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

Капраловой Регины Евгеньевны

Тема работы:

РАЗРАБОТКА И РЕАЛИЗАЦИЯ СРЕДСТВА ИНТЕГРАЦИИ LUNA-ПОДПРОГРАММ В ТРАДИЦИОННЫЕ ПАРАЛЛЕЛЬНЫЕ ПРОГРАММЫ

«К защите допущена»
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

Студентке Капраловой Регине Евгеньевне, группы 19201
(фамилия, имя, отчество, номер группы)

Тема Разработка и реализация средства интеграции LuNA-программ в традиционные параллельные программы
(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):

Разработка модели взаимодействия MPI- и LuNA-программ, разработка и реализация программного модуля, обеспечивающего возможность интеграции LuNA-программ в традиционные параллельные программы.

Структурные части работы:

Обзор предметной области, постановка задачи, формулировка требований к программе, реализация программы, тестирование.

Руководитель ВКР
Доцент каф. ПВ ФИТ НГУ,
к. т. н., доцент
Маркова В.П. /……..
(ФИО) / (подпись)
«07» ноября 2022 г.

Задание принял к исполнению
Капралова Р.Е. /……..
(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

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ 7

1.1 Charm++ 7

1.2 Legion 7

1.3 X10 8

1.4 Cilk 9

1.5 Выводы 9

2 СРЕДСТВО ИНТЕГРАЦИИ LuNA-ПРОГРАММ В MPI-ПРОГРАММЫ 10

2.1 MPI-программа 10

2.2 LuNA-программа 11

2.3 Задачи 14

2.4 Модель взаимодействия MPI- и LuNA-программ 15

2.5 Интерфейс 17

2.6 Дальнейшее развитие интерфейса 19

2.6.1 Контроль соответствия подаваемых аргументов сигнатуре фрагментированной подпрограммы 19

2.6.2 Определение локатора по ссылочному выражению 20

3 РЕАЛИЗАЦИЯ ПОДСИСТЕМЫ ЗАПУСКА LuNA-ПОДПРОГРАММ ИЗ MPI-ПРОГРАММ 21

3.1 Структура системы LuNA 21

3.2 Подсистема запуска LuNA-подпрограмм из MPI-программ 22

3.2.1 Параметр командной строки. Ключ --sub 22

3.2.2 Реализация определения точки входа для запуска выбранного фрагмента кода по его имени 23

3.2.3 Класс SP 24

3.2.4 Модификация класса RTS 27

3.3 Примеры использования класса SP 28
3.3.1 Простейший пример 28
3.3.2 Пример с обменом данными 30
4 ТЕСТИРОВАНИЕ 35
4.1 Тестирование программы, выполняющей умножение матриц 35
4.2 Тестирование программы, решающей уравнение Пуассона 36
ЗАКЛЮЧЕНИЕ 38
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ 40
ПРИЛОЖЕНИЕ А 43
ПРИЛОЖЕНИЕ Б 59

ВВЕДЕНИЕ

Параллельное программирование задач численного моделирования эффективнее последовательного. Но разработать эффективную параллельную программу [1] непросто: это требует определённой квалификации для решения проблем связанных с параллельным программированием, которые лежат за пределами предметной области, в которой проводится моделирование. Системы автоматического конструирования параллельных программ частично решают эти проблемы.

Но на практике их применение затрудняется тем, что переписывать весь код уже имеющейся программы в формат, пригодный для выбранной системы автоматического конструирования параллельных программ, трудоемко и затратно. Кроме того, в реальных задачах это и не является необходимым требованием: в них, как правило, имеется часть (или части) кода, которая занимает большую часть времени работы и действительно нуждается в эффективной параллельной реализации. Отсюда возникает необходимость обеспечить возможность вставлять в программу вызовы подпрограмм, сконструированных с помощью систем автоматического конструирования.

Здесь возникает следующая научная проблема. Обеспечить взаимодействие программ, которые написаны на разных языках программирования и, более того, с разными моделями памяти или вычислений, как правило, действительно сложно. И поскольку типового решения данной проблемы нет, необходимо решить её для каждой пары средств в отдельности.

Одной из таких систем программирования является система LuNA [2], которая основана на технологии фрагментированного программирования [3]. Для неё необходимо обеспечить связь с MPI [4] как с низкоуровневым высокопроизводительным и наиболее распространенным стандартом интерфейса обмена данными в параллельном программировании. Что и стало целью моей работы.

Для системы LuNA сейчас фактически единственная возможность вызвать LuNA-подпрограмму из MPI-программы – сохранить данные из программы во входной файл, отдельно запустить LuNA-программу, результат сохранить в выходные файлы и затем продолжить исполнение основной программы, прочитав данные из выходных файлов. Очевидно, что это не удобно и может быть сопряжено с трудностями.

В связи с этим была поставлена следующая **цель работы**: разработать и реализовать средства интеграции LuNA-подпрограмм в традиционные параллельные программы.

Задачи:

Были сформулированы следующие задачи:

1. Разработать модель взаимодействия MPI- и LuNA- программ.
2. Разработать алгоритмы и структуры данных, необходимые для обеспечения взаимодействия программ.
3. Реализовать средство интеграции LuNA-подпрограмм в MPI-программы в виде расширения системы LuNA.
4. Провести экспериментальные исследования полученного решения.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

Согласно поставленной цели, необходимо, обеспечить поддержку вызова LuNA-подпрограмм из MPI-программ. В мировом обществе уже накоплен опыт решения подобных задач. Возможно, удастся переиспользовать или адаптировать одно из существующих решений. Существуют разные языки, системы и средства параллельного программирования, в которых также присутствует интероперабельность в разных моделях, а именно, возможность вызова подпрограмм из MPI-программ. Проанализируем наиболее близкие.

1.1 Charm++

Charm++ [5] использует объектно-ориентированную модель : программы состоят из набора управляемых сообщениями chare-объектов, отображаемых на физические процессоры runtime системой Charm++. Объекты взаимодействуют с другими объектами, обмениваясь сообщениями и вызывая асинхронные методы входа удаленных объектов [6].

Вызов Charm++-подпрограммы из MPI-программы осуществляется на заданном коммуникаторе с помощью интерфейсных методов Charm++ путём создания прокси-объектов, инициализации сообщений и вызова методов входа прокси-объектов [7]. Передача управления от MPI к планировщику Charm++ и наоборот осуществляется пользователем явно с помощью специальных методов.

Поскольку в системе LuNA используется иная модель вычислений, данный подход не может быть применён для реализации возможности вызова LuNA-подпрограмм из MPI-программ.

1.2 Legion

Legion [8] — ориентированная на данные модель программирования. Программа Legion представляет собой дерево задач, каждая задача которого может порождать дочерние задачи и связана с логической областью - абстракцией, используемой для описания программных данных в приложениях Legion Логические области и задачи распределяются по узлам сети согласно интерфейсу сопоставления [9].

При вызове Legion-программы из MPI-программы на каждом MPI-процессе будет запущен один экземпляр задачи верхнего уровня [10]. Данные могут быть разделены между MPI и Legion путем присоединения буфера MPI в качестве локального региона Legion. Передача управления между программами может осуществляться с помощью блокирующих методов или с использованием фазового барьера с поддержкой поколений, инкапсулированных в специальные объекты.

Поскольку возможность вызова Legion-подпрограмм из MPI-программ реализована на основе таких абстракций, как логические области, дерево задач и интерфейс сопоставления, использовать данный подход для системы LuNA не представляется возможным.

1.3 X10

X10 [11] — строго типизированный, объектно-ориентированный [12] язык программирования, основанный на модели программирования APGAS [13] – модели асинхронного разделенного глобального адресного пространства. В такой модели память представляет собой глобальное адресное пространство, разделенное на части – места, каждое из которых имеет один или несколько одновременно работающих потоков управления. Программа X10 начинается с одного потока, который порождает другие потоки управления в разных местах, используя конструкции для поддержки асинхронности и переключения места выполнения.

X10 реализуется путем трансляции на язык C++ [14] или Java [15] исходного кода, который затем компилируется и выполняется с использованием существующих инструментов для конкретной платформы. Таким образом, полученная программа либо компилируется компилятором платформы C++ для создания исполняемого файла, либо компилируется в файлы классов, а затем выполняется в кластере JVM.

Именно за счет трансляции X10-программ на язык C++ или Java обеспечивается возможность вызова X10-подпрограмм из программ на этих

языках, использующих MPI. Поскольку в системе LuNA реализован другой способ компиляции программ, данный метод для достижения поставленной цели не применим.

1.4 Cilk

Модель программирования Cilk [16] основана на общей памяти и параллелизме задач. Языки Cilk/Cilk++ [17] расширяют C/C++ несколькими ключевыми словами для поддержки параллельного программирования задач. Программа пишется в семантике последовательного программирования, при этом программист определяет и описывает потенциальный параллелизм, а планировщик отображает его на реально существующую конфигурацию потоков [18]. Для работы с общей памятью используются специальные объекты – редукторы, которые реализуют механизм разрешения гонок за данными.

Поскольку система LuNA основана на иной модели вычислений, способ вызова Cilk-подпрограмм из MPI-программ не может быть использован в системе LuNA.

1.5 Выводы

Таким образом, был проведён анализ языков, систем и средств параллельного программирования, по итогу которого заключено, что для каждого средства поставленная проблема решается по-своему и ни одно из существующих решений не подходит для системы LuNA. Поэтому необходимо развивать данную работу.

2 СРЕДСТВО ИНТЕГРАЦИИ LuNA-ПРОГРАММ В MPI-ПРОГРАММЫ

MPI и система LuNA основаны на существенно разных моделях вычислений. Проанализируем эти модели в разделах 2.1 и 2.2 соответственно.

Для интеграции LuNA-подпрограмм в MPI-программы необходимо разработать модель взаимодействия программ. А также разработать и реализовать интерфейс, который это взаимодействие обеспечит.

Модель взаимодействия MPI- и LuNA- программ описана в разделе 2.4. А интерфейс приведен в разделе 2.5.

2.1 MPI-программа

MPI-программа [19] – описание множества последовательных процессов, взаимодействующих путем обмена сообщениями.

Процессы порождаются один раз, в момент запуска программы средствами среды исполнения MPI программ. Процессы, как правило, выполняются на различных узлах мультикомпьютера. Все процессы программы последовательно пронумерованы, начиная с 0. Номер процесса именуется рангом процесса.

Каждый процесс работает в своем адресном пространстве. Обмены данными между процессами осуществляются посредством явной отправки/получения сообщений. Сообщение процесса – набор данных стандартного определенного в MPI или пользовательского типа, который передается при коммуникации процессов.

Для локализации области взаимодействия процессов создаются специальные программные объекты – коммуникаторы. Один процесс может входить в разные коммуникаторы. Взаимодействия процессов проходят в рамках некоторого коммуникатора.

2.2 LuNA-программа

LuNA-программа – описание множества процессов, называемых фрагментами вычислений [20], которые не имеют побочных эффектов и могут обмениваться друг с другом данными только в начале и при завершении. Аналогом MPI сообщений в системе LuNA являются фрагменты данных,

которые могут быть посланы от одного процесса к другому, а также от одного ко многим.

LuNA-программа как описание фрагментированного алгоритма на языке LuNA [21] представляет собой последовательность объявлений именованных фрагментов кода.

Фрагмент кода (ФК) может быть одного из двух видов: атомарный или структурированный.

Атомарный фрагмент кода объявляется и вызывается в LuNA-программе, а его описание находится в подключаемой библиотеке фрагментов кода.

Структурированный фрагмент кода - это фрагментированная подпрограмма, которая может быть применена к набору фрагментов данных с получением структурированного фрагмента вычислений. Тело структурированного фрагмента кода представляет собой объявление множества имён фрагментов данных и конечного набора операторов.

Фрагментированная LuNA-подпрограмма имеет два вида входных и выходных сущностей: аргументы и фрагменты данных.

Аргументы – это параметры подпрограммы. Подпрограмма всегда имеет конечное число аргументов. Аргумент может быть константой (целочисленной, вещественной или строковой) или именем (name). Имена могут быть использованы для адресации фрагментов данных; из имён можно конструировать новые имена путем дописывания индексов. Все аргументы должны быть заданы при запуске подпрограммы.

Кроме того, в системе LuNA существует возможность передавать в подпрограмму нефункциональные аргументы. Для этого используется механизм передачи в фрагментированную LuNA-подпрограмму аргументов-констант. Данная функциональность была реализована Ивакиным Александром Олеговичем в рамках его выпускной квалификационной работы бакалавра по теме “Разработка языковых средств описания нефункциональных параметров подпрограмм и их поддержка в системе LuNA”.

В теле подпрограммы осуществляется работа с фрагментами данных (ФД) – именованными значениями единственного присваивания, адресуемыми именами (идентификаторами). Количество входных и выходных фрагментов данных потенциально бесконечно. Входные и выходные фрагменты данных могут быть созданы в процессе работы. Могут быть поданы в подпрограмму или получены из подпрограммы асинхронно в процессе ее выполнения. Но для того, чтобы их подать или получить, возможно использовать имена только из числа аргументов.

LuNA-подпрограмма существует как множество фрагментов вычислений; в начальный момент времени – как единственный фрагмент вычислений, называемый головным. Фрагмент вычислений (ФВ) — это единица вычислений, соответствующая оператору, имеющая входные и выходные параметры.

Любой фрагмент вычислений, кроме головного, порождается другим фрагментом вычислений, после чего происходит инициализация его состояния, которая заключается в передаче ему аргументов. Фрагменты вычислений должны инициализироваться в соответствии с соглашением, которому следует фрагментированная подпрограмма и которое заключается в том, что все аргументы (константы и имена) должны быть переданы в соответствии с сигнатурой фрагментированной подпрограммы.

Имена в LuNA-программе реализуются системными идентификаторами во время исполнения. Объекты, имеющие в LuNA-программе одно имя, должны иметь и один идентификатор во время исполнения. Для обеспечения такой тождественности все идентификаторы генерируются головным фрагментом вычислений уникальными в рамках LuNA-программы и передаются порожденным ФВ. За счёт этого все ФВ имеют одни глобальные идентификаторы.

Система LuNA имеет распределенное хранилище фрагментов. В нём фрагменты имеют уникальные имена – идентификаторы и расположение – локаторы. Локатор определяет, на каком узле хранится фрагмент данных или

вычислений в данный момент времени. Существуют механизмы для работы с хранилищем, которые позволяют находить фрагменты по именам и локаторам.

Программа фрагмента вычислений представляет собой цепочку C++-процедур, которые называют блоками. Наиболее близко понятию блока соответствует термин корутина с одним существенным отличием: корутины выполняются в пределах узла, а блоки могут мигрировать с узла на узел. Все блоки программы последовательно пронумерованы, начиная с 0.

Для каждого фрагмента вычислений определен стартовый блок, который запускается исполнительной системой. Выполнение LuNA-программы всегда начинается с нулевого блока. Каждый блок возвращает значение, предписывающее системе либо исполнять следующий заданный блок, либо осуществить миграцию на другой узел, либо ожидать входных фрагментов данных, либо завершить выполнение программы фрагмента вычислений.

Результирующая программа, представляющая собой последовательность блоков, генерируется в ходе компиляции LuNA-программы. Но информация о соответствии имён фрагментов кода номерам блоков оказывается утеряна. Она существуют на момент компиляция и отсутствует во время исполнения.

Фрагмент вычислений получает на вход фрагменты данных и исполняется по мере готовности данных путём запуска соответствующего блока, что приводит к выработке фрагментов вычислений и/или фрагментов данных на выход.

Исполнение LuNA программы – это исполнение всех фрагментов вычислений по мере готовности их входных фрагментов данных. При запуске LuNA-программы экземпляры runtime системы создаются по одному на каждом узле. Они вместе образуют общую распределённую сущность и работают согласованно, представляя собой единую runtime-систему.

MPI и система LuNA основаны на существенно разных моделях вычислений. Для интеграции LuNA-подпрограмм в MPI-программы необходимо

разработать модель взаимодействия программ. А также разработать и реализовать интерфейс, который это взаимодействие обеспечит.

2.3 Задачи

Для того, чтобы вызывать фрагментированные LuNA-подпрограммы из MPI-программ необходимо восстановить информацию о соответствии имен фрагментов кода номерам блоков. Необходимо иметь возможность инициализировать головной фрагмент вычислений, передавать системе LuNA и получать обратно фрагменты данных, указав их идентификатор и локатор.

Исходя из этого, были поставлены задачи разработать и реализовать:

1. Алгоритм определения номера блока по имени фрагмента кода.
2. Алгоритмы для подачи в LuNA-подпрограмму аргументов: констант и имён.
3. Алгоритм для контроля соответствия подаваемых аргументов сигнатуре фрагментированной подпрограммы.
4. Алгоритмы для обмена фрагментами данными между MPI-программой LuNA-подпрограммой.
5. Алгоритм определения локатора по ссылочному выражению, идентифицирующему фрагмент данных.
6. Алгоритмы запуска фрагментированной LuNA-подпрограммы.

Задачи 1, 2, 4, 5 являются обязательными к выполнению для обеспечения интеграции LuNA-подпрограмм в MPI-программы. Задачи 3 и 6 будут рассмотрены как возможное дальнейшие развитие работы.

2.4 Модель взаимодействия MPI- и LuNA-программ

Была предложена и проиллюстрирована на рисунке 1 следующая модель взаимодействия.

При вызове LuNA-подпрограммы из MPI-программы подпрограмма запускается на инициализированном в MPI-программе коммуникаторе. Выйти за пределы заданного коммуникатора LuNA-подпрограмма не может, но степень

параллелизма подпрограммы может повышаться за счет использования многопоточности внутри процессов.

При вызове LuNA-подпрограммы из MPI-программы экземпляры runtime системы создаются пользователем по одному на каждом MPI-процессе. Они вместе образуют общую распределённую сущность и работают согласованно, представляя собой единую runtime-систему. На этом множестве узлов подпрограмма отрабатывает единожды, используя выделенные ей ресурсы согласно описанной в ней стратегии.

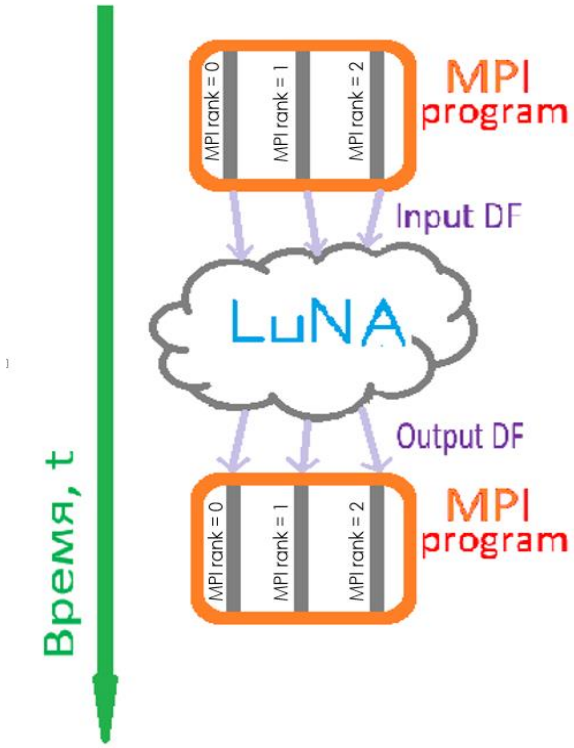

Рисунок 1 – Модель взаимодействия MPI- и LuNA-программ

Запуск LuNA-подпрограммы может осуществляться синхронно или асинхронно.

При запуске в синхронном режиме выполнение MPI-программы на процессе приостанавливается до тех пор, пока LuNA-подпрограмма не завершится и не вернет управление в MPI.

При запуске в асинхронном режиме создается отдельный поток, в котором будет происходить выполнение подпрограммы. После создания потока,

управление возвращается в MPI, и обе программы продолжают выполняться на процессе в многопоточном режиме.

При вызове LuNA-подпрограммы из MPI-программы порождается головной фрагмент вычислений, который должен быть инициализирован пользователем в соответствии с соглашением, которому следует фрагментированная программа. Для этого каждый процесс MPI-программы подает в LuNA-подпрограмму все аргументы.

Поскольку головной фрагмент вычислений должен быть инициализирован пользователем, идентификаторы используемые для адресации фрагментов данных также должны быть сгенерированы пользователем. Для достижения описанной ранее тождественности идентификаторы генерируются уникальными в рамках MPI-процесса и передаются в подпрограмму в качестве аргументов всеми процессами. Таким образом, объекты, имеющие в LuNA-программе одно имя, имеют и один идентификатор во всех MPI-процессах и во время исполнения.

Выполнение фрагментированной LuNA-подпрограммы начинается со стартового блока, который запускается исполнительной системой. Номер блока является идентификатором точки входа в подпрограмму и определяется автоматически по имени фрагмента кода.

MPI- и LuNA-программы имеют структуры данных собственного вида. Для вызова LuNA-подпрограммы из MPI-программы необходимо представить данные, имеющиеся в MPI-программе в форме, пригодной для работы с ними в LuNA-подпрограмме и передать системе, указав имя и локатор ФД. Система отработает с этими данными и по запросу, в котором также указываются имя и локатор ФД, вернёт результат.

Все фрагменты данных при подаче оных из MPI-программы в LuNA-подпрограмму отправляются на локатор, заданный в качестве параметра метода, осуществляющего подачу фрагментов данных. Аналогично для получения выходных фрагментов данных LuNA-подпрограммы они запрашиваются MPI-

программой с локатора, заданного в качестве параметра метода, осуществляющего получение фрагментов данных.

2.5 Интерфейс

Был разработан и реализован интерфейсный класс SP, который обеспечивает взаимодействие MPI- и LuNA-программ.

У класса SP есть все необходимые методы для передачи аргументов, обмена фрагментами данных и запуска LuNA-подпрограммы из MPI-программы. В конструкторе класса определяется, какая подпрограмма и на каком коммуникаторе будет запущена. Объекты класса создаются на каждом узле, при этом создадутся и экземпляры runtime системы.

Интерфейс класса SP приведен в листинге 1.

Листинг 1 – Файл sp.h

```
1 2     class SP {
3 4       public:
5 6         SP(const std::string &fp_path, const std::string &subprogram_name,
7 8             MPI_Comm comm = MPI_COMM_WORLD);
9           ~SP();
10          void push_arg(const DF &val);
11          Id push_id();
12          void push_df(Id id, const DF &val, const Locator &loc = CyclicLocator(0));
13          void request_df(Id id, DF &result, const Locator &loc = CyclicLocator(0));
14          void push_df_from(SP* source_sp, Id source_id,
15                  const Locator &source_loc = CyclicLocator(0),
16                  DF &result, Id target_id, const Locator &target_loc = CyclicLocator(0));
            int run();
            void run_acync();
            void join();
            void get_thread_object(std::thread* thread_);
          };
```

Интерфейс предоставляет пользователю следующие возможности:

- Вызывать выбранный фрагмент кода LuNA-программы – аргумент subprogram_name конструктора класса SP.
- Вызывать LuNA-подпрограмму из MPI-программы на заданном коммуникаторе – аргумент comm конструктора класса SP.
- Подавать в подпрограмму аргументы-константы и нефункциональные аргументы – метод push_arg().
- Подавать в подпрограмму аргументы-имена – метод push_id().
- Подавать входные фрагменты данных – метод push_df().
- Получать значения выходных фрагментов данных, которые были инициализированы в ходе выполнения LuNA-подпрограммы – метод request_df().
- Асинхронно передавать фрагменты данных из одной подпрограммы в другую по мере их вычисления в первой подпрограмме – метод push_df_from().
- Запускать LuNA-подпрограмму синхронно – метод run().
- Запускать LuNA-подпрограмму асинхронно – метод run_acync().
- При асинхронном запуске блокировать вызывающий поток до завершения потока выполнения подпрограммы– метод join().
- При асинхронном запуске получать объект std::thread, с которым связан поток выполнения LuNA-подпрограммы – метод get_thread_object().

2.6 Дальнейшее развитие интерфейса

Интерфейс обеспечивает все необходимые для интеграции LuNA-подпрограмм в MPI-программы возможности. Но имеются недостатки, которые могут быть устранены. Рассмотрим основные направления усовершенствования интерфейса.

2.6.1 Контроль соответствия подаваемых аргументов сигнатуре фрагментированной подпрограммы

Передача аргументов в LuNA-подпрограмму должна осуществляться в строгом их соответствии с сигнатурой подпрограммы. Поэтому следует добавить в разработанный программный модуль контроль списков аргументов. Но соответствие аргументов сигнатуре фрагмента кода возможно проверить лишь отчасти:

1. Количество подаваемых в подпрограмму аргументов-значений совпадает с количеством аргументов-значений, описанных в сигнатуре подпрограммы.
2. Количество подаваемых в подпрограмму аргументов-имён совпадает с количеством аргументов-имён, описанных в сигнатуре подпрограммы.
3. Типы подаваемых в подпрограмму аргументов-значений совпадают с типами аргументов-значений, описанных в сигнатуре подпрограммы.

Соответствие порядка подаваемых аргументов порядку, описанному в сигнатуре подпрограммы, проверить невозможно. Этот аспект должен отслеживаться пользователем вручную.

Для реализации в классе SP метода, осуществляющего проверку списков аргументов, необходимо наличие информации о сигнатуре подпрограммы. Но такой информации во время исполнения нет. Получить её можно из исходного кода LuNA-программы следующим образом. Во-первых, добавить в компилятор системы LuNA код, который будет генерировать метод, возвращающий необходимые данные. Во-вторых, в классе FP реализовать метод, который будет вызывать сгенерированный метод и возвращать его возвращаемое значение.

Далее реализовать в классе SP метод, который посредством объекта класса FP будет получать всю необходимую информацию о сигнатуре подпрограммы и сопоставлять списки подаваемых в подпрограмму аргументов с полученными данными.

Осуществлять такую проверку целесообразнее непосредственно перед запуском подпрограммы (то есть в методах SP::run() и SP::run_acync()).

2.6.2 Определение локатора по ссылочному выражению

Для того чтобы отправлять и получать фрагменты данных, не производя лишних пересылок (что снижает производительность программы), необходимо знать, на каком узле должен храниться и использоваться данный фрагмент данных в заданный момент времени. То есть необходимо знать локатор. Поэтому целесообразно реализовать автоматическое определение локатора по ссылочному выражению, идентифицирующему фрагмент данных.

Но во время исполнения нет сведений о соответствии локаторов фрагментам данных LuNA-программы. Получить их можно из исходного кода LuNA-программы следующим образом. Во-первых, добавить в компилятор системы LuNA код, который будет генерировать метод, определяющий необходимый локатор по идентификатору фрагмента данных. Во-вторых, в классе FP реализовать метод, который будет вызывать сгенерированный метод и возвращать его возвращаемое значение.

Далее в методах класса SP, предназначенных для обмена фрагментами данных (SP::push_df(), SP::request_df(), SP::push_df_from()), посредством объекта класса FP получать локатор и осуществлять подачу и получение фрагментов данных по соответствующему локатору.

3 РЕАЛИЗАЦИЯ ПОДСИСТЕМЫ ЗАПУСКА LuNA-ПОДПРОГРАММ ИЗ MPI-ПРОГРАММ

Был разработан и реализован программный модуль SP, который реализует интерфейс, описанный в разделе 3.5. Описание программного модуля SP приведено в приложении А.

Для начала в разделе 3.1 рассмотрим устройство системы LuNA, чтобы затем в разделе 3.2 рассмотреть реализацию предлагаемого решения в рамках данной системы.

В разделе 3.3 приведены примеры использования программного модуля SP.

3.1 Структура системы LuNA

Система LuNA [22] имеет структурные элементы: компилятор, исполнительную систему и интерфейс командной строки. Информация о фрагментированной подпрограмме передается в исполнительную систему через генерируемый компилятором код.

Исполнительная система содержит классы:

- DF, который представляет фрагмент данных и предоставляет методы для работы с ним.
- CF, который представляет фрагмент вычислений. Выполнение фрагмента вычислений начинается со стартового блока, которым по умолчанию значится нулевой. С помощью метода CF::create_id() головным фрагментом вычислений генерируются идентификаторы.
- Locator, который представляет локатор.
- RTS, который представляет runtime систему, метод RTS::run() инициирует запуск LuNA-программы.
- FP, через который информация о фрагментированной подпрограмме доступна исполнительной системе.
- Config, который работает с аргументами командной строки.

3.2 Подсистема запуска LuNA-подпрограмм из MPI-программ

Разработанный программный модуль SP является подсистемой запуска LuNA-подпрограмм из MPI-программ. Он распределён по ряду модулей системы LuNA и имеет следующие составные части.

1. Параметр командной строки «--sub=name», который задаёт имя LuNA-подпрограммы. И часть конструктора класса Config, осуществляющая его обработку.
2. Фрагмент компилятора, который генерирует метод get_block_by_name(), определяющий точку входа LuNA-программы по имени подпрограммы.
3. Метод get_block_by_name() класса FP, вызывающий одноимённый метод пункта 2.
4. Конструктор класса CF, в котором задаётся точка входа LuNA-программы.
5. Часть метода run класса RTS, которая отвечает за установку точки входа LuNA-программы и передачу аргументов в подпрограмму.
6. Метод create_id() класса RTS, который используется классом SP для генерации идентификаторов (аргументов-имён).
7. Класс SP, который содержит методы для запуска LuNA-подпрограммы из MPI-программы, подачи аргументов на вход подпрограммы и обмена фрагментами данных. Именно с объектами этого класса будет осуществляться работа в MPI-программе.

3.2.1 Параметр командной строки. Ключ --sub

Класс Config

Для добавления нового параметра командной строки необходимо расширить класс Config. В класс добавлен атрибут std::string subprogram_name_, который по умолчанию равен «main» и хранит имя фрагмента кода; и метод std::string &get_subprogram_name(), возвращающий значение атрибута. В конструкторе класса Config происходит обработка параметров командной

строки, в том числе нового параметра «--sub=name», где name – имя фрагмента кода. Использование ключа позволяет выполнять выбранный фрагмент кода LuNA-программы из командной строки.

3.2.2 Реализация определения точки входа для запуска выбранного фрагмента кода по его имени

Для того, чтобы вызвать выбранную LuNA-подпрограмму из MPI-программы, необходима возможность выбора точки входа в LuNA-программу, то есть возможность задавать номер блока, который будет считаться стартовым. Номер стартового блока определяется конструктором класса CF как равный 0 по умолчанию. Данный подход для решения поставленных задач не годится. Поэтому необходимо расширить сигнатуру конструктора, добавив в неё новый параметр – номер стартового блока.

Чтобы передать в конструктор класса CF нужный номер блока, необходимо реализовать алгоритм, который будет определять номер блока по имени LuNA-подпрограммы. Информация о соответствии номеров блоков фрагментам кода во время исполнения и в исходном коде LuNA-программы отсутствует. Она имеется на этапе компиляции программы. Для того, чтобы передать её от компилятора в исполнительную систему, необходимо, во-первых, в сгенерированный компилятором код добавить новый метод, который будет принимать в качестве параметра имя фрагмента кода и возвращать номер блока, в котором начинается его выполнение. Данный метод будет доступен из runtime системы, а его тело сгенерирует компилятор, который всю необходимую информацию имеет. Во-вторых, добавить в компилятор код, который будет генерировать данный метод. В-третьих, добавить в класс FP метод, который будет её вызывать.

Модификация компилятора (файл luna/scripts/fcmp2)

В компилятор был добавлен фрагмент кода, который конструирует метод int get_block_by_name(const std::string &name) в генерируемом файле test.cpp. Данный метод принимает в качестве параметра имя фрагмента кода и возвращает

номер блока, в котором начинается его выполнение. Имя фрагмента кода задаётся параметром командной строки или при создании объекта класса SP. Номер блока определяет точку входа.

Модификация класса FP

Метод int get_block_by_name(const std::string &subprogram_name), добавленный к класс FP, вызывает метод get_block_by_name() из файла test.cpp.

Модификация класса CF

В конструктор класса CF теперь передаётся параметр int first_block, по умолчанию равный 0, – номер блока, с которого начнётся выполнение LuNA-подпрограммы.

3.2.3 Класс SP

Запуск LuNA-подпрограммы осуществляется с помощью класса SP, объекты которого создаются на всех MPI-процессах, на которых она будет запущена. Именно через эти объекты будет происходить обмен аргументами и фрагментами данных.

Конструктор класса SP:

Конструктор имеет сигнатуру SP::SP(const std::string &fp_path, const std::string &subprogram_name, MPI_Comm comm) и принимает параметры:

- IN fp_path – путь к файлу build/libucodes.so, сгенерированному при компиляции LuNA-программы;
- IN subprogram_name – имя вызываемого фрагмента кода LuNA-программы;
- IN comm – коммуникатор области связи, параметр по умолчанию равный MPI_COMM_WORLD – набор MPI-процессов, на которых будет запущена LuNA-подпрограмма.

В конструкторе создаются объекты классов Config, FP, MpiComm и RTS, необходимые для работы класса SP.

Реализация передачи аргументов

Передача аргументов осуществляется методами void SP::push_arg(const DF &val) для значений и нефункциональных аргументов и Id SP::push_id() для имён.

Метод SP::push_arg(DF df) добавляет фрагмент данных df в список аргументов-значений std::vector<DF> args_ класса SP.

Метод SP::push_id() создаёт уникальное имя id с помощью метода Id RTS::create_id() и добавляет его в список аргументов-имён std::vector<Id> ids_ класса SP.

Все аргументы должны быть переданы в подпрограмму до её запуска.

При попытке подачи аргумента в выполняющуюся подпрограмму MPI-программа будет остановлена с ошибкой «Can not push argument: program is already running.». Для этого была реализована проверка статуса подпрограммы с использованием атрибута класса SP bool is_running_ - логической переменной, равной «true», если подпрограмма выполняется, и «false» иначе. Подробное описание атрибута будет приведено ниже.

Реализация передачи входных фрагментов данных

Метод void SP::push_df(Id id, const DF &val, const Locator &loc) вызывает метод void RTS::post(const Id &id, const DF &val, const Locator &loc, int req_count), который осуществляет подачу фрагмента данных в систему LuNA на локатор loc.

Реализация получения выходных фрагментов данных

Метод request_df_value(Id id, DF &result, const Locator &loc) инициализирует фрагмент данных DF result значением с именем id, которое было вычислено в ходе работы LuNA-подпрограммы и хранится на локаторе loc.

Данный метод реализован на основе метода RTS::request() с использованием лямбда-функции и оперирует фрагментами данных посредством их имён.

Реализация подачи в подпрограмму-2 в качестве входного фрагмента данных выходного фрагмента данных подпрограммы-1

Метод void SP::push_df_from(SP* source_sp, Id source_id, const Locator &source_loc, DF &result, Id target_id, const Locator &target_loc) инициализирует фрагмент данных DF result значением с именем source_id, которое было вычислено в ходе работы подпрограммы-1, и передаёт его в подпрограмму-2 по имени target_id.

Такая подача фрагментов данных осуществляется асинхронно по мере их вычисления в подпрограмме-1.

Параметры метода:

- Вызывающий объект класса SP – объект, с которым связана подпрограмма-2, в которую будет передан фрагмент данных.
- IN SP* source_sp – объект, с которым связана подпрограмма-1, в ходе работы которой будет вычислен фрагмент данных.
- IN Id source_id – имя выходного фрагмента данных, которое было передано ранее в подпрограмму-1 в качестве аргумента.
- IN const Locator &source_loc – локатор, на котором хранится фрагмент данных в подпрограмме-1.
- OUT DF &result – фрагмент данных, который будет инициализирован значением, вычисленным в подпрограмме-1.
- IN Id target_id – имя входного фрагмента данных, которое было передано ранее в подпрограмму-2 в качестве аргумента.
- IN const Locator &target_loc – локатор, на который фрагмент данных будет подан.

Метод реализован на основе метода RTS::request() с использованием лямбда-функции, в которой осуществляется вызов метода SP::push_df(Id id, const DF &val, const Locator &source_loc) для объекта класса SP, связанного с подпрограммой-2.

Реализация вызова LuNA-подпрограммы

LuNA-подпрограмма может выполняться синхронно или асинхронно, для этого используются методы SP::run() и SP::run_acync() соответственно.

Метод SP::run() вызывает метод RTS::run(), осуществляющий запуск LuNA-подпрограммы.

Метод SP::run_acync() создаёт новый поток std::thread, выполняющий статический метод SP::run_acync_(). В последнем вызывается метод RTS::run().

Метод SP::join() является надстройкой над std::thread::join() и блокирует вызывающий поток до завершения потока выполнения, связанного с вызывающим объектом.

Для случаев, которые требуют более сложной стратегии управления потоком, в классе SP имеется метод SP::get_thread_object(), который возвращает объект std::thread, с которым связан поток выполнения LuNA-подпрограммы.

bool is_running_ - логическая переменная, равная «true», если подпрограмма выполняется, и «false» иначе. Все методы, осуществляющие запуск LuNA-подпрограммы, устанавливают is_running_ = true. Методы SP::run() и SP::join() по завершении выполнения LuNA-подпрограммы устанавливают is_running_ = false.

Деструктор класса SP

Деструктор класса SP вызывает деструкторы классов Config, FP, MpiComm и RTS, объекты которых были созданы в конструкторе SP.

3.2.4 Модификация класса RTS

Модификация метода RTS::run()

Метод RTS::run() теперь принимает следующие аргументы:

- std::string &subprogram_name – имя LuNA-подпрограммы,
- std::vector<DF> args_ – список аргументов-значений LuNA-подпрограммы,
- std::vector<Id> ids_ – список аргументов-имён LuNA-подпрограммы.

Для установления точки входа добавлен вызов метода FP::get_block_by_name(), затем полученный номер блока передается в конструктор класса CF в качестве параметра.

Для передачи аргументов в подпрограмму добавлены два цикла, которые помещают аргументы-значения и аргументы-имена в std::vector<DF> args_ и std::vector<Id> ids_ класса CF соответственно.

Метод create_id()

В класс был добавлен метод Id RTS::create_id(), который генерирует уникальное в рамках MPI-процесса ссылочное выражение. Данный метод реализует модифицированный алгоритм метода CF::create_id(), который генерирует уникальный в рамках MPI-программы идентификатор, и используется методом SP::push_id().

Таким образом, при подачи аргументов-имён в LuNA-подпрограмму посредством вызова метода Id SP::push_id() для всех процессов возвращается одинаковое ссылочное выражение.

3.3 Примеры использования класса SP

В данном разделе приведены два примера вызова LuNA-подпрограмм из MPI-программ: простейший пример вызова LuNA-подпрограммы HelloWorld(); и пример вызова LuNA-подпрограммы, вычисляющей сумму элементов массива.

Файлы ucodes.cpp являются библиотеками фрагментов кода. LuNA-программы представлены в файлах test.fa, MPI-программы – в файлах test.cpp.

Руководство по использованию средства интеграции LuNA-подпрограмм в традиционные MPI-программы приведено в приложении Б.

3.3.1 Простейший пример

LuNA-программа, приведенная в листингах 2 и 3, печатает "Hello world!", она не имеет аргументов и не работает с фрагментами данных.

Листинг 2 – Файл ucodes.cpp

```
1    #include <cstdio>
```

```
2        extern "C"
3        void c_helloworld()
4        {
5            printf("Hello world!\n");
6        }
```

Листинг 3 – Файл test.fa

```
1        import c_helloworld() as hello_world;
2        sub main()
3        {
4            hello_world();
5        }
```

MPI-программа, приведенная в листинге 4, осуществляет вызов данной LuNA-подпрограммы.

Листинг 4 – Файл test.cpp

```
1        #include <mpi.h>
2        #include <cstdlib>
3        #include <unistd.h>
4        #include "sp.h"
5        int main(int argc, char** argv) {
6            int desired = MPI_THREAD_MULTIPLE, provided;
7            MPI_Init_thread(&argc, &argv, desired, &provided);
8            if (provided != desired) {
9                printf("MPI_Init_thread error\n");
10               return 1;
11           }
12           int size, rank;
13           MPI_Comm_size(MPI_COMM_WORLD, &size);
14           MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
15          std::string fp_path = "path_to_luna-program/build/libucodes.so";
16          std::string name = "main";
17          SP* sp = new SP(fp_path, name, MPI_COMM_WORLD);
18          printf("Start executing subroutine:\n");
19          sp->run();
20          printf("Finish executing subroutine.\n");
21          delete sp;
22          MPI_Finalize();
23          return 0;
24      }
```

Рассмотрим данную MPI-программу подробно.

В строках 6-11 происходит многопоточная инициализация MPI.

В строке 17 создаётся экземпляр класса SP, в конструктор в качестве параметров передаются путь к LuNA-подпрограмме, имя фрагментированной LuNA-подпрограммы и коммуникатор, на котором она будет исполняться.

В строке 19 LuNA-подпрограмма запускается синхронно.

В строке 21 вызывается деструктор класса SP, в строке 22 деинициализируется MPI.

Вывод данной программы приведен в листинге 5.

Листинг 5 – Вывод MPI-программы

```
Start executing subroutine:
Hello, world!
Finish executing subroutine.
```

3.3.2 Пример с обменом данными

LuNA-программа, приведённая в листингах 6 и 7, инициализирует массив arr и вычисляет сумму его элементов. Фрагментированная LuNA-подпрограмма sum(name arr, name res, int N) принимает в качестве аргументов имя

инициализированного массива arr, имя результата сложения res и размер массива N.

Листинг 6 – Файл ucodes.cpp

```
1    #include <cstdio>
2    #include "ucenv.h"
3    extern "C"
4    {
5        void c_init(int val, OutputDF &df)
6        {
7            df.setValue(val);
8        }
9        void c_sum(int x1, int x2, InputDF &y)
10       {
11           int y_data = x1 + x2;
12           y.setValue<int>(y_data);
13       }
14   }
```

Листинг 7 – Файл test.fa

```
1    import c_init(int, name) as init;
2    import c_sum(int, int, name) as c_sum;
3    sub sum(name arr, name res, int N)
4    {
5        df cmp;
6        init(arr[0], cmp[0]);
7        for i=1..N-1
8            c_sum(cmp[i-1], arr[i], cmp[i]);
9        init(cmp[N-1], res);
10   }
11   sub main()
12   {
```

```
13          df arr, res, N;
14          init(10, N);
15          for i=0..N-1
16              init(i, arr[i]);
17          sum(arr, res, N);
18      }
```

MPI-программа, приведенная в листинге 8, осуществляет вызов LuNA-подпрограммы, предварительно передав ей необходимые аргументы и фрагменты данных и запросив результат вычисления.

Листинг 8 – Файл test.cpp

```
1       #include <mpi.h>
2       #include <cstdlib>
3       #include <unistd.h>
4       #include "sp.h"
5       int main(int argc, char** argv) {
6           int desired=MPI_THREAD_MULTIPLE, provided;
7           MPI_Init_thread(&argc, &argv, desired, &provided);
8           if (provided!=desired) {
9               printf("MPI_Init_thread error\n");
10              return 1;
11          }
12          int size, rank;
13          MPI_Comm_size(MPI_COMM_WORLD, &size);
14          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15          std::string fp_path = "path_to_luna-program/build/libucodes.so";
16          std::string name = "sum";
17          SP* sp = new SP(fp_path, name);
18          int N = 5;
19          DF df_N;
20          df_N.setValue<int>(N);
```

```
21          sp->push_arg(df_N);
22          DF arr[N];
23          Id id_arr = sp->push_id();
24          for(int i = 0; i < N; i++) {
25             if((i % size) == rank) {
26                arr[i].setValue<int>(i);
27                printf("Process %d pushed arr[%d] = %d\n", rank, i, arr[i].getValue<int>());
28                sp->push_df(id_arr[i], arr[i]);
29             }
30          }
31          DF res;
32          Id id_res = sp->push_id();
33          sp->request_df(id_res, res);
34          sp->run();
35          if(rank == 0)
36             printf("Received res = %d\n", res.getValue<int>());
37          delete sp;
38          MPI_Finalize();
39          return 0;
40       }
```

Как и в MPI-программе из раздела 3.3.1 на каждом MPI-процессе создается по экземпляру класса SP. Затем каждый MPI-процесс подаёт в LuNA-программу все аргументы и только свою часть массива arr, запрашивает результат (который будет распечатан только нулевым процессом) и запускает подпрограмму. Рассмотрим данную MPI-программу подробно.

В строках 6-11 происходит многопоточная инициализация MPI.

В строке 17 создаётся экземпляр класса SP, в конструктор в качестве параметров передаются путь к LuNA-подпрограмме, имя фрагментированной LuNA-подпрограммы и коммуникатор, на котором она будет исполняться.

В строке 21 с помощью метода push_arg() в подпрограмму передается размер массива arr.

В строке 23 с помощью метода push_id() в подпрограмму передается имя массива arr. В строках 24-30 происходит инициализация фрагментов данных, представляющих собой элементы массива arr, и их передача в подпрограмму с помощью метода push_df().

В строке 32 с помощью метода push_id() в подпрограмму передается имя результата сложения res. В строке 33 запрашивается значение фрагмента данных с именем res.

В строке 34 LuNA-подпрограмма запускается синхронно. В строке 35-36 печатается результат выполнения LuNA-подпрограммы – сумма элементов заданного массива.

В строке 37 вызывается деструктор класса SP, в строке 38 деинициализируется MPI.

Вывод данной программы приведен в листинге 9.

Листинг 9 – Вывод MPI-программы

```
Process 0 pushed arr[0] = 0
Process 1 pushed arr[1] = 1
Process 0 pushed arr[0] = 2
Process 1 pushed arr[1] = 3
Process 0 pushed arr[0] = 4
Received res = 10
```

4 ТЕСТИРОВАНИЕ

Для проверки работоспособности программного модуля SP были реализованы MPI-программы, осуществляющие вызов LuNA-подпрограммы и проведены тесты.

Идея тестов состоит в том, чтобы применить интерфейсный класс SP для вызова LuNA-подпрограмм из MPI-программ и проверить работоспособность MPI-программ при разных параметрах и на разных конфигурациях, а также сравнить время выполнения таких программ с временем выполнения соответствующих LuNA-программ.

Тестирование проводилось на кластере НГУ [23].

4.1 Тестирование программы, выполняющей умножение матриц

Для данного теста была выбрана уже реализованная LuNA-программа, осуществляющая умножение матриц. Из нее удален код, инициализирующий массивы. Сигнатура LuNA-программы расширена для передачи в программу уже инициализированных массивов и их размеров. Теперь данная LuNA-программа может быть вызвана как подпрограмма из MPI-программы.

Реализована MPI-программа, каждый процесс которой инициализирует свои части массивов, затем вызывает LuNA-подпрограмму, получает и печатает свои части результирующего массива.

MPI-программа была выполнена на различных наборах ядер. График зависимости времени выполнения LuNA-программы и MPI-программы, вызывающей LuNA-подпрограмму, от количества ядер приведён на рисунке 2.

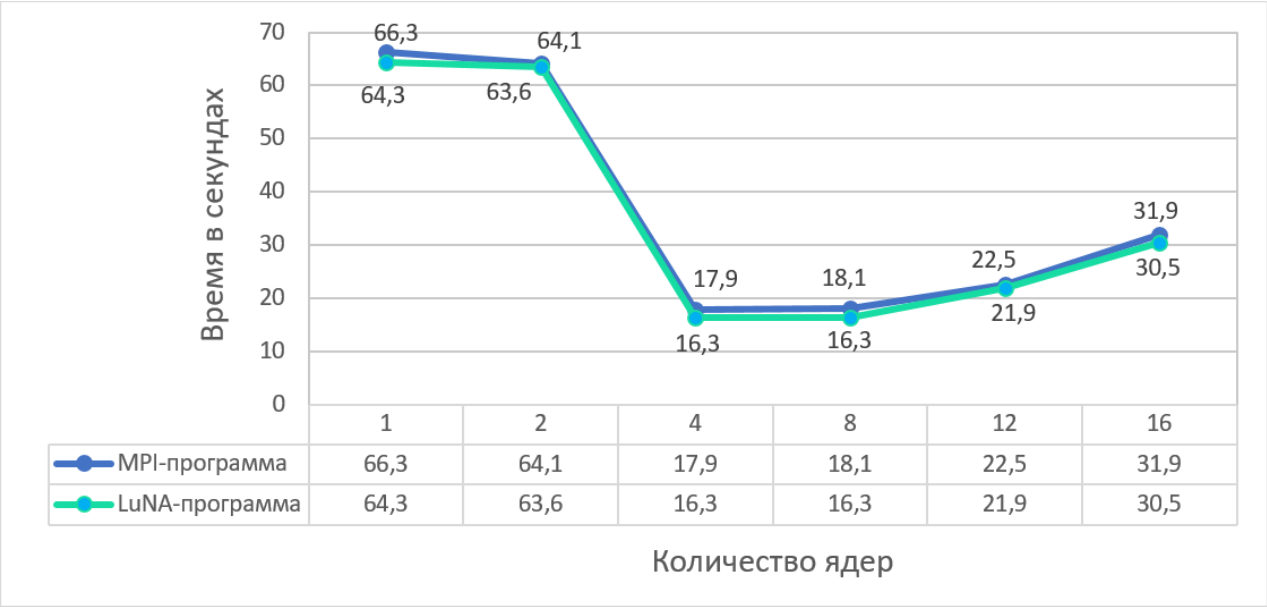

Рисунок 2 – График зависимости времени выполнения LuNA-программы и MPI-программы, вызывающей LuNA-подпрограмму, от количества ядер

Из результатов этого теста можно сделать вывод о том, что программа выполнилась без ошибок и вывела ожидаемый результат. Кроме того, время выполнения MPI-программы, вызывающей LuNA-подпрограмму, оказалось сопоставимым с временем выполнения соответствующей LuNA-программы. То есть применение разработанного программного средства не снизило эффективность выполнения программы.

4.2 Тестирование программы, решающей уравнение Пуассона

Для данного теста была выбрана уже реализованная LuNA-программа, вычисляющая решение уравнения Пуассона. Из нее удален код, инициализирующий входные массивы, и расширена сигнатура для передачи в программу уже инициализированных массивов. Теперь данная LuNA-программа может быть вызвана как подпрограмма из MPI-программы.

Реализована MPI-программа, каждый процесс которой инициализирует свои части массивов, затем вызывает LuNA-подпрограмму.

MPI-программа была выполнена на различных наборах ядер. График зависимости времени выполнения LuNA-программы и MPI-программы,

вызывающей LuNA-подпрограмму, от количества ядер приведён на рисунке 3.

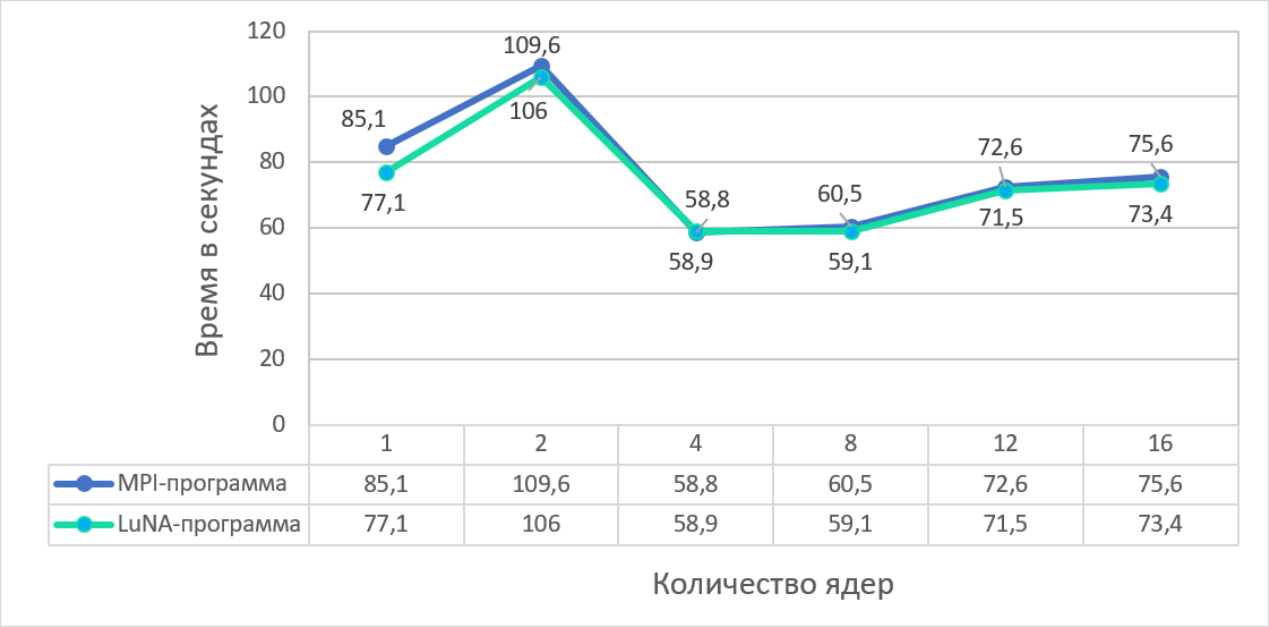

Рисунок 3 – График зависимости времени выполнения LuNA-программы и MPI-программы, вызывающей LuNA-подпрограмму, от количества ядер

Из результатов этого теста можно сделать вывод о том, что программа выполнилась без ошибок и вывела ожидаемый результат. Кроме того, время выполнения MPI-программы, вызывающей LuNA-подпрограмму, оказалось сопоставимым с временем выполнения соответствующей LuNA-программы. То есть применение разработанного программного средства не снизило эффективность выполнения программы.

4.3 Выводы

Таким образом, было проведено экспериментальное исследование, в ходе которого подсистема запуска LuNA-подпрограмм из MPI-программ показала свою работоспособность при разных параметрах и на разных конфигурациях. Кроме того, по итогам исследования было заключено, что применение разработанного программного средства не вносит существенных накладных расходов.

ЗАКЛЮЧЕНИЕ

Была обеспечена возможность интеграции LuNA-подпрограмм в традиционные параллельные программы.

В частности, была разработана модель взаимодействия MPI- и LuNA-программ. А также интерфейс, который это взаимодействие обеспечивает. И было реализовано программное средство, которое позволяет вызывать LuNA-подпрограмму из MPI-программы, передавать аргументы, осуществлять обмен данными между программами и выполнять LuNA-подпрограмму на заданном наборе узлов синхронно и асинхронно.

Работоспособность программного средства показана с помощью тестовых MPI-программ, осуществляющих вызов LuNA-подпрограмм: вычисляющей произведение матриц и решающей уравнение Пуассона.

Результаты работы были представлены на 61-й Международной научной студенческой конференции, г. Новосибирск, 2023 г. Тема доклада: “Разработка и реализация средства интеграции LuNA-подпрограмм в традиционные параллельные программы”.

Защищаемые положения:

1. Разработана модель взаимодействия MPI- и LuNA- программ.
2. Разработано и реализовано средство интеграции LuNA-подпрограмм в MPI-программы в виде расширения системы LuNA.
3. Проведено экспериментальное исследование полученного решения.

В дальнейшем планируется продолжить работу над программным модулем, обеспечивающем возможность интеграции LuNA-подпрограмм в MPI-программы. Возможными направлениями для развития являются:

1. Контроль соответствия подаваемых в подпрограмму аргументов сигнатуре LuNA-подпрограммы.
2. Автоматическое определение расположение фрагмента данных по его имени.

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

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

Капралова Регина Евгеньевна _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Крюков В. А. Разработка параллельных программ для вычислительных кластеров и сетей // Информационные технологии и вычислительные системы. – 2003. – Т. 1. – С. 2.
2. Перепёлкин В. А. Язык и система LuNA для автоматического конструирования программ численного моделирования // Проблемы информатики. - 2020. - №1. - С. 66-74.
3. Малышкин В. Э. Технология фрагментированного программирования // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. – №. 46 (305). – С. 45-55.
4. MPI Documents. Официальная документация стандарта MPI [Электронный ресурс]. URL: https://www.mpi-forum.org/docs/ (дата обращения 15.05.2023).
5. Kale L. V., Krishnan S. Charm++: a portable concurrent object oriented system based on c++ // Proceedings of OOPSLA'93. — 1993. — Vol. 28.
6. The Charm++ Parallel Programming System. Официальная документация Charm++ [Электронный ресурс]. – URL: https://charm.readthedocs.io/en/latest/charm++/manual.html#charm-mpi-interoperation (дата обращения 05.04.2023).
7. Nikhil Jain, Abhinav Bhatele, Jae-Seung Yeom, Mark F. Adams, Francesco Miniati, Chao Mei, Laxmikant V. Kale. Charm++ and MPI: Combining the Best of Both Worlds // IEEE International Parallel and Distributed Processing Symposium. - 2015.
8. M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In Supercomputing (SC), 2012.
9. Legion API Tutorial [Электронный ресурс]. – URL: https://legion.stanford.edu/tutorial/ (дата обращения: 06.04.2023).

10. Репозиторий Legion [Электронный ресурс]. URL: https://gitlab.com/StanfordLegion/legion (дата обращения: 06.04.2023).
11. The X10 programming language [Электронный ресурс]. URL: http://x10-lang.org/articles/79.html (дата обращения: 07.04.2023).
12. Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2021. — 448 с.: ил. — (Серия «Библиотека программиста»). ISBN 978-5-4461-1595-2.
13. The APGAS model [Электронный ресурс]. URL: https://x10.sourceforge.net/documentation/intro/latest/html/node1.html (дата обращения: 07.04.2023).
14. Stroustrup B. C++ Programming Language // Pearson Education. — 2013.
15. The Java Tutorials [Электронный ресурс]. – URL: https://docs.oracle.com/javase/tutorial/ (дата обращения: 01.04.2023).
16. Intel Cilk Plus [Электронный ресурс]. – URL: http://www.cilkplus.org/ (дата обращения: 01.04.2023).
17. Немнюгин, С.А.. Программирование на Intel Cilk Plus : Курс лекций / С.А. Немнюгин — Москва : Интуит НОУ, 2016. — 192 с. — URL: https://book.ru/book/918117 (дата обращения: 01.04.2023). — Текст : электронный.
18. OpenCilk [Электронный ресурс]. – URL: https://www.opencilk.org/ (дата обращения: 01.04.2023).
19. Intel® MPI Library Developer Guide for Windows* OS [Электронный ресурс]. URL: https://www.intel.com/content/www/us/en/develop/documentation/mpi-developer-guide-windows/top/analysis-and-tuning/mpi-tuning.html (дата обращения 15.05.2023).
20. 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.

21. Описание языка LuNA [Электронный ресурс] . URL: https://gitlab.ssd.sscc.ru/luna/luna5/wikis/luna_lang_v01 (дата обращения: 05.04.2023).
22. Репозиторий системы LuNA [Электронный ресурс]. URL: https://gitlab.ssd.sscc.ru/luna/luna (дата обращения: 05.04.2023).
23. Информационно-вычислительный центр Новосибирского государственного университета [Электронный ресурс]. URL: http://nusc.nsu.ru/wiki/doku.php/doc/index (дата обращения 17.05.2022).

ПРИЛОЖЕНИЕ А

Средство интеграции LuNA-подпрограмм в традиционные параллельные программы

Описание программы

Листов 16

Новосибирск 2023

АННОТАЦИЯ

В данном документе приведено описание программного модуля, называемого средством интеграции LuNA-подпрограмм в традиционные MPI-программы.

В данном документе, в разделе "Общие сведения" указаны обозначение и наименование программы, программное обеспечение, необходимое для функционирования программы и языки программирования, на которых написана программа.

В разделе "Функциональное назначение" указаны классы решаемых задач и (или) назначение программы и сведения о функциональных ограничениях на применение.

В разделе "Описание логической структуры" указаны алгоритм программы, используемые методы, структура программы с описанием функций составных частей и связи между ними, связи программы с другими программами.

. В разделе "Используемые технические средства" указаны типы электронных вычислительных машин и устройств, которые используются при работе программы.

В разделе "Вызов и загрузка" указаны способ вызова программы с соответствующего носителя данных.

В разделе "Входные данные" указаны характер, организация и предварительная подготовка входных данных.

В разделе "Выходные данные" указаны характер и организация выходных данных.

Оформление программного документа «Описание программы» произведено по требованиям ЕСПД (ГОСТ 19.101-77, ГОСТ 19.103-77, ГОСТ 19.104-78, ГОСТ 19.105-78, ГОСТ 19.106-78, ГОСТ 19.402-78, ГОСТ 19.604-78).

СОДЕРЖАНИЕ

1 ОБЩИЕ СВЕДЕНИЯ 46

2 ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ 47

3 ОПИСАНИЕ ЛОГИЧЕСКОЙ СТРУКТУРЫ 48

4 ИСПОЛЬЗУЕМЫЕ ТЕХНИЧЕСКИЕ СРЕДСТВА 55

5 ВЫЗОВ И ЗАГРУЗКА 56

6 ВХОДНЫЕ ДАННЫЕ 57

7 ВЫХОДНЫЕ ДАННЫЕ 58

1 ОБЩИЕ СВЕДЕНИЯ

Полное наименование программного модуля: “Средство интеграции LuNA-подпрограмм в традиционные параллельные программы”. Данный программный модуль является частью проекта LuNA.

Программное обеспечение необходимое для функционирования: gcc, g++, mpicxx, mpirun, openmpi, python3, LuNA.

Языки программирования, на которых написан программный модуль: C++, Python.

2 ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ

Разработанный программный модуль предназначен для интеграции LuNA-подпрограмм в MPI-программы. Она обеспечивает возможность выполнения следующих функций:

- Вызов выбранной фрагментированной LuNA-подпрограммы.
- Подача в LuNA-подпрограмму аргументов.
- Подача в LuNA-подпрограмму входных фрагментов данных.
- Получение значений выходных фрагментов данных.
- Асинхронная передача фрагментов данных из одной подпрограммы в другую.
- Запуск LuNA-подпрограммы на заданном в MPI-программе коммуникаторе.
- Синхронный запуск LuNA-подпрограммы.
- Асинхронный запуск LuNA-подпрограммы.
- Ожидание завершения потока выполнения LuNA-подпрограммы (при асинхронном запуске).
- Получение объекта, с которым связан поток выполнения LuNA-подпрограммы (при асинхронном запуске).

3 ОПИСАНИЕ ЛОГИЧЕСКОЙ СТРУКТУРЫ

Разработанный программный модуль SP является подсистемой запуска LuNA-подпрограмм из MPI-программ. Он распределён по ряду модулей системы LuNA и имеет следующие составные части.

1. Параметр командной строки «--sub=name», который задаёт имя LuNA-подпрограммы. И часть конструктора класса Config, осуществляющая его обработку.
2. Фрагмент компилятора, который генерирует метод get_block_by_name(), определяющий точку входа LuNA-программы по имени подпрограммы.
3. Метод get_block_by_name() класса FP, вызывающий одноимённый метод пункта 2.
4. Конструктор класса CF, в котором задаётся точка входа LuNA-программы.
5. Часть метода run класса RTS, которая отвечает за установку точки входа LuNA-программы и передачу аргументов в подпрограмму.
6. Метод create_id() класса RTS, который используется классом SP для генерации идентификаторов (аргументов-имён).
7. Класс SP, который содержит методы для запуска LuNA-подпрограммы из MPI-программы, подачи аргументов на вход подпрограммы и обмена фрагментами данных. Именно с объектами этого класса будет осуществляться работа в MPI-программе.

Параметр командной строки. Ключ --sub

Класс Config

Класс Config содержит атрибут std::string subprogram_name_, который по умолчанию равен «main» и хранит имя фрагмента кода; и метод std::string &get_subprogram_name(), возвращающий значение атрибута. В конструкторе класса Config происходит обработка параметров командной строки, в том числе параметра «--sub=name», где name – имя фрагмента кода. Использование ключа

позволяет выполнять выбранный фрагмент кода LuNA-программы из командной строки.

Определение точки входа для запуска выбранного фрагмента кода по его имени

Для того, чтобы вызвать выбранную LuNA-подпрограмму из MPI-программы, необходима возможность выбора точки входа в LuNA-программу, то есть возможность задавать номер блока, который будет считаться стартовым. Номер стартового блока передается в конструктор класса CF как параметр, равный 0 по умолчанию.

Чтобы передать в конструктор класса CF нужный номер блока, необходимо определить номер блока по имени LuNA-подпрограммы. Информация о соответствии номеров блоков фрагментам кода, которая имеется только на этапе компиляции программы. Для того, чтобы передать её от компилятора в исполнительную систему, часть кода компилятора генерирует метод, который принимает в качестве параметра имя фрагмента кода и возвращает номер блока, в котором начинается его выполнение. Данный метод доступен из runtime системы, а его тело генерирует компилятор, который всю необходимую информацию имеет. Затем данный метод вызывается одноименным методом класса FP.

Компилятор (файл luna/scripts/fcmp2)

Фрагмент кода компилятора конструирует метод int get_block_by_name(const std::string &name) в генерируемом файле test.cpp. Данный метод принимает в качестве параметра имя фрагмента кода и возвращает номер блока, в котором начинается его выполнение. Имя фрагмента кода задается параметром командной строки или при создании объекта класса SP. Номер блока определяет точку входа.

Метод класса FP

Метод int get_block_by_name(const std::string &subprogram_name) класса FP вызывает метод get_block_by_name() из файла test.cpp.

Конструктор класса CF

В конструктор класса CF передаётся параметр int first_block, по умолчанию равный 0, – номер блока, с которого начнётся выполнение LuNA-подпрограммы.

Класс SP

Запуск LuNA-подпрограммы осуществляется с помощью класса SP, объекты которого создаются на всех MPI-процессах, на которых она будет запущена. Именно через эти объекты будет происходить обмен аргументами и фрагментами данных.

Конструктор класса SP:

Конструктор класса имеет сигнатуру SP::SP(const std::string &fp_path, const std::string &subprogram_name, MPI_Comm comm) и принимает параметры:

- IN fp_path – путь к файлу build/libucodes.so, сгенерированному при компиляции LuNA-программы;
- IN subprogram_name – имя вызываемого фрагмента кода LuNA-программы – в контексте примера “sum”;
- IN comm – коммуникатор области связи, параметр по умолчанию равный MPI_COMM_WORLD – набор MPI-процессов, на которых будет запущена LuNA-подпрограмма.

В конструкторе создаются объекты классов Config, FP, MpiComm и RTS, необходимые для работы класса SP.

Реализация передачи аргументов

Передача аргументов осуществляется методами void SP::push_arg(const DF &val) для значений и нефункциональных аргументов и Id SP::push_id() для имён.

Метод SP::push_arg(DF df) добавляет фрагмент данных df в список аргументов-значений std::vector<DF> args_ класса SP.

Метод SP::push_id() создаёт уникальное имя id с помощью метода Id RTS::create_id() и добавляет его в список аргументов-имён std::vector<Id> ids_ класса SP.

Все аргументы должны быть переданы в подпрограмму до её запуска.

При попытке подачи аргумента в выполняющуюся подпрограмму MPI-программа будет остановлена с ошибкой «Can not push argument: program is already running.». Для этого была реализована проверка статуса подпрограммы с использованием атрибута класса SP bool is_running_ - логической переменной, равной «true», если подпрограмма выполняется, и «false» иначе. Подробное описание атрибута будет приведено ниже.

Реализация передачи входных фрагментов данных

Метод void SP::push_df(Id id, const DF &val, const Locator &loc) вызывает метод void RTS::post(const Id &id, const DF &val, const Locator &loc, int req_count), который осуществляет подачу фрагмента данных в систему LuNA на локатор loc.

Реализация получения выходных фрагментов данных

Метод request_df_value(Id id, DF &result, const Locator &loc) инициализирует фрагмент данных DF result значением с именем id, которое было вычислено в ходе работы LuNA-подпрограммы и хранится на локаторе loc.

Данный метод реализован на основе метода RTS::request() с использованием лямбда-функции и оперирует фрагментами данных посредством их имён.

Реализация подачи в подпрограмму-2 в качестве входного фрагмента данных выходного фрагмента данных подпрограммы-1

Метод void SP::push_df_from(SP* source_sp, Id source_id, const Locator &source_loc, DF &result, Id target_id, const Locator &target_loc) инициализирует фрагмент данных DF result значением с именем source_id, которое было вычислено в ходе работы подпрограммы-1, и передаёт его в подпрограмму-2 по имени target_id.

Такая подача фрагментов данных осуществляется асинхронно по мере их вычисления в подпрограмме-1.

Параметры метода:

- Вызывающий объект класса SP – объект, с которым связана подпрограмма-2, в которую будет передан фрагмент данных.
- IN SP* source_sp – объект, с которым связана подпрограмма-1, в ходе работы которой будет вычислен фрагмент данных.
- IN Id source_id – имя выходного фрагмента данных, которое было передано ранее в подпрограмму-1 в качестве аргумента.
- IN const Locator &source_loc – локатор, на котором хранится фрагмент данных в подпрограмме-1.
- OUT DF &result – фрагмент данных, который будет инициализирован значением, вычисленным в подпрограмме-1.
- IN Id target_id – имя входного фрагмента данных, которое было передано ранее в подпрограмму-2 в качестве аргумента.
- IN const Locator &target_loc – локатор, на который фрагмент данных будет подан.

Метод реализован на основе метода RTS::request() с использованием лямбда-функции, в которой осуществляется вызов метода SP::push_df(Id id, const DF &val) для объекта класса SP, связанного с подпрограммой-2.

Реализация вызова LuNA-подпрограммы

LuNA-подпрограмма может выполняться синхронно или асинхронно, для этого используются методы SP::run() и SP::run_acync() соответственно.

Метод SP::run() вызывает метод RTS::run(), осуществляющий запуск LuNA-подпрограммы.

Метод SP::run_acync() создаёт новый поток std::thread, выполняющий статический метод SP::run_acync_(). В последнем вызывается метод RTS::run().

Метод SP::join() является надстройкой над std::thread::join() и блокирует вызывающий поток до завершения потока выполнения, связанного с вызывающим объектом.

Для случаев, которые требуют более сложной стратегии управления потоком, в классе SP имеется метод SP::get_thread_object(), который возвращает объект std::thread, с которым связан поток выполнения LuNA-подпрограммы.

bool is_running_ - логическая переменная, равная «true», если подпрограмма выполняется, и «false» иначе. Все методы, осуществляющие запуск LuNA-подпрограммы, устанавливают is_running_ = true. Методы SP::run() и SP::join() по завершении выполнения LuNA-подпрограммы устанавливают is_running_ = false.

Деструктор класса SP

Деструктор класса SP вызывает деструкторы классов Config, FP, MpiComm и RTS, объекты которых были созданы в конструкторе SP.

Класс RTS

Метода RTS::run()

Метод RTS::run() принимает следующие аргументы:

- std::string &subprogram_name – имя LuNA-подпрограммы,
- std::vector<DF> args_ – список аргументов-значений LuNA-подпрограммы,
- std::vector<Id> ids_ – список аргументов-имён LuNA-подпрограммы.

Для установления точки входа осуществляется вызов метода FP::get_block_by_name(), затем полученный номер блока передаётся в конструктор класса CF в качестве параметра.

Передача аргументов в подпрограмму осуществляется в двух циклах, которые помещают аргументы-значения и аргументы-имена в std::vector<DF> args_ и std::vector<Id> ids_ класса CF соответственно.

Метод create_id()

Метод Id RTS::create_id() генерирует уникальное в рамках MPI-процесса ссылочное выражение. Данный метод реализует модифицированный алгоритм метода CF::create_id(), который генерирует уникальный в рамках MPI-программы идентификатор, и используется методом SP::push_id().

Таким образом, при подачи аргументов-имён в LuNA-подпрограмму посредством вызова метода Id SP::push_id() для всех процессов возвращается одинаковое ссылочное выражение.

4 ИСПОЛЬЗУЕМЫЕ ТЕХНИЧЕСКИЕ СРЕДСТВА

Программа предназначена для использования на персональном компьютере или кластере.

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 1.

Таблица 1 – Требования к параметрам устройства.

Характеристика	**Минимальные**	**Рекомендуемые**
Архитектура процессора	x86	x86
Количество процессоров	1	2
Количество ядер	2	8
Частота процессора	1 ГГц	2,5 ГГц
Кэш процессора	1 МБ	8 МБ
Оперативная память	1 ГБ	16 ГБ
Доступное место в хранилище	100 МБ	1 ГБ

5 ВЫЗОВ И ЗАГРУЗКА

Данный программный модуль является составной частью проекта LuNA, Загрузка данного программного модуля осуществляется посредством загрузки проекта LuNA.

Для применения данного модуля для написания собственных программ необходимо подключить заголовочный файл sp.h и инстанцировать класс SP.

6 ВХОДНЫЕ ДАННЫЕ

Входными данными программы являются имя фрагментированной LuNA-подпрограммы и путь к исполняемому файлу LuNA-программы в текстовом формате, а также аргументы и входные фрагменты данных LuNA-подпрограммы -- текстовые и числовые значения в формате DF и идентификаторы в формате ID.

Все аргументы фрагментированной LuNA-подпрограммы должны быть переданы в подпрограмму до её запуска. При попытке подать аргумент в выполняющуюся LuNA-подпрограмму, выдаётся сообщение “Can not push argument: program is already running.” или “Can not push id: program is already running.” для аргументов-констант и аргументов-имен соответственно. При этом выполнение программы немедленно прекращается.

7 ВЫХОДНЫЕ ДАННЫЕ

Выходными данными программы являются идентификаторы в формате ID, выходные фрагменты данных LuNA-подпрограммы - текстовые и числовые значения в формате DF; и объекты std::thread, связанные с потоком выполнения LuNA-подпрограммы.

ПРИЛОЖЕНИЕ Б

Средство интеграции LuNA-подпрограмм в традиционные параллельные программы

Руководство программиста

Листов 12

Новосибирск 2023

АННОТАЦИЯ

В данном документе приведено руководство программиста по использованию средства интеграции LuNA-подпрограмм в традиционные MPI-программы.

В данном документе, в разделе «Назначение и условия применения программы» указаны назначение и функции, выполняемые программой, условия, необходимые для выполнения программы.

В разделе «Характеристика программы» приведено описание основных характеристик и особенностей программы.

В разделе «Сообщения» указаны тексты сообщений, выдаваемых программисту или оператору в ходе выполнения программы, описание их содержания и действий, которые необходимо предпринять по этим сообщениям.

Оформление программного документа «Руководство программиста» произведено по требованиям ЕСПД (ГОСТ 19.101-77, ГОСТ 19.103-77, ГОСТ 19.104-78, ГОСТ 19.105-78, ГОСТ 19.106-78, ГОСТ 19.504-79, ГОСТ 19.604-78).

СОДЕРЖАНИЕ

1 НАЗНАЧЕНИЕ И УСЛОВИЯ ПРИМЕНЕНИЯ ПРОГРАММ 62

1.1 Назначение программы 62

1.2 Функции, выполняемые программой 62

1.3 Условия выполнения программы 62

2. ХАРАКТЕРИСТИКА ПРОГРАММЫ 64

3. ОБРАЩЕНИЕ К ПРОГРАММЕ 65

3.1 Компиляция LuNA-программы 65

3.2 Создание и удаление экземпляров класса SP 65

3.3 Передача аргументов 65

3.4 Передача фрагментов данных 66

3.5 Выполнение LuNA-подпрограммы 67

3.6 Компиляция и запуск полученной MPI-программы 68

4. ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ 69

5. СООБЩЕНИЯ 70

1 НАЗНАЧЕНИЕ И УСЛОВИЯ ПРИМЕНЕНИЯ ПРОГРАММ

1.1 Назначение программы

Разработанная программа предназначена для вызова LuNA-подпрограммы из MPI-программы, обмена данными между программами и управлением потоком выполнения Luna-подпрограммы.

1.2 Функции, выполняемые программой

Программа обеспечивает возможность выполнения перечисленных ниже функций:

- Вызов выбранной фрагментированной LuNA-подпрограммы.
- Подача в LuNA-подпрограмму аргументов.
- Подача в LuNA-подпрограмму входных фрагментов данных.
- Получение значений выходных фрагментов данных.
- Асинхронная передача фрагментов данных из одной подпрограммы в другую.
- Запуск LuNA-подпрограммы на заданном в MPI-программе коммуникаторе.
- Синхронный запуск LuNA-подпрограммы.
- Асинхронный запуск LuNA-подпрограммы.
- Ожидание завершения потока выполнения LuNA-подпрограммы (при асинхронном запуске).
- Получение объекта, с которым связан поток выполнения LuNA-подпрограммы (при асинхронном запуске).

1.3 Условия выполнения программы

Программа предназначена для использования на персональном компьютере или кластере.

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 2.

Необходимое программное обеспечение: gcc, g++, mpicxx, mpirun, openmpi, python3, LuNA.

Таблица 2 – Требования к параметрам устройства.

Характеристика	**Минимальные**	**Рекомендуемые**
Архитектура процессора	x86	x86
Количество процессоров	1	2
Количество ядер	2	8
Частота процессора	1 ГГц	2,5 ГГц
Кэш процессора	1 МБ	8 МБ
Оперативная память	1 ГБ	16 ГБ
Доступное место в хранилище	100 МБ	1 ГБ

2. ХАРАКТЕРИСТИКА ПРОГРАММЫ

Время вызова LuNA-подпрограммы не превышает 1 секунды. Время обмена готовыми данными между вызывающей и вызываемой программами не превышает 0,5 секунды.

Программа запрещает передачу в LuNA-подпрограмму аргументов после её запуска. Контроль правильности выполнения LuNA-подпрограммы с переданными ей данными осуществляется методами других компонентов системы LuNA.

Время восстановления после отказа соответствует времени остановки исполнения программы и ее повторного запуска, что не превышает 20 секунд.

3. ОБРАЩЕНИЕ К ПРОГРАММЕ

3.1 Компиляция LuNA-программы

Необходимо скомпилировать LuNA-программу с сохранением файлов компиляции, используя команду luna *--build-dir=build --compile-only <luna-program>.fa*. В результате этого появится папка build, в которой содержится файл libucodes.so, необходимый для дальнейшей работы.

3.2 Создание и удаление экземпляров класса SP

Запуск LuNA-подпрограммы из MPI-программы осуществляется с помощью класса SP, объекты которого создаются на всех MPI-процессах, на которых она будет запущена. Конструктор класса имеет сигнатуру SP::SP(const std::string &fp_path, const std::string &subprogram_name, MPI_Comm comm = MPI_COMM_WORLD) и принимает следующие параметры:

- fp_path – путь к файлу build/libucodes.so, сгенерированному при компиляции LuNA-программы;
- subprogram_name – имя вызываемого фрагмента кода LuNA-программы;
- comm – коммуникатор области связи – набор MPI-процессов, на которых будет запущена LuNA-подпрограмма.

Именно через эти объекты будет происходить передача аргументов и обмен фрагментами данных.

После завершения работы с подпрограммой необходимо вызвать деструктор для объектов класса SP, который имеет сигнатуру SP::~SP().

3.3 Передача аргументов

Аргументы (– параметры фрагментированной LuNA-подпрограммы, указанные в круглых скобках) должны быть заданы перед запуском подпрограммы.

Аргументы-константы и нефункциональные аргументы подаются в подпрограмму с помощью метода void SP::push_arg(const DF &val), который принимает параметр val – инициализированный фрагмент данных.

Аргументы-имена генерируются и передаются в подпрограмму с помощью метода Id SP::push_id(), который возвращает сгенерированное имя.

Все вызовы методов push_arg() и push_id() для аргументов должны осуществляться строго в том порядке, в котором они указаны в сигнатуре фрагментированной LuNA-подпрограммы.

3.4 Передача фрагментов данных

Входные и выходные фрагменты данных могут быть поданы в подпрограмму или получены из подпрограммы асинхронно в процессе её выполнения.

Подача фрагментов данных в фрагментированную LuNA-подпрограмму осуществляется с помощью метода void SP::push_df(Id id, const DF &val, const Locator &loc), который принимает следующие параметры:

- id – имя (идентификатор) фрагмента данных;
- val – инициализированный фрагмент данных;
- loc – локатор, на который будет подан фрагмент данных.

Получение значений фрагментов данных, которые были инициализированы в ходе выполнения LuNA-подпрограммы, осуществляется с помощью метода void SP::request_df(Id id, DF &result, const Locator &loc), который принимает параметры:

- id – имя (идентификатор) фрагмента данных;
- result – фрагмент данных, который будет инициализирован полученным значением;
- loc – локатор, на котором хранится фрагмент данных.

Инициализация фрагмента данных result произойдёт не сразу по вызове метода, а только после того, как это значение будет вычислено в подпрограмме.

Передача фрагментов данных из одной подпрограммы в другую (целевую) осуществляется с помощью метода void SP::push_df_from(SP* source_sp, Id

source_id, const Locator &source_loc, DF &result, Id target_id, const Locator &target_loc), который принимает параметры:

- вызывающий объект класса SP – объект, с которым связана целевая подпрограмма, в которую будет передан фрагмент данных:
- source_sp – объект, с которым связана подпрограмма, в ходе работы которой вычисляется фрагмент данных:
- source_id – имя выходного фрагмента данных, которое было передано ранее в первую подпрограмму в качестве аргумента:
- source_loc – локатор, на котором хранится фрагмент данных в первой подпрограмме.
- result – фрагмент данных, который будет инициализирован значением, вычисленным в первой подпрограмме:
- target_id – имя входного фрагмента данных, которое было передано в целевую подпрограмму ранее в качестве аргумента.
- target_loc – локатор, на который фрагмент данных будет подан.

Такая подача фрагментов данных осуществляется асинхронно по мере их вычисления в первой подпрограмме.

3.5 Выполнение LuNA-подпрограммы

Выполнение LuNA-подпрограммы может быть синхронным или асинхронным.

Для запуска подпрограммы в текущем потоке или асинхронно используются методы int SP::run() или void SP::run_acync() соответственно:

При запуске в асинхронном режиме создаётся отдельный поток, в котором будет происходить выполнение подпрограммы. Метод void SP::join() блокирует вызывающий поток до завершения потока выполнения, связанного с вызывающим объектом.

Для случаев, которые требуют более сложной стратегии управления потоком, в классе SP имеется метод void SP::get_thread_object(std::thread*

thread_), который инициализирует параметр thread_ объектом std::thread, с которым связан поток выполнения LuNA-подпрограммы.

3.6 Компиляция и запуск полученной MPI-программы

Для компиляции и выполнения MPI-программы используются команды:

mpicxx -lrts -L<path_to_luna>/lib <mpi-program>.cpp -o <mpi-program>.exe

mpirun -np <n> ./<mpi-program>.exe

4. ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ

Входными данными программы являются имя фрагментированной LuNA-подпрограммы и путь к исполняемому файлу LuNA-программы в текстовом формате, а также аргументы и входные фрагменты данных LuNA-подпрограммы -- текстовые и числовые значения в формате DF и идентификаторы в формате ID.

Выходными данными программы являются идентификаторы в формате ID, выходные фрагменты данных LuNA-подпрограммы - текстовые и числовые значения в формате DF; и объекты std::thread, связанные с потоком выполнения LuNA-подпрограммы.

5. СООБЩЕНИЯ

В ходе выполнения программы могут быть выданы два вида сообщений.

Сообщение “Can not push argument: program is already running.” выдается при попытке подать аргумент-константу в выполняющуюся LuNA-подпрограмму. При этом выполнение программы немедленно прекращается.

Сообщение “Can not push id: program is already running.” выдается при попытке подать аргумент-имя в выполняющуюся LuNA-подпрограмму. При этом выполнение программы немедленно прекращается.