

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 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Пирожкова Андрея Константиновича

Тема работы:

**РАЗРАБОТКА ПОДСИСТЕМЫ ПРЯМОГО УПРАВЛЕНИЯ ИСПОЛНЕНИЕМ
ФРАГМЕНТИРОВАННЫХ ПРОГРАММ В СИСТЕМЕ LUNA**

«К защите допущен»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э. /.....
(ФИО) / (подпись)
«31» мая 2023 г.

Руководитель ВКР
к.т.н., доцент
доцент каф. ПВ ФИТ НГУ
Маркова В.П. /.....
(ФИО) / (подпись)
«20» мая 2023 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Перепёлкин В. А. /.....
(ФИО) / (подпись)
«20» мая 2023 г.

Новосибирск, 2023

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 09.03.01 Информатика и вычислительная техника

Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В.Э.

(фамилия, И., О.)

.....

(подпись)

«07» ноября 2022г.

ЗАДАНИЕ

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Пирожкову Андрею Константиновичу, группы 19201

(фамилия, имя, отчество, номер группы)

Тема: Разработка подсистемы прямого управления исполнением
фрагментированных программ в системе LuNA

(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 07.11.2022 № 0330

Срок сдачи студентом готовой работы 20 мая 2023 г.

Исходные данные (или цель работы):

Разработать подсистему автоматического конструирования традиционных императивных параллельных LuNA-программ без использования исполнительной системы.

Структурные части работы:

Введение, обзор предметной области, постановка задачи, описание разработанного решения, реализация программы, тестирование, заключение.

Руководитель ВКР
доцент каф. ПВ ФИТ НГУ,

к. т. н., доцент

Маркова В.П. /.....

(ФИО) / (подпись)

«07» ноября 2022 г.

Задание принял к исполнению

Пирожков А.К. /.....

(ФИО студента) / (подпись)

«07» ноября 2022 г.

Соруководитель ВКР

ст. преп. каф. ПВ ФИТ НГУ,

Перепёлкин В.А. /.....

(ФИО) / (подпись)

«07» ноября 2022 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
Глава 1. Обзор предметной области.....	8
1.1 Категории языков программирования	8
1.2 Средства получения параллельной программы	8
1.3 Языки и системы автоматического конструирования параллельных программ	10
1.4 Трансляторы кода и приложения для трансляций.....	11
1.5 Итоги обзора	13
Глава 2. Разработка автоматического средства конструирования параллельных фрагментированных программ	14
2.1 Системе LuNA	14
2.2 Термины и вычислительный процесс в системе LuNA	14
2.3 Автоматическое конструирование параллельных программ генерацией кода	15
2.4 Постановка задачи.....	16
2.5 Входной язык описания фрагментированной программы.....	16
2.5.1 Термины и определения для структур входного языка	17
2.5.2 Описание структур входного языка	18
2.5.3 Анализ структуры описания фрагментов вычислений	19
2.6 Структура данных для конструирования параллельных программы	20
2.6.1 Термины и определения для структуры данных генератора параллельных программ	20
2.6.2 Структура данных конструкций	21
2.6.3 Виды сниплетов.....	23
2.6.4 Типы сниплетов.....	25
2.7 Алгоритм конструирования программы.....	26
2.7.1 Понятия необходимые для описания алгоритма конструирования программы.....	26
2.7.2 Построение программы без коммуникаций	27
2.7.3 Добавление коммуникаций в построенной программе.....	29

2.7.4	Генерация кода по построенной программе.....	31
2.8	Эффективность сконструированной программы.....	32
2.8.1	Оценка качества конструируемой программы.....	33
2.8.2	Использование градиентного спуска для оценки качества	34
2.9	Анализ полученных результатов	35
Глава 3.	Реализация.....	36
3.1	Входные данные генератора параллельных фрагментированных программ	36
3.2.	Реализация генератора параллельных фрагментированных программ....	38
3.2.1	Реализация структуры данных в генераторе	39
3.2.2	Реализация алгоритма конструирования	42
3.2.3	Генерация кода	43
3.2.4	Ограничения генератора параллельных программ.....	45
3.2.5	Запуск сгенерированной параллельной программы.....	46
3.3	Анализ реализации генератора параллельных программ	47
Глава 4.	Тестирование.....	48
4.1	Тестирование генератора на простых задачах	48
4.2	Тестирование задачи вычисления удвоенной площади треугольника	49
4.4.1	Подготовка LuNA-программы для тестирования.....	55
4.4.2	Создание входного представления.....	56
4.4.3	Отличия LuNA-программы и сгенерированной программы	56
4.4.4	Подготовка к запуску.....	57
4.4.5	Результаты тестирования	58
4.5	Итоги тестирования	58
ЗАКЛЮЧЕНИЕ	60
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ.....		62
ПРИЛОЖЕНИЕ А		65
ПРИЛОЖЕНИЕ Б.....		69
ПРИЛОЖЕНИЕ В		72
ПРИЛОЖЕНИЕ Г.....		84

ВВЕДЕНИЕ

Параллельное программирование позволяет решать задачи численного моделирования быстрее, чем последовательное программирование. Но чтобы разрабатывать параллельные программы, недостаточно навыка обычного программирования. Нужно также обладать навыками и опытом в области параллельного программирования. Поэтому существуют системы автоматического конструирования параллельных программ, позволяющие разрабатывать параллельные программы быстрее и без дополнительных навыков в области параллельного программирования.

Задача автоматического конструирования параллельных программ является алгоритмически труднорешаемой задачей. Системам автоматического конструирования сложно автоматически выбрать из большого количества вариантов параллельной программы достаточно эффективный вариант. И при работе такой сконструированной программы может получиться так, что программа выполняется медленнее и расходует память больше, чем при ручном программировании. Такая неэффективная реализация влечёт за собой затраты на электроснабжение и покупку комплектующих, таких как оперативная память, но зато эффективно расходуется время на разработку и требуется меньший уровень квалификации у разработчиков. Но решение задачи всё равно ещё может оставаться дорогим и ресурсозатратным.

Обычно автоматическое конструирование параллельных программ происходит с помощью систем автоматического конструирования и параллельных языков программирования, чтобы удобнее было разрабатывать параллельные программы. Но пользователь в таких системах должен иметь базовые навыки и понимание в области параллельных вычислений. Аналогичная ситуация и с системами автоматического распараллеливания, которые способны из последовательного кода сделать параллельную программу. Чаще всего такие системы работают не со всеми языками программирования и распараллеливают код на уровне циклов в разрабатываемой программе.

Автоматическое конструирование параллельных программ осуществляется двумя основными способами. Первый способ заключается в использовании исполнительной системы при конструировании параллельных программ. Второй способ заключается в генерировании параллельного кода без использования исполнительной системы. Такой способ конструирования параллельных программ лучше подходит для таких задач, у которых нет динамических свойств и которым не нужно адаптироваться во время исполнения параллельной программы. Потому что в ходе исполнения параллельной программы не будут тратиться ресурсы, которые использовались в исполнительной системе. Но такой способ конструирования не может быть реализован эффективно универсальным способом для всех систем автоматического конструирования. Поэтому важно развивать его для разных языков программирования.

Автоматическое конструирование параллельных программ может осуществлять система фрагментированного программирования LuNA [1]. Эта система активно развивается и использует перспективные подходы в автоматизации конструирования параллельных программ. Но в ней не проработана возможность генерации параллельного кода для статических программ без исполнительной системы. Поэтому в этой системе можно реализовать такой способ конструирования программ.

Задача конструирования параллельной программы для высокоуровневого языка программирования сводится к определению прямого управления программы. Эффективность получаемых параллельных программ зависит от того, как строили прямое управление. Контролирование прямого управления означает контролирование эффективности получаемых программ. В рамках этой работы осуществление и контроль прямого управления называется подсистемой прямого управления.

Для того чтобы генерировать параллельные программы, нужно определить средства прямого управления, которые будут являться базовыми блоками управления — элементами конструкций. Каждый элемент конструктора является

некоторой частью фрагментированной программы, отображенный на конкретный ресурс с конкретным управлением. Таким образом, **цель работы** состоит в том, чтобы разработать подсистему автоматического конструирования традиционных императивных параллельных LuNA-программ без использования исполнительной системы. Для достижения цели были сформулированы следующие **задачи**:

- Разработка языковых средств описания параллельной программы.
- Разработка структур данных и алгоритмов, обеспечивающих конструирование параллельных программ.
- Экспериментальное исследование эффективности конструируемых параллельных программ.

Научная новизна заключается в разработке алгоритмов конструирования императивных параллельных программ без использования исполнительной системы для фрагментированных параллельных программ.

Работа состоит из введения, 3 глав и заключения. Введение описывает общую проблему, актуальность, цель и задачи работы, а также научную новизну. В первой главе проводится анализ существующих решений и их возможность применимости в системе LuNA. Во второй главе формулируется постановка задачи, а также описание разработанного решения. В третьей главе приводятся практические результаты разработки и тестирование решения. В заключении подводятся итоги и выводы по проделанной работе.

Глава 1. Обзор предметной области

1.1 Категории языков программирования

Рассмотрим, какая категория языков программирования лучше подходит для генерации параллельного кода. Основными категориями считаются компилируемые и интерпретируемые языки программирования.

Компилируемые языки — это языки программирования, для которых исходный код программы транслируется в машинный код компьютера при помощи компилятора. Интерпретируемые языки — это языки программирования, для которых исходный код программы интерпретируется на лету во время выполнения программы интерпретатором. Таким образом, каждая строка кода в интерпретируемом языке должна быть проанализирована и выполнена интерпретатором во время выполнения программы. Это добавляет накладные расходы на процесс выполнения программы, что может привести к медленной работе по сравнению с компилируемыми языками.

Подход, при котором генератор параллельных программ создаёт параллельную программу приложения в виде интерпретируемого языка, является неэффективным. В данном случае лучше предпочесть языки, которые компилируют приложения в машинный код, чтобы избежать потери времени во время работы программы. Компиляция программы перед запуском может занять некоторое время, но это позволит избежать накладных расходов на интерпретацию кода в процессе выполнения и ускорит работу приложения в целом.

1.2 Средства получения параллельной программы

Рассмотрим, какие компиляторы позволяют получать параллельные программы, какие средства для них нужны и могут ли компиляторы из последовательного кода компилировать параллельную программу. Так как для параллельного программирования требуются дополнительные навыки в этой области, то для пользователей, умеющих разрабатывать последовательные программы, можно было бы использовать способы, которые способны превратить из последовательной программы — параллельную программу.

Показательным примером компилируемых языков является язык C и C++. Компилятор GCC [2] при компиляции кода на языке C проходит много этапов, перед тем как мы получим сгенерированный исполняемый файл, который будет оптимизирован под нашу архитектуру компьютера. Обычно перед компиляторами языков не стоит задача сгенерировать параллельную программу. GCC может сгенерировать параллельный машинный код, если сам программист целенаправленно писал код программы с использованием инструментов для разработки параллельной программы. Полностью автоматически GCC, G++ и прочие компиляторы на C и C++ не способны распараллеливать программы.

Чтобы генерировать параллельные программы, можно использовать директивы препроцессора и библиотечные функции с помощью OpenMP. Эта технология позволяет распараллеливать выполнение циклов, блоков кода и других участков программы. Но OpenMP [3] предоставляет довольно узкий перечень инструментов для работы с процессами и OpenMP не имеет инструментов управления в распределенной памяти. Также необходимо самим вручную расставлять все директивы и использовать библиотечные функции, чтобы программа стала параллельной. А вот технология MPI [4] обладает достаточным набором для разработки параллельных программ. Это интерфейс для передачи и обмена сообщениями между процессами, и он способен работать в распределенной памяти. Но все коммуникации расставляет сам программист в своём коде для получения параллельной программы.

Всё вышперечисленное в этом пункте не является автоматической системой генерации параллельного кода. Компиляторы только компилирует то, что написал сам программист, в том числе и примененные средства программистом для разработки параллельных программ. OpenMP работает в общей памяти, а нужно в распределенной, как в MPI. Это только инструменты для ручной разработки параллельных программ. Но эти технологии можно использовать для разработки собственного генератора параллельных программ.

1.3 Языки и системы автоматического конструирования параллельных программ

Рассмотрим средства автоматического конструирования параллельных программ и выявим, в чём является основной недостаток таких систем.

Существует множество специализированных языков программирования, которые были разработаны специально для параллельных вычислений. Такими языками являются Chapel [5], X10 [6], Titanium [7], Fortress [8]. Каждый язык имеет свои особенности и уникальные возможности, но общей целью является облегчение написания эффективного параллельного кода для распределенных систем. Также эти языки имеют встроенные библиотеки для работы с распределенными вычислениями. Хотя эти языки являются на уровень выше, чем MPI и OpenMP, предоставляя большой функционал и даже свой синтаксис языка, у них всё равно нет опции, которая бы в один клик или одну команду при компиляции сделала бы из последовательного кода параллельный эффективный код.

Еще все перечисленные языки (Chapel, X10, Titanium, Fortress) используют свою исполнительную систему. Она может оказывать негативное влияние на производительность программы и использование ресурсов системы во время работы программы, так как это добавляет накладные расходы на управление процессами и потоками исполнения.

Обычно в исполнительных системах есть специальные балансировщики нагрузки, например в Charm++ [9,10]. Это мощный фреймворк для разработки высокопроизводительных параллельных приложений. Балансировка нагрузок полезна, когда работа некоторых процессов или потоков имеют большую вычислительную нагрузку, чем другие. В таких случаях балансировка нагрузок может помочь ускорить выполнение работы путем распределения более тяжелых задач на свободные процессы. Но, если изначально программа не требует балансировки нагрузок или программа уже спроектирована таким образом, что балансировка нагрузок не нужна и что равномерная вычислительная нагрузка на процессах уже обеспечена, значит надо рассматривать варианты без

использования балансировки нагрузок и исполнительной системы. А обеспечивать такую возможность (без использования исполнительной системы) должен будет генератор параллельных программ на уровне генерации параллельного кода.

Таким образом, языки и системы автоматического конструирования параллельных программ способны создать параллельную программу, но в то же время могут потратить много ресурсов при работе из-за исполнительной системы.

1.4 Трансляторы кода и приложения для трансляций

Ранее рассматривались способы автоматического конструирования программ, где выяснилось, что в ходе исполнения программы тратятся ресурсы на исполнительную систему. Либо способ является не автоматический, а разрабатывается программистом с помощью таких инструментов, как MPI. Теперь рассмотрим такой подход, при котором автоматически преобразовывается код из последовательного в параллельный. Для таких целей существует отдельная категория программного обеспечения – трансляторы. Проанализируем их с точки зрения пригодности в работе в распределенной памяти и применимости на конкретном языке C++.

Трансляторы из последовательного кода в параллельный часто называют компиляторы категории source-to-source. Они чаще всего имеют в себе возможность использования технологии OpenMP, добавляя при трансляции в код программы строчки с директивами препроцессора. Это умеют делать такие трансляторы как ParaWise [11], Polaris [12], Cetus [13,14], Par4All [15], SAPO [16]. ParaWise, Polaris и SAPO разработаны для языка Fortran, Cetus разработан для C, C++ и Java, а Par4All – для C и Fortran.

Технология OpenMP работает с общей памятью. Для разработки генератора параллельных программ необходимо иметь инструмент, который позволяет работать с распределенной памятью, поэтому такие трансляторы как ParaWise, Polaris, SAPO и Cetus работающие только с помощью технологией OpenMP не подходят для разработки генератора. Все они предназначены для

работы в общей памяти. Более того из последних перечисленных трансляторов только один поддерживает C, а все остальные работают только с Fortran.

Самый продвинутый транслятор из списка является Par4All, поскольку он может распараллелить код не только с помощью технологий OpenMP, но и с технологией для графических ускорителей CUDA и OpenCL, а также возможна компоновка с MPI или с другими библиотеками. Работает с кодом программы написанный на Fortran или на C. Для разрабатываемого мною генератора необходима поддержка языка C++, поскольку генератор оперирует с классом DF, который разработан на C++.

Некоторые трансляторы переросли в полноценные приложения с графическим дизайном. Например, SAPO был интегрирован в CAPTools [17], где пользователь мог загрузить свой последовательный код и выбирать опции параллелизации кода. Приложение показывало информацию о том, какие части кода были распараллелены. Также в CAPTools появилась возможность распараллеливания программы с помощью MPI и PVM [18]. Также стоит отметить, что и ParaWise потом также был сделан в виде приложения с графическим дизайном под названием ParaWise Expert Assistant. Оно имело более углубленные проверки информационных зависимостей во многих процедурах, что позволяло еще более эффективно использовать технологии OpenMP. ParaWise Expert Assistant не использовал MPI при распараллеливании программ. Более того, CAPTools и ParaWise Expert Assistant работали только с языком программирования Fortran.

Всё, что выше перечислено, это иностранные наработки в области распараллеливания программ. В России есть проект под названием «Открытая распараллеливающая система» (ОРС [19]). Проект занимается разработкой в нескольких направлениях. Но меня интересует конкретно автоматический распараллеливатель программ. Он выполнен в полноценном приложении для компьютеров. Имеет в себе возможность работать с языками C и Fortran. Умеет строить графы зависимостей, умеет распараллеливать программу на C как с помощью OpenMP, так и с помощью MPI. В магистерской работе студента [20]

указано, что автоматическая генерация кода для циклов с помощью MPI работает как без информационных зависимостями внутри цикла, так и с ними. Таким образом, MPI чаще всего применяется для выполнения вычисления массивов в циклах на разных процессах. Поддержки C++ для OPC нет.

Рассмотренные трансляторы кода не имеют поддержки для языка C++. Также трансляторы чаще всего создают параллельный код для циклов с массивами. Возможность выполнения разных частей кода параллельно в рассмотренных выше трансляторах не обнаружилась.

1.5 Итоги обзора

В этом обзоре были рассмотрены способы получения параллельных программ. Для таких целей обычно используются специальные языки программирования, предназначенные для параллельных программ, на которых пишут программисты параллельный код. Но такие языки и системы имеют в себе исполнительную систему, которая дополнительно расходует ресурсы и увеличивает время работы программы. Для автоматической генерации параллельного кода используются специальные трансляторы или специальные приложения, которым на вход подается последовательный код, а на выходе получается параллельная программа. Но такие трансляторы сильно ограничены в работе с языками программирования, а также способами для распараллеливания последовательных программ.

Глава 2. Разработка автоматического средства конструирования параллельных фрагментированных программ

В данном разделе дано описание системе LuNA, объяснены необходимые термины, а также дана постановка задачи. Далее описаны основные теоретические результаты о средстве конструирования параллельных фрагментированных программ:

- Входной язык описания.
- Структура данных.
- Алгоритм конструирования.
- Оценка качества конструируемых программ.

2.1 Системе LuNA

LuNA (Language for Numerical Algorithms) — это система, которая автоматически создает параллельные программы для мультимикомпьютеров и поддерживает технологию фрагментированного программирования [21]. Язык программирования системы LuNA основан на единственном присваивании и затем транслируется в C++ [22].

2.2 Термины и вычислительный процесс в системе LuNA

Фрагментированная программа — это программа, описывающая набор фрагментов вычислений и фрагментов данных. Исполнительная система распределяет фрагменты данных и вычислений по процессам мультимикомпьютера, а также выбирает порядок их выполнения.

Фрагмент вычислений — это объект времени исполнения, который описан в фрагментированной программе, который содержит описание входных и выходных фрагментов данных, а также фрагмент кода, применяемый фрагментами вычислений.

Фрагмент кода — это процедура, которая используется в фрагментах вычислений как часть программного кода, который выполняет фрагмент вычислений. Фрагмент кода может быть выполнен независимо от других частей программы.

Фрагмент данных — это блок данных, который используется в фрагментированной программе как входной или выходной параметр для фрагмента вычислений. Фрагмент данных в фрагментированной программе может быть представлен в виде целочисленной или вещественной переменной, а также в виде массива или другого вида переменных.

Фрагментированная программа для системы LuNA пишется пользователем. При запуске фрагментированной программы запускается исполнительная система, которая управляет фрагментами вычислений и фрагментами данных. Заранее системе не известно количество фрагментов данных и фрагментов вычисления, так как фрагменты данных и фрагменты вычисления описаны в параметрическом виде. Исполнительная система конструирует программу в ходе её исполнения.

2.3 Автоматическое конструирование параллельных программ генерацией кода

Для упрощения реализации автоматического конструирования параллельных программ введём ограничение, что количество фрагментов вычислений и фрагментов данных должно быть заранее известно. Это ограничение является принципиальным, но не является критическим, так как многие практические задачи всё равно можно исполнять в таком ограничении. Снятие этого ограничения возможно в будущем, но оно выходит за рамки этой работы.

При генерации кода, которую будет осуществлять разрабатываемый генератор параллельных программ, число всех фрагментов данных и вычислений в программе будет определено заранее. Поэтому разрабатываемый генератор будет работать с операторами применения фрагментов вычислений и фрагментов данных. Отсюда следующие определения:

Оператор применения фрагмента вычисления — это часть фрагментированной программы, которая применяет фрагмент кода и в котором используются фрагменты данных. Фрагменты данных в операторе применения фрагмента вычисления также бывают как входные, так и выходные.

Далее понятие «фрагмент вычисления» в тексте будет пониматься как «применение оператора фрагмента вычисления».

2.4 Постановка задачи

Для достижения поставленной цели, необходимо разработать входное представление, которое опишет фрагментированную программу, состоящую из множества фрагментов данных и фрагментов вычислений, которые пойдут на вход генератору параллельных программ. Для упрощения постановки задачи в генераторе параллельных фрагментированных программ примем ограничение, которое было описано выше в п. 2.3. Тогда в генераторе будут использоваться только атомарные операторы. Для описания входного представления будут разработаны языковые средства описания фрагментированной программы, которые будут описывать множество фрагментов данных и вычислений. Для реализации прямого управления, нужно разработать конструкции, из которых будет конструироваться параллельная программа. Такие конструкции будут реализовывать часть фрагментированной программы, отображенного на конкретный ресурс и осуществлять конкретное управление. А алгоритм конструирования должен будет построить из конструкций параллельную программу в порядке информационных зависимостей, а затем сгенерировать параллельный код, который будет выполняться параллельно на нескольких процессах. Также необходимо рассмотреть вопрос эффективности генерируемых параллельных программ.

2.5 Входной язык описания фрагментированной программы

Для разработки средства автоматического конструирования параллельных программ нужно дать пользователю возможность описать, какую параллельную программу он хочет получить, а генератору, соответственно, понять, какую программу нужно сконструировать. Для этих целей создаётся входной язык описания фрагментированной программы. И в этом разделе будет рассмотрена его структура.

2.5.1 Термины и определения для структур входного языка

Фрагменты вычислений в генераторе параллельных программ могут использоваться как фрагменты данных, так и обычные переменные двух типов — целочисленные (int) и вещественные (double) из языка программирования C++. Для упрощения дальнейшего повествования понятие «фрагменты данных» будет включать в себя ещё и переменные, описанные выше, по умолчанию.

Фрагменты данных могут быть входными или выходными на уровне фрагментов вычислений. **Выходными** фрагментами данных называют те фрагменты, которые в ходе исполнения фрагмента вычислений инициализируются. **Входные** — это фрагменты данных, которые используются в фрагменте вычислений уже инициализированными. По-другому входные фрагменты данных называются **потребляемыми**, а выходные — **вырабатываемыми**.

Фрагменты вычислений в генераторе параллельных программ имеют 3 категории:

- инициализирующие;
- промежуточные;
- финальные.

В **финальных** фрагментах вычислений могут использоваться только входные фрагменты данных. В **инициализирующих** фрагментах данных могут использоваться только выходные фрагменты данных. В **промежуточных** фрагментах вычислений используются обязательно как выходные, так и выходные фрагменты данных. Причем инициализирующих и финальных фрагментов вычислений может максимум по одному в фрагментированной программе, в то время как промежуточных — заранее определенное конечное количество.

Также выделяется понятие входных и выходных фрагментов данных на уровне фрагментированной программы. **Входными** фрагментами данных называются такие фрагменты данных, которые используются в инициализирующем фрагменте вычислений или которые будут

инициализированы другим способом. **Выходные** — это фрагменты данных, которые используются в финальном фрагменте вычислений или которые будут использованы самым последними в фрагментированной программе.

2.5.2 Описание структур входного языка

Структура входного языка состоит из 5 частей:

- Перечисление файлов с описанием фрагментов кода.
- Входные и выходные фрагменты данных, количество используемых процессов.
- Перечисление фрагментов вычислений с подробным их описанием.
- Перечисление всех фрагментов данных с указанием типов.
- Указание переменных по умолчанию (define).

Фрагменты вычислений используют фрагмент кода для работы фрагментированной программы. Фрагменты кода представлены в специальном формате для удобства их использования в генераторе параллельных программ при конструировании. Реализация формата для описания фрагментов кода представлена в приложении Б. Обычно файл описания фрагментов кода один, но их может быть несколько.

В структуре необходимо указать входные и выходные фрагменты данных для фрагментированной программы, чтобы алгоритм генератора параллельных программ знал, с каких переменных начинать строить фрагментированную программу, а какими фрагментами данных завершать построение программы. Количество процессов, на которых будет выполняться сгенерированная программа, определяется заранее и указывается в структуре.

Структура с перечислением фрагментов вычислений является перечислением структур с описанием каждого фрагмента вычислений. Реализация формата для описания фрагментов кода представлена в приложении А. Состоит такая вложенная структура из следующих частей:

- Используемый фрагмент кода. Каждый фрагмент вычислений использует определённый фрагмент кода и в структуре он указывается.

- Тип фрагмента вычислений. Фрагменты вычислений имеют всего 3 типа, описанные в п. 2.5.1. В этой структуре указывается какой именно.
- Рекомендованный номер процесса. При генерировании параллельной программы в структуре можно указать на каком конкретно номере процесса должен будет исполнен конкретный фрагмент вычислений. Если его не указывать, генератор сам определит на свое усмотрение какой номер процесса использовать.
- Перечисление входных фрагментов данных. Перечисляются все потребляемые фрагменты данных в данном фрагменте вычислений.
- Перечисление выходных фрагментов данных. Перечисляются все вырабатываемые фрагменты данных в данном фрагменте вычислений.
- Связка с аргументами фрагмента вычислений и фрагментов кода. В структуре настраивается связь между фрагментами данных используемых в фрагменте вычислений и фрагментами данных, которые указаны в сигнатуре функции в фрагментах кода. Это делается для каждого фрагмента данных из фрагментов кода как перечисление.

После структуры с перечислением всех используемых фрагментов вычислений идет структура с перечислением всех фрагментов данных с указанием их типов.

Последней структурой является перечисление переменных со значением по умолчанию. Именно переменных, а не фрагментов данных. Это потом превратится в объявление директив `define`.

2.5.3 Анализ структуры описания фрагментов вычислений

Структура описания фрагментов вычислений предоставляет достаточную информацию для того, чтобы из неё конструировать фрагментированную программу. Но функциональность структуры не позволяет описывать условия, циклы «for» и «while» как в LuNA. Это связано с генератором параллельных программ, которому ещё пока что не добавлялись такие возможности. Циклы можно заменить обычной раскруткой фрагментов вычислений, а условия можно попытаться внести во внутрь фрагментов кода.

Входной язык имеет сходства с входным представлением LuNA и является более упрощенной его версией.

Входной язык описания фрагментированной параллельной программы позволяет осуществить многовариантность программ. Это может быть достигнуто двумя способами:

- Выбор между разными фрагментами вычислений.
- Распределение фрагментов вычислений по процессам может быть любым.

Последний способ многовариантности достигается назначением номера процесса в алгоритме конструирования параллельной программы. А первый способ многовариантности достигается пользователем в ходе создания входного описания. Если у пользователя есть несколько способов инициализировать один и тот же вырабатываемый фрагмент данных, то алгоритмом конструирования параллельной программы выберет один из фрагментов вычислений, где есть нужный вырабатываемый фрагмент данных.

2.6 Структура данных для конструирования параллельных программ

После того как у генератора параллельных программ появилась описание необходимой программы для генерации в виде входного языка, генератору нужно теперь создавать эту программу из «строительных материалов». В качестве таких материалов будет создана специальная структура данных, которая позволит внутри генератора конструировать параллельную программу. Об этой структуре данных и пойдет речь в этом разделе.

2.6.1 Термины и определения для структуры данных генератора параллельных программ

Конструкция — это структура, которая содержит в себе необходимые свойства и параметры, а также код для генерации программы, если эта конструкция после генерации параллельной фрагментированной программы выражается в коде.

В этих пунктах понятия: генератор, генератор программ, генератор параллельных программ подразумевают понятие — генератор параллельных фрагментированных программ.

Понятия «строится» и «конструируется» считаю синонимами, также как и «построится» и «конструируется». Аналогично со словом «генерируется» и «конструируется» — это тоже синонимы.

2.6.2 Структура данных конструкций

Для генератора параллельных программ используются следующие объекты:

- Программная спецификация.
- Ячейка памяти.
- Порт.
- Слиплет.
- Частично-определённая программа.

Программная спецификация — это конструкция с информацией о входном представлении фрагментированной программы. Её структура была описана в предыдущих пунктах. Входное представление позволяет алгоритму создавать другие объекты, заполняя данные других структур из программной спецификаций.

Ячейка памяти — это конструкция, содержащая фрагмент данных. В ячейке памяти хранятся данные о названии фрагмента данных и о его типе (если это переменная, то это либо целочисленный тип, либо вещественный). В процессе работы алгоритма построения определяются данные о том, на каком порту эта ячейка памяти инициализируется (вырабатывается) и на каких портах она будет использоваться (потребляться).

Порт — это конструкция, которая описывает используемый фрагмент данных. В сигнатуре функции в языке программирования C++ перечисляются переменные. Так вот каждая такая переменная в генераторе параллельных программ представляется в виде порта. Порт существует в каждом фрагменте вычислений для каждого фрагмента данных. В порте указывается информация о

названии фрагмента данных (как он будет объявлен в сгенерированной программе), тип (вырабатываемый или потребляемый фрагмент данных), номер процесса на котором используется этот порт (он такой же, как и номер процесса, на котором будет исполняться фрагмент вычислений) и информацию в каком именно сниплете используется этот порт. В процессе работы алгоритма добавляется информация, к какой ячейки памяти этот порт относится. В процессе генерации кода порт используется для вставки названия фрагмента данных в сниплетах.

Сниплет — это конструкция, предназначенная для выполнения операций, чаще всего фрагментов вычислений, которая в ходе генерации параллельной программы превратится в нужный кусок кода на языке C++. Внутри себя сниплет содержит перечисление портов. Чаще всего сниплет представляет собой описание фрагмента вычислений и содержит внутри себя фрагмент кода, который должен выполняться. Также в сниплете указывается информация о частично-определённой программе, номер процесса, на котором предполагается выполнения данного сниплета, тип сниплета (он такой же, как и тип фрагментов вычислений).

Частично-определённая программа — это конструкция, которая представляет собой перечисление сниплетов и ячеек памяти с построенными между ними связями. Причем во время работы алгоритма построения параллельной программы сниплеты и ячейки памяти постепенно добавляются, а связи между ними постепенно строятся в частично-определённой программе. Частично-определённая программа хранит в себе программную спецификацию, чтобы встраивать нужные описания в ячейки памяти, порты и сниплеты и инициализировать в них, а также на скольких предполагаемых процессах будет запущена сгенерированная параллельная программа. При генерации кода, частично-определённая программа генерирует весь необходимый код, который отвечает за инициализацию необходимых компонентов для генерируемой программы.

2.6.3 Виды сниплетов

Сниплеты в генераторе параллельных программ имеют несколько видов:

- Сниплет подстановки операции.
- Коммуникационный сниплет.
- Сниплет вывода в консоль.
- Сниплет ввода из переданных аргументов в программе.

Чаще всего используется два первых видов сниплетов. Их достаточно для того, чтобы описать фрагментированную программу. Во всех этих перечисленных сниплетах содержится номер процесса, на котором будет выполнен сгенерированный код. Также все сниплеты содержат внутри себя нескольких конструкций — портов. Это обеспечивается за счёт того, что сниплетам передается необходимая информация из конструкции с программной спецификацией. Также каждый порт сохраняет идентификатор сниплета, в котором он используется.

Сниплет подстановки операции

Этот сниплет используется для того, чтобы исполнялись фрагменты вычислений. Все фрагменты вычислений выполняют фрагмент кода. В программной спецификации для каждого фрагмента вычислений указан применяемый фрагмент кода. Сниплет подстановки операции осуществляет использование фрагмента кода с правильными переданными параметрами.

Количество портов в данном сниплете равно количеству фрагментов данных используемых в этом фрагменте вычислений. Фрагменты данных свой тип переносят в описание портов. Аналогично с названием фрагментов данных и номером процесса, на котором будет выполняться сгенерированный код.

В сгенерированном коде этот сниплет будет осуществлять вызов нужной операции (из фрагментов кода) с параметрами, описываемых в фрагменте вычислений и в программной спецификации.

Коммуникационный сниплет

Коммуникационный сниплет используется, когда нужно с одного процесса отправить фрагменты данных на другой процесс. Потребность в таком сниплете

возникает из-за того, что инициализируются фрагменты данных на конкретном процессе, а потребляется фрагмент данных на другом процессе. Поэтому коммуникационный снипплет отправляет фрагмент данных на нужный процесс, где этот фрагмент данных нужен для использования в других снипплетах.

Коммуникационный снипплет внутри себя хранит вместо одного номера процесса — два. Это номера процесса отправителя и получателя. Такая конструкция внутри себя имеет всегда 2 порта с одинаковым названием переменной. Фрагмент данных на первом порте имеет тип потребляемый, а на втором порте — вырабатываемый.

Генерируемый код для коммуникационного снипплета выбирается в зависимости от типа фрагмента данных (фрагмент данных, целочисленный или вещественная переменная). При генерации кода подставляется нужный выбор. Генерируемый код обеспечивает пересылку данных с одного процесса на другой.

Снипплет вывода в консоль

Снипплет вывода в консоль может использоваться как альтернатива снипплету подстановки операции. Такой снипплет может использоваться самым последним в генерируемой программе, и он предназначен для вывода значения в консоль. Подходит такой снипплет только для целочисленного типа (int) или вещественного типа (double). Такой снипплет содержит в себе только один порт. Генератор генерирует код, который выводит значение в консоль.

Снипплет ввода из переданных аргументов программы

Этот снипплет может использоваться как альтернатива снипплету подстановки операции. Такой снипплет может быть использован только в начале генерируемой программы в зависимости от количества запланированных переданных аргументов. Снипплет ввода предназначен для инициализации только переменных целочисленных (int) и вещественных (double) типов. На каждый такой снипплет используется только один порт. Порядок предполагаемых переданных аргументов должен совпадать с порядком, который записывается в описании фрагментированной программы в входных фрагментах данных.

Генератор генерирует код, который позволяет записать значение переменной из предполагаемых аргументов переданной сгенерированной программе.

2.6.4 Типы сниплетов

При рассмотрении описания фрагментов вычислений выделялось 3 типа: инициализирующий, промежуточный и финальный. Такая же характеристика присутствует у каждого сниплета, так как сниплет описывает фрагмент вычислений, у которого как раз есть такая категория.

Сниплет подстановки операции может принимать любой из типов сниплета. Но инициализирующий тип у этого вида сниплета может быть только один, либо вместо него может быть другой вид инициализирующего сниплета, причем в нескольких количествах. Похожая ситуация с финальным типом этого вида сниплета. Финальный вид сниплета в генерируемой программе всегда один. Он может быть либо сниплетом подстановки, либо другим сниплетом. Промежуточный типом сниплета может быть любое ограниченное конечное число в данном виде сниплета.

Коммуникационный сниплет всегда является промежуточным. Его количество зависит от количества необходимых коммуникаций между процессами.

Сниплет ввода из переданных аргументов программы может являться только инициализирующим сниплетом, так как предназначен только для присваивания значения переменных из аргументов программы, чтобы потом их использовали другие сниплеты. Их может быть несколько штук в зависимости от количества входных переменных, либо таковых не быть, так как инициализирующий сниплет уже выражен сниплетом подстановки операции.

Сниплет вывода в консоль является только финальным сниплетом. Он предназначен для завершения программы. Такой вид сниплета может быть только один, либо он заменяется сниплетом подстановки операции финальным типом.

2.7 Алгоритм конструирования программы

У генератора параллельных программ теперь есть специальные конструкции, которыми он может оперировать. Теперь ему необходимо научиться собирать из них параллельную программу, чтобы она была собрана в порядке информационных зависимостей, а между процессами были корректные коммуникации. Это обеспечится разработанным алгоритмом конструирования программы.

Генерируемая программа конструируется в несколько этапов:

1. Построение программы без коммуникаций.
2. Добавление коммуникаций в построенной программе.
3. Генерация кода для построенной программы.

Но перед выполнением этих этапов в начале генератора параллельных программ инициализируется конструкция с входным представлением — программная спецификация. А также создаётся частично-определённая программа, которой будет содержаться только что созданная программная спецификация. Также в частично-определённую программу добавляется финальный сниплет. После этого алгоритм приступает к первому этапу конструирования параллельной программы. Конструирование параллельной программы осуществляется от конца к началу.

2.7.1 Понятия необходимые для описания алгоритма конструирования программы

Бинд — это построение связи между текущим портом с определенной ячейкой памяти. Т.е. порт указывает (ссылается) на ячейку памяти, а ячейка памяти указывает (ссылается) на порт.

Сниплет считается завершенным, если у него каждый порт имеет бинд.

Порт, который содержит в себе потребляемую переменную, будем называть входным.

Порт, который содержит в себе вырабатываемую переменную, будем называть выходным.

Порт является свободным, если он не имеет бинда.

Ячейка памяти является проинициализированной (инициализированной), если у неё осуществлена связь с портом. Т.е. ячейка памяти указывает (ссылается) на определенный порт, где она вырабатывает (инициализирует) фрагмент данных. Также выражение «ячейка памяти инициализирована» означает, что ячейка уже ссылается на порт, который её инициализирует. Выражение «ячейка памяти стирает инициализирование» означает, что ячейка памяти теперь не ссылается на порт, где она инициализировалась.

Ячейка памяти является забинденной, если у неё осуществлена связь с портом. Т.е. ячейка памяти указывает (ссылается) на определенный порт, где она потребляет фрагмент данных. Также выражение «ячейка памяти используется» означает, что ячейка памяти ссылается на порт или несколько портов, где она используется. Выражение «ячейка памяти стирает использование» означает, что ячейка памяти теперь не ссылается на порт, где она использовалась.

2.7.2 Построение программы без коммуникаций

Первый этап программы представляет собой цикл из двух шагов, который повторяются каждый раз пока не выполнится критерий завершенности построения программы.

Критерий завершенности

Каждый сноплет в частично-определенной программе должен быть завершенным, и каждая ячейка памяти должна быть проинициализирована. Если хотя бы одно условие не выполняется, то цикл продолжает работать.

Шаг 1

У всех сноплетов, присутствующих в частично-определенной программе, для каждого входного свободного порта в цикле ищется ячейка памяти из существующих в частично-определенной программе, либо создается необходимая ячейка памяти с нужными данными из программной спецификаций. Далее у порта с ячейкой памяти осуществляется бинд.

Таким образом, первым шагом создаются новые ячейки памяти, у которых построены связи с входными портами.

Шаг 2

В начале второго шага создаётся пустой список для добавления в него промежуточных сниплетов подстановки операции. С помощью данных из программной спецификации осуществляется поиск фрагментов вычислений, которые не были ещё добавлены в частично-определённую программу в виде промежуточных сниплетов подстановки операции. Но добавляются в список только те сниплеты, у которых есть хотя бы один порт с фрагментом данных, для которого уже есть ячейка памяти.

Также дополнительно проверяются выходные порты добавляемого сниплета с список. Если хотя бы один выходной порт у данного сниплета совпадает хотя бы с одной ячейкой памяти, которая уже является инициализированной, то такой сниплет не добавляется в список. Такая дополнительная проверка нужна, чтобы корректно работала многовариантность между фрагментами вычислений. Если такое условие дополнительно существовать не будет, то фрагмент данных будет 2 раза инициализироваться, что невозможно в рамках определения фрагмента данных.

После того как формирование списка с сниплетами окончится может возникнуть две ситуации:

- Список пуст.
- В списке есть хотя бы один сниплет.

Рассмотрим сначала ситуации с имеющимся сниплетом в списке. Выбирается любой сниплет из списка, который добавляется в частично-определённую программу. Для каждого выходного порта добавленного сниплета осуществляется бинд с ячейкой памяти.

Теперь рассмотрим ситуацию, при которой список оказался пуст. Такая ситуация означает, что все возможные промежуточные сниплеты подстановки операции уже были добавлены. Это означает, что пора добавлять в частично-определённую программу инициализирующие сниплеты. Но перед этим нужно в частично-определённой программе проверить каждую неинициализированную ячейку памяти, что для неё нет сниплета с выходным портом, с которым можно

осуществить бинд. Если находятся такие выходные порты, то осуществляется бинд.

Далее из программной спецификации проверяется, есть ли инициализирующий сниппет подстановки операции. Если такой сниппет был описан во входном представлении, то он добавляется в частично-определённую программу. Иначе добавляются сниппеты ввода из переданных аргументов в программу, в которых инициализируются входные переменные, описанные во входном представлении программы. И после этого шага все ячейки памяти в частично-определённой программе являются инициализированными, а все порты в сниппетах являются завершёнными. А значит теперь сработает критерий завершенности построения частично-определённой программы.

2.7.3 Добавление коммуникаций в построенной программе

На 1 этапе уже может быть сконструирован частный случай параллельной программы — последовательная программа, если изначально во входном представлении указывалось, что фрагментированная программа должна будет сконструирована для 1 процесса. Если указано другое число процессов большее чем 1, то будут добавляться коммуникации.

Добавление коммуникационных сниппетов необходимо, когда в ячейке памяти инициализирование происходит на порту сниппета, у которого номер процесса отличается от номера процесса на порту сниппета, где эта ячейка памяти используется, а значит фрагмент данных, который будет потребляться в сниппете будет неинициализированным, что приведёт к ошибке во время исполнения сгенерированной программы. Для избежания этого добавляется коммуникационный сниппет и создаётся копия ячейки памяти.

Процесс добавление коммуникаций также начинается с конца. Создаётся очередь из названий фрагментов данных с правилом «первый пришёл — первый вышел». Изначально в очереди перечисляются названия выходных фрагментов данных фрагментированной программы, описанные во входном представлении. А после начинается цикл, который выполняется, пока в очереди есть названия фрагменты данных.

В цикле из очереди удаляется первый элемент с названием фрагмента данных и ищется в частично-определённой программе ячейка памяти с этим названием фрагмента данных. Затем для каждого использования ячейки памяти в сниплетах частично-определённой программы сравнивается номер процесса, на котором ячейка памяти используется и на каком номере процесса инициализируется. Если эти номера совпадают, значит коммуникация не требуется, так как используется ячейка памяти в рамках одного процесса. Если номера разные — значит надо создавать коммуникацию в частично-определённой программе.

При создании коммуникации происходит клонирование ячейки памяти в частично-определённой программе со всеми её данными. Но у клонированной ячейки памяти есть отличия. Такая ячейка памяти будет иметь ссылку на оригинальную ячейку памяти, чтобы они отличались. У клонированной ячейки памяти очищается текущее использование на порту сниплета, а у оригинальной ячейки памяти очищается инициализатор. Затем в частично-определённой программе создаётся коммуникационный сниплет. На входной порт коммуникационного сниплета осуществляется бинд с клонированной ячейкой памяти. А на выходной порт коммуникационного сниплета осуществляется бинд с оригинальной ячейкой памяти. Теперь номера процессов у оригинальной ячейки памяти для инициализации и использования одинаковые, и у клонированной ячейки памяти аналогичная ситуация, но с другим номером процесса.

Вернёмся в момент, когда алгоритм решал добавлять ли коммуникацию в частично-определённую программу или нет для каждого использования ячейки памяти в сниплетах. После прохождения этой процедуры для конкретной ячейки памяти нужно определить, какие будут следующие названия фрагментов данных для проверки на необходимость использования коммуникации в частично-определённой программе. Для этого определяется, в каком сниплете ячейка памяти была инициализирована. В этом сниплете выясняется названия фрагментов данных с помощью портов, в которых содержится информация о

названиях. И все названия фрагментов данных, которые не были ещё в очереди, добавляются в очередь на проверку необходимости добавления коммуникаций.

Цикл завершается, как только все названия фрагментов данных пройдут проверку на необходимость добавления коммуникаций в частично-определённую программу. После этого частично-определённая программа готова к генерации кода параллельной программы.

2.7.4 Генерация кода по построенной программе

Генерация кода частично-определённой программы начинается с того, что генерируется весь необходимый код, отвечающий за инициализацию всех необходимых компонентов, которые нужны для сгенерированной программы.

Частично-определённая программа сначала генерирует объявление всех используемых ячеек памяти как фрагментов данных в программе, а затем генерирует код для сниплетов. Для ячеек памяти особый порядок генерации не нужен, так как если их всех сгенерировать в самом начале, потом не придётся думать, перед каким сниплетом необходимо сгенерировать объявление ячейки памяти. Этим немного облегчается алгоритм генерации кода. А вот генерация сниплетов требует особого порядка. Сниплеты генерируют фрагменты вычислений, которые должны быть сгенерированы в правильном порядке — в порядке информационных зависимостей. Осуществить такой порядок помогут построенные связи в предыдущих этапах построения параллельной программы.

Осуществляется верный порядок в алгоритме с помощью создания двух списков из сниплетов. В одном списке перечислены все используемые сниплеты в частично-определённой программе. Такой список будем называть общим. Другой список изначально будет пустым, но постепенно будет пополняться в нужном порядке сниплетами. Такой список будем называть правильным. Из общего списка будут переноситься сниплеты в правильный список в нужном порядке сниплетов. Таким образом, получится, что общий список останется пустым, а правильный список будет со всеми сниплетами из частично-определённой программы, но только с правильным порядком. Останется просто преобразовать в составленном порядке каждый сниплет в нужный код.

Ещё для обеспечения нужного порядка перечисления сниплетов понадобится ещё один список с инициализированными ячейками памяти и с указанием, на каких номерах процессах каждая добавленная ячейка памяти была инициализирована. В этот список будет добавляться каждая ячейка памяти, которая инициализировалась в сниплете. Для каждого добавляемого сниплета в правильный список все фрагменты данных, которые являются вырабатываемым в данном сниплете, считаются инициализированными и добавляются в список с инициализированными ячейками памяти.

Чтобы обеспечить необходимый порядок, каждый сниплет проверяется на готовность переместиться в правильный список. Для этого в сниплете проверяются названия фрагментов данных на выходных портах, чтобы все ячейки памяти с такими же названиями не были ещё в списке с инициализированными ячейками памяти. При этом учитывается также номер процесса, используемый на сниплете. Необходимо, чтобы в ячейки памяти не присутствовал тот номер процесса, который указан в сниплете. Только с соблюдением этого условия можно добавлять сниплет в правильный список. А все выходные порты на данном сниплете добавляются в список с инициализированными ячейками памяти с номером процесса сниплета.

Таким образом, в ходе многократного перебора всех сниплетов из общего списка составляется правильный список с нужным порядком для генерации кода. Далее все сниплеты преобразуются в нужный код и получается готовый код параллельной программы на языке C++.

2.8 Эффективность сконструированной программы

Существенную ценность для любой поставленной задачи является поиск эффективных решений. В предыдущих пунктах описывался алгоритм конструирования параллельных программ. Но он конструирует обычную параллельную программу, к которой не применяются способы и алгоритмы получения именно эффективной параллельной программы.

Пользователь может самостоятельно увеличить эффективность генерируемой параллельной программы, если во входном представлении он

назначит каждому описываемому фрагменту вычислений рекомендуемый номер процесса. Такой способ повышения эффективности программы сработает, если сам пользователь понимает, как её увеличить с помощью назначения номеров процессов. Для самостоятельного назначения номеров процессов на фрагменты вычислений нужно обладать базовыми навыками в области параллельного программирования, чтобы конструировался эффективный вариант параллельной программы.

Но не все пользователи понимают, как эффективно распределить номера процессов по фрагментам вычислений во входном представлении. Поэтому в этом пункте я приведу 2 способа, которые могут повысить эффективность конструируемой программы.

2.8.1 Оценка качества конструируемой программы

В качестве первого способа повышения эффективности получаемой параллельной программы является подсчёт оценки качества для каждой конструируемой программы. В оценке качества представлены 2 критерия:

- Количество коммуникаций между процессами
- Распределение фрагментов вычислений между процессами

Первый критерий заключается в том, чтобы между процессами было наименьшее число коммуникаций. В данном случае, чем меньше коммуникационных сниплетов получится, тем быстрее сможет работать программа. Любая коммуникация в параллельной программе тратит какое-то количество времени, особенно когда от одного процесса к другому требуется перенести объёмный фрагмент данных.

Второй критерий будет оценивать равномерность распределения фрагментов вычислений по номерам процессов. Это нужно чтобы снизить вероятность простоев и обеспечить более равномерную нагрузку на оборудование.

Но оба эти критерия будут работать хорошо только в том случае, если время выполнения фрагмента вычислений приблизительно одинаковое у всех. Если в программе есть много фрагментов вычислений, у которых выполнение

может происходить несколько миллисекунд, а также есть много фрагментов вычислений, которые выполняются несколько секунд, то эти критерии могут неправильно осуществить распределения фрагментов вычислений между процессами. Может получиться так, что на один процесс попали все фрагменты вычислений, которые выполняются несколько миллисекунд, а на другой процесс попали фрагменты вычислений, выполняющиеся несколько секунд. Соответственно оба критерия должны учитывать, сколько по времени работает каждый из фрагментов вычислений, чтобы оценивать более корректно получаемую параллельную программу.

2.8.2 Использование градиентного спуска для оценки качества

В качестве второго способа повышения эффективности сгенерированной параллельной программы является применение метода градиентного спуска [23]. Он заключается в поиске самого быстрого варианта исполнения сгенерированной параллельной программы из нескольких вариантов.

Программа генерирует несколько разных вариантов параллельных программ, а затем пытается каждый вариант улучшить по времени работы. Каждый вариант постепенно достигает своего локального минимума по времени выполнения. После проведения, определённого количества попыток, которое может быть ограничено временем, выясняется, какой вариант получился самым быстрым при исполнении, т.е. какой вариант достиг лучшего глобального минимума по времени работы сгенерированной параллельной программы. Этот вариант сгенерированной параллельной программы предоставляется пользователю.

Улучшение работы сгенерированной параллельной программы можно осуществить с помощью изменения номеров процессов на сниппетах. Причём изменение номеров процессов должно быть на небольшом количестве сниппетов, например, 5% - 10% от общего количества сниппетов. За счёт изменения номеров процессов в сниппетах может получиться как более удачное распределение, так и менее удачное. Варианты, при которых не будут получаться улучшения во времени работы параллельной программы, будут откидываться, а

варианты, у которых получилось добиться улучшения, будут фиксироваться, и от них будут проводиться снова попытки получения более удачного распределения сниплетов по процессам.

2.9 Анализ полученных результатов

Таким образом, разработанная структура описания входного языка позволяет описать многие фрагментированные программы. Входное описание фрагментированной программы в генераторе представляется с помощью разработанных структур данных в виде конструкций, из которых генератор осуществляет конструирование параллельных фрагментированных программ. Представленный алгоритм конструирования позволяет генерировать код параллельных программ для разного количества процессов, обеспечивая коммуникации между ними. Также обеспечена многовариантность генерируемых программ, благодаря которым каждый раз будет генерироваться разные варианты параллельной фрагментированной программы. Рассмотрены теоретические способы повысить эффективность конструируемых программ.

Глава 3. Реализация

В этом разделе описано как форматировать входные данные для генератора параллельных программ. Представлена реализация структур данных и алгоритмов в генераторе. Описано, как осуществляется процесс генерации кода. Перечислены ограничения генератора параллельных программ, а также дано краткое руководство по генерации программы и её запуску.

3.1 Входные данные генератора параллельных фрагментированных программ

Генератору параллельных программ требуется для генерации параллельного фрагментированного кода как минимум два файла:

- Файл описания фрагментов вычислений — всегда один файл.
- Файл описания фрагментов кода — может быть более одного файла.

Файл с описанием фрагментов вычислений является входным описанием фрагментированной программы, представленным в п. 2.5. Для генератора параллельных программ файлы с описанием фрагментов кода описывают файл с фрагментами кода LuNA-программы. Такой файл для LuNA-программы содержит в себе функции, которые применяются в фрагментированной программе. Генератору нужно предоставить только сигнатуры этих функций с подробной информацией о фрагментах данных и переменных в удобном формате для работы алгоритма конструирования программы. Количество файлов описания фрагментов кода может быть несколько. Их количество зависит от выбора пользователя, если ему удобнее содержать описание фрагментов кода в нескольких файлах

Входной язык описания программы на данный момент пишется пользователем. Но в дальнейшем планируется автоматизировать этот процесс, поскольку файл с входным представлением не человеко-ориентированы, так как они создавались с целью машинной обработки. Входной язык будет получаться из программы, написанной на языке LuNA. Та же самая автоматизация планируется для файлов описания фрагментов кода, которые будут создаваться

из файла с фрагментами кода. Примеры формата файлов представлены на рисунках 1 и 2.

Все файлы, которые подаются на вход генератору параллельных программ описаны в формате json и состоят из пар «ключ-значение». В качестве «значения», может быть, вложенные объекты и массивы.

```
1  {
2    "using": [ "funcs.json" ],
3    "interface": {
4      "ranks": "1",
5      "input": [ "x", "y", "alpha", "two" ],
6      "output": "S2"
7    },
8    "operations": {
9      "calc_S": {
10       "function_type": "intermediate",
11       "code": "calc_area",
12       "inputs": [ "x", "y", "alpha" ],
13       "outputs": [ "S" ],
14       "arguments": {
15         "edge1": "x",
16         "edge2": "y",
17         "angle": "alpha",
18         "area": "S"
19       }
20     },
21     "double_S": {
22       "function_type": "intermediate",
23       "code": "mul_2",
24       "inputs": [ "S" ],
25       "outputs": [ "S2" ],
26       "arguments": {
27         "x": "S",
28         "y": "S2"
29       }
30     },
31     "mul_two": {
32       "function_type": "intermediate",
33       "code": "mul",
34       "inputs": [ "S", "two" ],
35       "outputs": [ "S2" ],
36       "arguments": {
37         "x": "S",
38         "y": "two",
39         "z": "S2"
40       }
41     }
42   },
43   "variables": {
44     "x": "real",
45     "y": "real",
46     "alpha": "real",
47     "S": "real",
48     "S2": "real",
49     "two": "real"
50   }
51 }
52
```

Рисунок 1 — пример формата файла описания фрагментов

```

1  {
2  "mul": {
3    "type": "c_function",
4    "parameters": [
5      {
6        "name": "x",
7        "ctype": "double",
8        "type": "real",
9        "access": "input"
10     },
11     {
12       "name": "y",
13       "ctype": "double",
14       "type": "real",
15       "access": "input"
16     },
17     {
18       "name": "z",
19       "ctype": "double*",
20       "type": "real",
21       "access": "output"
22     }
23   ],
24   "source": "ucodes.cpp",
25   "id": "c_mul"
26 },
27 "mul_2": {
28   "type": "c_function",
29   "parameters": [
30     {
31       "name": "x",
32       "ctype": "double",
33       "type": "real",
34       "access": "input"
35     },
36     {
37       "name": "y",
38       "ctype": "double*",
39       "type": "real",
40       "access": "output"
41     }
42   ],
43   "source": "ucodes.cpp",
44   "id": "c_mul2"
45 },
46 "calc_area": {
47   "type": "c_function",
48   "parameters": [
49     {
50       "name": "edge1",
51       "descr": "length of the first adjacent edge",
52       "ctype": "double",
53       "type": "real",
54       "access": "input"
55     },
56     {
57       "name": "edge2",
58       "descr": "length of the second adjacent edge",
59       "ctype": "double",
60       "type": "real",
61       "access": "input"
62     },
63     {
64       "name": "angle",
65       "descr": "angle between edges 1 & 2",
66       "ctype": "double",
67       "type": "real",
68       "access": "input"
69     },
70     {
71       "name": "area",
72       "descr": "area of the triangle",
73       "ctype": "double*",
74       "type": "real",
75       "access": "output"
76     }
77   ],
78   "source": "ucodes.cpp",
79   "id": "c_calc_area"
80 }
81 }

```

Рисунок 2 — пример формата файла описания фрагментов кода

Подробная инструкция по форматам файла будет в приложении:

- Приложение А — формат файла описания фрагментов вычислений.
- Приложение Б — формат файла описания фрагментов кода.

3.2 Реализация генератора параллельных фрагментированных программ

Генератор параллельных фрагментированных программ реализован на языке Python. Такой язык был выбран из-за простого и понятного синтаксиса языка, а также из-за высокой скорости разработки программ на этом языке. Реализация генератора находится в репозитории GitLab [24].

Генератор генерирует параллельный код на языке C++ и использует библиотеку программного интерфейса для передачи информации MPI между процессами.

Генератор параллельных фрагментированных программ переиспользует часть кода системы фрагментированного программирования. В частности из репозитория системы LuNA [25] используются файлы: `df.h`, `df.cpp`, `common.h`, `common.cpp`, `serializable.h`, `serializable.cpp`. Эти файлы позволяют использовать класс DF, а также использовать его в качестве типа для фрагментов данных в генераторе.

Генератору параллельных программ в качестве аргумента при запуске передаётся путь к файлу с входным представлением фрагментированной программы. Рекомендуется клонировать репозиторий и в корневой папке проекта разместить файлы с входным представлением. По завершению работы генератора пользователю в консоли выводится текст сгенерированной параллельной программы, а также создаётся папка «MPI_Test_code», если её не существует, в которой будет создан файл «MPI_code.cpp» с таким же кодом, который выводился в консоль. Более подробно о запуске сгенерированной программе в п. 3.2.5.

3.2.1 Реализация структуры данных в генераторе

Структуры, перечисленные в п. 2.6., реализованы в генераторе в качестве классов. Каждый этот класс в репозитории является отдельным файлом. Его название будет записано рядом с названием в скобках. Ниже будет изложена краткая информация о реализации каждой структуре.

Программная спецификация («prog_spec.py»)

В начале работы генератора создаётся объект этого класса, который загружает внутрь себя файлы с описанием фрагментов вычислений и фрагментов кода, чтобы потом быстро достать нужную информацию в процессе работы алгоритма конструирования. Также этот класс проверяет корректность формата файлов и следит, чтобы было нужное количество инициализирующих и финальных сниплетов в файле описания фрагментов вычислений.

Частично-определённая программа («partialprogram.py»)

После создания объекта программной спецификации создаётся объект частично-определённой программы. Единственным параметром, который передаётся при создании объекта класса частично-определённой программы, является объект программной спецификации. Частично-определённая программа инициализируется с пустыми списками объектов сниплетов и ячеек памяти. Объект программной спецификации присваивается приватной переменной «_ps». Количество используемых процессов также присваивается приватной переменной «_ranks», информация которых берётся из программной спецификации.

Далее в список сниплетов добавляется финальный сниплет. С помощью метода «get_final_operation()» выясняется и создаётся финальный сниплет. А затем добавляется в список частично-определённой программы.

В частично-определённой программе при добавлении ячеек памяти и сниплетов в списки присваивается порядковый идентификатор. Добавление осуществляется с помощью методов «add_snippet()» и «add_mcell()».

Ячейка памяти («mcell.py»)

При создании объекта ячейки памяти в ней инициализируются поля с названием переменной «_var_id», дополнительной информацией «_spec», в которой указан тип переменной, а также инициализируются пустой массив «_bindings», в котором указывается на каких портах и сниплетах используется эта ячейка памяти, и «_initializer_id», в котором указана та же самая информация, но только в единственном значении, так как ячейка памяти инициализируется только один раз. Ещё в ячейке памяти есть поле «_parent_id», которое по умолчанию принимает значение «None». Это поле создано, чтобы отличать ячейки памяти, которые были клонированы на других процессах. В таком случае в этом поле будет содержаться идентификатор оригинальной ячейки памяти, с которой осуществлялось клонирование.

Элементами массива «_bindings» являются перечисление строк из тройки чисел, разделённых двоеточием — «rank:snip_id:port_id». Первое число «rank»

указывает на номер процесса, на котором будет использоваться сниплет, второе число «`snip_id`» является идентификатором сниплета, а последнее число «`port_id`» — порядковый номер порта в сниплете. Т.е. построенная связь ячейки памяти с сниплетом идентификатора «2», на порте «1», работающем на «0» номере процесса, выглядит следующим образом: «0:2:1». Таким образом, с помощью метода «`notify_bound()`» осуществляется построения связи между ячейкой памяти и портом, добавляя в массив «`_bindings`» тройку чисел. Аналогично это работает и с методом «`set_initialized()`», однако в этом массиве может быть только одна тройка чисел.

Порт («`port.py`»)

При инициализации порт содержит порядковый номер «`pos`» (начинается с 0), информацию о типе порта (входной или выходной) в «`_param_spec`», названии используемой переменной «`_var_id`», фрагмент вычислений, в котором используется порт «`_op_id`», а также порт содержит некоторую информацию из сниплета — идентификатор сниплета «`snip_id`» и номер процесса сниплета «`_rank`». Ещё порт инициализирует поле «`_binding`» значением `None`.

Порт строит связь с ячейкой памяти с помощью поля «`_binding`», в котором указывается идентификатор ячейки памяти. Осуществляется это с помощью метода «`bind()`». Также в «`bind()`» используется метод «`get_uid()`», чтобы сформировать тройку чисел для ячейки памяти, которые нужны при применении методов «`notify_bound()`» при входном порте или «`set_initialized()`» при выходном порте. Таким образом, в порте есть информация о используемой ячейке памяти, а в ячейке памяти есть информация о используемом порте. Поэтому для построения связи между ними достаточно использовать метод объекта порта «`bind()`».

Сниплет («`Sniplet.py`»)

Сниплеты имеют разные виды, но все эти виды наследуются от основного класса «`Sniplet`». Сам класс в себе только содержит идентификатор сниплета, а потомки содержат в себе ещё некоторую другую информацию.

Сопоставим реализацию сниплетов между файлами в репозитории:

- Сниллет подстановки операции — «InvokeOperationSnip.py».
- Коммуникационный сниллет — «SendRecvSnip.py».
- Сниллет вывода в консоль — «VarToStdoutSnip.py».
- Сниллет ввода из переданных аргументов в программе — «ArgvSnip.py».

Название классов совпадает с названиями файлов. Далее я буду называть виды сниллетов через название классов.

При конструировании программы алгоритм назначает номер процесса, на котором будет выполняться сниллет. Если во входном представлении отсутствует рекомендованный номер процесса, то этот номер выберет датчик случайных чисел. Так работает для сниллетов InvokeOperationSnip и VarToStdoutSnip. Они в себе содержат переменную «_rank», в которой будет записан номер процесса. ArgvSnip по умолчанию имеет значение переменной «_rank» «-1», что означает, что выполнится сниллет на всех процессах. А в SendRecvSnip сниллете содержится номер процесса, куда следует отправлять фрагмент данных или переменную «_send_to_rank» и номер процесса, откуда следует получать «_recv_from_rank». В поле «_type_sinp» инициализируется название тип сниллета. Каждый сниллет инициализирует порт или список портов, если их несколько, в «_port» в соответствии с описанием в предоставленном входном представлении. Сниллет InvokeOperationSnip внутри себя содержит название применяемого фрагмента вычислений «_op_id», а также программную спецификацию в переменной «_op_spec». Каждый сниллет имеет метод «generate()», который сгенерирует необходимый код при генерации кода. Об этом подробнее будет в п. 3.2.3.

3.2.2 Реализация алгоритма конструирования

Алгоритм конструирования программы реализован в файле «main.py». В начале программы инициализируется программная спецификация и частично-определённая программа с финальным сниллетом, а затем начинается первый этап конструирования программы в цикле «while», где проверяется критерий завершения построения программы с помощью метода «is_final()». Пока программа не построена, делаем шаг 1 с помощью метода «step_one()» и затем

шаг 2 с помощью метода «step_two()». Подробное описание осуществления шагов можно посмотреть в п. 2.7.2.

Как только программа прошла критерий завершенности успешно то осуществляется следующий этап конструирования — добавление коммуникаций в программе с помощью метода «add_mpi_snippets()».

Далее осуществляется последний этап — генерация кода. Этот этап запускается методом «generate()» у частично-определённой программы. О нём более подробнее в следующем п. 3.2.3.

3.2.3 Генерация кода

Генерация кода осуществляется путём вызова у структурных элементов метода «generate()». Он есть у всех видов сниплетов, частично-определённой программы, портов и ячеек памяти.

У ячейки памяти метод «generate()» генерирует строку с объявлением переменной или фрагментов данных с помощью данных из переменных «_var_id» и «_spec».

Метод «generate()» у портов возвращает строку с названием переменной, чтобы использовать её для генерации названия переменных и фрагментов данных у сниплетов.

Каждый сниплет содержит часть генерируемого кода. Сниплет «InvokeOperationSnip» при вызове метода «generate()» определяет с помощью переменной «_op_spec», какой фрагмент кода (процедуру) нужно использовать при генерации кода. С помощью переменной «_port» для каждого порта вызывается метод «generate()», чтобы сгенерировать названия переменных, которые используются в сигнатуре вызываемой процедуры. С помощью переменной «_rank» перед вызываемой процедурой генерирует код с условием, на каком номере процесса должна выполняться процедура.

Сниплет SendRecvSnip использует в генерации кода программный интерфейс для передачи информации MPI. Для переменных типа int и double генерируются код с функциями «MPI_Send()» и «MPI_Recv». Данные, на каком процессе нужно использовать эти функции, а также на какой процесс отправлять

и на каком процессе принимать, берутся из переменных «_send_to_rank» и «_recv_from_rank». С помощью переменной «_var_id» в сниппете определяется название переменной, которая подставляется в MPI функции. Для фрагментов данных требуется дополнительные действия перед отправкой на другие процессы, поэтому для них отдельно реализована функция, которая генерируется в начале кода программы и затем SendRecvSnip сниппет использует её для осуществления коммуникаций.

ArgvSnip по умолчанию работает на всех процессах. Этот сниппет генерирует код вызова функции «sscanf()», чтобы прочитать аргументы переданной программы. А индекс нужной переменной для считывания из «argv» такой же, как и индекс входных переменных в программе в файле с входным представлением.

VarToStdoutSnip генерирует код вызова функции «printf()» для вывода переменной в консоль. Перед «printf()» пишется условие, чтобы выполнение функции осуществлялось на нужном процессе.

Частично-определённая программа внутри себя имеет некоторые части кода, которые просто вставляются в сгенерированный код. Такие части кода буду называть заготовленным кодом. При вызове метода «generate()» частично-определённая программа генерирует следующий код:

- Заготовленные команды компиляции и команду для выполнения сгенерированной программы на нескольких процессах.
- Заготовленные директивы «#include» с файлом стандартной библиотеки ввода и вывода stdio.h и библиотекой MPI. Также генерируются директивы «#include» с файлами фрагментов кода, которые используются во входном представлении при описании фрагментов кода с помощью метода «get_include_files()».
- Генерируются директивы «#define» из входного представления, если они были указаны.
- Генерируется заготовленный код, который используется SendRecvSnip сниппетом.

- Генерируются сигнатуры функций фрагментов кода, которые используются в сгенерированной программе.
- Вставляется заготовленный код с инициализацией коммуникационного интерфейса MPI.
- Генерируются объявления переменных и фрагментов кода с помощью вызова метода «generate()» для каждой ячейки памяти.
- Генерируются код для каждого сниплета также вызовом метода «generate()». Причем сначала сниплеты сортируют свой порядок в соответствии п.2.7.4. с помощью метода «_sorted_snip_ids()».
- Вставляется заготовленный код с завершением работы коммуникационного интерфейса MPI.

Таким образом, генерируется код программы, который записывается в файл «MPI_test_code.cpp».

3.2.4 Ограничения генератора параллельных программ

Генератор параллельных фрагментированных программ обладает ограниченным функционалом, поэтому генератор не способен конструировать часть фрагментированных программ, которые может конструировать система LuNA. Это связано с более упрощенным входным представлением конструируемой программы, чем входное представление в системе LuNA, а также с реализованными возможностями генератора. Ограничения у реализованного генератора, следующие:

- Вся сгенерированная программа содержится в единственной функции «main(int argc, char **argv)». Генератор параллельных программ не генерирует отдельные функции, помимо main, за исключением функции send_recv_df(), которая используется сниплетом SendRecvSnip.
- Поддерживаются целочисленные переменные типа «int» и вещественные типа «double». Также поддерживаются фрагменты данных, которые являются объектами класса DF из системы LuNA.

- Генератор не поддерживает оператора индексирования для фрагментов данных. Все фрагменты данных в программе должны иметь уникальные названия.
- Генератор работает только с заранее определенным конечным количеством фрагментов кода и переменных с фрагментами данных.
- Генератор не имеет конструкций, которые бы поддерживали циклы. Циклы нужно самостоятельно раскручивать во входном представлении.
- Генератор не имеет конструкций, позволяющие использовать условия.
- Генератор не конструирует выражения. Все выражения, осуществляющие математические операции, должны быть внутри фрагментов кода.
- При конструировании переменные и фрагменты данных объявляются всегда в начале генерируемого кода, а не перед тем, как они используются в функциях.
- Генератор не имеет ни в каком виде сборку мусора. Все инициализированные фрагменты данных остаются до завершения программы.
- Начальным сноплетом может быть только один инициализирующий сноплет или несколько сноплетов `ArgvSnip`. Причем `ArgvSnip` поддерживает только типы `int` и `double`.
- Последним сноплетом может быть только единственный финальный сноплет или единственный сноплет `VarToStdoutSnip`, который поддерживает типы только `int` и `double`.

3.2.5 Запуск сгенерированной параллельной программы

После генерации, параллельная фрагментированная программа будет находиться в папке «`MPI_Test_code`», где будет создан или обновлён файл «`MPI_code.cpp`». В этой же папке должны находиться файлы с фрагментами кода, а также 6 файлов из LuNA репозитория: `df.h`, `df.cpp`, `common.h`, `common.cpp`, `serializable.h`, `serializable.cpp`.

Далее необходимо осуществить компиляцию сгенерированного кода командой, предложенной генератором параллельных программ. Эта команда

записывается как комментарий в коде первой строкой, а второй строкой — команда для исполнения сгенерированной программы:

- `mpicxx -o run MPI_code.cpp df.cpp common.cpp serializable.cpp -O3`
- `mpirun -np 4 ./run`

3.3 Анализ реализации генератора параллельных программ

Входное представление в формате файлов «json» позволяет описать для генератора фрагментированную программу. В качестве языка программирования для генератора был выбран Python. Реализованные структуры данных в виде классов в генераторе обеспечивают алгоритму управлять конструкциями, чтобы строить программу в порядке информационных зависимостей. Фрагменты данных в сгенерированной программе являются объектами класса DF, который был взят из системы фрагментированного программирования LuNA. Также были перечислены все ограничения реализованного генератора и дано краткое руководство по запуску сгенерированных программ.

Глава 4. Тестирование

Структура раздела состоит из проведения тестирования с помощью подготовленных трёх задач:

1. Вычисление удвоенной площади треугольника.
2. Суммирование двух векторов.
3. Практическая задача: решение уравнения Пуассона в трехмерной области методом Якоби в одномерной декомпозиции.

4.1 Тестирование генератора на простых задачах

Для каждой задачи можно выделить одно общее требование: сгенерированный код программы должен компилироваться и выполняться без ошибок при верном формате входного представления.

Для первой задачи генератор параллельных программ должен сконструировать простое вычисление площади треугольника с помощью двух сторон и угла между ними, а затем удвоить результат площади. В конце программы надо вывести в консоль удвоенную площадь треугольника. Входные данные должны будут переданы с помощью аргументов при запуске сгенерированной программы.

С помощью первой задачи нужно было проверить следующие требования:

- Конструирование программы для исполнения на одном процессе.
- Конструирование программы для исполнения на нескольких процессах, а также построение корректных коммуникаций между процессами.
- Многовариантность между разными фрагментами вычислений.
- Работоспособность всех видов и разных типов сниплетов в программе.

Вторая задача инициализирует два единичных вектора, а затем складывает их. Каждый вектор разделён на 4 части. В конце программы считается сумма элементов результирующего вектора. Каждая часть вектора в этом тесте является фрагментом данных. В фрагменте кода «`c_initVector_params`» указывается размер каждой части фрагмента. При размере 10 результирующая сумма будет 80.

Основное требование этой задачи — проверить работоспособность фрагментов данных, а также корректную коммуникацию между процессами при использовании `SendRecvSnip` снippets с фрагментами данных. Также проверить генерацию параллельной программы с заранее выбранным номером процесса для выполнения фрагмента вычислений.

Третья задача является объёмной, её подготовка и результаты представлены в разделе 4.4.

4.2 Тестирование задачи вычисления удвоенной площади треугольника

Для самого первого примера с тестом я приведу полные листинги всех используемых файлов. Для других примеров будут представлены ограниченные листинги, так как они слишком длинные. На рисунке 1 представлено файл описания фрагментов вычислений, а на рисунке 2 — файл описания фрагментов кода (оба рисунка были представлены в п. 3.1.). На рисунке 3 файл с реализацией фрагментов кода. В этих файлах изначально предусмотрено два фрагмента

```
1  #include <iostream>
2  #include <stdio.h>
3  #include <math.h>
4  #include "df.h"
5
6  void c_calc_area(double e1, double e2, double an, double* ar) {
7      *ar = e1 * e2 * sin(an) * 0.5f;
8  }
9
10 void c_mul2(double x, double* y) {
11     *y = x * 2;
12 }
13
14 void c_mul(double x, double y, double* z) {
15     *z = x * y;
16 }
17
```

Рисунок 3 — реализация фрагментов кода для задачи вычисления удвоенной площади

вычислений, которые вырабатывают одну и ту же переменную `S2`. Это сделано для проверки многовариантности конструирования параллельной программы. Для её демонстрации приведено 2 листинга на рисунке 4 и 5. На 4 рисунке алгоритмом конструирования был выбран фрагмент вычислений «`double_S`» с

```

1  □//mpicxx -o run MPI_code.cpp df.cpp common.cpp serializable.cpp
2  □//mpirun -np 1 ./run
3
4  □#include <stdio.h>
5  □#include "mpi.h"
6
7  □#include "ucodes.cpp"
8
9  // user codes declarations
10 void c_mul(double x, double y, double* z);
11 void c_mul2(double x, double* y);
12 void c_calc_area(double edge1, double edge2, double angle, double* area);
13
14 □int main(int argc, char **argv) {
15     int rank, size;
16     MPI_Init(&argc, &argv);
17     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18     MPI_Comm_size(MPI_COMM_WORLD, &size);
19
20     double S2;
21     double S;
22     double x;
23     double y;
24     double alpha;
25     sscanf(argv[1], "%lf", &y);
26     sscanf(argv[2], "%lf", &x);
27     sscanf(argv[3], "%lf", &alpha);
28     if (rank == 0) c_calc_area(x, y, alpha, &S); // calc_S
29     if (rank == 0) c_mul2(S, &S2); // double_S
30     if (rank == 0) printf("%lf\n", S2);
31
32     MPI_Finalize();
33 }

```

Рисунок 4 — пример сгенерированной программы с фрагментом вычислений «double_S», который использует фрагмент кода «c_mul2()»

```

1  □//mpicxx -o run MPI_code.cpp df.cpp common.cpp serializable.cpp
2  □//mpirun -np 1 ./run
3
4  □#include <stdio.h>
5  □#include "mpi.h"
6
7  □#include "ucodes.cpp"
8
9  // user codes declarations
10 void c_mul(double x, double y, double* z);
11 void c_mul2(double x, double* y);
12 void c_calc_area(double edge1, double edge2, double angle, double* area);
13
14 □int main(int argc, char **argv) {
15     int rank, size;
16     MPI_Init(&argc, &argv);
17     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18     MPI_Comm_size(MPI_COMM_WORLD, &size);
19
20     double S2;
21     double S;
22     double x;
23     double y;
24     double alpha;
25     double two;
26     sscanf(argv[1], "%lf", &y);
27     sscanf(argv[2], "%lf", &x);
28     sscanf(argv[3], "%lf", &alpha);
29     sscanf(argv[4], "%lf", &two);
30     if (rank == 0) c_calc_area(x, y, alpha, &S); // calc_S
31     if (rank == 0) c_mul(S, two, &S2); // double_S
32     if (rank == 0) printf("%lf\n", S2);
33
34     MPI_Finalize();
35 }

```

Рисунок 5 — пример сгенерированной программ с фрагментов вычислением «mul_two», который использует фрагмент кода «c_mul()»

соответствующим фрагментом кода «mul_2», а на 5 рисунке алгоритм выбрал «mul_two» с фрагментом кода «mul».

В ходе конструирования использовались 3 сниплета: ArgvSnip, VarToStdoutSnip и InvokeOperationSnip промежуточного типа. Все сниплеты сгенерировали код, который компилируется консольной командой «mpicxx», а также выполняются с помощью консольной команды «mpirun» для 1 процесса. Команды для компиляции и исполнения генерируются вместе с кодом и пишутся комментарием в начале сгенерированного кода. На рисунке 4 и 5 представлен частный случай параллельной программы — последовательная программа, так как всё выполняется в рамках одного процесса.

```

18 void c_init_args(int argc, char** argv, double* x,
19 double* y, double* alpha, double* two) {
20     sscanf(argv[1], "%lf", x);
21     sscanf(argv[2], "%lf", y);
22     sscanf(argv[3], "%lf", alpha);
23     sscanf(argv[4], "%lf", two);
24 }
25
26 void c_print_result(double out) {
27     printf("%lf\n", out);
28 }

```

Рисунок 6 — добавленные фрагменты кода

Далее немного усложняем задачу. Добавим несколько фрагментов кода в файл, чтобы проверить InvokeOperationSnip в качестве инициализирующего и финального сниплета. Часть фрагментов кода, которые были добавлены, представлено на рисунке 6. Соответственно, добавленная часть файла описания фрагментов вычислений на рисунке 7, а для фрагментов кода — на рисунке 8. Также укажем, что программа будет конструироваться для 4 процессов. Тем самым проверим работоспособность SenRecvSnip сниплета и корректность построенных коммуникаций между процессами.

```

42 "init_args": {
43     "function_type": "initial",
44     "code": "init_args",
45     "inputs": [],
46     "outputs": [ "x", "y", "alpha", "two" ],
47     "arguments": {
48         "argc": "argc",
49         "argv": "argv",
50         "x": "x",
51         "y": "y",
52         "alpha": "alpha",
53         "two": "two"
54     }
55 },
56 "finish_program": {
57     "function_type": "final",
58     "code": "print_result",
59     "inputs": [ "S2" ],
60     "outputs": [],
61     "arguments": {
62         "S2": "S2"
63     }
64 }
65 },

```

Рисунок 7 — добавленная часть фрагментов кода в файл описания фрагментов кода

```

80     },
81     "init_args": {
82         "type": "c_function",
83         "parameters": [
84             {
85                 "name": "argc",
86                 "ctype": "int",
87                 "access": "system"
88             },
89             {
90                 "name": "argv",
91                 "ctype": "char**",
92                 "access": "system"
93             },
94             {
95                 "name": "x",
96                 "ctype": "double*",
97                 "type": "real",
98                 "access": "output"
99             },
100            {
101                "name": "y",
102                "ctype": "double*",
103                "type": "real",
104                "access": "output"
105            },
106            {
107                "name": "alpha",
108                "ctype": "double*",
109                "type": "real",
110                "access": "output"
111            }
112        ],
113        "source": "ucodes.cpp",
114        "id": "c_init_args"
115    },
116    "print_result": {
117        "type": "c_function",
118        "parameters": [
119            {
120                "name": "S2",
121                "ctype": "double",
122                "type": "real",
123                "access": "input"
124            }
125        ],
126        "source": "ucodes.cpp",
127        "id": "c_print_result"
128    },
129    {
130        "name": "alpha",
131        "ctype": "double*",
132        "type": "real",
133        "access": "output"
134    }

```

Рисунок 8 — добавленная часть фрагментов вычисления в файл описания фрагментов вычислений

На рисунках 9 и 10 представлено часть кода двух вариантов сгенерированной программы, у которых осуществлена многовариантность фрагментов вычислений, а также разное распределение фрагментов вычислений по процессам. Оба варианта кода компилируется и выполняется, значит все коммуникации и сниплеты сгенерировали корректный код и все требования к конструированию программы выполнены.

```

22     double S2;
23     double S;
24     double x;
25     double y;
26     double alpha;
27     double two;
28     c_init_args(argc, argv, &x, &y, &alpha, &two); // init_args
29     if (rank == 2) c_calc_area(x, y, alpha, &S); // calc_S
30     if (rank == 2) MPI_Send(&S, 1, MPI_DOUBLE, 3, 643, MPI_COMM_WORLD);
31     if (rank == 3) MPI_Recv(&S, 1, MPI_DOUBLE, 2, 643, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
32     if (rank == 3) c_mul2(S, &S2); // double_S
33     if (rank == 3) MPI_Send(&S2, 1, MPI_DOUBLE, 0, 412, MPI_COMM_WORLD);
34     if (rank == 0) MPI_Recv(&S2, 1, MPI_DOUBLE, 3, 412, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
35     if (rank == 0) c_print_result(S2); // finish_program

```

Рисунок 9 — пример сгенерированной части кода с фрагментом кода «c_mul2()».

```

26     double S2;
27     double S;
28     double two;
29     double x;
30     double y;
31     double alpha;
32     c_init_args(argc, argv, &x, &y, &alpha, &two); // init_args
33     if (rank == 1) c_calc_area(x, y, alpha, &S); // calc_S
34     if (rank == 1) MPI_Send(&S, 1, MPI_DOUBLE, 2, 693, MPI_COMM_WORLD);
35     if (rank == 2) MPI_Recv(&S, 1, MPI_DOUBLE, 1, 693, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
36     if (rank == 2) c_mul(S, two, &S2); // mul_two
37     if (rank == 2) MPI_Send(&S2, 1, MPI_DOUBLE, 0, 530, MPI_COMM_WORLD);
38     if (rank == 0) MPI_Recv(&S2, 1, MPI_DOUBLE, 2, 530, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
39     if (rank == 0) c_print_result(S2); // finish_program

```

Рисунок 8 — пример сгенерированной части кода с фрагментом кода «c_mul()».

4.3 Тестирование задачи суммы векторов

Для реализации операции сложения с векторами использовался класс DF из системы LuNA. Снимлет SendRecvSnip при генерации кода применяет функцию «send_recv_df()» для коммуникации между фрагментами вычисления, которая генерируется в начале кода (рисунок 11). Часть сгенерированного кода

```

13 void send_recv_df(DF& df, int send_rank, int recv_rank, int rank, int tag) {
14     if (rank == send_rank) {
15         size_t size_send = df.get_serialization_size();
16         void* buf_send = (void*)malloc(size_send);
17         size_t new_size_send = df.serialize(buf_send, size_send);
18         MPI_Send(buf_send, size_send, MPI_BYTE, recv_rank, tag, MPI_COMM_WORLD);
19         free(buf_send);
20     }
21     if (rank == recv_rank) {
22         int size_recv;
23         MPI_Status status;
24         MPI_Probe(send_rank, tag, MPI_COMM_WORLD, &status);
25         MPI_Get_count(&status, MPI_BYTE, &size_recv);
26         void* buf_recv = (void*)malloc(size_recv);
27         MPI_Recv(buf_recv, size_recv, MPI_BYTE, send_rank, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
28         size_t read = df.deserialize(buf_recv, (size_t)size_recv);
29         free(buf_recv);
30     }
31 }

```

Рисунок 9— реализация функции для отправки фрагмента данных с одного номера процесса на другой

представлена на рисунке 12. Сгенерированная программа компилируется и после выполнения вывела в консоль сумму элементов результирующего массива (рисунок 13).

В качестве ещё одного теста с данным примером — генерация параллельной программой с заранее определённым номером процесса для исполнения фрагмента вычислений. Для этого добавим к каждому

```

60     c_initVector_params(&size_vector, &value); // init_args
61     if (rank == 3) c_initVector(b_3, size_vector, value); // c_initVector_b_3
62     if (rank == 0) c_initVector(a_1, size_vector, value); // c_initVector_a_1
63     if (rank == 3) c_initVector(b_1, size_vector, value); // c_initVector_b_1
64     if (rank == 0) c_initVector(a_0, size_vector, value); // c_initVector_a_0
65     if (rank == 1) c_initVector(b_2, size_vector, value); // c_initVector_b_2
66     if (rank == 1) c_initVector(a_3, size_vector, value); // c_initVector_a_3
67     if (rank == 1) c_initVector(b_0, size_vector, value); // c_initVector_b_0
68     if (rank == 3) c_initVector(a_2, size_vector, value); // c_initVector_a_2
69     send_recv_df(a_0, 0, 1, rank, 792);
70     if (rank == 1) c_sumVectors(a_0, b_0, c_0, size_vector); // c_sumVectors_0
71     send_recv_df(c_0, 1, 0, rank, 777);
72     send_recv_df(a_1, 0, 3, rank, 630);
73     if (rank == 3) c_sumVectors(a_1, b_1, c_1, size_vector); // c_sumVectors_1
74     send_recv_df(c_1, 3, 0, rank, 577);
75     send_recv_df(a_2, 3, 1, rank, 4);
76     if (rank == 1) c_sumVectors(a_2, b_2, c_2, size_vector); // c_sumVectors_2
77     send_recv_df(c_2, 1, 0, rank, 216);
78     send_recv_df(a_3, 1, 3, rank, 967);
79     if (rank == 3) c_sumVectors(a_3, b_3, c_3, size_vector); // c_sumVectors_3
80     send_recv_df(c_3, 3, 0, rank, 213);
81     if (rank == 0) c_finilize(c_0, c_1, c_2, c_3, size_vector); // finish_program

```

Рисунок 12 — часть сгенерированной программы задачи суммы векторов без ручного корректирования по номерам процессов

описываемому фрагменту вычислений пару «ключ-значение» «rank» с нужным номером. А затем сгенерируем параллельную фрагментированную программу. Результат части сгенерированного кода представлен на рисунке 14. Он также является компилируемым и выполняемым. Результат компиляции и выполнения такой же, как и первом случае (рисунок 13). В данном случае код программы получился теоретически более быстрым при выполнении, так как коммуникаций между процессами меньше за счет грамотного распределения фрагментов данных по процессам.

В то же время на рисунке 12, генератор параллельных программ распорядился ресурсами не эффективно. Распределение между узлами получилось неравномерное. Третий процесс (rank = 2) остался без работы и простаивает всю программу, ожидая её завершения.

```

andrey@ZenBook: /mnt/d/Files/Visual Studio/LuNa/Python MPI Generator/MPI_test_code$
mpicxx -o run MPI_code.cpp df.cpp common.cpp serializable.cpp -O3
andrey@ZenBook: /mnt/d/Files/Visual Studio/LuNa/Python MPI Generator/MPI_test_code$
mpirun -np 4 ./run
sum = 80.000000

```

Рисунок 13 — результат компиляции и выполнения программы задачи суммы векторов

```

60     c_initVector_params(&size_vector, &value); // init_args
61     if (rank == 3) c_initVector(b_3, size_vector, value); // c_initVector_b_3
62     if (rank == 2) c_initVector(b_2, size_vector, value); // c_initVector_b_2
63     if (rank == 1) c_initVector(a_1, size_vector, value); // c_initVector_a_1
64     if (rank == 0) c_initVector(b_0, size_vector, value); // c_initVector_b_0
65     if (rank == 3) c_initVector(a_3, size_vector, value); // c_initVector_a_3
66     if (rank == 3) c_sumVectors(a_3, b_3, c_3, size_vector); // c_sumVectors_3
67     if (rank == 0) c_initVector(a_0, size_vector, value); // c_initVector_a_0
68     if (rank == 0) c_sumVectors(a_0, b_0, c_0, size_vector); // c_sumVectors_0
69     if (rank == 2) c_initVector(a_2, size_vector, value); // c_initVector_a_2
70     if (rank == 2) c_sumVectors(a_2, b_2, c_2, size_vector); // c_sumVectors_2
71     if (rank == 1) c_initVector(b_1, size_vector, value); // c_initVector_b_1
72     if (rank == 1) c_sumVectors(a_1, b_1, c_1, size_vector); // c_sumVectors_1
73     send_recv_df(c_1, 1, 0, rank, 170);
74     send_recv_df(c_2, 2, 0, rank, 597);
75     send_recv_df(c_3, 3, 0, rank, 892);
76     if (rank == 0) c_finalize(c_0, c_1, c_2, c_3, size_vector); // finish_program

```

Рисунок 10 — часть сгенерированной программы задачи суммы векторов с ручным корректированием по номерам процессов

4.4 Тестирование генератора на практической задаче

В этом разделе будет описана разработка входного представления для практической задачи — решение уравнения Пуассона в трехмерной области методом Якоби в одномерной декомпозиции. А также сравнение результатов работы LuNA-программы и сгенерированной параллельной фрагментированной программы.

4.4.1 Подготовка LuNA-программы для тестирования

У этой задачи есть реализация для системы LuNA, которая находится в репозитории в папке: `examples/tests_luna/poild`. Но его пришлось немного переработать. Во-первых, ограничиться 10 итерациями в цикле. Этого вполне будет достаточно для проведения тестирования. Во-вторых, выводить значение «`max[t]`» на каждой итерации для отслеживания, чтобы это значение стабильно уменьшалось. Это будет признаком, что программа корректно работает. В-третьих, удалить часть программы, которая не влияет на полученный результат. В данном случае из версии задачи с репозитория были удалены строки с 140 по 175. Итоговый вариант LuNA-программы сохранён в репозитории генератора параллельных программ в папку «`Test_LuNA`».

4.4.2 Создание входного представления

Входное представление пока что пишется вручную пользователем. Но для этого примера требуется создать большое входное представление, поскольку количества процессов и фрагментов вычислений будет в несколько раз больше, чем в предыдущих тестах. Для автоматизации процесса создания входного представления для этой задачи создан скрипт, который генерирует файл с описанием фрагментов вычислений.

При запуске скрипта в его аргументы передаётся три значения:

- количество процессов;
- количество итераций;
- флаг для генерации с правильным распределением фрагментов данных по процессам (цифра 1 в аргументе активирует этот флаг)

Скрипт создаёт файл описания фрагментов вычислений. Также скрипт создаёт файл описания нескольких фрагментов кода в отдельный файл и создаёт дополнительный файл с фрагментами кода на C++.

Скрипт находится в папке «poild_generator» и называется «poild_create.py». Файл описания фрагментов вычислений для четырёх процессов при создании будет называться «poild_4_generate.json». Файл с описанием фрагментов кода — «funcs_poild_4_generate.json», а файл с реализацией фрагментов кода — «ucodes_poild_4_generate.cpp», который сохранится в папку «MPI_test_code».

4.4.3 Отличия LuNA-программы и сгенерированной программы

В LuNA-программе используется объявление структурированного фрагмента кода «reduce_max()». Его я заменил на один фрагмент вычислений, который ищет максимальное число из локальных максимумов. Причем в зависимости от количества процессов этот фрагмент вычислений имеет разное количество выходных портов. Именно поэтому при создании входного представления меняется часть фрагментов кода и файл описания фрагментов кода.

Также в сгенерированной программе используются искусственные информационные зависимости с помощью целочисленных переменных, которые инициализируются в инициализирующем снипете. Их количество также зависит от количества процессов генерируемой программы. Аналогичная ситуация с финальным сниплетом. Количество входных портов зависит от количества процессов, на котором будет работать фрагментированная программа. Это также меняет количество аргументов во фрагменте кода.

Такие изменения не должны сильно повлиять на время исполнения программы.

4.4.4 Подготовка к запуску

Решение уравнения Пуассона в трехмерной области методом Якоби использует файл «pm.h» в качестве входных данных. В нём были изменены константы NX, NY, NZ на значение 640. Константа NP на 10000000. Также константа FG_COUNT заменена на значение 32, так как программа будет запускаться на 32 процессах. Такие значения подобраны для того, чтобы программа работала продолжительное время до 1 минуты.

Входное представление будет создано скриптом для 32 процессов на 10 итераций. В тексте LuNA программы указывается директива «#define» со значением 32.

Планируется сконструировать два варианта программы с помощью генератора параллельных фрагментированных программ. Первый вариант с распределением по процессам на усмотрение алгоритма конструирования, которое может являться любым, так как этим занимается датчик любых чисел, а второй вариант — с эффективным распределением. Причем распределение фрагментов вычислений по процессам будет максимально похожим на распределение в LuNA-программе, которое там регулируется с помощью локаторов.

Конструирование 10 итераций для 32 процессов осуществлялось почти что 30 минут. В то время как 10 итераций для 4 процессов конструируется 6 секунд.

4.4.5 Результаты тестирования

Тестирование двух сгенерированных программ и одной LuNA-программы произведены в информационно-вычислительном центре Новосибирского государственного университета [26]. Программы запускались на 4 узлах, в каждом из которых использовалось по 8 процессов. Каждая программа запускалась по 10 раз. Лучшее время каждой из программ следующие:

- Сгенерированная программа с неэффективным распределением — 57.158 с.
- Сгенерированная программа с эффективным распределением — 38.854 с.
- LuNA-программа с использованием локаторов — 44.892 с.

В результате тестирования сгенерированная программа с эффективным распределением имеет лучший результат. Такая программа быстрее, чем LuNA-программа с использованием локаторов на 13,5% и быстрее сгенерированной программы с неэффективным распределением на 32%. Это косвенно подтверждает, что LuNA-программа тратит некоторые дополнительные ресурсы на исполнительную систему. Также можно утверждать, что разработка алгоритмов эффективного конструирования необходима для генератора, так как результат без применений таких алгоритмов хуже, чем в LuNA-программе.

4.5 Итоги тестирования

В ходе проведения тестирования все сгенерированные программы смогли осуществить компиляцию и выполниться при правильном формате входного представления. Фрагментированные программы успешно конструируются как для одного процесса, так и для нескольких. Все коммуникации между процессами конструируются корректно. В генераторе обеспечена многовариантность как между фрагментами вычисления, так и на уровне распределения фрагментов данных между процессами. Все разработанные конструкции были проверены в ходе тестирования и все они осуществляют корректную генерацию кода. Переиспользование кода из системы фрагментированного программирования LuNA обеспечивает корректную работу фрагментов кода. Предоставленная возможность во входном языке описания

назначать номер нужного процесса для каждого фрагмента вычисления способна обеспечить скорость исполнения программы быстрее чем его LuNA-программ аналогов.

ЗАКЛЮЧЕНИЕ

В результате работы была разработана подсистема прямого управления исполнением фрагментированных программ. Были рассмотрены существующие решения автоматического конструирования программ. Разработанные языковые средства позволяют описать входное представление фрагментированной программы. Разработанные алгоритмы и структуры данных, используемые генератором, осуществляют генерацию параллельной программы. В ходе работы были выполнены все поставленные задачи тем самым была достигнута цель работы.

Результаты работы были представлены на 61-й Международной научной студенческой конференции, г. Новосибирск, 2023 г. Тема доклада: «Разработка подсистемы прямого управления исполнением фрагментированных программ в системе LuNA».

Защищаемые положения:

- Разработаны языковые средства описания фрагментированной программы.
- Разработана структура данных и реализованы алгоритмы конструирования, обеспечивающие генерацию параллельного кода.
- Проведены экспериментальные исследования, в которых сравнивалась работоспособность и эффективность сгенерированных программ.

Разработанное средство автоматического конструирования может иметь следующие направления развития:

- Автоматическая генерация входного представления для генератора параллельных фрагментированных программ.
- Увеличение количества конструкций для расширения класса поддерживаемых задач.
- Реализация в генераторе алгоритмов автоматического улучшения эффективности сконструированной параллельной программы.

В рамках ВКР разрабатывалось программное решение. Описание программы представлено в приложении В, а руководство оператора в приложении Г.

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 недопуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Пирожков Андрей Константинович

ФИО студента

Подпись студента

« _____ » _____ 20 __ г.
(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Malyshkin, V.E., Perepelkin, V.A. (2011). LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem. In: Malyshkin, V. (eds) Parallel Computing Technologies. PaCT 2011
2. GCC, the GNU Compiler Collection: [Электронный ресурс]. URL: <https://gcc.gnu.org/> (дата обращения: 10.05.2023).
3. OpenMP. Справочные руководства: [Электронный ресурс]. URL: <https://www.openmp.org/resources/refguides/> (дата обращения: 10.05.2023)
4. MPI: The Message Passing Interface: [Электронный ресурс]. URL: https://parallel.ru/tech/tech_dev/mpi.html (дата обращения: 10.05.2023)
5. Chamberlain, B.L. Chapel chapter. In Programming Models for Parallel Computing, Balaji, P. (ed.) // MIT Press, November 2015.
6. S. M. Blackburn, M. K. Gardner, C. Hoffmann, A. M. Khan, J. S. McKinley, R. N. Miller, and R. L. Sites. X10: an object-oriented approach to non-uniform cluster computing // In Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, – San Diego, California, USA, October 2005. – С. 519–538,
7. P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su and K. Yelick. Titanium Language Reference Manual. // U.C. Berkeley Tech Report, UCB/EECS-2005-15. – Berkeley, California 2005. – 102 С.
8. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G. L., Tobin-Hochstadt, S. The Fortress Language Specification Version 1.0 β. Sun Microsystems, Inc. // – 2007. – 442 С.
9. The Charm++ Parallel Programming System. Официальная документация Charm++ [Электронный ресурс]. – URL: <https://charm.readthedocs.io/en/latest/charm%2B%2B/manual.html> (дата обращения: 10.05.2023).
10. S. Bak, H. Menon, S. White, M. Diener and L. Kale, «Multi-Level Load Balancing with an Integrated Runtime Approach,» // 18th IEEE/ACM

- International Symposium on Cluster, Cloud and Grid Computing (CCGRID). – Washington, 2018. – С. 31-40.
11. Johnson, S., Jin, H., Ierotheou, C. The ParaWise Expert Assistant – Widening Accessibility to efficient, Scalable Tool Generated OpenMP Code. // In Proceedings of WOMPAT. – 2004. – С. 67-82.
 12. Polaris Developer's Document: [Электронный ресурс] – URL: http://polaris.cs.uiuc.edu/polaris/polaris_developer/polaris_developer.html (дата обращения: 05.04.2023).
 13. Lee, S.I., Johnson, T.A., Eigenmann, R. (2004). Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation. // Rauchwerger, L. (eds) Languages and Compilers for Parallel Computing. – Springer, Berlin, Heidelberg, 2007. – С.539-553.
 14. Cetus Documentation: [Электронный ресурс] – URL: <https://engineering.purdue.edu/Cetus/Documentation/manual/manual.html> (дата обращения: 05.04.2023).
 15. Par4All 1.4.6 documentation: [Электронный ресурс] – URL: <http://par4all.github.io/features.html> (дата обращения: 06.04.2023).
 16. Jin, H., Frumkin, M., Yah, J. Code Parallelization with CAPO — A User Manual. // NASA Advanced Supercomputing (NAS) Division, M/S T27A-2, NASA Ames Research Center, Moffett Field, CA 94035-1000. – 2002.
 17. D. O'Neal, R. Luczak, M. White, Proceedings of the DoD High Performance Computing Modernization // Program Users Group Conference. – Albuquerque, New Mexico, 2000.
 18. Коммуникационный интерфейс PVM [Электронный ресурс]. – URL: http://rsuib.cc.rsu.ru/tutor/high_performance_computing/chapter1/page12.html (дата обращения: 11.05.2023).
 19. Штейнберг Б.Я., Нис З.Я., Петренко В.В., Черданцев Д.Н., Штейнберг Р.Б., Шульженко А.М. Открытая распараллеливающая система. // Третья международная конференция «Параллельные вычисления и задачи управления». – Москва, 2006. – С. 526-541.

20. Науменко С.А. Автоматическая генерация MPI кода в открытой распараллеливающей системе: маг. дис. // – Ростов-на-Дону, 2005. – 37 с.
Режим доступа: https://www.ops.rsu.ru/download/ops/MPI_in_OPS.pdf (дата обращения: 11.04.23).
21. Малышкин В. Э. Технология фрагментированного программирования // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. – №. 46 (305). – С. 45-55.
22. Описание языка LuNA: [Электронный ресурс] – URL: <https://gitlab.ssd.sccc.ru/luna/luna/-/wikis/Описание%20языка%20LuNA> (дата обращения: 10.05.2023).
23. Метод градиентного спуска: [Электронный ресурс] – URL: http://www.machinelearning.ru/wiki/index.php?title=Метод_градиентного_спуска (дата обращения: 24.04.2023).
24. Репозиторий разработанного генератора параллельных программ: [Электронный ресурс] – URL: https://gitlab.ssd.sccc.ru/Pirozhkov_Andrey/mpi-code-generator-with-using-luna (дата обращения: 28.05.2023).
25. Репозиторий LuNA: [Электронный ресурс] – URL: <https://gitlab.ssd.sccc.ru/luna/luna/-/tree/master/> (дата обращения: 28.05.2023).
26. Информационно-вычислительный центр Новосибирского государственного университета [Электронный ресурс]. URL: <http://nusc.nsu.ru/wiki/doku.php/doc/index> (дата обращения 23.05.2023)

ПРИЛОЖЕНИЕ А

Формат файла описания фрагментов вычислений

Все описанные ниже поля будем считать обязательными, без которых генератор параллельных программ не будет работать. Могут также встречаться необязательные значения для некоторых ключей. Для таких случаев будет написано, что они являются необязательными, а также будет указано, какое значение по умолчанию будет применено в таких случаях. Все ключи и все значения являются строками. Даже цифры записываются в формате строки, т.е. в кавычках.

Файл описания фрагментов вычислений должен иметь в себе следующие ключи:

- «using»;
- «interface»;
- «operations»;
- «variables»;
- «define».

Далее перечислено, какие значения надо заполнять для этих ключей.

Ключ «using».

В «using» значение представляет собой массив из названий файлов, которые являются описанием фрагментов кода. Рекомендуется указывать полный путь к файлу, а не только его название. Используемые файлы в качестве описания фрагментов кода должны быть в формате json.

Ключ «interface».

Значение для ключа «interface» представляет собой вложенную структуру с 3 ключам:

- «ranks»;
- «input»;
- «output».

Вложенный ключ «ranks» в значении ключа «interface».

В ключе «ranks» указывается количество процессов, на котором будет работать сгенерированная параллельная программа. Пара «ключ-значение» «ranks» является необязательной, и, если в файле не указать эту пару, то значение для этого ключа будет использоваться по умолчанию значение «1», т.е. конструироваться программа будет для 1 процесса. Это получится частный случай параллельной программы — последовательная программа.

Вложенный ключ «input» в значении ключа «interface».

Необходимо указать в значение ключа «input» переменные и фрагменты данных в формате массива, которые будут использоваться для инициализации. Важно учесть, что фрагменты данных не могут быть использованы для инициализации, если в программе нет инициализирующего снippets. Если в программе есть инициализирующий снippet, в поле «input» необходимо указать все переменные, которые используются в этом снippetе, включая фрагменты данных. Исключением являются переменные, переданные программе (int argc, char* argv[]), их перечислять в «input» не нужно.

Вложенный ключ «output» в значении ключа «interface».

В значение ключа «output» необходимо указать переменные и фрагменты данных в формате массива, используемые для финализации. Если финального снippets нет, для финализации следует использовать только переменные. Если же присутствует финализирующий снippet, в поле «output» следует указать все переменные и фрагменты данных, используемые в этом снippetе.

Ключ «operations».

По этому ключу в значениях перечисляются вложенные структуры с фрагментами вычислений. В качестве ключа используется название каждого используемого фрагмента вычислений, которое должно быть уникальным. В качестве значения для каждого названия фрагмента вычислений выступает структура, которая содержит следующие пары «ключей-значений»:

1. «function_type»;
2. «code»;
3. «rank»;

4. «inputs»;
5. «outputs»;
6. «arguments».

Вложенный ключ «function_type» в значении ключа «operations».

Значение по ключу «function_type» указывается тип сниплета:

7. «initial» - инициализирующий;
8. «final» - финальный;
9. «intermediate» - промежуточный.

Вложенный ключ «code» в значении ключа «operations».

Значение для ключа «code» указывается название фрагмента кода из файлов с описанием фрагментов кода, которые указывались в значениях по ключу «using».

Вложенный ключ «rank» в значении ключа «operations».

Пара «ключ-значение» «rank» является необязательной. Значение для этого ключа пишется, если пользователь хочет, чтобы этот фрагмент вычислений исполнялся на конкретном номере процесса.

Вложенный ключ «inputs» в значении ключа «operations».

Значение для ключа «inputs» записываются в формате массива, элементами которого являются переменные и фрагменты данных, которые потребляются, т.е. являются входными.

Вложенный ключ «outputs» в значении ключа «operations».

Значение для ключа «inputs» записываются в формате массива, элементами которого являются переменные и фрагменты данных, которые вырабатываются. Т.е. являются выходными.

Вложенный ключ «arguments» в значении ключа «operations».

Значением ключа «arguments» является вложенная структура, которая перечисляет несколько пар «ключ-значение». Она является связующим звеном между названиями переменных и фрагментов данных, используемых в фрагментах вычислений, и названиями переменных и фрагментов данных, используемых в фрагментах кода. В качестве ключа является название

переменной или фрагмента данных из фрагмента кода, а значением — переменная или фрагмент данных из фрагмента вычислений.

Ключ «variables».

В этом ключе в качестве значения используется вложенная структура пар «ключ-значение», с помощью которой перечисляется все используемые переменные и фрагменты данных, которые генерируются параллельной программой. В качестве ключа название переменной или фрагмента кода, в качестве значения тип.

Ключ «define».

Пара «ключ-значение» «define» является необязательной. В этом ключе в качестве значения используется структура пар «ключ-значение», где ключом является название константы, а значением — значение константы. Это преобразуется в директиву define в начале кода генерируемой параллельной программы.

ПРИЛОЖЕНИЕ Б

Формат файла описание с фрагментами кода

Файл формата фрагментов кода состоит из перечислений описания фрагментов кода, используемых в генераторе параллельных программ. Каждый такой описанный фрагмент кода должен иметь уникальное название. Для удобства можно использовать название фрагмента кода такое же, как и название используемой функции на C++ в файле с фрагментами кода используемой LuNA-программой. Заполнение файла описания фрагментов кода происходит в формате «ключ-значение», где ключ — это название описываемого фрагмента кода, а значение — это вложенные структуры, описывающие этот фрагмент кода. О том, какие вложенные структуры в значениях будет рассказано ниже.

Все описанные ниже пары «ключ-значение» будем считать обязательными, без которых генератор параллельных программ не будет работать. Могут также встречаться необязательные пары «ключ-значение», для них конкретно будет написано, что они необязательные и какие значения по умолчанию применяется в этих случаях. Все ключи и все значения являются строками. Даже цифры записываются в формате строки, т.е. в кавычках.

Описание каждого фрагмента кода состоит из перечисления структур со следующими ключами:

- 10.«type»;
- 11.«parameters»;
- 12.«source»;
- 13.«id».

Ключ «type».

По этому ключу вписывается пока что единственное значение — «c_function».

Ключ «parameters».

В сигнатуре функции на C++ перечисляются параметры, каждый из которых состоит из типа переменной и названия переменной. Значения по ключу «parameters» представляет собой массив из структур пар «ключ-значение»,

которые описывают параметры функции фрагмента кода. Параметры, которые указываются в поле «parameters» следующие:

- «name»;
- «descr»;
- «ctype»;
- «type»;
- «access».

Вложенный ключ «name» в значении ключа «parameters».

По этому ключу пишется название, описываемой переменной из сигнатуры функции.

Вложенный ключ «descr» в значении ключа «parameters».

Эта пара «ключ-значение» является необязательным. В него записывается описание к переменной. Это поле носит информационный характер, и оно нигде в генераторе параллельных программ не используется. В значение ключа «descr» можно написать строку с пояснением, например, что собой представляет переменная, которая используется в этом фрагменте кода.

Вложенный ключ «ctype» в значении ключа «parameters».

В этом ключе записывается значение — тип переменной в C++. Генератор параллельных программ поддерживает только три типа переменных: int, double, DF. Причем переменные указываются могут быть со «*», если это указатель, или с «&», если это передача по ссылке. Таким образом, если это переменная типа int, то она может быть записана как: «int», «int*», «int&».

Вложенный ключ «type» в значении ключа «parameters».

В этом ключе записывается значение типа переменной в LuNA. Генератор параллельных программ поддерживает только 3 типа переменных LuNA: «int», «real», «df». Если в значении по ключу «ctype» есть указатель или передача по ссылке, это никак не влияет на заполнение значения.

Вложенный ключ «access» в значении ключа «parameters».

Если данная переменная или фрагмент данных потребляется, т.е. используется в качестве входной переменной, то указывается значение «input».

Если переменная или фрагмент данных вырабатывается, т.е. используется в качестве выходной переменной, то используется значение «output».

Ключ «source».

В этом ключе в качестве значения указывается источник описываемого фрагмента кода. В данном случае пишется название файла. В LuNA такие файлы обычно называются «ucodes.cpp». Указывать рекомендуется только файл, без пути. И файл фрагментов кода должен быть помещён в ту же директорию, в которой будет сгенерирован код параллельной программы.

Ключ «id».

Этот ключ содержит реальное название объявленной функции фрагмента кода в файле, который был указан в значении ключа «source».

ПРИЛОЖЕНИЕ В

ГЕНЕРАТОР ПАРАЛЛЕЛЬНЫХ ФРАГМЕНТИРОВАННЫХ ПРОГРАММ

ОПИСАНИЕ ПРОГРАММЫ

Листов 12

Новосибирск 2023

СОДРЕЖАНИЕ

АННОТАЦИЯ.....	74
1 Общие сведения.....	75
2 Функциональное назначение	76
2.1 Назначение программы	76
2.2 Сведения о функциональных ограничениях на применение.....	76
3 Описание логической структуры.....	77
3.1 Назначение программы	77
3.2 Алгоритм программы.....	78
3.3 Связи между составными частями программы.....	78
3.4 Связи программы с другими программами.....	78
4 Используемые технические средства.....	79
5 Вызов и загрузка.....	80
6 Входные данные	81
7 Выходные данные	82
8 Лист регистрации изменений.....	83

АННОТАЦИЯ

В данном программном документе приведено описание генератора параллельных фрагментированных программ, предназначенной для генерирования параллельного кода на языке программирования C++.

Исходным языком программы является Python. Средство разработки — интегрированная среда разработки Visual Studio 2022 от компании Microsoft. Основными функциями генератора параллельных программ является генерирование параллельного кода с помощью входного языка описания фрагментированной программы. В результате работы программы генерируется параллельный код фрагментированной программы на языке C++.

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

Полное наименование программы: «Генератор параллельных фрагментированных программ».

Программное обеспечение необходимое для функционирования:

- Язык программирования Python.
- Текстовый редактор. Желательно поддерживающий формат json для удобной работы.
- Программный интерфейс для передачи информации MPI.

Языки программирования, на которых написан программа: Python.

2 Функциональное назначение

2.1 Назначение программы

Разработанная программа предназначена для генерации параллельного кода фрагментированных программ LuNA. Она обеспечивает генерации кода по предоставленному входному языку описания программы.

2.2 Сведения о функциональных ограничениях на применение

Программа поддерживает только текстовые файлы json. Необходимый формат файлов описан в приложении А и приложении Б. Также у генератора есть следующие функциональные ограничения:

- Генератор работает только с заранее определенным конечным количеством фрагментов кода и переменных с фрагментами данных.
- Генератор не имеет конструкций, которые бы поддерживали циклы. Циклы нужно самостоятельно раскручивать во входном представлении на входном языке.
- Генератор не имеет конструкций, позволяющие использовать условия. Все условия могут быть только внутри фрагментов кода.
- Генератор не конструирует выражения. Все выражения, осуществляющие математические операции, должны быть внутри фрагментов кода.

3 Описание логической структуры

Структура программы представляет собой из одного основного класса (main.py), 9 классов, описывающих структуру данных для конструирования (ArgvSnip.py, InvokeOperationSnip.py, mcell.py, partialprogram.py, port.py, prog_сpec.py, SendRecvSnip.py, Sniplet.py, VarToStdoutSnip.py), и 2 дополнительных вспомогательных класса (functions.py, variable.py).

3.1 Структура программы

Описание классов программы:

- main.py — основной файл, который содержит внутри себя алгоритмы, генерирующие код.
- Sniplet.py — файл, описывающий класс, по которому конструируются программы. От него наследуются 4 класса:
 - InvokeOperationSnip.py — файл с классом, описывающий конструкцию подстановки операции.
 - SendRecvSnip.py — файл с классом, описывающий конструкции с коммуникациями в генерирующейся программе.
 - ArgvSnip.py — файл с классом, описывающий конструкцию, которая генерирует код для считывания переменных из переданных аргументов.
 - VarToStdoutSnip.py — файл с классом, описывающим конструкцию, которая будет выводить в конце программы переменную в консоль.
- mcell.py — файл с классом, описывающий переменные и фрагменты данных, которые будут использоваться в генерируемой программе.
- port.py — файл с классом, который описывает параметры, которые передаются при вызове функций в генерируемой программе.
- prog_сpec.py — файл с классом, который хранит входное описание генерируемой программы и служит инструкцией для других классов.
- partialprogram.py — файл с классом, который содержит в себе все конструкции, из которых будет генерироваться параллельная программа.

3.2 Алгоритм программы

- Пользователь описывает входное представление фрагментированной программы, для которой он хочет сгенерировать параллельный код.
- Пользователь передаёт файл описаний фрагментов вычислений в качестве аргумента при запуске `main.py` класса с помощью утилиты `python3`.
- Пользователь ожидает завершения генерации программы.
- В консоль командной строки выводится код параллельной сгенерированной программы. Также этот код появляется в файле «`MPI_Test_code.cpp`».

3.3 Связи между составными частями программы

Связи между основной процедурой и функциями программы выполняются в виде стандартных вызовов подпрограмм.

3.4 Связи программы с другими программами

Генератор параллельных фрагментированных программ переиспользует часть кода из системы фрагментированного программирования LuNA. Устанавливать эту систему не нужно, все необходимые файлы уже содержатся в генераторе параллельных фрагментированных программ.

4 Используемые технические средства

Программа предназначена для использования на персональном компьютере или кластере. Режим работы – в формате терминала с командной строкой. Входные и выходные данные хранятся на жестком диске.

Устройство, на котором запускается программа, должно иметь одну из следующих операционных систем: Windows, Linux, Mac OS, а также иметь графический дисплей и средства ввода такие, как клавиатура и компьютерная мышь. Требования к параметрам устройства приведены в таблице В.1.

Таблица В.1 – Требования к параметрам устройства

Характеристика	Минимальные	Рекомендуемые
Архитектура процессора	x86	x86
Количество процессоров	1	1
Количество ядер	1	1
Частота процессора	1 ГГц	2,5 ГГц
Кэш процессора	2 МБ	4 МБ
Оперативная память	256 МБ	512 МБ
Доступное место в хранилище	50 МБ	100 МБ

5 Вызов и загрузка

Запуск генератор параллельных фрагментированных программ осуществляется следующим способом: с помощью интерфейса командной строки оператору необходимо выполнить следующие команды:

- Перейти в директорию, где расположен генератор параллельных фрагментированных программ (команды навигации по файловой системе зависят от выбранной операционной системы).
- Выполнить команду: `python3 main.py <название файла>`, где названием файла является файл описания фрагментов вычислений, который должен находиться в этой же директории с файлом `main.py`.

6 Входные данные

Генератору параллельных программ требуется для генерации параллельного фрагментированного кода как минимум два файла:

- Файл описания фрагментов вычислений — всегда один файл.
- Файл описания фрагментов кода — может быть более одного файла.

Все эти файлы должны находиться в одной и той же директории, что и исполняемый файл `main.py`.

Формат заполнения файлов описан отдельно в приложениях:

- Приложение А — формат файла описания фрагментов вычислений.
- Приложение Б — формат файла описания фрагментов кода.

7 Выходные данные

В качестве выходных данных используется является текст генерируемой параллельной фрагментированной программы, которые предоставляется двумя способами сразу:

- В терминале командной строки как вывод текста.
- В файл «MPI_code.cpp», который будет находится в директории с исполняемым файлом «main.py» в директории «MPI_test_code».

В обоих способов текст получаемой программы будет одинаковый.

8 Лист регистрации изменений

Таблица В.2 – Лист регистрации изменений в программном документе «Описание программы»

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц) в документе	№ документа	Входящий № сопроводительного документа	Подпись	Дата
Номер изм.	измененных	замененных	новых	аннулированных					

ПРИЛОЖЕНИЕ Г

**ГЕНЕРАТОР ПАРАЛЛЕЛЬНЫХ ФРАГМЕНТИРОВАННЫХ
ПРОГРАММ**

РУКОВОДСТВО ОПЕРАТОРА

Листов 8

Новосибирск 2023

СОДРЕЖАНИЕ

АННОТАЦИЯ.....	86
1 Назначение программы.....	87
1.1 Функциональное назначение программы.....	87
1.2 Эксплуатационное назначение программы.....	87
1.3 Состав функций.....	87
2 Условия выполнения программы.....	88
2.1 Минимальный состав аппаратных средств.....	88
2.2 Минимальный состав программных средств.....	88
2.3 Требования к оператору.....	88
3 Выполнение программы.....	89
3.1 Загрузка и запуск программы.....	89
3.2 Выполнение программы.....	89
3.3 Завершение работы программы.....	89
4 Сообщения оператору.....	90
5 Лист регистрации изменений.....	91

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению и эксплуатации генератора параллельных фрагментированных программ. В руководстве оператора описано функциональное и эксплуатационное предназначение этой программы, условия выполнения программы и описан процесс выполнения программы.

Оформление программного документа «Руководство оператора» произведено по требованиям ГОСТ 19.505-79 «ЕСПД. Руководство оператора» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

1 Назначение программы

1.1 Функциональное назначение программы

Генератора параллельных фрагментированных программ предназначен для генерирования кода параллельных фрагментированных программ на языке C++.

1.2 Эксплуатационное назначение программы

Программное обеспечение ориентировано на эксплуатацию пользователей системы фрагментированного программирования LuNA. Конечными пользователями программы должны быть физические лица, занимающиеся программированием задач численного моделирования

1.3 Состав функций

- Входной язык описания фрагментированной программы.
- Генерация параллельного фрагментированного кода на языке C++.

2 Условия выполнения программы

2.1 Минимальный состав аппаратных средств

Для работы генератора параллельных фрагментированных программ требуется персональный компьютер, включающий в себя:

- Процессор, совместимый с архитектурой x86 с тактовой частотой 1 ГГц или выше.
- Оперативную память объемом 1 Гб или выше.
- Доступное место на жестком диске объемом 50 Мб или выше.
- Монитор с разрешением экрана не менее 800 на 600 пикселей.
- Клавиатура.
- Мышь.

2.2 Минимальный состав программных средств

Системные программные средства для генератора параллельных фрагментированных программ требуются следующие:

- Любая из следующих операционных систем: Windows, Lunix, Mac OS.
- Python версии 3.0 и выше.
- Текстовый редактор. Желательно поддерживающий формат json для удобной работы.
- Программный интерфейс для передачи информации MPI.

2.3 Требования к оператору

Конечный оператор (пользователь) программы должен обладать практическими навыками работы с графическим пользовательским интерфейсом операционной системы. Пользователь должен обладать навыками использования системы программирования фрагментированных программ LuNA.

3 Выполнение программы

3.1 Загрузка и запуск программы

Запуск генератор параллельных фрагментированных программ осуществляется следующим способом: с помощью интерфейса командной строки оператору необходимо выполнить следующие команды:

- Перейти в директорию, где расположен генератор параллельных фрагментированных программ (команды навигации по файловой системе зависят от выбранной операционной системы).
- Выполнить команду: `python3 main.py <название файла>`, где названием файла является файл описания фрагментов вычислений, который должен находиться в этой же директории с файлом `main.py`.

3.2 Выполнение программы

Выполнение программы не предполагает никаких дополнительных действий со стороны оператора. В процессе выполнения программы пользователь ожидает получения результата.

3.3 Завершение работы программы

Программа завершает свою работу в командной строке выводом сгенерированного параллельного кода фрагментированной программы.

4 Сообщения оператору

В ходе работы генератора параллельных фрагментированных программ в качестве сообщения оператора может прийти только ошибка формата входного файла. Других сообщений оператору не предусмотрено.

5 Лист регистрации изменений

Таблица Г.1 – Лист регистрации изменений в программном документе
«Руководство оператора»

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц) в документе	№ документа	Входящий № сопроводительного документа	Подпись	Дата
Номер изм.	измененных	замененных	новых	аннулированных					