СОДЕРЖАНИЕ

СОДЕРЖАНИЕ ... 3

ВВЕДЕНИЕ ... 5

Обзор ... 6

MPI и другие сетевые библиотеки ... 6

Charm++ ... 8

LuNA ... 8

DVM ... 9

Результаты анализа ... 10

Цель работы ... 11

Требования ... 11

Задачи ... 11

Термины и определения ... 12

1 Аналитический обзор узких мест алгоритмов LuNA ... 14

1.1 Неэффективность конструирования ОФУ ... 14

1.2 Неэффективность порядка исполнения АФВ ... 15

1.3 Интерпретация LuNA программы ... 16

1.4 Освобождение памяти ... 16

1.5 Вывод ... 16

2 Предлагаемые решения ... 18

2.1 Алгоритм конструирования ОФУ для зависимых цепочек ... 18

2.1.1 Изложение алгоритма ... 18

2.1.2 Анализ предложенного решения ... 21

2.2 Конструирование ОФУ и приоритетов АФВ генетическим алгоритмом ... 23

2.2.1 Выбор подхода ... 23

2.2.2 Функция приспособленности ... 24

2.2.3 Генетический алгоритм ... 26

2.2.4 Анализ предложенного решения ... 27

2.3 Другие предлагаемые решения ... 29

3 Реализация routing_table_pathfinder и chains_routing_table ... 31

3.1 chains_routing_table ... 31

3.2 gen_routing_table ... 32

3.3 routing_table_pathfinder ... 32

4 Тестирование ... 33

ЗАКЛЮЧЕНИЕ ... 35

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ ... 37

ПРИЛОЖЕНИЕ А ... 39

ПРИЛОЖЕНИЕ Б ... 40

ПРИЛОЖЕНИЕ В ... 41

ВВЕДЕНИЕ

Сегодня в науке актуальны задачи, требующие сложных вычислений над большим объемом данных. Нет возможности производить такие вычисления на одном компьютере: требуются мультикомпьютеры, состоящие из сотен и тысяч вычислительных узлов, объединенных сетью. Однако программирование программ, использующих такой большой объем вычислителей, представляет сложность, ведь программе приходится не только заниматься вычислениями, но и управлять этими вычислениями: решать задачи распределения ресурсов, балансировки нагрузки, синхронизации и другие. Кроме того, как правило, затруднено переиспользование таких программ в других программах.

Высокопроизводительные программы численного моделирования возможно условно разбить на 2 части:

- Прикладная часть - производит вычисления, связанные с прикладной областью. Как правило, это главная часть, определяющая цель создания программы.
- Системная часть - управляет вычислениями. Как правило, эта часть не является самоцелью, но появляется впоследствии, чтобы обеспечить нужные качества вычислений, главным образом - производительность.

С усложнением вычислителей и увеличением объема задачи системная часть имеет тенденцию к усложнению.

Есть потребность в абстрагировании от программистов системной части в больших научных расчетах, по следующим причинам:

- Программирование системной части в больших научных расчетах - сложная задача, потому что большие вычислительные задачи не решаются на одиночных вычислителях, но решаются на мультикомпютерах. Разработка программ распределенных высокопроизводительных вычислений вручную требует большого объема знаний об особенностях используемого мультикомпьютера, а также навыков системного программирования и программирования распределенных высокопроизводительных программ.
- У физиков, биологов и других ученых не всегда есть время и знания для программирования системной части.
- Рост сложности системной части при увеличении объема задачи (например, увеличении размерности) и усложнении мультикомпьютера, что становится препятствием при решении объемных задач.

Так как в научных расчетах востребованы в первую очередь численные методы, требуется средство программирования распределенных высокопроизводительных программ, обеспечивающее простоту программирования численных методов, производительность мультикомпьютера на численных методах. Возможность программирования задач за пределами численных методов также имеет значение.

Одна из основных причин, из-за которой такого средства до сих пор нет, заключается в том, что абстрагирование от программиста системной части приводит, в общем случае, к неэффективности системной части в смысле большого объема накладных расходов, появляющихся из-за универсальности используемых алгоритмов в абстрагированной системной части. Универсальные алгоритмы не всегда обеспечивают необходимую производительность мультикомпьютера на численных методах.

Исследуемая в данной работе проблема заключается в том, что абстрагирование от программиста системной части приводит к ее неэффективности.

Возможны различные способы достижения высокой производительности на суженном до численных методов классе задач при сочетании с высоким уровнем абстракции. Один из них - разработка и реализация алгоритмов конструирования частных управляющих схем (частная управляющая схема - специальный системный алгоритм) для частных классов задач численных методов, что является актуальной темой и темой данной работы.

Обзор

Необходимо обозреть актуальные наработки в области поставленной проблемы. В настоящем разделе будут рассмотрены различные средства программирования высокопроизводительных распределенных программ численных методов с высоким уровнем абстракции.

MPI и другие сетевые библиотеки

Существует множество сетевых библиотек, протоколов и препроцессоров, упрощающих программирование сетевых приложений в целом и высокопроизводительных распределенных программ в частности.

Например: MPI [18], ZeroMQ [20], boost [14], ZeroC ICE [17], Akka [13].

Такие продукты могут реализовывать различные сетевые архитектурные подходы. Например:

- Близкий к системным вызовам операционных систем (пример: ZeroMQ, boost) - предоставляют большую эффективность и универсальность, хотя и представляют

собой более удобную “обертку” над системными вызовами. Во многих частных случаях существуют менее трудоемкие подходы.

- Publisher-subscriber (пример: ZeroMQ): описывает декомпозицию системы на производителей и потребителей. Производители отправляют сообщения без знания об особенностях своих подписчиков. Подписчики принимают сообщения, отправляемые потребителями. Определены “подписки” между потребителями и сообщениями производителей. Такой подход в некоторых случаях позволяет масштабировать вычислительное приложение по узлам, потому что производителя не используют знаний о потребителях. Тем не менее, этот подход в высокопроизводительных вычислениях имеет ограничения, и, не определяет специфических для вычислений подходов.
- Модель акторов (пример: Akka) [9] - описывает декомпозицию системы на акторы, которые являются универсальными примитивами параллельного численного расчёта. Сетевой контракт актора ограничивается возможностью асинхронно принимать сообщения от других акторов, асинхронно передавать сообщения другим акторам, создавать акторов и иметь произвольное состояние. Такая модель может обеспечивать масштабирование, потому что акторы, как правило, не используют информацию об узле, на котором находится получатель сообщения. Тем не менее, этот подход не всегда удобен для программирования, например, численных методов, потому что язык описания модели акторов далек от языка описания численных методов. Может быть частным случаем publisher-subscriber.
- RPC (Remote Procedure Call, удаленный вызов процедур, пример: ZeroC ICE): этот подход позволяет вызывать процедуры в другом адресном пространстве, передавая информацию по сети. Эти вызовы могут быть как синхронными, так и асинхронными. Этот подход также может использоваться и используется в высокопроизводительных вычислениях, но, без дополнительной логики этот подход слишком общий: например, не определяет распределение ресурсов.

В контексте распределенных вычислений отдельного внимания заслуживают MPI[18] (Message Passing Interface) библиотеки, потому что это одно из самых популярных средств для высокопроизводительных распределенных вычислений в науке. MPI - это интерфейс C/C++ функций, имеющий несколько реализаций в различных библиотеках.

Эти функции предназначены для обмена данными, упрощения адресации сообщений. MPI имеет ряд сетевых операций и упрощений, хорошо подходящих для программирования

численных методов в частности и расчетов вообще. Например: коллективные операции reduce, barrier, broadcast, gather, scatter и другие.

Кроме того, MPI приспособлен для программирования сетевого взаимодействия в различных мультикомпьютерах. Например, MPI предоставляет возможность для описания топологии сети, инкапсулирует создание соединений между различными узлами.

Стоит упомянуть, что MPI не имеет средств для параллельного программирования в системе с общей памятью, поэтому часто используется в сочетании со средствами параллельного программирования в системе с общей памятью (например: OpenMP [19]). Это существенный недостаток, так как усложняет программирование системной части.

MPI, как и другие схожие библиотеки, не абстрагирует системную часть полностью, а только предоставляет более удобные средства для ее программирования.

Charm++

Charm++ [15] - система поддержки параллельных вычислений в параллельной среде: объектно-ориентированный язык программирования на базе C++, компилятор и runtime система.

Программа, написанная на языке Charm++, описывает модель акторов, где акторы называются чарами (chare). Краткое описание модели акторов возможно найти в обзоре сетевых библиотек или, например, в [9].

Распределение чаров по узлам происходит в runtime системе Charm++, и, не использует широкий статический анализ программы. Runtime система Charm++ реализует динамическую балансировку нагрузки между узлами. В дополнение, Charm++ оставляет возможность для программиста самостоятельного распределения чаров между узлами.

Хотя Charm++ и обеспечивает довольно высокий уровень абстракций (прозрачное распределение и балансирование чаров по вычислительным узлам), описание задачи в терминах модели акторов и в терминах математических моделей мало совпадают, что вызывает сложности при программировании численных методов. Кроме того, Charm++ не использует статический анализ текста программы для конструирования распределения чаров на узлы - что оказывается критичным при решении некоторых задач.

LuNA

Проект LuNA тесно связан с технологией фрагментированного программирования (ТФП)[11, 12], разрабатываемой в лаборатории синтеза параллельных программ Института Вычислительной Математики и Математической Геофизики Сибирского Отделения Российской Академии Наук. ТФП является подходом в параллельном программировании,

нацеленным на автоматизацию реализации динамических свойств параллельных программ, и обеспечению переносимости программ между различными мультикомпьютерами. Одно из главных применений ТФП - реализация больших численных моделей.

Фрагментированные программы представляются состоящими из множества зависимых задач - фрагментов вычислений (ФВ), динамически размещаемых на узлах мультикомпьютера. ФВ не имеют состояния и побочных эффектов, поэтому возможно передавать ФВ по сети, не теряя корректности исполнения программы. Данные, передаваемые между ФВ, называются фрагментами данных (ФД).

Зависимости между ФВ и ФД могут быть 2-х типов:

- ФВ является потребителем ФД
- ФВ является источником ФД

Структура LuNA программы, включая зависимости между ФВ и ФД, определяется специальным декларативным функциональным языком, называемым LuNA.

Исполняет LuNA программы специальная runtime система, выполняющая, как и Charm++, динамические распределение и балансировку ФВ и ФД между вычислительными узлами.

LuNA система мало учитывает данные статического анализа LuNA программы для динамического распределения и балансировки нагрузки ФВ и ФД в текущей реализации, применяя универсальные алгоритмы.

В целом, LuNA предоставляет подходящие для численных методов абстрактные средства, и, позволяет описывать любые параллельные программы. Однако имеет недостаточную производительность на задачах численных методов.

DVM

Distributed Virtual Machine/Memory (DVM) [16] является расширением языков C и Fortran и реализуется в форме директив компилятору.

После этапа препроцессинга эти директивы преобразуются в директивы OpenMP, операторы MPI и CUDA.

Стоит отметить, что исходный код с DVM директивами, как и исходный код с OpenMP [19] директивами, может быть скомпилирован компилятором C или Fortran, игнорируя директивы DVM.

DVM позволяет осуществлять параллельное выполнение циклов, запуск параллельных задач на нескольких узлах или внутри одного узла.

DVM содержит ряд директив для описания распределения данных по высокоуровневым или низкоуровневым правилам. DVM не может построить эти правила без

участия программиста. DVM требует явных директив для работы с теневыми гранями. Теневой гранью называют множество данных при декомпозиции вычислений на N-мерной сетке, которое требуется передать между соседними узлами или вычислителями.

Как и OpenMP, DVM поддерживает операции редукции.

Хотя такую степень абстракции, в сравнении с LuNA, и нельзя назвать очень высокой, DVM отлично подходит для решения целого ряда задач, обходя LuNA и Charm++ по производительности на многих задачах численных методов. Тем не менее, DVM имеет следующие недостатки в контексте темы работы:

- Так как DVM - расширение последовательных процедурных языков, он недостаточно хорошо скрывает системную часть в численных методах.
- DVM - расширение последовательных процедурных языков, что кардинально усложняет статический анализ.
- DVM не балансирует задачи динамически.

Результаты анализа

Анализ различных средств программирования высокопроизводительных распределенных программ показал, что среди них нет средства, сочетающего высокое абстрагирование от системной части на задачах численных методов и высокую производительность на задачах численных методов. На таблице 1 представлено краткое сравнение:

Таблица 1 – Краткое сравнение средств программирования высокопроизводительных распределенных программ

	MPI и другие сетевые библиотеки	Charm++	DVM	LuNA
Абстрагирование системной части на задачах численных методов	-	+-	-+	+
Производительность распределенных вычислений на задачах численных методов	+	-+	+-	-

Также анализ показал, что некоторые рассмотренные проекты возможно использовать как базу для решения поставленной проблемы.

Технология фрагментированного программирования, и, LuNA в частности - подходящая база для вклада в решение поставленной проблемы по следующим причинам:

- Возможности статического анализа языка LuNA широки из-за естественного параллелизма языка, иммутабельности данных и отсутствия состояния у задач, единственности исполнения задач, наличия идентичности у задач на уровне языка.
- Так как LuNA использует наиболее близкий способ описания вычислений к численным методам, в контексте численных методов LuNA является самым высокоуровневым средством из имеющихся.

Цель работы

LuNA имеет недостаточную производительность на задачах численных методов в распределенных вычислениях, вследствие применения универсальных алгоритмов. В случае, если LuNA будет иметь достаточную производительность на задачах численных методов, LuNA сделает существенный вклад в решение поставленной проблемы.

Цель данной работы - разработка и реализация алгоритмов автоматического конструирования частных управляющих схем, позволяющих снизить накладные расходы LuNA на задачах численных методов.

Требования

- Практическая значимость: существенное улучшение производительности LuNA на задачах численных методов.
- Интегрированность в систему существующих средств оптимизации LuNA.

Задачи

- Локализовать узкие места в алгоритмах LuNA.
- Разработать алгоритмы конструирования частных управляющих схем, снижающие накладные расходы LuNA в узких местах алгоритмов LuNA.
- Реализовать и протестировать часть разработанных алгоритмов конструирования частных управляющих схем, снижающих накладные расходы LuNA в узких местах алгоритмов LuNA.

Термины и определения

Так как цель работы связана с LuNA, необходимо определить терминологию, связанную с LuNA, и прочую терминологию:

- Имя фрагмента - уникальное имя, состоящее из строки (именуемой базовым именем или basename) и вектора чисел (именуемых индексами).
- Фрагмент данных (ФД) - иммутабельная структурированная переменная, имеющая имя фрагмента.
- Фрагмент вычислений (ФВ) - применение процедуры, принимающей на вход фрагменты данных и производящей на выход фрагменты данных. Каждый фрагмент вычислений имеет идентичность (имя фрагмента), не имеет побочных эффектов при исполнении и может быть исполнен только один раз.
- Фрагмент - ФД или ФВ.
- Оператор ФВ - оператор LuNA программы, обозначающий вызов ФВ
- Оператор ФД - оператор LuNA программы, обозначающий связь между ФД и ФВ вида: ФВ - потребитель ФД или ФВ - источник ФД.
- Оператор фрагмента - оператор ФВ или оператор ФД.
- Объявление ФД - оператор языка над basename B внутри оператора фигурных скобок, определяющий, что внутри оператора фигурных скобок будут использоваться ФД с basename B.
- Выражение - дерево операторов языка (+, -, * и другие) над операторами ФД, используемое для вычисления значение-аргумента оператора ФВ.
- Атомарный ФВ (АФВ) - ФВ, имеющий конечное заданное множество входных и выходных ФД. Вычисление атомарного ФВ является вызовом функции C++, использующей C++ интерфейс атомарных ФВ LuNA.
- Оператор АФВ - оператор атомарного ФВ.
- А-фрагмент - ФД или АФВ.
- Оператор фигурных скобок - оператор ФВ, раскрывающий свое структурированное тело: объявления ФД, другие операторы ФВ.
- Оператор примитивной рекурсии (цикл for) - ФВ, содержащий 3 параметра: целочисленное начало, целочисленный конец и оператор ФВ. Раскрывает оператор ФВ на каждой итерации, вычисляя все итерации от параметра начала до параметра конца. Началом и концом может быть константа, ФД или выражение.
- Оператор минимизации (цикл while) - ФВ, содержащий 4 параметра: выражение-предикат, целочисленное начало цикла, имя выходного ФД, оператор ФВ. Алгоритм

исполнения предполагает исполнение итераций до выполнения предиката, и, описан более подробно в приложении A.

- Условный оператор (if) - ФВ, содержащий 2 параметра: предикат и оператор ФВ. Предикатом может быть выражение, ФД или константа. В зависимости от истинности предиката оператор раскрывает или не раскрывает оператор ФВ.
- Рекомендации - директивы в LuNA, по аналогии с директивой #pragma в C, позволяют программисту передать информацию компилятору LuNA. Вводятся разработчиками LuNA по мере необходимости.
- Отображение фрагментов на узлы (ОФУ) - задание расположения фрагментов на вычислительных узлах.
- Приоритет АФВ - свойства АФВ, определяющие приоритет исполнения АФВ. Если на узле доступно для исполнения несколько АФВ, в первую очередь будут исполнены АФВ с большим приоритетом.
- Запрос ФД - сообщение, пересылаемое от узла A к узлу B, означающее: переслать ФД D от узла B на узел A.
- Упреждающая передача ФД - передача ФД до запроса ФД.
- Интерпретация LuNA программы - процесс интерпретации операторов LuNA языка runtime системой.
- Степень параллелизма - количество доступных ФВ для исполнения в некоторый в заданный момент.
- Разворачивание цикла (loop unrolling) - построение цепочки операторов ФВ, содержащей все итерации заданного цикла. Заданный цикл должен иметь константное число итераций, известное на этапе компиляции.
- loop tiling - построение нового цикла, в котором увеличено количество итераций заданного цикла, выполняемых за одну итерацию нового цикла.
- Функция приспособленности (fitness function, генетический алгоритм) [6] - функция оценки, определяющая меру качества особи.
- Эффективность – эффективность вычислений всегда будет пониматься в смысле высокой полезной загруженности вычислителей. Полезная нагрузка отличается тем, что относится к прикладной части и не может быть, в неформальном смысле, оптимизирована.

1 Аналитический обзор узких мест алгоритмов LuNA

Необходимо рассмотреть различные узкие места в алгоритмах LuNA с позиции 2-х характеристик:

- Величина возникающих накладных расходов на задачах численных методов.
- Наличие возможности у программиста (пользователя) устранить эти накладные расходы с помощью обходных путей и оценка простоты этой возможности.

Оценка по этим 2-м характеристикам необходима для приоритезации локализованных узких мест и построения дальнейших планов исследования. Пояснение к 2-му параметру: при сравнимых накладных расходах узкое место, которое нельзя или сложно закрыть обходным путем, должно иметь более высокий приоритет с позиций практической значимости.

1.1 Неэффективность конструирования ОФУ

ОФУ в LuNA определяет модуль pathfinder, который имеет несколько реализаций, применяемых в зависимости от параметров запуска:

- hash - фрагменты отображаются на узлы с помощью хэш-функции. Используется по умолчанию.
- linear - фрагменты отображаются на линейку узлов по значению (последнего) индекса.

По умолчанию используется hash pathfinder, отображающий фрагменты хаотично.

Рассмотрим проблему эффективности ОФУ на основе хэш-функции в программах LuNA на примере следующего графа исполнения LuNA программы на рисунке 1, выполняющей упрощенный map-reduce:

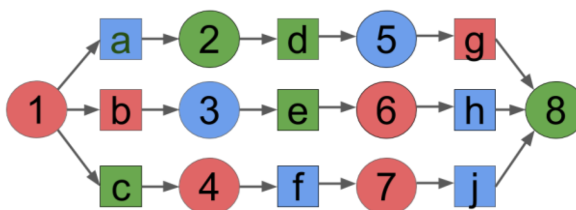

Рисунок 1 – Граф исполнения, где круг - АФВ, квадрат - ФД. Направленная стрелка означает зависимость между АФВ и ФД (см. обзор LuNA). Цвет элемента обозначает узел, на который отображен описываемый АФВ или ФД

Передача ФД к АФВ на разных узлах означает передачу данных по сети. На примере выше таких передач 34: на всех стрелках, кроме 1-b и 2-d.

Рассмотрим этот же граф, но с другим ОФУ, на рисунке 2:

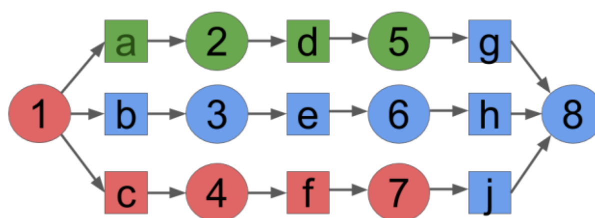

Рисунок 2 – Граф исполнения с другим ОФУ

Здесь передач данных 4: 1-a, 1-b, 5-g, 7-j.

Эти примеры иллюстрируют зависимость количества и объема пересылок данных между узлами от ОФУ и неэффективность построения ОФУ в программах LuNA на основе хэш-функции во многих случаях.

У программиста есть только два способа повлиять на ОФУ: выбор pathfinder между hash и linear и реализация собственного pathfinder. linear pathfinder на некоторых задачах строит эффективное ОФУ, однако класс этих задач относительно узкий. Кроме того, исходя из его определения, результат его работы напрямую зависит от таких специфических особенностей, как границы циклов, поэтому даже на подходящей задаче результат может быть плохим. И хотя у программиста и есть возможность несколько улучшить ОФУ, оптимизируя свою программу с учетом знания о том, что будет применен linear pathfinder, это требует от программиста понимания принципов работы linear pathfinder и причин, по которым он может быть эффективным. Другими словами, это в некоторой степени противоречит абстрагированию от системной части.

Как показывает практика, не системным программистам этими знаниями овладеть трудно, поэтому у физиков, биологов и других ученых этих знаний, как правило, нет.

1.2 Неэффективность порядка исполнения АФВ

Степень параллелизма в момент времени в общем случае зависит от выбора порядка исполнения АФВ. Это свойство проще всего рассмотреть на примере на псевдокоде:

```
1.   d0 = f0() // вызов АФВ f0 без аргументов, результат в ФД d0
2.   d1 = f1()
3.   for i in [0, 9]
4.       g1[i](d1) // вызов АФВ g1[i] с аргументом d1
```

Если не исполнено (и не исполняется) ничего, степень параллелизма равна 2 (доступны f0 и f1).

Если исполнено только f0, степень параллелизма будет равна 1 (f1).

Если исполнено только f1, степень параллелизма будет равна 11 (f1, g1[0], g1[1], …, g1[9])

В LuNA есть рекомендации, позволяющие ограничить возможный порядок исполнения операторов АФВ вручную. Понимание, где их применять, строго говоря, требует от программиста понимания особенностей вычислений в LuNA.

1.3 Интерпретация LuNA программы

Интерпретация LuNA программы связана с накладными расходами на интерпретацию операторов языка, поиск АФВ готовых к исполнению и многие другие задачи. Общие правило таково: если среднее время исполнения АФВ велико, то относительная величина этих накладных расходов будет незначительной.

Стоит отметить, что у программиста есть возможность уменьшить количество АФВ, увеличив объем обрабатываемых данных каждым АФВ.

Однако это уменьшает параллелизм в LuNA программе. Кроме того, иногда это увеличивает трудоемкость программирования LuNA программы, потому что, как правило, это связано с добавлением дополнительных циклов в АФВ для обработки большего множества данных.

1.4 Освобождение памяти

В LuNA есть узкое место, связанное с запоздалым освобождением памяти, что приводит к записи неактуальных ФД на диск. В LuNA есть рекомендации, позволяющие явно освобождать ФД после исполнения некоторого оператора АФВ.

1.5 Вывод

Узкое место в эффективности ОФУ имеет первостепенное практическое значение, так как делает невозможным на многих актуальных задачах численных методов использование большого количества вычислительных узлов из-за накладных расходов, объем которых зависит от количества вычислительных узлов.

Второе по практическому значению узкое место - выбор порядка исполнения АФВ. Ограничивая уровень параллелизма, это узкое место также может приводить к невозможности использования большого количества вычислительных узлов. Более того, сравнивая это узкое место с узким местом эффективности ОФУ, важно отметить, что ограниченный уровень параллелизма - более существенная проблема, потому что максимальное число используемых узлов ограничено степенью параллелизма. Классы задач, на которых эти узкие места имеют практическое значение, сравнимы. Однако, в LuNA есть

рекомендации для управления порядком исполнения операторов АФВ, поэтому это узкое место имеет меньшее практическое значение.

Другие рассмотренные узкие места также имеют практическое значение.

2 Предлагаемые решения

Настоящая глава содержит предлагаемые решения на основе статического и динамического анализа LuNA программы. Опорным направлением анализа станет анализ зависимостей по ФД в программе.

2.1 Алгоритм конструирования ОФУ для зависимых цепочек

В настоящем разделе пойдет речь о статическом анализе LuNA программы, построении дерева зависимостей между операторами АФВ и упрощенном конструировании ОФУ на основе этих данных для подпрограмм без циклов.

Первый подраздел 2.1.1 посвящен изложению предлагаемого решения.

Следующий и последний подраздел 2.1.2 посвящен анализу предложенного решения.

2.1.1 Изложение алгоритма

2.1.1.1 Преобразование оператора if

Для упрощения дальнейшего описания статического анализа следует описать преобразование оператора if в тело if (ФВ) и АФВ:

if возможно представить как тело if и АФВ, имеющий в качестве входных аргументов все входные аргументы if и один выходной аргумент, принимаемый на вход телом if.

Новый АФВ будет вырабатывать выходной ФД в случае истины предиката и никогда не вырабатывать в случае лжи. Таким образом, тело if - ФВ - будет исполнено в случае истинности предиката и не может быть исполнено в случае ложности предиката.

Однако такое преобразование не является корректным с точки зрения соглашений LuNA: в LuNA определено требование, что все ФВ должны завершиться. Это преобразование противоречит этому требованию. Однако для дальнейшего статического анализа это не имеет значения.

2.1.1.2 Вводимые термины

Все а-фрагменты, согласно определению имени а-фрагмента, возможно разбить на 2 типа:

- Неиндексированные а-фрагменты: имеют только базовое имя (basename) и не имеют индексов.
- Индексированные а-фрагменты: имеют basename и положительное число индексов.

Зависимость между операторами АФВ - кортеж из 3-х элементов:

- from - оператор АФВ, являющийся источником.
- to - оператор АФВ, являющийся потребителем.
- with - операторы ФД, для которых from - источник и to - потребители.

Зависимость между между оператором АФВ и оператором ФД - кортеж из 2-х элементов:

- from - оператор АФВ, являющийся источником.
- with - один оператор ФД, для которого from - источник.

Введем следующую семантику зависимостей, включающую 2 типа:

- Прямая зависимость - зависимость, истинная всегда.
- Возможная зависимость - отсутствие возможной зависимости доказывает невозможность прямой зависимости.

2.1.1.3 Базовые особенности анализа индексированных а-фрагментов

В анализе индексированные ФД значительно сложнее, и, в некоторых случаях, невозможно установить на этапе компиляции какой АФВ станет источником заданного ФД, не анализируя возможного значения индексов. Однако, всегда возможно установить возможную зависимость по basename.

Пример на псевдокоде:

1. y0 = f0()
2. y1 = f1()
3. x = f2()
4. a[y0] = g0()
5. a[y1] = g1()
6. problem(d[x]) // c == x или c == y

Невозможно установить, не анализируя содержимого АФВ выше, какое точное имя фрагмента у d[x]. Но возможно установить, что d[x] имеет basename d, поэтому АФВ problem возможно зависит от g0 или g1 по некоторым данным с basename d.

2.1.1.4 Построение графа зависимостей (ГЗ)

Результатом построения графа зависимостей станет граф, элементами которого являются операторы АФВ из дерева разбора программа, а ребра - зависимости по ФД между элементами графа.

Построение ГЗ между операторами программы - не новая задача, описанная в публикациях, например: [3].

2.1.1.5 Базовые особенности конструирования ОФУ

Заметим, что существуют сложности с формализацией отображения ФД на узлы в виде конечного набора правил.

Рассмотрим эти сложности на примере на псевдокоде:

1. b = g()
2. d[b] = h()
3. f(a[b])

Предположим, что стоит задача отображения ФД d[b] на узел N.

В этом примере возможно задать отображение следующими правилами на этапе компиляции:

1. 1-й выходной ФД h отправить на узел N.
2. 1-й входной ФД f запросить с узла N.

Назовём эти правила правилами роутинга ФД относительно источника.

Однако, если у a[b] есть потребители, неразрешенные такими правилами на этапе компиляции, данные для этих потребителей не будут найдены. Например:

1. b = g() // b = 0
2. d[b] = h()
3. f(d[b])
4. s(d[0])

Существует и альтернативный подход: задавать отображение ФД как некоторую таблицу роутинга, доступную на всех узлах, состоящую из правил, определяющих отображение множеств ФД на множества узлов. Однако, не существует способа определить точно множества ФД на этапе компиляции, без использования информации из runtime системы, если неизвестны правила, по которым они строятся. Поэтому этот подход заведомо не универсален.

Возможно применить комбинацию этих подходов: применять правила роутинга относительно источника параллельно с таблицей роутинга. А именно: отправить на все узлы, указанные pathfinder и правилами роутинга относительно источника (при их наличии). Однако, в этом случае передач данных по сети может стать значительно больше.

Существует и более радикальный подход: передача информации о нахождении ФД в runtime.

Все 3 описанных подхода возможно применять вместе. Более того, они дополняют друг друга.

Стоит пояснить, что проблем с отображением операторов АФВ нет: согласно реализации LuNA, каждому оператору АФВ возможно сопоставить уникальный basename.

2.1.1.6 Конструирование ОФУ

Введем следующие рекомендации LuNA программ:

1. Отобразить заданные АФВ на заданный узел.
2. Отправить заданный по порядковому номеру выходной аргумент АФВ на заданные узлы.
3. Принять заданный по порядковому номеру аргумент АФВ с заданного узла.
4. Отобразить заданные ФД на заданные узлы.

Конструирование введенных рекомендаций LuNA на основе ГЗ включает в себя 2 шага, проиллюстрированные на рисунке 3:

1. Отображение цепочек АФВ в ГЗ на одинаковые узлы.
2. Отображение всех ФД в цепочке на узлы, соответствующие источнику и потребителю.

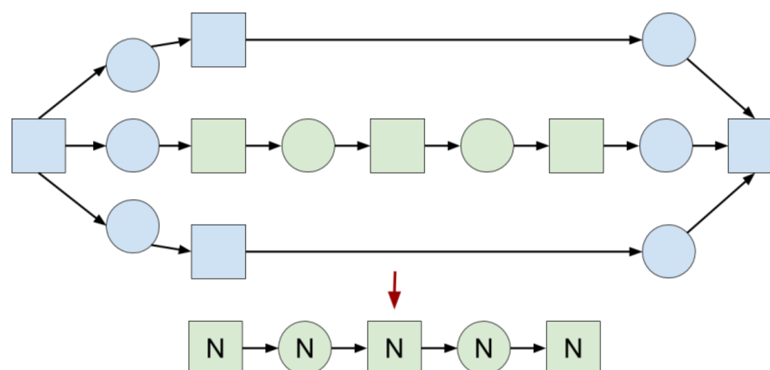

Рисунок 3 – Квадрат - оператор АФВ, круг - оператор ФД, число внутри фигуры - узел, на который отображен а-фрагмент. Зеленым цветом выделена цепочка

2.1.2 Анализ предложенного решения

В настоящем подразделе будет приведен анализ предложенного решения на предмет корректности, области его применимости и практической значимости.

2.1.2.1 Корректность

Введенные рекомендации никогда не нарушают корректность LuNA программы. Потому что для корректности ОФУ достаточно отображения всех фрагментов - не важно, на какие узлы.

При конструировании ОФУ, в котором некоторый АФВ отображен на несколько узлов, возможно предложить несколько вариантов, каждый из которых восстановит корректность ОФУ:

- Отобразить противоречивый АФВ стандартным образом.

- Отобразить АФВ только на один узел.

Такое ОФУ в дальнейшем будет называться противоречивым.

2.1.2.2 Базовый анализ

Так как предложенный алгоритм отображает цепочки связанных по данным операторов АФВ вместе, на один узел, этот алгоритм строит удачное ОФУ в случаях, когда в программе достаточно много цепочек связанных операторов АФВ. Достаточно много - не меньше, чем вычислительных ядер в системе.

Важно, что максимальное количество используемых вычислительных узлов при вычислениях на этих цепочках ограничено количеством таким цепочек. Количество цепочек ограничено сверху количеством строк в программе. Это характеризует узость этого метода: он не учитывает параллелизм между итерациями параллельных циклов, если их тело не развернуто.

2.1.2.3 Сравнение применимости с linear pathfinder

Стоит отметить, что в компиляторе LuNA есть оптимизационный модуль для loop unrolling. Поэтому есть прикладная возможность использовать предложенный алгоритм для оптимизации более широкого класса программ LuNA: например, в случаях, когда в программе есть параллельные циклы с известным на этапе компиляции количеством итераций, с цепочками на каждой итерации. Примеры: умножение матрицы на вектор, последовательные циклы, последовательность транспонирований матриц и другие.

Из этого списка linear pathfinder позволяет эффективно построить ОФУ для задачи умножения матрицы на вектор. Преимуществом linear pathfinder, в сравнении с предложенным методом, является то, что linear pathfinder не требует наличия цепочек в программе.

Поэтому области применимости предложенного метода и linear pathfinder пересекаются, но не совпадают.

В сочетании с loop tiling, и приведением к одному имени индексов различных циклов в одном операторе фигурных скобок, этот метод позволит улучшать ОФУ в циклах и между циклами с неизвестным на этапе компиляции числом итераций, с цепочками между итерациями. Создание модуля loop tiling для LuNA компилятора возможно. Сейчас такого модуля нет.

2.2 Конструирование ОФУ и приоритетов АФВ генетическим алгоритмом

Настоящий раздел посвящен вкладу в решение задач конструирования ОФУ и приоритетов АФВ для подпрограмм без циклов.

В первом подразделе 2.2.1 будет обоснован выбор генетического алгоритма как подхода для решения этих задач.

Следующий подраздел 2.2.2 посвящен описанию вводимой функции приспособленности генетического алгоритма.

Подраздел 2.2.3 - описанию генетического алгоритма.

Завершает раздел подраздел 2.2.4 анализа предложенного решения.

2.2.1 Выбор подхода

Для дальнейшего вклада в решение важен более глубокий анализ ГЗ, позволяющий отображать не только цепочки, но и другие АФВ. Степень практической значимости этой задачи возможно увидеть на примере. Разбиение на фрагменты задачи свёртки последовательностей [5] может совсем не содержать цепочек, как проиллюстрировано на рисунке 4:

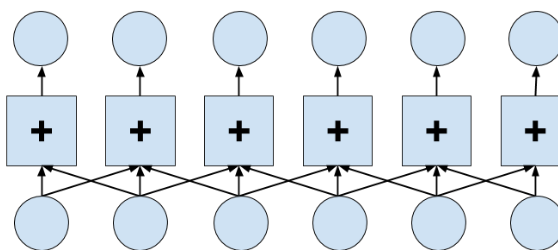

Рисунок 4 – Квадрат - АФВ, выполняющий свёртку последовательностей [5] элементов, круг - ФД

Кроме того, актуальна задача оценки приоритетов исполнения АФВ. Высокая степень практической значимости узкого места порядка исполнения АФВ обоснована в разделах 1.2 и 1.5.

Задачи формулируются так: нужно раскрасить ГЗ из операторов АФВ в заданное число пар цветов (узел и приоритет исполнения) так, чтобы минимизировать функцию оценки стоимости исполнения от раскраски этого ГЗ. Под стоимостью исполнения имеется в виду некоторая величина, явно коррелирующая или совпадающая со временем исполнения. Функция оценки стоимости исполнения может быть определена по-разному, и, от ее определения зависит результат.

В настоящей главе под “задачей” будет пониматься сформулированная задача построения раскраски ГЗ (РГЗ).

Важно отметить, что речь идет именно о раскраске ГЗ из операторов АФВ, но не операторов ФД. Потому что из раскраски операторов АФВ может следовать ОФУ операторов ФД: отображение каждого оператора ФД на объединение узлов, на которые отображены его потребители.

Предлагается 2 направления, предположительно, позволяющих получить приближенное решение задачи:

1. Применение генетического алгоритма или другого смежного метода для конструирования РГЗ с функцией приспособленности на основе оценки стоимости исполнения раскрашенного графа зависимостей.
2. Применение нейронной сети подходящей архитектуры, обученной на подходящей выборке РГЗ, для конструирования РГЗ или начальной оценки РГЗ.

Решение с помощью нейронной сети видится не приоритетным по следующим причинам:

- Получение подходящей выборки связано с неоправданными для такой задачи трудозатратами, даже при автоматическом или полуавтоматическом построении выборки. Для автоматического построения выборки нужно решить исходную задачу.
- Существует ряд сложностей, связанных с выбором подходящей архитектуры нейронной сети для этой задачи: количество целевых узлов, объем входных данных, целевые вычислители и сетевая инфраструктура переменны. Существуют готовые архитектуры нейронных сетей, позволяющие преодолеть эти сложности, однако решение остается по-прежнему неоправданно сложным.
- Так как решение задачи будет применяться на этапе компиляции, не критична высокая производительность, поэтому способность нейронной сети выдавать ответ за фиксированный объем вычислений не является решающим преимуществом.

Генетический алгоритм с функцией приспособленности на основе оценки стоимости исполнения РГЗ выбран как направление дальнейшего исследования.

2.2.2 Функция приспособленности

Для конструирования РГЗ с помощью генетического алгоритма требуется функция приспособленности, основанная на оценке стоимости исполнения РГЗ. Описанию этой функции посвящен настоящий подраздел.

Для целей оценки стоимости раскраски, ГЗ должен быть дополнен следующей информацией:

- У всех вершин должны быть положительные стоимости, обозначающие оценку стоимости исполнения.

- У всех вершин должно быть указание, вычислители какого типа (и в каком объеме) они задействуют: CPU, GPU, FPGA или другие.
- У всех ребер должны быть неотрицательные стоимости, обозначающие оценку стоимости передачи данных.

Оценка стоимости РГЗ - вариация классической задачи моделирования производственной линии [8]. Поэтому стоит привести только краткое описание особенностей решения этой задачи в рамках LuNA:

- Вход алгоритма: РГЗ, конфигурация вычислителей и сетевых интерфейсов на узлах, матрица стоимости сетевых путей между узлами.
- Работник - абстрактное представление ядра CPU, GPU, сетевого интерфейса или других устройств, с помощью которого возможно моделировать исполнение задачи, имеющей заданную продолжительность (стоимость).
- Каждый узел имеет изолированный пул работников. Передачи данных и исполнение АФВ выполняются разными работниками. Работников исполнения АФВ возможно интерпретировать как ядра узла и сопроцессоры, работников передач данных - как сетевые интерфейсы.
- Если данные передаются внутри узла, стоимость ребра должна интерпретироваться как 0.
- При наличии нескольких готовых к исполнению задач с одинаковым приоритетом, выбирать случайную задачу, оценивать всю РГЗ несколько раз с разными случайными числами.
- Выход алгоритма: 1 деленная на итоговую оценку стоимости исполнения.

Важно отметить, что оценивание с учетом топологии сети и конфигурации вычислителей - это очень важная возможность в рамках многих задач.

Стоит отметить, что в качестве оценки стоимости исполнения РГЗ могут быть использованы результаты профилировки (время исполнения) конечного приложения. Также результаты профилировки могут (и должны) быть использованы для оценки стоимости вершин и ребер.

Также стоит отметить, что данные о стоимостях также возможно получить из рекомендаций пользователя. При отсутствии данных возможно выбрать константные значения стоимостей.

2.2.3 Генетический алгоритм

Настоящий подраздел посвящен изложению предлагаемого решения задачи конструирования РГЗ с помощью генетического алгоритма [1, 6, 7] на основе рассмотренной функции приспособленности.

Определения перед алгоритмом:

- Приоритеты исполнения АФВ ограничены множеством [0, MAX_PRIORITY].

Алгоритм состоит из нескольких этапов:

1. Построение начальной популяции: начальную популяцию стоит построить случайно, выбрав достаточно большую мощность выборки. Распределение узлов предлагается выбрать равновероятным, а распределение приоритетов - нормальным, с малой дисперсией и математическим ожиданием равным MAX_PRIORITY деленный на 2.
2. Отбор. Так как в этой задаче может быть очень много локальных минимумов, стоит выбрать алгоритм отбора, препятствующий преждевременной сходимости. Например, “сигма-отсечение (sigma truncation)” [6].
3. Выбор n родителей. Есть предположение, что родителей стоит выбирать случайно, делая проблему сваливания в один локальный минимум менее актуальной, жертвуя скоростью сходимости. Существуют и другие подходы [6].
4. Скрещивание. Выбор случайного множества из каждого родителя, заполнение остальных случайными цветами, идентично этапу построения. Долю приобретаемой от родителей части стоит подобрать на практике, делая перевес в сторону малой доли. Следующий шаг - конец алгоритма или переход к отбору, в зависимости от выбранного критерия останова.

Оценка возможного вывода этого алгоритма на рисунке 5 (три применения свёртки последовательностей [5] или другая подобная по структуре зависимостей задача):

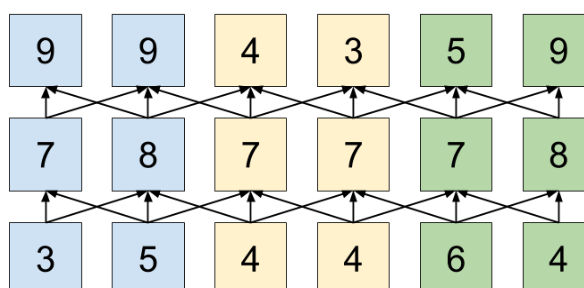

Рисунок 5 – Квадрат - оператор АФВ, стрелка - передача ФД, цвет - вычислительный узел. Число - приоритет исполнения. Все стоимости равны

2.2.4 Анализ предложенного решения

Настоящий подраздел посвящен анализу предложенного решения на основе генетического алгоритма.

2.2.4.1 Сходимость генетического алгоритма

Сходимость алгоритма формально доказана или опровергнута не будет ввиду нетривиальности этой задачи, но предлагается к проверке с помощью тестирования.

Если предположить, что алгоритм не сходится, не находит “хорошее” решение за “приемлемое” время, необходимо выбрать другой способ эвристической оптимизации перебора раскрасок. Определяющий компонент в этом алгоритме - функция приспособленности, а не метод ее приближенной максимизации.

2.2.4.2 Базовый анализ

Если предположить, что алгоритм сходится и находит решение с введенной функцией приспособленности, близкой к максимальной на данных входных значениях, на значимо большом множестве входных значений, возможно сделать следующие выводы:

Полученный подход позволяет строить ОФУ ГЗ, а также приоритеты исполнения. Качество конструирования ОФУ зависит, преимущественно, от расстановки стоимостей и от количества найденных зависимостей (отсутствия индексации по ФД), и, при достаточном объеме данных, будет близким к эталонному.

Иными словами, при возможности развернуть всю или существенную часть программы в конечный ГЗ без циклов и при явной индексации в операторах ФД, при эффективной расстановке стоимостей или нечувствительности задачи к расстановке стоимостей, этот алгоритм строит эффективное ОФУ.

Хотя подход и учитывает основные особенности графа зависимостей в расстановке приоритетов, приоритеты подвержены “шумам”, так как во многих случаях для оценщика стоимости исполнения минимально эффективная и избыточная расстановки приоритетов имеют равные оценки стоимости исполнения. Под минимально эффективной расстановкой имеется ввиду использование минимума возможных приоритетов при высоком значении функции приспособленности.

Есть гипотеза, что штраф в функции приспособленности за количество используемых приоритетов улучшит качество расстановки приоритетов.

Также есть гипотеза, что принесет улучшение разбиение алгоритма на 2 этапа:

1. Расстановка приоритетов операторов АФВ с одним ядром на каждом узле, и использованием функции приспособленности на основе усредненной степени параллелизма.
2. Конструирования ОФУ с расставленными приоритетами.

Расстановку приоритетов планируется улучшить в будущих работах.

Также есть следующая негативная особенность: существуют разные раскраски, функции приспособленности от которых совпадают. Например: при переименовании цветов, с учетом однородности сети и вычислителей, значение функции приспособленности не изменится. Есть следующая гипотеза: выбор родителей с похожим ОФУ чаще может улучшить решение задачи.

2.2.4.3 Применимость

Умножение матриц [2], свёртка последовательностей [5] (одиночная или n-кратная), n-мерная задача Пуассона, LU-разложение [4] и многие другие. Потому что в этих задачах всю или большую часть программы возможно развернуть в конечную последовательность операторов, для которой может будет построен ГЗ. По этой причине решаются и многие комбинации этих задач.

Значимые узкие места:

- Необходимость разворачивания циклов или использования loop tiling.
- Предложенный критерий оценки качества расстановки приоритетов недостаточно адекватен, как показано в предыдущем пункте 2.2.4.2.
- Не всегда возможно найти все зависимости между операторами АФВ на этапе компиляции при такой формализации ГЗ. Например, иногда операторы АФВ (именно операторы, а не АФВ) зависят от себя. В некоторых случаях зависимость между операторами АФВ появляется только при определенных условиях, известных только в runtime.
- Не учитывается возможное изменение стоимостей между запусками программы.
- Узкое место, к появлению которого есть тенденция: слишком большое время компиляции на больших ГЗ.

2.3 Другие предлагаемые решения

2.3.1 Динамический анализ в runtime

Описанные средства анализа использовали только статический анализ. Однако их возможно применять и в runtime, как перед запуском приложения, так и в процессе его работы. Этот подход имеет аналогии с JIT [10].

У этого подхода есть преимущество, связанное с тем, что во время запуска полностью известно оборудование, на котором запускается программа. Недостаток связан с тем, что время анализа может занимать большое время, измеряемое в десятках секунд, минутах, десятках минут и часах. Однако, если удастся уменьшить время анализа, этот подход станет основным.

2.3.2 Динамическая балансировка нагрузки на узлы

Предлагается 2 подхода к динамической балансировке нагрузки на узлы, в соответствии с наработками в конструировании ОФУ:

- разбиение на большее число узлов, чем существует физически. Эти узлы назовём логическими. Динамическая балансировка нагрузки на узлы заключается в динамической балансировке логических узлов между физическими. Этот подход имеет недостатки: перемещение логического узла требует и передачи всех ФД на нем, это требует применения ленивой (по запросу) или частично ленивой передачи ФД.
- отображение на особые логические узлы, называемые chair. Chair отображается на физический узел в runtime, исходя из того, какой узел первым начал исполнять АФВ из этого chair. Таким образом, chair может иметь 2 состояния: свободен, занят узлом. Досконально не изучены подходы к реализации этого механизма в рамках существующих системных алгоритмов LuNA. Возможный подход: барьер перед обращением на исполнение (или запись) к chair с состоянием “свободен”, ожидание при обращении на чтение (или запись).

Стоит отметить, что актуальна проблема, связанная с тем, что в runtime изменятся веса стоимости передачи данных, что не будет учтено при статическом анализе.

2.3.3 Освобождение памяти

Если известно, что у некоторого АФВ возможных зависимостей (определение из подраздела 2.1.1) нет, выходные данные возможно сразу удалить из памяти после отправки потребителям, потому что в таком случае известно, что у этих данных больше нет потребителей.

Это позволяет строить рекомендации очищения памяти, уменьшающие расход памяти LuNA программ.

2.3.3 Объединение АФВ

Полученные решения кластеризуют (разбивают на узлы) а-фрагменты. Кластеры операторов АФВ возможно объединять в один оператор АФВ, что улучшит производительность в случае, когда АФВ исполняются очень-очень быстро. Это позволяет в ряде случае избавить пользователя от необходимости программировать в АФВ обработку блоков данных, перекладывая создание блочности вычислений на LuNA.

Однако, есть сложности: сейчас в LuNA каждый ФД содержит C++ указатель, и, не хватает способов описания структуры памяти ФД. У этой проблемы возможны решения, предполагающие введение более подробной типизации данных в LuNA.

Другая сложность - векторизация вычислений, ведь часто мало лишь объединения вычислений в блок - этот блок зачастую нужно векторизовать. Существуют обширные наработки в области автоматической векторизации циклов в рамках компиляторов icc и gcc для x86-64. Они применимы, потому что в некоторых случаях объединенные операторы АФВ возможно представить как цикл над объединяемыми операторами АФВ.

3 Реализация routing_table_pathfinder и chains_routing_table

В настоящей главу будет описана реализация части предложенных алгоритмов конструирования частных управляющих схем конструирования ОФУ.

Ограничим реализацию следующим функционалом:

1. Оптимизационный модуль компилятора chains_routing_table, строящий следующие рекомендации ОФУ для цепочек зависимых АФВ:
   - Отобразить заданные АФВ на заданный узел
   - Отобразить заданные ФД на заданные узлы
2. Оптимизационный модуль компилятора gen_routing_table, строящий рекомендации ОФУ на основе прототипа генетического алгоритма.
3. LuNA pathfinder, называемый routing_table_pathfinder, интерпретирующий рекомендации ОФУ.

3.1 chains_routing_table

В настоящем разделе будет описана реализация оптимизационного модуля, конструирующего ОФУ для зависимых цепочек.

Оптимизационный модуль написан на языке python 2.7, как и остальные оптимизационные модули LuNA компилятора.

На вход оптимизационному модулю поступает количество вычислительных узлов и дерево разбора программы. На выход модуль выдает дерево разбора программы с рекомендациями.

Рекомендации представлены в форме таблицы роутинга (JSON документа), содержащего правила следующего вида: fragment names -> вычислительный узел.

fragment names может задавать множество а-фрагментов такого вида (пример): a[*][*][1]. “*” имеет тот же смысл, что и в регулярном выражении.

Конструирование рекомендаций состоит из 2-х этапов:

1. Выделение цепочек в ГЗ.
2. Отображение цепочек на узлы по правилу round-robin.

Отображение каждого оператора АФВ происходит с помощью указания basename оператора АФВ. Противоречий в таблице не будет, потому что все basename операторов АФВ уникальны (после постобработки).

Отображение операторов ФД более сложное: везде, где на этапе компиляции индекс известен как константа, эта константа заносится в таблицу роутинга. Иначе, в таблицу заносится “*”. Пример:

a[i][0] в таблицу роутинга будет записано как a[*][0]

В приложении Б представлена диаграмма классов (в нотации UML) оптимизационного модуля.

3.2 gen_routing_table

В настоящем разделе будет описана реализация оптимизационного модуля компилятора gen_routing_table, строящего рекомендации ОФУ на основе прототипа генетического алгоритма.

chains_routing_table использует chains_routing_table для построения ГЗ, для объединения цепочек зависимых операторов АФВ в логические блоки, отображаемые на узлы целиком. Операторы ФД отображаются согласно отображению операторов АФВ, как предложено в подразделе 2.2.1.

Реализованная функция приспособленности, по сравнению с описанной в подразделе 2.2.2, упрощена и является единицей деленной на оценку количества передач ФД в ГЗ при заданном ОФУ.

Реализован вырожденный генетический алгоритм, сводящийся к случайному блужданию: популяция всегда состоит из одной особи, мутация отбрасывается при отрицательном сдвиге функции приспособленности. Функция приспособленности задана адекватно, потому что количество отображенных операторов АФВ на разные узлы всегда поддерживается отличающимся не более чем не единицу, а решаемые задачи предполагаются однородными.

Реализация, по сравнению с предложенным в разделе 2.2 решением, существенно упрощена. Это сделано намеренно: в первую очередь нужно убедиться в применимости генетического или смежных алгоритмов для вклада в решение поставленной проблемы.

3.3 routing_table_pathfinder

pathfinder, интерпретирующий таблицу роутинга, каждое полученное имя фрагмента ищет в таблице роутинга. Если имя было найдено, pathfinder возвращает узел, указанный в найденной записи. Иначе, возвращает значение от default pathfinder. Если записей было найдено несколько, указывающих на разные узлы, реализованный pathfinder выбирает первую запись, потому что в текущей реализации LuNA pathfinder может возвращать только одно значение. Первая запись совпадает на всех узлах, потому что на всех узлах таблица роутинга совпадает.

4 Тестирование

В настоящей главе будет приведено тестирование реализации gen_routing_table и routing_table_pathfinder с целью подтверждения применимости генетического или смежных алгоритмов для вклада в решение поставленной проблемы.

Тестирование проводилось на задаче LU-разложения [4] с размером матрицы 16x16 ФД и 16x16 элементов внутри каждого ФД. Листинг программы приведен в приложении В.

Все циклы for в программе были развернуты с помощью модуля unrolling_for LuNA компилятора. Количество поколений генетического алгоритма: 100000. Время работы реализованного оптимизационного модуля составило порядка 20 минут на процессоре Intel® Core™ i5-6500.

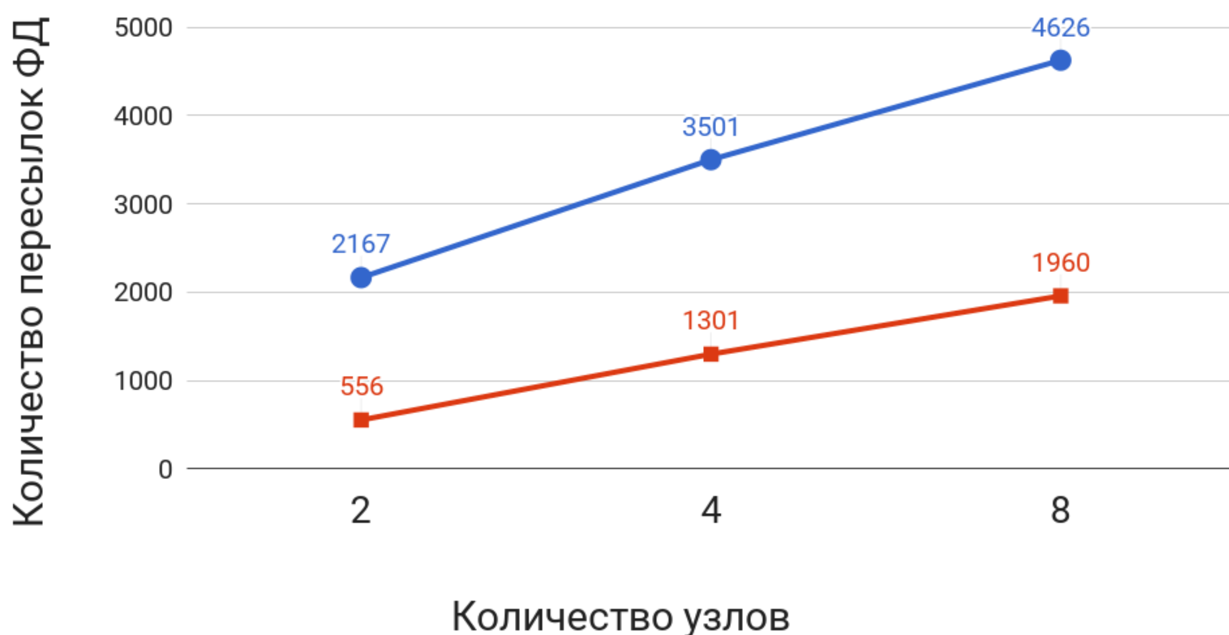

Рисунок 6 – Тестирование, по горизонтали - количество используемых узлов, по вертикали – количество пересылок ФД между узлами

На рисунке 6 приведен график, на котором красной линией обозначены результаты gen_routing_table и routing_table_pathfinder, синей - hash_pathfinder, описанный в разделе 1.1. В обоих случаях программа задействовала все доступные узлы, отобразив имеющиеся АФВ и ФД на них приблизительно равномерно, согласно реализации gen_routing_table и hash_pathfinder. С учетом этого и с учетом однородности задачи, количество пересылок ФД - содержательная характеристика в тестировании, потому что объем накладных сетевых расходов зависит от него. Как можно видеть из результатов тестирования, реализованные routing_gen_pathfinder и chains_routing_table показывают применимость генетического или

смежных алгоритмов для вклада в решение поставленной проблемы, и, предлагаются для дальнейшего улучшения.

ЗАКЛЮЧЕНИЕ

В работе предложено два алгоритма распределения задач и данных на узлы для средства программирования высокопроизводительных распределенных программ LuNA:

- Алгоритм конструирования распределения задач и данных на узлы для подпрограмм без циклов, основанный на статическом анализе LuNA программ, построении дерева информационных зависимостей между операторами программы и компоновке цепочек в графе зависимостей в блоки, отображении блоков на узлы.
- Алгоритм конструирования распределения задач и данных на узлы и приоритетов задач для подпрограмм без циклов. На основе информации о ресурсоемкости операций, количестве и типах вычислителей, топологии сети решается задача оценки качества распределения и качества расстановки приоритетов АФВ. Алгоритм основан на применении генетического алгоритма с функцией приспособленности на основе оценки качества распределения и расстановки приоритетов задач.

Также предложены другие смежные алгоритмы, вытекающие из двух основных. Применение предложенных алгоритмов обеспечивает существенное сокращение накладных расходов на передачу данных при использовании LuNA, что показано с помощью тестирования.

Реализованные оптимизационные модули интегрированы в системы существующих оптимизационных модулей, используя их возможности и предоставляя библиотечные средства.

В настоящей работе выносится на защиту:

- Аналитический обзор узких мест в алгоритмах LuNA.
- Разработка алгоритмов конструирования частных управляющих схем в рамках LuNA, вносящих вклад в решение поставленной проблемы.
- Реализация алгоритма конструирования частных управляющих схем отображения цепочек зависимых задач и данных на узлы в виде оптимизационного модуля LuNA chains_routing_table, вносящего вклад в решение поставленной проблемы
- Реализация прототипа алгоритма конструирования частных управляющих схем конструирования распределения задач и данных на узлы на основе генетического алгоритма в виде оптимизационного модуля LuNA gen_routing_table, вносящего вклад в решение поставленной проблемы.
- Тестирование реализованного оптимизационного модуля LuNA gen_routing_table.

Дальнейшее развитие планируется по нескольким направлениям:

- Реализация остальных алгоритмов конструирования разработанных частных управляющих схем.
- Разработка решения других смежных задач. Например, стоит предложить представление информационных зависимостей, позволяющее выполнять их анализ, параметризованный представлением runtime данных в программе. Также стоит предложить различные, в том числе обобщенные, методы анализа информационных зависимостей в этом представлении в runtime, позволяющие использовать более широкие методы динамической балансировки нагрузки. В этом направлении уже есть наработки.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Лысенко Егор Олегович	____________________________
ФИО студента	Подпись студента

« ____ »___________2017 г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Емельянов, В. В. Теория и практика эволюционного моделирования / В. В. Емельянов, В. В. Курейчик, В. М. Курейчик — М.: Физматлит, 2003. — 432 с.
2. Корн, Г. Алгебра матриц и матричное исчисление / Г. Корн, Т. Корн // Справочник по математике. — 4-е издание. — М.: Наука, 1978. — С. 392—394
3. Новиков, А. С. Построение графа информационных зависимостей по машинному коду / А. С. Новиков // Известия Тульского государственного университета. — 2016. — №2. — С. 180-187
4. Ортега, Дж. Введение в параллельные и векторные методы решения линейных систем / Дж. Ортега — М.: Мир, 1991. — 376 с.
5. Рабинер, Л. Теория и применение цифровой обработки сигналов / Л. Рабинер — М.: Мир, 1978. — С. 72-81. — 848 с
6. Рутковская, Д. Нейронные сети, генетические алгоритмы и нечеткие системы / Д. Рутковская, М. Пилиньский, Л. Рутковский — 2-е изд. — М.: Горячая линия-Телеком, 2013. — 452 с.
7. Скобцов, Ю. А. Основы эволюционных вычислений / Ю. А. Скобцов — Донецк: ДонНТУ, 2008. — 326 с.
8. Толуев, Ю. Имитационная модель производственной линии на базе сложной конвейерной системы / Ю. Толуев, Т. Змановская // Автоматизация в промышленности. — 2013. — №7. — С. 37-41
9. Федотов, И. Модели параллельного программирования / И. Федотов — М.: Солон-Пресс, 2012. — С. 384.
10. Aycock, J. A brief history of just-in-time // ACM Computing Surveys. — 2003. — Volume 35, Issue No 2. — P. 97-113.
11. Kireev, S.E., Malyshkin, V.E. Fragmentation of Numerical Algorithms for Parallel Subroutines Library // *J. Supercomputing*. — 2011. — Volume 57, Issue No 2. — P. 161-171.
12. Malyshkin, V.E., Perepelkin, V.A. Optimization Methods of Parallel Execution of Numerical Programs in the LuNA Fragmented Programming System // *J. Supercomputing*. — 2012. — Volume 61, Issue No 1. — P. 235-248.
13. Akka [Электронный ресурс] — Режим доступа: http://akka.io/ — Загл. с экрана.
14. Boost.Asio [Электронный ресурс] — Режим доступа: http://boost.org/doc/libs/1_64_0/doc/html/boost_asio.html — Загл. с экрана.

15. Charm++ Parallel Programming Library [Электронный ресурс] — Режим доступа: http://charm.cs.uiuc.edu/research/charm — Загл. с экрана.
16. DVM System [Электронный ресурс] — Режим доступа: http://www.keldysh.ru/dvm — Загл. с экрана.
17. Ice - Comprehensive RPC Framework [Электронный ресурс] — Режим доступа: https://zeroc.com/products/ice — Загл. с экрана.
18. MPI Documents [Электронный ресурс] — Электрон. дан. — Режим доступа: http://mpi-forum.org/docs — Загл. с экрана.
19. OpenMP [Электронный ресурс] — Режим доступа: http://openmp.org/ — Загл. с экрана.
20. ZeroMQ Distributed Messaging [Электронный ресурс] — Режим доступа: http://zeromq.org/ — Загл. с экрана.

ПРИЛОЖЕНИЕ А

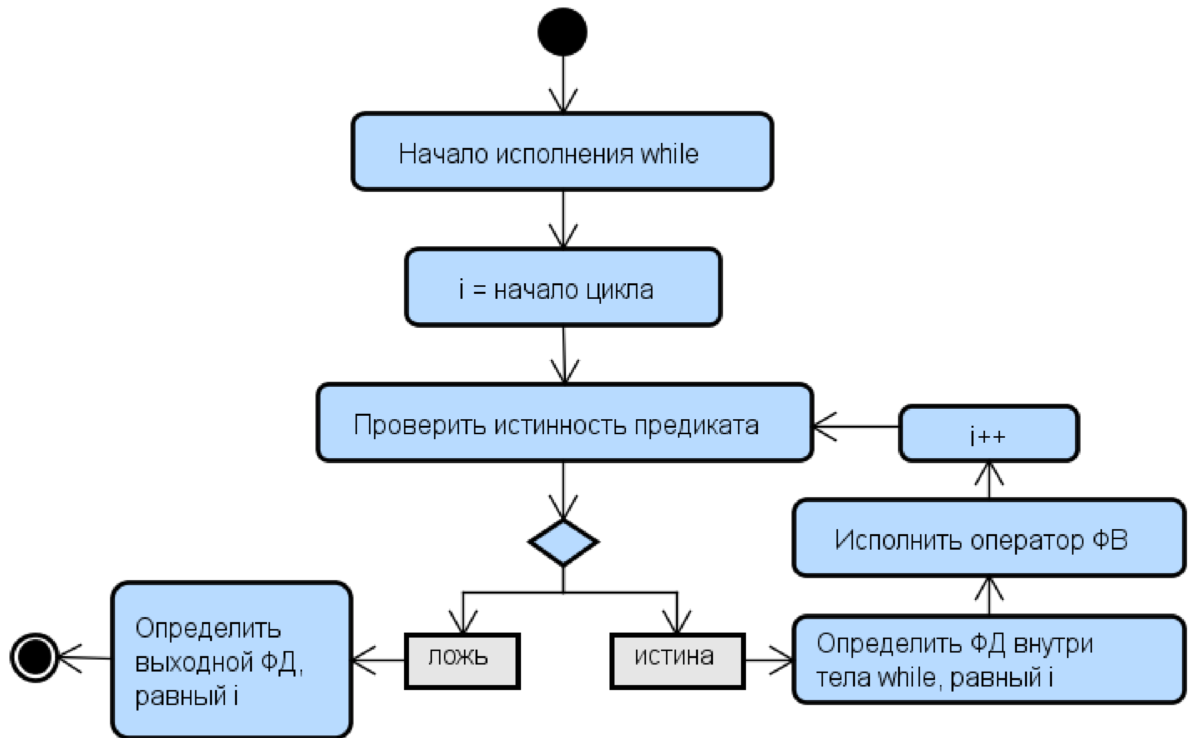

Рисунок А.1 – Блок-схема алгоритма исполнения ФВ while

ПРИЛОЖЕНИЕ Б

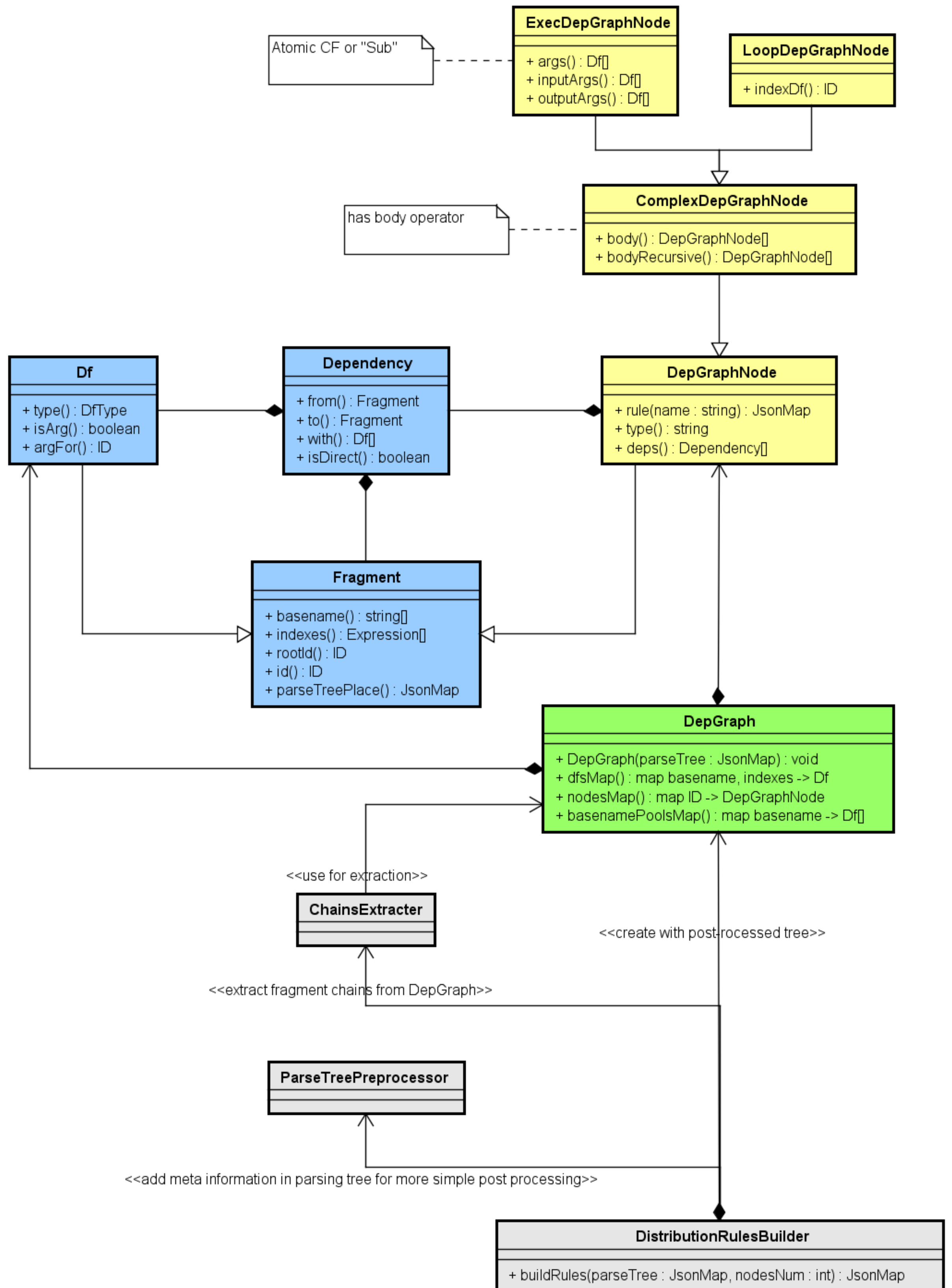

Рис Б.1 – Диаграмма классов chains_routing_table

ПРИЛОЖЕНИЕ В

```
##const N 16
##const M 16

import c_init_submatrix(int, int, int, name) as init_matrix;
import c_proc_du(value, value, name) as proc_du;
import c_proc_dl(value, value, name) as proc_dl;
import c_proc_dlu(value, name)  as proc_dlu;
import c_proc_dg(value, value, value, name) as proc_dg;
import c_init_zero_matrix(int, int, name) as init_zero_matrix;
import c_copy_matrix(value, name) as copy_matrix;
import c_make_diagonal_l(value, name) as make_diag_l;
import c_make_diagonal_u(value, name) as make_diag_u;

sub main()
{
   // N - size of input matrix in number of fragments in a row.
   // M - size of submatrix. submatrix is matrix (MxM) of doubles
   //A[0] - input matrix. It will evolve to A[i] and finally to L and U.
   df  L, U, A;

   for i = 0 .. N-1
   {
      for j = 0 .. N-1
      {
         init_matrix(i, j, 0, A[0][i][j]);
      }
   }

   for i = 0..N-1
   {
      proc_dlu(A[i][i][i], A[i+1][i][i]);
      for k=i+1..N-1
      {
         proc_du(A[i+1][i][i],A[i][i][k], A[i+1][i][k]);
```

```
    }
    for j=i+1 .. N-1
    {
        proc_dl(A[i+1][i][i],A[i][j][i], A[i+1][j][i]);
    }
    for j=i+1 .. N-1
    {
        for k=i+1 .. N-1
        {
            proc_dg(A[i+1][j][i], A[i+1][i][k], A[i][j][k], A[i+1][j][k]);
        }
    }
}

for k = 1 .. N
{
    //copy row to U
    make_diag_u(A[k][k-1][k-1], U[k-1][k-1]);
    for i = k .. N -1
    {
            copy_matrix(A[k][k-1][i], U[k-1][i]);
    }
    //init another blocks in a row to zeros
    for i = 0 .. k-2
    {
        init_zero_matrix(M, 0, U[k-1][i]);
    }
    //copy col to L
    for i = k .. N-1
    {
        copy_matrix(A[k][i][k-1], L[i][k-1]);
    }
    //rest of col will be zeroed, except diagonal block
    for i = 0 .. k-1
    {
```

```
            if i != k-1
            {
                init_zero_matrix(M, 0, L[i][k-1]);

            }
            if i == k-1
            {
                  make_diag_l(A[k][i][i], L[i][i]);
            }
        }
    }
}
```