invB, basic_vars)
        @debug("P $P")

        pivotal_rows = []
        @sync begin
            for p in P
                pi = @spawn BTRAN(invB, p)
                pivotal_row = @spawn PRICE(N, fetch(pi))
                push!(pivotal_rows, fetch(pivotal_row))
            end
        end

        while !isempty(P)
            p = CHUZR_MI(P)
            @debug("p $p")
            q = CHUZC(m.c[m.nonbasis], pivotal_rows[1])
            @debug("q $q")
            UPDATE_MI()
        end

        basis_change = []
        for change in basis_change
            FTRAN1()
            FTRAN2()

            magic_condition = false

            if magic_condition
                INVERT()
            else
                UPDATE()
            end
        end

        terminate = true
    end
end

function status(m::DrsMathProgModel)
    @debug("status")
end

function getreducedcosts(m::DrsMathProgModel)
    @debug("getreducedcosts")
end
```

```
function getconstrduals(m::DrsMathProgModel)
    @debug("getconstrduals")
end

function getobjval(m::DrsMathProgModel)
    @debug("getobjval")
end

function getsolution(m::DrsMathProgModel)
    @debug("getsolution")
end

end
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Amdahl, G. Validity of the single-processor approach to achieving large scale computing capabilities. / G.M. Amdahl. —AFIPS Press, Reston, Va., 1967. —pp. 483-485.
2. Balbaert, I. Julia: High Performance Programming / I. Balbaert, A. Sengupta, M. Sherrington. —2016. —697 pages.
3. Boffley, T. Implementing Parallel simplex algorithms. / T.B. Boffley, R. Hay. —Cambridge University Press, 1989. —pp. 169-176.
4. Cvetanovic, Z. Efficient decomposition and performance of parallel pde, fft, monte-carlo simulations, simplex, and sparse solvers. / Z. Cvetanovic, E.G. Freedman, C. Nofsinger // Journal of Supercomputing. — 1991. — no. 5. —P. 19–38.
5. Dantzig, G. Linear Programming and Extensions / G.B. Dantzig. — Princeton, NJ: Princeton Univ. Press., 1963.
6. Dantzig, G. B. Programming in a linear structure / G. B. Dantzig // Washington, Comptroller, USAF. —1948.
7. Data-parallel implementations of dense simplex mmethod on the connection machine cm-2. / J. Eckstein, I.I. Boduroglu, L. Polymenakos, D. Goldfarb // ORSA Journal on Computing. —1995. —no. 7. —P. 402–416.
8. An empirical evaluation of the korbx algorithms for military airlift applications / W. J. Carolan, J. E. Hill, J. L. Kennington [et al.] // Operations Research, 38. —1990. —P. 240–248.
9. Finkel, R. Large-grain parallelism – three case studies. / R.A. Finkel ; Ed. by L.H. Jamieson, D. Gannon, R.J. Douglas. —MIT Press, Cambridge, MA, 1987. —pp. 21-63.
10. Forrest, J. Steepest-edge simplex algorithms for linear programming. / J.J. Forrest, D. Goldfarb // Mathematical Programming. — 57:341–374, 1992.
11. Four vector-matrix primitives. / A. Agrawal, G.E. Blelloch, R.L. Krawitz, C.A. Phillips // ACM Symposium on parallel Algorithms and Architectures. —1989. —P. 292–302.

12. Gay, D. M. Electronic mail distribution of linear programming test problems / D. M. Gay // Mathematical Programming Society COAL Newsletter, 13. —1985. —P. 10–12.
13. Goldfarb, D. A practical steepest-edge simplex algorithm. / D. Goldfarb, J.K. Reid // Mathematical Programming. —12:361–371, 1977.
14. Hall, J. Towards a practical parallelisation of the simplex method / J.A.J. Hall // Computational Management Science. — 7 (2010), pp. 139–170.
15. Hall, J. A high performance dual revised simplex solver / J.A.J. Hall, Q. Huangfu // Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics. — Vol. Part I, PPAM'11, – Berlin, Heidelberg, 2012. Springer-Verlag. – pp. 143–151.
16. Harris, P. Pivot selection method of the devex lp code. / P.M.J. Harris // Mathematical Programming. —5:1–28, 1973.
17. Introduction to Parallel Computing: Design and Analysis of Algorithms. / V. Kumar, A. Grama, A. Gupta, G. Karypis. — 2nd edition. — Addison-Wesley, 2003.
18. Karl-Heinz, B. The simplex method: A probabilistic analysys. / B. Karl-Heinz // Berlin: Springer-Verlag. —1987. —Vol. 1.
19. Kwon, C. Julia Programming for Operations Research: A Primer on Computing / C. Kwon. —2016. —246 pages.
20. Lougee-Heimer, R. The coin-or initiative: Open source accelerates operations research progress / R. Lougee-Heimer // ORMS Today, 28. —2001. —P. 20–22.
21. Luo, J. Linear programming on transputers. / J. Luo, G.L. Reijns // Algorithms, Software, Architecture / Ed. by J. van Leeuwen. —Vol. A-12. —Elsevier, 1992. —P. 525–534.
22. A practical anti-cycling procedure for linear constrained optimization. / P.E. Gill, W. Murray, M.A. Saunders, M.H. Wright // Mathematical Programming. —45:437–474, 1989.
23. Rohit, J. Julia Cookbook / J.R. Rohit. — Packt Publishing, 2016. — 172 pages.

24. Schrijver, A. Theory of linear and integer programming. / A. Schrijver // John Wiley & sons. —1998.
25. Seven More Language in Seven Weeks / B. Tate, F. Daoud, J. Moffit, I. Dees. —The Pragmatic Programmers, 2014. —350 pages.
26. Some computation results on mpi parallel implementation of dense simplex method. / E.-S. Badr, M. Moussa, K. Papparrizos [et al.] // Transactions on Engineering, Computiong and Technology. — 2006. —December. — no. 17. —P. 228–231.
27. Stunkel, C. Linear optimization via message-based parallel processing. / C.B. Stunkel // International Conference on Parallel Processing. —Vol. III. —August 1988. —P. 264–271.
28. A survey of Parallel algorithms for linear programming. / J. Luo, G.L. Reijns, F. Bruggeman, G.R. Lindfield ; Ed. by E.F. Deprettere, A.J. van der Veen. —Elsevier, 1991. —Vol. B. —pp. 485-490.
29. Thomadakis, M. An efficient steepest-edge simplex algorithm for simd computers. / M.E. Thomadakis, J.-C. Liu // International Conference on Supercomputing. —1996. —P. 286–293.
30. Yarmish, G. A Distributed Implementation of the Simplex Method. : Ph. D. thesis / G. Yarmish ; Polytechnic University. — Brooklyn, NY : 2001. — March.
31. Zenios, S. Parallel numerical optimization: current status and annotated bibliography. / S.A. Zenios // ORSA Journal on Computing. —1989. —no. 1(1). —P. 20–43.
32. Бабаев, Д. Параллельный алгоритм решения задач линейного программирования. / Д.А. Бабаев, С.С. Марданов // Журнал Вычислительной Математики и Математической Физики. —1991. —№ 31. – С. 86-95.