МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.04.01 Информатика и вычислительная техника
Направленность (профиль): Технология разработки программных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Копылова Михаила Юрьевича

Тема работы:

РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМОВ ДИНАМИЧЕСКОГО ПРИНЯТИЯ РЕШЕНИЙ ДЛЯ СИСТЕМЫ АКТИВНЫХ ЗНАНИЙ

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э./……..
(ФИО) / (подпись)
«……»……2025г.

Руководитель ВКР
д.т.н., профессор
зав. каф. ПВ ФИТ НГУ
Малышкин В.Э./……...
(ФИО) / (подпись)
«……»……2025г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Матвеев А.С./……..
(ФИО) / (подпись)
«……»……2025г.

Новосибирск, 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий
Кафедра параллельных вычислений
Направление подготовки: 09.04.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА
Направленность (профиль): Технология разработки программных систем

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В.Э.
(фамилия, И., О.)

…….
(подпись)
«21» октября 2024 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ МАГИСТРА

Студенту Копылову Михаилу Юрьевичу, группы 23223
(фамилия, имя, отчество, номер группы)

Тема Разработка и реализация алгоритмов динамического принятия решений для системы активных знаний
(полное название темы выпускной квалификационной работы магистра)

утверждена распоряжением проректора по учебной работе от 03.11.23 № 0412,
скорректирована распоряжением проректора по учебной работе от 21.10.24 № 0383
Срок сдачи студентом готовой работы 20 мая 2025 г.
Исходные данные (или цель работы):
Разработать модель адаптивного динамического принятия решений. Реализовать компонент исполнения фрагментов кода в системе активных знаний LuNA.
Структурные части работы:
Обзор родственных работ, модель адаптивного динамического принятия решений, реализация компонента исполнения фрагментов кода, тестирование.

Руководитель ВКР
зав. каф. ПВ ФИТ НГУ,
д.т.н., профессор
Малышкин В.Э./……..
(ФИО) / (подпись)
«21» октября 2024 г.

Задание принял к исполнению
Копылов М.Ю./……..
(ФИО студента) / (подпись)
«21» октября 2024 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Матвеев А.С./……..
(ФИО) / (подпись)
«21» октября 2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность темы исследования. Современные вычислительные системы характеризуются сложными архитектур, включая многоядерные процессоры, гетерогенные ускорители и распределенные кластеры. Все это предоставляет большие вычислительные мощности, которые могут использоваться для решения задач во многих областях, например в биологии, астрофизики, геологии, геофизике и других. Тем не менее написание программ, эффективно использующих мощности таких систем, сопряжено с рядом сложностей. Традиционные подходы к разработке параллельных программ требуют от программиста не только знаний предметной области задачи, но и знаний специфики аппаратного обеспечения, ручной оптимизации взаимодействия между вычислительными узлами и трудоемкой отладки, что существенно замедляет процесс создания программного обеспечения и увеличивает вероятность ошибок.

Одним из путей решения этих проблем является использование активно развивающихся систем автоматического конструирования программ, которые призваны упростить процесс разработки за счет абстрагирования от низкоуровневых деталей реализации, таких как управление памятью, сетевое взаимодействие и параллельное выполнение. Эти системы позволяют программистам работать на высоком уровне абстракции, снижая требования к их квалификации в области оптимизации и параллельных вычислений.

Однако при этом в системах автоматического конструирования программ возникает проблема обеспечения эффективности генерируемых программ, под которой понимается достижение приемлемых показателей времени выполнения и потребления ресурсов для их практического использования. В общем виде задача оптимального исполнения является алгоритмически труднорешаемой проблемой. Поэтому для достижения приемлемой эффективности генерируемых программ система должна обладать специализацией под конкретный класс приложений и вычислительных архитектур, реализуя эту

специализацию через применение частных алгоритмов и эвристических подходов.

Для обеспечения эффективности работы сгенерированных программ системы автоматического конструирования применяют разнообразные механизмы оптимизации на различных этапах жизненного цикла программы. Они могут включать в себя статический анализ и трансформацию исходного кода на этапе компиляции, генерацию промежуточного представления, системы исполнения, способные адаптировать процесс исполнения к текущему состоянию вычислительной системы, использование метрик для оценки эффективности, разработка схем распределения ресурсов и создание механизмов динамической балансировки нагрузки.

Одной из систем автоматического конструирования программ является система LuNA [1], реализующая концепцию активных знаний [2] и развиваемая на кафедре параллельных вычислений ФИТ НГУ. Система LuNA содержит в себе различные варианты генерации и исполнения автоматически сконструированных программ, позволяющих либо исполнять полностью статически сгенерированную программу, в которой все решения о ходе ее исполнения приняты статически либо итеративно выполнять ее части, принимая все решения динамически по ходу ее исполнения. Каждый из этих вариантов обладает своими достоинствами и недостатками с точки зрения эффективности исполнения автоматически сконструированной программы. В зависимости от задачи более эффективными могут быть как статический, так и динамический подходы, либо же их комбинация, когда часть решений принимается статически, а часть динамически, что позволяет объединить преимущества обоих подходов. Также ввиду того, что спектр динамических алгоритмов принятия решений широк, важным аспектом системы является обеспечение возможности использования всего многообразия таких алгоритмов, как уже разработанных, так и тех, что могут быть предложены в будущем и их адаптации для конкретной задачи для эффективного исполнения.

Цель данной работы – разработка и реализация механизма адаптивного динамического принятия решений в системе активных знаний LuNA.

Для этого необходимо выполнить следующие **задачи**:

1. Проанализировать механизмы принятия решений в существующих системах.
2. Разработать модель адаптивного динамического принятия решений для системы LuNA.
3. Реализовать компонент “исполнитель” для системы LuNA.

Объектом исследования является процесс автоматического конструирования программы. **Предметом исследования** механизм адаптивного динамического принятия решений в системе LuNA.

Новизна научной работы состоит в разработке модели адаптивного динамического принятия решений для системы активных знаний.

Практическая ценность работы состоит в реализации компонента “исполнитель” для системы LuNA, который позволяет обеспечить исполнение фрагментов кода.

Структура работы: работа состоит из введения, трех глав и заключения. Введение раскрывает актуальность, объект, предмет, цель, задачи и методы исследования, раскрывает практическую значимость работы. В первой главе приведен обзор существующих решений. Во второй главе описан механизм адаптивного динамического принятия решений об исполнении для системы LuNA. В третьей главе описываются детали реализации компонента “исполнитель” для системы LuNA и результаты его тестирования. В заключении представлены итоги работы.

1 ОБЗОР АНАЛОГИЧНЫХ СИСТЕМ

Обзор существующих решений необходим для оценки степени разработанности проблемы принятия решений и анализа разнообразных подходов, используемых в существующих системах автоматического распараллеливания программ и системах автоматического конструирования программ. Поэтому проведение такого анализа раскроет сразу несколько аспектов проблемы принятия решений: во-первых, обозначит текущее состояние проработки проблемы, во-вторых позволит учесть накопленный опыт в разработке алгоритмов принятия решений, во-третьих, поможет выявить актуальные нерешенные проблемы в данной области.

Представленный ниже анализ существующих систем проводился в соответствии со следующими критериями:

- Методы и гибкость механизмов принятия решений – какими механизмами принятия решений обладает система, позволяет ли она дополнять систему новыми механизмами. Высокая гибкость позволяет учитывать различные нефункциональные свойства, что важно для систем, применяемых в широком спектре предметных областей;
- Модель вычислений – используемая модель вычислений определяет эффективность решения той или иной задачи;
- Степень вовлеченности пользователя – чем ниже степень вовлеченности, тем выше уровень автоматизации, что способствует снижению требований к квалификации пользователя и упрощает применение системы.

В рамках анализа рассматривались специализированные модели вычислений [3–6], исполнительные системы с параллелизмом уровня задач [7,8], системы с управляемым исполнением [9–12] и системы автоматизированного распараллеливания последовательного кода [13–16]. В рамках обзора подробно рассмотрим некоторых характерных представителей.

1.1 MapReduce (Hadoop)

Парадигма MapReduce [3], первоначально предложенная и реализованная компанией Google для обработки и генерации больших наборов данных, получила широкое распространение благодаря своей открытой реализации в рамках проекта Apache Hadoop [4]. Apache Hadoop представляет собой фреймворк, предназначенный для разработки и выполнения распределенных программ, обрабатывающих большие объемы данных. Ключевыми свойствами системы являются масштабируемость, отказоустойчивость, достигаемая за счет репликации данных и автоматического перезапуска сбойных задач, а также относительно высокий уровень абстракции, скрывающий от разработчика многие сложности параллельного программирования и распределенного хранения данных посредством файловой системы HDFS (Hadoop Distributed File System).

Система Hadoop в ее канонической реализации MapReduce предоставляет весьма ограниченные возможности для динамического выбора между функционально эквивалентными операциями. Разработчик обязан явно определить функции Map и Reduce, которые будут применяться ко всему набору входных данных. Фреймворк не содержит встроенных механизмов для автоматического выбора между, например, двумя различными реализациями функции Map, даже если они семантически эквивалентны по входам и выходам, на основе каких-либо динамических критериев в ходе одного задания. Любой такой выбор должен быть осуществлен либо на этапе проектирования и кодирования самим разработчиком, либо путем запуска различных заданий с разными реализациями. Таким образом, автоматизация и адаптивность при выборе варианта выполнения операции в рамках задания MapReduce имеют ограниченную реализацию. Принятие решений системой сосредоточено не на выборе какую операцию исполнить из набора альтернатив, а на том как исполнить уже заданную операцию.

Динамическое принятие решение в Hadoop выражается прежде всего в виде распределения ресурсов и управления исполнением. Управляемыми ресурсами выступают блоки данных, хранящиеся в HDFS, и вычислительные слоты (контейнеры) на узлах кластера (DataNodes, на которых также функционируют NodeManagers). Распределение исходных данных по узлам HDFS происходит при их загрузке и направлено на равномерное заполнение дискового пространства и обеспечение репликации. На этапе выполнения задания MapReduce, управляющий компонент (ResourceManager в архитектуре YARN) динамически назначает задачи Map узлам, стремясь повысить локальность данных, т.е. выполнять обработку на тех же узлах, где хранятся соответствующие блоки данных и уменьшить сетевой трафик. Задачи Reduce также распределяются по доступным слотам. Динамизм проявляется в том, что при освобождении ресурсов или при сбоях задач происходит перераспределение. Критериями оптимизации являются локальность данных, балансировка нагрузки (заполнение доступных слотов) и отказоустойчивость (перезапуск задач). Однако, это распределение касается скорее экземпляров предопределенных задач, а не изменения логики вычислений.

Гранулярность управления – это отдельные задачи Map и Reduce. Система осуществляет динамическое планирование их запуска. Если какая-либо задача выполняется аномально долго (является "отстающей"), система может запустить ее дублирующую копию на другом узле, и результат будет принят от той копии, которая завершится первой. Это помогает справляться с проблемами "медленных" узлов или временными флуктуациями производительности. При сбое задачи или целого узла, Hadoop автоматически перезапускает затронутые задачи на других доступных узлах. Однако, управление исполнением, как и распределение ресурсов, подчинено общей, довольно жесткой структуре потока данных "Map → Shuffle → Reduce", и не предполагает глубокой реконфигурации этого потока во время выполнения. Мониторинг состояния задач и ресурсов осуществляется централизованно.

С точки зрения архитектурных особенностей и модели вычислений, Hadoop предлагает высокий уровень абстракции, что упрощает разработку распределенных приложений для определенных классов задач, но ограничивает гибкость в динамическом принятии решений за пределами заложенных в фреймворк механизмов. Система времени исполнения (YARN ResourceManager и ApplicationMaster для каждого задания) играет ключевую роль в оркестрации, но направлена на управление ресурсами и отказоустойчивость в рамках заданной парадигмы, а не на глубокий анализ и выбор альтернативных вычислительных стратегий. Поддержка гетерогенных вычислительных сред в классическом MapReduce ограничена; YARN предоставляет более гибкую платформу, позволяя запускать разнообразные типы приложений, но эффективное использование специфических аппаратных ускорителей (например, GPU) в рамках стандартной задачи MapReduce требует значительных дополнительных усилий и не является встроенной функцией динамического выбора.

1.2 PaRSEC

Система PaRSEC [9] (Parallel Scheduling and Execution Controller), и ее последующее развитие в лице PaRSEC (Parallel Runtime Scheduling and Execution Controller), разработанные в Innovative Computing Laboratory (ICL) Университета Теннесси, Ноксвилл, представляют собой программные окружения, ориентированные на реализацию модели параллелизма, основанной на задачах (task-based parallelism). Фундаментальным конструктом данных систем является явное описание параллельной программы в виде графа задач, часто параметризованного (Parameterized Task Graph, PTG) [17], где вершины соответствуют вычислительным задачам, а дуги – зависимостям по данным между ними. Ключевые свойства PaRSEC включают динамическое планирование исполнения задач на гетерогенных архитектурах, уменьшение издержек на перемещение данных и предоставление инструментов для построения параллельных программ.

В аспекте вывода алгоритма, система PaRSEC по своей основной архитектуре не предполагают встроенных механизмов для автономного динамического выбора системой времени исполнения между семантически эквивалентными, но алгоритмически различными, реализациями одной и той же логической операции в ходе одного запуска. Выбор конкретного алгоритмического пути преимущественно осуществляется на этапе конструирования PTG разработчиком или специализированным генератором кода. Тем не менее, сама парадигма допускает определение различных версий задач (task variants) внутри PTG, например, оптимизированных для разных типов вычислителей (CPU, GPU) или для различных характеристик входных данных (например, размер матрицы). Однако, решение о выборе такой вариантной задачи, если оно и происходит динамически, управляется скорее логикой, заложенной при формировании графа, или внешними эвристиками, передаваемыми системе времени исполнения, нежели автономным анализом альтернатив самой средой исполнения. Таким образом, автоматизация и адаптивность в данном контексте ограничены. Основания для принятия решения о выборе варианта задачи, если таковые предусмотрены, обычно включают тип доступного вычислительного ресурса или параметры, указанные при запуске.

Динамическое принятие решений в PaRSEC аналогично Hadoop. Управляемыми ресурсами выступают вычислительные ядра процессоров и графические ускорители или другие специализированные вычислительные устройства. Система времени исполнения динамически назначает готовые к выполнению задачи (т.е. те, у которых удовлетворены все зависимости по входным данным) на свободные вычислительные ресурсы. При этом учитывается принцип локальности данных: планировщик распределяет задачи на вычислительные узлы, содержащие необходимые данные, либо выбирает узлы с меньшими затратами на их передачу. Динамизм проявляется в непрерывном процессе мониторинга состояния задач и ресурсов и принятия

решений о размещении новых задач по мере их готовности. Критериями при распределении задач являются уменьшение общего времени выполнения программы, повышение загрузки доступных вычислительных ресурсов и соблюдение зависимостей по данным, что позволяет сократить объем передаваемых данных и время ожидания. Балансировка нагрузки достигается за счет непрерывного распределения готовых задач и гибкого управления очередями задач для каждого вычислительного ресурса.

Управление исполнением в PaRSEC осуществляются на уровне отдельных задач, определенных в PTG, что обеспечивает мелкозернистый контроль над процессом вычислительный. Центральным элементом системы является динамический планировщик задач (scheduler), который функционирует по принципу, близкому к dataflow: задача становится кандидатом на исполнение только после того, как все ее входные данные произведены и доставлены предшествующими задачами в графе зависимостей. Это позволяет системе адаптироваться к реальному порядку завершения задач и асинхронности вычислительного процесса, что важно на гетерогенных платформах с различной производительностью вычислителей. Планировщик способен обрабатывать направленные ациклические графы (DAG) зависимостей, включая параметризованные описания, позволяющие генерировать экземпляры графа для конкретных размеров входных данных.

Модель вычислений в системе PaRSEC требует от разработчика явного и детального описания параллелизма посредством PTG. Этот подход предоставляет системе времени исполнения информацию о структуре зависимостей между задачами. На основе этой информации runtime-система в PaRSEC позволяет оркестрировать исполнение задач на основе анализа графа зависимостей и текущего состояния вычислительных ресурсов.

С точки зрения степени вовлеченности пользователя работа с PaRSEC предполагает вовлеченность программиста на этапе декомпозиции приложения и описания графа задач. Программисту необходимо разбить алгоритм на

дискретные задачи и корректно специфицировать все зависимости по данным между ними. Для упрощения этой задачи могут использоваться специализированные языки описания графов задач (например, JDF – Job Description Format в PaRSEC) или инструменты/компиляторы, которые генерируют PTG из более высокоуровневых представлений (как это сделано в некоторых библиотеках, использующих PaRSEC, например, DPLASMA [18]).

1.3 САПФОР

Система автоматизированного распараллеливания программ САПФОР [14] (SAPFOR – System FOR Automated Parallelization), разрабатываемая в Институте прикладной математики им. М. В. Келдыша РАН, предназначена для автоматического преобразования последовательных программ в параллельные эквиваленты для многопроцессорных и распределенных архитектур. Основу системы составляют методы статического анализа зависимостей данных, обеспечивающие выявление параллелизма в циклических конструкциях и регулярных вычислениях, а также высокоуровневые средства аннотирования кода, позволяющие пользователям указывать области потенциального параллелизма без глубокого знания особенностей целевой платформы. Система также включает средства динамического анализа, которые дополняют статический, позволяя выявлять, конвейерные зависимости, переменные, которые можно сделать локальными для каждого потока исполнения и другие закономерности во время выполнения.

Для оптимизации итоговой параллельной программы САПФОР может выполнять различного рода преобразования исходного кода (например, преобразование замкнутых циклов, инвариантное выносы, раскрутку циклов) как автоматически, так и по запросу пользователя. Пользователь может устанавливать дополнительные свойства программы (например, гарантировать независимость данных), которые невозможно вывести чисто анализом, и САПФОР включит их в процесс распараллеливания. Таким образом, при выборе стратегии распараллеливания система комбинирует статические и

динамические методы: она автоматически строит и оценивает несколько вариантов распределения данных и вычислений (например, на основе остовных деревьев в графе использования массивов) и при необходимости использует динамическую информацию для уточнения решений.

Модель DVMH [19], на которой базируется САПФОР, дополнительно поддерживает динамическую настройку выполнения параллельных программ на выделенных ресурсах. Это позволяет реализовать адаптивные стратегии: САПФОР генерирует код с высокоуровневыми директивами (например, использование директив ACROSS для конвейеризации циклов), а DVMH runtime при исполнении может перенастраивать параметрические стратегии (количество потоков, разбивка по GPU и CPU и пр.) в зависимости от конфигурации системы. В целом, САПФОР сочетает статическое построение схем параллелизма с возможностью динамического уточнения и настройки, что обеспечивает гибкость управления исполнением и учет гетерогенности вычислительных ресурсов.

САПФОР ориентирован на работу с программами на языках C/C++ и Fortran: пользователь предоставляет последовательный код, а система выполняет преобразование его в параллельную версию с директивами DVMH. Целевым языком является расширение DVMH (C-DVMH или Fortran-DVMH), которое инкапсулирует детали MPI/OpenMP/CUDA и позволяет описывать распараллеливание через директивы распределения данных и вычислений. Программист выбирает, какие циклы или процедуры подлежат распараллеливанию, просмотреть строящийся граф зависимости массивов, а также задавать дополнительные свойства (assertions) исходного кода. Преобразования выполняются автоматически, но инициируются и контролируются пользователем. Тем не менее количество аннотаций в коде может быть небольшим: сам САПФОР может добавлять необходимые DVMH-директивы DISTRIBUTE, ALIGN, ACROSS и т. п. Пользователь же должен обеспечить, чтобы программа была «приготовлена» к

распараллеливанию (например, содержала регулярные независимые циклы или была преобразована в параллельный вид). При этом возможен и полностью автоматический вывод: для некоторых классов исходных программ САПФОР сконструировать итоговую параллельную версию без вмешательства пользователя. В остальных случаях требуется участие программиста. Пользователь должен понимать принципы распараллеливания: уметь выявлять независимые итерации циклов, правильно структурировать данные, а при необходимости вносить изменения (например, устранить зависимости).

1.4 Выводы

Существующие системы автоматического распараллеливания программ и системы автоматического конструирования программ имеют разнообразные подходы к реализации механизмов принятия решений, моделей вычисления и степени вовлеченности пользователя. Одни системы дают конкретный фреймворк, предназначенный для строго определенного класса задач. Другие системы позволяют описать программы в более общей модели и специфицировать части для распараллеливания. Принятие решений в различных системах также организуется по разному – некоторые системы обладают строго фиксированными точками принятия статических и динамических решений, другие – позволяют выбирать какие решения следует принимать статически, а какие динамически.

Тем не менее вопрос принятия решений в существующих системах автоматического распараллеливания программ и системах автоматического конструирования программ проработан не в полной мере. Существующие системы ограничены в разделении статически и динамически принимаемых решений, либо перекладывают реализацию принятия таких решений на пользователя. Требуется решение, позволяющее автоматизированно определять какие решения необходимо принять статически, а какие динамически.

2 АДАПТИВНОЕ ПРИНЯТИЕ ДИНАМИЧЕСКИХ РЕШЕНИЙ В СИСТЕМЕ АКТИВНЫХ ЗНАНИЙ

Глава посвящена описанию подхода к адаптивному динамическому принятию решений в системе активных знаний LuNA. В разделе 2.1 излагаются основные положения концепции активных знаний, лежащей в основе рассматриваемой системы. Раздел 2.2 описывает принципы функционирования системы активных знаний LuNA, а также методы исполнения программ в рамках данной системы. В разделе 2.3 формулируется постановка задачи разработки модели, обеспечивающей эффективное разделение решений на принимаемые статически и динамически. Раздел 2.4 содержит описание предлагаемой теоретической модели адаптивного принятия решений, включающей формализацию конфигураций решений, структуру пространства решений и механизм оценки их эффективности. В завершающем разделе 2.5 проводится анализ применимости модели в контексте системы LuNA и демонстрируется ее способность обеспечивать выбор достаточно эффективной промежуточной модели вычислений, соответствующей заданным нефункциональным требованиям. В конце раздела 2.5 также подводятся итоги всей главы.

2.1 Концепция активных знаний

Система LuNA [1] основана на концепции активных знаний [2], которая, развивает идеи теории синтеза параллельных программ на вычислительных моделях [20]. Ключевыми идеями в концепции активных знаний являются описание вычислительной модели, постановка задачи на данной модели и вывод решения данной задачи. Вычислительную модель можно рассматривать как конечный двудольный ориентированный граф, доли которого образуют два множества вершин, называемые множеством операций и множеством переменных. Переменные можно разделить на два типа: входные переменные операции, дуги которых ведут от самих переменных к операции и выходные

переменные операции, у которых дуги ведут от операции к переменным. Операции соответствуют некоторым программным модулям, которые вычисляют значения выходных переменных по значениям входных.

Вычислительная модель формализует описание некоторой предметной области. Свойства объектов представляются множеством переменных, значения которых соответствуют значениям этих свойств, а зависимости между свойствами, позволяющие вычислять значения одних свойств на основе других, описываются множеством операций. Например, В предметной области анализа электрических цепей переменные могут описывать такие параметры, как сила тока, напряжение, сопротивление и мощность. Операции, определенные на модели, в свою очередь, могут включать в себя вычисление напряжения на основе силы тока и сопротивления согласно закону Ома, определение общей мощности цепи по значениям напряжения и силы тока, а также другие преобразования, соответствующие физическим зависимостям в электрических системах.

Для того чтобы поставить задачу на вычислительной модели, необходимо определить два подмножества переменных вычислительной модели: входные переменные V, значения которых являются исходными данными для задачи, и выходные переменные W, значения которых должны быть получены в результате исполнения. Подграф вычислительной модели такой, что множество входных переменных V и множество выходных переменных W входят в множество переменных данного подграфа и в котором все переменные из W могут быть вычислены из V с помощью данных операций называется VW-планом.

В качестве примера можно рассмотреть рисунок 1. На нем изображена вычислительная модель, содержащая переменные X, Y, Z, T, M, P и операции A, B, C, D. На модели поставлена задача по вычислению значений выходных переменных из множества W, состоящего из переменной Z, исходя из значений входных переменных X и Y из множества V. В качестве VW-плана для решения

поставленной задачи можно рассмотреть множество переменных и операций выделенных на рисунке сплошными контурами. Переменные и операции, которые не входят в план отмечены пунктиром.

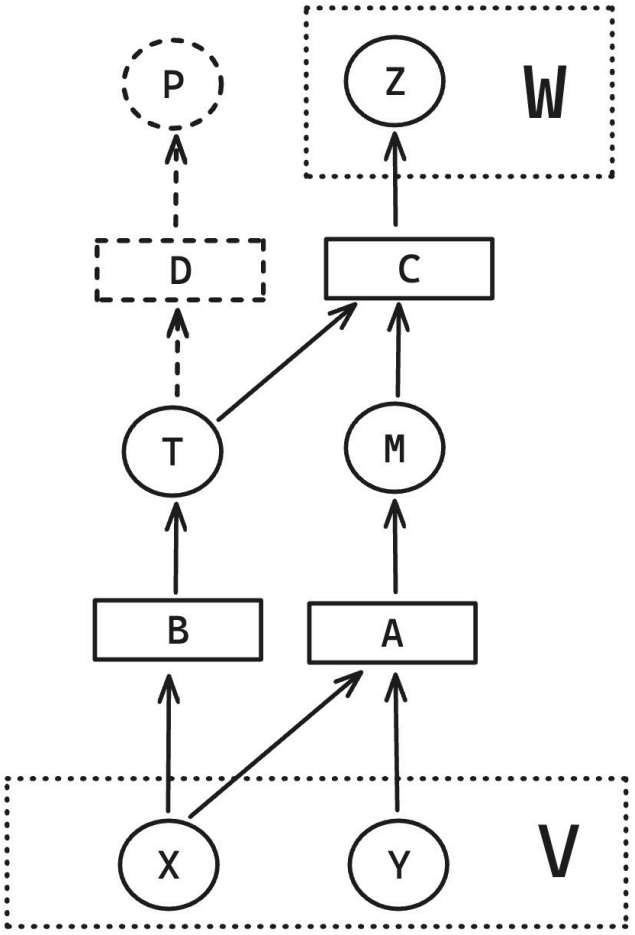

Рисунок 1 – Пример простой вычислительной модели и VW-плана

Для обеспечения возможности вычисления значений выходных переменных операции из значений ее входных переменных каждой операции в соответствие поставлен модуль.

Операция представляет собой триплет вида (in, mod, out), где

- in – конечное подмножество переменных, являющихся входными для операции;
- out – конечное подмножество переменных, являющихся выходными для операции;

- mod – программный модуль (например, последовательная подпрограмма на языке C), который производит вычисления значений выходных переменных out на основе значений входных переменных in.

Триплет, таким образом, задает возможность вычисления одних величин

предметной области (переменных) на основе других. Вычислительную модель же можно определить как множество триплетов, которые задают все возможные пути вычисления одних величин предметной области из других. Решение задачи, в таком случае, представляет собой последовательное применение вычислительных модулей mod к доступным входным переменным in, чьи значения либо заданы изначально, либо получены на предыдущих шагах. В результате каждого такого применения вычисляются значения соответствующих выходных переменных out, что создает условия для применения других операций, использующих эти переменные в качестве входных. Итеративный процесс продолжается до достижения целевого состояния, когда определены значения всех требуемых выходных переменных из множества W, что соответствует решению поставленной задачи.

2.2 Система активных знаний LuNA

Система активных знаний LuNA является реализацией концепции активных знаний. Вместе с базами активных знаний, хранящими в себе вычислительные модели и фрагменты кода и описание нефункциональных свойств этих фрагментов кода. Фрагментами кода называются программные модули, которые хранятся вместе с машинно-ориентированной спецификацией, благодаря которой система активных знаний может автоматически запускать данные модули, а также передавать входные и получать выходные аргументы между модулями. Система активных знаний позволяет на основе задачи, поставленной на вычислительной модели, автоматически сконструировать и исполнить программу, которая будет решать поставленную задачу.

Основными этапами при конструировании и исполнении программы в системе активных знаний являются:

1. вывод алгоритма – определяет множество операций, которые должны быть реализованы, то есть строит VW-план;

2. распределение ресурсов – определяет отображение на ресурсы вычислителя во времени;

3. управление исполнением – определяет задание управления, то есть в каком порядке выполнять данное множество операций.

Проходя описанные этапы, система может изменять модель вычислений программы, которая определяет множество допустимых операций, правила их применения, а также учитывает ресурсы, необходимые для вычисления. Также на данных этапах в системе активных знаний принимаются две категории решений:

- статически принимаемые решения – решения, которые выполняются на этапе конструирования программы;
- динамически принимаемые решения – решения, которые выполняются в процессе исполнения сконструированной программы.

В общем случае, по поставленной задаче система активных знаний может сконструировать большое, возможно бесконечное множество итоговых программ. Эти программы будут обладать различными нефункциональными свойствами, исходя из того какие решения были приняты основных этапах конструирования и исполнения программы. Тем не менее, для использования данного подхода на практике, необходимо учитывать эффективность того или иного решения для той предметной области, в рамках которой описана вычислительная модель и поставлена задача, и выбирать, если не оптимальную, то достаточно эффективную программу.

Эффективность программы означает, что сконструированная программа удовлетворяет заданным нефункциональным требованиям, например, время работы, потребление оперативной памяти, использование сетевых коммуникаций и т.д. В общем случае нефункциональные требования могут существенно отличаться в различных прикладных областях. Соответствие

обозначенным требованиям имеет принципиальное значение для практического применения итоговой программы и, в зависимости от заданного набора нефункциональных требований, которым должна удовлетворять сконструированная программа, принятие решений в ней следует реализовывать различным образом.

Исполнение сконструированных программ также связано с необходимостью обеспечения нефункциональных свойств. Например, программы должны эффективно адаптироваться к изменяющимся нагрузкам, перераспределяя задачи для производительности. Кроме того системе нужно учитывать особенности различных архитектур и сред исполнения, чтобы уменьшить задержки, связанные с передачей данных, и повысить использование доступных вычислительных мощностей.

В системе LuNA поддерживаются 2 основных подхода к исполнению:

1) Трансляция – построение программы на основе VW-плана и фрагментов кода, конструирующее программу в виде единственного фрагмента кода, который в свою очередь передается на исполнение. В этом подходе программа из исходной модели вычислений переводится сразу в итоговую модель вычислений, для которой есть исполнитель, реализующий эту модель. Зачастую итоговой моделью вычислений является модель вычислений аппаратного вычислителя (CPU или GPU). Весь недетерминизм разрешается во время трансляции и в выходном представлении программы весь ход вычислений строго определен.

2) Интерпретация – способ планирования, генерирования и исполнения программы, при котором фрагменты кода выполняются итеративно и каждый следующий этап (следующий шаг плана) выбирается с учетом результата выполнения предыдущего. В таком подходе система исполняет исходную модель вычислений. В случае, когда ход вычислений строго не определен, то система принимает данное решение во время исполнения. В этом смысле

система принимает все свои решения динамически. В отличие от трансляции весь недетерминизм разрешается постепенно в процессе исполнения.

Оба этих подхода обладают своими плюсами и минусами. В то время как для одной задачи эффективнее будет работать интерпретация, для другой задачи, наоборот, трансляция может быть более эффективной. Таким образом возникает необходимость в гибридном подходе, который будет объединять преимущества статического и динамического подходов.

В гибридном подходе к принятию решений часть решений принимается статически, на этапе трансляции, а остальная часть – динамически во время исполнения. Эффективность гибридного подхода зависит от того, как проведено это разделение. В этом смысле при гибридном подходе формируется промежуточная модель вычислений, в которую, принимая часть решений статически, транслируется сконструированная программа, а после исполняется данная промежуточная модель в режиме интерпретации, динамически принимая решения, которые не были приняты статически. Таким образом, чтобы эффективно применять гибридный подход, возникают вопросы для поиска наиболее подходящего промежуточного представления программы:

1) Какая промежуточная модель вычислений должна быть в конкретном случае?

2) Какие решения необходимо принимать статически, а какие динамически в выбранной модели вычислений?

3) Какую runtime-систему выбрать для исполнения полученной промежуточной модели вычислений?

2.3 Постановка задачи

Для достижения поставленной цели и ответа на обозначенные вопросы, необходимо разработать теоретическую модель, которая позволит адаптивно выбирать эффективное разделение принимаемых решений на статические и динамические в системе активных знаний LuNA.

Для этого необходимо следующее:

- Разработать формализм принятия решений, который позволит описывать принятие решений на основе концепции активных знаний.
- На основании данного формализма разработать модель, описывающую эффективность принимаемых решений. Такая модель должна описывать общий подход, подходящий для использования как с уже разработанными в рамках системы активных знаний механизмами принятия решений, так и для добавляемых в будущем.
- Описанная модель должна иметь возможность отвечать на вопрос о том, какую промежуточную модель вычислений необходимо выбрать для данной задачи. Промежуточная модель вычислений, предлагаемая на основе разработанной модели, должна быть эффективной в смысле, описанном в п. 2.2. данной главы.

2.4 Модель адаптивного принятия решений

Пусть

- $\mathcal{V}$ – конечное множество всех переменных вычислительной модели;
- $\mathcal{M}$ – конечное множество доступных программных модулей.

Тогда вычислительную модель можно представить в виде конечного множества триплетов

$$\mathcal{B} = \{\, \tau_i = (in_i,\ mod_i,\ out_i) \mid i = 1, \ldots, n \},$$

где

- $in_i \subseteq \mathcal{V}$ – набор входных переменных, необходимых модулю;
- $mod_i \in \mathcal{M}$ – программный модуль (подпрограмма), выполняющий операцию;
- $out_i \subseteq \mathcal{V}$ – набор выходных переменных, которые модуль производит.

Опр. 1. Для вычислительной модели существует множество решений, которые можно принимать при конструировании программы. Назовем такое

конечное множество **множеством решений** $\mathcal{C}$ вычислительной модели $\mathcal{B}$, а его элементы $c \in \mathcal{C}$ в свою очередь – **решениями**.

Множество решений $\mathcal{C}$ определяет все те решения вычислительной модели $\mathcal{B}$, которые можно принять при конструировании и исполнении программы. В процессе конструирования подмножество $\mathcal{S} \subseteq \mathcal{C}$ данного множества решений разбивается на две категории: статически и динамически принимаемые решения.

Опр. 2. Определим **конфигурацию** решений на вычислительной модели как отображение $\alpha \colon \mathcal{S} \to \{s, d\}$, где

- $\mathcal{S} \subseteq \mathcal{C}$ – непустое подмножество множества решений вычислительной модели;
- s – статическое принятие решения;
- d – динамическое принятие решения.

То есть конфигурация позволяет сопоставить каждому решению $c \in \mathcal{S}$ подмножества множества решений вычислительной модели стратегию принятия решения $\alpha(c)$. Таким образом, конфигурация описывает разделение решений на две категории.

Стоит отметить, что не все подмножества множества решений вычислительной модели $\mathcal{S} \subseteq \mathcal{C}$ позволяют на их основе сконструировать программу. Тем не менее, без ограничений общности, для практической применимости модели и для упрощения изложения далее под $\mathcal{S}$ будем подразумевать такое подмножество решений, используя которые система может сконструировать программу, решающую некоторую поставленную задачу.

В рамках модели необходимо иметь некоторую структуру над множеством конфигураций решений, которая позволит сравнивать конфигурации между собой. Для этого необходимо определить некоторое отношение между конфигурациями. Данное отношение, в свою очередь, позволяет упорядочить их

согласно заданному критерию и, следовательно, структурировать множество конфигураций.

Для конфигурации α одним из ключевых ее свойств является динамичность. Свойство динамичности конфигурации отражает то, насколько много решений данная конфигурация принимает в процессе исполнения сконструированной программы.

Опр. 3. Введем **уровень динамичности** – отношение частичного порядка $\preceq$ на множестве $\mathcal{A} = \{\alpha\}$ всех таких конфигураций решений на вычислительной модели:

$$\alpha \preceq \beta : \ \alpha, \beta \in \mathcal{A}$$
$$\Updownarrow$$
$$\{ c \in \mathcal{S} \mid \alpha(c) = d \} \ \subseteq \ \{ c \in \mathcal{S} \mid \beta(c) = d \}$$

Таким образом, конфигурация β считается “не менее динамичной”, если динамически принимаемые решения в ней назначены не реже, чем в конфигурации α.

С помощью описанного отношения частичного порядка задается структура множества $\mathcal{A}$ конфигураций решений на вычислительной модели.

Опр. 4. Назовем **картой** частично упорядоченное по уровню динамичности конечное множество конфигураций $(\mathcal{A}, \preceq)$.

Стоит отметить, что:

- наименьшим элементом в карте является полностью статическая конфигурация;
- наибольшом элементом в карте – полностью динамическая конфигурация.

Далее, для удобства и простоты, под обозначением $\mathcal{A}$ будем подразумевать карту, а не просто множество конфигураций решений на вычислительной модели.

Карта $\mathcal{A}$ образует решетку, где

- объединение $\alpha \cup \beta$ – конфигурация, в которой динамические принятия решений назначены для всех решений, где хотя бы одна из конфигураций α, β динамична;
- пересечение $\alpha \cap \beta$ – конфигурация, в которой динамические принятия решений назначены для всех решений, где обе конфигурации α, β динамичны.

Опр. 5. Наличие операций объединения и пересечения на множестве конфигураций позволяет ввести понятие **соседних конфигураций** – пар конфигураций, отличающихся выбором стратегии принятия ровно одного решения. Формально, конфигурации α и β называются соседними, если существует такое решение $c \in \mathcal{S}$, что $\alpha(c) \neq \beta(c)$, а для всех других решений $c' \in \mathcal{S} \setminus \{c\}$ выполнено равенство $\alpha(c') = \beta(c')$. В этом случае конфигурации α и β связаны ребром на решетке, и переход между ними соответствует изменению категории принятия решения $c \in \mathcal{S}$ с динамической на статическую или наоборот. Понятие соседних конфигураций позволяет выполнять поиск эффективных конфигураций с помощью жадных или эвристических алгоритмов, последовательно исследуя карту без полного перебора всех ее элементов.

Еще одним важным компонентом модели адаптивного динамического принятия решений является аппарат, обеспечивающий формальное описание эффективности принимаемых решений. Данный аппарат, основанный на описании нефункциональной характеристики программы, позволяет осуществлять выбор эффективной конфигурации решений на вычислительной модели.

Опр. 6. Определим **характеристику** конфигурации как функционал следующего вида $e : \mathcal{A} \to \mathbb{R}$. Характеристика $e(\alpha)$ конфигурации α является формализацией некоторой нефункциональной характеристики программы, сконструированной по данной конфигурации. Такими

характеристиками, например, могут выступать время исполнения, объем используемой памяти или степень распараллеливания.

2.5 Анализ предлагаемой модели

Покажем, что предлагаемая модель действительно позволяет отвечать на вопрос о том, какую промежуточную модель вычислений необходимо выбрать для некоторой задачи, поставленной на вычислительной модели и что промежуточная модель вычислений, предлагаемая на основе данной модели, достаточно эффективна, то есть удовлетворяет таким заданным нефункциональным требованиям, как, например, время исполнения или потребление оперативной памяти.

Для этого в терминах, предлагаемой в предыдущем пункте, теоретической модели опишем следующее:

- крупноблочные решения, принимаемые в системе активных знаний;
- множество промежуточных моделей вычислений программ в системе активных знаний;
- нефункциональные требования, предъявляемые к программе;
- алгоритм выбора достаточно эффективной промежуточной модели вычислений, основываясь на предъявленных нефункциональных требованиях.

Как уже было сказано в п. 2.2., в системе активных знаний переход от вычислительной модели, являющейся исходной моделью вычислений, к исполнению программы в итоговой модели вычислений происходит в три основных этапа: вывод алгоритма, распределение ресурсов и управление исполнением. Обозначим данные 3 этапа как основные крупноблочные решения, которые будут приниматься на вычислительной модели.

Таким образом множество решений $\mathcal{C}$ будет иметь вид $\mathcal{C} = \{c^{alg}, c^{res}, c^{ctrl}\}$, где

- c^{alg} – решение о выводе алгоритма. Статическое решение означает, что VW-план строится на этапе трансляции программы, а динамическое решение, что план будет выводиться итеративно на этапе интерпретации.
- c^{res} – решение о распределение ресурсов. Статическое решение закрепляет фрагменты кода за узлами на этапе трансляции. Динамическое решение является балансированием нагрузки в процесс исполнения.
- c^{ctrl} – решение об управлении исполнением. Статическое решение закрепляет последовательность выполнения фрагментов кода при конструировании программы, динамическое решение, в свою очередь, позволяет определять какой фрагмент кода следует выполнить следующим на основе готовности входных данных во время исполнения.

Карта $\mathcal{A}$ на вычислительной модели с данным множеством решений, по существу, является множеством промежуточных моделей вычислений программы, конструируемой системой активных знаний.

В качестве характеристики конфигурации $e(\alpha)$ возьмем время работы программы с конфигурацией α. На практике такую характеристику можно задать в виде таблице соответствия конфигураций и времени исполнения программы, сконструированной по данной конфигурации. Время исполнения программы в системе активных знаний может собираться, например, с помощью компонента профилирования, который позволяет проводить динамический съем характеристик исполнения.

Таким образом задачу поиска оптимальной конфигурации можно сформулировать как задачу минимизации характеристики $e(\alpha) \to \min$. В общем виде такая задача является труднорешаемой. Тем не менее, на практике вместо поиска оптимальной конфигурации достаточно будет найти такую эффективную конфигурацию, которой будет достаточно для решения прикладной задачи. Для решения данной задачи можно использовать простейший жадный алгоритм обхода по решетке.

Опишем этапы данного алгоритма:

1. Инициализация: выбирается некоторое множество так называемых “опорных” конфигураций. Каждая из таких “опорных” конфигураций α_0 будет являться первым приближением достаточно эффективной конфигурации.

2. Итерации: На каждом шаге рассматриваются конфигурации β, являющиеся соседними по решетке для конфигурации α_i. Среди них выбирается конфигурация α_{i+1} такая, что характеристика $e(\beta)$ минимальна среди всех соседних конфигураций. Если характеристика $e(\alpha_i)$ меньше всех характеристик соседних конфигураций, то итерации заканчиваются. Иначе алгоритм переходит к следующей итерации, где рассматриваются все соседи по решетке для конфигурацию α_{i+1}.

3. Итоговая оценка: после окончания итераций для каждой из опорных конфигураций среди них, аналогично шагу итерации, с помощью характерстики $e(\alpha)$ выбирается итоговая достаточно эффективная конфигурация α.

Данный алгоритм обеспечивает монотонное улучшение характеристики, но не всегда находит глобальный минимум. Впрочем, для решения прикладных задач в системе активных знаний LuNA он позволяет получить достаточно эффективную конфигурацию, удовлетворяющую компромиссу между сложностью поиска и качеством.

Таким образом, в данной главе представлена теоретическая основа для построения механизма адаптивного динамического принятия решений в системе активных знаний LuNA. Концепция активных знаний позволяет на основе формализации вычислительной модели и постановки задачи на ней определить множество решений, принимаемых в процессе конструирования и исполнения программ. Предложенное описание конфигурации в виде отображения, разделяющего решения на статические и динамические, позволяет ввести понятие карты и структурировать множество конфигураций в виде решетки уровню их динамичности. Введенная характеристика дает

возможность формализовать критерии эффективности программы, а также предложить способ выбора достаточно эффективной конфигурации с учетом заданных нефункциональных требований. Разработанная структура конфигураций решений, организованная в виде решетки по уровню динамичности, позволяет реализовать алгоритмы поиска эффективной конфигурации по заданному критерию эффективности. Проведенный анализ подтверждает применимость разработанной теоретической модели в рамках системы активных знаний LuNA.

3 КОМПОНЕНТ “ИСПОЛНИТЕЛЬ”

На текущем этапе технологического развития проекта системы активных знаний LuNA ей необходим компонент, на базе которого будет возможно внедрение реализаций алгоритмов адаптивного динамического принятия решений и механизмов поддержки принятия динамических решений соответствующими runtime системами.

В разделе 3.1 описана общая архитектура системы активных знаний LuNA и место Исполнителя в ней. В разделе 3.2 представлена архитектура компонента “Исполнитель”. В разделе 3.3 описаны детали реализации частей архитектуры “Менеджер” и “Машина”. В разделе 3.4 описано тестирование реализации компонента. Наконец, в разделе 3.5 подводятся итоги главы. Также для реализованного компонента в приложении А приведено руководство оператора, а в приложении Б – описание программы.

3.1 “Исполнитель” в архитектуре системы LuNA

Архитектура системы активных знаний LuNA представлена на рисунке 2. В целом система LuNA состоит из набора отдельных компонентов-сервисов, каждый из которых отвечает за свою часть функциональности всей системы.

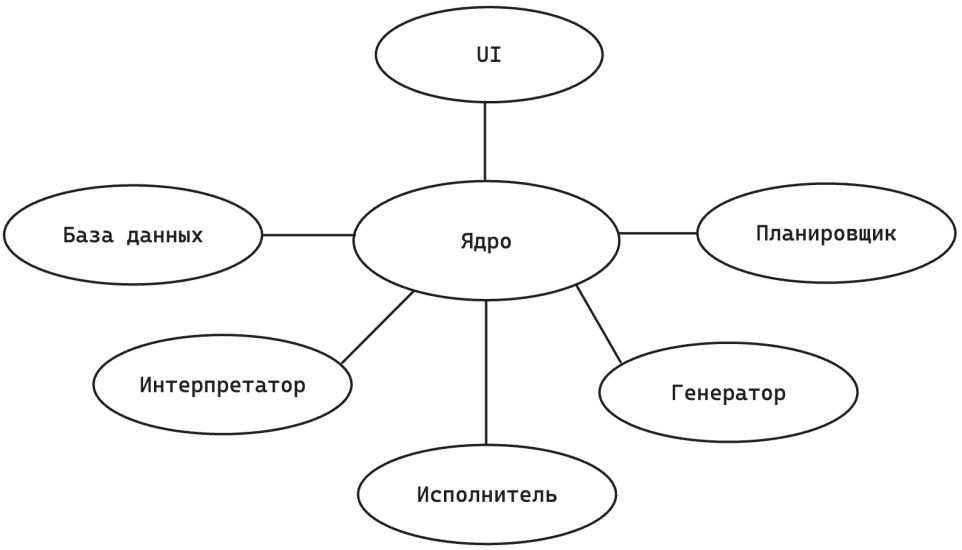

Рисунок 2 – Архитектура системы активных знаний LuNA

Рассмотрим подробнее некоторые компоненты системы, которые тем или иным образом взаимодействуют с компонентом “Исполнитель”.

Ядро функционирует как центральный компонент системы, ответственный за организацию взаимодействия между остальными компонентами. Оно предоставляет унифицированный интерфейс, позволяющий абстрагировать конкретные реализации компонентов друг от друга. В рамках данной архитектуры исполнитель поддерживает связь исключительно с ядром, получая через него все входящие запросы от других компонентов и аналогично выполняя исходящие запросы к другим компонентам. При дальнейшем описании архитектуры системы следует учитывать, что любые упоминания о непосредственном взаимодействии между компонентами фактически подразумевают их опосредованную коммуникацию через ядро. Такое соглашение принято для упрощения изложения, но не отменяет роль ядра в качестве единого канала всех системных взаимодействий.

База данных, используемая в системе активных знаний LuNA, представляет собой структурированное хранилище, предназначенное для поддержки автоматического применения прикладных модулей при решении задач. В её состав входит библиотека фрагментов кода, содержащая программные модули с машино-ориентированной спецификацией, которая обеспечивает возможность их автоматизированного исполнения. Каждый модуль описан с указанием входных и выходных параметров, а также сопровождается информацией о его нефункциональных характеристиках, таких как время выполнения, потребление памяти и другие параметры, влияющие на эффективность исполнения. Библиотека обеспечивает возможность использования модулей различной природы, включая статические библиотеки, исполняемые файлы и внешние сервисы, что достигается путем стандартизации форм представления и обработки. Также база данных содержит хранилище значений переменных, предназначенное для накопления и повторного использования данных, полученных в ходе экспериментов, численного

моделирования или аналитических расчётов. Эти значения привязаны к переменным вычислительных моделей и могут быть использованы при формулировке задач и в процессе исполнения, с учетом их характеристик и ограничений.

Генератор является компонентом системы активных знаний, предназначенным для автоматического конструирования исполняемой программы по заданному VW-плану. В рамках своей функциональности генератор трансформирует абстрактное представление решения в конкретную программную реализацию, ориентированную на заданную архитектуру вычислителя. Конструируемая программа может быть как последовательной, так и параллельной, в зависимости от степени независимости операций, входящих в план. В процессе генерации могут учитываться особенности целевой платформы, включая типы процессоров, наличие специализированных ускорителей, специфику распределенной памяти и другие параметры архитектуры вычислителя. Генератор может быть оснащен средствами оптимизации, позволяющими учитывать нефункциональные характеристики и повышать эффективность исполняемого кода. Таким образом, генератор обеспечивает переход от постановки задачи к ее конкретной реализации, соответствующей как функциональным, так и нефункциональным требованиям.

Интерпретатор выполняет функции исполнения VW-плана в режиме интерпретации и предназначен для выполнения фрагментов кода на основе текущего состояния вычислительной модели и доступных значений переменных. В отличие от статически сгенерированных программ, интерпретатор позволяет осуществлять динамическое принятие решений в процессе выполнения, позволяя адаптировать к изменяющимся условиям исполнения, включая загрузку ресурсов или характеристики входных данных. Работа интерпретатора организуется как итеративный процесс, в рамках которого вычислительные ресурсы по мере их освобождения используются для выполнения доступных операций, соответствующих текущему фронту

вычислений. Для этого на каждом шаге интерпретатор взаимодействует с планировщиком, формируя актуальный подграф VW-плана и выбирая следующую операцию для исполнения. Интерпретатор позволяет учитывать локальность данных, минимизировать издержки на перемещение данных между узлами, оптимизировать распределение задач и обеспечивать устойчивое поведение при частичных сбоях или непредсказуемых изменениях в конфигурации среды исполнения.

Исполнитель представляет собой компонент системы LuNA, непосредственно реализующий запуск фрагментов кода на доступных вычислительных ресурсах в соответствии с VW-планом. Его основная задача заключается в организации выполнения операций вычислительной модели, обеспечивая передачу входных данных, вызов соответствующих модулей и получение выходных значений. Взаимодействие исполнителя с другими компонентами системы, включая интерпретатор и генератор, обеспечивает согласованность процессов вычисления и возможность реагирования на изменения состояния среды исполнения. В зависимости от стратегии исполнения исполнитель может функционировать как в рамках строго заданной последовательности, так и в условиях динамического выбора задач, что особенно актуально при работе на гетерогенных и распределенных платформах. Эффективность работы исполнителя определяется способностью учитывать архитектурные особенности целевого вычислителя, включая иерархии памяти, особенности синхронизации и механизмы управления потоками. При этом исполнитель может обеспечивать как однопоточную реализацию, так и многопоточное или распределенное исполнение с соответствующей координацией между узлами. Таким образом, исполнитель завершает процесс исполнения задачи, реализуя конкретные действия, определенные в вычислительной модели.

3.2 Архитектура компонента

Архитектура компонента Исполнитель разрабатывалась с учетом следующих требований к реализации: расширяемость, поддержка различных уровней управления исполнением, а также возможность получения информации о состоянии вычислительных машин. Указанные требования обусловлены необходимостью применения системы активных знаний в различных предметных областях, отличающихся по специфике вычислительных сред и типам используемых программных модулей.

Для удовлетворения этих требований спроектирована распределенная архитектура Исполнителя, в рамках которой выделяются две ключевые сущности: менеджер и машина. Схема архитектуры представлена на рисунке 3.

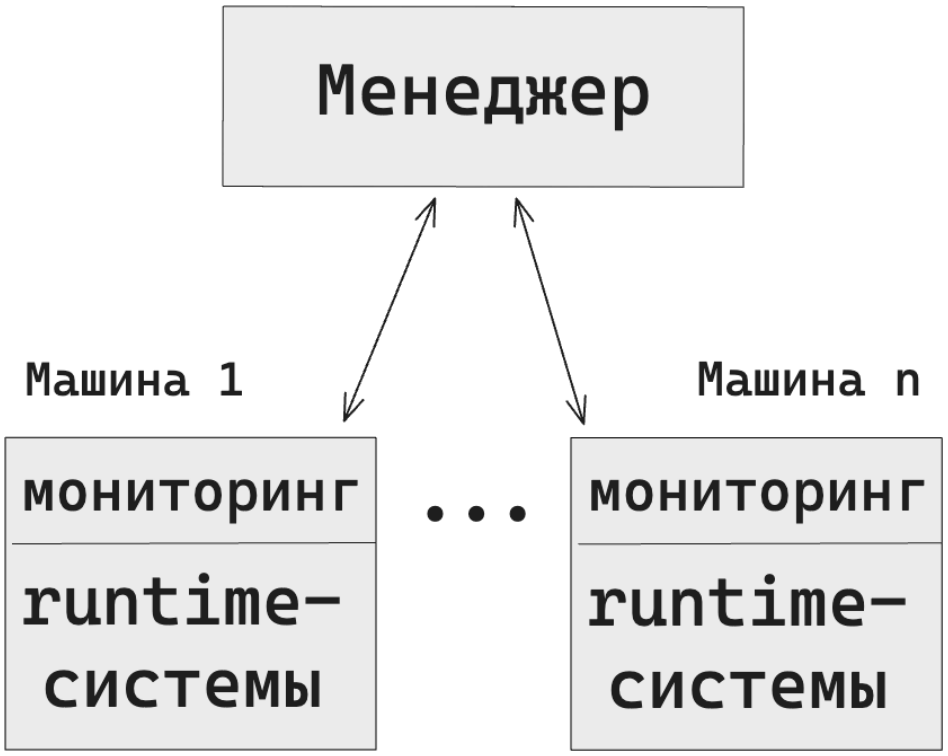

Рисунок 3 – схема архитектуры компонента “Исполнитель”

Менеджер является центральной частью архитектуры, которая обеспечивает взаимодействие между системой активных знаний и компонентом Исполнителя через определенный JSON RESTful API интерфейс. Он осуществляет сбор и агрегирование информации о состоянии всех машин,

принимает решения о выборе подходящей runtime-системы для исполнения операций и управляет процессом исполнения на глобальном уровне. Таким образом, менеджер выполняет функции координации, планирования и маршрутизации задач, назначаемых на конкретные вычислительные ресурсы.

Машина представляет собой компонент, развертываемый на каждом узле. Она отвечает за мониторинг состояния, подготовку вычислительной среды, а также непосредственное выполнение операций. Внутренняя структура машины подразделяется на две части: систему мониторинга и runtime-систему. Такое разделение обеспечивает модульность и упрощает адаптацию компонента для различных платформ. В частности, наличие обособленного мониторинга позволяет использовать существующие решения при работе с кластерами или внедрять специализированные механизмы для других типов машин. Выделение runtime-систем предоставляет возможность интеграции новых механизмов исполнения без необходимости модификации остальной части системы.

Распределенная архитектура компонента Исполнитель обеспечивает гибкость, необходимую для поддержки различных сценариев исполнения. При работе с Генератором Исполнитель получает на вход программу, сгенерированную на основе VW-плана, и запускает её в виде монолитного фрагмента кода. При работе с Интерпретатором выполнение осуществляется итеративно, где каждая операция отправляется на запуск интерпретатором по мере исполнения задачи. В этом случае компонент Исполнитель предоставляет информацию о доступных машинах и их состоянии, что необходимо для выбора запуска очередной операции.

Таким образом, предложенная архитектура обеспечивает выполнение всех предъявляемых требований, сохраняя при этом модульность и расширяемость, необходимые для масштабируемого и адаптивного применения системы активных знаний в разнообразных вычислительных средах.

3.3 Детали реализации “Исполнителя”

Для реализации компонента “Исполнитель” был выбран язык программирования Go [21]. Выбор языка Go обусловлен рядом факторов:

- Язык характеризуется лаконичным и прозрачным синтаксисом, что облегчает ознакомление с кодовой базой и способствует вовлечению новых участников в процесс разработки;
- Go является компилируемым языком с поддержкой кросс-компиляции, что значительно упрощает развертывание программного обеспечения в различных вычислительных средах, устраняя необходимость в дополнительной настройке окружения и минимизируя проблемы, связанные с переносимостью;
- Язык Go предоставляет высокоуровневые средства для организации параллельного и конкурентного выполнения задач – в первую очередь за счет примитивов в виде горутин (goroutines) и каналов (channels). Это позволяет реализовать архитектуру, включающую взаимодействие менеджером и машинами без необходимости использовать внешние библиотеки или усложненные модели синхронизации;
- Наличие развитой стандартной библиотеки и широкой экосистемы сторонних пакетов позволяет использовать готовые решения и тем самым ускорить процесс разработки. Язык изначально ориентирован на разработку микросервисов и распределенных систем. Встроенные пакеты, такие как *net/http*, *net/rpc*, *context*, *encoding/json* и другие позволяют создавать веб-сервисы без необходимости привлечения сторонних фреймворков, что упрощает структуру кода и снижает количество зависимостей.

Протокол взаимодействия между менеджером и машинами построен на сочетании удаленного вызова процедур (RPC) и протокола передачи гипертекста (HTTP), где каждый из них выполняет специализированные функции в зависимости от типа коммуникации. Такое разделение протоколов повышает модульность системы, упрощает масштабирование за счет

оптимизации внутренних операций и обеспечивает надежность при взаимодействии с внешними компонентами.

Протокол HTTP используется для внешних взаимодействий обеспечивая стандартизированный универсальный формат обмена данными. Протокол HTTP API реализован в соответствии с принципами REST [22] с передачей данных в формате JSON [23], где ресурсы идентифицируются через маршруты (например, */execute* для выполнения задач), а методы HTTP (POST, GET) определяют тип операций.

Протокол RPC, напротив, применяется для внутренних коммуникаций, таких как регистрация машин и мониторинг их состояния, обеспечивая эффективный обмен данными между компонентами системы. Преимущества RPC в реализации заключаются в его простоте, низких накладных расходах и высокой производительности при передаче управляющих сообщений, что упрощает интеграцию и поддержку внутренних процессов. Реализация протокола RPC основана на пакете *net/rpc* [24] стандартной библиотеки Go. Этот пакет обеспечивает создание серверов и клиентов для асинхронного обмена данными через транспортный протокол TCP, что гарантирует надежную доставку сообщений с минимальными накладными расходами. Данные сериализуются с использованием формата gob [25], который позволяет компактно и эффективно кодировать структуры данных, автоматически обрабатывая их преобразование для передачи между узлами. Сервисы описываются как структуры, содержащие методы, доступные для удаленного вызова, с регистрацией на сервере через *rpc.Register*. Каждый метод должен соответствовать определенному формату сигнатуры, принимая два аргумента (запрос и указатель на ответ) и возвращая ошибку, что обеспечивает единообразие и простоту интеграции.

3.3.1 Описание реализации подкомпонента “Менеджер”

Исполнение программы менеджера начинается с инициализации конфигурации, создания и запуска HTTP-сервера. Программа начинается с обработки пути к конфигурационному файлу, который может быть задан через аргументы командной строки или переменную окружения *EXECUTOR_MANAGER_CONFIG_PATH*. Если путь не указан, используется значение по умолчанию – *config.yaml*. Структура конфигурации *ManagerConfig* содержит хост и порт, на котором будет запущен HTTP-сервер. После загрузки конфигурации создается экземпляр HTTP-сервера с помощью функции *server.New*, которая инициализирует маршрутизатор (*http.ServeMux*) для обработки запросов. Основными путями являются */rpc* для взаимодействия с машинами через RPC, */execute* для получения запросов на выполнение фрагментов кода и */machines_info* для получения информации о зарегистрированных машинах.

Основном объектом менеджера является сервис, отвечающий за регистрацию машин, их мониторинг и распределение задач.

Листинг 1 – структура описания информации о Машине

```
type Machine struct {
   ID      string
   Address string
   Info    *sysinfo.SystemInfo
}
```

Сервис хранит информацию о машинах в словаре *machines*, где ключом является уникальный идентификатор машины, а значением – структура *Machine,* представленная на листинге 1, которая содержит уникальный идентификатор машины, ее адрес и системную информацию об узле, на котором запущена машина.

Регистрация новой машины выполняется методом *RegisterMachine*, который генерирует уникальный идентификатор машины в рамках сервиса и

добавляет ее в словарь. Для обеспечения потокобезопасности доступ к словарю осуществляется с использованием мьютекса. Метод *GetMachine* возвращает машину, выбранную с помощью экземпляра интерфейса *MachineChooser*, который реализует стратегию выбора машины для выполнения задачи. В случае отсутствия зарегистрированных машин возвращается ошибка. Для мониторинга состояния машин используется метод *StartPingingMachines*, который периодически отправляет запросы на проверку доступности (*Ping*) через RPC. Если машина не отвечает, она удаляется из словаря доступных машин.

Обработка запросов на выполнение фрагментов кода осуществляется через маршрут */execute*, реализованный в функции *executeHandler*. Логика работы функции заключается в следующем:

1. Получение доступной машины с помощью метода *service.GetMachine*.
2. Создание HTTP-запроса к выбранной машине с пересылкой тела исходного запроса и заголовков.
3. При получении результата исполнения от машины передача заголовков и тела ответа от машины клиенту, с обработкой возможных ошибок.

Функция *executeHandler* передает запрос на выполнение фрагмента кода выбранной машине, не вмешиваясь в процесс его обработки, что обеспечивает распределенную архитектуру системы. В случае ошибок, таких как отсутствие доступных машин или сбой при выполнении запроса, клиенту возвращается JSON-ответ с описанием ошибки и соответствующим HTTP-статусом.

Маршрут */machines_info* реализован в функции *machinesInfoHandler* и предоставляет информацию о зарегистрированных машинах. Метод *service.GetMachinesInfo* возвращает словарь, содержащий системную информацию о каждой машине, включая данные о процессоре, памяти, графических устройствах и сетевых интерфейсах. Структура ответа сериализуется в JSON-формат для передачи клиенту.

Для проверки работоспособности менеджера реализован маршрут */healthcheck*, который возвращает HTTP-статус 200 и текстовое сообщение *“aks-executor is OK”*. Это позволяет внешним системам проверять состояние менеджера.

3.3.2 Описание реализации подкомпонента “Машина”

Программа машины, аналогично менеджеру, начинается с загрузки конфигурации с использованием функции *config.LoadMachineConfig*, которая обрабатывает параметры сервера, менеджера, хранилища фрагментов кода и хранилища значений. Структура конфигурации *MachineConfig* представлена в листинге 2.

Листинг 2 – структуры, описывающие конфигурацию Машины

```
type MachineConfig struct {
    Server          Server          `mapstructure:"server"`
    Manager         ManagerClient   `mapstructure:"manager"`
    FragmentStorage FragmentStorage `mapstructure:"fragment_storage"`
    ValueStorage    ValueStorage    `mapstructure:"value_storage"`
}

type ManagerClient struct {
    Address        string        `mapstructure:"address"`
    RequestTimeout time.Duration `mapstructure:"request_timeout"`
}

type FragmentStorage struct {
    Address        string        `mapstructure:"address"`
    RequestTimeout time.Duration `mapstructure:"request_timeout"`
}
```

```
type ValueStorage struct {
    Address        string        `mapstructure:"address"`
    RequestTimeout time.Duration `mapstructure:"request_timeout"`
}
```

После загрузки конфигурации машина регистрируется в менеджере с помощью функции *managerrpc.NewClient* и метода *RegisterMachine*. Для этого собирается системная информация, включающая данные о процессоре, памяти, графических устройствах и сетевых интерфейсах. Регистрация передает адрес машины и ее системные характеристики менеджеру, что позволяет включить её в пул доступных узлов. Затем создается HTTP-сервер с помощью функции *server.New*, который обрабатывает запросы по маршрутам */rpc* и */execute*.

Основная функциональность машины реализована в обработчике запросов */execute*, определенном в функции *executeHandler*. Логика обработки запроса включает следующие этапы:

1. Десериализация JSON-запроса в структуру *executeRequest*, содержащую информацию о фрагменте кода, входных и выходных параметрах, а также локальные хранилища фрагментов и значений (если переданы).
2. Создание агрегатора хранилищ значений *AggregatedValueStorage*, который объединяет локальное хранилище запроса, хранилище ответа и удаленное хранилище, определенное в конфигурации.
3. Инициализация провайдера фрагментов кода *FragmentProvider* в зависимости от типа хранилища, указанного в запросе (request или remote).
4. Исполнение фрагмента кода с использованием функции *exec.Execute*.
5. Сериализация результатов выполнения и отправка ответа клиенту.

Модуль *internal/valstore* обеспечивает работу с хранилищами значений, поддерживая три типа: локальное хранилище запроса *request*, локальное хранилище ответа *response* и удаленное хранилище *remote*. Агрегатор

AggregatedValueStorage позволяет машине обращаться к нужному хранилищу в зависимости от идентификатора, указанного в запросе. Для удаленного хранилища используется HTTP-клиент, обращающийся к библиотеке фрагментов кода по адресу из конфигурации.

Интерфейс провайдера фрагментов кода *FragmentProvider*, позволяет получать фрагменты кода либо из локального хранилища запроса, либо из удаленного сервиса. В качестве возвращаемого значения провайдер использует обобщенный интерфейс для всех фрагментов кода, описанный в листинге 3.

Листинг 3 – Общий интерфейс всех фрагментов кода

```
type Fragment interface {
Run(inputs []Input, outputs []OutputDescription) ([]Output, error)
Cleanup() error
}
```

Конкретные структуры фрагментов кода, например *CppFragment* или *LunaFragment*, определяют конечный список полей фрагмента и реализуют методы *Run* и *Cleanup* для настройки окружения и runtime-системы, исполнения фрагмента кода, и в случае необходимости очистки мусора после исполнения. В листинге 4 представлен пример структуры *CppFragment*, используемой для представления фрагментов кода на языке C++.

Листинг 4 – структура данных представления C++ фрагмента

```
type CppFragment struct {
   Include       string   `json:"include"`
   Function      string   `json:"function"`
   FunctionName  string   `json:"name"`
   InputVarTypes  []string `json:"input_var_types"`
   OutputVarTypes []string `json:"output_var_types"`
}
```

Функция *exec.Execute* из пакета *internal/exec* является ключевой для всего процесса исполнения фрагментов кода на машине. Алгоритм, реализуемый

данной функцией, спроектирован для обеспечения универсальной обработки фрагментов, независимо от их конкретного типа, что делает его применимым ко всем видам фрагментов кода, которые реализуют общие интерфейсы. Опишем основные этапы алгоритма.

Получение фрагмента кода: функция принимает объект *FragmentProvider*, который предоставляет доступ к фрагменту кода. Провайдер абстрагирует вызов источник фрагмента и возвращает объект, реализующий интерфейс *Fragment*.

Сбор входных данных: функция получает массив структур входных спецификаций *InputSpec*, представленных на листинге 5, содержащих идентификаторы хранилищ и значений, а также идентификаторы параметров фрагмента. Для каждого входного параметра функция запрашивает значение из агрегированного хранилища значений. Значения собираются в массив структур *Input*, где каждый элемент связывает идентификатор параметра с соответствующим значением. Описание структуры *Input* дано на листинге 5.

Листинг 5 – структуры данных входного значения

```
type InputSpec struct {
StorageID string `json:"storage_id"`
ValueID   string `json:"value_id"`
ParamID   string `json:"param_id"`
}

type Input struct {
ParamID string
Value   any
}
```

Подготовка выходных параметров: на основе массива выходных спецификаций *OutputSpec* формируется список описаний выходных параметров

OutputDescription, содержащих идентификатор параметра. Обе структуры представлены на листинге 6.

Листинг 6 – структуры данных выходного значения

```
type OutputSpec struct {
StorageID string `json:"storage_id"`
ParamID   string `json:"param_id"`
Action    string `json:"action"`
}

type OutputDescription struct {
ParamID string
}

type Output struct {
OutputDescription
Value string
}
```

Выполнение фрагмента кода: функция вызывает метод *Run* интерфейса *Fragment*, передавая ему собранные входные данные и описания выходных параметров. Метод *Run* реализует логику выполнения фрагмента, которая зависит от его типа, и возвращает массив результатов *Output*. Структура *Output,* представленная на листинге 6, содержит соответствующий идентификатор параметра и значение, полученное в процессе выполнения.

Обработка выходных данных: Для каждого сохраняемого значения создается запись в хранилище, и формируется структура *ValueLocation*, содержащая идентификатор хранилища и значения. Код структуры дан на листинге 7. Для каждого результата анализируется *OutputSpec*, определяющая действие *action*, которое необходимо применить к выходному значению:

- Если указано действие *skip*, значение игнорируется и не сохраняется;
- Если указано действие *store*, значение сохраняется в указанном хранилище (локальном или удаленном) с созданием уникального идентификатора;
- Если указано действие *store_content*, предполагается, что значение представляет путь к файлу, содержимое которого считывается и кодируется в base64 перед сохранением в хранилище.

Очистка ресурсов: после выполнения фрагмента вызывается метод *Cleanup* интерфейса *Fragment*, который отвечает за освобождение временных ресурсов, таких как созданные файлы или директории, что обеспечивает корректное завершение работы и предотвращает накопление временных данных.

Формирование результата: функция возвращает структуру *ValueLocationMap*, содержащую идентификаторы выходных значений, связанных с их хранилищами. Если в процессе выполнения возникли ошибки, функция возвращает соответствующую ошибку, которая обрабатывается вызывающей стороной.

Листинг 7 – структуры данных для хранимых значений

```
type ValueLocation struct {
StorageID string `json:"storage_id"`
ValueID   string `json:"value_id"`
}

type ValueLocationMap map[string]ValueLocation
```

3.4 Тестирование

Целью тестирования являлась проверка корректности работы компонента “Исполнитель” в целом, а также проверка работоспособности Исполнителя при

взаимодействии с другими компонентами системы. Для этого было проведено два теста.

Целью первого теста является проверка корректности обработки и исполнения пользовательского фрагмента кода, предоставленного в составе запроса, а также корректной работы механизма сопоставления входных и выходных параметров, заданных в явной форме. Содержание запроса описывает задачу, в рамках которой исполнитель должен выполнить пользовательский фрагмент кода, реализованный на языке C++ и предназначенный для повторения входной строки заданное число раз. В качестве входных данных задаются: символьная строка *str* со значением *"repeatme"* и целое число *cnt*, определяющее количество повторений. На выходе ожидается строка, содержащая конкатенацию входной строки указанное количество раз. Таким образом, проверяется не только сам факт исполнения, но и корректность возврата значения в заданное место хранилища выходных данных. Такая постановка задачи теста служит минимальной воспроизводимой единицей для верификации функциональности обработки произвольных пользовательских фрагментов.

Тестирования проводилось на MacBook Air с 8 ядерным процессором M1 и 16 гигабайт оперативной памяти. В рамках теста были запущены менеджер и 3 экземпляра программы машины – 1 экземпляр на операционной системе хоста (MacOS) и 2 экземпляра в виртуальных машинах с Linux, созданных с помощью QEMU [26]. Компонент при получении запроса, содержащего 3 экземпляра описанной задачи, должен выполнить следующие действия:

- Распознать и интерпретировать описание фрагмента кода, содержащегося в разделе *fragment_storage*;
- Подготовить и корректно связать входные значения, переданные в разделе *value_storage*, с параметрами функции;
- Произвести подготовку и настройку среды исполнения;

- Выполнить вызов функции с заданными аргументами и сохранить результат в выходное хранилище в соответствии с параметром *outputs*.

Результатом выполнения теста являлся полученный ответ с корректными значениями трех выходных переменных, представляющих собой строки *"repeatmerepeatmerepeatme"*. Данный результат свидетельствует о корректной работе механизма обработки задания, привязки параметров, исполнения и возврата результата. С помощью мониторинга процессов на каждом из хостов было проверено, что задачи были верно распределились между подключенными хостами.

Для проверки взаимодействия с другими компонентами системы был проведен второй тест. Поставленная задача заключалась в вычислении корреляционной свертки некоторых данных сейсмотрасс. Входными данными фрагмента кода являются *filename1* – текстовый файл со списком сейсмотрасс, *filename2* – файл опорного сигнала в формате PC-A, *T* – интервал корреляции в секундах. Помимо пунктов, проверенных в первом тесте, в данном случае главным образом проверялось взаимодействие исполнителя с другими компонентами системы.

Тестирования проводилось на специально выделенном для развертывания всех компонент системы активных знаний LuNA сервере под управлением операционной системы Linux (дистрибутив Ubuntu) с 8 ядрами и 8 гигабайтами оперативной памяти. В рамках теста были запущены менеджер и экземпляр программы машины. Компонент, помимо действий из первого теста, должен был выполнить следующее:

- Запросить и обработать фрагмент кода у библиотеки фрагментов кода, находящейся по адресу из *fragment_storage*;
- Запросить и подготовить входные значения, полученные от хранилища значений по адресу из *value_storage*;
- Передать результат исполнения в хранилище значений по адресу из *value_storage*.

Результатом выполнения теста являлись данные результата свертки и отчет о работе, выводящего программой из фрагмента кода свертки, записанные в запущенное на сервере хранилище значений. Результат свидетельствует о корректной работе исполнителя при взаимодействии с остальными компонентами системы.

3.5 Итоги главы

В данной главе описана реализация компонента “Исполнитель” системы активных знаний LuNA, которые позволяет исполнять фрагменты кода с помощью “машин”, запущенных на узлах. Поясняются детали реализации частей архитектуры компонента. Также приводятся результаты тестирования, подтверждающие работоспособность реализованного решения при работе на нескольких узлах и в рамках всей системы активных знаний.

ЗАКЛЮЧЕНИЕ

В ходе работы были успешно выполнены все поставленные задачи и достигнута ее цель. По итогам работы для системы активных системы разработана модель адаптивного динамического принятия решений. Для технической реализации модели в системе LuNA представлен компонент “Исполнитель”, позволяющий исполнять фрагменты кода. Также проведено тестирования реализованного компонента.

На защиту выносятся следующие положения:

- Модель адаптивного динамического принятия решений;
- Архитектура компонента “Исполнитель”;
- Реализация компонента “Исполнитель”.

Дальнейшим направлением развития данной темы является разработка алгоритмов поиска эффективных конфигураций на карте и расширение компонента “Исполнитель” новыми runtime-системами.

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

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

_____ _____

ФИО студента *Подпись студента*

« ___ »_____20 __ г.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Малышкин В.Э., Перепелкин В.А. Построение баз активных знаний для автоматического конструирования решений прикладных задач на основе системы LuNA // Параллельные вычислительные технологии (ПаВТ’2024): Материалы XVIII Всероссийской научной конференции с международным участием. Короткие статьи и описания плакатов, Челябинск, 02–04 апреля 2024 года. — Челябинск: Издательский центр ЮУрГУ, 2024. — С. 126–136

2. Malyshkin V. Active Knowledge, LuNA and Literacy for Oncoming Centuries // Bodei C., Ferrari G., Priami C. (eds) Programming Languages with Applications to Biology and Security. Lecture Notes in Computer Science, vol. 9465. Springer, Cham, 2015. DOI: 10.1007/978–3–319–25527–9_19

3. Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters // OSDI’04: Sixth Symposium on Operating System Design and Implementation. — 2004. — P. 137–150

4. White T. Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale. — 4th edition. — O’Reilly Media, 2015. — 756 p. — ISBN–13 978–1491901632

5. Документация YTsaurus [Электронный ресурс]. — Режим доступа: https://ytsaurus.tech/docs/ru/ (дата обращения: 27.05.2025)

6. Ferreira R.A. et al. 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. — P. 159–166. — DOI 10.1109/CAHPC.2005.12

7. Acun B., et al. Parallel programming with migratable objects: Charm++ in practice // SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. — IEEE, 2014

8. Абрамов С.М., Московский А., Роганов В.А. 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

9. Bosilca G., Bouteiller A., Danalis A. et al. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability // Computing in Science and Engineering. — 2013. — Vol. 15. — P. 36–45

10. Aiken A., Bauer M. Programming with Legion // 2022.

11. Slaughter E., et al. Regent: a high–productivity programming language for HPC with logical regions // Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. — 2015.

12. Jordan H. et al. The allscale runtime application model //2018 IEEE International Conference on Cluster Computing (CLUSTER). – IEEE, 2018. – С. 445–455.

13. Chandra R. Parallel programming in OpenMP. – Morgan kaufmann, 2001.

14. Клинов М.С. Подход к автоматизированному распараллеливанию в системе САПФОР // Научный сервис в сети Интернет: все грани параллелизма: Труды Международной суперкомпьютерной конференции, Новороссийск, 23–28 сентября 2013 года. — Новороссийск: Издательство Московского государственного университета, 2013. — С. 291–295

15. Автоматизация распараллеливания программных комплексов / В. А. Бахтин, О. Ф. Жукова, Н. А. Катаев [и др.] // Научный сервис в сети Интернет : труды XVIII Всероссийской научной конференции, Новороссийск, 19–24 сентября 2016 года / ИПМ им. М.В. Келдыша РАН. – Новороссийск: Институт прикладной математики им. М.В. Келдыша РАН, 2016. – С. 76–85.

16. Sato M. et al. XcalableMP 2.0 and future directions //XcalableMP PGAS Programming Language: From Programming Model to Applications. – 2021. – С. 245–262.

17. Danalis A. et al. PTG: an abstraction for unhindered parallelism //2014 Fourth International Workshop on Domain–Specific Languages and High–Level Frameworks for High Performance Computing. – IEEE, 2014. – С. 21–30.

18. Bosilca G. et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA //2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. – IEEE, 2011. – С. 1432–1441.

19. Bondarev A. E., Galaktionov V. A., Kuvshinnikov A. E. Parallel solutions of parametric problems in gas dynamics using DVM/DVMH technology //Programming and Computer Software. – 2020. – Т. 46. – С. 176–182.

20. Вальковский В.А., Малышкин В.Э. Синтез параллельных программ и систем на вычислительных моделях. — Новосибирск: Наука, Сибирское отд–е, 1988. — 129 с.

21. Go programming language [Электронный ресурс] — Режим доступа: https://go.dev (дата обращения: 27.05.2025)

22. Что такое RESTful API? // Amazon Web Services. Режим доступа: https://aws.amazon.com/ru/what–is/restful–api (дата обращения: 27.05.2025)

23. Введение в JSON [Электронный ресурс]. — Режим доступа: https://www.json.org/json–ru.html (дата обращения: 27.05.2025)

24. rpc package – Go Packages [Электронный ресурс]. — Режим доступа: https://pkg.go.dev/net/rpc (дата обращения: 27.05.2025)

25. Gobs of data – The Go Programming Language [Электронный ресурс]. — Режим доступа: https://pkg.go.dev/encoding/gob (дата обращения: 27.05.2025)

26. QEMU [Электронный ресурс]. — Режим доступа: https://www.qemu.org/ (дата обращения: 27.05.2025)

ПРИЛОЖЕНИЕ А

КОМПОНЕНТ ИСПОЛНЕНИЯ ФРАГМЕНТОВ КОДА ДЛЯ СИСТЕМЫ LUNA

РУКОВОДСТВО ОПЕРАТОРА

Листов 8

Новосибирск 2025

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению компонента исполнения фрагментов кода для системе LuNA.

В данном программном документе, в разделе «Назначение программы» указаны сведения о назначении программы и информация, достаточная для понимания функций программы и ее эксплуатации.

В разделе «Условия выполнения программы» указаны требования, необходимые для выполнения программы.

В разделе «Выполнение программы» указана последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы.

Оформление программного документа «Руководство оператора» произведено по требованиям ЕСПД: 19.101-77, 19.105-78, ГОСТ 19.505-79.

1 Назначение и условия программы

1.1 Функциональное назначение программы

Программа предназначена для исполнения фрагментов кода, соответствующих операциям вычислительной модели системы активных знаний LuNA, на доступных вычислительных ресурсах.

1.2 Эксплуатационное назначение программы

Программа используется в составе системы LuNA и обеспечивает выполнение задач, связанных с исполнением вычислений, с возможностью динамического управления и учета характеристик целевой среды исполнения.

1.3 Состав функций

Программа обеспечивает выполнение следующих функций:

- Получение фрагментов кода и их параметров от ядра системы LuNA;
- Назначение задач на вычислительные узлы с учетом состояния машин;
- Передача входных данных в исполняемый модуль и получение выходных данных;
- Ведение мониторинга состояния вычислительных машин;
- Управление процессом исполнения через REST API и RPC интерфейсы.

2 Условия выполнения программы

2.1 Минимальный состав аппаратных средств

Программа предназначена для использования на персональном компьютере или вычислительном кластере. Минимальный перечень технических средств, обеспечивающих работу программы:

1. Оперативная память объемом не менее 4 ГБ;
2. Жесткий диск объемом не менее 64 ГБ;
3. Процессор с двумя или более ядрами, тактовой частотой не менее 2 ГГц;
4. Сетевое соединение Ethernet или Wi-Fi;
5. Клавиатура;
6. Монитор.

2.2 Минимальный состав программных средств

1. Операционная система семейства Linux;
2. Компилятор языка Go версии 1.20 или выше;
3. Командный интерпретатор bash;

2.3 Требование к персоналу

Конечный пользователь программы (оператор) должен обладать практическими навыками использования интерфейса командной строки Linux.

3 Выполнение программы

3.1 Загрузка и запуск программы

Запуск программы осуществляется через запуск управляющего компонента и запуск исполнительных компонентов на соответствующих узлах. Запуск управляющего компонента осуществляется выполнением исполняемого файла aks-executor-manager. Необходимо указать путь к конфигурационному файлу с помощью параметра командной строки *-config*. Аналогично, запуск исполнительного компонента осуществляется вызовом исполняемого файла aks-executor-machine.

Пользователь может указать следующие опциональные параметры:

- -config [путь к конфигурационному файлу] — указание пути к YAML-файлу с параметрами запуска.
- -help — вывод краткой справки о допустимых параметрах командной строки и завершение работы.
- Задание переменной окружения EXECUTOR_MANAGER_CONFIG_PATH или EXECUTOR_MACHINE_CONFIG_PATH в случае, если параметр -config не указан.

3.2 Выполнение программы

После запуска управляющий компонент ожидает подключения исполнительных компонентов. Каждый исполнительный компонент, при старте, передает информацию о своей конфигурации и регистрируется в управляющем компоненте. После регистрации компонент готов к исполнению запросов от других компонентов системы LuNA.

3.3 Завершение работы программы

Завершение работы программы осуществляется отправкой сигнала завершения с помощью комбинации клавиш Ctrl+C.

4 Сообщения оператору

В ходе работы компонента в качестве сообщения оператору может прийти только ошибка исполнения фрагмента кода. Других сообщений оператору не предусмотрено.

5 Лист регистрации изменений

Таблица А.1 – Лист регистрации изменений

Лист регистрации изменений									
№ Изм.	Номера листов (страниц)				Всего листов (страниц) в докум.	№ докум.	Входящий № сопровод. докум. и дата	Подп.	Дата
	Измен.	Замен.	Новых	Аннулир.					

ПРИЛОЖЕНИЕ Б

КОМПОНЕНТ ИСПОЛНЕНИЯ ФРАГМЕНТОВ КОДА СИСТЕМЫ LUNA

ОПИСАНИЕ ПРОГРАММЫ

Листов 11

Новосибирск 2025

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данном документе приведено описание компонента исполнения фрагментов кода системы LuNA.

Оформление программного документа «Описание программы» произведено по требованиям ГОСТ 19.402-78 «ЕСПД. Описание программы» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

1 Общие сведения

1.1 Обозначение и наименование программы

Полное наименование программы — компонент исполнения фрагментов кода “Исполнитель” для системы активных знаний LuNA.

1.2 Программное обеспечение, необходимое для функционирования программы

Программа функционирует в составе системы LuNA. Для работы необходимы: операционная система Linux, установленный компилятор языка Go (не ниже версии 1.20), интерпретатор bash и поддержка сетевых соединений.

1.3 Языки программирования

Исходным языком программы является Go. В качестве среды разработки может использоваться любой текстовый редактор, поддерживающий работу с этим языком, например Visual Studio Code.

2 Функциональное назначение

2.1 Назначение программы

Программа предназначена для исполнения фрагментов кода в рамках системы активных знаний LuNA.

2.2 Сведения о функциональных ограничениях на применение

В текущей реализации компонент не предусматривает исполнение фрагментов кода, не предназначенных для операционных систем семейства Unix.

3 Описание логической структуры

3.1 Алгоритм программы

Программа регистрирует вычислительные машины у управляющего компонента, получает задания на исполнение, загружает и исполняет фрагмент кода с переданными параметрами и возвращает результат выполнения в формате JSON.

3.2 Используемые методы

Для взаимодействия с системой применяются методы REST и RPC. Передача данных обеспечивается через структурированные сообщения в формате JSON.

3.3 Структура программы

Программа состоит из двух основных исполняемых компонентов: aks-executor-manager (управляющий компонент) и aks-executor-machine (исполнительный компонент). Каждый компонент включает набор внутренних пакетов, реализующих обработку конфигурации, сетевые интерфейсы, систему регистрации и исполнения.

3.4 Связи между составными частями программы

Взаимодействие между компонентами обеспечивается посредством вызовов API и межпроцессного RPC-взаимодействия. Машины передают информацию менеджеру, а менеджер распределяет задачи между машинами.

3.5 Связи программы с другими программами

Программа взаимодействует с внешним хранилищем фрагментов кода, хранилищем значений, а также с ядром системы LuNA, выступающим в роли инициатора вычислений.

4 Используемые технические средства

Программа работает на стандартных вычислительных узлах с ОС Linux, имеющих доступ к сети и достаточные ресурсы исполнения фрагментов кода.

5 Вызов и загрузка

Загрузка осуществляется вручную через вызов исполняемых файлов aks-executor-manager и aks-executor-machine с указанием пути к конфигурационному файлу. Имеется поддержка переменных окружения.

6 Входные данные

Входные данные включают: идентификаторы фрагментов кода, параметры входных переменных, а также настройки хранения и маршрутизации данных, передаваемые в формате JSON через API.

7 Выходные данные

Выходные данные включают: идентификаторы и значения переменных, полученные в результате исполнения фрагмента, а также сведения о статусе выполнения. Данные возвращаются в формате JSON.

8 Лист регистрации изменений

Таблица Б.1 – Лист регистрации изменении описания программы

Лист регистрации изменений									
№ Изм.	Номера листов (страниц)				Всего листов (страниц) в докум.	№ докум.	Входящий № сопровод. докум. и дата	Подп.	Дата
	Измен.	Замен.	Нов.	Аннулир.					