Федеральное государственное бюджетное учреждение науки
Институт вычислительной математики и математической геофизики
Сибирского отделения Российской академии наук

На правах рукописи

Перепёлкин Владислав Александрович

СИСТЕМА LUNA АВТОМАТИЧЕСКОГО КОНСТРУИРОВАНИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ ЧИСЛЕННОГО МОДЕЛИРОВАНИЯ НА МУЛЬТИКОМПЬЮТЕРАХ

Специальность 2.3.5 — математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель
доктор технических наук, профессор
Малышкин Виктор Эммануилович

Новосибирск — 2022

Оглавление

Введение

Актуальность темы работы

Разработка и отладка параллельных программ численного моделирования на суперЭВМ сложна из-за необходимости решать проблемы, специфичные для параллельного программирования, такие как декомпозиция данных и вычислений, программирование коммуникаций, синхронизация работы процессов и потоков, распределение и динамическое перераспределение данных и вычислений по вычислительным узлам. В некоторых случаях необходимо обеспечение динамической балансировки нагрузки на вычислительные узлы, сохранение контрольных точек, обеспечение отказоустойчивости. Программа для суперЭВМ должна эффективно использовать аппаратные ресурсы, быть переносима, учитывать особенности вычислителя: производительность вычислительных узлов, топологию и производительность сети, доступный объём памяти, динамику загрузки узлов. Также программа должна быть масштабируемой на большое количество вычислительных узлов и использовать, по возможности, все доступные ресурсы вычислительной системы.

Для решения обозначенных проблем программисту требуется специальная квалификация — квалификация в системном параллельном программировании, что затрудняет использование суперЭВМ обычными пользователями — специалистами в своих предметных областях. Вследствие этого важную роль играют средства автоматизации конструирования параллельных программ, способные частично взять на себя решение этих проблем и сократить трудоёмкость, времяёмкость и сложность разработки, отладки и сопровождения параллельных программ, снизить требования к квалификации программиста и повысить качество конструируемых программ.

Таким образом, автоматизация конструирования параллельных программ численного моделирования для мультикомпьютеров является актуальной темой.

Объект и предмет исследования

Объектом исследования является процесс автоматического конструирования параллельной программы по её высокоуровневой спецификации. Предмет исследования — языки и алгоритмы автоматического конструирования параллельных программ и алгоритмы исполнения параллельных программ.

Степень разработанности темы

Проблема автоматизации конструирования высокопроизводительных параллельных программ активно прорабатывается в науке на протяжении десятилетий. Ввиду сложности этой проблемы в общей постановке развиваются различные частные подходы, методы и алгоритмы, ориентированные на свои классы приложений.

Важный вклад в проработку проблемы внесли Ю.И. Янов [1, 2], В.Е. Котов [3–5], В.А. Вальковский [6], В.Э. Малышкин [7], работа которых привела к созданию теории структурного синтеза параллельных программ на вычислительных моделях [8] и концепции активных знаний [9] (на этот базис опирается и настоящая работа). Язык Утопист был разработан Э.Х. Тыугу [10] при существенном вкладе А.Л. Фуксмана [11]. Работы И.Б. Задыхайло, А.Н. Андрианова привели к созданию системы Норма [12]. Важны работы В.А. Крюкова над DVM-системой [13, 14], специализирующейся на «распараллеливании» последовательного кода, работы С.М. Абрамова над Т-Системой [15, 16].

Среди зарубежных исследователей отметим работы [17, 18], проекты Charm++ [19] (L. Kale) , PaRSEC [20] и DPLASMA [21] (J. Dongarra), Legion [22] и Regent [23] (A. Aiken).

Разработки в области систем программирования и систем автоматизации конструирования параллельных программ, в отличие от низкоуровневых средств программирования (напр., MPI [24], OpenMP [25]) и от узкоспециализированных пакетов моделирования (напр., Fluent [26], NAMD [27], PICADOR [28] и пр. [29, 30]), сочетают в себе гибкость (возможность запрограммировать широкий спектр прикладных алгоритмов) и относительно высокий уровень программирования. Системы программирования, как правило, уступают в производительности как низкоуровневым средствам программирования, так и узкоспециализированным пакетам моделирования, но оказываются полезными на практике, так как, во-первых, их применение проще, чем использование низкоуровневых средств разработки, за счёт того, что они берут решение части проблем на себя, и, во-вторых, область применения шире, чем у специализированных пакетов, ввиду большей универсальности систем. От используемых в системе программирования модели вычислений и системных алгоритмов, а также качества их реализации, зависит область практической применимости системы. Проработка проблемы автоматизации конструирования параллельных программ численного моделирования на мультикомпьютерах выражается, главным образом, в совершенствовании моделей вычислений и системных алгоритмов. В результате область применения систем программирования расширяется, повышается удобство их использования и качество конструируемых программ.

Широкая распространённость низкоуровневых средств разработки параллельных программ и активное развитие широкого спектра средств автоматизации конструирования параллельных программ показывает, что несмотря на большое количество наработок и полезных инструментов эта проблема далеко не решена, и требуется её дальнейшая проработка.

В монографии В.А. Вальковского и В.Э. Малышкина [8] предлагается подход, суть которого состоит в автоматической сборке параллельных программ из последовательных модулей (подпрограмм) на основе формального описания этих модулей и спецификации конструируемой программы. Подход обладает рядом преимуществ перед другими подходами в области автоматизации конструирования численных параллельных программ, т.к., во-первых, опирается на сборку параллельной программы из уже наработанных в данной предметной области программных модулей, что обеспечивает более высокое качество конструируемой программы, позволяет иметь дело с реальными, а не небольшими тестовыми задачами. Во-вторых, подход подразумевает явно выраженный параллелизм и явно выраженную декомпозицию данных и вычислений, что делает его практичным, т.к. позволяет избежать постановки перед системой таких алгоритмически труднорешаемых проблем как распараллеливание последовательного кода или автоматическая декомпозиция данных. И, в-третьих, подход позволяет существенно автоматически управлять ходом вычислений с точки зрения нефункциональных свойств — распределять и перераспределять данные и вычисления по узлам мультикомпьютера, планировать вычисления и выполнять иные манипуляции статически и динамически для обеспечения тех или иных нефункциональных свойств программы.

Данный подход теоретически проработан [8], но практическое его использование для автоматического конструирования параллельных программ численного моделирования требует разработки языка описания вычислительных моделей и алгоритмов конструирования и исполнения параллельных программ. Настоящая работа вносит вклад в заполнение этого пробела.

Цель и задачи исследования

Целью диссертационного исследования является разработка экспериментальной системы автоматического конструирования параллельных программ численного моделирования для мультикомпьютеров на основе теории структурного синтеза параллельных программ на вычислительных моделях.

Для достижения поставленной цели в работе ставятся и решаются следующие задачи:

— разработка языка конструирования параллельных программ LuNA (Language for Numerical Algorithms),
— разработка алгоритмов трансляции и исполнения LuNA-программ,
— реализация разработанных системных алгоритмов в виде экспериментальной системы конструирования параллельных программ LuNA,
— экспериментальное исследование нефункциональных свойств системы LuNA и конструируемых программ.

Методология и методы исследования

Проведённое исследование базируется на результатах теории алгоритмов и математической логики. За основу при разработке модели фрагментированного алгоритма взято понятие вычислительной модели из теории синтеза параллельных программ на вычислительных моделях [8]. Результаты теории вычислимости [31, 32] использованы для доказательства свойства универсальности модели фрагментированного алгоритма. При разработке алгоритмов использовались результаты теории алгоритмов [33–35], в том числе методы разработки параллельных программ в общей и распределённой памяти [36–38] с учётом сложившихся практик разработки численных параллельных программ [39, 40]. При разработке архитектуры и программного кода системы LuNA применялись методы и сложившиеся практики разработки программного обеспечения, в частности, методика объектно-ориентированного программирования [41–44]. При разработке транслятора использовались результаты теории компиляции [45].

Научная новизна работы

— На основе существующего понятия вычислительной модели предложена модель фрагментированного алгоритма,
— Разработан язык LuNA описания фрагментированных алгоритмов,
— Разработаны алгоритмы, обеспечивающие трансляции и распределённого исполнения фрагментированных алгоритмов, описанных на языке LuNA:
    — базовый алгоритм распределённой интерпретации фрагментированных алгоритмов,

— алгоритм Rope of Beads динамического отображения фрагментированных алгоритмов на узлы мультикомпьютера,
— алгоритм распределённой сборки мусора,
— алгоритм распределённого обнаружения остановки системы.

Теоретическая значимость работы

Предложенная модель фрагментированного алгоритма позволяет ставить задачу конструирования распределённой программы, соответствующей заданной функциональной спецификации, предназначенной для исполнения на мультикомпьютере с заданной конфигурацией и обладающей требуемыми нефункциональными свойствами.

На основе разработанной системы LuNA возможно экспериментальное исследование различных системных алгоритмов конструирования и исполнения параллельных программ (алгоритмов управления распределением ресурсов, планирования вычислений, балансировки вычислительной нагрузки по узлам мультикомпьютера и т.п.).

Практическая значимость работы

Разработанная система LuNA упрощает разработку, отладку и модификацию параллельных программ численного моделирования на мультикомпьютерах за счёт автоматизации их конструирования. В частности, автоматизируется программирование (и исключается отладка) коммуникаций, синхронизации процессов и потоков, распределения и динамического перераспределения данных и вычислений по узлам мультикомпьютера, сборки мусора и пр.

Положения, выносимые на защиту

1. Модель фрагментированного алгоритма,
2. Язык LuNA описания фрагментированных алгоритмов,
3. Системные алгоритмы трансляции и распределённого исполнения фрагментированных алгоритмов, описанных на языке LuNA,
4. Проект системы LuNA автоматического конструирования параллельных программ для мультикомпьютеров.

Степень достоверности и апробация результатов

Достоверность полученных результатов обеспечивается применением использованных методов исследования, а также подтверждается теоретическим анализом и результатами экспериментального исследования. Все основные результаты работы докладывались на конференциях, научных семинарах и опубликованы в профильных рецензируемых печатных изданиях.

Личный вклад автора

Личный вклад автора состоит в обсуждении постановки задачи, разработке модели фрагментированного алгоритма на основе существующего понятия вычислительной модели, разработке языка LuNA описания фрагментированных алгоритмов, разработке системных алгоритмов трансляции и исполнения фрагментированных алгоритмов, представленных на языке LuNA, разработке всех основных компонентов системы LuNA. Все выносимые на защиту результаты получены автором лично.

В совместных работах с научным руководителем В.Э. Малышкиным ему принадлежит идея фрагментированного программирования как подхода к автоматизации конструирования параллельных программ, при котором параллельная программа состоит из фрагментов во время исполнения, и эти фрагменты могут перераспределяться между вычислительными узлами и исполняться в разном порядке. Автором выполнялась разработка конкретных алгоритмов и программных модулей, реализующих этот подход. В совместных работах с другими соавторами автором диссертационной работы была выполнена реализация базовой версии системы LuNA и её алгоритмов, а соавторами была выполнена разработка отдельных дополнительных модулей системы или разработка прикладных LuNA-программ. Детали по конкретным публикациям изложены в главе 4.

Структура диссертации

В главе 1 анализируются существующие средства автоматического конструирования параллельных программ. В главе 2 представлены теоретические результаты работы — модель фрагментированного алгоритма, и системные алгоритмы, обеспечивающие распределённую реализацию фрагментированных алгоритмов на мультикомпьютере. В главе 3 представлена программная система LuNA, реализующая предложенные в главе 2 результаты и на их основе осуществляющая автоматическое конструирование и исполнение параллельных программ. Рассматриваются и разрешаются технические вопросы реализации системы LuNA. В главе 4 представлены экспериментальные исследования разработанной системы на ряде синтетических и прикладных задач, позволяющие судить об эффективности предложенного решения, в том числе количественно. Завершает работу заключение и 5 приложений.

1. Обзор предметной области

В основе систем автоматизации конструирования параллельных программ лежит некоторая модель вычислений, в терминах которой пользователь описывает прикладной[1] (численный) алгоритм. От этой модели вычислений зависит, какие требуемые нефункциональные свойства конструируемой программы потенциально могут быть обеспечены автоматически, а от алгоритмов системы зависит насколько этот потенциал реализуется. Модель вычислений и системные алгоритмы, таким образом, определяют область применения системы. Проанализируем существующие языки, системы и средства параллельного программирования по основным их классам на примере характерных представителей с точки зрения автоматизации конструирования эффективных параллельных программ численного моделирования на мультикомпьютерах. Тут и далее под эффективностью будем неформально понимать «экономность» системы по тем или иным ресурсам — времени выполнения, расходу памяти, нагрузке на сеть и т.п. Требование эффективности хотя бы по некоторым параметрам является необходимым при использовании высокопроизводительных вычислительных систем.

1.1 Специализированные модели вычислений

Подходы к автоматизации конструирования параллельных программ на основе частных моделей вычислений, например, MapReduce [46], Hadoop [47], Anthill [48], обеспечивают хорошие результаты как по производительности, так и по другим свойствам (отказоустойчивость, динамическая балансировка загрузки и т.п.), но имеют ограниченную область применения ввиду неуниверсальности модели вычислений. Эти ограничения не могут быть преодолены в полной мере путём развития системных алгоритмов без изменения модели вычислений, лежащей в основе системы.

1.2 Распределённая общая память

В рамках систем с общей распределённой памятью (PGAS [49]), таких как Titanium [50], Unified Parallel C [51], Co-Array Fortran [52], X10 [53], HPX [54] и пр., авторы исследуют

[1] В работе алгоритмы разделяются на прикладные и системные. Прикладные алгоритмы — это те, которые нужно запрограммировать, а системные — это те, которые составляют систему программирования, они отвечают за конструирование и исполнение параллельной программы, реализующей прикладной алгоритм, поданный системе на вход.

возможности создания иллюзии общей памяти на системах с распределённой памятью. При этом неизбежно возникает неоднородность памяти с точки зрения времени доступа к ней и пропускной способности. Игнорирование этого факта со стороны пользователей системы приводит к неэффективным программам, что практически не может быть преодолено со стороны системы ввиду сложности этой задачи. Как следствие, пользователям приходится учитывать фактическую неоднородность памяти в программах, что снижает их переносимость. Разработчики систем применяют частные и эвристические подходы, обеспечивающие удовлетворительное качество конструируемых программ, что определят область практического применения систем.

1.3 Исполнительные системы и параллелизм задач

В подходах, основанных на параллелизме задач (task-based), декомпозиция данных и вычислений осуществляется пользователем, а планирование вычислений и распределение данных и вычислений по узлам осуществляется автоматически. В Иллинойском университете ещё с 1980-х годов развивается исполнительная система Charm++ [19] для исполнения параллельных программ, представленных в виде множества мигрирующих взаимодействующих объектов (L. Kale et al.), что позволяет автоматически обеспечивать динамическую балансировку нагрузки на узлы и другие свойства программы. Подобный подход исследуется в работах С.М. Абрамова (Т-Система [15, 16]).

1.4 Управляемое исполнение

Общей слабостью большинства подходов является использование такого представления прикладного алгоритма, эффективное исполнение которого обеспечивается частными и эвристическими подходами лишь на ограниченном круге задач, который и составляет область практического применения системы. Автору известно три исключения, описанных ниже. В этих работах помимо собственно представления прикладного алгоритма имеются отдельные средства, позволяющие описать, как прикладной алгоритм следует исполнять. Это позволяет явно разделить функциональную отладку программы и обеспечение её нефункциональных свойств. В частности, над повышением эффективности программы может работать отдельный специалист в области системного параллельного программировании без риска внести функциональную ошибку в программу.

Так, в университете Теннесси для поддержки библиотеки матрично-векторных операций DPLASMA [21] развивается подход (J. Dongarra et al.) к конструированию и исполнению параллельных программ на основе машинно-независимого представления прикладного алгоритма в виде бесконтурного ориентированного графа (Directed Acyclic Graph, DAG), что позволяет обеспечивать высокопроизводительную реализацию представленных прикладных алгоритмов благодаря планированию вычислений и динамической поддержке исполнительной системы (PaRSEC [20]).

В Стэндфордском университете (A. Aiken) разрабатывается подход, при котором отдельно описывается прикладной алгоритм и способ его отображения на вычислительные ресурсы. Коллективом разработана система Legion [22], реализующая этот подход. Недостатком системы является отсутствие автоматизации в конструировании способа отображения прикладного алгоритма на вычислительные ресурсы.

Во фреймворке AllScale [55] явно разделяются уровни описания вычислительной части в терминах предметной области, уровень описания параллелизма и локальности данных, а также системный уровень отображения данных и вычислений на аппаратные ресурсы.

1.5 Системы автоматизированного распараллеливания последовательного кода

В работах HPF (K. Kennedy) [56], DVM/САПФОР (В.А. Крюков) [13, 14], XcallableMP [57] исследуются подходы к автоматизации конструирования параллельных программ из последовательных путём статического анализа последовательного кода, возможно, аннотированного. Подход позволяет относительно малыми усилиями пользователя получать параллельный код из последовательного, но возникающие при этом проблемы выявления параллелизма в последовательном коде, декомпозиции данных и вычислений, планирования вычислений и распределения нагрузки существенно ограничивают область практического применения системы. Частично это преодолевается аннотированием кода.

1.6 Высокоуровневые языки и системы

Существуют языки и системы, которые ставят задачу автоматического конструирования параллельной программы по высокоуровневому описанию алгоритма в общем виде. Такие системы характеризуются ресурсно-независимым описанием алгоритма или схемы вычислений

в терминах отдельных переменных (целых и вещественных чисел) и операций (сложения, умножения и т.п.), математических уравнений, систем рекуррентных соотношений и т.п. К таким работам относятся языки Пифагор [58], Sisal [59], Cilk [60], Model [61].

Также сюда можно отнести «чистые» функциональные языки, такие как Haskell [62, 63]. Показательно, что такие языки и системы обычно не выходят за пределы вычислителей с общей памятью, т.к. конструирование эффективной распределённой программы по такому описанию алгоритма слишком сложно.

1.7 Системы фрагментированного программирования

Подход, исследуемый в работе, развивался на протяжении десятилетий. В частности, стоит отметить работу А.А Цыгулина [64] и его кандидатскую диссертацию [65], имеющую близкую постановку задачи. Её автор использовал подход конструирования параллельных программ на основе заранее подготовленных «скелетонов» [66] распределённых программ, содержащих готовую параметризованную схему параллельной программы. В такой скелетон пользователь подставляет блоки кода, отвечающие за содержательную часть вычислений, доопределяя скелетон до программы. Вопросам вывода алгоритмов на вычислительных моделях посвящена кандидатская диссертация А.В. Засыпкина [67].

Особенности исполнения крупноблочно-параллельных программ в общей памяти на основе исполнительной системы исследовались в кандидатской диссертации С.Б. Арыкова [68] и в работах [69, 70] по разработанной им системе Аспект, а также в работе [71] и пр.

Отдельные системные алгоритмы для последующего применения в системах фрагментированного программирования разрабатывались и без собственно исполнительных систем, например в работе [72] исследовались проблемы эффективной параллельной реализации метода частиц-в-ячейках с динамической балансировкой нагрузки на вычислительные узлы.

1.8 Другие работы по автоматизации параллельного программирования

Отметим следующие работы, развивающие различные подходы в области автоматизации параллельного программирования: LINDA [73], язык DINO [74], система MC# 2.0 [75], Q-determinant [76], о переиспользовании кода [77] и пр. [78-80].

1.9 Выводы

Обзор существующих работ позволяет сделать следующие основные выводы. Во-первых, тема действительно является актуальной, на протяжении десятилетий ведётся активная её проработка большим количеством научных коллективов. Во-вторых, проблема далеко не закрыта. Имеется множество частных подходов, обеспечивающих приемлемое по качеству конструирование параллельных программ в отдельных предметных областях, но доминирующим в научном компьютерном моделировании до сих пор является низкоуровневое программирование с использованием различных коммуникационных библиотек. Это означает, что высокоуровневые средства программирования ещё далеки от удовлетворения потребностей пользователей суперЭВМ.

2. Представление алгоритма и системные алгоритмы

В главе прорабатываются теоретические вопросы, необходимые для создания требуемой системы автоматического конструирования параллельных программ. Глава структурирована следующим образом. В разделе 2.1 изложена неформальная постановка задачи автоматического конструирования параллельной программы, в разделе 2.2 разрабатываются требования к системе автоматического конструирования параллельных программ, далее в разделе 2.3 вводятся формальные определение представления алгоритма и его исполнения, а в разделе 2.4 предлагаются алгоритмы, на основе которых может быть построена требуемая система. Завершается глава обсуждением полученных результатов и возможностей дальнейшего развития подхода (раздел 2.5).

2.1 Неформальная постановка задачи

В основе настоящей работы лежит подход, предложенный в [8]. Рассмотрим неформально, как в рамках этого подхода ставится задача синтеза (конструирования) параллельной программы. В качестве исходных данных для синтезатора даётся формальное описание предметной области в виде рекурсивно перечислимого множества модулей знаний. Каждый модуль знаний представляет собой триплет ⟨in, mod, out⟩, где in и out — два конечных подмножества величин предметной области (переменных), а mod — программный модуль (например, последовательная подпрограмма) с помощью которого можно вычислить значения величин out, подав значения величин in ему на вход. Триплет, таким образом, задаёт возможность вычисления одних величин предметной области из других. Множество же триплетов в целом называется вычислительной моделью и задаёт все возможности вычисления одних величин предметной области из других, которые могут быть потенциально использованы для автоматического синтеза программ решения задач в этой предметной области.

Задача конструирования программы ставится путём определения на вычислительной модели двух множеств переменных: ⟨V, W⟩ (эта пара множеств называется постановкой задачи на вычислительной модели). При этом имеется ввиду, что требуемая программа должна получать на вход значения переменных V, а в результате исполнения вырабатывать значения переменных W. Работа конструируемой программы сводится к запуску модулей триплетов вычислительной модели в подходящем порядке. А именно, путём применения модулей к переменным in, значения для которых даны или уже вычислены, вычисляются значения переменных out. Это, в свою

очередь, открывает возможность применения других модулей к полученным значениям. Этот процесс продолжается до тех пор, пока значения всех переменных из W не окажутся вычисленными, что и будет решением задачи. При этом подразумевается, что в описании предметной области имеются все необходимые триплеты для вычисления значений переменных из W, если значения всех переменных из V даны. Таким образом, по формальному описанию предметной области в виде вычислительной модели и постановке задачи на этой вычислительной модели возможно автоматически конструировать программу, в том числе параллельную, которая будет вычислять значения переменных из W, если ей подать на вход значения переменных из V. На рисунке 2.1 приведён пример вычислительной модели и постановки задачи на ней. Для решения поставленной задачи конструируемая программа должна будет применить сначала модуль a или b, затем модуль d. Модуль c для решения данной задачи не требуется.

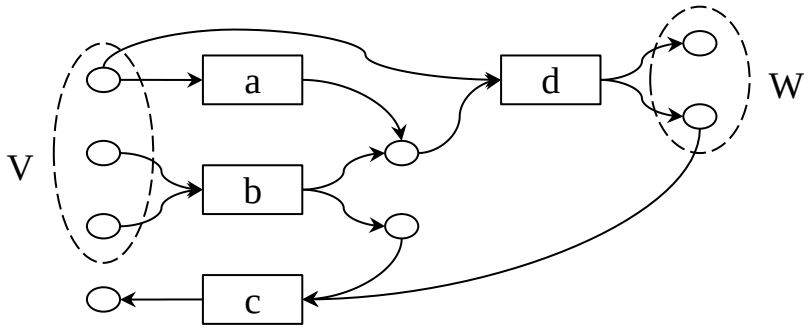

Рисунок 2.1. Пример вычислительной модели и постановки задачи на ней. Кругами обозначены переменные, прямоугольниками — модули триплетов, а дуги показывают принадлежность переменной к множествам in (исходящие) и out (входящие) модулей.

В рамках настоящей работы эта постановка задачи конструирования параллельной программы рассматривается в упрощённом виде. А именно, в исходной постановке описание предметной области содержит множество триплетов, из которых, вообще говоря, лишь подмножество требуется для решения поставленной задачи ⟨V, W⟩ (в примере на рисунке 2.1 это могут быть модули b и d). В рамках теории структурного синтеза параллельных программ на вычислительных моделях предусмотрен этап отсечения ненужных триплетов, который называется этапом вывода алгоритма или этапом планирования. В результате этого этапа формируется множество триплетов для решения конкретной задачи (⟨V, W⟩). В рамках настоящей работы рассматриваются только такие вычислительные модели, в которых имеются только необходимые триплеты, а этап планирования, следовательно, вырождается (результатом вывода является всё множество триплетов). Таким образом, рассматривается задача автоматического конструирования параллельной программы, реализующей вычисление величин

W из V, по описанию алгоритма в виде рекурсивно-перечислимого множества триплетов путём вызова модулей в допустимом порядке. При этом собственно конструирование такой программы проблемы не представляет, если не брать во внимание нефункциональные требования. Проблему представляет конструирование программы, соответствующей нефункциональным требованиям. В области численного моделирования на суперЭВМ основным нефункциональным требованием является эффективность параллельной программы.

С точки зрения эффективности параллельной программы определяющим фактором является принятие решений о способе исполнения триплетов, таких как выбор конкретного частичного порядка выполнения триплетов, выбор узла мультикомпьютера, на котором исполняется триплет, выбор узла (узлов) для хранения значений переменных в те или иные моменты выполнения программы и т.п. Также важным фактором являются накладные расходы на реализацию принятых решений.

Также отметим, что в соответствии с подходом [8] задача автоматического конструирования параллельных программ рассматривается в уже хорошо изученных предметных областях в том смысле, что для таких предметных областей уже имеются отработанные эффективные алгоритмы и их программные реализации, хотя бы последовательные. Это ограничение позволяет рассматривать задачу конструирования параллельной программы не вообще, а из относительно крупных отработанных последовательных модулей-блоков, тем самым снижая относительную долю накладных расходов на организацию параллельного исполнения и взаимодействия этих модулей. В настоящей работе класс приложений ограничен следующей практически значимой областью. Класс прикладных алгоритмов — матрично-векторные операции и итерационные численные методы на регулярных сетках, класс входных данных (по объёму) — данные, требующие применения мультикомпьютеров, класс вычислителей — мультикомпьютеры суперкомпьютерного типа.

2.2 Разработка требований к системе и её компонентам

Система автоматического конструирования эффективных и масштабируемых параллельных программ в соответствии с изложенной неформальной постановкой задачи должна соответствовать следующим требованиям.

Во-первых, реализация множества триплетов должна осуществляться в общем случае в два этапа — трансляция и исполнение (а не реализовываться только как транслятор или только как интерпретатор). Трансляция нужна для повышения эффективности реализации множества

триплетов за счёт преобразования исходно декларативного описания этого множества в реализующую его конкретным способом императивно-декларативную распределённую программу. Т.о., программа является «свёрткой» (в смысле [1]) потенциально бесконечного множества триплетов в конечные ресурсы мультикомпьютера во времени на основе решений, принятых транслятором и/или исполнительной системой. Исполнение, вообще говоря, должно осуществляться под управлением некоторой распределённой исполнительной системы, обеспечивающей динамические свойства программы, например, динамически балансируя нагрузку на вычислитель или уточняя порядок реализации триплетов в зависимости от конкретной сложившейся ситуации. Исполнительная система может рассматриваться как часть конструируемой транслятором программы или как библиотека времени исполнения — в данном случае это непринципиально. Принципиально, что конструируемая транслятором программа должна сохранять свою триплетную структуру хотя бы отчасти, чтобы обеспечить возможность динамического влияния на ход исполнения программы.

Во-вторых, исходное, декларативное описание алгоритма в виде множества триплетов не должно быть привязано к вычислителю и не должно ограничивать возможности переупорядочивания исполнения триплетов (кроме имеющихся информационных зависимостей между ними) чтобы не ограничивать переносимость описания алгоритма и возможности его настройки на конкретное оборудование или особенности обрабатываемых данных.

В-третьих, система должна быть расширяемой по системным алгоритмам транслятора и исполнительной системы. Это связано с тем, что в разных предметных областях потребуются различные частые и эвристические подходы к повышению эффективности конструируемых программ, а сама система должна развиваться путём накопления различных системных алгоритмов.

Также анализ исследуемой предметной области позволяет сформулировать основные требования к системе, соответствующие теме работы:

- Явно-параллельное крупноблочное представление данных и вычислений прикладного алгоритма,
- явное представление информационных зависимостей между частями прикладного алгоритма,
- отсутствие привязки данных и вычислений к аппаратным ресурсам, в частности, отсутствие ограничений на порядок выполнения операций реализуемого алгоритма (кроме информационных зависимостей),
- возможность автоматически динамически перераспределять данные и вычисления по узлам мультикомпьютера,

- возможность настраивать исполнение прикладного алгоритма,
- возможность расширения системы автоматического конструирования новыми системными алгоритмами, расширяющими область применения системы и/или улучшающими качество конструируемых программ,
- возможность вручную влиять на управление ходом исполнения конструируемой программы.

Существенно, что конструирование параллельной программы подразумевает строгое соответствие функциональной спецификации (т.е. программа должна выдать верный результат в итоге), но нефункциональные требования могут быть удовлетворены в большей или меньшей степени, в зависимости от возможностей конкретной системы, особенностей вложенных в неё частных алгоритмов и эвристик и их соответствия конкретной задаче. Тут уместно говорить о компромиссе между эффективностью программы, построенной автоматически, и экономией усилий на её разработку.

2.3 Фрагментированный алгоритм

Для представления множества триплетов в форме, подходящей для конструирования параллельных программ, было разработано понятие фрагментированного алгоритма, которое стало основой для входного языка разрабатываемой системы.

2.3.1 Модель фрагментированного алгоритма

Ниже излагается ряд определений, вводящих понятие фрагментированного алгоритма и его исполнения. Неформально идею, стоящую за этим формализмом, можно выразить следующим образом. Для представления триплета вводится понятие фрагмента вычислений (ФВ). Каждый ФВ — это единица вычислений, имеющая входные и выходные параметры, идентифицируемые индексированными именами, для представления которых вводится понятие ссылки.

Сначала определим понятие фрагментированного алгоритма, а затем определим понятие исполнения фрагментированного алгоритма.

Опр 1. Пусть имеется некоторое конечное множество N, которое будем называть **множеством имён**, а его элементы — **именами**.

Опр. 2. Пусть имеется некоторое конечное множество I, которое будем называть **множеством индексных переменных**, а его элементы — **индексными переменными**.

Опр. 3. Пусть имеется некоторое конечное множество A, которое будем называть **множеством фрагментов кода**, а его элементы — **фрагментами кода** (ФК).

Опр. 4. Пусть для каждого ФК $a \in A$ задана величина $in(a) \in \mathbb{N}$, называемая **числом входных параметров** ФК.

Опр. 5. Пусть для каждого ФК $a \in A$ задана величина $out(a) \in \mathbb{N}$, называемая **числом выходных параметров** ФК.

Опр. 6. Определим понятие **ссылки** следующим образом:

1. Любое имя $n \in N$ — ссылка;
2. Любая индексная переменная $i \in I$ — ссылка;
3. Пусть $n \in N$ и $r_1,...,r_L$ — ссылки, где $L \in \mathbb{N}$ и $0<L$, тогда кортеж $\langle n,r_1,...,r_L\rangle$ — ссылка;
4. Любое целое число $m \in \mathbb{N}$ — ссылка

Опр. 7. Определим понятие **оператора** следующим образом:

1. Пусть $a \in A$, $L \in \mathbb{N}$, $L=in(a)+out(a)$ и $r_1,...,r_L$ — ссылки; тогда $\langle a,r_1,...,r_L\rangle$ — оператор. Операторы такого вида будем называть **операторами исполнения.**
2. Пусть $i \in I$, f и l — ссылки, а B — конечное множество операторов, тогда $\langle i,f,l,B\rangle$ — оператор. Операторы такого вида будем называть **операторами арифметического цикла.**
3. Пусть $i \in I$, c, b и e — ссылки, а B — конечное множество операторов. Тогда $\langle i,c,b,e,B\rangle$ — оператор. Операторы такого вида будем называть **операторами цикла с предусловием.**

Опр. 8. **Фрагментированный алгоритм** (ФА) — это кортеж вида $\langle N,I,A,O\rangle$, где O — это конечное множество *операторов*.

Теперь определим исполнение ФА, предварительно изложив идею неформально. Исполнение ФА определяется как дискретная последовательность состояний, каждое из которых характеризуется множеством ФВ, которые требуется исполнить, и множеством фрагментов данных (ФД), которые являются значениями параметров ФВ. Определяется начальное состояние и правило получения следующего состояния из предыдущего, что соответствует исполнению одного ФВ.

Опр. 9. **Фрагмент данных** (ФД) — это кортеж вида $\langle n,d_1,...,d_L\rangle$, где $n \in N$, $L \in \mathbf{N}$, $L \geq 0$, а $d_1,...,d_L$ — целые числа.

Опр. 10. Определим частичную функцию int: $DF \rightarrow \mathbb{Z}$, где DF — множество всех фрагментов данных. Если функция int определена на некотором ФД d, то будем называть такой ФД **целочисленно означенным**, а его целочисленным значением будем считать $int(d) \in \mathbb{Z}$.

Опр. 11. **Контекстом** будем называть частичную функцию ctx: $I \to \mathbb{Z}$, а индексные переменные, на которых эта функция определена, будем называть **заданными** в контексте.

Опр. 12. **Фрагмент вычислений** (ФВ) — это кортеж вида ⟨o,c⟩, где o — это оператор, а c — контекст.

Опр. 13. **Состоянием** будем называть пару ⟨DF,CF⟩, где DF — это конечное множество фрагментов данных, а CF — конечное множество фрагментов вычислений.

Опр. 14. **Расширенным контекстом** будем называть пару ⟨DF,c⟩, где DF — множество ФД, а c — контекст.

Опр. 15. Будем говорить, что ссылка r **целочисленно означена** в расширенном контексте ⟨DF,c⟩, в одном из следующих случаев:

1. если $r \in N$, $\langle r \rangle \in DF$ и int(⟨r⟩) определена — целочисленным значением считается int(⟨r⟩);
2. если $r \in I$ и r задана в контексте c — целочисленным значением считается c(r);
3. если $r=\langle n,r_1,...,r_L \rangle$, ссылки $r_1,...,r_L$ целочисленно означены значениями $d_1,...,d_L$ соответственно, $\langle n,d_1,...,d_L \rangle \in DF$ и $int(\langle n,d_1,...,d_L \rangle)$ определена — целочисленным значением считается $int(\langle n,d_1,...,d_L \rangle)$;
4. если $r \in \mathbb{N}$, то $int(r)=r$.

Целочисленное значение ссылки r в расширенном контексте ⟨DF,c⟩ будем обозначать int(r, DF, c).

Опр. 16. Пусть дан расширенный контекст ⟨DF,c⟩. Будем говорить, что ссылка $r\langle n,r_1,...,r_q \rangle$ **соответствует** ФД $df=\langle n,d_1,...,d_p \rangle$ в этом расширенном контексте, если $\forall i \in \{1,p\}$ ссылка r_i целочисленно означена в расширенном контексте ⟨DF,c⟩, и её значение равно d_i.

Опр. 17. Пусть даны ФД $df=\langle n,d_1,...,d_p \rangle$ и ФВ $cf=\langle o,c \rangle$, где $o=\langle a,r_1,...,r_q \rangle$ — оператор исполнения, и состояние $S=\langle DF,CF \rangle$. ФД df называется **аргументом** для cf в состоянии S, если если $\exists k \in \mathbf{N}$, $1 \le k \le q$, такое что r_k соответствует ФД df в расширенном контексте ⟨DF,c⟩. При этом если $k \le in(a)$, то аргумент называется **входным**, иначе — **выходным**. Множество всех входных аргументов ФВ cf в состоянии S будем обозначать ins(cf,S), а выходных — outs(cf,S). Ткаже будем называть входными аргументами $ins(\langle i,f,l,B \rangle,S)=\{f,l\}$ и $ins(\langle i,c,b,e,B \rangle)=\{c,b\}$ для операторов арифметического цикла и цикла с предусловием соответственно.

Опр. 18. Будем говорить, что состояние $S'=\langle DF',CF' \rangle$ **получается из состояния** $S=\langle DF,CF \rangle$ путём выполнения ФВ $cf=\langle o,c \rangle \in CF$, где o — оператор исполнения, если:

1. CF'=CF\cf;
2. ins(cf, S) ⊆ DF;
3. DF ∪ outs(cf,S)=DF'.

Опр. 19. **Подстановкой** целочисленного значения $k \in \mathbb{Z}$ индексной переменной i в контекст c будем называть контекст c'=subst(c,i,k), определяемый следующим образом. Пусть $j \in I$, тогда c'(j)=k, если i=j и c'(j)=c(j) если $i \neq j$.

Опр. 20. Будем говорить, что состояние $S'=\langle DF',CF'\rangle$ **получается из состояния** $S=\langle DF,CF\rangle$ путём выполнения ФВ $cf=\langle o,c\rangle \in CF$, где $o=\langle i,f,l,B\rangle$ — оператор арифметического цикла, если:

1. $F \leq L$, где F=int(f,DF,c), L=int(l,DF,c);
2. $CF'=CF\backslash cf \cup \{B_F,...,B_L,\}$, где $B_k= \cup_{u \in B}\langle u,subst(c,i,k)\rangle$;
3. DF'=DF;
4. $ins(cf, S) \subseteq DF$.

Опр. 21. Будем говорить, что состояние $S'=\langle DF',CF'\rangle$ **получается из состояния** $S=\langle DF,CF\rangle$ путём выполнения ФВ $cf=\langle o,c\rangle \in CF$, где $o=\langle i,c,b,e,B\rangle$ — оператор цикла с предусловием, если $ins(cf, S) \subseteq DF$ и:

1. При int(c,DF,subst(c,i,int(b,DF,c)))=0:
    a. $CF'=CF\backslash cf$;
    b. $DF'=DF \cup df_e$, где df_e соответствует ссылке e в расширенном контексте $\langle DF,c\rangle$ и $int(df_e)=int(b,DF,c)$;
2. При $int(c,DF,subst(c,i,int(b,DF,c))) \neq 0$:
    a. $CF'=CF\backslash cf \cup B' \cup cf'$, где $B'=\cup_{u \in B}\langle u,subst(c,i,int(b,DF,c))\rangle$, а $cf'=\langle i,c,int(b,DF,c)+1,e,B\rangle$;
    b. DF'=DF.

Опр. 22. Состояние S называется **финальным**, если не существует состояния S', которое может быть получено из состояния S ни одним из трёх описанных выше способов.

Опр. 23. **Исполнением ФА** $\langle N,I,A,O\rangle$ над некоторым конечным множеством входных ФД X называется конечная или бесконечная последовательность состояний $R=S_0,S_1,...$, удовлетворяющая следующим свойствам:

1. $S_0=\langle X,CF \rangle$, где $CF=\cup_{o \in O}\langle o,c_0\rangle$, $c_0:\emptyset \rightarrow \mathbb{Z}$;
2. Если последовательность состояний конечна ($R=S_0,...,S_F$), то последнее состояние S_F является финальным и только оно.
3. Для любой пары смежных состояний $\langle S_i,S_{i+1}\rangle$ верно, что S_{i+1} получается из S_i одним из трёх вышеописанных способов (опр. 18, 20 или 21).

Опр. 25. **Интерпретацией** исполнения ФА назовём произвольную частично определённую функцию T, которая ставит в соответствие ФД элемент из некоторого множества D, называемый значением этого ФД; и ставит в соответствие ФК $a \in A$ функцию из D^n в D^m, где n=in(a), а m=out(b).

Опр. 26. Интерпретация T является **непротиворечивой**, если для любого ФВ $f=\langle o,c\rangle$, где $o=\langle a,r_1,...,r_{n+m}\rangle$ — оператор исполнения, выполняется равенство $T_a(T(d_1),...,T(d_n))=\langle T(d_{n+1}),...,T(d_{n+m})\rangle$, где $T_a=T(a)$, а $d_1,...,d_{n+m}$ — ФД, соответствующие ссылкам $r_1,...,r_{n+m}$ в расширенном контексте $\langle DF,c\rangle$.

С иллюстративным примером записи алгоритма вычисления числа Фибоначчи в виде фрагментированного алгоритма можно ознакомиться в приложении А.

Отметим, что одно и то же начальное состояние может соответствовать более чем одному исполнению ФА, т.к. из некоторого состояния может получиться более одного следующего состояния (могут исполниться разные ФВ). Предлагаемой моделью этот порядок не определяется, и это отражает недетерминизм исполнения асинхронных программ. Какая конкретно функция вычисляется тем или иным ФК определяется функцией интерпретации. Что может являться значением ФД — вынесено за рамки модели за исключением случаев косвенной индексации, когда значение индекса в индексированном имени зависит от значения фрагмента данных, а также за исключением входных параметров операторов циклов. В этих случаях значение ФД должно быть приводимо к целочисленному значению, что отражено в предлагаемой модели частично определённой на множестве ФД функцией int (опр. 10).

ФА является универсальным представлением алгоритма в смысле возможности представить любую частично-рекурсивную функцию (полнота по Тьюрингу). Это существенно, так как этим гарантируется принципиальная возможность выразить любой алгоритм средствами ФА. Доказательство этого свойства вынесено в приложение Б. Оно основывается на повторении в терминах ФА определения частично-рекурсивной функции.

2.3.2 Эффективная параллельная реализация фрагментированного алгоритма

В подразделе рассматривается, как модель ФА может использоваться на практике применительно к конструированию и исполнению эффективных параллельных программ.

Процесс вычислений рассматривается как *последовательность* сменяющих друг друга состояний, хотя при практическом использовании модели ФА это не означает последовательного исполнения. При параллельном исполнении можно считать, что выходные ФД некоторого ФВ появляются не обязательно в состоянии, следующем за исполнением ФВ, а через некоторое конечное число состояний впоследствии, причём через какое именно — не уточняется. С такой поправкой мы получаем ситуацию, что различные ФВ могут исполняться как последовательно, так и параллельно.

При реализации алгоритмов обычно выделяют входные, выходные и промежуточные данные. Входные данные программа получает на вход, в своей работе использует входные и промежуточные данные, а в результате своей работы выдаёт выходные данные. В модели ФА в начальном состоянии данных нет вообще, и считается, что часть ФВ отвечает за ввод данных в систему. Например, это могут быть ФВ без входных ФД, но имеющие выходные ФД, которые и образуют входные данные алгоритма. Некоторое подмножество ФД в конечном состоянии, аналогично, образует выходные ФД. Какие именно ФД являются выходными выносится за рамки модели.

Применять ФА на практике предлагается следующим образом. Разрабатывается язык описания ФА, пригодный для использования компьютером. Пользователь описывает ФА на этом языке. Кроме того, для каждого ФК он предоставляет в оговоренном формате модуль, его реализующий — например, процедуру, разработанную на традиционном языке последовательного программирования (таком, как C++). Оговаривается также способ вычисления функции int для целочисленно означенных ФД. После этого начинает работать система, которая исполняет ФА на заданных входных данных на мультикомпьютере. Это может быть распределённый интерпретатор, прямо разбирающий и реализующий операторы языка; либо это может быть транслятор, который сгенерирует параллельную программу; либо смесь этих двух вариантов (см. главу 3). Результатом исполнения становится получение выходных значений задачи в оговоренном виде (например, в виде ФВ, потребляющих значения заданных ФД и сохраняющих их на файловую систему).

Для рассмотрения вопроса эффективности исполнения ФА введём понятие **поведения** как множества допустимых вариантов исполнения ФА на заданных классах входных данных и вычислителей. Поясним это определение на примерах. Если ФА состоит из цепочки информационно зависимых ФВ, то порядок их выполнения однозначно определён и не может варьироваться (тут и далее ФВ b считается информационно зависимыми от a, если $out(a) \cap in(b) \neq \emptyset$). Если исполнять этот ФА на мультикомпьютере с единственным узлом, то поведение будет содержать всего один элемент — последовательное исполнение в порядке информационных зависимостей. Если, например, добавить второй узел в мультикомпьютер, то ФА можно будет исполнить 2^N способами, где N — количество ФВ в цепочке, и каждый ФВ может быть отображён на первый или второй узел. Если в ФА будут присутствовать информационно-независимые ФВ, то их можно будет исполнять в любом порядке друг относительно друга, в том числе параллельно. Все возможные варианты и образуют множество поведение.

Поведение играет ключевую роль с точки зрения эффективности исполнения ФА и охватывает такие аспекты исполнения ФА как порядок выполнения ФВ (в рамках имеющихся

информационных зависимостей), распределение и динамическое перераспределение ФВ и ФД по узлам мультикомпьютера, сборка мусора (удаление ненужных более ФД).

Для повышения эффективности исполнения ФА на практике предлагается использовать «подсказки» пользователя — ограничения на поведение. Например, программист может явно указать, какое требуется (или, наоборот, недопустимо) распределение фрагментов по узлам мультикомпьютера, порядок выполнения ФВ и т.п. Будем называть такие ограничения **рекомендациями**. Рекомендации позволяют частично редуцировать поведение ФА (т.е. исключить из этого множества часть элементов). Возможность вручную задавать рекомендации имеет принципиальное значение ввиду того, что выбор эффективного элемента из поведения является алгоритмически труднорешаемой задачей. Это, в свою очередь, означает, что удовлетворительной эффективности за приемлемое время возможно достичь лишь на ограниченном круге задач, определяемых эвристиками и специализированными системными алгоритмами, заложенными в систему программирования. Использование же рекомендаций позволяет программисту частично определять поведение, сводя задачу поиска удовлетворительной реализации в поведении к выполнимой за разумное время. Например, человек может указать желаемое распределение ФД по узлам мультикомпьютера, основываясь на котором система программирования сможет эффективно реализовать ФА, в то время как без данной подсказки эффективность исполнения ФА имеющимися эвристиками была бы неудовлетворительной.

2.3.3 Анализ предлагаемой модели

ФА не привязан к ресурсам (памяти, процессорным элементам, сети), а порядок выполнения ФВ ограничен только информационными зависимостями. Это позволяет автоматически, в том числе динамически, отображать фрагменты на ресурсы и выбирать порядок выполнения ФВ, тем самым настраивая реализацию ФА на особенности конкретного вычислителя и обрабатываемых данных.

Структура ФА, существенная с точки зрения организации параллельных вычислений, выражена явно: явно видны независимые части вычислений (информационно независимые ФВ), явно видны зависимости между отдельными частями вычислений, явно видно, с какими данными идёт работа в той или иной части программы. Всё это позволяет статически и динамически анализировать ФА, в значительной степени прогнозировать ход его вычислений, автоматически выявлять ненужные более данные (сборка мусора) и т.п.

Фрагментом кода может выступать любая процедура без побочных эффектов, реализуемая на одном вычислительном узле. Это позволяет работать на уровне крупноблочного параллелизма, уменьшая накладные расходы на реализацию ФА. Также это позволяет использовать накопленный человечеством багаж отлаженных высокоэффективных последовательных подпрограмм, применять развитый инструментарий последовательного программирования (в первую очередь — последовательные компиляторы) в разработке ФК, а также использовать в качестве ФК подпрограммы для сопроцессоров (напр., CUDA).

ФА как представление алгоритма не ограничивает масштабируемости представляемого алгоритма, т.к. не накладывает необходимости ни в применении централизованных алгоритмов, ни в коммуникациях.

2.4 Основные системные алгоритмы

В целом создание системы (например, интерпретатора), реализующей ФА по его формальному описанию, не вызывает существенных затруднений. Например, она может быть реализована «наивным» способом как композиция распределённой базы данных и распределённого же портфеля задач (можно рассматривать ФД как единицы информации в базе данных, а ФВ — как задачи в портфеле задач). Но такая наивная реализация наталкивается на ряд «подводных камней», ограничивающих её эффективность и масштабируемость. В разделе рассматриваются эти проблемы и предлагаются их решения в виде распределённых децентрализованных масштабируемых системных алгоритмов. Все предложенные решения не являются наилучшими или оптимальными в каком-либо смысле, потому что проблемы, ими решаемые, относятся к алгоритмически труднорешаемым. На практике такие задачи должны решаться частными узкоспециализированными или эвристическими алгоритмами, обеспечивающими приемлемое качество решения в конкретной практической ситуации (для конкретного класса приложений). Тем не менее, хотя бы какими-то удовлетворительными решениями эти проблемы закрыть необходимо, чтобы обеспечить реализуемость задачи автоматического конструирования параллельной программы по описанию ФА, чему и посвящён настоящий раздел. В дальнейшем эти решения можно и нужно улучшать, что требует отдельных исследований.

Большинство представленных в разделе алгоритмов являются распределёнными, децентрализованными и динамическими, поэтому их описание строится по следующей схеме. Алгоритм описывается в виде перечня ситуаций, в которой может находиться тот или иной вычислительный узел, и описания порядка действий в каждой из такой ситуации.

2.4.1 Децентрализованный поиск объектов в распределённой системе

При распределённом исполнении ФА возникает проблема поиска местоположения объектов — ФД и ФВ. Под местоположением ФД понимается вычислительный узел, на котором хранится значение этого ФД. Под местоположением ФВ понимается вычислительный узел, на котором ФВ исполняется. ФД вырабатывается на некотором узле мультикомпьютера, а потреблён должен быть, вообще говоря, на другом узле (других узлах), причём количество и местоположение ФВ-потребителей, а также место выработки ФД, может определяться динамически. Проблема осложняется тем, что количество ФД и ФВ может зависеть от входных данных, ФВ[2] и ФД могут мигрировать с узла на узел в результате динамической балансировки нагрузки на узлы. Требование масштабируемости не допускает использование централизованных хранилищ, и даже хранение полной информации о местоположении ФД и ФВ на одном или нескольких узлах не допускается, т.к. при больших размерах мультикомпьютера и задачи даже такая информация может не умещаться в памяти одного узла, а издержки, возникающие из-за дополнительного обращения к такому узлу, станут недопустимо высокими. Как следствие, нужен децентрализованный масштабируемый механизм, обеспечивающий поиск ФД и ФВ на узлах, который бы использовал только локальные взаимодействия между вычислительными узлами и при этом работал эффективно для широкого класса задач.

Ввиду заведомой неуниверсальности таких алгоритмов поиска необходимо обеспечить возможность использования различных алгоритмов поиска, которые бы использовались в подходящих случаях, заменяя один другой. Для обеспечения этой возможности вводится понятие локационной функции locate, которая для заданного узла r в момент времени t произвольному идентификатору id ставит в соответствие номер узла n=locate(r, t, id), $n \in N_r$, где N_r — множество узлов, соседних с узлом r в смысле сетевой топологии, $r \in N_r$. Для заданного начального узла r_0 и идентификатора id в некоторый момент времени t локационная функция задаёт последовательность узлов $r_0, r_1, ..., r_k$, где r_{i+1}=locate(r_i,t,id). Будем считать, что если r=locate(r,t,id), то узел r является местоположением объекта с идентификатором id в момент времени t. Потребуем от функции locate, чтобы всегда существовало некоторое k, такое что $r_{k+1}=r_k$, т.е. чтобы локационная функция приводила за конечное количество переходов к конкретному узлу, который и является местоположением объекта в момент времени t. Рациональный смысл этой функции в осуществлении навигации, она представляет некоторый способ найти искомый объект по его идентификатору следуя маршруту, выдаваемому локационной функцией. Зависимость от

[2] под миграцией ФВ понимается динамическое изменение решения о том, на каком вычислительном узле ФВ должен исполняться.

времени отражает тот аспект, что распределение объектов (ФД и ФВ) по мультикомпьютеру может динамически изменяться. Идентификаторы (индексированные имена) ФД определены в опр. 6 (раздел 2.3.1). Идентификаторы ФВ могут быть определены аналогичным образом. Иллюстрация к идее локационной функции приведена на рисунке 2.2.

Рисунок 2.2. Локационная функция — это абстракция алгоритма децентрализованного поиска объекта по его идентификатору. На каждом узле мультикомпьютера она определяет соседний узел для продолжения поиска объекта по его идентификатору.

Если существует какой-то конкретный способ децентрализованного поиска объектов в распределённой системе по их идентификаторам, то его можно рассматривать как некоторую локационную функцию. Алгоритм Rope of Beads, рассматриваемый в разделе 2.4.2, предлагает практичный способ реализации локационной функции, а алгоритмы разделов 2.4.3–2.4.5 рассчитывают на то, что эта функция как-то реализована (не обязательно алгоритмом Rope of Beads).

Алгоритм поиска объекта по локационной функции может быть выражен листингом 2.1. Тут и далее временной параметр и номер текущего узла функции locate опускаются; под этим имеется ввиду, что функция locate возвращает значение на момент её вычисления и для текущего узла в процессе выполнения программы, что соответствует практике её применения. Собственно, эти параметры и вводились для того, чтобы отразить тот факт, что значение функции будет вычисляться на конкретном узле в конкретный момент времени.

Листинг 2.1. Алгоритм поиска объекта по локационной функции.

```
⊳ Вызывается при получении поискового запроса локально или по сети
01: PARAM r                    ⊳ номер текущего вычислительного узла
02: PARAM id                   ⊳ идентификатор искомого объекта (напр., ФД)
03: PARAM action               ⊳ действие, которое требуется совершить над объектом
04: SET n ← locate(id)         ⊳ определить следующий узел для поиска
05: IF n=r THEN                ⊳ если искомый узел — текущий
06:   EXECUTE action           ⊳ то выполнить действие
07: ELSE                       ⊳ иначе
```

08: **SEND** ⟨id, action⟩ **TO** n ⊳ переадресовать запрос на узел n
09: **END IF**

Преимуществом использования идеи локационной функции является возможность абстрагироваться от конкретных алгоритмов распределения и динамического перераспределения объектов, но обеспечить при этом реализацию ФА. Такая абстракция не ограничивает общности рассмотрения, т.к. если существует какой-либо способ поиска объектов, то он может рассматриваться как локационная функция. Также локационная функция не ограничивает масштабируемости, т.к. подразумевает непосредственные коммуникации лишь с соседними узлами мультикомпьютера.

2.4.2 Базовый алгоритм интерпретатора

В разделе представлен распределённый децентрализованный алгоритм, обеспечивающий реализацию ФА на мультикомпьютере. Он назван базовым потому, что показывает принципиальную схему распределённой реализации ФА, в которую включаются как составные части остальные алгоритмы раздела 2.4.

Пусть имеется мультикомпьютер, на каждом вычислительном узле которого могут находиться объекты четырёх видов:

Фрагмент данных (ФД) — пара ⟨id, val⟩, где id — индексированный идентификатор (см. опр. 9), a val — значение ФД.

Фрагмент вычислений (ФВ) — пара ⟨id, o, c⟩, где id — индексированный идентификатор, o и c — оператор и контекст (см. опр. 7 и 11).

Запрос — пара индексированных идентификаторов ⟨dfid, cfid⟩ ФД и ФВ соответственно.

Ответ — пара ⟨r, val⟩, где r — запрос, a val — значение ФД.

Примечание — Значением ФД может быть любая величина, например, число или блок матрицы; в данном случае это несущественно, главное, чтобы это значение было представимо в пределах одного вычислительного узла (помещалось в память узла).

Для объектов всех четырёх видов будем считать, что определена локационная функция, т.к. все четыре вида объектов включают в себя идентификаторы, при этом для запроса локационная функция применяется к идентификатору ФД, а для ответа — к идентификатору ФВ, т.е. locate(⟨dfid, cfid⟩)=locate(dfid), locate(⟨⟨dfid, cfid⟩, val⟩)=locate(cfid).

В начальный момент времени по произвольным узлам в единственном экземпляре распределяется «начальное множество ФВ» — множество ФВ, соответствующих ФВ в начальном

состоянии исполнения ФА (опр. 23). Далее каждый вычислительный узел работает по алгоритму, представленному на листинге 2.2:

Листинг 2.2. Базовый алгоритм интерпретатора.

```
▷ Точка входа
01: PARAM r                                  ▷ номер текущего узла
02: IF r=корневой узел THEN                  ▷ на одном (корневом) узле
03:   FOR ⟨id, o, c⟩ ∈ начальное множество ФВ DO
04:     CALL create_cf(⟨id, o, c⟩)           ▷ создать все ФВ из начального множества
05:   END FOR
06: END IF

07: SUB create_cf(⟨id, o, c⟩)                ▷ подпрограмма создания ФВ
08:   FOR dfid ∈ ins(o) DO                   ▷ для всех входных ФД заданного ФВ
09:     SPAWN ⟨dfid, id⟩                     ▷ создать запрос на ФД для ФВ
10:   END FOR
11: END SUB

12: ON EVENT при появлении ФВ ⟨id, o, c⟩     ▷ т.е. при создании или получении по сети
13:   LET n ← locate(id)                     ▷ определить узел размещения ФВ
14:   IF n≠r THEN                            ▷ если это не текущий узел
15:     SEND ⟨id, o, c⟩ TO n                 ▷ то отправить ФВ на узел n
16:   ELSE                                   ▷ иначе
17:     CALL check_run_cf(id)                ▷ попробовать исполнить ФВ
18:   END IF
19: END EVENT

20: ON EVENT при появлении запроса ⟨dfid, cfid⟩ ▷ т.е. при создании или получении по сети
21:   LET n ← locate(dfid)                   ▷ определить узел размещения запрашиваемого ФД
22:   IF n=r THEN                            ▷ если узел совпадает с текущим
23:     LET val ← дождаться ФД(dfid)         ▷ дождаться появления ФД на узле
24:     SPAWN ⟨⟨dfid, cfid⟩, val⟩            ▷ создать ответ на запрос
25:   ELSE                                   ▷ иначе
26:     SEND ⟨dfid, cfid⟩ TO n               ▷ переслать запрос на узел n
27:   END IF
28: END EVENT

29: ON EVENT при появлении ответа ⟨⟨dfid, cfid⟩, val⟩
30:   LET n ← locate(cfid)                   ▷ определить узел размещения ФВ
31:   IF n=r THEN                            ▷ если узел совпадает с текущим
32:     CALL check_run_cf(cfid)              ▷ попробовать исполнить ФВ
```

```
33:   ELSE                                  ⊳ иначе
34:     SEND ⟨⟨dfid, cfid⟩, val⟩ TO n       ⊳ переслать ответ на узел n
35:   END IF
36: END EVENT

37: SUB check_run_cf(cfid)                  ⊳ исполнить ФВ, если все входные ФД имеются на узле
38:   IF ФВ cfid есть на узле THEN           ⊳ убедиться, что сам ФВ имеется на узле
39:     LET ⟨cfid, o, c⟩ ← получить ФВ(cfid)
40:     FOR dfid ∈ ins(o) DO                        ⊳ для всех входных ФД этого ФВ
41:       IF ответ ⟨⟨dfid, cfid⟩, val⟩ отсутствует на узле THEN
42:         RETURN                          ⊳ если ФД отсутствует, выйти из подпрограммы
43:       END IF
44:     END FOR
45:   EXEC ⟨cfid, o, c⟩                     ⊳ если дошли до этой точки, значит все ФД в наличии, исполнить ФВ
46: END SUB
```

В описании алгоритма отсутствует сборка мусора, она отдельно рассмотрена в разделе 2.4.5. Также опущено рассмотрение ситуации, когда локационная функция изменяется динамически. В этом случае все объекты на узле (ФВ, ФД, запросы и ответы) должны быть пересланы на узлы, определяемые новыми значениями локационной функции. На работу алгоритма это не влияет.

Рассмотрим, для примера, как реализуется динамическая балансировка нагрузки на узлы. В соответствии с подходом, описанным в подразделе 2.4.1, локационная функция зависит от времени, и в определённый момент может оказаться, что значения функции для некоторого количества идентификаторов изменились. В этом случае, в соответствии с шагом 2.а все объекты, для которых значение функции изменилось, будут мигрировать с узла на узел до тех пор, пока не окажутся в новом местоположении, до следующего изменения локационной функции.

И ФД, и запросы на этот ФД окажутся за конечное число миграций на одном и том же узле, при условии, что локационная функция меняется достаточно медленно (что соответствует практике применения динамической балансировки нагрузки на узлы). То же самое верно и для ответов, которые за конечное число миграций окажутся на том же узле, что и соответствующий ФВ. Динамическая балансировка нагрузки не нарушает этого свойства.

В разделе 2.3 не было введено идентификаторов для ФВ, т.е. это не было необходимо для определения ФА и его исполнения. Но удобно ввести идентификаторы для ФВ чтобы единообразно управлять распределением и динамическим перераспределением и ФД, и ФВ по узлам мультикомпьютера. Идентификаторы для ФВ могут назначаться вручную или

автоматически, и делаться это должно с учётом локационной функции, т.к. от неё будет зависеть распределение ФВ по вычислительным узлам.

Порождение новых ФВ и ФД в процессе вычислений будет происходить на строке 45 как результат исполнения ФВ.

2.4.3 Алгоритм Rope of Beads динамического распределения фрагментов по узлам мультикомпьютера

Алгоритм Rope of Beads является одним из возможных алгоритмов вычисления локационной функции. Он обеспечивает отображение ФВ и ФД на узлы мультикомпьютера и обеспечивает динамическую балансировку вычислительной нагрузки путём перераспределения ФВ и ФД по узлам. Прежде, чем изложить сам алгоритм, рассмотрим необходимые сопутствующие вопросы.

Во-первых, для осуществления динамической балансировки нагрузки на вычислитель должна быть обеспечена возможность измерения текущей нагрузки на текущий вычислительный узел. Существует множество способов измерять нагрузку, например, по количеству ФВ (готовых к исполнению или общего количества) на узле, по фактической загрузке процессорных элементов узла, по объёму занятой памяти и т.п., и в разных случаях подходящими оказываются разные способы. Полезным или даже необходимым может оказаться осреднение загрузки узла по времени (причём выбор интервала осреднения играет существенную роль), а также возможны варианты рассмотрения не текущей загрузки вычислительного узла, а прогнозируемой через некоторое время. Способ измерения текущей нагрузки выходит за рамки алгоритма Rope of Beads и является его параметром.

Во-вторых, должна быть обеспечена информация о текущей загруженности соседних узлов. Эта информация может быть доступна с задержкой, обусловленной скоростью передачи сообщений по сети и периодичностью передачи таких сообщений. Чем выше частота обновления этой информации, тем более точно может работать динамическая балансировка нагрузки, но тем выше будет и нагрузка на сеть. Возможно также обновлять текущую нагрузку узла при значительном её изменении, а не по таймеру. Порядок обновления информации о загрузке соседних также выходит за рамки алгоритма.

В-третьих, с точки зрения эффективности конструируемой параллельной программы, первостепенную важность имеет вопрос распределения и динамического перераспределения ФВ и ФД по узлам мультикомпьютера. От этого зависит объём и дальность коммуникаций, а также

равномерность нагрузки на вычислительные узлы. И то, и другое существенно влияет на время выполнения программы и другие нефункциональные свойства.

Далее опишем алгоритм Rope of Beads. Введём вспомогательную функцию coord: ID → [0;1), которая каждой ссылке из множества ссылок ID ставит в соответствие вещественное число от 0 включительно до 1. Тут и далее будем считать, что не только ФД идентифицируются ссылками (опр. 9), но и ФВ. Прагматический смысл этой функции в том, чтобы задать отображение ФД и ФВ на некий виртуальный вычислитель с линейной сетевой топологией (отрезок). Тем самым задаётся пространственная координата всех ФД и ФВ друг относительно друга. Пусть множество вычислительных узлов линейно упорядочено. Далее разобьём отрезок [0;1) на несколько неравных частей по числу вычислительных узлов: $[0;1)=[a_0;a_1) \cup [a_1;a_2) \cup ... \cup [a_{n-1};a_n)$, где n — число узлов мультикомпьютера. Пусть любой ФД или ФВ с идентификатором id хранится на узле k, где $a_k \le \text{coord}(id) < a_{k+1}$. И пусть изменение в процессе работы алгоритма величин a_i приводит к соответствующему перемещению ФД или ФВ (листинг 2.3).

Листинг 2.3. Алгоритм Rope of Beads динамической балансировки нагрузки на узлы.

```
00: PARAM T                    ⊳ период ожидания между проверками дисбаланса
01: PARAM Θ                    ⊳ порог дисбаланса
02: PARAM r                    ⊳ номер текущего вычислительного узла
03: WHILE program running
04:   SET ΔL ← L_r − L_{r−1}   ⊳ ΔL — разница нагрузок с предыдущим узлом
05:   IF ΔL > Θ THEN           ⊳ если превышен порог дисбаланса
06:     a_r ← a_r+ΔL(a_{r+1}−a_r)/L_r   ⊳ сместить a_{r−1} пропорционально дисбалансу нагрузки
07:   END IF
08:   SET ΔL ← L_r − L_{r+1}   ⊳ ΔL — разница нагрузок со следующим узлом
09:   IF ΔL > Θ THEN           ⊳ если превышен порог дисбаланса
10:     a_{r+1} ← a_{r+1}−ΔL(a_{r+1}−a_r)/L_r   ⊳ сместить a_{r+1} пропорционально дисбалансу нагрузки
11:   END IF
12:   SLEEP T                  ⊳ ожидать следующей проверки дисбаланса
13: END WHILE
```

В листинге 2.3 подразумевается, что величина L_r (мера загруженности текущего узла) изменяется динамически внешним образом модулем оценки текущей нагрузки, а L_{r-1} и L_{r+1} — последние величины оценочной нагрузки соседних узлов, полученные по сети. Также подразумевается, что изменение границы отрезка a_r или a_{r+1} приводит к изменению этой границы и на соответствующем соседнем узле. Реализация операции сдвига границы должна реализоваться путём обмена сообщениями по сети между двумя соседними узлами, разделяющими эту границу, и только между ними.

В алгоритме поиск объектов обеспечивается следующим образом. Допустим на узле с номером r требуется найти некоторый ФД или ФВ по ссылке id ∈ ID. Тогда локационная функция будет иметь следующий вид (листинг 2.4):

Листинг 2.4. Локационная функция в алгоритме Rope of Beads.

```
01: IF coord(id) < a_r THEN
02:   RETURN r-1
03: ELSE IF coord(id) ≥ a_r+1 THEN
04:   RETURN r+1
05: ELSE
06:   RETURN r
```

В листинге 2.3 в качестве условия инициации процесса балансировки выступает условие превышения разницы оценочной нагрузки на соседние узлы пороговой величины Θ (строка 05). На практике это условие может быть заменено на другое, более подходящее конкретной ситуации, например при превышении пропорции нагрузки на двух узлах, или при превышении нагрузки в течение заданного интервала времени, и т.п. То же касается и объёма передаваемой нагрузки, он может вычисляться и иным подходящим образом.

Алгоритм может быть усовершенствован учётом неоднородности распределения вычислительной нагрузки по отрезку текущего узла. Так как фрагменты распределены по координатному отрезку неравномерно (функцией coord), то и передавать на соседний узел (т.е. изменять границу между подотрезками) следует не пропорциональную передаваемой нагрузке часть отрезка, а с учётом распределения нагрузки по отрезку.

В свою очередь, порог дисбаланса отражает тот факт, что передача нагрузки по сети не мгновенна, и зависит, в том числе, от объёма данных, которые необходимо передать. Поэтому пороговая функция может учитывать распределение ФД по координатному отрезку.

Все эти параметры алгоритма создают многообразие вариантов, позволяющих адаптировать алгоритм к конкретным условиям работы. Выбор параметров может осуществляться и вручную, и эмпирически, и на основе формулировки и решения оптимизационной задачи. Рассмотрение этого вопроса выходит за рамки настоящей работы.

Существенно, что отображение множества подотрезков на множество узлов должно выполняться с учётом сетевой топологии мультикомпьютера, т.е. чтобы соседние подотрезки находились, по возможности, на соседних узлах мультикомпьютера (подобно гамильтонову пути в графе). Это неформальное соображение будем дальше называть соседством данных и/или вычислений, понимая под этим тот факт, что важно учитывать, что чем дальше в смысле сетевой топологии данные перемещаются по мультикомпьютеру, тем больше издержек это обычно влечёт.

Рассмотрим преимущества алгоритма Rope of Beads. Алгоритм полностью децентрализован. Все взаимодействия осуществляются лишь с парой соседних узлов. Объем дополнительной информации, которую требуется хранить на каждом из узлов, пренебрежимо мал (два числа) и не зависит ни от объёма задачи, ни от количества узлов мультикомпьютера, ни от чего-либо ещё. Операция сдвига границы между подотрезками выполняется парой соседних узлов без необходимости в какой-бы то ни было синхронизации с другими узлами. Алгоритм позволяет учитывать соседство данных и вычислений путём их отображения на координатный отрезок. Учёт структуры данных и вычислений достигается тем, что соседние в смысле информационных зависимостей ФВ и ФД отображаются на близкие координаты на отрезке. В силу отображения подотрезков на узлы справедливо утверждение, что если координата находится между двух других координат, то и назначенный ей узел будет находиться между узлами, соответствующими двум другим координатам (либо совпадать с одним или обоими узлами). Другими словами, в каком порядке фрагменты отображены на отрезок — в том они и будут отображены на цепочку узлов мультикомпьютера. Алгоритм поддерживает динамическую балансировку нагрузки, причём независимо от «интенсивности» балансировки нарушения соседства данных не накапливается (что характерно для алгоритмов динамической балансировки нагрузки диффузионного типа).

Рассмотрим недостатки алгоритма. Построение эффективного отображения имён на координатный отрезок в общем случае является отдельной сложной проблемой и выносится за рамки алгоритма (и настоящей работы). Под эффективностью в данном случае понимается равномерность распределения фрагментов по координатному отрезку и относительно малый объём коммуникаций, возникающих из-за удалённости связанных фрагментов. При этом ясно, что для простых случаев эффективное отображение может быть построено автоматически, для более сложных случаев это отображение может быть дано пользователем, для ещё более сложных его построение может оказаться практически невозможным. Отметим, что эта проблема поиска эффективного отображения объектов на узлы не специфична для данного алгоритма или вообще автоматического конструирования программ, а характерна в принципе для задачи распределённого исполнения алгоритма и должна решаться, так или иначе, в любой распределённой программе.

Ещё одним недостатком алгоритма является ограниченность средств выражения знаний о соседстве данных и вычислений в виде функции coord, которая представляет собой одномерный координатный отрезок. Если задача имеет, например двух- и более мерную решётчатую структуру, или древесную (и то, и другое часто встречается на практике), то выразить это соседство в виде отображения на одномерный отрезок можно лишь приблизительно. Тем не

менее, на практике этого, во-первых, часто будет достаточно; и, во-вторых, аналогичный алгоритм может быть естественным образом расширен на случаи бóльших размерностей (n-мерный куб), а также для других координатных структур (см. [81]). Другой слабостью алгоритма является то, что и мультикомпьютер не обязательно имеет линейную сетевую топологию, поэтому при отображении отрезка на узлы учёт соседства также оказывается приблизительным. Но реальные мультикомпьютеры обычно имеют топологии (напр., толстое дерево, решётка), в которую линейная вкладывается относительно хорошо.

Практическая реализация алгоритма с использованием вещественных чисел не рекомендуется в силу того, что вещественные числа в компьютере представляются с ограниченной точностью. Это может привести к тому, что два соседних узла будут иметь различные приближённые значения для своей общей границы, что может привести к некорректной работе алгоритма. А именно, некоторая координата может считаться принадлежащей не к ровно одному, а нулю или двум и более подотрезкам вследствие ошибок округлений. Поэтому на практике следует использовать дискретный отрезок заданной длины с целочисленными границами. Алгоритм также может быть переформулирован вообще без координатного отрезка путём введения любого отношения нестрогого линейного порядка на множестве имён фрагментов.

2.4.4 Алгоритмы доставки ФД

В разделе предложено два алгоритма доставки ФД (к потребляющим ФВ), которые могут быть использованы совместно.

Алгоритм доставки по запросу. Данный алгоритм был кратко описан как часть базового алгоритма интерпретации (раздел 2.4.2). Рассмотрим его тут подробнее, сначала неформально, и в сравнении с другим алгоритмом доставки ФД. Пусть при выработке ФД некоторым ФВ на некотором вычислительном узле *А* ФД передаётся на хранение на некоторый вычислительный узел *Б*. Когда некоторый ФВ после своего создания мигрирует на вычислительный узел *В*, где будет исполняться, то он отправляет запрос на входные ФД, в том числе запрос отправляется на упомянутый узел *Б*. Когда запрос и ФД оказываются на одном узле *Б*, то копия ФД отправляется на узел *В*, где потребляется запросившим ФВ. Все передачи запроса и ФД осуществляются посредством локационной функции, т.е. локальными пересылками, между вычислительными узлами, соседними в сетевой топологии.

В листинге 2.5 представлено описание алгоритма доставки ФД. Тут предполагается, что ФВ породил запросы на свои входные ФД, а целью алгоритма является перемещение значений

запрошенных ФД (в виде ответов на запросы) на узел пребывания запросившего ФВ. Факт пребывания на одном узле и ФВ, и его входных ФД приводит к тому, что этот ФВ сможет быть исполнен.

Листинг 2.5. Алгоритм доставки ФД по запросу.

```
01: VAR D                                  ⊳ коллекция ожидающих ФД на узле
02: VAR C                                  ⊳ коллекция ожидающих ФВ на узле
03: VAR R                                  ⊳ коллекция ожидающих запросов на узле
04: VAR V                                  ⊳ коллекция ожидающих ответов на узле
05: PARAM r                                ⊳ номер текущего узла

06: ON EVENT при появлении запроса ⟨dfid, cfid⟩ ⊳ т.е. при создании или получении по сети
07:   LET n ← locate(dfid)                 ⊳ определить узел размещения запрашиваемого ФД
08:   IF n=r THEN                          ⊳ если узел совпадает с текущим
09:     IF dfid ∈ D THEN                   ⊳ если запрашиваемый ФД имеется на узле
10:       LET val ← D[dfid]                ⊳ получить значение ФД
11:       SPAWN ⟨⟨dfid, cfid⟩, val⟩        ⊳ создать ответ на запрос
12:     ELSE                               ⊳ иначе
13:       LET R ← R ∪ {⟨dfid, cfid⟩}       ⊳ поместить запрос в коллекцию ожидающих запросов
14:     END IF
15:   ELSE                                 ⊳ иначе (если n≠r)
16:     SEND ⟨dfid, cfid⟩ TO n             ⊳ переслать запрос на узел n
17:   END IF
18: END EVENT

19: ON EVENT при появлении ФД ⟨id, val⟩    ⊳ т.е. при создании или получении по сети
20:   LET n ← locate(id)                   ⊳ определить узел размещения запрашиваемого ФД
21:   IF n=r THEN                          ⊳ если узел совпадает с текущим
22:     FOR ⟨dfid, cfid⟩ ∈ R | dfid=df DO  ⊳ для каждого запроса на этот ФД
23:       SPAWN ⟨⟨dfid, cfid⟩, val⟩        ⊳ создать ответ на запрос
24:       LET R ← R \ {⟨dfid, cfid⟩}       ⊳ удалить запрос из R
25:     END FOR
26:   ELSE                                 ⊳ иначе (если n≠r)
27:     SEND ⟨id, valid⟩ TO n              ⊳ переслать ФД на узел n
28:   END IF
29: END EVENT

30: ON EVENT при появлении ответа ⟨⟨dfid, cfid⟩, val⟩
31:   LET n ← locate(cfid)                 ⊳ определить узел размещения ФВ
32:   IF n=r THEN                          ⊳ если узел совпадает с текущим
33:     IF cfid ∈ C THEN                   ⊳ если запрашиваемый ФВ имеется на узле
```

```
34:     CALL check_run_cf(C[cfid])          ⊳ попробовать исполнить ФВ
35:   ELSE                                  ⊳ иначе
36:     LET V ← V ∪ {⟨⟨dfid, cfid⟩, val⟩}   ⊳ поместить ответ в коллекцию ожидающих ответов
37:   END IF
38:  ELSE                                   ⊳ иначе (если n≠r)
39:   SEND ⟨⟨dfid, cfid⟩, val⟩ TO n         ⊳ переслать ответ на узел n
40:  END IF
41: END EVENT

42: ON EVENT при появлении ФВ ⟨id, o, c⟩    ⊳ т.е. при создании или получении по сети
43:  LET n ← locate(id)                     ⊳ определить узел размещения ФВ
44:  IF n=r THEN                           ⊳ если узел совпадает с текущим
45:   LET C ← C ∪ {⟨id, o, c⟩}             ⊳ поместить ФВ в коллекцию ожидающих ФВ
46:   CALL check_run_cf(cfid)              ⊳ попробовать исполнить ФВ
47:  ELSE                                  ⊳ иначе (если n≠r)
48:   SEND ⟨id, o, c⟩ TO n                  ⊳ переслать ФВ на узел n
49:  END IF
50: END EVENT

51: SUB check_run_cf(⟨id, o, c⟩)           ⊳ исполнить ФВ, если все входные ФД имеются на узле
52:   FOR dfid ∈ ins(o) DO                 ⊳ для всех входных ФД этого ФВ
53:     IF dfid ∉ V THEN
54:       RETURN                           ⊳ если ФД отсутствует, выйти из подпрограммы
55:     END IF
56:   END FOR
57:  EXEC ⟨cfid, o, c⟩                     ⊳ если дошли до этой точки, значит все ФД в наличии, исполнить ФВ
58:   LET C ← C \ {⟨id, o, c⟩}             ⊳ удалить ФВ из коллекции ожидающих
59:   FOR dfid ∈ ins(o) DO                 ⊳ для всех входных ФД этого ФВ
60:     LET V ← V \ {⟨dfid, cfid⟩}⊳ удалить ответ из коллекции ожидающих ответов
61:   END FOR
62: END SUB
```

В этом листинге опущено рассмотрение ситуации, когда значение локационной функции динамически изменяется. В этом случае все объекты коллекций C, D, R и V, для которых значение локационной функции изменилось, должны быть удалены из коллекций и переданы на другие узлы в соответствии с новыми значениями локационной функции. Реакция узлов на получение этих объектов по сети остаётся одной и той же вне зависимости от того, были они перемещены вследствие изменения локационной функции или по другой причине.

Этот алгоритм обеспечивает гарантированную доставку всех входных ФД для каждого ФВ вне зависимости от того, на каких узлах они располагаются и в каком порядке асинхронно происходят события выработки ФД, отправки запроса, прилёта запроса на узел и т.п. Работоспособность алгоритма сохраняется также в случаях, когда заранее неизвестно, скольким ФВ понадобится тот или иной ФД на вход, и когда конкретные индексы входных ФД вычисляются динамически.

Преимуществом алгоритма является его совместимость с динамической балансировкой нагрузки на узлы. А именно, если в результате динамического перераспределения ФД по узлам окажется, что запрошенный ФД был назначен на другой узел, то и ФД, и все запросы, относящиеся к этому ФД, будут пересылаться на новый узел в соответствии с локационной функцией, и поэтому окажутся на одном и том же узле. То же верно и при миграции ФВ: если за то время, пока запрошенный ФД доставлялся до запросившего ФВ этот ФВ мигрировал, то и ответ на запрос в соответствии с локационной функцией будет передаваться не на узел исходного расположения ФВ, а на узел текущего расположения.

Недостатком алгоритма является наличие дополнительных коммуникаций на запросы и на пересылку ФД на узел хранения. Но эти недостатки оказываются практически несущественными, если задано «хорошее» отображение ФВ и ФД на узлы, учитывающее структуру данных и вычислений ФА. «Хорошим» отображением является такое, где производство и потребление ФД осуществляется на одном и том же узле, либо (менее желательно) на соседних в сетевой топологии. В частности, если узел *Б* совпадает с узлом *В*, а узел *А* является соседним, то лишних коммуникаций не будет вообще. Такое распределение не всегда возможно, например, если ФД потребляется несколькими ФВ, находящимися на разных узлах. Но даже в этом случае не обязательно будет возникать дополнительное ожидание, связанное с отправкой запроса, если запрос будет отправлен заранее. Ввиду асинхронности исполнения ФА это часто (хотя и не всегда) возможно обеспечить для численных алгоритмов. Поэтому в худшем случае алгоритм гарантирует доставку ценой дополнительных издержек, а в лучшем случае гарантирует доставку и не влечёт дополнительных издержек.

Алгоритм может быть доработан возможностью задания более чем одного места хранения ФД (репликация ФД) для повышения доступности данных на мультикомпьютере, а также для сокращения задержек на доставку ФД к ФВ ценой дополнительного расхода памяти на вычислительных узлах и издержек на пересылку ФД на узлы хранения.

Алгоритм упреждающей посылки ФД. Идея этого алгоритма (листинг 2.6) состоит в том, чтобы при выработке ФД не отправлять его на узел хранения, а сразу направить копии всем его ФВ-потребителям. Для этого требуется статически определить список всех ФВ, потребляющих

этот ФД. ФВ, потребляющие этот ФД, в свою очередь, не отправляют на него запрос, а просто ожидают получения значения.

Листинг 2.6. Алгоритм упреждающей посылки ФД.

```
01: ON EVENT при завершении исполнения ФВ ⟨id, o, c⟩
02:   FOR ⟨dfid, val⟩ ∈ выходные ФД для ФВ ⟨id, o, c⟩ DO   ⊳ для всех выходных ФД данного ФВ
03:     FOR cfid ∈ ФВ, потребляющие ФД ⟨dfid⟩ DO          ⊳ для всех потребителей данного ФД
04:       SPAWN ⟨⟨dfid, cfid⟩, val⟩                       ⊳ породить ответ (без запроса)
05:     END FOR
06:     DELETE ⟨dfid, val⟩                              ⊳ удалить ФД
07:   END FOR
08: END EVENT
```

В строке 03 алгоритм полагается на то, что для заданного ФД можно определить список всех ФВ, для которых этот ФД является входным, что возможно не всегда. В строке 04 порождается ответ, но без запроса. Далее этот запрос обрабатывается так же, как если бы он был порождён в ответ на запрос (см. листинг 2.5). В строке 06 ФД удаляется, а не хранится на каком-либо узле. Остаются лишь его копии, направленные в ответах к ФВ, их потребляющим.

Преимуществом алгоритма является отсутствие дополнительных пересылок запроса от ФВ к месту хранения ФД, и ФД от места выработки до места хранения. Данные напрямую отправляются потребителям.

Недостатком алгоритма является то, что потребители выработанного ФД могут появиться ещё не скоро, но копии ФД для них будут созданы сразу, как только значение ФД будет вычислено. В результате это повлечёт перерасход памяти, особенно в случаях, когда у ФД много потребителей, которые будут порождены существенно позже, чем ФД будет выработан. Другим недостатком алгоритма является то, что не всегда статически возможно определить перечень всех ФВ, потребляющих тот или иной ФД. Это связано с тем, что множество потребителей может зависеть от входных данных задачи из-за условных операторов и операторов цикла, где количество итераций цикла не может быть вычислено статически, а также в случаях косвенной адресации (когда индекс в идентификаторе определяется значением ФД).

Таким образом, алгоритм упреждающей посылки ФД дополняет основной алгоритм доставки по запросу в случаях, когда он применим. В остальных случаях предпочтителен алгоритм доставки по запросу.

Обеспечение высокой эффективности доставки ФД является сложной проблемой, которая должна решаться различными частными и эвристическими способами в зависимости от особенностей приложения, поэтому в предлагаемую систему заложена возможность использовать различные алгоритмы доставки ФД, в том числе в рамках одной программы. Два из

них предложены в настоящем разделе. Доставка ФД может быть усовершенствована другими алгоритмами и техниками, отражающими особенности других предметных областей. Например, целесообразным представляется введение возможности полной репликации отдельных ФД по всем узлам (удобно для рассылки легковесных ФД, многократно потребляемых на всех или почти всех вычислительных узлах) и техники кэширования подмножества ФД (когда копия ФД не удаляется с узла сразу после использования, а некоторое время сохраняется в памяти узла на случай, если ФД понадобится на этом узле позже). Выбор подходящего алгоритма хранения и доставки ФД для того или иного ФД является отдельной проблемой, выходящей за рамки работы.

2.4.5 Алгоритмы сборки мусора

Задача сборки мусора при распределённой реализации ФА кратко может быть сформулирована следующим образом. После последнего потребления ФД он должен быть удалён, причём чем раньше, тем лучше. Соответственно, главной проблемой является обнаружение ситуации, когда ФД более не будет потреблён. В сравнении с традиционной постановкой задачи сборки мусора ситуация осложняется распределённостью вычислений, где информация о том, что происходит на других узлах доступна не напрямую на текущем узле, а лишь посредством передачи сообщений. При этом обнаружение ситуации возможности удаления ФД, вообще говоря, требует информации с других узлов мультикомпьютера. С другой стороны, сборка мусора проще, чем в традиционной постановке за счёт того, что невозможны «циклические зависимости» между ФД, т.к. на ФД могут ссылаться только ФВ, а не ФД.

Для решения этой проблемы предлагается три алгоритма, которые могут применяться совместно в рамках одной программы для различных (возможно, пересекающихся) подмножеств ФД.

Алгоритм подсчёта количества потреблений. Идея этого алгоритма (листинг 2.7) состоит в том, чтобы статически определить количество потреблений для ФД. Статически требуется определить не обязательно конкретное количество, это может быть и выражение, вычисляемое динамически. Это выражение фиксируется в виде рекомендаций. Для каждого ФД ведётся счётчик количества потреблений, и когда он доходит до заданного значения, то ФД уничтожается.

Листинг 2.7. Алгоритм удаления ФД на основе подсчёта количества потреблений.

```
01: ON EVENT при формировании ответа с ФД ⟨id, val⟩
02:   LET counter(id) ← counter(id)−1        ⊳ уменьшить счётчик потреблений на 1
03:   IF counter(id)=0 THEN                  ⊳ если счётчик достиг нуля
```

```
04:    DELETE ⟨dfid, val⟩                ⊳ то удалить ФД
05:  END IF
06: END EVENT
```

Количество потреблений вычисляется в момент вычисления ФД и хранится (и мигрирует) вместе с ФД в виде счётчика. Каждый раз, когда для данного ФД формируется ответ на запрос, то счётчик декрементируется, а при достижении счётчиком нуля ФД уничтожается.

Задача вывода выражения, определяющего количество потреблений, является типичной задачей статического анализа и в работе не рассматривается. Отметим лишь, что в общем случае эта задача является алгоритмически неразрешимой[3] в силу того, что количество потреблений может определяться количеством итераций цикла с предусловием. Как следствие, этот алгоритм сборки мусора не может быть использован как единственный, но ввиду того, что он практически не требует накладных расходов во время исполнения его следует использовать во всех случаях, когда это возможно.

Директивная сборка мусора. В данном алгоритме путём статического анализа или вручную определяется некоторое легко фиксируемое событие, которое гарантированно произойдёт позже, чем состоится последнее потребление ФД, после чего к данному событию прикрепляется рекомендация на удаление ФД при наступлении этого события.

Поясним на примере. Допустим, некоторый ФД x имеет множество потребителей, но в силу информационных зависимостей ясно, что некоторый ФВ a будет исполнен после того, как все потребления ФД x уже произойдут. Например, ФВ a обрабатывает результат подпрограммы[4], где используется ФД x, а более этот ФД нигде не используется. К описанию ФВ a добавляется рекомендация на удаление ФД x. Когда исполнительная система будет осуществлять исполнение ФВ a, то, в соответствии с рекомендацией, будет отдана команда на удаление ФД x.

Главная сложность при директивной сборке мусора состоит в том, чтобы определить событие, гарантированно происходящее после последнего потребления ФВ, и при этом во времени находящееся, по возможности, как можно ближе к этому самому последнему потреблению. Эта задача также является типичной задачей статического анализа и в работе не рассматривается. Аналогично предыдущему алгоритму отметим, что эта задача, хотя и является алгоритмически разрешимой, но на практике есть надежда автоматически решить её за разумное время лишь в частных случаях. Действительно, в качестве события, гарантированно

---

[3] Алгоритмическая неразрешимость следует из того, что к этой проблеме можно свести классическую проблему останова, которая является алгоритмически неразрешимой.

[4] Понятие подпрограммы применительно к ФА вводится в главе 3, но в данном случае достаточно обычного представления о подпрограммах.

находящегося после последнего потребления мы можем взять событие завершения программы. Но практической пользы в этом мало. Также отметим, что некоторые подходящие события могут быть установлены статически, но практически динамическое обнаружение таких событий будет затруднительно. В качестве примера приведём случай потребления ФД известным множеством ФВ. В качестве события естественно взять событие окончания выполнения последнего ФВ из этого множества. Но установить факт завершения выполнения последнего ФВ из заданного множества в распределённой системе, вообще говоря, невозможно без дополнительных накладных расходов, которые могут оказаться нецелесообразно большими. Следующий алгоритм сборки мусора по завершению области видимости раскрывает этот вопрос более детально.

Алгоритм сборки мусора по завершению области видимости. Этот алгоритм предназначен для случаев, когда все потребления ФД ограничиваются некоторой областью видимости. Под областью видимости понимается тело оператора цикла, т.е. все потребления некоторого ФД ограничены конкретным оператором цикла. Также этот алгоритм может быть естественным образом расширен на понятие подпрограмм и локальных объявлений ФД, которые будут введены в 3 главе. Как следствие, если все ФВ, принадлежащие телу этого оператора, исполнены, то и ФД уже не может быть запрошен, и его можно удалять. Этот алгоритм является распределённым аналогом освобождения автоматической памяти по завершению блока программы в языках C/C++. Важное отличие состоит в том, что динамическое обнаружение факта завершения всех таких ФВ, асинхронно исполняющихся, вообще говоря, на разных узлах мультикомпьютера, особенно в условиях динамической балансировки нагрузки, является проблемой. Предлагаемый алгоритм решает эту проблему ценой существенных накладных расходов (листинг 2.8).

Листинг 2.8. Алгоритм сборки мусора на основе завершения области видимости ФД.

```
01: VAR T                    ⊳ вспомогательное дерево счётчиков
02: VAR D                    ⊳ коллекция имён ФД
03: PARAM r                  ⊳ номер текущего узла

04: ON EVENT при завершении исполнения ФВ ⟨id, o, c⟩
05:   IF o — блочный THEN                          ⊳ если ФВ блочный
06:     LET c ← количество операторов в теле o     ⊳ записать количество дочерних операторов
07:     LET T ← T ∪ {⟨id, c, parent(id)⟩}          ⊳ в структуру T в виде счётчика
08:     FOR cf ∈ ФВ, порождённые оператором o DO    ⊳ для всех дочерних ФВ
09:       MARK parent(cf) ← ⟨r, id⟩                ⊳ запомнить узел и id родительского ФВ
10:     END FOR
```

```
11:  ELSE                                  ▷ иначе (ФВ не блочный)
12:    LET ⟨n, id⟩ ← parent(id)            ▷ на узел исполнения родительского ФВ
13:    SEND декремент id TO n              ▷ послать сообщение о декременте счётчика
14:  END IF
15: END EVENT

16: ON EVENT создан ФД ⟨dfid, val⟩ фрагментом cfid
17:  LET ⟨n, id⟩ ← parent(cfid)            ▷ по записи о родительском ФВ
18:  SEND записать ФД ⟨dfid, id⟩ TO n       ▷ отправить на узел родительского ФВ запись о созданном ФД
19: END EVENT

20: ON MESSAGE записать ФД ⟨dfid, cfid⟩    ▷ при получении сообщения о записи ФД
21:  LET D ← D ∪ {⟨cfid, dfid⟩}             ▷ записать dfid в коллекции D с привязкой к cfid
22: END MESSAGE

23: ON MESSAGE декремент id                ▷ при получении сообщения о декременте счётчика в T по id
24:  LET ⟨id, c, parent⟩ ← T[id]
25:  LET c ← c−1                           ▷ декрементировать счётчик (незавершённых дочерних ФВ)
26:  IF c=0 THEN                           ▷ если все дочерние ФВ исполнены, то
27:    LET ⟨n, parent_id⟩ ← parent
28:    SEND декремент parent_id TO n       ▷ декрементируется родительский счётчик
29:    FOR ⟨cfid, dfid⟩ ∈ D | cfid=id DO   ▷ все ФД, привязанные к данному ФВ
30:      IF dfid объявлен в ФВ cfid THEN   ▷ либо удаляются, если они были объявлены
31:        DELETE dfid                     ▷ в пространстве имён ФВ cfid
32:        LET D ← D \ {⟨cfid, dfid⟩}      ▷ (и удаляется из D)
33:      ELSE                              ▷ либо переприкрепляются
34:        SEND записать ФД ⟨dfid, parent_id⟩ TO n ▷ к родительскому ФВ
35:      END IF
36:    END FOR
37:    LET T ← T \ {⟨id, c, parent⟩}       ▷ а текущая запись со счётчиком удаляется
38:  END IF
39: END MESSAGE
```

Информация о счётчике родительского ФВ (строка 09) сохраняется вместе с ФВ, и вместе с ним мигрирует по другим узлам, чтобы по завершению этого ФВ он мог переслать на узел с этим счётчиком сообщение о декременте (строки 13 и 28). Сообщения о декременте (строки 13 и 28) и о порождённом ФД (строки 18 и 34) могут выполняться и локально, если n=r. Это непринципиальное обстоятельство было выпущено из листинга из соображений ясности изложения. Также не учитывался случай, когда родительской записи нет. В этом случае

соответствующий декремент не посылается (строка 28). Удаление ФД по его идентификатору (строка 31) осуществляется, вообще говоря, удалённо, т.к. этот ФД может храниться не на том узле, где было обнаружено, что его можно удалять. Удаление ФД на другом узле не рассматривается отдельно, оно осуществляется тем же способом, что и поиск ФД, но запрос на удаление приводит не к формированию ответа на запрос, а к удалению ФД.

По сути, алгоритм формирует распределённое дерево исполнения вложенных блочных ФВ, где в каждом узле дерева хранится список ФД, выработанных текущим блоком. Когда лист дерева помечается как выполненный, то все ФД, объявленные в текущем блоке, удаляются, а остальные — поднимаются вверх по дереву.

Этот алгоритм гарантирует удаление всех ФД после завершения выполнения блоков, в которых они объявлены, но имеет ряд существенных недостатков. *Во-первых,* это накладные расходы памяти на хранение системных записей. Размер этих записей может неограниченно расти для больших множеств ФД. С учётом того, что в системных записях хранятся лишь ссылки, а сами значения ФД, как правило, будут гораздо объёмнее, эти записи будут занимать малую долю памяти, но ввиду того, что они аккумулируются на одном узле (для каждого блока), то память одного узла может переполниться раньше, чем общая память мультикомпьютера переполнится основными данными (значениями ФД). *Во-вторых,* распределённая структура дерева системных записей означает, что декременты счётчиков и передача ссылок в родительские системные записи влечёт дополнительную нагрузку на сеть. Хотя объёмы таких сообщений будут, в основном, небольшими, но их количество может быть большим. *В-третьих,* запросы на удаление ФД отправляются по одному на каждый ФД. Удаление всего «массива[5]» ФД со всего мультикомпьютера не является простой операцией т.к. в общем случае неизвестно сколько элементов и с какими индексами имеет тот или иной «массив». Это означает кратковременную, но интенсивную нагрузку на сеть в моменты удаления больших множеств ФД из-за большого количества посылаемых запросов. *В-четвёртых,* недостатком алгоритма также является то, что он практически не применим для итерационных процессов, когда множество ФД индексируется, в том числе, по номеру итерации или шага по времени. В этом случае на протяжении исполнения всего цикла, который может занимать подавляющее большинство времени выполнения программы, всё множество ФД не будет удалено сборщиком мусора до самого конца выполнения блочного оператора.

Несмотря на все эти важнейшие недостатки алгоритм всё же обеспечивает сборку мусора достаточно хорошо, чтобы поддерживать выполнение широкого класса ФА, исполнение которых

[5] Собственно понятия «массив» в ФА нет, в данном случае имеется ввиду множество всех ФД, чьи идентификаторы различаются только значениями индексов.

без сборки мусора сразу переполняет память мультикомпьютера. Главным же преимуществом этого алгоритма является его полная автоматизация. Даже в отсутствие алгоритмов статического анализа и рекомендаций, заданных человеком, этот алгоритм сборки мусора отрабатывает на любом ФА, с бóльшим или меньшим успехом. Ввиду больши́х накладных расходов использования этого алгоритма следует избегать в пользу двух предыдущих.

2.4.6 Оптимизация исполнения ФА на основе профилирования

Профилирование — это сбор информации о характеристиках исполнения программы и её частей в процессе её исполнения на конкретном вычислителе и конкретных входных данных. Профилирование используется как источник дополнительной информации о программе и особенностях её выполнения в конкретных условиях для применения специфичных оптимизаций программы и её настройки на эти условия. При исполнении ФА профилирование может применяться для конструирования более эффективных программ. Следующий алгоритм был разработан для повышения эффективности конструируемых из ФА программ путём изменения отображения множества ФВ и ФД на вычислительные узлы на основе фиксации информации о простоях вычислительных узлов.

Основная идея алгоритма состоит в том, что при исполнении ФА фиксируются времена начала и окончания исполнения каждого ФВ. Далее на основе этой информации строится функция зависимости загрузки каждого узла от времени. Далее эта функция анализируется на предмет выявления периодов с выраженным дисбалансом вычислительной нагрузки, после чего часть ФВ переназначается с перегруженных узлов на недогруженный путём добавления (или изменения существующих) рекомендаций, задающих отображение ФВ на вычислительные узлы. ФА с модифицированными рекомендациями может быть проведён через такую процедуру многократно, постепенно приводя время выполнения конструируемой программы в окрестность некоторого локального оптимума.

Входом для алгоритма является последовательность ФВ в порядке их исполнения (P). Для каждого ФВ определены моменты времени начала и окончания выполнения, а также узел, на котором ФВ выполнялся. В алгоритме используется функция load определения нагрузки на узел в заданный момент времени, которая задаётся как количество ФВ, которые выполняются в данный момент на узле, т.е. $load(t, r) = |\{c \in P \mid T_{start}(c) \leq t < T_{end}(c)\}|$. Выходом алгоритма является модифицированный словарь rank, в котором хранится отображение множества ФВ на узлы. Алгоритм представлен на листинге 2.9.

Листинг 2.9. Алгоритм коррекции рекомендаций на основе профилирования.

```
01: PARAM Θ=2                          ⊳ порог дисбаланса
00: FOR c ∈ P DO                       ⊳ цикл по всем ФВ в порядке их начала выполнения
01:   SET t ← T_start(c)               ⊳ пусть t — время начала выполнения c
02:   SET r ← rank(c)                  ⊳ пусть r — узел выполнения c
03:   SET m ← min_n load(t, n)         ⊳ выбрать наименее загруженный узел в момент t
04:   IF load(t, r) ≥ load(t, m)+Θ THEN ⊳ если превышен порог дисбаланса Θ между r и m
05:     SET rank(c) ← m                ⊳ установить новый номер узла для c в m
06:   END IF
07: END FOR
```

Преимуществом алгоритма является то, что он улучшает распределение ФВ по вычислительным узлам в случаях выраженного дисбаланса для отдельных ФВ («точечный», а не массовый дисбаланс). Также его преимуществом является полностью автоматическая работа и независимость от других системных алгоритмов (он работает со сложившимся по факту распределением ФВ по узлам вне зависимости от того как это распределение сложилось).

Недостатком является то, что алгоритм игнорирует информационные зависимости, что может приводить к возникновению «дальних» коммуникаций (в смысле сетевой топологии мультикомпьютера), поэтому область его применения ограничена приложениями, критичными по вычислениям и с массовым параллелизмом.

Алгоритм имеет очевидные пути улучшения, например, учёт времени вычисления ФВ, учёт распределения ФД, учёт информационных зависимостей и т.п., но это требует дополнительных исследований и выходит за рамки настоящей работы. В данном случае алгоритм демонстрирует принципиальную возможность применения профилировочной информации для оптимизации исполнения ФА.

2.4.7 Алгоритм обнаружения завершения работы системы

ФВ распределены по узлам мультикомпьютера и асинхронно выполняются, иногда порождая новые ФВ. В определённый момент последний выполняющийся ФВ оказывается исполненным, после чего ИС должна прекратить свою работу. Обнаружение этой ситуации в распределённой системе децентрализованным образом представляет отдельную проблему, которая решается предлагаемым алгоритмом. Эта проблема, известная в англоязычной литературе как termination detection in distributed system, является хорошо разработанной и имеет множество известных решений. Предлагаемый алгоритм является адаптацией алгоритма Дейкстры—Шольтена [82] к конкретной ситуации. Алгоритм подразумевает, что все узлы мультикомпьютера логически организованы в кольцо (желательно, но не обязательно, чтобы

узлы, соседние в кольце, были соседями и в сетевой топологии). В алгоритме используется объект-счётчик, который располагается в каждый момент времени на одном из вычислительных узлов и может передаваться по сети. Также в алгоритме подразумевается, что каждый вычислительный узел может находиться в одном из двух состояний — «есть работа» и «нет работы». Первое означает, что на узле имеется как минимум один ФВ, либо как минимум одно исходящее сообщение, которое ещё не доставлено.

В листинге 2.10 Считается, что на одном из узлов помещён счётчик со значением 0. Специальное значение счётчика **nil** обозначает, что счётчик находится на другом узле.

Листинг 2.10. Алгоритм обнаружения остановки системы.

```
01: VAR dirty     ⊳ флаг о наличии работы с момента последнего прохода счётчика
02: VAR idle      ⊳ признак отсутствия работы на узле
03: VAR counter ⊳ счётчик (равен nil если не на текущем узле)
04: PARAM s       ⊳ количество узлов в системе

05: ON EVENT idle ∧ counter≠nil ∧ ¬dirty ⊳ если на узле со счётчиком нет работы
06:   LET counter ← counter+1              ⊳ инкрементировать счётчик
07:   IF counter=s×2 THEN                  ⊳ если счётчик достиг порогового значения
08:     STOP                               ⊳ остановить систему
09:   ELSE                                 ⊳ иначе
10:     SEND counter TO следующий узел     ⊳ переслать счётчик на следующий узел в кольце
11:   END IF
12:   LET dirty ← FALSE                          ⊳ снять флаг
13: END EVENT

14: ON EVENT счётчик получен по сети
15:   IF dirty THEN                  ⊳ если работа есть или была с момента прошлого прохода
16:     LET counter ← 0              ⊳ обнулить счётчик
17:   END IF
18:   IF idle THEN
19:     LET dirty ← FALSE            ⊳ снять флаг; это может привести срабатыванию события на строке 05
20:   END IF
21: END EVENT

22: ON EVENT ¬idle ∧ ¬dirty      ⊳ если на узле появилась работа
23:   LET dirty ← TRUE           ⊳ установить флаг
24: END EVENT
```

Идея этого алгоритма в том, что счётчик движется по кругу по всем узлам, но лишь в том случае, если на них нет работы. Счётчик сбрасывается, если работа на узле есть, либо была с момента прошлого прохода счётчика (флаг dirty). Соответственно, если счётчик сделал два полных круга без сброса счётчика, значит ни на одном узле уже нет работы. Докажем это:

Пусть значение счётчика достигло значения в N, где N — число узлов в кольце. Это значит, что счётчик сделал полный круг, не встретив ни работ, ни отметок о том, что работы там были. В этот момент возможны две ситуации:

1. Ещё где-то на мультикомпьютере работа есть, и тогда как минимум на одном узле будет отсутствовать отметка об отсутствии работы. В этом случае в одном из N следующих переходов счётчик будет сброшен.
2. Уже нигде на мультикомпьютере нет работы. В этом случае счётчик достигнет значение в 2N не позднее, чем через 3N шагов. Действительно, даже если все остальные узлы в этот момент времени не имеют отметки об остутствии работы, эта отметка будет установлена на все узлы за N шагов, после чего значение счётчика уже не будет сбрасываться, а только расти и достигнет значения в 2N.

Таким образом, значение счётчика достигнет значения в 2N в том, и только в том случае, если на мультикомпьютере больше нет работы — что и требовалось доказать.

Преимуществом алгоритма является крайне малое количество накладных расходов (одна булева переменная на узел и одна целочисленная переменная на счётчик, плюс передача максимум одного сообщения одновременно размером в несколько байт). Более того, в случае загруженного мультикомпьютера алгоритм вообще бездействует, активизируясь только при освобождении как минимум того узла, где находится счётчик, поэтому та нагрузка, которую алгоритм всё же даёт, приходится на свободные ресурсы, не отнимая, тем самым, ресурсы от полезных вычислений. Также к преимуществам алгоритма можно отнести его универсальность — он способен работать в любой системе, где есть понятия наличия и отсутствия работы, не обязательно при исполнени ФА. Этот алгоритм может также быть применён для обнаружения зависаний в системе (в случае ошибки в ФА). Для этого достаточно не считать за нагрузку те ФВ, которые ожидают своих входных ФД.

Недостатком алгоритма является задержка обнаружения окончания выполнения программы. С момента завершения последнего ФВ потребуется от 2 до 3 полных кругов счетчика по мультикомпьютеру. Этот недостаток не является существенным, т.к. сколько-нибудь заметное время задержка будет составлять лишь на очень больших мультикомпьютерах, но на таких мультикомпьютерах и время реального расчёта должно быть несравнимо бóльшим, чем прогон легковесного счётчика по кругу.

2.4.8 Воспроизведение трасс

Под трассировкой понимается исполнение ФА, при котором сохраняется информация обо всех событиях, произошедших с фрагментами: где и когда исполнялись ФВ, на каких узлах в какие моменты времени находились ФД, когда ФВ или ФД мигрировали и с какого узла на какой и т.п. Сохранённая информация называется трассой. Трасса содержит информацию о ходе исполнения ФА достаточно полную для того, чтобы осуществить на её основе повторный запуск того же самого ФА на том же вычислителе, т.е. воспроизвести трассу. Главным преимуществом воспроизведения трассы по сравнению с нормальным исполнением ФА является отсутствие большей части накладных расходов, связанных с динамическим принятием решений.

Минимальная необходимая и достаточная информация для воспроизведения трассы — это информация о времени и номере узла двух видов событий — это начало и окончание исполнения ФВ. Другая информация также может быть полезна при оценке нефункциональных свойств трассы, но в настоящей работе это рассматриваться не будет.

Ниже (листинг 2.11) представлен алгоритм исполнения трассы (для каждого вычислительного узла). Под очередью ФВ в листинге понимается множество ФВ, исполнявшихся на вычислительном узле, линейно упорядоченное по времени начала выполнения ФВ.

Листинг 2.11. Алгоритм воспроизведения трассы на узле.

```
00:  PARAM r                    ⊳ номер текущего вычислительного узла
01: PARAM T_r                   ⊳ очередь ФВ в трассе на текущем узле
02:  WHILE ¬empty(T_r)          ⊳ пока очередь T_r не пуста
03:    SET c ← T_r              ⊳ извлечь следующий ФВ
04:    FOR d ∈ inputs(c) DO     ⊳ для каждого входного ФД d
05:      IF prod_rank(d)=r THEN ⊳ если ФД порождается на текущем узле
06:        WAIT computed(d)     ⊳ дождаться пока значение ФД d будет вычислено
07:      ELSE
08:        RECV d               ⊳ дождаться получения значения ФД d по сети
09:      END IF
10:    END FOR
11:    EXEC c                   ⊳ исполнить ФВ c
12:    FOR d ∈ outputs(c) DO    ⊳ для каждого выходного ФД d
13:      FOR e ∈ consumers(d) DO⊳ для каждого потребителя ФД d
14:        IF rank(e)≠r THEN    ⊳ если потребитель ФД e на другом узле
15:          SEND d TO rank(e)  ⊳ отправить копию ФД e на узел потребления
16:        END IF
17:      END FOR
```

17: **END FOR**
18: **END WHILE**

Для повышения эффективности воспроизведения трассы она может быть проанализирована и модифицирована. Например, по трассе возможно выявить периоды дисбаланса нагрузки, когда одни узлы были перегружены, а другие — недогружены. В этом случае трасса может быть модифицирована так, чтобы часть ФВ была переназначена с перегруженных узлов на недогруженные.

Главным недостатком воспроизведения трасс является то, что множество ФВ может зависеть от входных данных. Например, если ФА реализует некоторый итерационный процесс, то количество итераций может зависеть от входных данных. Соответственно, воспроизведение трассы на других данных может привести к ошибке. Этот недостаток частично компенсируется двумя обстоятельствами. Во-первых, для многих классов прикладных алгоритмов структура вычислений может не зависеть от входных данных. Например, многие матрично-векторные алгоритмы на плотных матрицах обладают этим свойством. Во-вторых, если множество ФВ зависит от входных данных, то несоответствие множества ФВ в трассе и при нормальном исполнении ФВ может быть обнаружено при воспроизведении трассы. В частности, в ФА есть только два оператора, которые могут приводить к зависимости множества ФВ от данных — это операторы циклов. Также множество ФВ может зависеть от входных данных при косвенной адресации (т.е. когда индекс в идентификаторе задаётся ФД). При воспроизведении трассы нет проблемы в том, чтобы сверить значения соответствующих ФД (параметров циклов и значений ФД в индексах идентификаторов), и если они не совпадают, прервать воспроизведение с соответствующим уведомлением пользователя или переключить исполнение ФА на базовый алгоритм.

Также можно сформулировать задачу «свёртки» участков трасс, порождённых операторами циклов, в параметрическую форму, не зависящую от количества итераций. В этом случае трасса может быть динамически адаптирована под конкретные диапазоны циклов. Решение этой задачи возможно по крайней мере в частных случаях, но её рассмотрение выходит за рамки настоящей работы.

Другим недостатком является отсутствие адаптивности при воспроизведении трассы, например, отсутствие динамической балансировки нагрузки на вычислительные узлы. Подобная функциональность, в принципе, может быть встроена в алгоритм воспроизведения трассы, хотя и ценой дополнительных накладных расходов во время исполнения.

2.5 Анализ системных алгоритмов

Такого рода система автоматического конструирования параллельных программ заведомо не может быть универсальной, т.к. в общей постановке эффективная реализация ФА на мультикомпьютере является алгоритмически труднорешаемой проблемой (принадлежит классу NP, если требовать оптимальной по времени программы). Поэтому система должна быть специализированной на конкретный класс приложений и вычислителей и обеспечивать приемлемую эффективность конструируемых программ за счёт использования частных и эвристических системных алгоритмов.

На практике система может применяться, если в конкретной предметной области она обеспечивает разумный компромисс между простотой использования и эффективностью конструируемой программы. В перспективе, по мере накопления достаточного «багажа» системных алгоритмов эффективность конструируемых программ должна стать выше, чем у обычных параллельных программ, разработанных вручную среднестатистическим прикладным параллельным программистом. Аналогичный переход уже произошёл в последовательных компиляторах с процедурных языков, вытеснив ручное программирование на ассемблере как массовую профессию.

Модель ФА может служить основой для такого накопления системных алгоритмов в области автоматического конструирования параллельных программ. Также существенно, что имеющийся текущий недостаток хороших частных и эвристических системных алгоритмов на практике может быть частично скомпенсирован применением рекомендаций, когда программист дополняет ФА подсказками, упрощающими для системы задачу конструирования эффективной параллельной программы до подъёмной. Применение рекомендаций не требует низкоуровневого параллельного программирования, а изменение рекомендаций не требует повторной функциональной отладки конструируемой программы, т.к. не влияет на её функциональные свойства.

Таким образом, ценность предлагаемого решения не только в конкретных системных алгоритмах трансляции и исполнения, но и в обеспеченной возможности такие алгоритмы аккумулировать в будущем.

3. Система LuNA

В главе представлена реализация предложенных в главе 2 модели ФА и системных алгоритмов в виде языка описания фрагментированных алгоритмов LuNA и одноимённой экспериментальной системы автоматического конструирования параллельных программ по описанию на этом языке. Аббревиатура **LuNA** означает **L**ang**u**age for **N**umerical **A**lgorithms (язык для численных алгоритмов).

В целом глава 2 решает основные проблемы, возникающие при создании требуемой системы, но ряд реализационных вопросов требует отдельного рассмотрения. Этому и посвящена настоящая глава.

Глава организована следующим образом. В разделе 3.1 представлен язык LuNA, в том числе его внутреннее исполняемое представление, используемое в исполнительной системе (ИС). В разделе 3.2 рассматриваются вопросы организации файловой структуры на различных этапах трансляции и исполнения LuNA-программ. В разделе 3.3 рассматриваются вопросы компонентной организации самой системы LuNA (транслятора и исполнительной системы). В разделе 3.4 рассматриваются проблемные места реализации системы и их решения. Завершает главу раздел 3.5, обсуждающий предлагаемую систему LuNA в целом.

Во избежание терминологической путаницы в тексте ниже собственно транслятором будет называться транслятор, осуществляющий преобразование программ с одного языка в другой, а компилятором будем называть транслятор, результатом работы которого является машинный код. Общая архитектура системы представлена на рисунке 3.1.

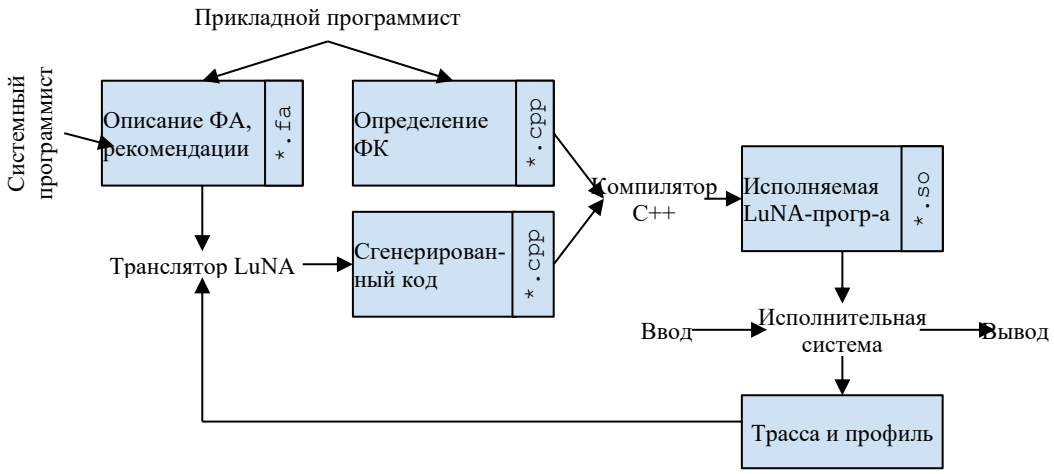

Рисунок 3.1. Архитектура системы LuNA.

3.1 Язык LuNA

Для практического применения модели ФА на компьютере необходим язык описания ФА, приспособленный для использования и компьютером, и человеком. Язык должен принципиально позволять описывать произвольный ФА и быть удобным для представления ФА в некотором практически значимом классе приложений. Последнее означает, что в языке необходимы средства модульной декомпозиции, возможность взаимодействия исполняющегося ФА с внешним миром (interoperability), а также возможность переиспользования накопленного последовательного кода для реализации ФК. Язык должен иметь синтаксис и грамматику, допускающие разбор известными алгоритмами (и инструментами). По возможности, язык должен быть прост и практичен. Из соображений переносимости язык должен иметь текстовое представление.

В соответствии с этими требованиями был разработан язык, который получил название LuNA — Language for Numerical Algorithms (язык для численных алгоритмов). Название подчёркивает предметную область, на которую ориентирована предлагаемая модель ФА и сам язык LuNA — на реализацию алгоритмов численного моделирования.

Собственно исходным кодом, программой для системы LuNA считается набор файлов двух видов: один и более файл на языке LuNA (LuNA-программа), описывающий ФА, а также ноль и более исходных файлов на языке C++, описывающих фрагменты кода. Файлов второго вида может не быть, т.к. в синтаксисе языка LuNA имеется возможность определять фрагменты кода прямо в листинге LuNA-программы.

3.1.1 Базовый синтаксис

В разделе неформально, на примерах представлены основные средства языка LuNA, напрямую соответствующие модели ФА. Ввиду близкого соответствия понятийного аппарата языка LuNA модели ФА такое описание представляется достаточным. В язык LuNA также были добавлены разнообразные вспомогательные средства («синтаксический сахар»), не влияющие на принципиальные свойства языка, но поднимающие удобство его использования до практически пригодного. Эти средства неформально описаны в приложении В. Полная грамматика языка в БНФ представлена в приложении Г.

LuNA-программа — описание ФА на языке LuNA — представляет собой текстовый файл, содержащий описание ФК, операторов и множества имён. Эти описания приводятся одно за другим, причём порядок описания значения не имеет, а описанные операторы, ФК и имена

рассматриваются как множество. Символы пробелов, табуляций и переноса строк считаются разделителями лексем. Каждый оператор, описание имён и ФК заканчивается символом точки с запятой кроме **блочных операторов** — операторов арифметического цикла и цикла с предусловием.

Фрагмент кода описывается следующим образом:

import proc_name(**value**, **name**) **as** code_name;

Примечание — Тут и далее терминальные символы выделены полужирным шрифтом.

Тут после терминального символа **import** следует нетерминал proc_name, который задаёт идентификатор последовательной процедуры, реализующей ФК. Далее в скобках описывается список параметров через запятую, каждый из которых является либо входным, либо выходным, что задаётся, соответственно, терминалами **value** и **name**. Количество параметров может быть любым конечным (0 и более). Далее, после терминала **as** следует нетерминал, определяющий имя ФК. Это имя будет использоваться в LuNA-программе для обозначения этого ФК, в то время как proc_name будет идентифицировать внешнюю последовательную процедуру, реализующую ФК. Идентификатор proc_name будет означен в процессе линковки программы, например, это будет процедура из традиционной статической или динамической библиотеки, написанной на языке C++ или Fortran. Отметим, что, как правило, LuNA-программа снабжается исходными кодами таких процедур, либо объектными или бинарными файлами, содержащими определения таких процедур. Какой интерфейс должна иметь процедура в зависимости от описания ФК — является предметом соглашения, которое выходит за рамки спецификации языка LuNA. Примером объявления процедуры на C++ для примера выше может служить следующее:

void proc_name(**const** InputDF &x, OutputDF &y);

Где InputDF и OutputDF — некоторые интерфейсные классы, которые обеспечивают доступ процедуры к входным и выходным параметрам ФК.

Множество имён описывается следующим образом:

df x, N, some_df;

Тут за терминалом **df** следует перечень имён через запятую. Все имена, используемые в ссылках, должны быть перечислены таким образом.

Ссылка — это слово следующего вида:

x[a][b[c]]

Тут за нетерминалом, задающим именную часть ссылки, следует ноль или более индексных выражений в квадратных скобках, каждое из которых само является ссылкой.

Оператор исполнения описывается следующим образом:

oper(x, y[N], z[y[N]]);

Тут за нетерминалом oper, идентифицирующим ФК, следует перечень ссылок, задающих аргументы описываемого ФВ в порядке, соответствующем порядку параметров ФК в его определении.

Оператор арифметического цикла описывается следующим образом:

for i=x..y {...}

Тут за терминалом **for** следует нетерминал i, определяющий имя индексной переменной, x и y — ссылки, определяющие первое и последнее значения индексной переменной, а фигурные скобки содержат описание множества операторов, составляющих тело цикла.

Оператор цикла с предусловием описывается следующим образом:

while y[i] i=x .. **out** N {...}

Тут за терминалом **while** следует ссылка на предусловие, после чего нетерминалом задаётся индексная переменная и начальное значение индексной переменной. Далее следует терминал **out**, за которым следует ссылка на выходной ФД, куда будет записано первое значение индексной переменной, для которого условие оказалось ложным. Завершает описание блок — множество операторов, составляющих тело цикла и обрамлённых в фигурные скобки.

Этим базовые средства исчерпываются.

В качестве иллюстрации рассмотрим пример описания ФА для вычисления числа Фибоначчи с заданным номером n (листинг 3.1), который приведён в разделе 2.3.1.

Листинг 3.1. ФА для вычисления числа Фибоначчи.

```
01: import set_0(name) as zero;
02: import set_1(name) as one;
03: import sum(value,value,name) as sum;
04: import set_n_minus_2(name) as init;
05: df z, o, F, n, J, K;
06: zero(z);
07: one(o);
08: one(F[z]);
09: one(F[o]);
10: init(n);
11: for i=z..n {
12:   sum(i, o, J[i]);
13:   sum(J[i], o, K[i]);
14:   sum(F[i], F[J[i]], F[K[i]]);
15: }
```

В этом примере ФК zero и one вычисляют значение единственного выходного параметра в 0 и 1 соответственно, sum суммирует два первых параметра в третий, а init задаёт значение

выходному параметру в n минус 2, где n — номер искомого числа Фибоначчи. В результате исполнения ФА искомое число будет значением соответствующего ФД F[n].

Приведём в пример (листинг 3.2) тот же ФА, описанный с использованием расширенного синтаксиса (приложение А) языка LuNA.

Листинг 3.2. ФА для вычисления числа Фибоначчи в расширенном синтаксисе.

```
01: import init(int expr, name val);
02: sub main() {
03:   df F, n;
04:   init(F[0], 1);
05:   init(F[1], 1);
06:   for i=2..10 init(F[i-1]+F[i-2], F[i]);
07: }
```

3.1.2 Исполняемое представление фрагментированного алгоритма

Исходя из требований к исполняемому представлению (раздел 2.2), оно должно быть, по-возможности, императивным, но сохранять при этом фрагментированную структуру, позволяющую обеспечивать динамические свойства исполнения программы со стороны исполнительной системы. В ФА естественным образом можно выделить декларативную и императивную части — это множество ФВ, явно присутствующее во время исполнения (декларативная часть), где каждый ФВ реализуется императивно. С учётом требований распределённости и децентрализованности подходящим решением является применение мультиагентного подхода, где каждый ФВ реализуется одним агентом. Так фрагментированная структура выражена распределённым по узлам множеством взаимодействующих агентов, каждый из которых управляется императивной программой. Конкретизуем мультиагентную модель.

Сначала рассмотрим предлагаемое исполняемое представление неформально, на уровне ключевых идей, а затем предъявим конкретную спецификацию агентов и среды их существования через описание интерфейса их взаимодействия со своим окружением.

Ключевая идея состоит в том, что имеется множество вычислительных узлов мультикомпьютера, составляющих среду существования агентов. Эта среда является вместилищем для объектов двух видов — собственно агентов, реализующих ФВ, и объектов данных, реализующих ФД (далее просто будем говорить об объектах данных как о ФД, распределённых по узлам). И агенты, и ФД существуют в каждый момент времени на конкретном узле, но могут мигрировать с узла на узел. Можно сказать, что среда является, с одной стороны,

распределённой базой данных, хранящей ФД, а с другой стороны — распределённым портфелем задач, где «задачей» является агент. В каждый момент времени мультиагентная система как целое соответствует конкретному состоянию (в смысле опр. 13) исполнения ФА, где агент соответствует ФВ, объект данных соответствует ФД, а эволюция мультиагентной системы либо оставляет её в соответствии с текущим состоянием, либо переводит её в состояние, соответствующее новому состоянию исполнения ФА в соответствии с опр. 18.

Целью каждого агента является реализация соответствующего ему ФВ, что выражается в том, что агент дожидается появления в системе всех входных ФД соответствующего ему ФВ, сохраняя в своей внутренней локальной памяти копии этих ФД, после чего фиксирует своё расположение на одном из узлов среды и исполняется, т.е. реализует исполнение соответствующего ему ФВ. Как именно реализуется исполнение конкретного ФВ в зависимости от оператора, описывающего этот ФВ, иллюстрируется в приложении Д, но, так или иначе, с точки зрения мультиагентной системы это приводит к порождению новых ФД (выходных для соответствующего ФВ) и новых агентов, если семантика оператора подразумевает появление новых ФВ в следующем (в смысле определений 18, 20 и 21) состоянии. Таким образом, мультиагентная система прямо моделирует исполнение ФА.

Теперь дадим конкретное определение агента как произвольного последовательного сериализуемого процесса, имеющего состояние и не имеющего связей с окружающим миром кроме взаимодействия со средой через следующий интерфейс. Интерфейс приводится на языке C++ в том виде, в котором он реализован в системе LuNA. Тут класс DF представляет ФД, класс Id представляет сериализуемое имя, класс CF представляет агента, а класс Locator — локационную функцию.

- DF wait(Id id). Получить значение входного ФД по ссылке на него. Ожидать, если значение ещё не получено.
- Id create_id(). Создать новый объект ссылочного типа. Используется для конструирования ссылочных выражений.
- CF fork(**int** entry_point). Породить дочернего агента. Параметр entry_point задаёт идентификатор точки входа в программу порождаемого ФВ (подробнее о точках входа см. раздел 3.2.2).
- **void** post(Id id, DF val, Locator locator, **int** req_count). Передать системе порождённый выходной ФД с заданным идентификатором id, значением val, который должен храниться в месте, определяемом локатором[6] locator, и к которому должно быть совершено req_count обращений (запросов). Если req_count не задан,

[6] В разделе локатором называется модуль, вычисляющий локационную функцию

то количество обращений считается неизвестным, и ФД должен быть удалён явно вызовом destroy.

- **void** request(Id id, Locator locator). Запросить входной ФД по ссылке id, хранящийся в месте, определяемом локатором locator. Доступ к доставленному значению будет осуществляться с помощью метода wait.
- **bool** migrate(Locator locator). Мигрировать на другой узел по направлению к месту, задаваемому локатором locator. Метод возвращает **true** если миграция состоится и **false** если текущий узел уже является требуемым местом.
- **void** destroy(Id id, Locator locator). Отдать команду на удаление ФД по ссылке id, находящемуся в месте, определяемом локатором locator. На момент вызова этого метода ФД должен уже быть потреблённым всеми ФВ, для которых он является входным.
- **void** push(Id dfid, DF val, Id cfid, Locator locator). Отправить ФД с идентификатором dfid и значением val получателю (ФВ с идентификатором cfid), находящемуся в месте, определяемом локатором locator, напрямую, без промежуточного хранения ФД (упреждающая посылка — раздел 2.4.3).

Таким образом, агент может быть задан в виде программы на традиционном последовательном языке программирования, применённой к конкретным входным данным. Примером программы агента может служить C++-процедура, состоящая, преимущественно, из вызовов вышеприведённых методов и обеспечивающая через них реализацию семантики соответствующего оператора ФА.

Отметим, что хотя количество ФВ (а, соответственно, и количество агентов) может быть потенциально бесконечным, количество программ агентов всегда будет конечным, т.к. каждая программа агента соответствует одному оператору ФА, а количество операторов в ФА конечно согласно определению 8.

3.1.3 Анализ предлагаемых языковых средств

Практичность предлагаемого языка LuNA обусловлена тем, что LuNA-программа, фактически, представляет собой описание требуемых множеств в терминах, близких к модели ФА. Пользоваться таким языком может любой человек, имеющий базовые знания по математике и теории множеств. Текстовый синтаксис языка позволяет использовать практически любой текстовый редактор или интегрированную среду разработки. Читаемость исходных кодов может быть повышена средствами подсветки синтаксиса — они широко распространены и многие

имеют возможность настройки на заданный синтаксис. Реализация ФК не требует специальных знаний в области параллельного программирования, достаточно обычных знаний и умений последовательного процедурного программирования на языках C/C++ и Fortran, а также совместимых с ними на этапе линковки. В частности, накопленные человечеством последовательные подпрограммы (например, последовательные библиотеки подпрограмм) могут быть использованы в составе ФК. Вопросы, связанные с поведением, решаются пользователем путём добавления рекомендаций, которые не могут нарушить работоспособность LuNA-программы. Это обуславливает простоту отладки как функциональной, так и поведенческой. Язык поддерживает средства модульности, как по файлам, так и по подпрограммам, необходимые для реализации больших проектов, а также базовые средства препроцессора, полезность которых проверена практикой языков C/C++. В языке имеются базовые типы данных, позволяющие практически исключить потребность в описании «мелких» ФК, связанных с границами циклов, условиями и арифметическими операциями. LuNA-программа, по сути, представляет собой не что иное, как описание ФА, поэтому свойства модели ФА наследуются и LuNA-программами. Грамматика языка имеет C-подобный синтаксис, что упрощает освоение языка людьми, имеющими опыт программирования на C/C++, Java, Javascript или других C-подобных языках. В частности, нюансы синтаксиса, такие как приоритет операций и области видимости имён в точности соответствуют таковым языка C. Возможность определения реализаций ФК на языке C++ обеспечивает возможность повторного использования кода, который может быть использован из C++, а также обеспечивает соответствующую поддержку взаимодействия с внешним миром (interoperability).

Грамматика языка является простой, вследствие чего может быть описана формально в виде, доступном для большинства популярных инструментов автоматического разбора лексики и синтаксиса. Это же касается и директив препроцессора.

Исполняемое представление (ИП) удовлетворяет всем предъявляемым к нему требованиям. А именно, оно позволяет представить в исполняемом виде любой ФА, обеспечивает возможность статически (транслятору) вложить в ИП конкретное управление и распределение ресурсов, обеспечивает возможность исполнительной системе влиять на управление и распределение ресурсов динамически и обеспечивать динамические свойства, такие как динамическая балансировка нагрузки и пр.

Также ИП позволяет осуществлять профилирование, трассировку и контроль за исполнением ФА в терминах ФВ и ФД, т.к. эти объекты явно присутствуют во время исполнения, а генерируемая программа агента может быть инструментирована необходимым профилировочным, отладочным или иным кодом. Использование традиционного языка

программирования (C++) в качестве языка описания программы агента позволяет использовать мощнейший инструментарий последовательного программирования (в первую очередь, компиляторы C++) для оптимизации и анализа конструируемого кода. Программа агента конструируется на конкретный частный случай, что, вообще говоря, более эффективно, чем использование интерпретатора, вынужденного обрабатывать общий случай. Сам по себе мультиагентный подход по существу соответствует имеющейся ситуации, когда агент не знает всей картины о ходе вычислений и вынужден опираться на локально доступные данные и взаимодействие с окружением и агентами в своей окрестности. В рамках предложенного ИП остаются широкие возможности по усложнению как программ агентов, так и их окружения (ИС), что позволяет организовывать сложные схемы распределённых вычислений. Как пример, алгоритм Rope of Beads (раздел 2.4.2) естественным образом реализуется в виде распределённого динамического балансировщика нагрузки как модуля ИС и программного кода агента по миграции в соответствии не с номером узла, а с предписаниями этого модуля.

Таким образом, и язык LuNA, и ИП обеспечивают все текущие потребности в описании ФА на разных уровнях работы системы, а также обеспечивают широкие возможности расширения, востребованные в долгосрочной перспективе развития проекта.

3.2 Форматы представления данных

В разделе представлена информация об особенностях представления ФА на разных стадиях трансляции и исполнения.

3.2.1 Внутреннее JSON-представление LuNA-программы в трансляторе

В результате синтаксического анализа в трансляторе формируется абстрактное синтаксическое дерево LuNA-программы в машинно-ориентированном представлении (в отличие от человеко-ориентированного синтаксиса языка LuNA). В качестве такого представления выбрано представление на базе распространённого формата JSON (JavaScript Object Notation). Преимуществом такого выбора является возможность осуществлять анализ LuNA-программы, проверку на ошибки, оптимизацию, конструирование рекомендаций и выполнять другие задачи транслятора как отдельные преобразования над JSON-представлением программы, которые могут выполняться как различными модулями, так и отдельными программами. Также JSON-представление может использоваться для получения любой

информации о программе исполнительной системой, т.к. JSON-представление является частью итоговой сборки LuNA-программы. JSON имеет хорошую инструментальную поддержку во многих языках программирования, включая используемые в системе LuNA языки Python и C++.

3.2.2 Использование C++ для представления программ агентов

Исполняемое представление LuNA-программы представляет собой, по сути, набор программ агентов, сгенерированных по исходной LuNA-программе. Программа агента — это control-flow последовательность команд, которые должен выполнить агент за время своего существования. Для её представления можно было бы использовать различные языки программирования, в том числе интерпретировать собственный набор команд. Выбор сделан в пользу C++ по следующему ряду причин. Во-первых, ИС написана на C++ и, как следствие, наиболее совместима с C++. Во-вторых, C++ имеет чрезвычайно хорошую инструментальную поддержку, включая компиляторы — программа агента, скомпилированная компилятором C++, будет хорошо оптимизирована. Это позволяет переложить многие задачи программной оптимизации последовательного кода (программ агентов) с системы LuNA на компилятор C++.

Несмотря на преимущества использования языка C++ в качестве языка описания программ агентов возникает следующая проблема. В соответствии с моделью исполняемого представления LuNA-программы возможны ситуации, когда агент ожидает доставки ФД, либо мигрирует на другой узел и продолжает там выполнение своей программы. Прямая реализация этого механизма на C++ означала бы прерывание работы C++-функции с последующим продолжением, возможно, на другом узле, что технически затруднительно сделать средствами C++[7]. Для решения этой проблемы было принято решение представлять программу ФВ не одной C++-функцией, а цепочкой C++-функций, которые будем называть блоками программы ФВ. Для каждого ФВ определён стартовый блок, который запускается ИС. Каждый блок возвращает значение, предписывающее системе либо исполнять следующий заданный блок, либо осуществить миграцию на другой узел, либо ожидать входных ФД, либо завершить выполнение программы ФВ. Получив возвращаемое значение блока, ИС выполняет запрошенное действие, возможно, передаёт состояние ФВ на другой узел и запускает заданный блок. Этот механизм позволил решить проблему прерывания и продолжения исполнения C++-функции.

Применение этого подхода для больших программ, где количество блоков может измеряться тысячами, на практике встретило ограничение в компиляторах C++ на максимальное

[7] В стандарте C++20 появилась поддержка т.н. «ко-рутин» (coroutine), которые позволяют прерывать и продолжать выполнение C++-функции, но лишь в пределах одного узла.

количество функций, которые могут быть определены в одном файле. В связи с этим программы фрагментов генерируются не в один, а в несколько файлов (по 200 на файл в текущей реализации), чтобы не превышать имеющееся ограничение.

3.2.3 Единый файл собранной программы

Результат трансляции LuNA-программы в исполняемое представление является комплектом из нескольких разных объектов. Это JSON-представление LuNA-программы, сгенерированный и скомпилированный код программ агентов, скомпилированный код пользовательских модулей, скомпилированные внешние (foreign) блоки и другие объекты. Из соображений удобства все эти объекты упаковываются в виде стандартной динамической библиотеки (файл *.so в Linux), с известным интерфейсом (соглашением по определённым в ней функциям и их именованию). Все фрагментированные подпрограммы (атомарные и структурированные) представлены C++-функциями в библиотеке, а также имеются сервисные функции, перечисляющие множество блоков программ ФВ и предоставляющие JSON-представление LuNA-программы. Использование единого файла удобно тем, что всё собрано в одном месте. Использование стандартного формата динамической библиотеки *.so также имеет ряд преимуществ. Во-первых, для работы с таким файлом может быть использовано множество стандартных инструментов. Во-вторых, с точки зрения ИС неважно, как этот файл был получен, лишь бы он следовал соглашению. Например, неважно, были разработаны фрагменты кода на C++ или Фортране, или другом языке. Само описание ФА могло быть исходно описано на некотором другом языке описания ФА (не LuNA; сейчас таких языков нет, но они могут появиться в будущем). Существует стандартный механизм удовлетворения внешних зависимостей библиотеки.

3.2.4 Позиционная информация

Под термином позиционная информация понимается информация о том, в каком месте (номер строки, смещение в строке) какого исходного файла находится то или иное описание. Это удобно для вывода сообщений об ошибках. Например, если пользователь пытается использовать некоторое имя x в качестве выходного аргумента некоторого ФК, а имя x в данном контексте обозначает входной параметр подпрограммы (что является недопустимой ситуацией), то система должна вывести осмысленное сообщение об ошибке и, желательно, показать, в каком месте

какого исходного файла находится ошибочное употребление имени x, а также где это имя было объявлено. Эта простая и привычная идея наталкивается в реализации на проблему, что к моменту обнаружения ошибки исходный листинг уже претерпел ряд изменений при препроцессинге (или вовсе трансформирован в одно из внутренних представлений). В отличие от большинства языков программирования в трансляторе LuNA ситуация осложняется тем, что в листинге LuNA программы может быть вложен внешний блок с C++-кодом некоторого ФК (подробнее о внешних блоках см. раздел 3.2.2). На практике это означает, что внешний блок будет вынесен в отдельный специально для этого созданный C++ файл для компиляции компилятором C++, к этому файлу будут добавлены необходимые префиксы и суффиксы, сгенерирован заголовок C++-функции. И если при всём при этом в этом файле возникнет ошибка компиляции C++, то позиция ошибки будет иметь мало общего с расположением ошибки в исходном файле, видимом пользователю. Для того, чтобы обеспечить соответствие позиционной информации на всех этапах трансляции и исполнения каждый текстовый (не JSON) файл снабжается таблицей соответствия позиций символов промежуточных файлов позициям этих определений в исходных файлах, а ко всем подходящим объектам JSON-файлов добавляется поле с позиционной информацией. Это позволяет преобразовывать и выводить пользователю информацию о позициях ошибок в тех файлах, которые видны пользователю. Кроме ошибок эта информация будет востребована в отладчиках и профилировщиках LuNA-программ.

3.3 Организация системы LuNA

3.3.1 Организация исходного кода

Исходный код системы LuNA организован обычным образом. А именно, имеется репозиторий в системе контроля версий git[8], доступный для скачивания из сети Интернет по адресу https://gitlab.ssd.sscc.ru/luna/luna. В репозитории имеется инструкция по установке, которая сводится к установке зависимых библиотек и переменных окружения системы. Дальнейшее управление осуществляется стандартной утилитой GNU Make, которая позволяет скомпилировать систему, запустить пакет тестов или очистить репозиторий от промежуточных файлов.

Дерево директорий имеет стандартную структуру: в папке include находятся заголовочные файлы C++, в папке src находятся исходные коды системы на C++, а также грамматика языка

[8] https://git-scm.com/

LuNA в формате утилит flex и bison. В папке scripts находятся исходные файлы системы на языке Python, а также shell-скрипт конвертации исходных файлов LuNA с предыдущей версии системы. В репозитории имеются десятки примеров LuNA программ разного уровня сложности, в том числе учебный набор LuNA-программ, предназначенных для обучения пользователей основам языка LuNA. Также частью проекта является различная документация к системе LuNA в формате вики-страниц и система отслеживания ошибок (баг-треккер) на базе системы gitlab [83]. Документация и баг-треккер доступны по вышеуказанной ссылке.

Система LuNA является экспериментальной системой, предназначенной не только для параллельной реализации численных алгоритмов, но и для практического исследования свойств системных алгоритмов трансляции и исполнения LuNA-программ. Кроме того, система такого типа должна со временем наращивать «багаж» частных системных алгоритмов и эвристик, обеспечивающих всё более эффективную работу системы для всё более широкого класса приложений. Как следствие, система должна обеспечивать модульность, позволяющую заменять различные системные алгоритмы и накапливать частные решения в виде системных модулей. Такая возможность поддержана наличием внутренних системных интерфейсов и реализующих их модулей для всех ключевых решений в системе. К таким интерфейсам относятся:

— Расширяемый синтаксис описания рекомендаций.
— Статический анализ LuNA-программы и статическое принятие решений, реализуемые модулями транслятора, доопределяющими её JSON-представление.
— Общий вид алгоритмов локации, поддерживающих динамическое отображение (раздел 2.4.1).
— Слабосвязанные модули реализации ФД.
— Слабосвязанный модуль динамической балансировки нагрузки
— Слабосвязанные модули сборки мусора
— Слабосвязанный модуль выбора ФВ на исполнение из числа готовых.

3.3.2 Транслятор

В целом транслятор LuNA организован стандартным для трансляторов образом, он имеет конвейерную структуру. Помимо собственно трансляции LuNA-программы в исполняемое представление транслятор представляет собой расширяемую платформу для включения и накопления частных системных и эвристических алгоритмов, управляющих поведением ФА.

LuNA-транслятор получает на вход описание ФА и множества ФК в виде набора исходных файлов на языках LuNA и C++, а на выход выдаёт исполняемое представление заданного ФА в

виде динамически подключаемой библиотеки (файл с расширением .so). В этой библиотеке хранятся в двоичном машинном коде программы всех агентов, ФК и другая техническая информация, необходимая для реализации ФА исполнительной системой.

Пользователь описывает LuNA-программу в виде набора файлов на языке LuNA и, возможно, набора файлов на языке C++, представляющих реализации ФК. Эти файлы обрабатываются препроцессором, который удаляет комментарии, подставляет включаемые файлы, делает макроподстановки, выделяет внешние блоки. Затем выполняются стандартные этапы лексического и синтаксического анализа — разбирается текст программы для построения абстрактного синтаксического дерева программы, которое хранится в формате JSON. Далее выполняется семантический анализ — построенное дерево программы проверяется на ряд семантических ошибок, и к нему применяются оптимизирующие преобразования. После этого системой (статически) принимается часть решений об исполнении LuNA-программы, что выражается в доопределении рекомендаций. Далее генерируется внутреннее исполняемое представление LuNA-программы в виде последовательного кода, который компилируется последовательным компилятором вместе с выделенными внешними блоками, пользовательскими модулями, реализующих операции, и сгенерированных программ ФВ в стандартную динамическую библиотеку. Эта библиотека подаётся на вход исполнительной системе, которая и осуществляет распределённое исполнение LuNA-программы. В результате выполнения LuNA-программы образуются (если требовалось) журнальные файлы, которые анализируются инструментами анализа и визуализации.

Ввиду простоты и относительной нестандартности, связанной с внешними блоками и сохранением позиционной информации, препроцессор разработан вручную на языке Python (а не использован какой-либо из множества существующих инструментов препроцессинга). Для лексического и синтаксического анализа использованы стандартные утилиты flex и bison. Часть транслятора, осуществляющая семантический анализ, принятие статических решений и генерацию внутреннего представления, реализована на языке Python (что непринципиально). Программы агентов генерируются на языке C++ (и, соответственно, компилируются C++ компилятором).

Работа препроцессора, разбор лексики и синтаксический анализ в тексте диссертации не рассматриваются, т.к. не содержат оригинальных алгоритмов и почти тривиально реализуются на базе существующего инструментария. Поэтому будем считать абстрактное синтаксическое дерево построенным и рассмотрим основной цикл работы LuNA-транслятора.

LuNA-транслятор является однопроходным. Он обрабатывает по очереди все подпрограммы LuNA-программы, и эта обработка сводится, главным образом, к рекурсивному

разбору всех операторов в теле подпрограммы. Рекурсивность заключается в том, что блочные операторы (циклы и условные операторы) также имеют тело, состоящее из операторов, к которым применяется та же процедура обработки. Обработка различных подпрограмм осуществляется полностью независимо. Кроме разбора подпрограмм транслятор также генерирует программу запуска головного агента, состоящую в операторе исполнения головной подпрограммы (аналог функции main в языке C++).

3.3.3 Исполнительная система

Исполняемое представление ФА, выработанное транслятором LuNA подаётся на вход исполнительной системе, которая представляет собой распределённую параллельную программу. Исполнительная система (ИС) осуществляет распределённое исполнение LuNA-программы. В результате выполнения LuNA-программы образуются (если требовалось) журнальные файлы (логи, профиль), которые анализируются инструментами анализа и визуализации. Исполнительная система написана на языке C++ из соображений производительности. Для реализации коммуникаций используется библиотека MPICH [84], реализующая стандарт MPI [24].

3.3.4 Профилировщик

Модель ФА даёт возможность дифференцированного профилирования исполнения LuNA-программ в таких терминах, как размер ФД, время выполнения ФВ, распределение ФД или ФВ по узлам во времени, нагрузка на сеть, созданная тем или иным ФД, количество существовавших копий того или иного ФД, время жизни ФД после последнего его потребления и до удаления сборщиком мусора, время от вычисления ФД до его потребления, а также множество других параметров, специфичных для модели ФА. Профилирование обладает большим потенциалом с точки зрения оптимизации поведения ФА (т.е. улучшения нефункциональных характеристик исполнения ФА). В отличие от традиционного профилирования вся эта информация доступна для автоматического сбора и анализа. В стандартных профилировщиках автоматически собирается лишь более низкоуровневая информация, а для фиксации более значимых событий пользователь должен самостоятельно инструментировать свой код, в том числе при использовании профилировщиков, таких как MPE [85]. При этом обычные профилировщики также могут быть использованы для профилирования LuNA-программ. В системе LuNA реализованы базовые

средства профилирования, фиксирующие ключевые события в исполнении LuNA-программы в привязке ко времени и вычислительному узлу. Постобработка профиля представлена некоторыми простыми инструментами, осуществляющими анализ и визуализацию полученных данных.

Особый интерес представляет возможность автоматической настройки исполнения LuNA-программы для серии сходных экспериментов. Результаты профилирования подаются на вход транслятору, который учитывает их для конструирования улучшенных рекомендаций. Такая оптимизация на основе профилирования может использоваться для автоматического учёта характеристик входных данных и особенностей (например, неоднородностей или характера посторонней нагрузки) вычислителя. В автоматическом режиме такая настройка была реализована для одной из предыдущих версий системы LuNA, где показала положительные результаты. Подробнее об этом см. раздел 4.5.

3.3.5 Отладчик

С целью повышения информативности сообщений об ошибках, что важно для практического использования системы, требуется (в том числе ценой дополнительных накладных расходов) сохранять дополнительную отладочную информацию, не являющуюся необходимой для собственно исполнения LuNA-программы. К такой информации относится информация об именах ФВ и ФД, а также о позициях в исходном коде тех или иных данных. Учитывая, что LuNA-программа проходит ряд преобразований, начиная с препроцессора, расположение объектов в листинге относительно исходного кода изменяется, поэтому соответствие должно специально сохраняться и поддерживаться. Для этого в системе используется несколько дополнительных представлений (см. раздел 3.2.4).

Ещё одним местом сохранения отладочной информации служат сгенерированные программы агентов. Когда транслятор генерирует ту или иную команду в листинг программы агента, то он обладает разной информацией, полезной для отладки, но не являющейся необходимой для исполнения LuNA-программы. Эта информация добавляется в листинг в виде комментариев, а также вносится в системные структуры данных времени исполнения. Примером может служить информация об именовании ФД. Во время трансляции известно имя ФД вплоть до места его объявления в коде и места его употребления в каждом конкретном случае. Во время исполнения эта информация отсутствует, имеется лишь системный идентификатор, представляющий собой кортеж из нескольких целых чисел. При генерации кода формирования ФД транслятор может также сгенерировать код, который бы формировал также необходимую

отладочную информацию о данном ФД. Проблема состоит в том, что хранение этой информации требует дополнительных ресурсов, что не всегда целесообразно, поэтому эта информация добавляется только в отладочном режиме трансляции, а исполнительная система компилируется в двух версиях — отладочной и обычной. Как следствие, в обычном (не отладочном) режиме никаких дополнительных накладных расходов нет, а в отладочном режиме требуемая информация добавляется.

Интерактивная отладка LuNA-программ возможна, хотя и обладает своей спецификой. Например, традиционный механизм установки точек останова (breakpoint) принимает такую форму, что на ФВ устанавливается точка останова, и исполнение ФВ «замораживается», в то время как выполнение других ФВ продолжается за исключением тех, что информационно завияст от остановленного ФВ. При этом последовательные ФК можно отлаживать стандартными средствами отладки, например GDB [86]. Ввиду иммутабельности данных стандартный механизм отслеживания модификации памяти вырождается в отслеживание состояния вычисленности ФД и не требует задействования низкоуровневых механизмов для реализации. Частично вопрос интерактивной отладки был проработан в выпускной квалификационной работе бакалавра Р.Н. Мустакова [87], но за рамки настоящей работы это выходит.

3.3.6 Специализированные способы реализации ФА

Исполнение LuNA-программ частного вида возможно на базе исполнительных систем частного вида. Если ФА имеет некоторый частный вид, имеющий специальную поддержку, то его исполнение может быть проведено на исполнительной системе частного вида. Примером такой системы служит LuNA-Framework [88], разработанный в том же коллективе, что и система LuNA. Он использует императивную модель вычислений, основанную на сетях Петри, и способен исполнять ФА частного вида в мультипроцессорах с обеспечением в ряде случаев более высокой эффективности, чем базовая ИС. Возможны и гибридные варианты исполнения, когда часть ФА реализуется на базе одной ИС, а часть — на базе другой, причём ФД передаются от одной системе к другой в соответствии с имеющимися информационными зависимостями в ФА. Подобные эксперименты описаны в [89]. В перспективе могут быть разработаны и другие исполнительные системы, обеспечивающие эффективное исполнение различных классов LuNA-программ (напр., [90]). Сюда же относится возможность конструирования параллельных программ, реализующих заданный ФА без исполнительной системы (напр., [91]).

Также была разработана экспериментальная исполнительная система, ориентированная на широкий спектр устройств для исследования вопросов эффективной реализации ФА в условиях

мультикомпьютера повышенной неоднородности [92]. Эта ИС была реализована на языке Javascript, что позволило исполнять LuNA-программы на любых устройствах, имеющих стандартный браузер или среду NodeJS (смартфоны, планшеты, ноутбуки, сервера, суперЭВМ).

Алгоритм воспроизведения трасс (раздел 2.4.8) был реализован в виде системного модуля записи трасс и отдельной подсистемы воспроизведения трасс. Подсистема описана в разделе 3.4.5, а результаты её экспериментального исследования представлены в разделе 4.6.

3.3.7 Отладочный стенд для алгоритмов динамической балансировки нагрузки на узлы

В рамках работы был создан т.н. отладочный стенд для исследования различных алгоритмов динамической балансировки вычислительной нагрузки [93]. Отладочный стенд представляет собой программный инструмент, позволяющий через унифицированный интерфейс подключить внешний алгоритм балансировки нагрузки на вычислительные узлы в виде отдельного модуля, задать режим нагрузки на различные вычислительные узлы в task-based модели, имитирующей исполнение ФА, и снять характеристики балансировки нагрузки на различных параметрах.

3.4 Технические вопросы трансляции и исполнения LuNA-программ

3.4.1 Отслеживание имён

Имена, употребляемые в ссылках, являются нетерминалами и могут означать разные объекты в зависимости от контекста употребления — это может быть индексная переменная, параметр подпрограммы или имя ФД (см. листинг 3.3).

Листинг 3.3. Пример проблемы определения значения имён при трансляции.

```
01: sub my_routine(int n) {
02:   df n;
03:   cf a: init(7, n); // n - это ФД или параметр подпрограммы?
04: }
```

Для того, чтобы отслеживать это соответствие в транслятор была введена структура **пространство имён** (ПИ), хранящая информацию о «видимых» именах для каждого оператора. Пространства имён организованы в древовидную структуру, корнем которой является ПИ подпрограммы. Дочерние ПИ формируются для каждого оператора и для каждого блока (в

блочных операторах). В частности, часть этой информации должна быть доступна агенту во время выполнения, храниться в его локальной памяти, сохраняемой при миграции. В этой памяти необходимо хранить все объекты ссылочного типа, необходимые для построения ссылок на все входные и выходные ФД. Например, если ФВ находится в теле некоторой подпрограммы, но использует из всех имён только часть (для обозначения входных и выходных ФД), то пространство имён этого ФВ будет ограничено только этими именами. Практический смысл такого ограничения в том, чтобы сократить объём ресурсов (памяти), требуемый на реализацию ФВ. В частности, ФВ, задаваемые блочными операторами, такими как for, while, if, имеют своё собственное пространство имён, необходимое им для исполнения, а также пространство имён следующего уровня вложенности для своего тела. Операторы в теле также будут иметь свои пространства имён, формируемые на основе пространства имён тела блочного оператора. Каждый ФВ может использовать имена всех пространств имён, в которые он вложен, и при реализации такого ФВ необходимо обеспечить наличие в его контексте этой ссылочной информации, а также в контексте всех пространств имён, в которые это пространство имён вложено. Алгоритм, представленный ниже в подразделе, решает как раз эту задачу.

Каждому имени в пространстве имён может быть поставлено в соответствие значение одного из следующих видов:

- **Ссылка.** Должна быть получена из родительского контекста, или создана новая (для ФД, определённых во вложенном пространстве имён)
- **Параметр подпрограммы.** Возможно только для корневых пространств имён.
- **Конечный код.** Непосредственный код на языке C++, который должен быть вставлен в выходной листинг (в генерируемое внутреннее представление), и который реализует обращение к объекту, которому соответствует это имя. Например, если это имя ФД, то конечный код будет осуществлять обращение к ФД (метод wait в API, см раздел 3.1.2), если это параметр цикла, то конечный код будет представлять имя переменной генерируемого C++ листинга, реализующей параметр цикла. Это также может быть константа или арифметическое выражение в исполняемом представлении.

Входом для алгоритма является абстрактное синтаксическое дерево LuNA-программы. Выходом является часть кода программы агента (на языке C++), отвечающая за инициализацию начального состояние агента на основе состояния родительского агента. При работе алгоритма предполагается, что в трансляторе осуществляется рекурсивный обход всех операторов и выражений в соответствии с абстрактным синтаксическим деревом программы, начиная с подпрограмм. При этом каждой подпрограмме и каждому вложенному блочному оператору заводится системная структура, представляющая пространство имён. Каждая такая структура

(кроме корневой структуры подпрограммы) связана с родительской структурой, а родительская содержит информацию о параметрах подпрограммы — их именах и типах. Далее при рекурсивном обходе дерева программы каждый раз, когда встречается ссылка, то для её разрешения вызывается процедура resolve_name или resolve_value, в зависимости от того, применяется эта ссылка синтаксически как ссылка или как значение. Алгоритм работы этих функций представлен в листинге 3.4.

Листинг 3.4. Алгоритмы разрешения имён и значений.

```
01: SUB resolve_name(S, n)
02:   PARAM S                          ▷ структура, представляющая ПИ
03:   PARAM n                          ▷ имя, которое требуется разрешить
04:   IF n ∈ resolved_names(S) THEN     ▷ если n было разрешено ранее в этом ПИ
05:     RETURN resolved_names(S)[n]    ▷ то вернуть это значение
06:   END IF
07:   IF n объявлено в блоке как имя ФВ или ФД THEN     ▷ если в этом ПИ n было объявлено
08:     LET resolved_names(S)[n] ← создать новую ссылку  ▷ как имя ФД или ФВ, то создать
09:     RETURN resolved_names(S)[n]                      ▷ и сохранить новую ссылку для n
10:   END IF
11:   IF n — параметр подпрограммы базового типа THEN   ▷ если n — это параметр подпрограммы типа
12:     ERROR «доступ к базовому типу как к ссылке»      ▷ int, real или string, то сообщить об ошибке
13:   END IF
14:   IF parent(S)=nil THEN                  ▷ ссылка не разрешена, и родительского ПИ нет
15:     ERROR «имя не определено»            ▷ значит имя не удалось разрешить
16:   END IF
17:   LET p ← resolve_name(parent(S), n)     ▷ попробовать разрешить имя в родительском ПИ
18:   LET resolved_names(S)[n] ← привязать к ссылке p   ▷ и привязать имя к ссылке родительского ПИ
19:   RETURN resolved_names(S)[n]
20: END SUB

21: SUB resolve_value(S, n)
22:   PARAM S                          ▷ структура, представляющая ПИ
23:   PARAM n                          ▷ имя, которое требуется разрешить
24:   IF n ∈ resolved_values(S) THEN   ▷ если имя было разрешено ранее в этом ПИ,
25:     RETURN resolved_values(S)[n]   ▷ то вернуть это значение
26:   END IF
27:   LET p ← CALL resolve_name(S, n)  ▷ попытаться разрешить имя как ссылку (а не как значение)
28:   IF p≠ERROR THEN                  ▷ если это удалось, то считать имя обращением к ФД
29:     LET resolved_values(S)[n] ← обращаться как к значению ФД по ссылке p
30:     RETURN resolved_values(S)[n]
31:   END IF
```

```
32:  IF parent(S)=nil THEN                          ▷ если до сих пор имя не разрешено и нет родительского ПИ,
33:    ERROR «имя не определено»                    ▷ то сообщить об ошибке
34:  END IF
35:  LET p ← CALL resolve_value(parent(S), n)       ▷ попытаться разрешить имя в родительском контексте
36:  LET resolved_values(S)[n] ← привязать обращение к значению p     ▷ если успешно, то привязать
37:  RETURN resolved_values(S)[n]                   ▷ имя к значению в родительском контексте
38: END SUB
```

В листинге предполагается, что оператор ERROR приводит к аварийной остановке (ошибка трансляции). Исключение составляют строки 27–28, тут случай ошибки в подпрограмме обрабатывается явно.

Поясним, что понимается под созданием и привязкой к ссылкам (строки 08, 18, 29). В каждом конкретном исполнении LuNA-подпрограммы или тела блочного оператора имена локальных ФД (объявленных в соответствующем блоке) должны соответствовать разным ФД. Для этого всякий раз при исполнении блочного оператора для всех определённых в нём имён динамически создаются системные идентификаторы. Каждый вложенный ФВ, который использует эти имена должен при этом сохранить системные идентификаторы, соответствующие этим именам. Это осуществляется при создании дочерних ФВ. Вопрос, который должен быть при этом решён — это какие имена каждый ФВ использует, и каким системным идентификаторам они соответствуют. Этот вопрос и решают процедуры из листинга ?, и результатом для каждого имени может быть одно из следующих. Если имя ФД или ФВ впервые появляется в ПИ, то при создании соответствующего ФВ будет создан новый системный идентификатор (строка 08). Если имя объявлено в родительских ПИ, то при создании дочернего ФВ ему будет передан системный идентификатор от родительского ФВ (строка 18). Если обращение к имени осуществляется по значению (строка 27; например, ФД является входным для некоторого ФВ), то это имя так же будет создано или передано от родительского ПИ. Если же имя определяет не ФД, а параметр подпрограммы или счётчик цикла, то соответствующее выражение будет использовано для инициализации соответствующего имени при создании ФВ.

3.4.2 Поэтапное запрашивание входных ФД

Для реализации алгоритма доставки ФД по запросу (раздел 2.4.4) необходимо получить перечень идентификаторов всех входных ФД для заданного ФВ. Эта задача осложняется тем, что индексные выражения ссылки также могут содержать ссылки на другие ФД, значения которых тоже необходимо получить, чтобы узнать значения индексов. Наивное решение этой проблемы состоит в рекурсивном разборе ссылки, где каждое индексное выражение рекурсивно

разбирается. В случае, если обнаруживается ссылка на ФД, который отсутствует на текущем узле — он запрашивается и ожидается, после чего процесс повторяется. Недостатком этого алгоритма являются задержки от ожидания очередного ФД в случаях, когда можно было бы запросить более одного ФД одновременно. Запросить же все сразу не всегда возможно, если значение индекса зависит от ФД, значение которого ещё неизвестно (или не имеется на текущем вычислительном узле). Поэтому запрашивать значения ФД нужно сразу, как только все индексы в ссылке на него становятся вычислимыми. Но на практике такое решение сложно получить из-за того, что программа запросов и ожиданий для ФВ должна быть сгенерирована статически (как часть программы агента), когда ещё неизвестно, в каком порядке будут появляться значения ФД.

Для решения этой проблемы предлагается алгоритм поэтапного запрашивания ФД, который представляет собой компромисс между простотой конструирования программы агента и своевременностью отправления запросов. Идея алгоритма состоит в том, что на первом этапе запрашиваются все ФД, которые могут быть запрошены сразу (т.е. индексные выражения которых не содержат других ссылок). На каждом последующем этапе запрашиваются те ФД, которые могут быть запрошены при условии, что все ФД предыдущего этапа получены. Хотя этот алгоритм и не является оптимальным в том смысле, что некоторые ФД будут иногда запрашиваться позже, чем могли бы, но у него есть ряд преимуществ. Во-первых, он обладает малыми накладными расходами времени исполнения. Во-вторых, между этапами нет строгой границы, и ФД следующего этапа могут запрашиваться по мере поступления значений ФД предыдущего этапа (хотя и в фиксированном порядке внутри этапа). Рассмотрим этот алгоритм.

Алгоритм работает рекурсивно, применяясь к выражению. Имеется глобальное состояние, состоящее из множества D зависимостей, которые уже считаются разрешёнными. Зависимостью, в данном случае, является ссылочное выражение, т.е. ссылка, индексами которой являются выражения, а под разрешённостью понимается ситуация, когда все ФД, использованные в ссылочном выражении (или подвыражении), имеются на текущем узле. В начале множество D пусто. Результатом работы алгоритма является множество зависимостей, которые могут быть (и должны быть) непосредственно запрошены с использованием D.

Шаги алгоритма запросов ФД:

1. Если выражение является константой, то вернуть пустое множество.
2. Если выражение является ссылкой, то применить алгоритм ко всем индексным выражениям и вернуть объединение полученных множеств (за вычетом D). Если объединение этих множеств пусто, то вернуть множество с единственным элементом, содержащим ссылку (за вычетом D).

3. Если выражение является арифметической операцией или операцией приведения типа, то применить алгоритм ко всем операндам и вернуть объединение полученных множеств (за вычетом D).

Результатом работы алгоритма является множество ссылок, которые должны быть запрошены на первом этапе. Затем это множество добавляется во множество D и алгоритм повторяется, возвращая ФД следующего этапа. Процесс продолжается до тех пор, пока результатом работы алгоритма не станет пустое множество.

Другой проблемой запрашивания ФД является то, что ссылки на один и тот же ФД могут быть синтаксически разными, например x[4] и x[2+2], а также x[N], где N — это ФД и его значение равно 4. Данный алгоритм работает со ссылками, т.е. для выражения x[4]+x[2+2] ФД x[4] будет запрошен дважды, по ссылке x[4] и по ссылке x[2+2]. На первый взгляд это кажется существенным недостатком, потому что ФД x[4] может оказаться весьма объёмным и/или расположенным в отдалённой части мультикомпьютера, что негативно скажется на эффективности. Поэтому разберём этот вопрос подробнее.

Начнём с того, что упрощение выражений как стандартная техника, применяемая в компиляторах, может с успехом решать эту проблему во многих случаях. Достаточно лишь выполнить эту технику перед исполнением рассматриваемого алгоритма. Тогда транслятор распознает, что ссылки x[4] и x[2+2] тождественны. Тем не менее, не все выражения транслятор сможет упростить. Но более важно то, что не всегда равенство индексов в принципе может быть определено статически, например, для пары ссылок x[4] и x[N], где значение ФД N вычисляется динамически. Значение N может зависеть от входных данных расчёта, и потому решение о равенстве или различии ссылок в принципе не может быть принято статически. В этом случае единственным гарантированным решением со стороны транслятора может быть запрос ФД по обеим ссылкам.

Но даже и в этом случае лишние накладные расходы можно погасить следующим динамичечским механизмом. Пусть в локальной памяти агента сохраняется множество уже запрошенных ФД, и если оказывается, что некоторый ФД уже был запрошен ранее, то повторный запрос не делается. Таким образом все необходимые ФД окажутся доставленными, и запрос на каждый из них будет отправлен ровно один раз.

Полностью же динамический алгоритм определения зависимостей хотя и возможен, но более накладен в силу того, что для этого потребовалась бы полная динамическая интерпретация всех индексных выражений.

3.4.3 Техника удалённых указателей

В исполнительной системе в ряде случаев возникает потребность в применении обратных вызовов (callback) удалённо, когда требуемое событие происходит на одном узле, а реакция на него должна происходить на другом узле. Наивный подход подразумевает наличие некоторой структуры данных (напр., словаря), где регистрируются (и получают идентификатор) обратные вызовы, а затем, при получении соответствующего сетевого сообщения с идентификатором в этом словаре осуществляется поиск записи и вызов функции обратного вызова. Такой подход означает в лучшем случае логарифмическое время поиска объекта в словаре, что означает накладные расходы, особенно при больших размерах таких словарей. В работе реализован другой подход, который получил название удалённых указателей, который обеспечивает константное время поиска обратного вызова.

Удалённым указателем называется пара ⟨node, pointer⟩, где node — это идентификатор узла мультикомпьютера, а pointer — это обычный указатель в памяти данного узла. Удалённые указатели удобно использовать для реализации механизма обратных вызовов (callback), затрагивающей несколько узлов. Например, если некоторый ФД запрошен некоторым ФВ, то к запросу прикрепляется удалённый указатель на запросивший ФВ. Когда ФД передаётся на узел, на котором находится запросивший ФВ, то локальный указатель (валидный на этом узле) разыменовывается, преобразуется к указателю на локальный объект, и к нему делается обращение. Это позволяет за константное (а не логарифмическое) время находить объект на узле.

В том числе этот механизм применён в системе LuNA и для работы с функциональными замыканиями (closure), когда в динамической памяти создаётся лямбда-функция, имеющая замыкание, удалённый указатель на неё передаётся по сети, а локальный указатель «теряется». Но утечки памяти не происходит, т.к. позже по сети приходит удалённый обратный вызов по этому указателю, а сама лямбда-функция удаляется в теле самой лямбда-функции.

Использование удалённых указателей не только обеспечивает константное время поиска указателей на обратные вызовы, но и экономит место в памяти узла на хранение адресов динамических объектов.

3.4.4 Поддержка спецвычислителей

Одним из преимуществ системы LuNA является возможность автоматизации использования спецвычислителей (например, на базе средств CUDA [94]). Для этого пользователь должен предоставить лишь процедуру (CUDA-ядро и т.п.), реализующую заданный

ФК для спецвычислителя поддерживаемого вида. Системная поддержка заключается в том, что в ИС реализована передача данных из памяти CPU в память спецвычислителя, организация запуска процедуры на спецвычислителе с нужными параметрами и передача данных обратно в память CPU. Такая поддержка была реализована в системе LuNA для спецвычислителей, доступных средствами CUDA. Такая поддержка существенно упрощает применение спецвычислителей на практике, т.к. пользователю достаточно разработать процедуру для наиболее ресурсоёмких ФК, а техническую работу по передаче данных, планированию и организации вычислений система берёт на себя. Результаты работы по автоматизации поддержки спецвычислителей представлены в разделе 4.7.

3.4.5. Особенности воспроизведения трасс

Наивная реализация алгоритма воспроизведения трасс (см. раздел 2.4.8) заключается в том, чтобы сгенерировать традиционную распределённую программу (например, на базе C++/MPI), в которой для каждого вычислительного узла будет строка за строкой вызываться очередное действие (получение сообщения, запуск ФВ, отправка ФД и т.п.). Этот подход обладает минимальными возможными накладными расходами. Он был реализован в рамках работы, но оказался неприменим на практике вследствие того, что в реальных приложениях количество строк сгенерированного кода измерялось сотнями тысяч, и компиляция таких MPI-программ могла занимать часы. Поэтому для практического использования был разработан простой интерпретатор трассы, которая предварительно конвертировалась в бинарный формат для экономии памяти и времени интерпретации. Его производительность существенно не отличалась от наивного подхода. Результаты экспериментального исследования этого интерпретатора трасс представлены в разделе 4.6.

3.4.6 Динамическая балансировка загрузки методом Work Stealing

В рамках работ по проекту под научным руководством автора настоящей работы студентами С.А. Трениным и А.В. Чмилем был разработан альтернативный алгоритму Rope of Beads алгоритм динамической балансировки нагрузки на вычислительные узлы на основе метода Work Stealing. Метод Work Stealing известен и применяется для динамической балансировки нагрузки как в общей, так и в распределённой памяти. Этот метод был адаптирован для использования в системе LuNA. Суть разработанного алгоритма в том, что недогруженный узел

может «украсть» ФВ с соседнего перегруженного узла, если все входные ФД для данного ФВ уже имеются на узле. Таким образом, лишь финальный этап жизненного цикла ФВ — исполнение последовательного модуля — осуществляется на соседнем узле, что существенно не сказывается на общем ходе работы системы, но дисбаланс нагрузки уменьшается.

Существенно, что эти два алгоритма (Rope of Beads и Work Stealing) могут работать одновременно, дополняя друг друга. Область применения алгоритмов различна — первый больше подходит для устранения дисбалансов нагрузки на относительно длительных промежутках времени и на нескольких (или многих) соседних узлах. Второй более ориентирован на устранение «точечных дисбалансов» нагрузки, затрагивающие одиночные узлы (перегрузка или недогрузка). Совместная работа двух алгоритмов не исследовалась, но предполагается, что для класса приложений она будет более эффективной, чем каждый алгоритм по отдельности. Также показательно, что модель ФА, системные алгоритмы и архитектура системы LuNA без существенных проблем позволили применение разных алгоритмов динамической балансировки нагрузки, что подтверждает соответствие системы требованию расширяемости по системным алгоритмам.

3.5 Анализ и системы LuNA

Система LuNA является академической экспериментальной системой, поэтому и качество её кода, и удобство пользования системой не претендуют на соответствующие показатели промышленной разработки программ. Тем не менее, разработанная система LuNA реализует идеи и алгоритмы, предложенные во 2-й главе. Это позволяет сделать вывод о работоспособности подхода в целом и создаёт основу для экспериментального его исследования. Система LuNA даёт возможность создавать параллельные программы численного моделирования вообще без необходимости для пользователя параллельного программирования как такового. Процесс разработки параллельной программы сводится к разработке последовательных частей (ФК) и описания схемы требуемой параллельной программы на предметно-ориентированном языке LuNA. При этом параллельная программа будет построена и исполнена автоматически.

На текущем этапе развития системы качество (эффективность) конструируемой программы может быть существенно ниже, чем при ручном параллельном программировании (см. главу 4), но, во-первых, это отчасти компенсируется простотой разработки LuNA программ по сравнению с использованием низкоуровневых средств типа MPI (не требуется собственно параллельно программировать и отлаживать параллельный код). Во-вторых, качество конструируемой программы может быть значительно улучшено применением рекомендаций (см.

результаты в главе 4). Существенно, что применение рекомендаций не нарушает работоспособности программы, поэтому, в частности, разработкой рекомендаций может заниматься отдельный специалист (например, системный параллельный программист) без риска внести ошибку в программу. В-третьих, научным и программистским сообществами постоянно разрабатываются и совершенствуются системные алгоритмы и инструменты анализа, оптимизации и конструирования параллельных программ. Будучи встроенными в систему LuNA (а такая возможность заложена в систему архитектурно) эти алгоритмы и инструменты будут со временем повышать качество конструируемых программ, в том числе для ранее разработанных LuNA-программ (аналогично тому, как это происходит с последовательными программами по мере совершенствования традиционных компиляторов). Существенно, что в качестве ФК могут быть использованы высокоэффективные и хорошо отработанные подпрограммы из накопленного человечеством багажа последовательных подпрограмм. Также в качестве ФК могут быть использованы последовательные подпрограмы, конструируемые автоматически (см., напр., [76]), т.к. в последовательном и многопоточном программировании автоматическое конструирование программ более развито, чем в распределённой памяти.

Таким образом, система LuNA также формирует задел на будущее развитие системы, представляя собой платформу, вместилище для новых системных алгоритмов.

Также необходимо отметить, что высокое качество конструируемых параллельных программ по описанию ФА подразумевает использование специализированных исполнительных систем (см. раздел 3.3.6). Разработанная же система LuNA не является узкоспециализированной, но зато обеспечивает выполнение любого ФА.

Ниже (таблица 3.1) представлена таблица основных действий, составляющих параллельное программирование задач численного моделирования на мультикомпьютерах, и указано, какие виды деятельности система LuNA автоматизирует. Видно, что собственно параллельного программирования как такового при использовании системы LuNA нет, хотя некоторые задачи, связанные именно с параллельным программированием, решать приходится вручную. Например, декомпозиция данных и вычислений на фрагменты данных и вычислений. Выполнение этой задачи требует хотя бы базового понимания того, что такое параллельная программа и как будет проходить её исполнение, и от решения этой задачи существенно зависит эффективность конструируемой программы. Тем не менее, формулировать решение этой задачи программисту приходится не путём разработки соответствующей части параллельной программы на низком уровне, а путём описания построенной декомпозиции на языке LuNA, являющимся предметно-ориентированным языком, более простым в освоении и применении.

Поясним, что отладка параллельного кода после модификации программы в случае системы LuNA не требуется вследствие того, что параллельный код для модифицированной программы перегенерируется автоматически.

Для иллюстрации (таблица 3.1) выбрано сравнение со средствами низкоуровневого программирования в связи с тем, что они наиболее распространены в практике численного моделирования. Многие языки и системы программирования тоже автоматизируют те или иные этапы. Сравнение с ними и обоснование выбора подхода синтеза параллельных программ на вычислительных моделях в качестве основы системы LuNA изложено в главе 1 и во введении в разделе «степень разработанности темы».

Таблица 3.1. Автоматизация разных видов деятельности при параллельном программировании в системе LuNA и с использованием низкоуровневых средств (MPI+OpenMP).

Вид деятельности	**LuNA**	**MPI+OpenMP**
Декомпозиция данных и вычислений	вручную	
Разработка и отладка последовательного кода	вручную	
Разработка CUDA-ядра	вручную	
Программирование коммуникаций	автоматически	вручную
Программирование синхронизации потоков и процессов	автоматически	вручную
Настройка программы на конфигурацию вычислителя/особенности данных	автоматически	вручную
Распределение данных и вычислений по узлам мультикомпьютера	автоматически	вручную
Динамическая балансировка нагрузки на узлы мультикомпьютера	автоматически	вручную
Отладка параллельного кода после модификации программы	не требуется	требуется
Программирование взаимодействия CPU-GPU	автоматически	вручную

4. Экспериментальное исследование

Целью работ, представленных в главе, является экспериментальное определение реальных характеристик системы LuNA и, косвенно, идей и алгоритмов, лежащих в её основе. В последующих разделах представлены результаты разных лет, и система LuNA находилась на разных этапах своего развития, что учитывается при интерпретации результатов. Экспериментальные исследования можно условно разделить на два вида — это специальные тесты, выявляющие конкретную характеристику или особенность системы, и общие тесты, в которых реализовывалась какая-либо реальная задача, чтобы оценить работоспособность и качество работы системы в реальных условиях.

Все эксперименты, представленные в главе, проводились на мультикомпьютере МВС-10П ТОРНАДО Межведомственного суперкомпьютерного центра РАН[9], если явно не оговорено иное. В состав мультикомпьютера входят двухпроцессорные узлы на базе 8-ядерных процессоров Intel Xeon E5-2690 (Sandy Bridge), 2.9 ГГц, с поддержкой технологии HyperThreading (по 2 аппаратных потока на каждое ядро). Пиковая производительность процессора — 185,6 GFlops. Объём оперативной памяти на узле — 64 ГБайт. Коммуникационная сеть — Infiniband FDR.

4.1 Исследование динамической балансировки нагрузки алгоритмом Rope of Beads

Целью эксперимента являлось исследование характеристик алгоритма динамической балансировки нагрузки на вычислительные узлы Rope of Beads на примере задачи моделирования эволюции самогравитирующего пылевого протопланетного диска методом частиц-в-ячейках. Метод частиц-в-ячейках является характерным примером метода, распределённая реализация которого часто требует динамической балансировки нагрузки на узлы мультикомпьютера, т.к. объём вычислений в каждом узле напрямую зависит от количества частиц, приходящихся на домен пространства моделирования, обрабатываемый данным узлом. При этом распределение частиц по доменам изменяется по мере моделирования вообще говоря непредсказуемым образом, вследствие чего статическая балансировка вычислительной нагрузки на узлы мультикомпьютера оказывается неэффективной. Конкретное приложение метода частиц-в-ячейках, использованное в эксперименте, исследовалось в работах С.Е. Киреева [95], который предоставил последовательный код для реализации в системе LuNA и консультировал автора на предмет

[9] http://www.jscc.ru/resources/hpc/

правильности параллельной реализации приложения. Результаты экспериментального исследования были опубликованы в [96].

В исследуемом приложении имеется ограниченное прямоугольным параллелепипедом пространство моделирования, в котором летают модельные частицы, обладающие координатами и массой (рисунок 4.1). Частицы взаимодействуют с гравитационным полем, дискретизуемым на регулярной прямоугольной сетке. Моделирование осуществляется дискретными временными шагами фиксированной величины. На каждом шаге на сетке вычисляется плотность, затем гравитационный потенциал (решается уравнение Пуассона методом Зейделя), затем вычисляются новые координаты и скорости частиц в зависимости от действующих на них сил.

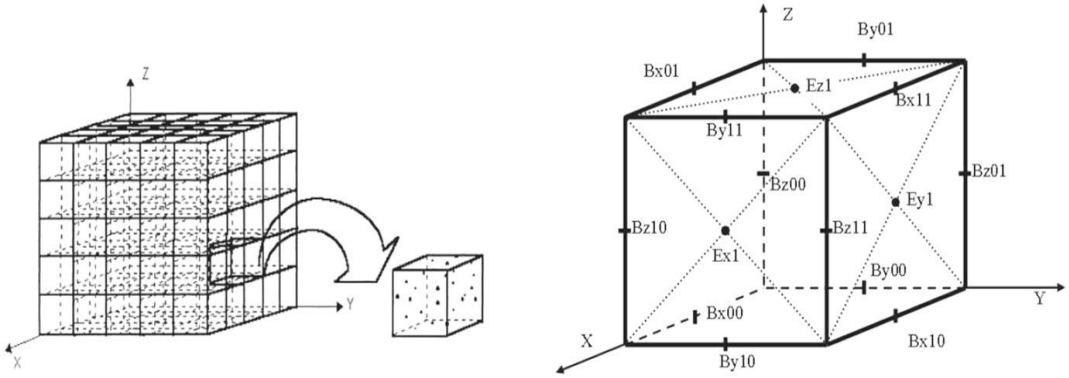

Рисунок 4.1. Структуры данных в приложении метода частиц-в-ячейках.

В проведённом эксперименте пространство моделирования было разбито на домены по трём измерениям с формированием соответствующих фрагментов данных и вычислений. Фрагменты, соответствующие каждому домену, были привязаны к координате в соответствии с алгоритмом Rope of Beads. Изначально каждый вычислительный узел получил равное количество доменов, но распределение вещества в пространстве моделирования было неравномерно — основная часть модельных частиц находилась в центральной области пространства моделирования, в результате чего распределение нагрузки на узлы оказывалось неравномерным. Далее проводился расчёт в двух вариантах — с динамической балансировкой нагрузки алгоритмом Rope of Beads (рисунок 4.3) и без неё (рисунок 4.2). Размер сетки: 64^3, количество частиц: 10^7, количество доменов: 16^3 (по 16 на каждое измерение). Количество временных шагов моделирования: 10. Количество вычислительных узлов: 64. Перечисленные параметры теста являются характерными для вычислительных экспериментов такого рода. Результаты работы LuNA-программы сравнивались на соответствие результатам последовательной программы.

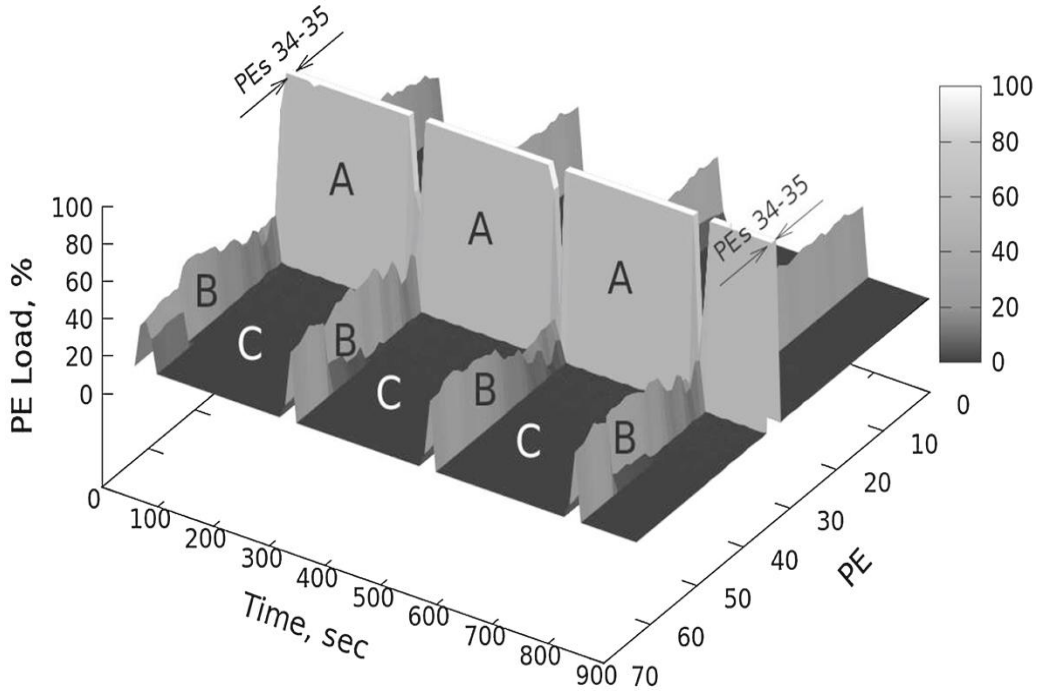

Рисунок 4.2. Зависимость загрузки узлов (PE) от времени. Обозначены области: A — повышенная нагрузка на узлы, на которые приходится основная масса частиц, B — всплески вычислительной нагрузки, связанные с решением уравнения Пуассона, нагрузка распределена равномерно; C — узлы простаивают в отсутствии работы.

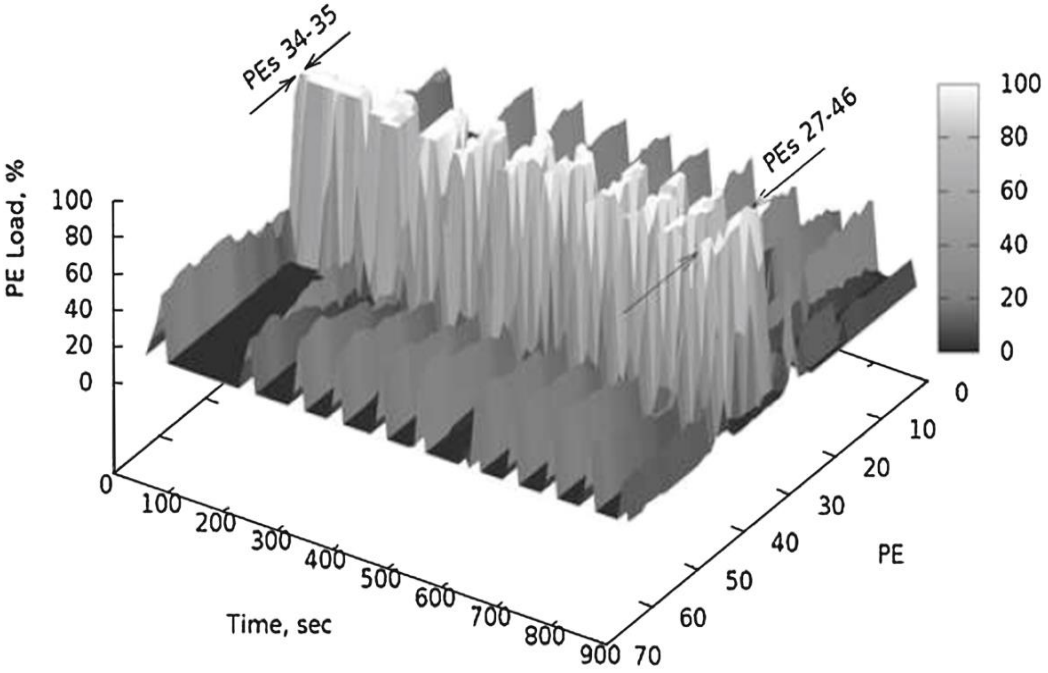

Рисунок 4.3. Зависимость загрузки узлов (PE) от времени при наличии динамической балансировки нагрузки на узлы.

Суммарное время выполнения программы составило 2320 с. и 890 с. для варианта без динамической балансировки нагрузки и с динамической балансировкой соответственно, что

говорит о том, что динамическая балансировка нагрузки оказалась эффективной. По графикам загруженности вычислительных узлов видно, что в варианте 1 большую часть времени загружены лишь несколько узлов, и с течением времени эта картина сохраняется. В варианте 2 с течением времени загрузка распространяется на всё большее количество узлов, от изначально загруженных центральных к периферии, что происходит вследствие работы алгоритма Rope of Beads. Видно также, что каждый следующий шаг модельного времени вычисляется быстрее предыдущего, т.к. нагрузка всё более равномерно распределяется по вычислительным узлам.

Таким образом, динамическая балансировка вычислительной нагрузки была обеспечена автоматически, время проведения расчёта сократилось примерно в 2,6 раза.

В этой работе [96] идея реализации динамической балансировки вычислительной нагрузки принадлежит В.Э. Малышкину, разработка и реализация алгоритмов и экспериментальное исследование выполнено автором, обсуждение алгоритмов и результатов выполнялось совместно.

4.2 Исследование применимости системы LuNA к решению задач численного моделирования

Целью экспериментальных исследований, представленных в подразделе, являлось исследование применимости системы LuNA в целом для решения практических задач численного моделирования, а также исследование характеристик производительности программ, конструируемых системой LuNA в реальных условиях в сравнении с ручной реализацией тех же прикладных алгоритмов стандартными средствами (MPI). Представленные ниже эксперименты были получены в результате совместных работ с соавторами из Казахского Национального Университета (г. Алма-Ата). Специалистами из Казахстана решались их прикладные задачи, связанные с численным моделированием процесса фильтрации многофазных жидкостей в задачах нефтедобычи. При этом они самостоятельно разрабатывали программы решения этих задач на базе MPI и на базе системы LuNA. Задачей автора являлось обеспечение работоспособности системы LuNA в реальных условиях, обучение коллег использованию системы и их консультирование в процессе разработки ими LuNA-программ. Также в задачах, представленных в разделах 4.2.1 и 4.2.2, автором выполнялась настройка поведения LuNA-программ с помощью рекомендаций. Правильность всех программ с точки зрения вычисляемых результатов контролировалась коллегами из Казахстана.

Отметим, что содержательная составляющая численного моделирования не имеет принципиального значения с точки зрения темы работы и выходит как за её рамки, так и за рамки

компетенции автора (эта компетенция обеспечивалась Казахстанской стороной). В связи с этим ниже приводятся только ключевые особенности решаемых задач, значимые с точки зрения их параллельной реализации и системы LuNA. Математические постановки задач, методы их решения и другие особенности численного моделирования могут быть найдены в соответствующих публикациях.

4.2.1 Моделирование фильтрации трёхфазной жидкости в системе «нефть — вода — газ»

В подразделе излагается информация о серии работ, связанных с реализацией численных моделей для задачи фильтрации многофазной жидкости в различных постановках. В работах [97, 98] рассматривалась одномерная краевая задача фильтрации для системы нефть-вода. В [99] решалась двумерная краевая задача фильтрации для системы нефть-вода-газ, а в работе [100] — эта же задача для трёхмерного случая. Во всех работах сравнивалось время выполнения LuNA-программы и MPI-программы. Рассмотрим эти результаты.

Одномерный случай. Решается одномерная краевая задача фильтрации для системы «нефть — вода» [97, 98]. Задача решается итерационно, на каждой итерации применяется метод прогонки. Параллельная реализация выполняется методом пространственной декомпозиции. При этом решение в подобластях (доменах) находится параллельно методом прогонки, а затем решение на границах подобластей находится последовательно, на одном из узлов.

Численный алгоритм был реализован в четырёх вариантах: последовательная программа на языке Java, параллельная программа на языке Java с использованием коммуникационной библиотеки MPJ Express (реализация стандарта MPI для языка Java), последовательная фрагментированная программа на языке C++ и LuNA-программа с фрагментами кода на C++ (см. рисунок 4.4). Параллельная версия выполнялась на 4-ядерном компьютере, LuNA-программа - на 32 узлах мультикомпьютера, по одному ядру на узел. Размер сетки — от 50 до 5100 узлов, число временных шагов — $1{,}5\times10^5$, каждый временной шаг состоит из примерно 10 итераций. Результаты эксперимента представлены на рисунке 4.4.

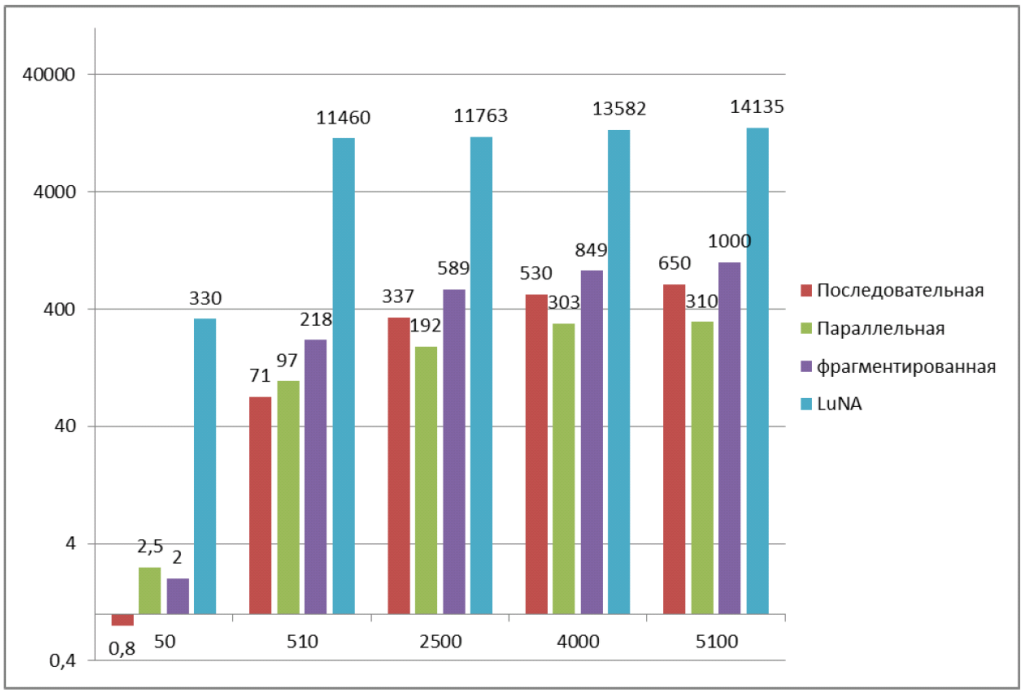

Рисунок 4.4. Время выполнения программы (с) в зависимости от размера сетки.

Параметры тестирования LuNA-программы и остальных программ существенно отличаются (распределённый и многоядерный вычислитель соответственно), что необходимо учитывать при анализе. LuNA-программа выполняется медленнее, чем другие примерно на 1,5 порядка. Причины этого заключаются в том, что, во-первых, задача является одномерной, объём данных относительно небольшой, и критической подсистемой является сеть (а распределённо исполнялась только LuNA-программа). Во-вторых, система LuNA имеет накладные расходы, связанные с каждым фрагментом. Учитывая, что объём фрагментов мал (т.к. мал объём данных и вычислений в задаче), доля накладных расходов может быть велика. Тем не менее, LuNA-программа успешно отработала и выдала верный результат.

Двумерный случай. Решается двумерная краевая задача фильтрации трёхфазной жидкости для системы нефть—вода—газ (рисунок 4.5).

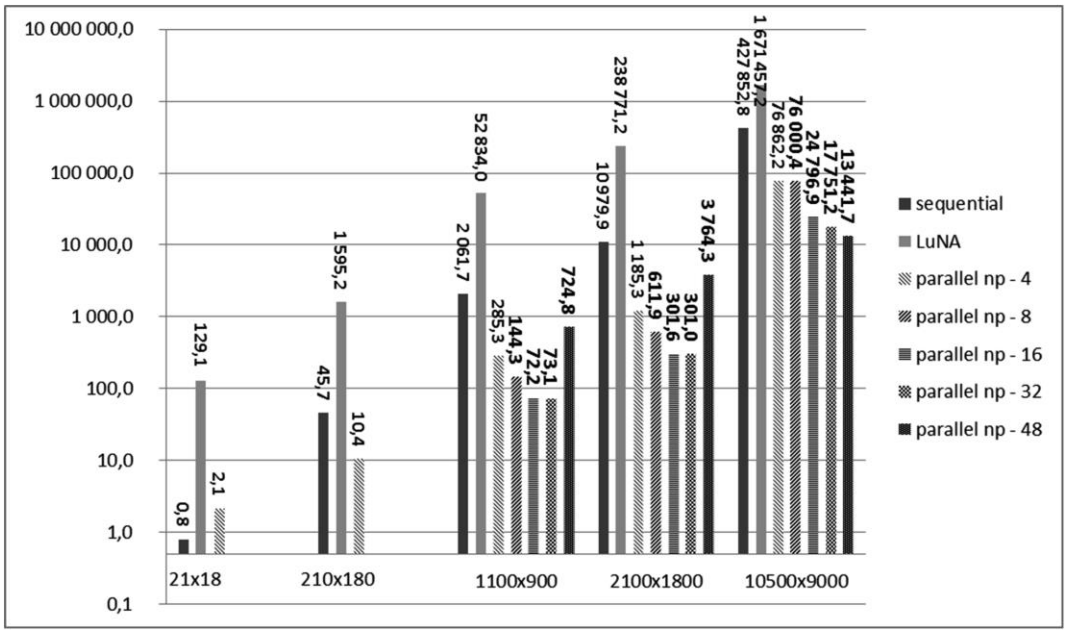

Рисунок 4.5. Зависимость времени выполнения программы (с) от параметров задачи.

Решается задача моделирования процесса фильтрации трёхфазной жидкости в системе «нефть — вода — газ» [99, 100]. Процесс описывается системой дифференциальных уравнений, которая решается итерационным методом, неявным по давлению и явным по насыщенности. Каждый временной шаг дробится на три промежуточных шага. Для вычислений используется трёхмерная сетка, которая реализуется распределённо методом пространственной декомпозиции по одному измерению, причём на каждом втором промежуточном шаге измерение, по которому выполняется декомпозиция, изменяется (см. рисунок 4.5). Основная доля вычислений приходится на решение СЛАУ методом прогонки.

В следующем эксперименте (рисунок 4.6) сравнение выполнялось для четырёх версий программы — последовательной (sequential), параллельной программы, написанной вручную (MPI), LuNA-программы (LuNA) и LuNA-программы, в которой распределение фрагментов по узлам и управление были заданы вручную (LuNA-fw). Для реализации последнего варианта использовалась специализированная исполнительная система, также разработанная автором, но выходящая за рамки настоящей работы.

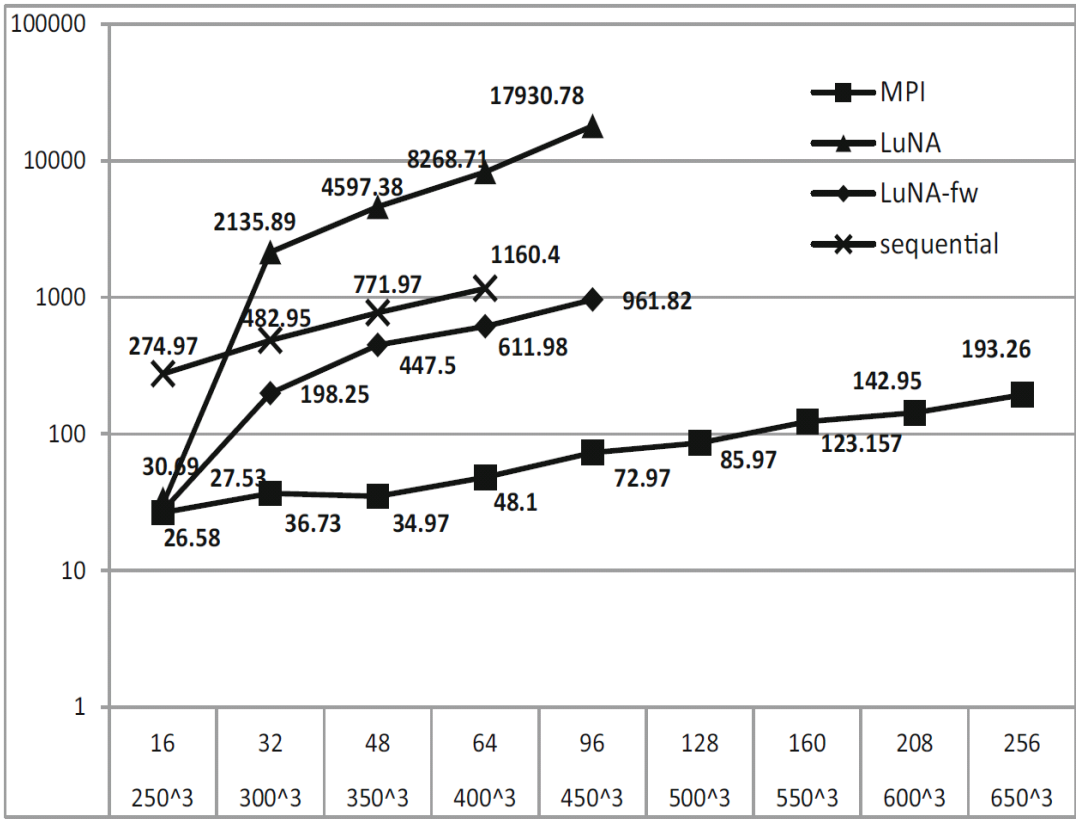

Рисунок 4.6. Зависимость времени выполнения (сек.) четырёх версий программы от размера сетки и количества ядер мультикомпьютера.

По результатам эксперимента видно, что система LuNA успешно справилась с исполнением ФА, хотя производительность LuNA-программ существенно уступает ручной реализации (MPI). Разница в производительности объясняется малым размером задачи (и, соответственно, относительно большим количеством накладных расходов на работу исполнительной системы). Кроме того, в этом эксперименте использовалась одна из предыдущих версий[10] системы LuNA. Эксперимент также показывает, что производительность LuNA-программы может быть существенно улучшена за счёт ручного управления её поведением посредством рекомендаций. Также эксперимент подтверждает тот факт, что настройка поведения LuNA-программы может выполняться отдельным лицом, не являющимся специалистом в предметной области, без риска нарушить правильность работы программы (настройка поведения осуществлялась автором).

[10] По мере проработки темы совершенствовались и системные алгоритмы LuNA, и их программная реализация. См. раздел 4.2.3 для сравнения эффективности этой и текущей версии системы на аналогичной задаче.

4.2.2 Решение модельного уравнения теплопроводности в единичном кубе

В эксперименте исследовались характеристики производительности системы LuNA при решении модельной задачи — уравнения теплопроводности в единичном кубе. На этой модельной задачи исследовались свойства решателя для задачи, описанной в п. 4.2.1. для случая трёхмерной декомпозиции трёхмерной сетки.

Для исследования решателя в трёхмерной области была выбрана модельная задача решения уравнения теплопроводности в единичном кубе [101, 102]. В данном приложении используется трёхмерная регулярная прямоугольная сетка с декомпозицией на домены по трём измерениям. Вычислительный процесс — итерационный, где на каждой итерации решение сводится к решению трёхдиагональной СЛАУ методом прогонки (по каждому из трёх измерений сетки). Метод прогонки был реализован распределённо конвейерным методом [103]. Размер сетки варьировался от 100^3 до 1000^3, количество узлов мультикомпьютера — от 8 (2^3) до 216 (6^3).

На рисунке 4.7 приведены результаты измерения времени выполнения программы в зависимости от параметров запуска. Время выполнения приведено для трёх вариантов программы: MPI (программа, разработанная вручную), LI — LuNA Interpreter, одна из предыдущих версий системы LuNA и LC — LuNA Compiler[11], текущая версия системы, описанная в настоящей работе. Сравнение с предыдущей версией системы показывает положительную динамику улучшения производительности системы LuNA по мере совершенствования её системных алгоритмов и их программной реализации.

[11] Названия «interpreter» и «compiler» отражают тот аспект, что в версии «compiler» применяется мультиагентный подход (раздел 3.2.2), где программы агентов компилируются, а не интерпретируются.

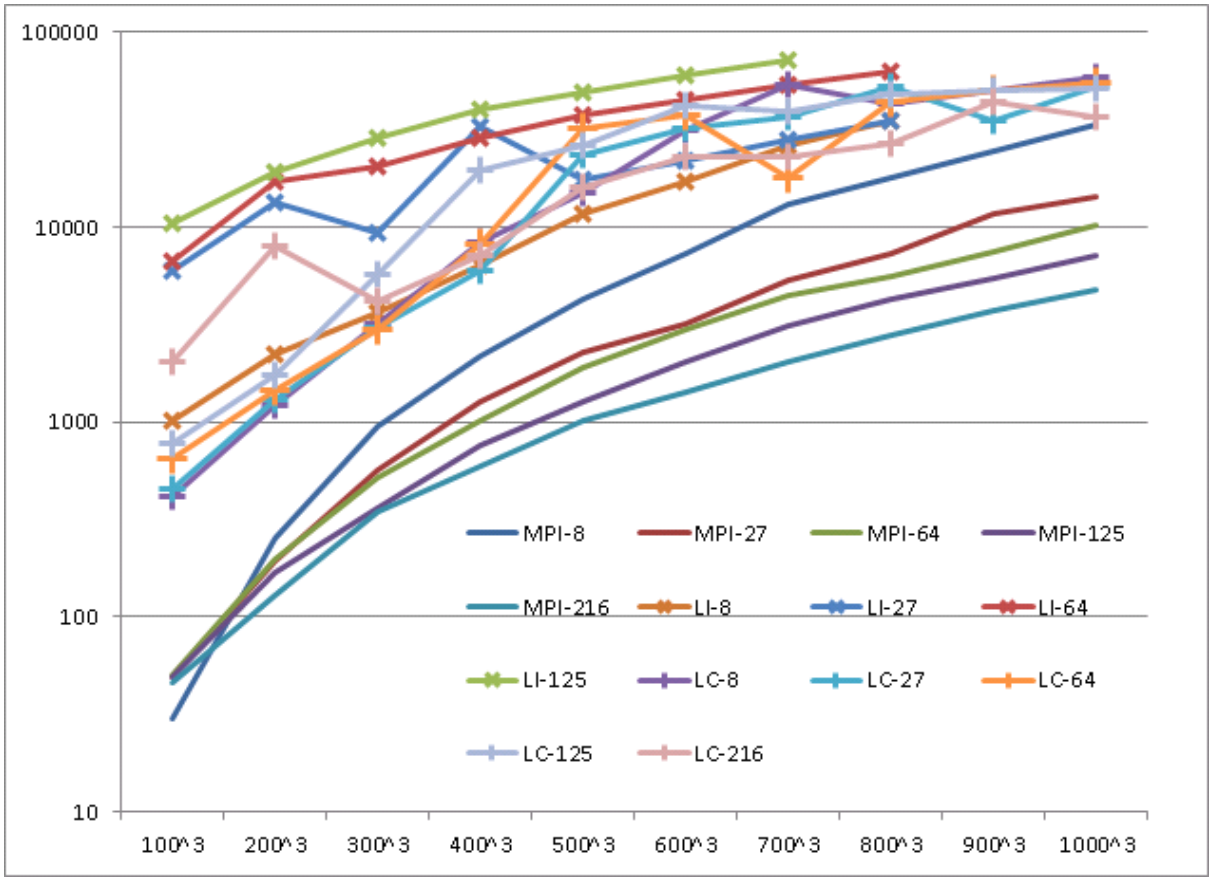

Рисунок 4.7. Зависимость времени выполнения различных версий программы (c) от количества узлов. MPI — ручная реализация, LI (LuNA Interpreter) — предыдущая версия системы, LC (LuNA Compiler) — текущая версия системы.

По результатам эксперимента видно, что время выполнения LuNA-программы существенно больше, чем время выполнения MPI-программы. Но даже несмотря на то, что речь идёт о высокопроизводительных вычислениях такое отставание (на порядок и менее) уже не является критичным по следующим причинам. Во-первых, разработка, отладка и модификация LuNA-программы существенно проще, чем MPI-программы (см. раздел 3.5), поэтому имеет место компромисс между временем разработки и временем вычислений. Во-вторых, по мере дальнейшей проработки системы LuNA качество конструируемых программ будет улучшаться без изменения самих прикладных LuNA-программ. Примером могут служить варианты LI и LC, где LuNA-программа одна и та же, но для их исполнения используются разные версии системы LuNA. Принципиальная достижимость производительности конструируемых системой LuNA программ, сравнимой с производительностью ручных программ, показана, например, в работах [90, 91, 104]. Это зависит, в первую очередь, от того, насколько тот или иной класс прикладных алгоритмов поддержан в системе специализированными системными алгоритмами. Разработка

таких алгоритмов для конкретных предметных областей является отдельной проблемой и выходит за рамки настоящего исследования.

4.2.3 Анализ результатов

Выполненные экспериментальные исследования позволяют сделать вывод о том, что система LuNA оказалась практически пригодна для решения реальных задач численного моделирования. Существенно, что разработкой LuNA-программ и применением системы LuNA во всех трёх случаях занимались не разработчики системы LuNA, а прикладные программисты — специалисты в своей предметной области. Аналогичным образом была выполнена работа [105], где исследовалась производительности системы LuNA в сравнении с разработанной вручную MPI-программой на примере решения двумерного эллиптического уравнения для модельной задачи.

Также исследования показали, что качество конструируемых программ ниже, чем программ, разработанных вручную на основе MPI, что существенно, т.к. речь идёт о высокопроизводительных вычислениях. С учётом теоретического анализа системы LuNA и её алгоритмов, выполненных в предыдущих главах, можно утверждать, что данное обстоятельство не является принципиальным ограничением системы, а отражает текущее состояние реализации системы. А именно, отсутствие специализированной поддержки системными алгоритмами приложений такого класса, а также качество программного кода, реализующего имеющиеся системные алгоритмы. По мере разработки новых системных алгоритмов, а также по мере оптимизации программного кода исполнительной системы качество конструируемых программ будет возрастать без необходимости изменения существующих LuNA-программ. В частности, по публикациям разных лет видно тенденцию к улучшению качества конструируемых системой LuNA-программ. Также на практике было подтверждено, что разработка и отладка LuNA-программ проще, чем MPI-программ ввиду отсутствия необходимости низкоуровневого параллельного программирования (подробнее это изложено в разделе 3.5). Т.е., имеет место компромисс между временем разработки, отладки и модификации программы и временем выполнения конструируемой программы, поэтому даже с текущим уровнем обеспечиваемой производительности конструируемых программ применение системы LuNA на практике представляется оправданным во многих случаях.

4.3 Повышение эффективности исполнения LuNA-программ на основе профилирования

Целью эксперимента является исследование характеристик производительности LuNA-программ для случая, когда отображение фрагментов на узлы мультикомпьютера конструируется автоматически на основе профилирования в соответствии с алгоритмом, описанным в разделе 2.4.6.

Работа алгоритма была протестирована [104] на задаче умножения плотных матриц . Тестирование проводилось на мультикомпьютере НКС-30Т Сибирского суперкомпьютерного центра[12] (г. Новосибирск). Расчёт проводился на вычислительных узлах на базе двойных блейд-серверов HP BL2x220c (CPU (4 x) 6-ядерный Intel Xeon X5670, 2.93 GHz (Westmere), RAM 24 Гбайт). Для сравнения была выбрана реализация той же задачи в MPI, основанная на алгоритме Кэннона [106]. В узле использовался блочный алгоритм умножения матриц. В LuNA-программе в качестве фрагментов кода использовалась та же процедура блочного умножения матриц, что и в MPI-варианте. Таким образом, MPI-реализацию можно условно считать теоретически достижимым пределом по времени работы для LuNA-программы. На рисунке 4.8 изображена зависимость времени работы программы для последовательности запусков. Тут первый запуск выполнялся со случайным распределением фрагментов вычислений по узлам, а каждый следующий запуск получался из предыдущего в соответствии с исследуемым алгоритмом.

Видно, что время выполнения LuNA-программы уменьшается, что подтверждает способность исследуемого алгоритма повышать качество конструируемых системой LuNA-программ на основе трассировки по крайней мере для некоторых классов прикладных задач. В данном случае было достигнуто время выполнения MPI-программы, что является хорошим результатом. Это можно объяснить тем, что задача умножения матриц относится к классу критичных по вычислениям (большой объём вычислений на единицу данных) и характеризуется массовым параллелизмом. Также этот эксперимент подтверждает, что в принципе исполнение LuNA-программ может быть сравнимым по эффективности с программами, разработанными вручную на базе низкоуровневых средств параллельного программирования.

[12] http://www.sscc.icmmg.nsc.ru/

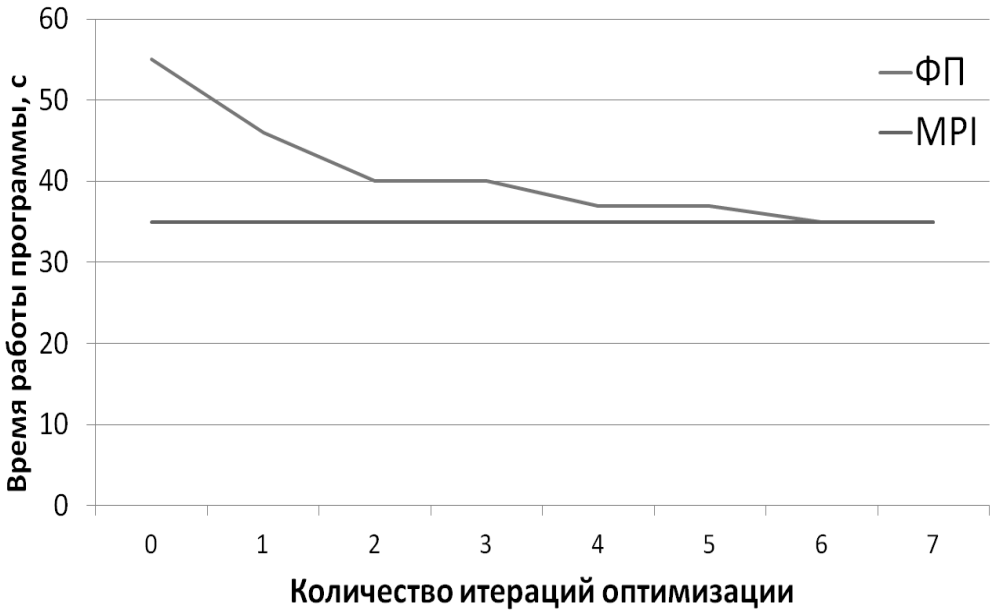

Рисунок 4.8. Зависимость времени выполнения программы (с) от номера запуска. ФП — фрагментированная программа, MPI — версия, разработанная вручную.

4.4 Применение специализированной исполнительной системы

Целью эксперимента являлось исследование возможности применения специализированных системных алгоритмов поддержки исполнения LuNA-программ частного вида. В разделе кратко приводятся результаты трёх работ, две из них совместные с автором [88, 90], роль которого заключалась в том, чтобы обеспечить встраивание частных исполнительных систем в систему LuNA, а сами частные системы были разработаны соавторами. Также в работе [91] Н.А. Беляевым была разработана ещё одна частная исполнительная система, предназначенная для встраивания в систему LuNA и основанная на её модели.

В работе [88] А.А. Ткачёвой был разработан алгоритм «монолитизации», суть которого заключается в том, что группа ФВ объединяется в один укрупнённый ФВ, последовательно выполняющий внутри себя все объединённые им ФВ. Это позволяет снизить накладные расходы исполнительной системы за счёт сокращения общего количества ФВ и введения императивного управления на множестве объединённых ФВ. При этом, возможно, часть информационно независимых ФВ, становятся зависимыми по управлению, что может привести к уменьшению максимального количества одновременно исполняемых ФВ.

Экспериментальное исследование проводилось на мультипроцессоре на базе 6-ядерного процессора Intel Xeon X5600 с тактовой частотой 2.8 ГГц, а также на мультикомпьютере

Новосибирского Государственного Университета[13]. В качестве прикладной задачи была выбрана операция редукции фрагментированного массива чисел как часто встречающаяся на практике операция. При этом массив фрагментов разбивается на части, каждый из которых обрабатывается последовательно или параллельно, а разные части могут обрабатываться параллельно. Оба варианта были протестированы с использованием алгоритма «монолитизации» и без него (рисунок 4.9).

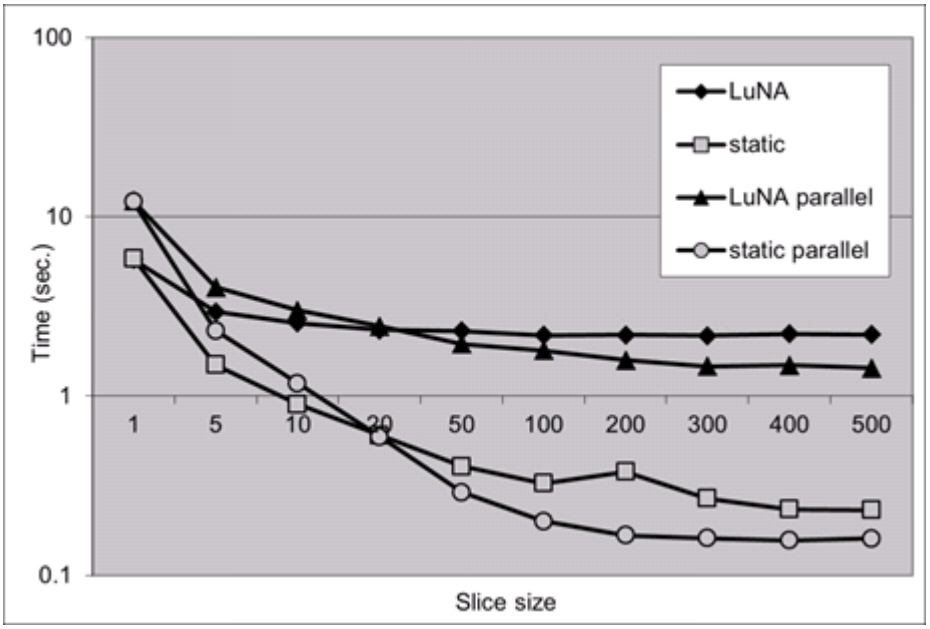

Рисунок 4.9. Зависимость времени выполнения ФА (с) от размера группы фрагментов, тест на системе с общей памятью, размер массива — 3×10^4.

В следующем эксперименте (рисунок 4.10) аналогичный эксперимент выполнялся на мультикомпьютере, где также была показана выгода от использования частной исполнительной системы по сравнению с базовой исполнительной системой LuNA.

Частные алгоритмы поддерживали, в том числе, динамическую балансировку вычислительной нагрузки, что демонстрирует тот факт, что частные системные алгоритмы и конструируемые ими программы могут быть сложными. В этой работе потребовалось снабдить LuNA-программу дополнительной информацией о том, какие из индексов ФД отвечают за временную, а какие — за пространственные координаты. Это вписывается в подход, применяемый в системе LuNA. Также эта работа показательна в том отношении, что и система LuNA, и частное решение по эффективности оказались сопоставимыми с ручной реализацией той же задачи средствами MPI, что говорит о том, что даже без специализированной поддержки данного класса приложений система LuNA вполне применима на практике.

[13] http://nusc.nsu.ru/

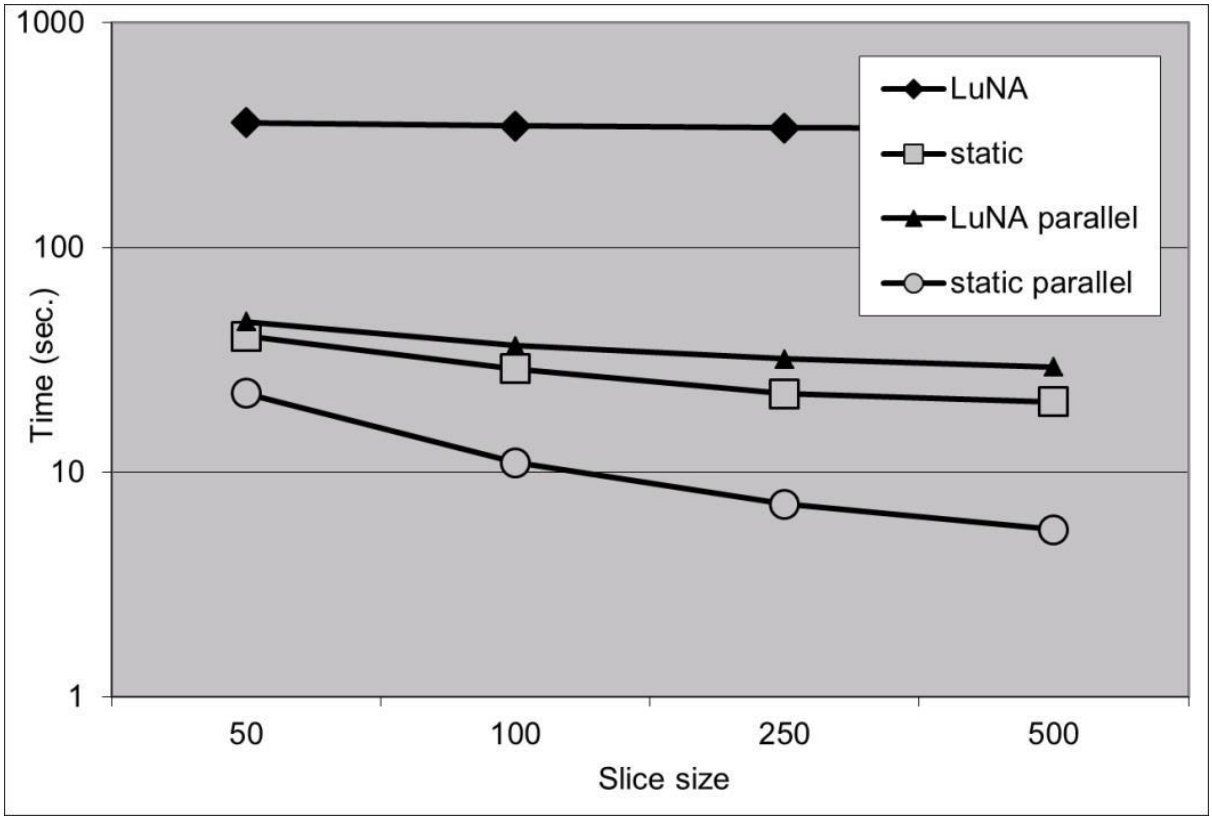

Рисунок 4.10. Зависимость времени выполнения ФА (c) от размера группы фрагментов, тест на мультикомпьютере. Размер массива — 5×10^5. Количество вычислительных узлов: 8.

В работе [90] Н.А. Беляевым была предложена и реализована исполнительная система для LuNA-программ частного вида — а именно, класса плотных матрично-векторных операций, среди которых разложения LU, LL^T, LDL^T. Экспериментальное исследование показывает, что LuNA-программа разложения Холецкого, исполненная предложенной исполнительной системой, показывает более высокую эффективность, чем решение, входящее в распространённую профильную библиотеку ScaLAPACK (см. рисунок 4.11). Тестирование выполнялось на 4 узлах мультикомпьютера для разложения матриц размером 32768^2 и 65536^2.

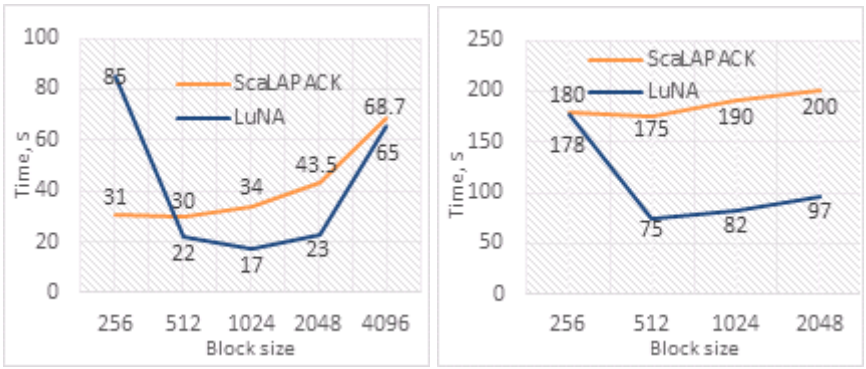

Рисунок 4.11. Зависимость времени выполнения LuNA-программы и реализации ScaLAPACK для различных размеров блока. Размер матрицы: 32768 (слева) и 65536 (справа).

Результаты, представленные в подразделе, подтверждают, что система LuNA вполне подходит для включения в неё частных системных алгоритмов конструирования и исполнения параллельных программ, обеспечивающих высокую эффективность конструируемых программ для отдельных классов приложений.

4.5 Автоматизация использования GPU для реализации ФВ

Целью эксперимента являлось исследование временных характеристик исполнения LuNA-программ при автоматизированной поддержке применения графических ускорителей для реализации отдельных ФВ (см. раздел 3.4.4). Поддержка исполнения ФВ на GPU была реализована в составе системы LuNA Н.А. Беляевым. Автором настоящей работы была обеспечена организация (проектирование) системы LuNA таким образом, чтобы автоматизированная поддержка GPU была возможной.

Суть автоматизации применения GPU для реализации ФВ состоит в том, что для некоторых ФК пользователь может предоставлять вместо обычной C++-процедуры (или вместе с ней) CUDA-ядро, реализующее ту же функцию. Система LuNA, как следствие, может реализовать соответствующие ФВ на GPU. Для этого входные ФД должны быть переданы из ОЗУ в память GPU, а выходные ФД — переданы обратно, либо использованы в качестве входных ФД для других ФВ на GPU. Также Н.А. Беляевым был разработан планировщик, который принимал решение о том, какие ФВ запускать на GPU, а какие — на CPU, и в каком порядке. Подробности этой работы опубликованы в [107].

Все тесты проводились на мультипроцессоре на базе двух процессоров Xeon 5670 (3 GHz), оснащённом графическим ускорителем Nvidia Tesla M 2090. В качестве прикладной задачи была использована модельная явная конечно-разностная схема, где объём вычислений на единицу данных являлся параметром (LOAD на рисунке 4.12). При этом доля ФВ, исполняемых на GPU и CPU была параметром. По результатам тестирования видно, что, во-первых, минимальное время достигается при совместном использовании GPU и CPU, и, во-вторых, оптимальная доля вычислений, приходящихся на GPU, зависит от параметра LOAD.

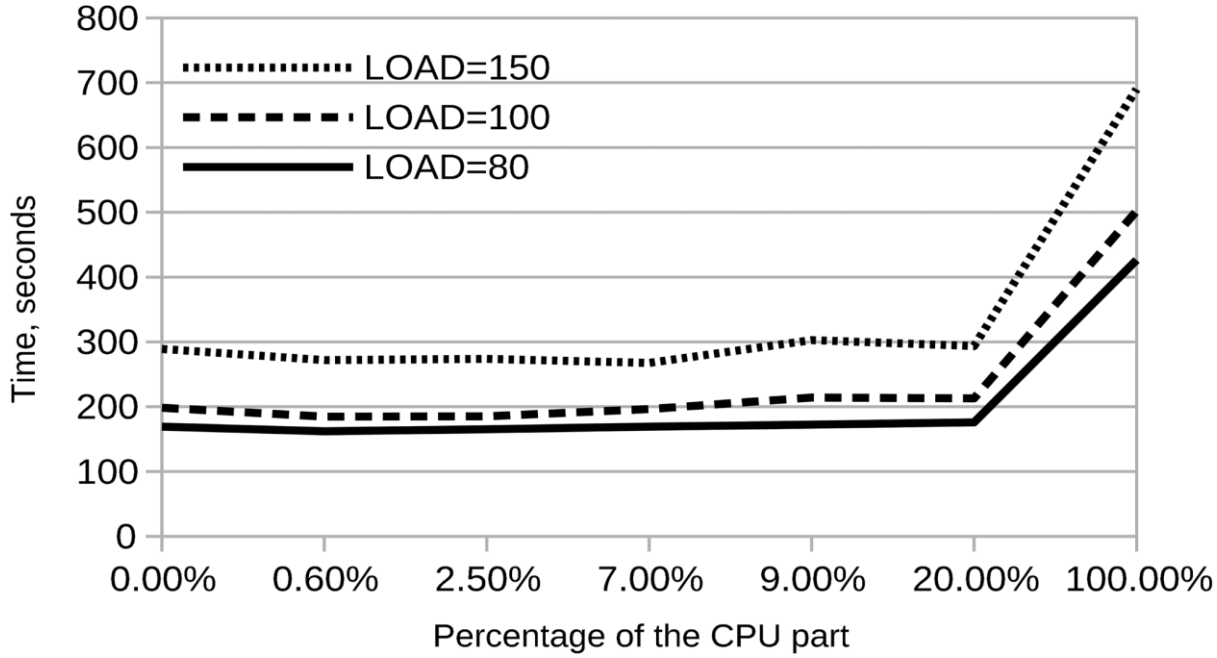

Рисунок 4.12. Зависимость времени выполнения программы от доли вычислений, приходящихся на CPU.

Был также проведён эксперимент для случая, когда объём данных, приходящихся на GPU, превышал объём памяти GPU, поэтому по крайней мере часть этих данных должна была постепенно «протаскиваться» через память GPU (доля этих данных — параметр drag-through на рисунке 4.13).

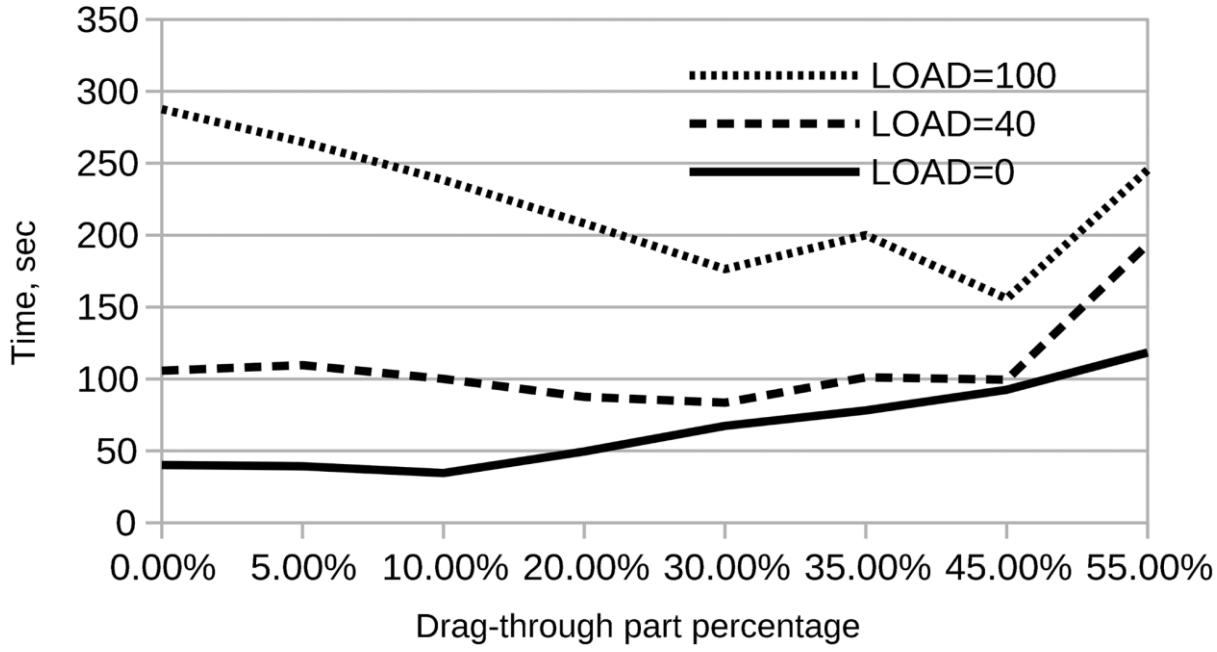

Рисунок 4.13. Зависимость времени выполнения программы от параметра drag-through для различных значений параметра LOAD.

Тестирование показало, что минимальное время для различных параметров LOAD достигается при различном значении параметра drag-through.

В совокупности тесты показывают, что автоматизация применения GPU в исполнении LuNA программ возможна, при этом с пользователя снимается значительная часть рутинной работы, связанной с передачей данных между памятями центрального процессора и графического ускорителя, синхронизацией работы CPU и GPU над отдельными подзадачами, а также диспетчеризацией вычислений по CPU и GPU. Особо отметим, что наиболее эффективным оказался вариант с совместным использованием и CPU, и GPU, причём часть данных находилась всё время на GPU, а часть — временно загружалась на каждой итерации прикладного алгоритма. Реализация такой схемы вручную слишком сложна для среднего прикладного программиста, а LuNA обеспечивает такое исполнение автоматически.

4.6 Воспроизведение трасс

В работе [108] автором диссертации был реализован алгоритм воспроизведения трасс (см. раздел 2.4.8) в соответствии с особенностями, изложенными в разделе 3.4.5. В.Э. Малышкиным в этой работе выполнялось обсуждение постановки задачи и предлагаемых алгоритмов. Экспериментальное исследование проводилось на том же приложении метода частиц-в-ячейках, что описано в разделе 4.1. Эффективность сравнивалась для трёх реализаций одного и того же прикладного алгоритма — LuNA-программа, реализуемая базовой исполнительной системой LuNA, воспроизведение трассы и ручная реализация на базе MPI.

Экспериментальное исследование (таблица 4.1) показывает, что воспроизведение трасс существенно снижает время выполнения LuNA-программы, обеспечивая эффективность, сравнимую с эффективностью MPI-программы. Это подтверждает применимость подхода воспроизведения трасс по крайней мере для некоторого класса приложений.

4.7 Анализ результатов экспериментальных исследований

Выполненное экспериментальное исследование показывает, что язык LuNA позволяет описывать различные прикладные алгоритмы, и что система LuNA по такому описанию обеспечивает автоматическое конструирование параллельной программы, реализующей заданный алгоритм на вычислителях с распределённой памятью. При этом собственно параллельно программировать от пользователя системы не требуется, все вопросы организации распределённой обработки данных, сетевых коммуникаций, синхронизаций потоков и других низкоуровневых задач (включая взаимодействие с GPU) система берёт на себя. Вычисления

действительно осуществляются параллельно, настраиваются на доступные вычислительные ресурсы, динамическая балансировка нагрузки на вычислительные узлы выполняется автоматически.

Таблица 4.1. Результаты экспериментального исследования.

Parameters			**Execution time (sec.)**		
Mesh size	**Particles**	**Cores**	**MPI**	**LuNA-TB**	**LuNA**
100^3	10^6	64	5.287	13.69	355.5
150^3	10^6	64	18.896	31.088	732.833
150^3	10^7	64	23.594	111.194	2983
150^3	10^7	125	23.352	118.32	3086.411
150^3	10^6	343	33.697	39.651	1008.655

Успешный опыт разработки приложений другими пользователями, не являющихся разработчиками системы LuNA, показывает определённую зрелость языка и системы. Помимо представленных в разделе результатов система LuNA была успешно применена в некоторых других работах (напр., в [109]).

Показано, что исполнение фрагментированных алгоритмов может осуществляться на базе узкоспециализированных частных исполнительных систем, в том числе для отдельных частей фрагментированного алгоритма совместно с работой основной исполнительной системы. Показана возможность автоматической настройки поведения фрагментированных алгоритмов на конкретный вычислитель на основе анализа профиля предыдущих реализаций фрагментированного алгоритма.

Важно отметить, что выполненные исследования не являются исчерпывающими. Факторы, существенно влияющие на нефункциональные свойства, многообразны, поэтому экспериментальные результаты являются приблизительными, но, тем не менее, информативными.

Эффективность конструируемых программ в сравнении с реализацией тех же алгоритмов вручную в большинстве случаев ниже (в отдельных — выше). В среднем наблюдается замедление в пределах одного порядка. Несмотря на то, что эффективность параллельной программы является одной из важнейших характеристик в области высокопроизводительных вычислений такое отставание в эффективности на текущем этапе развития системы является, по мнению автора, нормальным по следующим причинам. **Во-первых,** интерпретация как способ реализации языка программирования неизбежно будет обладать накладными расходами, но это компенсируется возможностью автоматического обеспечения динамических свойств, таких как динамическая балансировка нагрузки на узлы, обеспечение отказоустойчивости и пр. **Во-вторых,** по мере совершенствования системных алгоритмов, эффективность конструируемых программ будет повышаться так же, как это произошло с компиляторами традиционных языков программирования. По хронологии публикаций проекта LuNA (большей частью представленной в разделе 4) также видно, что эффективность конструируемых программ со временем повышается. За годы развития проекта большое количество усилий было вложено в улучшение производительности. **В-третьих,** будучи универсальной, исполнительная система будет, как правило, проигрывать по эффективности частным исполнительным системам и алгоритмам трансляции, и по мере накопления таких частных систем и алгоритмов в составе системы LuNA всё большая часть фрагментированных алгоритмов будет исполняться узкоспециализированным эффективным образом, в то время как универсальная исполнительная система будет доделывать те части фрагментированного алгоритма, для которых не нашлось специальной реализации. Это можно сравнить с тем, как в языке Python, удобном в прототипировании, но интерпретируемом и медленном (по сравнению, например, с C++) удобно проводить высокопроизводительные расчёты за счёт наличия большого количества высокоэффективных прикладных библиотек.

Таким образом, можно сделать вывод о том, что экспериментальное исследование системы LuNA подтверждает выводы теоретического анализа, демонстрирует пригодность системы к практическому использованию и характеризует текущий уровень эффективности конструируемых программ в сравнении с ручным программированием.

Заключение

Главным итогом работы является апробация исследуемого подхода к автоматическому конструированию параллельных программ на базе теории структурного синтеза параллельных программ на вычислительных моделях частного вида. Для этого разработаны необходимые системные алгоритмы, они реализованы в виде экспериментальной системы LuNA, и на базе этой системы исследованы реализации ряда модельных и реальных приложений. Теоретический анализ и экспериментальное исследование позволяют качественно и количественно судить о сильных и слабых сторонах подхода и предложенных алгоритмов. Создана основа для дальнейшего накопления и исследования системных алгоритмов конструирования и исполнения параллельных программ численного моделирования.

В результате диссертационного исследования были получены следующие основные результаты:

1. Предложена модель фрагментированного алгоритма, позволяющая представлять алгоритм в виде, допускающем автоматическое конструирование параллельных программ, реализующих данный алгоритм на распределённых вычислителях (мультикомпьютерах).
2. Разработан язык LuNA описания фрагментированных алгоритмов.
3. Разработаны системные алгоритмы, обеспечивающие конструирование и исполнение распределённых параллельных программ по заданному фрагментированному алгоритму.
4. Разработанные системные алгоритмы реализованы в виде экспериментальной системы LuNA автоматического конструирования и исполнения параллельных программ, реализующих фрагментированный алгоритм по его описанию на языке LuNA.
5. Выполнено экспериментальное исследование характеристик системы LuNA на ряде приложений.

Таким образом, цели диссертационного исследования достигнуты в полной мере.

Полученные результаты создают задел для дальнейшего исследования темы в нескольких направлениях. Во-первых, это разработка новых системных алгоритмов управления исполнением ФА для обеспечения более высокого качества конструируемых программ в разных предметных областях. Во-вторых, это экспериментальное исследование на базе системы LuNA нефункциональных характеристик различных системных алгоритмов распределения ресурсов, планирования вычислений, динамической балансировки нагрузки на вычислительные узлы, распределённой сборки мусора и пр. В-третьих, система LuNA может быть использована как основа для дальнейшего перехода к реализации идей структурного синтеза параллельных

программ на вычислительных моделях, в частности, исследование возможностей синтеза на вычислительных моделях алгоритмов решения задачи с заданными нефункциональными свойствами.

Список сокращений и условных обозначений

ИС — исполнительная система

мультикомпьютер — параллельный вычислитель с распределённой памятью; узлами мультикомпьютера могут быть мультипроцессоры

мультипроцессор — параллельный вычислитель с общей памятью

ПИ — пространство имён

ФА — фрагментированный алгоритм

ФВ — фрагмент вычислений

ФД — фрагмент данных

ФК — фрагмент кода

GPU — Graphics Processing Unit

MPI — Message Passing Interface

Список литературы

1. **Янов, Ю.И.** Метод свёрток для разрешения свойств формальных систем. — М. : ИПМ им. М.В.Келдыша, 1977. — Вып. 11. — 41 с. — (Институт прикладной математики АН СССР. Препринт; № 11 за 1977 г.). — URL: https://library.keldysh.ru/preprint.asp?id=1977-11.
2. **Янов Ю.И.** О логических схемах алгоритмов // Проблемы кибернетики. — 1958. — Вып. 1. — С. 75–127.
3. **Котов В.Е.** О практической реализации асинхронных параллельных вычислений // Системное и теоретическое программирование. — Новосибирск: ВЦ СО АН СССР, 1972. — С. 110–125.
4. **Котов В.Е.** МАРС: архитектуры и языки для реализации параллелизма // Системная информатика ; Под ред. В.Е. Котова. — Новосибирск : Наука. Сиб. отд-ние, 1991. — Вып 1 : Проблемы современного программирования. — С. 174–194.
5. **Котов В. Е., Сабельфельд В. К.** Теория схем программ. — М. : Наука, 1991.
6. **Вальковский В. А.** Распараллеливание алгоритмов и программ. Структурный подход. — М. : Радио и связь, 1989. — 176 с.

7. **Малышкин В. Э.** ОПАЛ — язык описания параллельных алгоритмов / В кн. Теоретические вопросы параллельного программирования и многопроцессорные ЭВМ : сборник научных трудов / АН СССР, Сибриское отд-ние, ВЦ ; под редакцией В.Е. Котова. — Новосибирск : ВЦ СО АН СССР, 1983. — С. 91–109.
8. **Вальковский В.А., Малышкин В.Э.** Синтез параллельных программ и систем на вычислительных моделях. — Новосибирск : Наука. Сиб. отд-ние, 1988. — 129 с.
9. **Malyshkin V.** Active Knowledge, LuNA and Literacy for Oncoming Centuries // Programming Languages with Applications to Biology and Security : Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday. Lecture Notes in Computer Science / Bodei C., Ferrari G., Priami C. (ред.). — Springer Cham, 2015. — Т. 9465. — 375 С. — DOI 10.1007/978-3-319-25527-9_19.
10. **Тыугу Э.Х.** Концептуальное программирование. — М. : Наука. Главная редакция физико-математической литературы, 1984. — 256 с. — (Проблемы искусственного интеллекта).
11. **Мяннислау М.А., Тыугу Э.Х., Унт М.И., Фуксман А.Л.** Язык Утопист / Алгоритмы и организация решения экономических задач. Сборник статей под ред. В.М. Савинкова. — М. : «Статистика», 1977. — № 10. — С. 80–118.
12. Язык НОРМА / А.Н. Андрианов [и др.] // Препринты ИПМ им. М.В. Келдыша. — М. : ИПМ им. М.В. Келдыша, 2019. — № 132. — 48 с. — DOI 10.20948/prepr-2019-132.
13. **В.А. Бахтин** [и др.]. Расширение DVM-модели параллельного программирования для кластеров с гетерогенными узлами // Вестник Южно-Уральского университета. — Челябинск: Издательский центр ЮУрГУ, 2012. — Серия: Математическое моделирование и программирование. — № 18 (277). — Выпуск 12. — С. 82–92.
14. **N.A. Kataev, A.S. Kolganov.** The experience of using DVM and SAPFOR systems in semi automatic parallelization of an application for 3D modeling in geophysics // The Journal of Supercomputing. — US: Springer, 2018. — С. 1–11.
15. **Абрамов С.М., Адамович А.И., Позлевич Р.В.** Т-система — среда программирования с поддержкой автоматического динамического распараллеливания программ // Программные системы: Теоретические основы и приложения : сборник (под ред: А.К. Айламазян). — М. : Наука, Физматлит, 1999.
16. **С. М. Абрамов, А. Московский А., В. А. Роганов** [и др.]. Open TS: архитектура и реализация среды для динамического распараллеливания вычислений // Научный сервис в сети Интернет: технологии распределенных вычислений : Труды Всероссийской научной конференции. — М. : МГУ, 2005. — С. 79–81.

17. **Hoare, C. A. R.** An axiomatic basis for computer programming // Communications of the ACM. — 1969. — № 12(10). С. 576–580. — DOI 10.1145/363235.363259.
18. **Lamport, L.** The temporal logic of actions // ACM Transactions on Programming Languages and Systems. — 1994. — № 16 (3). — С. 872–923. — DOI 0.1145/177492.177726.
19. **Kale, Laxmikant V. and Bhatele, Abhinav.** Parallel Science and Engineering Applications: The Charm++ Approach / Taylor & Francis Group. — CRC Press, 2013. — ISBN 9781466504127.
20. **George Bosilca, Aurélien Bouteiller, Anthony Danalis** [и др.]. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability // Computing in Science and Engineering. — 2013. — Т. 15. — С. 36–45.
21. **George Bosilca, Aurelien Bouteiller, Anthony Danalis** [и др.]. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA // IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. — 2011. — DOI 10.1109/IPDPS.2011.299.
22. **Michael Bauer, Sean Treichler, Elliott Slaughter and Alex Aiken.** Legion: expressing locality and independence with logical regions // Conference on High Performance Computing Networking, Storage and Analysis, SC'12. — Salt Lake City, UT, USA, November 11 – 15. — 2012. — DOI 10.1109/SC.2012.71.
23. **E. Slaughter, W. Lee, S. Treichler, M. Bauer and A. Aiken.** Regent: a high-productivity programming language for HPC with logical regions // SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. — 2015. — С. 1–12. — DOI 10.1145/2807591.2807629.
24. **William Gropp, Ewing Lusk, Anthony Skjellum.** Using MPI: Portable Parallel Programming with the Message-Passing Interface / MIT Press : 3 изд. — 2014. — 336 С. — ISBN-13 978-0262527392.
25. **Barbara Chapman, Gabriele Jost, Ruud van der Pas.** Using OpenMP: Portable Shared Memory Parallel Programming / Scientific and engineering computation. — The MIT Press, 2007. — 353 с. —ISBN 0262533022.
26. Ansys Fluent: Fluid Simulation Software : сайт. — URL: https://www.ansys.com/products/fluids/ansys-fluent (дата обращения: 1.06.2022).
27. **James C. Phillips, David J. Hardy, Julio D. C. Maia** [и др.]. Scalable molecular dynamics on CPU and GPU architectures with NAMD // Journal of Chemical Physics. — 2020. — № 153 (044130). — DOI 10.1063/5.0014475.

28. **Surmin I.A., Bastrakov S.I., Efimenko E.S.** [и др.]. Particle-in-Cell laser-plasma simulation on Xeon Phi coprocessors // Computer Physics Communications. — 2016. — № 202. — С. 204–210.
29. **G. Baumgartner, D.E. Bernholdt, D. Cociorva** [и др.]. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry // ACM/IEEE Supercomputing 2002 Conference : труды конференции. — 2002. — С. 5.
30. **Ed Bueler.** PETSc for Partial Differential Equations: Numerical Solutions in C and Python / SIAM. — ISBN 978-1-611976-30-4. — 2020. — DOI 10.1137/1.9781611976311.
31. **Клини С.** Математическая логика. — М. : Мир, 1973.
32. **Мендельсон Э.** Введение в математическую логику. — М. : Наука, 1971.
33. **Колмогоров А.Н.** Теория информации и теория алгоритмов. — М. : Наука, 1987.
34. **Ахо Альфред В.** Структуры данных и алгоритмы / Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман ; [пер. с англ. и ред. А. А. Минько]. - Москва [и др.] : Вильямс, 2010. — 391 с. — ISBN 978-5-8459-1610-5.
35. **Кнут Д.Э.** Искусство программирования : пер. с англ. / Дональд Э. Кнут. — Москва [и др.] : Диалектика ; Санкт-Петербург : Диалектика, 2020. — (Классический труд. Новое издание). — ISBN 978-5-8459-1980-9.
36. **В.Э. Малышкин.** Параллельное программирование мультикомпьютеров : Учеб. пособие / В. Э. Малышкин ; М-во образования Рос. Федерации. Яросл. гос. ун-т им. П. Г. Демидова. — Ярославль, 1999. — 134 с. — ISBN 5-8397-0052-5.
37. **Andrews, G. R.** Foundations of Multithreaded, Parallel, and Distributed Programming. — Reading, MA: Addison-Wesley (русский перевод Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. — М. : Издательский дом "Вильямс", 2003)
38. **Buyya, R.** High Performance Cluster Computing. — Prentice Hall PTR, Prentice-Hall Inc, 1999.
39. **Bertsekas, D.P., Tsitsiklis, J.N.** Parallel and distributed Computation. Numerical Methods. — Prentice Hall, Englewood Cliffs, New Jersey. — 1989.
40. **Dongarra, J.J., Duff, L.S., Sorensen, D.C.** [и др.]. Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools) / Soc for Industrial & Applied Math. — 1999.
41. **Иан Грэхем.** Объектно-ориентированные методы. Принципы и практика = Object-Oriented Methods: Principles & Practice. — 3-е изд. — М. : «Вильямс», 2004. — 880 с. — ISBN 0-201-61913-X.

42. **Турский В.** Методология программирования. — М. : Мир, 1981. — 264 с.
43. **Молчанов А.Ю.** Системное программное обеспечение: учебник для вузов. 3-е изд. — СПб. : Питер, 2010. – 400 с.
44. **Eric Steven Raymond.** The Art of UNIX Programming. Addison-Wesley, 2004. — ISBN-13 978-0131429017.
45. **Альфред В. Ахо, Моника С. Лам, Рави Сети** [и др.]. Компиляторы: принципы, технологии и инструментарий = Compilers: Principles, Techniques, and Tools. — 2 изд. — М. : Вильямс, 2008. — ISBN 978-5-8459-1349-4.
46. **Jeffrey Dean and Sanjay Ghemawat.** MapReduce: Simplified Data Processing on Large Clusters / OSDI'04: Sixth Symposium on Operating System Design and Implementation. — 2004. — С. 137–150.
47. **Tom White.** Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale / O'Reilly Media ; 4 изд. = 4th edition. —2015. — 756 с. — ISBN-13 978-1491901632.
48. **R. A. Ferreira** [и др.]. Anthill: a scalable run-time environment for data mining applications // 17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05) : труды конференции. — 2005. — С. 159-166. — DOI 10.1109/CAHPC.2005.12.
49. PGAS: Partitioned Global Address Space : [сайт]. — URL: http://www.pgas.org/ (дата обращения: 1.06.2022).
50. **Yelick, Katherine A.** [и др.]. Titanium: A High-Performance Java Dialect // ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford, California. — 1998.
51. **W. Carlson, J. Draper, D. Culler** [и др.]. — Introduction to UPC and Language Specification // CCS-TR-99-157, IDA Center for Computing Sciences, 1999.
52. **Chivers I., Sleightholme J.** Coarray Fortran. In: Introduction to Programming with Fortran. — Springer Cham, 2015. — DOI 10.1007/978-3-319-17701-4_32.
53. **Marc Tajchman.** Programming Experiences Using the X10 Language // Computing in Science and Engineering. — № 12(6). — 2010. — С. 62–69.
54. **H. Kaiser, T. Heller, B. Adelstein-Lelbach** [и др.]. HPX: a task based programming model in a global address space // Proceedings of the International Conference on Partitioned Global Address Space Programming Models, ACM, New York, USA. — 2014.
55. **Herbert Jordan, Philipp Gschwandtner, Peter Thoman** [и др.]. The allscale framework architecture // Parallel Computing. — Т. 99 (102648). — 2020. — ISSN 0167-8191. — DOI 10.1016/j.parco.2020.102648.

56. **Andreas Müller, Roland Rühl.** Extending high performance Fortran for the support of unstructured computations // ICS '95: Proceedings of the 9th international conference on Supercomputing. — 1995. — С. 127–136. — DOI 10.1145/224538.224552.
57. **Lee, J., & Sato, M.** Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems //39th International Conference on Parallel Processing Workshops. — 2010. — DOI 10.1109/icppw.2010.62.
58. **Легалов А. И.** Функциональный язык для создания архитектурно-независимых параллельных программ // Вычислительные технологии : журнал. — 2005. — Т. 10, № 1. — С. 71–89.
59. **Касьянов В. Н., Бирюкова Ю. В., Евстигнеев В. А.** Функциональный язык Sisal // Поддержка супервычислений и интернет-ориентированные технологии. — Новосибирск: ИСИ СО РАН, 2001. — С. 54–67.
60. **Frigo, M., Leiserson, C. E., & Randall, K. H.** The Implementation of the Cilk-5 Multithreaded Language // PLDI 1998 : [труды конференции]. — 1998. — С. 212–223.
61. **Prywes, Noah; Pnueli, Amir.** Automatic program generation in distributed cooperative computation // IEEE Transactions on Systems, Man, and Cybernetics, SMC-14(2). — 1984. — С. 275–286. — DOI 10.1109/tsmc.1984.6313210.
62. **Simon Marlow.** Parallel and Concurrent Programming in Haskell. — O'Reilly Media, Inc., 2013. — ISBN 9781449335946.
63. **Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones.** Towards Haskell in the cloud // Proceedings of the 4th ACM symposium on Haskell (Haskell '11) / Association for Computing Machinery, New York, NY, USA. — 2011. — С. 118–129. DOI 10.1145/2034675.2034690.
64. **Малышкин В. Э., Цыгулин А. А.** ParaGen — генератор параллельных программ, реализующих численные модели // Автометрия. — 2003. — Т. 39, № 3. — С. 124–135.
65. **Цыгулин А.А.** Метод и алгоритмы автоматической генерации параллельных программ, реализующих численные методы на регулярных сетках : Автореф. дис. ... канд. техн. наук: 05.13.11; [Место защиты: Новосибирский Государственный Технический Университет]. — Н., 2004. — 10 с.
66. **Murray Cole.** Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming // Parallel Computing. — Т. 30, № 3. — 2004. С. 389–406. — ISSN 0167-8191. — DOI 10.1016/j.parco.2003.12.002.
67. **Засыпкин, Алексей Владимирович.** Алгоритмы планирования на вычислительных моделях : диссертация ... кандидата технических наук : 05.13.11. — Новосибирск, 1989. — 144 с.

68. **Арыков, Сергей Борисович.** Язык и система фрагментированного параллельного программирования задач численного моделирования : диссертация ... кандидата физико-математических наук : 05.13.11 / Арыков Сергей Борисович ; [Место защиты: Ин-т систем информатики им. А.П. Ершова СО РАН]. — Новосибирск, 2010. — 195 с. : ил. РГБ ОД, 61 11-1/381.
69. **С. Б. Арыков, В. Э. Малышкин**. Система асинхронного параллельного программирования “Аспект” // Выч. мет. программирование. — Т. 9, № 1. — 2008. — С. 48–52.
70. **Арыков С.Б.** Решение прикладных задач в системе параллельного программирования Аспект // Вестник Томского государственного университета. Управление, вычислительная техника и информатика. — №45. — 2018. — С. 59–67. — DOI: 10.17223/19988605/45/7.
71. **Kalgin K.V., Malyshkin V.E., Nechaev S.P.** [и др.]. Runtime system for parallel execution of fragmented subroutines // Proc. of the 9th Int. Conf. on Parallel Computing Technologies (PaCT-2007). Lecture Notes in Computer Science. — Т. 4671. — Berlin : Springer, 2007. — С. 544–552.
72. **Kraeva M.A., Malyshkin V.E.** Assembly technology for parallel realization of numerical models on MIMD-multicomputers // Future Generation Computer Systems. — 2001. — Т. 17. — С. 755–765.
73. **Nicholas J Carriero, David Gelernter, Timothy G Mattson** [и др.]. The Linda alternative to message-passing systems // Parallel Computing. — Т. 20, № 4. — 1994. — С. 633–655. — ISSN 0167-8191. — DOI 10.1016/0167-8191(94)90032-9.
74. **Matthew Rosing, Robert B. Schnabel, Robert P. Weaver.** The DINO parallel programming language // Journal of Parallel and Distributed Computing. — Т. 13, № 1. — 1991. С. 30–42. — ISSN 0743-7315. — DOI 10.1016/0743-7315(91)90107-K.
75. **А. В. Петров, Ю. П. Сердюк.** Система параллельного распределенного программирования MC# 2.0 // Выч. мет. программирование. — Т. 9, № 1. — 2008. — С. 1–11.
76. **Aleeva V.N., Aleev R.Zh.** High-Performance Computing Using the Application of the Q-determinant of Numerical Algorithms // 2018 Global Smart Industry Conference (GloSIC). — IEEE, 2018. — С. 1–8. — DOI 10.1109/GloSIC.2018.8570160.
77. **Горбунов-Посадов, М.М.** Формы многократно используемых компонентов программы // Препринт ИПМ. — № 37. — Москва, 1997.

78. **Augonnet, C., & Namyst, R.** A Unified Runtime System for Heterogeneous Multi-core Architectures. — Lecture Notes in Computer Science. — 2009. С. 174–183. — DOI 10.1007/978-3-642-00955-6_22.
79. **C. Silvano, K. Slaninová, J. Bispo** [и др.]. The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems. // Proceedings of the ACM International Conference on Computing Frontiers - CF '16. — 2016. — С. 288–293. — DOI 10.1145/2903150.2903470.
80. **Brad Chamberlain, Elliot Ronaghan, Ben Albrecht** [и др.]. Chapel Comes of Age: Productive Parallelism at Scale // CUG 2018, Stockholm Sweden, 2018.
81. **Malyshkin V.E., Schukin G.A.** Distributed Algorithm of Dynamic Multidimensional Data Mapping on Multidimensional Multicomputer in the LuNA Fragmented Programming System. // Parallel Computing Technologies - 14th International Conference, PaCT 2017. — 2017.
82. **Edsger W. Dijkstra, C.S. Scholten.** Termination detection for diffusing computations // Information Processing Letters. — Т. 11, № 1. — 1980. — С. 1–4. — ISSN 0020-0190. — DOI 10.1016/0020-0190(80)90021-6.
83. GitLab is the open DevOps platform : [сайт]. — URL: https://about.gitlab.com/ (дата обращения: 1.06.2022).
84. **W. Gropp, E. Lusk, and R. Thakur.** Using MPI-2: Advanced Features of the Message-Passing Interface. — MIT Press, 1999.
85. **C. Eric Wu, Anthony Bolmarcich, Marc Snir** [и др.]. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems // Proceedings of SC2000: High-Performance Networking and Computing. — 2000.
86. GDB: The GNU Project Debugger : [сайт]. — URL: https://www.gnu.org/software/gdb/ (дата обращения: 1.06.2022).
87. **Р.Н. Мустаков.** Разработка и реализация поведенческого отладчика для системы фрагментированного программирования : выпускная квалификационная работа бакалавра. — 2013. — URL: https://lib.nsu.ru/xmlui/handle/nsu/352?show=full (дата обращения: 1.06.2022).
88. **V.E. Malyshkin, V.A. Perepelkin, A.A. Tkacheva.** Control Flow Usage to Improve Performance of Fragmented Programs Execution // Proc 13th International Conference on Parallel Computing Technologies. LNCS. — Т. 9251. — Springer, 2015. — С. 86–90. — DOI 10.1007/978-3-319-21909-7_9.
89. **Перепёлкин В.А., Софронов И.В., Ткачёва А.А.** Автоматизация конструирования численных параллельных программ с заданными нефункциональными свойствами на

базе вычислительных моделей // Проблемы информатики. — № 4 (37). — ИВМиМГ СО РАН. — 2017. — С. 47–60.

90. **Belyaev N., Perepelkin V.** High-Efficiency Specialized Support for Dense Linear Algebra Arithmetic in LuNA System // Parallel Computing Technologies (PaCT 2021). Lecture Notes in Computer Science. — Т. 12942. — Springer Cham, 2021. — DOI 10.1007/978-3-030-86359-3_11.
91. **Belyaev, N., & Kireev, S.** LuNA-ICLU compiler for automated generation of iterative fragmented programs // Parallel Computing Technologies — 15th International Conference, PaCT 2019, Proceedings. Lecture Notes in Computer Science. — Т. 11657. — Springer-Verlag GmbH and Co. KG., 2019. — DOI 10.1007/978-3-030-25636-4_2.
92. **Ажбаков А.А., Перепёлкин В.А.** Разработка и реализация переносимых алгоритмов распределенного исполнения фрагментированных программ на неоднородных вычислителях // Проблемы информатики. — № 1 (42). — ИВМиМГ СО РАН. — 2019. — С. 51–69.
93. **В.А. Перепелкин, И.И. Сумбатянц.** Стенд для отладки и тестирования качества работы локальных системных распределенных алгоритмов динамической балансировки нагрузки // Вестник Южно-Уральского Государственного Университета, секция «Вычислительная математика и информатика». — Т. 4, № 3. — 2015. — С. 55–66.
94. **John Cheng, Max Grossman, Ty McKercher.** Professional CUDA C Programming. — ISBN 978-1-118-73932-7. — 2014. — 528 с.
95. **Kireev S.** A parallel 3D code for simulation of self-gravitating gas-dust systems // International Conference on Parallel Computing Technologies 2009. — Berlin, Heidelberg : Springer, 2009. — С. 406–413.
96. **Victor E. Malyshkin, Vladislav A. Perepelkin.** The PIC Implementation in LuNA System of Fragmented Programming // The Journal of Supercomputing, Special Issue on Parallel Computing Technologies. — Springer, 2014. С. 89–97. — DOI 10.1007/s11227-014-1216-8.
97. **Д.В. Лебедев, В.А. Перепелкин.** Численное решение одномерной краевой задачи фильтрации жидкости для системы "нефть-вода" и ее реализация в системе фрагментированного программирования LuNA // Вестник Казахского национального университета им. Аль-Фараби, серия математика, механика информатика. — № 3 (82). — 2014. — С. 64–73.
98. **Д.В. Лебедев, В.А. Перепелкин.** Реализация одномерной краевой задачи нефть-вода в системе фрагментированного программирования LuNA // Материалы XIV

Международной конференция "Высокопроизводительные параллельные вычисления на кластерных системах" / ПНИПУ, г. Пермь. — 2014.

99. **Akhmed-Zaki D.Zh., Lebedev D.V., Perepelkin V.A.** Implementation of a Three-Phase Fluid Flow (“Oil-Water-Gas”) Numerical Model in the LuNA Fragmented Programming System // Proc 13th International Conference on Parallel Computing Technologies. LNCS. — Т. 9251. — Springer, 2015. — С. 489–497. — DOI 10.1007/978-3-319-21909-7_47.
100. **Akhmed-Zaki, D., Lebedev, D., Perepelkin, V.** Implementation of a three dimensional three-phase fluid flow (“oil–water–gas”) numerical model in LuNA fragmented programming system // Journal of Supercomputing. — № 73 (2). — Springer, 2017. — С. 624–630. — DOI 10.1007/s11227-016-1780-1.
101. **Akhmed-Zaki, D., Lebedev, D., Perepelkin, V.** Implementation of a 3D model heat equation using fragmented programming technology // J Supercomput. — 2019. — С. 7827–7832. — DOI 10.1007/s11227-018-2710-1.
102. **Akhmed-Zaki, D., Lebedev, D., Malyshkin, V., Perepelkin, V.** Automated construction of high performance distributed programs in LuNA system // 15th International Conference on Parallel Computing Technologies, PaCT 2019; Almaty; Kazakhstan. LNCS 11657. — Springer, 2019. — С. 3–9. — DOI 10.1007/978-3-030-25636-4_1.
103. **Сапронов, И.С.** Параллельно-конвейерный алгоритм / И.С. Сапронов, А.Н. Быков // Атом. — 2009. — №4 (44). — С. 26–27. — URL: https://rucont.ru/efd/558587 (дата обращения: 1.06.2022).
104. **Перепелкин В.А.** Оптимизация исполнения фрагментированных программ на основе профилирования // Шестая Сибирская конференция по параллельным и высокопроизводительным вычислениям : Программа и тезисы докладов. — Томск: Изд-во Том. ун-та, 2011. — С. 117–122.
105. **B. Daribayev, V. Perepelkin, D. Lebedev** [и др.]. Implementation of the Two-Dimensional Elliptic Equation Model in LuNA Fragmented Programming System // 2018 IEEE 12th International Conference on Application of Information and Communication Technologies (AICT). — 2018. — С. 1–4.
106. **Lynn Elliot Cannon.** A cellular computer to implement the Kalman Filter Algorithm. — Montana State University, 1969. — (Technical report, Ph.D. Thesis).
107. **Nikolay B., Perepelkin. V.** Automated GPU Support in LuNA Fragmented Programming System // Parallel Computing Technologies (PaCT) 2017. Lecture Notes in Computer Science. — Т. 10421. — Springer, Cham, 2017. — С. 272–277. — DOI 10.1007/978-3-319-62932-2_26.

108. **Malyshkin V., Perepelkin V.** Trace-Based Optimization of Fragmented Programs Execution in LuNA System // Parallel Computing Technologies. PaCT 2021. Lecture Notes in Computer Science. — Т. 12942. — Springer, Cham, 2021. — DOI 10.1007/978-3-030-86359-3_1.

109. **С.Е. Киреев, В.А. Перепёлкин.** Исследование производительности реализации метода IADE в системе фрагментированного программирования LuNA // Параллельные вычислительные технологии (ПаВТ'2016) : труды международной научной конференции. — Челябинск: Издательский центр ЮУрГУ, 2016. — С. 780.

Список иллюстративного материала

Приложение А. Пример простого фрагментированного алгоритма

Рассмотрим, как алгоритмы могут быть выражены в виде фрагментированного алгоритма (ФА) на иллюстративном примере — вычисление числа Фибоначчи φ_n с заданным номером n по известным формулам:

1. $\varphi_0=\varphi_1=1$;
2. $\varphi_{n+2}=\varphi_{n+1}+\varphi_n$, $n \in \mathbf{N}$.

Для определения ФА необходимо задать кортеж⟨N,I,A,O⟩.

Определим три ФК в множестве A={one,sum,init}:

1. Пусть one ∈ A, in(one)=0, out(one)=1. За рамками модели имеем ввиду, что ФК one вычисляет значение 1 для своего единственного выходного ФД.
2. Пусть sum ∈ A, in(sum)=2, out(sum)=1. Этот ФК будет суммировать два числа (входные ФД), вычисляя их сумму в единственный выходной ФД.
3. Пусть init ∈ A, in(init)=0, out(init)=1. Этот ФК будет инициализировать выходной ФД входным параметром n-2.

Определим множество имён N={φ,n,J,K}.

Определим одноэлементное множество индексных переменных I={i}.

Пусть множество операторов $O=\{o_1,...,o_4\}$, где:

- o_1= ⟨one,⟨φ,0⟩⟩;
- o_2= ⟨one,⟨φ,1⟩⟩;
- o_3= ⟨init,n⟩;
- o_4=⟨i,0,n,B⟩, где $B=\{b_1,b_2,b_3\}$:
    - b_1=⟨sum,i,⟨φ,0⟩,⟨J,i⟩⟩;
    - b_2=⟨sum,⟨J,i⟩,⟨φ,0⟩,⟨K,i⟩⟩;
    - b_3=⟨sum,⟨φ,i⟩,⟨φ,⟨J,i⟩⟩,⟨φ,⟨K,i⟩⟩⟩.

Для наглядности приведём этот же ФА в псевдокоде (листинг А.1).

Листинг А.1. Пример ФА вычисления числа Фибоначчи.

```
00: φ0=one()
01: φ1=one()
02: n=init()
03: for i=0..n {
04:   Ji=sum(i,φ0)          ▻ вычисляем i+1
05:   Ki=sum(Ji,φ0)         ▻ вычисляем i+2
06:   φKi=sum(φi,φJi)       ▻ вычисляем φi+2
07: }
```

В результате исполнения алгоритма будет вычислено требуемое число Фибоначчи. Отметим, что операторы o_1, o_2 и o_3 могут быть исполнены в любом порядке, в том числе параллельно. Оператор o_4 может быть исполнен сразу после того, как будет исполнен o_3, в том числе, не дожидаясь исполнения o_1 и o_2. Несмотря на последовательный характер записи описываются именно множества, а порядок выполнения ФВ определяется готовностью ФД.

Приложение Б. Доказательство универсальности ФА

ФА является универсальным представлением алгоритма в том смысле что оно может быть использовано для представления любой вычислимой функции. Докажем это, показав, что с помощью ФА можно представить любую частично-рекурсивную функцию. Для этого повторим индуктивное определение частично-рекурсивной функции в терминах ФА.

Пусть дана некоторая n-местная функция f. Будем говорить, что ФА $\langle N,I,A,O\rangle$ с интерпретацией T и множеством ФД $\{X_1,...,X_n,Y\}$ **представляет функцию** f, если для любого набора переменных $x_1,...,x_n$ выполняются следующие условия:

- Существует непротиворечивая интерпретация $T' \mid \forall\, i \in \{1,...,n\}\ T'(X_i)=x_i$, $T'(Y)=f(x_1,...,x_n)$ если f определена на $x_1,...,x_n$ и $T'(a)=T(a)$ для любых a из области определения T.
- Тогда, и только тогда, когда f определена, верно, что для любого исполнения ФА $R=S_0,S_1,...$ над множеством ФД $X_1,...,X_n$ существует конечное i, такое что $S_i=\langle CF,DF\rangle$ и $Y \in DF$.

Сначала рассмотрим представление в виде ФА базовых функций o, s и I_n^m.

Функция нуля $y=o=0$ может быть представлена ФА $\langle N,I,A,O\rangle$ с интерпретацией T и множеством ФД $\{Y\}$, где $Y=\langle y\rangle$, $y \in N$ следующим образом:

- $N=\{y\}$,
- $I=\emptyset$,
- $A=\{a_0\}$,
- $O=\{\langle a_0,y\rangle\}$,
- $T(a_0)=o$, где o — нульместная функция 0.

Функция следования $y=s(x)=x+1$ может быть представлена ФА $\langle N,I,A,O\rangle$ с интерпретацией T над множеством ФД $\{X,Y\}$ следующим образом (пусть $X=\langle x\rangle$, $Y=\langle y\rangle$):

- $N=\{x,y\}$,
- $I=\emptyset$,
- $A=\{a_s\}$,
- $O=\{\langle a_s,x,y\rangle\}$,
- $T(a_s)=s$, где s — функция следования.

Функция выбора $I_n^m(x_1,...,x_n)=x_m$ $(1\le m\le n)$ может быть представлена ФА $\langle N,I,A,O\rangle$ с интерпретацией T над множеством ФД $\{X_1,...,X_n,Y\}$ следующим образом (пусть $X_i=\langle x_i\rangle$, $Y=\langle y\rangle$):

- $N=\{x_1,...,x_n,y\}$,
- $I=\emptyset$,
- $A=\{a_e\}$,

- $O=\{\langle a_e,x_m,y\rangle\}$,
- $T(a_e)=e$, где e — тождественная функция $e(x)=x\ \forall x \in \mathbb{N}$.

Теперь рассмотрим, как представить функции, получаемые с помощью операторов суперпозиции, примитивной рекурсии и минимизации из вычислимых функций.

Оператор суперпозиции. Пусть имеется m n-местных вычислимых функций $f_1,...,f_m$ и m-местная вычислимая функция g. Тогда при помощи оператора суперпозиции можно определить вычислимую n-местную функцию $S(f_1,...,f_m,g)=h$, где $h(x1,...,xm)=g(f1(x1,...,xm),...,fn(x1,...,xm))$. Пусть функции $f_1,...,f_m,g$ представимы в виде ФА $\langle N^{f1},I^{f1},A^{f1},O^{f1}\rangle,...,\langle N^{fm},I^{fm},A^{fm},O^{fm}\rangle,\langle N^{g},I^{g},A^{g},O^{g}\rangle$ с интерпретациями $T^{f1},...,T^{fm},T^{g}$ над множествами ФД $\{X_1^{f1},...,X_n^{f1},Y^{f1}\},...,\{X_1^{fm},...,X_n^{fm},Y^{fm}\},\{X_1^{g},...,X_m^{g},Y^{g}\}$ соответственно. Определим ФА $\langle N,I,A,O\rangle$ с интерпретацией T над множеством ФД $\{X_1,...,X_n,Y\}$, представляющий функцию h.

Без ограничения общности будем считать что попарное пересечение любых множеств из $N^{f1},...,N^{fm},N^{g}$ и попарное пересечение любых множеств из $I^{f1},...,I^{fm},I^{g}$ пусты, $a_e \notin A^{f1}\cup...\cup A^{fm}\cup A^{g}$ а также: $X_i=\langle x_i\rangle$, $X_i^{fj}=\langle x_i^{fj}\rangle$, $X_j^{g}=\langle x_j^{g}\rangle$, $Y^{fj}=\langle y^{fj}\rangle$, $Y^{g}=\langle y^{g}\rangle$, $Y=\langle y\rangle$, $i \in \{1,...,n\}$, $j \in \{1,...,m\}$.

- $N=N^{f1}\cup...\cup N^{fm}\cup N^{g}$,
- $I=I^{f1}\cup...\cup I^{fm}\cup I^{g}$,
- $A=A^{f1}\cup...\cup A^{fm}\cup A^{g}\cup\{a_e\}$,
- $O=O^{f1}\cup...\cup O^{fm}\cup O^{g}\cup O_1\cup O_2\cup O_3$, где:
  - $O_1=\{\langle a_e,x_i,x_i^{fj}\rangle | i \in \{1,...,n\}, j \in \{1,...,m\}\}$,
  - $O_2=\{\langle a_e,y^{fi},x_i^{g}\rangle | i \in \{1,...,m\}\}$,
  - $O_3=\{\langle a_e,y^{g},y\rangle$,
- $T(a_e)=e$, где e — тождественная функция $e(x)=x\ \forall x \in \mathbb{N}$, а также положим T тождественной $T^{f1},...,T^{fm},T^{g}$ на их (непересекающихся) областях определения.

Оператор примитивной рекурсии. Пусть имеются вычислимая n-местная функция f и вычислимая n+2-местная функция g. Тогда n+1-местная функция, полученная оператором примитивной рекурсии $h=R(f, g)$, где $h(x_1,...,x_n,0)=f(x_1,...,x_n)$ и $h(x_1,...,x_n,k+1)=g(x_1,...,x_n,k,h(x_1,...,x_n,k))$ также вычислима.

Пусть функции f и g представимы в виде ФА $\langle N^{f},I^{f},A^{f},O^{f}\rangle$, $\langle N^{g},I^{g},A^{g},O^{g}\rangle$ с интерпретациями T^{f}, T^{g} над множествами ФД $\{X_1^{f},...,X_n^{f},Y^{f}\}$, $\{X_1^{g},...,X_{n+2}^{g},Y^{g}\}$ соответственно. Определим ФА $\langle N,I,A,O\rangle$ с интерпретацией T над множеством ФД $\{X_1,...,X_n,Y\}$, представляющий функцию h.

Примечание — Тут и далее в качестве ссылок будем дополнительно использовать целочисленные константы 0 и 1 и простые арифметические выражения — инкремент и декремент). При этом под каждой константой понимается ссылка на некоторый уникальный ФД, который вычисляется с помощью тривиального ФК, задающего эту константу; а под каждым

арифметическим выражением понимается ссылка на ФД, который также вычисляется простым ФК из соответствующих аргументов. Использование констант и арифметических операций в доказательстве не принципиально и сделано из соображений упрощения изложения.

Без ограничения общности будем считать что $N^f \cap N^g = \emptyset$, $z \notin N^f \cup N^g$, $I^f \cap I^g = \emptyset$, $i \notin I^f \cup I^g$, $a_e \notin A^f \cup A^g$, а также: $X_i = \langle x_i \rangle$, $X_i^f = \langle x_i^f \rangle$, $X_i^g = \langle x_i^g \rangle$, $Y^f = \langle y^f \rangle$, $Y^g = \langle y^g \rangle$, $Y = \langle y \rangle$, $X_{n+2}^g = \langle x_{n+2}^g \rangle$ $i \in \{1,...,n=1\}$.

- $N = N^f \cup N^g \cup \{z\}$,
- $I = I^f \cup I^g \cup \{i\}$,
- $A = A^f \cup A^g \cup \{a_e\}$,
- $O = O^f \cup O_1 \cup O_2 \cup O_6$
    - $O_1 = \{\langle a_e, x_j, x_j^f \rangle | j \in \{1,...,n\}\} \cup \{\langle a_e, y^f, \langle z,0 \rangle \rangle\}$,
    - $O_2 = \{ \langle i,0,x_{n+1}, O'^g \cup O_3 \cup O_4 \cup O_5 \rangle\}$, где:
        - O'^g получается из O^g путём добавления индекса i в конец каждой ссылки, не являющейся индексной переменной,
        - $O_3 = \{\langle a_e, x_j, \langle x_j^g, i \rangle \rangle | j \in \{1,...,n\}\}$,
        - $O_4 = \{\langle a_e, i, \langle x_{n+1}^g, i \rangle \rangle\} \cup \{\langle a_e, \langle z,i \rangle, \langle x_{n+2}^g, i \rangle \rangle\}$,
        - $O_5 = \{\langle a_e, \langle y^g, i \rangle, \langle z, i+1 \rangle \rangle\}$,
    - $O_6 = \{\langle a_e, \langle z, x_{n+1} \rangle, y \rangle\}$.
- $T(a_e) = e$, где e — тождественная функция $e(x) = x$ $\forall x \in \mathbb{N}$, а также положим T тождественной T^f и T^g на их (непересекающихся) областях определения.

Оператор минимизации. Пусть определены вычислимая n+1-местная функция f. Тогда n-местная функция g, определённая оператором минимизации $g = M(f)$, где $g(x_1,...,x_n) = \min y \mid f(x_1,...,x_n,y) = 0$, также является вычислимой.

Пусть функция f представима в виде ФА $\langle N^f, I^f, A^f, O^f \rangle$ с интерпретацией T^f над множествами ФД $\{X_1^f,...,X_{n+1}^f, Y^f\}$. Определим ФА $\langle N,I,A,O \rangle$ с интерпретацией T над множеством ФД $\{X_1,...,X_n,Y\}$, представляющий функцию g.

Без ограничения общности будем считать $z \notin N^f$, $i \notin I^f$, $a_e \notin A^f$, а также $X_i = \langle x_i \rangle$, $X_i^f = \langle x_i^f \rangle$, $Y^f = \langle y^f \rangle$, $Y = \langle y \rangle$, $i \in \{1,...,n\}$.

- $N = N^f \cup \{z\}$,
- $I = I^f \cup \{i\}$,
- $A = A^f \cup \{a_e\}$,
- $O = O_1 \cup O^f \cup \{\langle i, \langle z,i \rangle, 0, y, O'^f \cup O_2 \rangle\}$, где:
    - $O_1 = \{\langle a_e, x_j, x_j^f \rangle | j \in \{1,...,n\}\} \cup \{\langle a_e, y^f, \langle z,0 \rangle \rangle\}$,
    - O'^f получается из O^f путём добавления индекса i в конец каждой ссылки, не являющейся индексной переменной,

- $O_2=\{\langle a_e,x_j,\langle x_j^f,i\rangle\rangle | j \in \{1,...,n\}\} \cup \{\langle a_e,\langle y^f,i\rangle,\langle z,i+1\rangle\rangle\}$,

- $T(a_e)=e$, где e — тождественная функция $e(x)=x\ \forall x \in \mathbb{N}$, а также положим T тождественной T^f на области применения последней.

Таким образом, мы показали представимость в виде ФА всех базовых функций, и всех функций, которые могут быть из них получены из них конечным количеством применений операторов суперпозиции, примитивной рекурсии и минимизации, что означает представимость в виде ФА любой частично-рекурсивной функции, что и требовалось доказать.

Приложение B. Расширенные синтаксические средства языка LuNA

Рассмотрим дополнительные средства языка LuNA.

Комментарии. В целом язык LuNA имеет C-подобный синтаксис (фигурные скобки, операторы for/while, точки с запятой как разделители операторов, и т.п.), хотя его семантика существенно отличается. В частности, в язык были добавлены C-подобные комментарии — концевые и многострочные. Также были добавлены однословные комментарии, полезные для коротких ремарок. Слово, начинающееся с символа ` (обратный апостроф) игнорируется.

```
df N; // N denotes the problem size
init(`in m, `out N); /* Note: `in and `out
        are one-word comments */
```

Директивы препроцессора. Введена возможность определения макросов, похожих на макросы препроцессора C, но с некоторыми отличиями. Директива определяется так:

```
#define greet(arg1, arg2) Hello, $arg1 \
        and $arg2.
#define N 10
#define FLAG
```

Отличие состоит в том, что в теле макроса параметры подставляются по именам параметра, предварённым символом $. Символ \ в конце строки позволяет определять многострочные макросы. Предусматриваются также условные подстановки вида:

```
#ifdef <macro_name> <body1> [#else <body2>] #endif
```

и

```
#ifndef <macro_name> <body1> [#else <body2>] #endif
```

Применение макросов также использует символ $:

```
$N // gives 10
$greet(John, Marry) // gives: Hello, John
                                    and Marry
```

Включение. Для поддержки модульности на уровне файлов добавлена возможность включения одних файлов в другие с помощью механизма, повторяющего механизм препроцессора C++ — директиву #include:

```
#include "path/to/file"
```

Она включает содержимое файла по указанному в кавычках пути в текущее место.

Внешние блоки (foreign blocks). Концепция внешнего блока подразумевает вставку в исходный код на языке LuNA фрагмента кода на некотором другом языке. Благодаря этому механизму возможно прямо в теле LuNA-программы определять процедуры, реализующие ФК. В связи с тем, что внешний блок может содержать, вообще говоря, произвольные последовательности символов, маркер окончания блока является параметром конструкции, что позволяет гибким образом избежать коллизий:

```
${<END}some textEND
```

Тут последовательность символов ${<...} является маркером начала внешнего блока, а вместо многоточия указывается произвольная последовательность символов кроме символа }, которая должна считаться маркером конца блока. В примере выше это три символа END. Телом внешнего блока, соответственно, будет являться последовательность из 9 символов «some text».

Базовые типы данных и выражения. Для удобства введены базовые типы данных — целочисленный (**int**), вещественный (**real**) и строковой (**string**). Принадлежность ФД к тому или иному типу означает, что его значение является, соответственно, целым числом, вещественным числом или строкой. В противном случае ФД считается «чёрным ящиком», значение которого используется только внутри ФК. Те же типы характеризуют и выражения, которые могут быть использованы в качестве значений параметров. Семантика стандартных операций (+, −, *, /, %) соответствует их семантике в языке C. Типы данных могут быть использованы в подпрограммах в качестве входных.

Подпрограммы. Для поддержки модульности на уровне подпрограмм введена возможность определения подпрограмм. На верхнем структурном уровне исходного кода пользователь может определять ФК и подпрограммы, причём одна из подпрограмм считается головной (аналог функции main в языке C), и её тело и задаёт множество операторов, составляющих ФА. По умолчанию головной подпрограммой считается подпрограмма с именем main.

```
import initialize(name) as init;
import assign(int, name) as assign;
sub routine(int val, name K) {
        assign(val, K);
}
sub main() {
        df N, M;
        init(N);
```

```
    routine(N, M);
}
```

Подпрограммы могут иметь параметры, причём в подпрограммах тип параметра **name** означает ссылочный тип, т.е. тип, который может быть использован как ссылка, в том числе для построения новых ссылок путём добавления в конец ссылки индексного выражения. В ФК ссылочный тип **name** означает выходной параметр, но в подпрограммах ссылка может быть использована как для обозначения входного, так и выходного параметров (в теле подпрограммы).

Логические выражения и условный оператор. Вводится условный оператор следующего вида:

```
if (cond) {...}
if (cond) {...} else {...}
```

Тут cond — целочисленное выражение, которое считается ложным, если его значение равно 0 и истинным в противном случае. Если условие истинно, то операторы, определённые в теле цикла, будут исполнены (описываемое ими множество ФВ будет добавлено в следующее состояние), а если ложно — то нет. Если определена ветка **else**, то в случае ложного условия исполнены будут её операторы. Для работы с логическими выражениями добавлены базовые операции &&, ||, !, аналогичные соответствующим операторам в языке C.

Определение ФК с помощью внешних блоков. Так как процедуры, реализующие ФК, описываются на последовательном языке программирования, то любая нетривиальная LuNA-программа должна снабжаться как минимум ещё одним исходным, объектным или бинарным файлом (библиотекой), содержащим реализации ФК. В язык была введена возможность определения ФК с помощью «внешних блоков». В этом синтаксисе указывается язык внешнего блока (из числа предопределённых), её параметры и тело. Пример использования внешнего блока для определения процедуры, реализующей ФК на языке C++:

```
C++ sub test() ${<END_CPP}{
    printf("Hello, single file!\n");
}END_CPP
```

Специально для языка C++ была введена стандартная пара маркеров ${{...$}}, которая дополнительно обрамляет внешний блок в фигурные скобки. Это позволяет сократить запись и сделать её более читаемой:

```
C++ sub test() ${{
    printf("Hello, single file!\n");
$}}
```

Это описание полностью эквивалентно предыдущему.

Имена во вложенных блоках. Добавлена возможность объявления имён во вложенных блоках. При этом если такое имя уже определено в вышестоящем блоке или параметре подпрограммы, то использование имени считается по ближайшему по вложенности блоку (вложенное объявление затеняет внешнее):

```
sub main() {
        df x;
        if (1) {
                df x;
                // can use the inner x here
        }
}
```

Имена во вложенных блоках удобны для промежуточных значений, чтобы не вводить многоиндексные имена глобальной области видимости для вложенных циклов.

Имена фрагментов вычислений. Для каждого оператора исполнения может быть задано имя следующим образом:

```
cf a: init(x[0]);
cf b[i]: calc(x[i], x[i+1]);
```

После терминала **cf** следует ссылка, которая считается именем ФВ. Именование ФВ является необязательным, и уникальность имён не требуется. Эти имена могут использоваться для идентификации ФВ в сообщениях об ошибках или в описании рекомендаций.

Рекомендации. Виды рекомендаций, их смысл и практическое применение рассматриваются в разделе 2.3.2. Здесь лишь опишем общий расширяемый синтаксис, который был введён в язык для описания нескольких классов рекомендаций. Этот синтаксис может быть расширен или изменён впоследствии, он является слабо связанным с остальным синтаксисом языка. Приведённые тут виды рекомендаций закрывают потребности в описании рекомендаций на текущий момент.

Рекомендации могут описываться в конце оператора или в конце определения подпрограммы:

```
sub main() {
        ...
        cf a: compute("some_text", N[K+L], M) @ {
                request K, L;
                request N[K+L];
                req_count N: 5;
```

```
            stealable, log;
        }
        ...
} @ {
        locator_cyclic N: 5;
        locator_cyclic x[i]: i/2;
}
```

Пример иллюстрирует 3 поддерживаемых вида рекомендаций, каждая из которых начинается нетерминальным символом и заканчивается точкой с запятой. Флаговые (stealable, log) задают флаги. Нетерминал определяет смысл флага. В примере stealable означает подверженность ФВ динамической балансировке нагрузки методом Work Stealing (см. раздел 3.4.6), a log — необходимость журналирования событий, связанных с этим ФВ. Перечисляющие (request K, L) содержат список ссылок, а также нетерминал, определяющий смысл этого списка. В примере request определяет рекомендуемую очерёдность запрашивания значений входных ФД. Отображающие (locator_cyclic N: 5) задают некоторое отображение через ссылку и выражение. В примере locator_cyclic задаёт формулу отображения ФД на координатную структуру (см. раздел 2.4.3). Смысл отображения задаётся нетерминалом. Расширение языка новыми видами рекомендаций часто будет сводится к введению нового нетерминала, определяющего смысл рекомендаций.

Параметры командной строки. Предусмотрена возможность передачи аргументов запуска программы из командной строки. Для этого в головной подпрограмме (main) должны быть перечислены параметры базовых типов. Каждый параметр будет считан с командной строки и приведён к соответствующему базовому типу. Если количество параметров не совпадает с количеством аргументов, то программа не будет запущена, а система выдаст сообщение об ошибке:

```
sub main(string first_arg, int second arg) {...}
```

Приложение Г. БНФ языка LuNA

В приложении приводится БНФ языка LuNA в синтаксисе распространённой утилиты yacc/bison. Словами из заглавных букв обозначены лексемы и терминальные символы. Например, лексемы, обозначаемые с префиксом KW_ обозначают ключевые слова языка LuNA. Так, KW_SUB означает лексему «sub», KW_BLOCK — лексему «block» и т.п. Другие слова из заглавных букв обозначают, в основном, пунктуацию. Так, COLON означает символ двоеточия, SCOLON — точку с запятой, LCB — левую фигурную скобку (Left Curly Bracket), и т.п.

```
program: sub_def
  | program sub_def

sub_def: KW_SUB control_pragma code_id opt_params block
  | KW_CPP KW_SUB code_id opt_params KW_BLOCK LB INT RB
  | KW_IMPORT code_id LB opt_ext_params RB KW_AS code_id SCOLON
  | KW_IMPORT code_id LB opt_ext_params RB KW_AS code_id COLON KW_CUDA SCOLON
  | KW_IMPORT code_id LB opt_ext_params RB KW_AS code_id COLON KW_CUDA COMMA KW_NOCPU
SCOLON
  | KW_CPP NAME KW_BLOCK LB INT RB

opt_ext_params: ext_params_seq
                | %empty

ext_params_seq: type code_df
                | ext_params_seq COMMA type code_df

code_df: NAME
  | %empty

type: KW_INT
  | KW_REAL
  | KW_STRING
  | KW_NAME
  | KW_VALUE AMP
  | KW_VALUE

block: statement
  | LCB opt_dfdecls statement_seq RCB opt_behavior
```

```
opt_dfdecls: dfdecls
           | %empty

dfdecls: KW_DF name_seq SCOLON

name_seq: NAME
  | name_seq COMMA NAME

statement_seq: statement
            | statement_seq statement

control_pragma: LARR where_type COMMA expr RARR
                  | LARR where_type COMMA expr COMMA expr RARR
                  | LARR where_type RARR
                  | %empty

statement: cf_statement
   | let_statement
   | for_statement
   | while_statement
   | if_statement

cf_statement: opt_label code_id opt_exprs opt_setdf_rules opt_rules opt_behavior SCOLON

opt_behavior: AT LCB behv_pragmas_seq RCB
            | %empty

behv_pragmas_seq: behv_pragma
                    | behv_pragmas_seq behv_pragma

behv_pragma: NAME id EQ expr SCOLON
           | NAME id EQG expr SCOLON
           | NAME id_seq SCOLON
           | NAME COLON expr SCOLON
           | name_seq SCOLON

id_seq: id
 | id_seq COMMA id

let_statement: KW_LET assign_seq block
```

```
for_statement: KW_FOR control_pragma NAME EQ expr DIAP expr block

while_statement: KW_WHILE control_pragma expr COMMA NAME EQ expr DIAP KW_OUT id block

if_statement: KW_IF expr block

assign_seq: assign
          | assign_seq COMMA assign

assign: NAME EQ expr

opt_label: KW_CF id COLON
    | %empty

id: NAME
| id LSB expr RSB

opt_exprs: LB exprs_seq RB
    | LB RB

exprs_seq: expr
    | exprs_seq COMMA expr

opt_setdf_rules: RARR opt_exprs
                | %empty

opt_rules: ARROW opt_exprs
    | %empty

code_id: NAME

expr: INT
  | REAL
  | STRING
  | KW_INT LB expr RB
  | KW_REAL LB expr RB
  | KW_STRING LB expr RB
  | expr PLUS expr
  | expr MINUS expr
  | expr MUL expr
```

```
  | expr DIV expr
  | expr MOD expr
  | expr LT expr
  | expr GT expr
  | expr LEQ expr
  | expr GEQ expr
  | expr DBLEQ expr
  | expr NEQ expr
  | expr DBLAMP expr
  | expr DBLPIPE expr
  | LB expr RB
  | id
  | expr QMARK expr COLON expr

opt_params: LB params_seq RB
          | LB RB

params_seq: param
          | params_seq COMMA param

param: type NAME

where_type: KW_RUSH
          | KW_STATIC
          | KW_STATIC_FOR
          | KW_UNROLLING
```

Приложение Д. Примеры реализации операторов ФА в виде программ агентов

Рассмотрим, как ФВ, описываемые различными операторами, реализуются (выражаются) в виде программы агента.

Оператор исполнения задаёт ФВ, который вычисляет конечное количество выходных ФД из конечного количества входных ФД путём применения фрагмента кода. Пользователь должен предоставить модуль, реализующий этот фрагмент кода, и применение этого модуля к значениям входных ФД для получения значений выходных ФД и составляет основную цель агента. Поэтому псевдокод программы агента может иметь, например, следующий вид (листинг Д.1).

Листинг Д.1. Ориентировочная структура программы агента.

```
для каждого входного ФД: {
        запросить ФД
        ожидать ФД
}
мигрировать на случайный узел
выходные ФД <--  применить модуль ко входным ФД
для каждого выходного ФД: ввести ФД в систему
```

Оператор арифметического цикла может быть реализован, например, следующим образом:

```
для каждого входного ФД: {
        запросить ФД
        ожидать ФД
}
для всех i от f до l: {
        мигрировать на случайный узел
        для всех операторов тела цикла: {
                породить нового агента
                ввести агента в систему
        }
}
```

Тут f и l соответствуют значениям ссылок f и l в определении оператора арифметического цикла (опр. 7.2). Другой возможный алгоритм действий агента представлен на листинге Д.2.

Листинг Д.2. Другой пример псевдокода агента.

```
для каждого входного ФД: {
        запросить ФД
        ожидать ФД
```

```
}
если
для всех i от f до l: {
        мигрировать на случайный узел
        для всех операторов тела цикла: {
                породить нового агента
                ввести агента в систему
        }
}
```

Программа агента, реализующего оператор цикла с предусловием может быть представлена на листинге Д.3.

Листинг Д.3. Алгоритм работы агента, реализующего оператор цикла с предусловием.

```
для каждого входного ФД: {
        запросить ФД
        ожидать ФД
}
если истинно с при значени счётчика b, то: {
        для всех операторов тела цикла: {
                породить нового агента
                ввести агента в систему
        }
        породить нового агента A, соответствующего B' (опр. 21)
        ввести агента A в систему
} иначе: {
        ввести в систему ФД со ссылкой e и значением b
}
```

В приведённых выше примерах представлен один из возможных вариантов, иллюстрирующих подход. На практике конкретная программа агента должна конструироваться не только так, чтобы агент реализовывал ФВ, но и делал это эффективно и с учётом поведения остальных агентов. В частности, в программы агентов может быть включена логика миграции не просто на какой-то случайный узел, а некоторым более разумным образом. Также и интерфейс взаимодействия агента с системой может быть расширен (и должен быть расширен на практике), например, введением в интерфейс вызовов, предоставляющих информацию о загруженности узлов и т.п.