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**

**Брусникина Михаил Андреевича**

Тема работы:

# РАЗРАБОТКА СРЕДСТВ ПРЯМОГО УПРАВЛЕНИЯ ИСПОЛНЕНИЕМ ФРАГМЕНТИРОВАННЫХ ПРОГРАММ И ИХ РЕАЛИЗАЦИЯ В СИСТЕМЕ LUNA

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

**Руководитель ВКР**
к.т.н.
доц. каф. ПВ ФИТ НГУ
Власенко А. Ю. /……...
(ФИО) / (подпись)
«20» мая 2023 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Перепелкин 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
(название кафедры)

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)

«7» ноября 2022 г.

**ЗАДАНИЕ**
**НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

Студент*у(ке)* Брусникину Михаилу Андреевичу, группы 19209
(фамилия, имя, отчество, номер группы)

Тема: Разработка средств прямого управления исполнением фрагментированных программ и их реализация в системе LuNA
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 7 ноября 2022 г. №0330,

Срок сдачи студентом готовой работы «20» мая 2023 г.

Исходные данные (или цель работы):

Разработка и реализация программных средств прямого управления исполнением LuNA-программ.

Структурные части работы: обзор родственных работ управления исполнением программ, формальная постановка задачи прямого управления исполнением программ, определение требований и описание предлагаемого решения, описание разработанных программных средств, описание реализации программных средств, тестирование разработанных программных средств и описание полученных результатов.

Руководитель ВКР

Власенко А. Ю. / ……..
*(ФИО) / (подпись)*

«7» ноября 2022 г.

Задание принял к исполнению

Брусникин М.А. /……..
*(ФИО студента) / (подпись)*

«7» ноября 2022 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ

Перепелкин В. А. /……..
(ФИО) / (подпись)

«7» ноября 2022 г.

# СОДЕРЖАНИЕ

**ВВЕДЕНИЕ** **5**

**Глава 1 Обзор существующих решений** **7**

1.1 Legion 7

1.2 PaRSEC 8

1.3 DPLASMA 10

1.4 Charm++ 11

1.5 Вывод 12

**Глава 2 Разработка программных средств прямого управления исполнением фрагментированных в системе LuNA** **13**

2.1 Технология фрагментированного программирования 13

2.2 Система LuNA 14

2.3 Обзор структуры LuNA-программы 15

2.4 Проблема производительности системы LuNA 16

2.5 Понятие свертки и прямого управления исполнением программ 17

2.6 Идея предлагаемого решения 17

2.7 Язык свертки 18

2.8 Промежуточное состояние свертки 21

2.9 Алгоритм генерации MPI-программы 21

2.10 Требования к языку свертки 21

2.11 Характеристика решения 21

**Глава 3 Реализация программных средств прямого управления исполнением фрагментированных в системе LuNA** **23**

3.1 Схема работы 23

3.2 Реализация модуля парсинга свертки и генерации ее промежуточного состояния 24

3.3 Реализация модуля генерации MPI-программы 24

3.3.1 Метаинформация LuNA-программы 26

3.3.2 Процесс генерации MPI-программы 27

3.3.2.1 Анализ метаинформации LuNA-программы 27

3.3.2.2 Генерация исходного кода MPI-программы 29

3.3.2.3 Компиляция сгенерированной MPI-программы в исполняемый файл 31
3.4 Запуск скомпилированной MPI-программы. 32
3.5 Ограничения программных средств 32
3.6 Тестирование производительности 32
**ЗАКЛЮЧЕНИЕ 36**
**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ 38**
**ПРИЛОЖЕНИЕ А 40**
**ПРИЛОЖЕНИЕ Б 42**
**ПРИЛОЖЕНИЕ В 44**
**ПРИЛОЖЕНИЕ Г 46**
**ПРИЛОЖЕНИЕ Д 55**

# ВВЕДЕНИЕ

В современном мире существует множество задач в различных сферах, требующих больших компьютерных вычислений, например, задачи математического моделирования. С каждым годом математические модели усложняются, что в свою очередь усложняет разработку программ для таких моделей и увеличивает их время исполнения.

Чтобы сократить время на вычисления, используют параллельное программирование. Но разрабатывать параллельные программы гораздо сложнее последовательных. Поэтому есть необходимость в поддержке процесса параллельного программирования. Для этого создают высокоуровневые языки и системы программирования [1]. Однако, у них есть существенный недостаток — они не всегда обеспечивают удовлетворительную эффективность исполнения программы.

Чтобы достичь удовлетворительной эффективности, предлагается предоставить программисту возможность напрямую управлять распределением ресурсов вычислительной системы и описывать детали исполнения программы на высокоуровневом языке. Таким образом, можно достичь необходимой эффективности исполнения.

Прямое управление исполнением параллельных программ подразумевает возможность вручную задать распределение ресурсов вычислительной системы узлы и порядок исполнения программы.

Универсальных решений управления исполнением программ нет — это открытая научная проблема. На данный момент разрабатывается LuNA [2] — система автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах, реализующая идею фрагментированного программирования [3]. Предлагается создать программное средство, частично реализующее вышеизложенную идею прямого управления исполнением параллельных программ, на примере системы LuNA.

Целью работы является разработка и реализация средств прямого управления исполнением фрагментированных программ на высокоуровневом языке в системе LuNA.

В рамках ВКР были поставлены следующие задачи:

- Провести анализ требований к программным средствам.
- Определить функциональность средств.
- Разработать средства.
- Реализовать средства.
- Тестирование реализованных средств.

К разрабатываемым средствам предъявляются следующие требования:

- Управление исполнением программы описывается на высокоуровневом языке.
- С помощью средств можно явно задать распределение ресурсов вычислительной системы и определить порядок исполнения программы.

В данной работе были разработаны и реализованы программные средства прямого управления исполнением фрагментированных программ в системе LuNA.

Практическая ценность работы обусловлена тем, что использование разработанных программных средств может существенно повысить эффективность исполнения программ для системы LuNA.

Работа изложена в трех главах. В первой главе приводится обзор существующих средств и методов управления распределением ресурсов вычислительной системы. Во второй главе проведен анализ предметной области, формулирование требований к разрабатываемым программным средствам и разработанное решение. В третьей главе описывается реализация разработанного решения, его тестирование и оценка производительности.

## Глава 1 Обзор существующих решений

Во введении была поставлена проблема эффективного исполнения программ в высокоуровневых языках и системах программирования. В связи с этим был проведен обзор существующих средств и методов управления поведением программ. Целью обзора является исследование на предмет выразительности и универсальности существующих решений в различных языках и системах программирования.

Средства распределения ресурсом вычислительной системы для системы LuNA должны предоставлять возможность назначать исполнение задач на узлы, обеспечивать обмен данными между узлами, не иметь лишних накладных расходов, например, на runtime-систему, учитывать специфику системы LuNA и иметь простой и лаконичный интерфейс, чтобы программисту не требовалось профильных знаний для их использования. По этим критериям будет происходить анализ существующих решений.

### 1.1 Legion

Система Legion — это C++ фреймворк для параллельных вычислений на кластерах, он предоставляет мощный набор инструментов для управления распределенными вычислениями, включая возможность создания и управления задачами, доступ к памяти и синхронизацию между потоками выполнения [4]. Одной из ключевых особенностей является его способность работать с большим количеством процессоров и узлов кластера, что позволяет динамически распределять задачи между ними для ускорения вычислений. Фреймворк также обеспечивает возможность эффективного использования памяти и оптимизации производительности.

Интерфейс Legion Mapper — это набор функций и классов, предоставляемых фреймворком Legion для управления распределением задач на кластере [5]. Он включает в себя набор классов и функций, которые позволяют программистам управлять процессом распределения задач на кластере. Он позволяет определять, какие задачи должны быть выполнены на каких узлах

кластера, какие данные должны быть переданы между узлами и какие ресурсы должны быть выделены для каждой задачи.

Данный интерфейс обеспечивает возможность оптимизации производительности приложения, позволяя разработчикам определять, какие части приложения должны быть выполнены параллельно и какие ресурсы должны быть выделены для каждой из них. С его помощью разработчики могут создать собственный Mapper, который может быть оптимизирован для конкретных приложений и аппаратных конфигураций.

Mapper (маппер) в системе Legion — это компонент, который отвечает за преобразование вычислительных задач, описанных в терминах логических регионов, в конкретные вычисления. Использование своего маппера позволяет повысить производительности приложения и эффективно использовать ресурсы кластера.

Данное решение отвечает критериям возможности назначения исполнения задач на узлы, хранение и обмен данными между узлами, однако, имеет низкий уровень абстракции и не исключает накладные ресурсы runtime-системы.

### 1.2 PaRSEC

PaRSEC — это фреймворк для параллельных вычислений на кластерах. В нем используется модель потока задач (task-flow model), которая позволяет описывать параллельные приложения в терминах потоков задач, которые автоматически распределяются между узлами [6]. Программы на PaRSEC представлены в виде прямого бесконтурного графа задач с помеченными ребрами, обозначающими зависимости данных.

Основным компонентом PaRSEC является динамический планировщик задач, который автоматически распределяет задачи между узлами. Планировщик использует информацию о зависимостях между задачами, чтобы оптимизировать порядок их выполнения и минимизировать время ожидания. Это позволяет достичь высокой степени параллелизма и ускорить выполнение приложений.

PaRSEC также имеет механизмы для управления памятью и передачей данных между задачами. Он использует специальные буферы для хранения данных, которые могут быть распределены между узлами в зависимости от их характеристик и доступности. Это позволяет оптимизировать использование ресурсов и ускорить выполнение приложений.

Кроме того, он имеет ряд инструментов для анализа и оптимизации производительности приложений. Они позволяют собирать статистику выполнения задач и анализировать ее для выявления узких мест и возможных улучшений. Также он предоставляет возможность настройки параметров системы и алгоритмов планирования для достижения максимальной производительности.

В PaRSEC распределение ресурсов вычислительной системы задается через конфигурационный файл, который определяет количество процессоров, используемых для выполнения задач, и их связи между собой. Этот файл может быть настроен вручную или с помощью специальных инструментов, таких как PaRSEC Configuration Manager. Файлы конфигурации имеют формат INI и содержат информацию о количестве вычислительных узлов, доступных ресурсах на каждом узле, а также описания задач, которые должны быть выполнены на этих узлах. В файле конфигурации также можно задать параметры, определяющие количество потоков и размер буферов для каждой задачи, а также определить, какие ресурсы доступны на каждом узле, например, количество ядер процессора, доступную память и т.д.

Для более точного задания распределения ресурсов в PaRSEC можно использовать специальные аннотации, которые позволяют указать, какие задачи должны выполняться на каких процессорах. Например, аннотация "rank" позволяет указать номер процессора, на котором должна выполняться задача.

Данное решение отвечает критериям возможности назначения исполнения задач на узлы, хранение и обмен данными между узлами, однако, не исключает накладные ресурсы runtime-системы.

### 1.3 DPLASMA

DPLASMA (Distributed Parallel Linear Algebra Software for Multicore Architectures) — это библиотека подпрограмм линейной алгебры для распределенных систем [7]. Она предназначена для обеспечения стабильной производительности распределенных систем, где каждый узел оснащен несколькими сокетами многоядерных процессоров и, если доступно, ускорителями, использующими среду выполнения PaRSEC в качестве серверной части.

Одним из ключевых компонентов DPLASMA является параллельный алгоритм LU-разложения с частичным выбором элементов (Partial Pivoting LU). Этот алгоритм позволяет эффективно решать системы линейных уравнений и находить обратные матрицы. Он быстро адаптируется к различным архитектурам и может использоваться на многоядерных процессорах, GPU и кластерах.

Распределение ресурсов вычислительной системы задается с помощью специальных параметров, которые передаются при запуске приложения. В частности, приложение может принимать следующие аргументы командной строки:

- -p — количество процессов, которые будут использоваться для выполнения приложения;
- -n — размер матрицы, которая будет использоваться в приложении;
- -b — размер блока, на который матрица будет разбита для распределения по узлам;
- -i — количество итераций, которые будут выполнены в приложении;
- -h — отображение справки о доступных параметрах запуска приложения.

При запуске приложения с указанием этих параметров, DPLASMA автоматически распределяет задачи между процессами и узлами вычислительной системы, используя блочную схему распределения данных.

Так как данное PLASMA использует PaRSEC в качестве серверной части, то обладает теми же преимуществами и недостатками.

### 1.4 Charm++

Charm++ — параллельный объектно-ориентированный язык программирования в основе которого используется язык C++ [8]. Программы на языке Charm++ разбиваются на несколько взаимодействующих посредством сообщений объектов, которые называются чарами (chare). Когда программа вызывает метод объекта, система исполнения Charm++ посылает сообщение вызванному объекту, который может обрабатываться на узле при параллельных вычислениях. Это сообщение вызывает исполнение метода внутри чара для асинхронной обработки сообщения. Чары могут быть объединены в массивы чаров (chare array) и сообщения могут быть посланы индивидуальным чарам из массива и всему массиву чаров одновременно.

В Charm++ распределение ресурсов вычислительной системы происходит автоматически во время выполнения программы. Распределение происходит таким образом, чтобы минимизировать задержки при обмене данными между объектами и узлами системы. Это достигается благодаря особенностям объектной модели, на которой основан он основан. В этом языке каждый объект представляет отдельную единицу параллельной работы, и приложение состоит из множества объектов, которые взаимодействуют друг с другом.

Для того чтобы задать параметры распределения ресурсов в Charm++, можно использовать специальные директивы в исходном коде программы. Например, директива charmrun позволяет задать параметры запуска программы, такие как количество процессов и узлов вычислительной системы.

Данное решение отвечает критериям возможности назначения исполнения задач на узлы, хранение и обмен данными между узлами, однако, не исключает накладные ресурсы runtime-системы.

### 1.5 Вывод

Из обзора существующих языков и систем программирования видно, что существует множество различных методов и подходов к вопросу распределения ресурсов вычислительной системы, обладающих своими преимуществами и недостатками.

Все рассмотренные решения не соответствуют в полной мере требуемым критериям и не учитывают специфику системы LuNA, что не позволяет в применить одно из рассмотренных решений. В связи с этим разработка средств прямого управления исполнением программ, ориентированные на систему LuNA и соответствующие требуемым критериям, является актуальной.

## Глава 2 Разработка программных средств прямого управления исполнением фрагментированных в системе LuNA

### 2.1 Технология фрагментированного программирования

Основная проблема при разработке эффективных параллельных программ для больших численных моделей заключается в достаточно большой сложности их программирования, что в свою очередь увеличивает время разработки и усложняет отладку программ.

Для решения этой проблемы требуется поддержка процесса параллельного программирования для таких моделей. Необходимая технология – технология фрагментированного программирования (ТФП) и реализующие ее язык и система LuNA разрабатываются в ИВМиМГ СО РАН.

Суть данной технологии заключается в представлении прикладного алгоритма в виде не более чем счетного множества фрагментов вычислений (ФВ), выполняющих роль вычислительных задач в ТФП. Фрагмент вычислений — независимая единица программы, содержащая описание входных/выходных переменных, соответственно, называемые фрагментами данных (ФД), и фрагмента кода, операцией на входными ФД, которая представляет собой процедуру или модуль (рисунок 1).

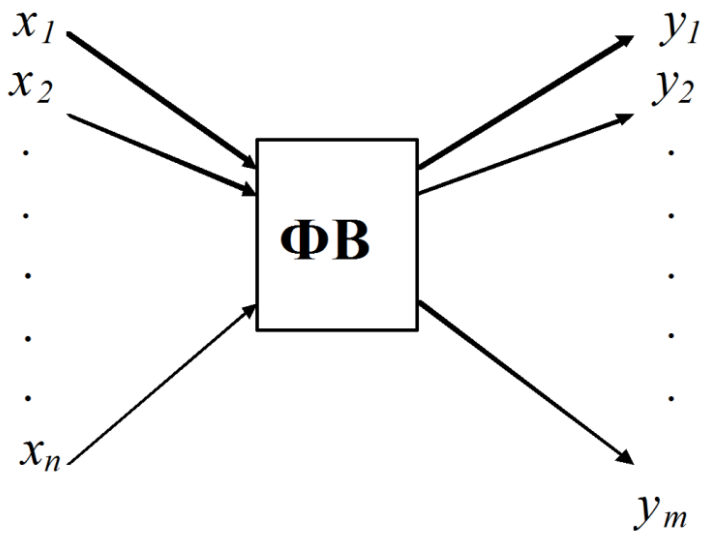


Рисунок 1 — Фрагмент вычислений

Фрагментированный алгоритм (ФА) – это описание рекурсивно перечислимое множество ФВ и их входных/выходных ФД. Таким образом, ФА определяется как множества ФД и ФВ. В процессе исполнения фрагментированная структура сохраняется.

Базовый алгоритм исполнения фрагментированной программы заключается в следующих правилах:

- ФВ исполняется, если все его входные ФД получили значения.
- После исполнения ФВ всего его выходные ФД получают значения.
- Алгоритм может реализовываться в сгенерированной программе, либо runtime-системой. В LuNA выбрано исполнение runtime-системой в целях обеспечения динамических свойств.

Вышеуказанные правила определяют асинхронное исполнение ФП с архитектурой dataflow, иными словами, по информационным зависимостям. В процессе исполнения runtime-система старается эффективнее исполнять алгоритм путем балансировки задач и данных между узлами.

### 2.2 Система LuNA

Экспериментальная система автоматического конструирования параллельных программ на мультикомпьютерах LuNA (Language for Numerical Algorithms) была разработана с поддержкой ТФП. Язык программирования, входящий в эту систему, позволяет описывать ФВ, ФД и ФК.

ФК может атомарным и структурированным. Атомарный ФК — это процедура, описанная на языке C++ и использующая программный интерфейс для работы с ФД, предоставляемой системой LuNA. Структурированный ФК — это процедура на языке LuNA [9]. Система транслирует программу во внутреннее представление, которое выполняется исполнительной системой.

Порядок исполнения ФВ не соответствует порядку их описания, т.к. согласно алгоритму исполнения фрагментированной программы ФВ вычисляются по мере их готовности по информационным зависимостям. Таким

образом, программист определяет какие вычисления должны быть произведены и над какими данными, а не их порядок выполнения.

### 2.3 Обзор структуры LuNA-программы

Рассмотрим структуру LuNA-программы на простом примере. На рисунке 2 изображен исходный код LuNA-программы инициализации и вывода значения ФД, а также вывода приветственного сообщения.

```
import c_init(int, name) as init;
import c_iprint(int) as iprint;
import c_sayHello() as sayHello();

sub sayHelloOnce() {
    sayHello();
}

sub main()
{
    df x;
    cf initX: init(7, x[4]);
    iprint(x);
    sayHelloOnce();
}
```

Рисунок 2 — Простая LuNA-программа

Атомарные ФК объявляются конструкцией import, в которой указываются названия параметров и их типы. Они описываются в виде процедур на языке C++. Пример описания атомарных ФК изображен на рисунке 3.

```
#include <cstdio>
#include "ucenv.h"

extern "C" {

void c_init(int val, OutputDF &df) {
      df.setValue(val);
      printf("c_init: %d, size: %d\n", val, (int)df.getSize());
}

void c_iprint(int val) {
      printf("c_iprint %d\n", val);
}

void c_sayHello() {
    printf("Hello world!\n");
}

}
```

Рисунок 3 — Описание атомарных фрагментов кода

Структурированные ФК объявляются конструкцией sub, внутри тела конструкции описываются объявления ФД конструкцией df и применения ФВ. Структурированный ФК с именем main является отправной точкой исполнения программы.

ФВ могут иметь имена. Область видимости имен ФД и ФВ ограничивается телом структурированного ФК, в котором они используются.

### 2.4 Проблема производительности системы LuNA

Исходя из тестов [10], на текущий момент существует большая разность эффективности исполнения LuNA в сравнении с аналогичной MPI-программой. В идеальном случае LuNA может обеспечить лишь приблизительно эквивалентную эффективность исполнения с ее аналогом на MPI — это связано с тем, что LuNA имеет различные накладные расходы. Одним из источников таких расходов является runtime-система, задачей которой является балансировка нагрузки между узлами, стратегия которой не всегда оказывается

правильной и может наоборот привести к ухудшению производительности. Также runtime-система реализована в виде интерпретатора, который будет всегда медленнее нативного кода.

### 2.5 Понятие свертки и прямого управления исполнением программ

Прямое управление исполнением программ — это отображение ФД и ФВ на узлы вычислительной системы. Конкретно заданное отображение на узлы называется сверткой (рисунок 4).

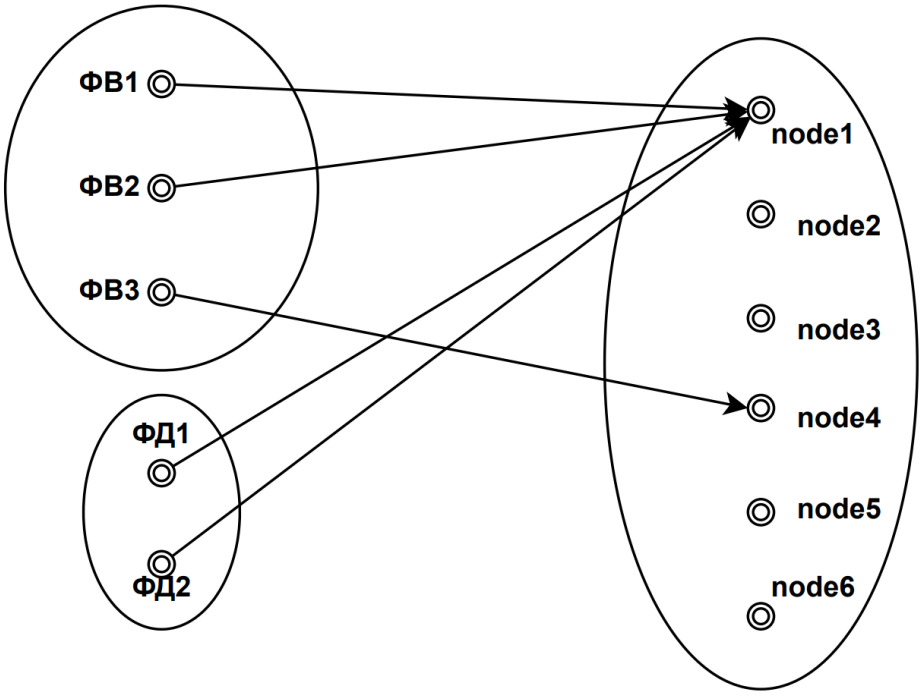


Рисунок 4 — Пример свертки

Runtime-система LuNA в процессе исполнения программы динамически определяет отображение ФВ и ФД на узлы и пересылает ФД их между узлами по мере необходимости, что в свою очередь тоже является неявной сверткой.

### 2.6 Идея предлагаемого решения

Суть идеи заключается в следующем: так как LuNA основана на MPI [11], то для LuNA-программы для нее можно описать свертку на высокоуровневом языке, а после сгенерировать аналогичную MPI-программу, которая будет иметь отображение ФВ и ФД на узлы согласно описанной свертке. Сгенерированную программу можно скомпилировать в исполняемый файл и исполнить.

Для этого необходимо разработать язык свертки и алгоритм генерации MPI-программы.

### 2.7 Язык свертки

Грамматика языка свертки определяет структуру, изображенную на рисунке 5.

```
СВЕРТКА:
    БЛОК_МАКРООПРЕДЕЛЕНИЙ:
        СПИСОК(ОПЕРАТОР_ОБЪЯВЛЕНИЯ_МАКРОПЕРЕМЕННОЙ)
    БЛОК_ИСПОЛНЕНИЯ:
        СПИСОК(ОПЕРАТОР_УПРАВЛЕНИЯ)
```

Рисунок 5 — Структура языка свертки

В блоке макроопределений содержатся применения оператора объявления числовой макроконстанты, которые можно использовать в блоке описания исполнения. Для применения значения макроконстанты в блоке описания исполнения необходимо использовать соответствующий синтаксис (рисунок 6). На рисунке 7 представлен синтаксис данного оператора.

```
$ИМЯ_МАКРОПЕРЕМЕННОЙ
```

Рисунок 6 — Синтаксис применения значения макропеременной

```
ИМЯ_КОНСТАНТЫ = ЦЕЛОЧИСЛЕННОЕ_ЗНАЧЕНИЕ;
```

Рисунок 7 — Синтаксис оператора объявления макро константы

В блоке описания исполнения программы содержатся применения операторов управления. Порядок описания операторов определяет порядок исполнения программы. В языке определены следующие операторы управления:

1. Оператор управления define df. Данный оператор объявляет ФД с заданным именем. Синтаксис данного оператора представлен на рисунке 8.

2. Оператор управления run. Данный оператор предписывает выполнение ФВ с заданным именем на определенном узле. Синтаксис данного оператора представлен на рисунке 9.
3. Оператор управления send. Данный оператор выполняет пересылку ФД между заданными узлами. Синтаксис данного оператора представлен на рисунке 10.
4. Оператор управления for. Данный оператор определяет множество применений операторов управления. В его теле описывается применение операторов управления send, run и for в контексте цикла с заданным итератором и его начальным и конечным значением. Каждая итерация цикла генерирует подмножество применения операторов управления согласно их порядку описания в теле. Синтаксис данного оператора представлен на рисунке 11.

```
define df ИМЯ_ФД;
```

Рисунок 8 — Синтаксис оператора управления define df

```
send ИМЯ_ФД from НОМЕР_УЗЛА_ОТПР to НОМЕР_УЗЛА_ПОЛУЧ;
```

Рисунок 9 — Синтаксис оператора управления send

```
run ИМЯ_ФВ on НОМЕР_УЗЛА_ИСПОЛ;
```

Рисунок 10 — Синтаксис оператора управления run

```
for ИМЯ_ИТЕРАТОРА = НАЧ_ЗНАЧ_ИТЕРАТОРА .. КОН_ЗНАЧ_ИТЕРАТОРА {
    ОПЕРАТОР_УПРАВЛЕНИЯ_1;
    ОПЕРАТОР_УПРАВЛЕНИЯ_2;
    …

}
```

Рисунок 11 — Синтаксис оператора управления for

Структуры операторов управления изображены на рисунках 12-15.

```
ОПЕРАТОР_УПРАВЛЕНИЯ_DEFINE_DF:
    ИМЯ_ФД
```

Рисунок 12 — Структура оператора define df

```
ОПЕРАТОР_УПРАВЛЕНИЯ_SEND:
    ИМЯ_ФД
    НОМЕР_УЗЛА_ОТПРАВИТЕЛЯ
    НОМЕР_УЗЛА_ПОЛУЧАТЕЛЯ
```

Рисунок 13 — Структура оператора send

```
ОПЕРАТОР_УПРАВЛЕНИЯ_RUN:
    ИМЯ_ФВ
    НОМЕР_УЗЛА_ИСПОЛНИТЕЛЯ
```

Рисунок 14 — Структура оператора run

```
ОПЕРАТОР_УПРАВЛЕНИЯ_FOR:
    ИМЯ_ИТЕРАТОРА
    НАЧАЛЬНОЕ_ЗНАЧЕНИЕ_ИТЕРАТОРА
    КОНЕЧНОЕ_ЗНАЧЕНИЕ_ИТЕРАТОРА
    СПИСОК(ОПЕРАТОР_УПРАВЛЕНИЯ)
```

Рисунок 15 — Структура оператора for

Пример описанной свертки изображен на рисунке 16.

```
define {
    N = 4;
}

execution {
    define df x;
    run initX on rank 0;
    send x from 0 to 1;
    for i=1..$N {
        run printValue on rank 1;
    }
}
```

Рисунок 16 — Пример описанной свертки

### 2.8 Промежуточное состояние свертки

Промежуточное состояние свертки – это JSON-файл [12], представляющий собой свертку в структурированном виде. Промежуточное состояние является более удобным вариантом для обработки информации в алгоритме генерации MPI-программы. Промежуточное состояние в JSON-формате повторяет структуру свертки (см. Приложение А).

### 2.9 Алгоритм генерации MPI-программы

На вход алгоритма поступает LuNA-программа, атомарные фрагменты кода и свертка. На выходе алгоритма скомпилированная MPI-программа (см. Приложение Б).

### 2.10 Требования к языку свертки

Для языка описания сверток выдвигаются следующие требования:

- Свертка описывается на высокоуровневом языке.
- Язык свертки содержит базовый набор операторов управления исполнением, достаточный для описания исполнения программы на LuNA. Операторы должны поддерживаться существующими алгоритмами.
- Свертка конструктивно и явно задает порядок исполнения фрагментов вычислений.

### 2.11 Характеристика решения

Данное решение позволит генерировать более эффективные MPI-программы на основе LuNA-программ и описанной для нее свертки. Основным плюсом такого решения является простота использования — программисту необходимо всего лишь описать свертку для готовой LuNA-программы.

Архитектура решения состоит из двух модулей — модуля парсинга свертки и генерации ее промежуточного состояния и модуля генерации MPI-программы. Второй модуль использует первый для получения

промежуточного состояния свертки. Модульность позволяет легко изменить реализацию конкретного модуля.

Моделями в данном решении является программа на языке LuNA, файл с описанными атомарными фрагментами кода на языке C++ [13] и файл с описанной сверткой.

Решение использует один алгоритм — алгоритм генерации промежуточного состояния свертки и алгоритм генерации MPI-программы.

## Глава 3 Реализация программных средств прямого управления исполнением фрагментированных в системе LuNA

### 3.1 Схема работы

Работа разработанных средств схематически представлена на рисунках 17-19.

Рисунок 17 — Схема парсинга файла свертки в промежуточное состояние

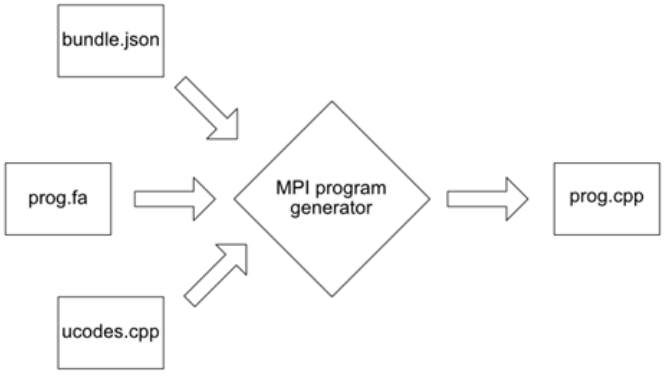


Рисунок 18 — Схема генерации MPI-программы

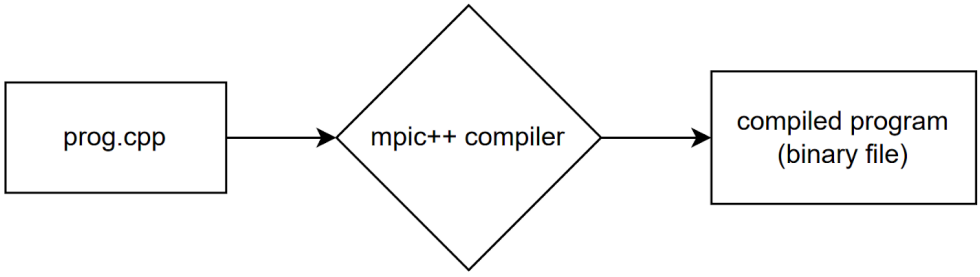


Рисунок 19 — Схема компиляции сгенерированной MPI-программы

Компиляция LuNA-программы на основе свертки происходит в три этапа:

1. Модуль парсинга свертки обрабатывает файл с описанной сверткой и создает json-файл с ее промежуточным состоянием.
2. Модуль генерации MPI-программы на основе полученного промежуточного состояния и LuNA-программы генерирует файл с исходным кодом MPI-программы.
3. Модуль генерации MPI-программы компилирует сгенерированный файл с помощью компилятора mpic++ в исполняемый файл.

### 3.2 Реализация модуля парсинга свертки и генерации ее промежуточного состояния

Рассмотрим подробнее реализацию модуля парсинга свертки и генерации промежуточного состояния. Как и в системе LuNA, здесь используется лексический анализатор Flex [14] и синтаксический анализатор Bison [15]. С помощью данных утилит исходный код на языке свертки преобразуется в удобочитаемый JSON формат. Данный модуль выполнен в виде утилиты командной строки.

Данная утилита имеет название luna-bundle-parser и принимает на вход два аргумента — файл с описанной сверткой формата .bndl и имя выходного JSON-файла. Пример использования представлен на рисунке 20.

```
$ luna-bundle-parser input.bndl bundle.json
```

Рисунок 20 — Пример использования модуля парсинга свертки и генерации промежуточного состояния

### 3.3 Реализация модуля генерации MPI-программы

Рассмотрим подробнее реализацию модуля генерации MPI-программы. Данный модуль был написан на языке Python [16], т.к. данный язык имеет удобные инструменты для парсинга JSON-файлов и генерации текстовых файлов. С его помощью LuNA-программа на основе полученного промежуточного состояния свертки генерирует исходный код MPI-программы и

компилирует в исполняемый файл. Данный модуль выполнен в виде утилиты командной строки.

Данная утилита имеет название luna-mpi-generator и имеет два способа передачи аргументов.

Первый способ заключается в использовании отдельных опций для передачи требуемых файлов. Для генерации MPI-программы требуются следующие файлы: файл с описанной сверткой формата .bndl, файл с описанными фрагментами кода для LuNA-программы ucodes.cpp и файл с исходным кодом LuNA-программы формата .fa. Для их передачи в качестве аргументов утилите определены опции --bundle-file, --cpp-codes и --luna-src соответственно. Пример использования утилиты представлен на рисунке 21.

```
$ luna-mpi-generator --bundle-file input.bndl --cpp-codes
ucodes.cpp --luna-src prog.fa
```

Рисунок 21 — Пример использования модуль генерации MPI-программы

Второй способ более лаконичный — можно описать требуемые файлы в конфигурационном JSON-файле и передать утилите через опцию --config-file. Данный конфигурационный файл содержит описания всех требуемых файлов для генерации MPI-программы, а также еще два параметра — MPI-header и имя выходного исполняемого файла. Данный параметры можно задать и в первом способе при помощи опций --mpi-header и -o/--output соответственно. Для опций --mpi-header и -o/--output определены значения по умолчанию — “mpi.h” и “prog” соответственно. Пример конфигурационного JSON-файла представлен на рисунке 22.

```
{
    "bundle_json_path": "bundle.json",
    "luna_src_path": "prog.fa",
    "cpp_codes_path": "ucodes.cpp",
    "mpi_header": "mpi.h"
}
```

Рисунок 22 — Пример конфигурационного файла

### 3.3.1 Метаинформация LuNA-программы

Компилятор LuNA имеет опцию --compile-only, которая сообщает компилятору о необходимости только скомпилировать LuNA-программу без ее исполнения. Пример компиляции LuNA-программы с данной опцией: luna --compile-only --build-dir=build prog.fa. Результатом данной команды будет скомпилированная LuNA-программа вместе с её различными дополнительными файлами при сборке в директории build. Содержимое данной папки представлено на рисунке 23. Опция --build-dir определяет папку, в которой будет хранится результат компиляции программы.

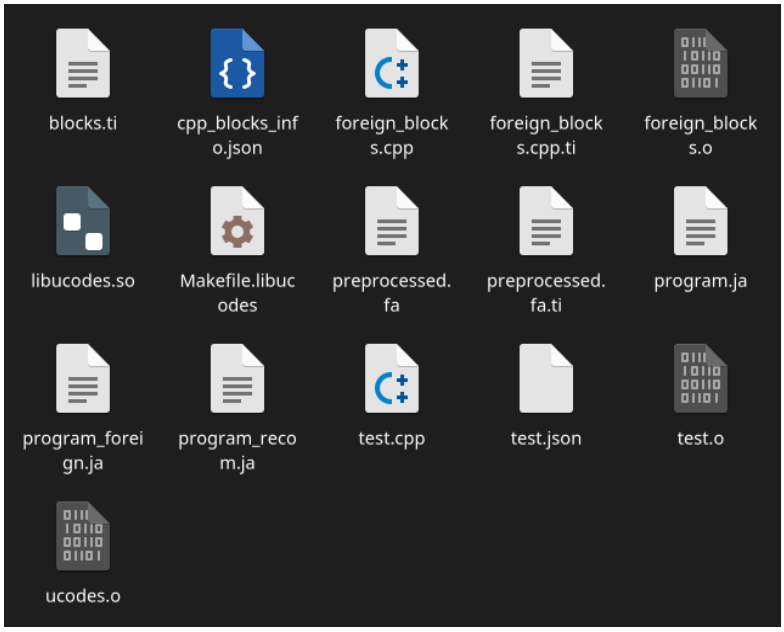


Рисунок 23 — Результат компиляции LuNA-программы с опцией --compile-only

Среди данных файлов есть файл с program_recom.ja, который содержит подробную информацию о LuNA-программе. Проанализировав данный файл, можно извлечь информацию о всех фрагментах вычислений, связанных с ними фрагментах кода с заданными аргументами, объявления импорта атомарных фрагментов кода в LuNA-программу и многое другое.

Модуль генератора MPI-программы компилирует LuNA-программу с опцией --compile-only для дальнейшего анализа данного файла. Модуль составляет информацию о всех фрагментах вычислений и фрагментах кода.

### 3.3.2 Процесс генерации MPI-программы

После того, как получено промежуточное состояние свертки и была скомпилирована LuNA-программа, утилита начинает генерацию исходного кода MPI-программы.

Генерация происходит следующим образом: утилита выполняет обход массива элементов поля execution файла с промежуточным состоянием свертки, которые представляют собой описания применения операторов управления с заданными аргументами, транслирует каждый элемент в блок кода согласно заданному шаблону и вставляет этот блок кода в тело функции main() генерируемой MPI-программы.

Каркас исходного кода MPI-программы следующий: в первую очередь включены в код необходимые заголовочные файлы, далее экспортируются все фрагменты кода, а после задается каркас функции main(). Описанные действия происходят до поэлементной итерации поля execution, описанной в предыдущем абзаце.

После полной итерации добавляется кусок кода с finalize-функциями, например, одна из них — MPI_Finalize(). Таким образом, получаем исходный код MPI-программы.

Далее файл с исходным кодом программы компилируется компилятором mpic++ в исполняемый файл, который можно запустить при помощи mpirun или mpiexec.

#### 3.3.2.1 Анализ метаинформации LuNA-программы

В процессе генерации MPI-программы, собирается информация об всех фрагментах вычисления и фрагментов кода — она содержится в файле program_recom.ja, который можно получить добавив опцию --compile-only и указав папку сборки с помощью опции --build-dir. На рисунках 24 представлена LuNA-программа, в файле program_recom.ja содержится метаинформация о данной программе (см. Приложение В).

```
import c_helloWorld(int) as hello_world;
sub main()
{
      cf a: hello_world(1);
      cf b: hello_world(2);
      cf c: hello_world(3);
}
```

Рисунок 24 — Простая LuNA-программа

На первом уровне вложенности описаны импортированные атомарные фрагменты кода и все структурированные фрагменты кода. Импортированные атомарные фрагменты кода имеют в поле type значение “extern”, структурированные — значение “struct”.

Структуры, описывающие импортированные атомарные фрагменты кода, содержат описание типов всех входных параметров в значении поля “args”, имя связанной процедуры на языке C++ в значении поля “code”, описанной в файле ucodes.cpp, а также дополнительную различную информацию. Имя поля, значение которого представляет ранее описанную структуру, это имя фрагмента кода в LuNA-программе — данное имя используется в фрагментах вычислений.

Структуры, описывающие структурированные фрагменты кода, содержат в значении поля “body” объявления фрагментов данных, описание фрагментов вычислений, циклов и другой метаинформации. Для объявления фрагментов данных поле “type” имеет значение “dfs”, значении поля “names” — это список имен объявляемых фрагментов. Для описания фрагментов вычислений поле “type” имеет значение “exec”, значение поля “id” содержит имя импортированного атомарного фрагмента кода, значение поля “args” содержит список заданных аргументов.

В процессе парсинга заполняется следующая структура, представленная на рисунке 25.

```
     # Stores parsed fragments from program_recom.ja file
class LunaFragments:
    def __init__(self):
        self.data_fragments = dict()
        self.code_fragments = dict()
        self.calculation_fragments = []
```

Рисунок 25 — Структура, содержащая информацию о всех фрагментах данных, кода и вычислений LuNA-программы

#### 3.3.2.2 Генерация исходного кода MPI-программы

После составления структуры, содержащей информацию о всех фрагментах данных, кода и вычислений LuNA-программы, начинается поэлементная генерация кода MPI-программы. Генератор поэлементно итерируется по массиву значений поля execution в файле промежуточного состояния свертки, элементы которого представляют описание применение операторов управления свертки с заданными аргументами.

Тип оператора указан в элементе в значении поля “type”, всего четыре значения — “df”, “run”, “send”, обозначающие операторы управления соответственно. Оператор “for” раскручивается в эквивалентный список операторов управления.

При обработке элемента генерируется код на языке C++ согласно заданному шаблону и вставляется в файл исходного кода программы. За генерацию кода и вставке в файл отвечает класс CPPFileHandler. Для каждого оператора управления есть соответственный метод для операции генерации и вставки кода, они представлены на рисунках 26-28.

```
    def include_define_df(self, df):
        self.write_line(f' \
            dfManager.addNewDF(new DFDescriptor("{df.name}")); \
        ')
```

Рисунок 26 — Метод для вставки кода оператора управления df

```
    def include_cf_execution(self, calculation_fragment,
code_fragment, rank):
        args = ''
        for arg in calculation_fragment.args:
            match arg.type:
                case 'iconst':
                    args += str(arg.value) + ', '
                case 'rconst':
                    args += str(arg.value) + ', '
                case 'sconst':
                    args += f'\"{arg.value}\", '
                case 'var':
                    cpp_list_define = f'"{arg.name}"'
                    for cf_ref_part in arg.ref:
                        cpp_list_define += f', "{cf_ref_part}"'
                    args += f'*dfManager.getDFByFullName({{
{cpp_list_define} }}), '
        args = args[:-2]
        self.write_line(f'if (rank == {rank}) {{ \
            {code_fragment.code}({args}); \
        }}')
```

Рисунок 27 — Метод для вставки кода оператора управления run

```
def include_df_send(self, df_name, from_rank, to_rank):
        cpp_list_define = f'"{df_name[0]}" '
        for df_name_part in df_name[1:]:
            cpp_list_define += f', "{df_name_part}"'
        self.write_line(f' \
            dfManager.sendDfBetweenNodes({{ {cpp_list_define} }}, rank,
{from_rank}, {to_rank}); \
        ')
```

Рисунок 28 — Метод для вставки кода оператора управления send

#### 3.3.2.3 Компиляция сгенерированной MPI-программы в исполняемый файл

За компиляцию сгенерированный программы отвечает класс MPIProgramCompiler, реализующий интерфейс IProgramCompiler, содержащий один метод compile. В реализация данного метода в классе MPIProgramCompiler

включаются зависимости необходимых C++ классов исходного кода LuNA и компилируется сгенерированный ранее исходный код MPI-программы при помощи компилятора mpic++. Реализация данного метода представлена на рисунке 29.

```
def compile(self, mpi_prog_cpp, ucodes_cpp, output_name):
        luna_src_headers_path = self._luna_home_path + '/include'
        luna_src_sources_path = self._luna_home_path + '/src/rts'
        bundle_lib_src_headers_path = self._bundle_home_path +
'/mpi-generator/include'

        # Include luna src headers
        os_command = f'{self._mpi_compiler} -O3 -std=c++11
build/luna_manual_mpi_program_src.cpp {ucodes_cpp} -I
{luna_src_headers_path} -I {bundle_lib_src_headers_path} '

        # Including luna src dependencies
        luna_source_dependencies = {'df.cpp', 'common.cpp',
'serializable.cpp'}
        for dependency in luna_source_dependencies:
            os_command += f'{luna_src_sources_path}/{dependency} '
        bundle_lib_dependencies = {'dfmanager.cpp'}
        for dependency in bundle_lib_dependencies:
            os_command += f'{bundle_lib_src_headers_path}/{dependency}
'
        # Specify output name
        os_command += f'-o {output_name}'

        # Compile mpi program
        logging.info(f'Compiling MPI program >>> {os_command}')
        err_code = os.system(os_command)
        if err_code != 0:
            raise OsCommandExecutionException('Error while compiling
MPI program')
```

Рисунок 29 — Реализация метода compile в классе MPIProgramCompiler.

### 3.4 Запуск скомпилированной MPI-программы.

Запуск скомпилированной программы осуществляется также, как и запуск других MPI-программ. Для этого предназначены утилиты командной строки mpirun и mpiexec. Пример запуска программы представлен на рисунке 30.

```
$ mpiexec -n 4 ./prog
```

Рисунок 30 — Запуск скомпилированной MPI-программы.

### 3.5 Ограничения программных средств

Реализованные программные средства имеют ряд некоторых ограничений:

1. Исполнение описывается только для структурированного фрагмента кода main.
2. ФД и ФВ обязаны обладать уникальными именами, т.к. являются глобально видимыми.

В процессе разработки данные ограничения будут устранены.

### 3.6 Тестирование производительности

Для оценки эффективности реализованных средств были подготовлены несколько тестов. В тестах проверяется время исполнения LuNA-программы и сгенерированной MPI-программы при помощи средств прямого управления исполнением (СПУИ). Замер времени выполнения в секундах проводился для 1, 2, 4 и 8 узлов на информационно-вычислительном центре Новосибирского государственного университета [17].

Первый тест заключается в 320-кратном умножении целочисленных матриц размером 1000x1000 между собой. Его суть заключается в проверке эффективности исполнения для легкого случая распараллеливания, когда все задачи имеют одинаковую вычислительную сложность и можно равномерно распределить все задачи между узлами. Результаты теста представлены в таблице 1 и на рисунке 31.

Таблица 1 – Результаты теста №1

| Кол-во узлов | LuNA | СПУИ |
|---|---|---|
| 1 | 977,77 | 720,47 |
| 2 | 871,73 | 332,85 |
| 4 | 523,038 | 183,067 |
| 8 | 261,519 | 109,811 |

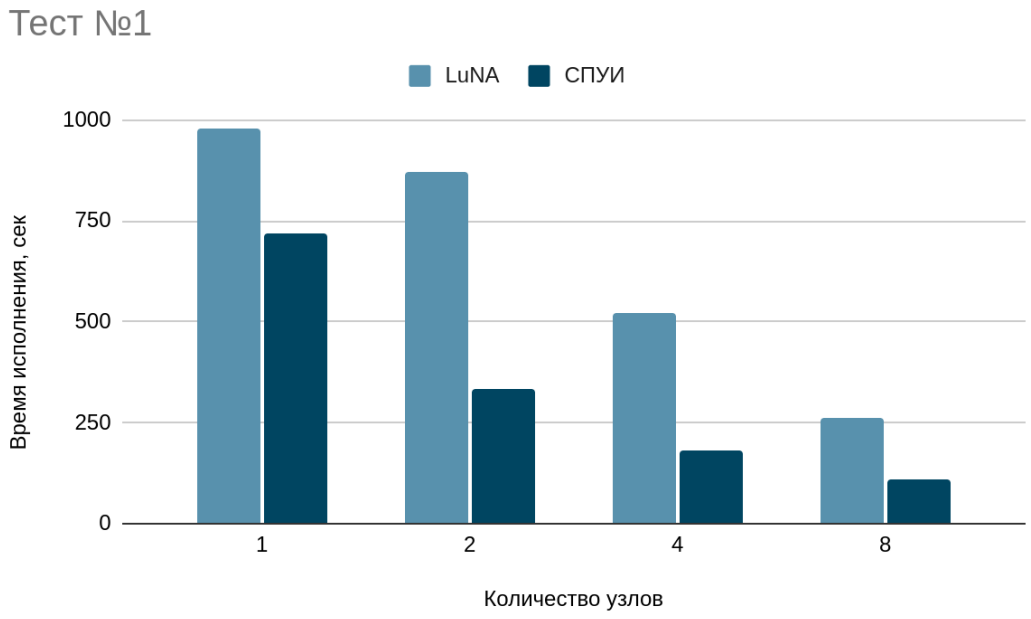


Рисунок 31 — Результаты теста №1 в виде диаграммы

Исходя из результатов данного теста, можно сделать вывод, что разработанные средства позволяют сократить время исполнения LuNA-программы.

Второй тест представляет собой 80-кратное умножение целочисленных матриц с размерами 2000x1000 и 1000x2000 и 320-кратное умножении матриц размером 1000x1000 между собой. Одна задача имеет вычислительную сложность в 4 раза больше, чем вторая, но при этом имеет в 4 раза меньшую кратность выполнения. Все 400 задач можно распределить между узлами в свертке так, чтобы нагрузка на каждый. Однако, не факт, что runtime-система

LuNA построит наилучшую стратегию распределения задач. Результаты теста представлены в таблице 2 и на рисунке 32.

Таблица 2 – Результаты теста №2

| Кол-во узлов | LuNA | СПУИ |
|---|---|---|
| 1 | 1 324,72 | 1191,541 |
| 2 | 1179,8 | 479,78 |
| 4 | 707,413 | 249,213 |
| 8 | 459,55 | 151,99 |

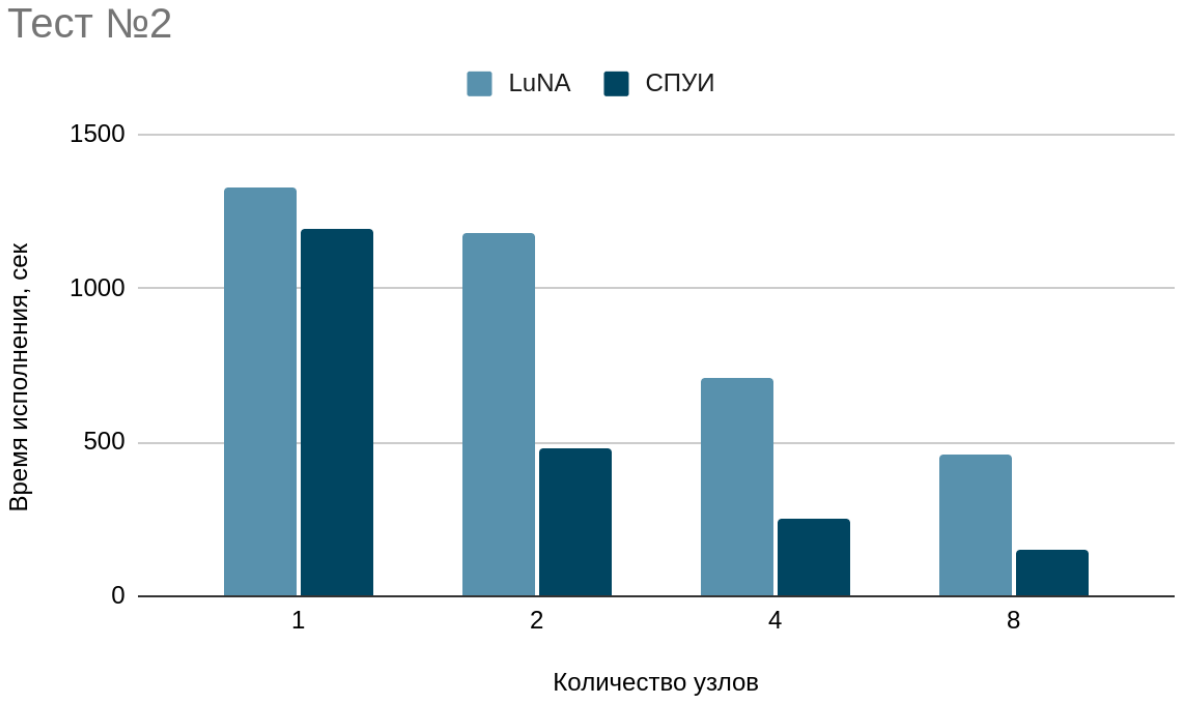


Рисунок 32 — Результаты теста №2 в виде диаграммы

По результатам данного теста видно, что LuNA не всегда оптимально распределяет задачи с разными вычислительными сложностями между узлами, а при ручном равномерном распределении при помощи реализованных средств этого можно избежать.

Третий тест заключается в 32-кратном скалярном произведения целочисленных векторов длиной 10000000. Каждый узел вычисляет свою часть скалярного произведения, затем частичные суммы суммируются в одну общую. Суть данного теста заключается в проверке на более сложной задаче, чем

которые были в тестах 1 и 2. Результаты теста представлены в таблице 3 и на рисунке 33.

Таблица 3 – Результаты теста №3

| Кол-во узлов | LuNA | СПУИ |
|---|---|---|
| 1 | 4,58 | 4,312 |
| 2 | 3,896 | 3,888 |
| 4 | 3,201 | 3,842 |
| 8 | 2,902 | 5,253 |

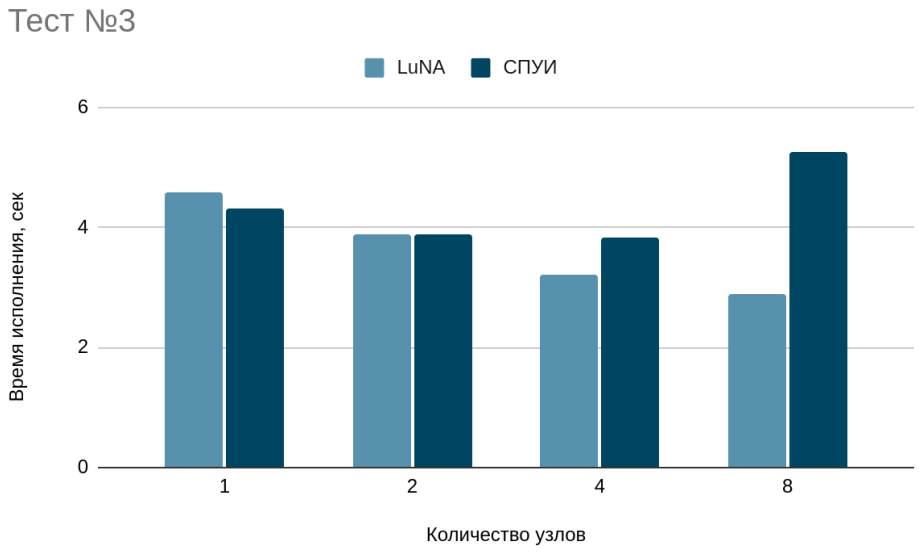


Рисунок 33 — Результаты теста №3 в виде диаграммы

По результатам теста видно, что разработанные средства показали отрицательный результат для 4 и 8 узлов. Причина такого результата следующая: реализованный алгоритм генерации MPI-программы довольно примитивен, т.к. почти весь код генерируемой программы содержится в функции main(). Из-за большого размера функции увеличивается время компиляции и уменьшается время исполнения такой программы, чем если разбивать программу на подпрограммы.

Из этого можно сделать вывод, что требуется усовершенствование алгоритма генерации MPI-программы путем оптимизации исходного кода генерируемой MPI-программы.

# ЗАКЛЮЧЕНИЕ

В результате выполнения работы были разработаны и реализованы язык описания сверток и алгоритм генерации MPI-программы на основе LuNA-программы и описанной для нее свертки. Алгоритм был реализован в виде программных средств.

Проведено тестирование разработанного алгоритма генерации MPI-программы. Тестирование показало, что алгоритм позволяет сгенерировать аналогичную MPI-программу с более эффективным исполнением.

Защищаемые положения:

1. Разработан и реализован язык описания сверток, позволяющий описывать распределение ФВ и ФД на узлы.
2. Разработан алгоритм генерации MPI-программы на основе LuNA-программ и описанной для нее свертки.
3. Алгоритм генерации MPI-программы реализован и протестирован для некоторых LuNA-программ. Тестирование показало, что для некоторого класса задач эффективность исполнения программ увеличилась.

В дальнейшем планируется оптимизировать алгоритм генерации MPI-программы, а также расширение функциональности разработанных программных средств по мере анализа полноты на различных задачах.

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».

Брусникин Михаил Андреевич _____

*ФИО студента* *Подпись студента*

«____ »_____2023 г.

*(заполняется от руки)*

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ


1. Бондаренко В.В., Козич В.Г., Плахотная Л.А. Достоинства и недостатки параллельного программирования // Современные научные исследования и инновации. 2016. № 6 [Электронный ресурс]. URL: https://web.snauka.ru/issues/2016/06/69538 (дата обращения: 10.05.2023).
2. V.E. Malyshkin, V.A. Perepelkin. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem. Proceedings of the 11th International Conference on Parallel Computing Technologies (PaCT-2011), LNCS, Vol. 6873, 2011, pp. 53-61. DOI: 10.1007/978-3-642-23178-0_5.
3. Малышкин В. Э. Технология фрагментированного программирования // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. – №. 46 (305). – С. 45-55.
4. Programming with Legion. Официальная документация Legion [Электронный ресурс]. – URL: https://legion.stanford.edu/pdfs/legion-manual.pdf (дата обращения 15.04.2023).
5. Документация Legion Mapper API [Электронный ресурс]. – URL: https://legion.stanford.edu/mapper/ (дата обращения 15.04.2023).
6. Официальная документация PaRSEC [Электронный ресурс]. – URL: https://github.com/icldisco/parsec/wiki (дата обращения 15.04.2023).
7. Официальный репозиторий DPLASMA [Электронный ресурс]. – URL: https://github.com/ICLDisco/dplasma (дата обращения 15.04.2023).
8. The Charm++ Parallel Programming System. Официальная документация Charm++ [Электронный ресурс]. – URL: https://charm.readthedocs.io/en/latest/charm%2B%2B/manual.html (дата обращения 15.04.2023).
9. Описание языка LuNA [Электронный ресурс]. URL: https://gitlab.ssd.sscc.ru/luna/luna5/wikis/luna_lang_v01 (дата обращения

15.04.2023).

10. Ахмед-Заки Д. Ж., Лебедев Д. В., Малышкин В. Э., Перепелкин В. А. Автоматизация конструирования распределенных программ численного моделирования в системе LuNA на примере модельной задачи // журнал "Проблемы информатики", 2019, № 4, с.53-64. DOI: 10.24411/2073-0667-2019-00016.
11. MPI Documents. Официальная документация стандарта MPI [Электронный ресурс]. – URL: https://www.mpi-forum.org/docs/ (дата обращения 24.04.2023).
12. Введение в JSON [Электронный ресурс]. – URL: https://www.json.org/json-ru.html (дата обращения: 24.04.2023).
13. Stroustrup B. C++ Programming Language // Pearson Education. — 2013.
14. Официальная документация GNU Flex [Электронный ресурс]. – URL: https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html (дата обращения: 24.04.2023).
15. Официальная документация GNU Bison [Электронный ресурс]. – URL: https://www.gnu.org/software/bison/manual/bison.html (дата обращения 24.04.2023).
16. Документация языка Python [Электронный ресурс]. – URL: https://docs.python.org/3/ (дата обращения 24.04.2023).
17. Информационно-вычислительный центр Новосибирского государственного университета [Электронный ресурс]. – URL: http://nusc.nsu.ru/ (дата обращения 24.04.2023).

## ПРИЛОЖЕНИЕ А

Пример промежуточного состояния свертки

```
{
 "define": {
   "N": "4"
 },
 "execution": [
   {
     "type": "df",
     "name": "x"
   },
   {
     "type": "run",
     "rank": {
       "type": "iconst",
       "value": 0
     },
     "cf": [
       "initX"
     ]
   },
   {
     "type": "send",
     "data": [
       "x"
     ],
     "from": {
       "type": "iconst",
       "value": 0
     },
     "to": {
       "type": "iconst",
       "value": 1
     }
   },
   {
     "type": "for",
     "iterator": "i",
     "startValue": {
       "type": "iconst",
       "value": 1
     },
     "endValue": {
```

```
        "type": "iconst",
        "value": 4
      },
      "body": [
        {
          "type": "run",
          "rank": {
            "type": "iconst",
            "value": 1
          },
          "cf": [
            "printValue"
          ]
        }
      ]
    }
  ]
}
```

# ПРИЛОЖЕНИЕ Б

Алгоритм генерации MPI-программы

```
parse_for_block(metainformation, for_block_json):
    for subblock in for_block_json['body']:
        if subblock.type == "exec":
            metainformation.calculation_fragments.add(block)
        elif subblock.type == "for":
            parse_for_block(metainformation, subblock)


parse_luna_metainformation(luna_program_recom_json):
    var metainformation := {
        data_fragments: list,
        calculation_fragments: list,
        code_fragments: list
    };

    # parsing import code fragments
    for block in luna_program_recom_json:
        if block['type'] == "extern":
            metainformation.code_fragments.add(block)

    # parsing calculation fragments
    for block in luna_program_recom_json['main']['body']:
        if block['type'] == "dfs":
            metainformation.data_fragments.add(block)
        elif block['type'] == "exec":
            metainformation.calculation_fragments.add(block)
        elif block['type'] == "for":
            parse_for_block(metainformation, block)

    return metainformation


unwrap_for_operator(result, for_operator):
    for operator in for_operator.body:
        if operator.type == "run":
            result += cpp_file_handler.generate_run_cf_on_rank(operator)
        elif operator.type == "send":
            result += cpp_file_handler.generate_send_df(operator)
        elif operator.type == "for":
            unwrap_for_operator(result, operator)


generate_main_func(bundle, metainformation, result):
    for operator in bundle.execution_block:
        if operator.type == "df":
            result += cpp_file_handler.generate_df_declaration(operator)
        elif operator.type == "run":
            result += cpp_file_handler.generate_run_cf_on_rank(operator)
        elif operator.type == "send":
```

```
            result += cpp_file_handler.generate_send_df(operator)
        elif operator.type == "for":
            unwrap_for_operator(result, operator)


# Алгоритм генерации MPI-программы
mpi_program_generation(luna_program_file, bundle, atomic_code_fragments):
    var luna_compile_result := luna.compile(luna_program_file, atomic_code_fragments)
    var parse_luna_metainformation_result :=
parse_luna_metainformation(luna_compile_result.luna_program_recom_json)
    var result := ""
    mpi_cpp_compiler.init_mpi_program(result)
    generate_main_func(bundle, parse_luna_metainformation_result, result)
    mpi_cpp_compiler.finalize_mpi_program(result)
    mpi_cpp_compiler.compile(result, atomic_code_fragments)
```

## ПРИЛОЖЕНИЕ В

Содержимое файла program_recom.ja

```
{
    "hello_world": {
        "args": [
            {
                "type": "int"
            }
        ],
        "code": "c_helloWorld",
        "cuda_support": "false",
        "nocpu": "false",
        "type": "extern"
    },
    "main": {
        "args": [],
        "begin": 41,
        "body": [
            {
                "names": [],
                "type": "dfs"
            },
            {
                "args": [
                    {
                        "is_expr": true,
                        "type": "iconst",
                        "value": 1
                    }
                ],
                "begin": 61,
                "code": "hello_world",
                "id": [
                    "a"
                ],
                "rules": [
                    {
                        "begin": 61,
                        "items": [],
                        "property": "req_unlimited",
                        "ruletype": "enum"
                    }
                ],
                "type": "exec"
            },
            {
                "args": [
                    {
                        "is_expr": true,
                        "type": "iconst",
                        "value": 2
```

```
                    }
                ],
                "begin": 84,
                "code": "hello_world",
                "id": [
                    "b"
                ],
                "rules": [
                    {
                        "begin": 84,
                        "items": [],
                        "property": "req_unlimited",
                        "ruletype": "enum"
                    }
                ],
                "type": "exec"
            },
            {
                "args": [
                    {
                        "is_expr": true,
                        "type": "iconst",
                        "value": 3
                    }
                ],
                "begin": 107,
                "code": "hello_world",
                "id": [
                    "c"
                ],
                "rules": [
                    {
                        "begin": 107,
                        "items": [],
                        "property": "req_unlimited",
                        "ruletype": "enum"
                    }
                ],
                "type": "exec"
            }
        ],
        "end": 124,
        "type": "struct",
        "where": {
            "type": "luna"
        }
    }
}
```

**ПРИЛОЖЕНИЕ Г**

**Средства прямого управления исполнением LuNA-программ**

**Руководство оператора**

Листов 9

Новосибирск 2023 г.

## СОДЕРЖАНИЕ

**АННОТАЦИЯ** **48**

**1 Назначение программы** **49**

1.1 Назначение программных средств 49

1.2 Эксплуатационное назначение программы 49

1.3 Состав функций 49

**2 Условия выполнения программы** **50**

2.1 Необходимое программное обеспечение 50

2.2 Требования к оператору 50

**3 Выполнение программы** **51**

3.1 Запуск программных средств 51

3.2 Аргументы запуска и выходные данные модуля парсинга и генерации промежуточного состояния описания распределения ресурсов вычислительной системы 51

3.3 Опции запуска и выходные данные модуля генерации MPI-программы 51

3.4 Завершение программы 53

**4 Сообщения оператору** **54**

# АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению и эксплуатации средств прямого управления исполнением LuNA-программ. Данный программный комплекс являются автономным модулем для системы LuNA. Комплекс состоят из двух программных модулей – модуля парсинга и генерации промежуточного состояния описания распределения ресурсов и модуля генерации MPI-программы.

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

В разделе «Условия выполнения программы» перечислены условия, являющиеся необходимыми для выполнения программы.

Раздел «Выполнение программы» содержит последовательность действий оператора, которые необходимы для загрузки, запуска, выполнения и завершения программы, приведено описание функций, формата и возможных вариантов команд, с помощью которых оператор осуществляет загрузку и управляет выполнением программы, а также ответы программы на эти команды.

В разделе «Сообщения оператору» приведены тексты сообщений, выдаваемых в ходе выполнения программы, описание их содержания и соответствующие действия оператора (действия оператора в случае сбоя, возможности повторного запуска программы и т.п.).

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

Разработанный программный комплекс предназначен для генерации MPI-программы с заданным распределением ресурсов вычислительной системы на основе LuNA-программы. Данное решение позволяет повысить эффективность исполнения программы в случае грамотного распределения, также исключаются накладные ресурсы в виде runtime-системы LuNA.

## 1.2 Эксплуатационное назначение программы

Программный комплекс предназначен для использования разработчиками фрагментированным программ в системе LuNA.

## 1.3 Состав функций

- Генерация MPI-программы на основе LuNA-программы и описанного для нее распределения ресурсов вычислительной системы.
- Компиляция сгенерированной MPI-программы в исполняемый файл.
- Генерация промежуточного состояния описания распределения ресурсов вычислительной системы.

## 2 Условия выполнения программы

### 2.1 Необходимое программное обеспечение

Для работы программного комплекса необходимо следующее программное обеспечение:

1. Система LuNA, а также все программное обеспечение, необходимое для ее работы;
2. Python 3 версии 3.6 и выше;
3. Python3-pip;
4. Библиотека pyinstaller.

### 2.2 Требования к оператору

Конечный оператор (пользователь) программы должен иметь навык параллельного программирования, обладать знаниями языка C++ и системы LuNA.

## 3 Выполнение программы

### 3.1 Запуск программы

Запуск программы осуществляется через интерфейс командной строки. Для каждого модуля программного комплекса необходимо выполнить свою команду с необходимыми опциями или аргументами. Например, пример запуска для модуля генерации MPI-программы: luna–mpi-generator --config-file config.json. Пример запуска для модуля парсинга и генерации промежуточного состояния описания распределения ресурсов вычислительной системы: luna-bundle-parser bundle.bndl bundle.json.

Модуль генерации MPI-программы в процессе своей работы использует модуль парсинга и генерации промежуточного состояния распределения ресурсов вычислительной системы.

### 3.2 Аргументы запуска и выходные данные модуля парсинга и генерации промежуточного состояния описания распределения ресурсов вычислительной системы

Программа принимает два входные аргумента: путь к файлу с описанным распределением ресурсов и путь к выходному JSON-файлу. Выходными данными является сгенерированный JSON-файл, представляющий собой промежуточное состояние описания распределения ресурсов.

### 3.3 Опции запуска и выходные данные модуля генерации MPI-программы

Для корректной работы программы необходимо через опции передать следующие файлы: файл с LuNA-программой, файл с описанными атомарными фрагментами кода и файл с описанным распределениему. Помимо них есть необязательные параметры, в случае отсутствия указания конкретных значения для которых определены значения по умолчанию. Также для удобства все необходимые аргументы можно передать при помощи конфигурационного JSON-файла. Список опций следующий:

1. --luna-src путь_к_файлу — устанавливает путь к исходному коду LuNA-программы. Является обязательной опцией в случае отсутствия опции --config-file;
2. --cpp-codes путь_к_файлу — устанавливает путь к файлу с описанными атомарными фрагментами кода для LuNA-программы. Является обязательной опцией в случае отсутствия опции --config-file;
3. --bundle-file путь_к_файлу — устанавливает путь к файлу с описанным распределением ресурсов вычислительной системы. Является обязательной опцией в случае отсутствия опции --config-file;
4. --config-file путь_к_файлу — устанавливает путь к конфигурационному файлу. Является обязательной опцией в случае отсутствия трех опций --luna-src, --cpp-codes и --bundle-file;
5. --mpi-header название_header_mpi — устанавливает требуемый header файл для MPI. По умолчанию, используется значение “mpi.h”;
6. --output путь_к_исполняемому_файлу — устанавливает путь и название выходного исполняемого файла в случае компиляции; По умолчанию, используется значение “luna_manual_mpi_program”.
7. --buildOnly — определяет, будет ли компиляция сгенерированной MPI-программы в исполняемый файл. По умолчанию, используется значение False. В случае указания опции присваивается значение True.

Пример запуска программы с тремя обязательными опциями: luna-mpi-generator --luna-src prog.fa --cpp-codes ucodes.cpp --bundle-file bundle.bndl. Пример запуска программы с обязательной опцией --config-file: luna-mpi-generator --config-file config.json. Пример конфигурационного файла представлен на рисунке 1.

```
{
    "bundle_json_path": "bundle.json",
    "luna_src_path": "prog.fa",
    "cpp_codes_path": "ucodes.cpp",
    "mpi_header": "mpi.h"
}
```

Рисунок 1 — Пример конфигурационного файла

Выходными данными является сгенерированная MPI-программа и исполняемый файл в случае, если опция --buildOnly имеет значение False.

### 3.4 Завершение программы

Для завершения работы программы в интерфейсе командной строки оператору необходимо прервать работу программы. Команда прерывания зависит от выбранной операционной системы.

## 4 Сообщения оператору

В процессе работы программы обоих модулей программного комплекса могут показывать сообщения в командной строке.

Модуль парсинга и генерации промежуточного состояния распределения ресурсов вычислительной системы в случае ошибки покажет сообщение с ее описанием. В случае успешного завершения работы сообщения отсутствуют.

Модуль генерации MPI-программы в процессе исполнения выводит в виде сообщения логи каждого этапа работы В случае ошибки выводится сообщение с ее описанием.

В случае некорректного завершения оператору необходимо изучить сообщение, содержащее описание причины ошибки, исправить и повторить попытку запуска.

# ПРИЛОЖЕНИЕ Д

**Средства прямого управления исполнением LuNA-программ**

Листов 15

Новосибирск 2023 г.

## СОДЕРЖАНИЕ

**АННОТАЦИЯ** **57**

**1 Общие сведения** **58**

1.1 Обозначение и наименование программы 58

1.2 Программное обеспечение, необходимое для функционирования программы 58

**2 Функциональное назначение** **59**

**3 Описание логической структуры** **60**

3.1.1 Структура модуля парсинга и генерации промежуточного состояния описания распределения ресурсов 60

3.1.2 Структура модуля генерации MPI-программы 62

3.3 Алгоритм программы 64

3.4 Связи между составными частями программы 65

3.5 Связи программы с другими подпрограммами 65

**4 Используемые технические средства** **66**

**5 Вызов и загрузка** **67**

**6 Входные данные** **68**

**7 Выходные данные** **69**

# АННОТАЦИЯ

В данном программном документе приведено описание программных средств прямого управления исполнением LuNA-программ, предназначенной для генерации MPI-программы с заданным распределением ресурсов вычислительной системы на основе LuNA-программы. Программный комплекс состоит из двух программных модулей — модуля парсинга и генерации промежуточного состояния описания распределения ресурсов и модуля генерации MPI-программы.

Основной функцией программных средств является генерация MPI-программы с заданными распределением ресурсов вычислительной на основе LuNA-программы.

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

## 1 Общие сведения

### 1.1 Обозначение и наименование программы

1. Модуль парсинга и генерации промежуточного состояния описания распределения ресурсов: luna-bundle-parser.
2. Модуль генерации MPI-программы: luna-mpi-generator.

### 1.2 Программное обеспечение, необходимое для функционирования программы

Для работы программного комплекса необходимо следующее программное обеспечение:

1. Система LuNA, а также все программное обеспечение, необходимое для ее работы;
2. Python 3 версии 3.6 и выше;
3. Python3-pip;
4. Библиотека pyinstaller.

### 1.3 Языки программирования, на которых написана программы

Модули программного комплекса реализованы на двух языках программирования. Языком программирования для модуля парсинга и генерации промежуточного состояния описания распределения ресурсов является C++, для модуля генерации MPI-программы— Python.

## 2 Функциональное назначение

Программный комплекс предназначен для генерации MPI-программы с заданным распределением ресурсов вычислительной системы на основе LuNA-программы. Данное решение позволяет повысить эффективность исполнения программы в случае грамотного распределения, также исключаются накладные ресурсы в виде runtime-системы LuNA. Стоит отметить, что данные программные средства не гарантируют повышение эффективности исполнения, а в некоторых случаях могут показать отрицательный результат. Использование средств программистом на языке LuNA является опциональным.

## 3 Описание логической структуры

### 3.1 Структура программы

Программный комплекс состоит из двух программных модулей — модуля парсинга и генерации промежуточного состояния описания распределения ресурсов и модуля генерации MPI-программы. Главным модулем является второй модуль генерации, использующий первый модуль.

#### 3.1.1 Структура модуля парсинга и генерации промежуточного состояния описания распределения ресурсов

Данный модуль отвечает за генерацию промежуточного состояния описания распределения ресурсов вычислительной системы. Для парсинга описания распределения ресурсов используются утилиты Flex и Bison. Основной алгоритм описан в функции main(). В ней вызывается метод yyparse(), предоставляемый Bison. После завершения метода заполняется глобальная переменная типа BundleContainer, содержащая описания распределения ресурсов. При помощи объекта класса JSONHandler генерируется файл, содержащий промежуточное состояние описанного распределения ресурсов.

Основные классы программы:

- Класс: BundleContainer: содержит описание распределения ресурсов.
- Класс: JSONHandler: генерирует промежуточное состояние описания распределения ресурсов на основе объекта класса BundleContainter.

Далее в таблицах 1-2 перечислены описания публичных методов каждого класса.

Таблица 1 — Перечень публичных методов класса JSONHandler

| Имя метода | Тип результата | Назначение |
|---|---|---|
| generateJSON | void | Генерация файла с промежуточным состоянием описанного распределения ресурсов на основе объекта класса BundleContainer |

Таблица 2 — Перечень публичных методов класса BundleContainer

| Имя метода | Тип результата | Назначение |
|---|---|---|
| getMacroVars | std::map<std::string, std::string> & | Генерация файла с промежуточным состоянием описанного распределения ресурсов на основе объекта класса BundleContainer |
| registerMacroVar | void | Добавление макропеременной и ее значения |
| getMacroVarValueByName | std::string | Получение значения макропеременной по ее имени |
| registerNewBlock | std::string | Добавление дескриптора оператора управления ресурсами |
| getBlockByUUID | IExecuteSubblock* | Получение дескриптора оператора управления ресурсами по его UUID |
| registerNewTask | std::string | Добавление дескриптора имени фрагмента вычислений |
| getTaskByUUID | TaskDescriptor* | Получение дескриптора имени фрагмента вычислений по его UUID |
| registerNewContext | std::string | Добавление дескриптора контекста исполнения |
| getContextByUUID | ExecutionContext* | Получение дескриптора контекста исполнения по его UUID |
| setMainContext | void | Задание главного контекста исполнения |
| getMainContext | ExecutionContext* | Получение главного контекста исполнения по его UUID |

| | | |
|---|---|---|
| registerNewDFD | std::string | Добавление дескриптора фрагмента данных |
| getDFDByUUID | DFDescriptor* | Получение дескриптора фрагмента данных по его UUID |
| registerNewExpression | std::string | Добавления дескриптора арифметического выражения |
| getExpressionByUUID | IIntExpression* | Получение дескриптора арифметического выражения по его UUID |

### 3.1.2 Структура модуля генерации MPI-программы

Данный модуль отвечает за генерацию MPI-программы согласно описанному распределению ресурсов вычислительной системы на основе LuNA-программы.

Основные классы программы:

- Класс BuildConfigParser: ответственен за парсинг входные аргументов.
- Класс MPIProgramBuilder: основной класс, ответственный за генерацию MPI-программы, использует методы остальных классов.
- Класс ProgramRecomHandler: ответственен за парсинг метаинформации о LuNA-программе для составления информации об всех фрагментах данных, фрагментов кода, фрагментов вычислений и их параметров и аргументов.
- Класс CPPFileHandler: ответственен за генерацию участков кода в исходном коде генерируемой MPI-программы.
- Класс MPIProgramCompiler: ответственен за компиляцию сгенерированной MPI-программы в исполняемый файл.

Далее в таблицах 3-7 перечислены описания публичных методов каждого класса.

Таблица 3 — Перечень публичных методов класса BuildConfigParser

| Имя метода | Тип результата | Назначение |
|---|---|---|
| get_build_config | BuildConfig | Обработка переданных аргументов программы и составления структуры с информацией, необходимой для генерации и компиляции MPI-программы |

Таблица 4 — Перечень публичных методов класса MPIProgramBuilder

| Имя метода | Тип результата | Назначение |
|---|---|---|
| build | void | Генерация исходного кода MPI-программы на основе переданного объекта-дескриптора класса BuildConfig |

Таблица 5 — Перечень публичных методов класса ProgramRecomHandler

| Имя метода | Тип результата | Назначение |
|---|---|---|
| parse_program_recom_json | void | Парсинг файла с метаинформацией о LuNA-программе и составление структуры, содержащей информацию о всех фрагментах данных, фрагментах кода, фрагментов вычислений и их параметров и аргументов |

Таблица 6 — Перечень публичных методов класса CPPFileHandler

| Имя метода | Тип результата | Назначение |
|---|---|---|
| write_line | void | Запись строки с символом переноса строки в файл |
| write_empty_line | void | Запись символа переноса строки в файл |
| include_user_header | void | Генерация и запись строки в файл с |

| | | включением C++ header-файла из общей библиотеки. Например, #include <mpi.h> |
|---|---|---|
| include_std_header | void | Генерация и запись строки в файл с включением пользовательского C++ header-файла. Например, #include “mylib.h”. |
| include_extern_void_func | void | Генерация и запись строки в файл с объявлением функции |
| include_define_df | void | Генерация и запись строки в файл с объявлением фрагмента данных |
| include_cf_execution | void | Генерация и запись строки в файл с использованием фрагмента вычислений |
| include_df_send | void | Генерация и запись строки в файл с пересылкой фрагмента данных между узлами |
| finalize | void | Закрыть файл |

Таблица 7 — Перечень публичных методов класса MPIProgramCompiler

| Имя метода | Тип результата | Назначение |
|---|---|---|
| compile | void | Компиляция сгенерированной MPI-программы |

### 3.3 Алгоритм программы

1. Модуль парсинга описания распределения ресурсов программы и генерации промежуточного состояния обрабатывает файл с описанными распределением и создает json-файл с промежуточным состоянием.
2. Модуль генерации MPI-программы на основе полученного промежуточного состояния и LuNA-программы генерирует файл с исходным кодом MPI-программы.

3. Модуль генерации MPI-программы компилирует сгенерированный файл с помощью компилятора mpic++ в исполняемый файл.

### 3.4 Связи между составными частями программы

Модуль генерации MPI-программы в процессе своей работы использует модуль парсинга и генерации промежуточного состояния распределения ресурсов вычислительной системы.

### 3.5 Связи программы с другими подпрограммами

Модуль генерации MPI-программы использует систему LuNA для компиляции метаинформации об LuNA-программе и утилиту mpic++ для компиляции сгенерированной MPI-программы в исполняемый файл.

## 4 Используемые технические средства

Программа может эксплуатироваться на персональном компьютере или сервере, которое может запускать необходимое программное обеспечение для работы данного программного комплекса. На вычислительном устройстве должна быть установлена ОС семейства Unix.

## 5 Вызов и загрузка

Запуск программного модуля парсинга описания распределения ресурсов программы и генерации промежуточного состояния осуществляется через командную строку путем вызова команды luna-bundler-parser, команде передаются необходимые аргументы. Пример использования: luna-bundle-parser bundle.bndl bundle.json.

Запуск программного модуля генерации MPI-программы осуществляется через командную строку путем вызова команды luna-mpi-generator, также задаются необходимые опции. Пример использования luna-mpi-generator --config-file config.json.

## 6 Входные данные

Программный модуль парсинга описания распределения ресурсов программы и генерации промежуточного состояния принимает два входные аргумента: путь к файлу с описанным распределением ресурсов и путь к выходному JSON-файлу.

Программный модуль генерации MPI-программы принимает на вход следующие файлы: файл с LuNA-программой, файл с описанными атомарными фрагментами кода и файл с описанным распределениему. Помимо них есть необязательные параметры, в случае отсутствия указания конкретных значения для которых определены значения по умолчанию. Также для удобства все необходимые аргументы можно передать при помощи конфигурационного JSON-файла.

## 7 Выходные данные

Выходными данными программного модуля парсинга описания распределения ресурсов программы является сгенерированный JSON-файл, представляющий собой промежуточное состояние описания распределения ресурсов.

Выходными данными программного модуля генерации MPI-программы является сгенерированная MPI-программа и исполняемый файл в случае, если опция указана соответствующая опция.