МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Параллельные вычислительные технологии
(полное название кафедры)

Анастасия Александровна Ткачёва
(И., О., фамилия студента – автора работы)

Оптимизация исполнения фрагментированных программ с использованием управляющих структур

(полное название темы магистерской диссертации)

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
по направлению высшего профессионального образования

010400.68 – Прикладная математика и информатика
(код и наименование направления подготовки магистра)

факультет прикладной математики и информатики
(факультет)

Тема диссертации утверждена приказом по НГТУ № 1538/2 от «26» марта 2013 г.

Руководитель

Маркова В.П.
(фамилия, И., О.)

к.т.н., доцент
(уч. степень, уч. звание)

Новосибирск, 2013 г.

Аннотация

ОПТИМИЗАЦИЯ ИСПОЛНЕНИЕ ФРАГМЕНТИРОВАННОЙ ПРОГРАММЫ С ИСПОЛЬЗОВАНИЕМ УПРАВЛЯЮЩИХ СТРУКТУР

Количество страниц 49, рисунков 13, 1 таблица.

В работе предлагается модель представления фрагментированного алгоритма на основе сетей Петри, включающая средства для задания императивного управления, называемая RuSh-программа. Предложенная модель реализована в виде программного модуля RuSh. Приводится описание разработанного и реализованного алгоритма миграции частей RuSh-программы между узлами вычислителя, а также доказательство его корректности.

Были разработаны RuSh-программы для следующих численных задач: решение уравнения Пуассона методом Якоби, моделирование самогравитирующегося вещества методом «частиц в ячейках», реакция окисления *CO* на поверхности палладия блочно-синхронным клеточным автоматом. Проведено исследование производительности мультикомпьютера при реализации этих RuSh-программ и сделан сравнительный анализ производительности RuSh-программ с аналогичными MPI-программами. Сделаны выводы о применимости предложенной модели при реализации описанных численных алгоритмов и алгоритмов близких к ним по структуре вычислений.

Оглавление

Введение

Развитие параллельных вычислительных технологий позволяет решать задачи с большим объемом данных и вычислений, которые раньше невозможно было реализовать на последовательных компьютерах. Но возможности параллельных вычислений непосредственно связаны со сложностями параллельного программирования. Параллельной программе приходится управлять во времени системами, достигающими тысяч процессоров и терабайт оперативной памяти.

Можно выделить два способа описания алгоритма – декларативный и императивный. В императивном описании алгоритм представляется в виде набора действий (инструкций), которые должны быть выполнены, т.е. описывается решение задачи в терминах, понятных компьютеру. Главным плюсом такого описания является то, что накладные расходы при реализации алгоритма, описанного императивно, как правило, несущественны. В декларативном описании программа представляет собой постановку задачи, т.е. спецификацию того, что требуется получить в качестве результата. Главным плюсом такого описания является возможность создания программы в терминах близких к предметной области, а это означает, что данной системой могут пользоваться специалисты в этой предметной области. При этом декларативная программа, как правило, имеет множество вариантов исполнения, которые обладают различными характеристиками, такими как время выполнения и потребление памяти. В общем случае свойства конкретного варианта исполнения автоматически сложно предсказуемы и выбор наилучшего (или близкого к нему) способа исполнения затруднителен.

Возникает проблема, как сочетать наглядность декларативного описания и эффективность исполнения на компьютере, для этого в ряде языков программирования и систем, использующих декларативное представление, таких как LISP [1], Prolog [2] и т.д., вводятся некоторые дополнительные императивные средства, которые позволяют пользователю влиять на поведение программы. Здесь и далее под поведением программы будет пониматься множество возможных вариантов исполнения программы. Предоставляемые пользователю средства могут отличаться в разных языках или системах. Например, может быть явное задание порядка выполнения операций или определение частичного порядка выполнения на некотором множестве операций.

В рамках настоящей работы требовалось разработать аналогичные средства для системы фрагментированного программирования LuNA. Фрагментированное программирование – это технология параллельного программирования, предназначенная для реализации больших численных моделей для суперкомпьютеров. В ИВМиМГ СО

РАН в Лаборатории синтеза параллельного программ разрабатывается язык и система фрагментированного программирования Language for Numerical Algorithm (LuNA) [3]. Во фрагментированном программировании программа делится на отдельные фрагменты, и такое представление сохраняется в течение всего исполнения программы. Система LuNA использует декларативный способ описания алгоритма. Проблема выбора хорошего сценария из поведения прикладной программы во фрагментированном программировании сводится к выбору порядка исполнения операций и распределению потенциально бесконечного количества операций и переменных на ограниченное количество ресурсов суперкомпьютера (процессорные элементы и память).

Эта задача является труднорешаемой, и, в общем случае, её можно решить только полным перебором вариантов исполнения. Для того чтобы уменьшить сложность решаемой задачи можно выделить два подхода: сузить круг решаемых задач или ввести некоторые дополнительные императивные средства.

При этом эти средства должны удовлетворять следующим требованиям:

- позволять описывать желаемый порядок исполнений операций и распределение ресурсов для некоторого достаточно широкого класса численных алгоритмов;
- представление программы должно позволять эффективное исполнение на суперкомпьютере.

Целью магистерской работы было разработать такие средства для системы фрагментированного программирования LuNA.

Научной новизной данной работы является предложенная модель управления вычислениями во фрагментированных программах на основе сетей Петри, а также алгоритмы для её реализации.

Глава 1. Обзор языков и систем программирования, использующих декларативное представление алгоритма

Целью данного обзора было рассмотреть, какие дополнительные императивные средства, позволяющие влиять на поведение программы, введены в существующие декларативные языки и системы параллельного программирования.

1.1. Язык LISP

LISP (LISt Processing language – «язык обработки списков») – семейство языков программирования, программы и данные в которых представляются системами линейных списков символов [1]. Традиционный LISP имеет динамическую систему типов. Большим достоинством LISP является его функциональная направленность, т. е. программирование ведется в терминах функций. Причем функция понимается, как правило, сопоставляющее элементам некоторого класса соответствующие элементы другого класса.

В качестве дополнительных императивных средств для управления поведением программы в язык были введены специальные операторы (IF, WHEN, LOOP и т.д.), которые сильно облегчают жизнь компилятору или интерпретатору. Так же LISP включает развитую систему LISP-макросов, позволяющую во многом расширить синтаксис языка за счет ввода в неё собственных операторов, в том числе, есть возможность ввести новые управляющие операторы.

Таким образом, в языке LISP есть императивные средства, которые позволяют оптимизировать исполнение программ, и они играют важную роль в языке.

1.2. Язык Prolog

Prolog – язык и система логического программирования, основанные на языке предикатов математической логики дизъюнктов Хорна, представляющей собой подмножество логики предикатов первого порядка [2].

Язык включает специальный механизм «отсечение». Отсечение позволяет указать, какие из сделанных ранее выборов не следует пересматривать при возврате по цепочке согласованных целевых утверждений.

Существуют две причины, побуждающие включать в программу такие указания:

• Программа будет выполняться быстрее, так как не будет тратиться время на попытки найти новые сопоставления для целей, о которых заранее известно, что они не внесут более ничего нового в решение.

• Программа может занимать меньше места в памяти ЭВМ, так как отсутствие необходимости запоминать точки возврата для последующего анализа позволяет более экономно использовать память.

Таким образом, в язык Prolog в качестве дополнительного императивного средства было введено отсечение. В тоже время использование отсечения позволяет, во-первых, уменьшить объем вычислений, а во-вторых, уменьшить количество используемой памяти.

1.3. Система программирования SMP Superscalar

Система программирования SMP Superscalar [4] позволяет создавать параллельные асинхронные программы для систем с общей памятью. Она включает расширение языка Си, компилятор и runtime-систему. Система отличается простотой использования (расширение включает всего несколько директив компилятора).

В основе используемой модели программирования лежит представление программы в виде ориентированного бесконтурного графа (DAG – directed acyclic graph). Узлы графа представляют собой фрагменты вычислений, дуги – информационные зависимости, каждой из которых соответствует фрагмент данных (область памяти). Граф формируется динамически в процессе последовательного исполнения основной программы, причем зависимости по данным выявляются автоматически, а ложные зависимости устраняются с помощью механизма переименования. Одновременно с выполнением основной программы runtime-система осуществляет асинхронное исполнение фрагментов вычислений в графе с учетом зависимостей.

Эту же модель представления и исполнения программы используют системы программирования GRID Superscalar [5] для распределенных систем и Cell Superscalar для процессоров Cell [6]. Для пользователя эта модель удобна тем, что не надо отображать программу на процессорные элементы, это делается автоматически путём построения графа программы и его отображению на процессорные элементы.

В то же время такой подход исключает возможность статического планирования вычислений и оставляет пользователю мало средств управлением вычислениями. Единственным средством позволяющим влиять на поведение программы является возможность задавать повышенный приоритет некоторым фрагментам вычисления.

Надо отметить, что задание приоритетов не позволяет описывать сложное поведение и с увеличением размера задачи теряется читабельность и увеличивается вероятность ошибок управления.

1.4. Система программирования Charm++

Система программирования Charm++ [7] предназначена для создания асинхронных параллельных программ для систем с общей и распределенной памятью (с поддержкой процессоров Cell и GPU). Она включает язык (расширение языков C++, Fortran 90), компилятор, runtime-систему, а также множество подключаемых библиотечных модулей, реализующих системные функции, такие как различные стратегии коллективных обменов данными (рассылка, редукция), алгоритмы динамической балансировки загрузки, работа с контрольными точками.

Программа на Charm++ представляет собой распределенное множество объектов (инкапсулирующих в себе и данные, и код), которые взаимодействуют между собой посредством передачи сообщений. В Charm++ отсутствует статическое планирование вычислений. Поэтому способы организации управления, не предусмотренные runtime-системой и библиотеками, должны быть зафиксированы в программе, что уменьшает ее переносимость. Указанные проблемы частично решены созданием большого числа специализированных библиотек и программных систем на основе Charm++, предназначенных для решения отдельных классов задач.

При этом система Charm++ делает автоматическую начальную настойку на ресурсы и есть возможность динамической балансировки загрузки процессоров, используя ограниченный набор стандартных алгоритмов управлением поведением. В качестве дополнительных средств задания управления в Charm++ предусмотрены встроенные в систему инструменты позволяющие задавать свой вариант алгоритма для динамической балансировки загрузки процессоров. Использования подобных инструментов связано со сложностями, т.к. программист должен сам разработать некоторый алгоритм динамической балансировки и реализовать его для своей программы, что есть не самая тривиальная задача.

1.5. Система программирования OpenTS

OpenTS [8] – это система параллельного программирования для широкого спектра параллельных архитектур: многоядерные процессоры, SMP-установки, кластерные и распределённые вычислительные системы. Разработка этой системы ведется в Институте программных систем им. А.К.Айламазяна РАН.

Параллельные реализации задач численного моделирования в системе OpenTS отличаются компактностью и простотой в сравнении с подобными MPI-реализациями, и при этом показывают эффективность распараллеливания такую же или сравнимую [9].

В качестве основной парадигмы рассматривается функциональное программирование. Программа представляет собой набор чистых функций. Каждая функция может иметь несколько аргументов и несколько результатов. В то же время тела функций могут быть описаны в императивном стиле (на языках типа FORTRAN, C и т. п.). Важно только, чтобы: всю информацию извне функция получала только через свои аргументы; вся информация из функции передавалась только через явно описанные аргументы. OpenTS реализует автоматическое динамическое распараллеливание программ. Это означает, что многие аспекты организации параллельного счета (распределение задач по узлам кластера, синхронизация процессов, организация пересылки данных) выполняются не программистом, а самой системой. Во многих случаях Т-системе удается удачно организовать все эти виды работ и получить хороший уровень распараллеливания программ. Однако было бы ошибкой считать, что в Т-системе «автоматизированы» все аспекты организации параллельного счета. В первую очередь при реализации программ для Т-системы программист обязан адекватно изложить алгоритм в функциональном стиле — описать программу в виде набора «чистых» Т- функций. Кроме этого, он должен стремиться выбрать оптимальный размер гранулы параллелизма,— то есть оптимально подобрать среднюю вычислительную сложность Т- функций. Фрагменты должны быть: - достаточно крупными, чтобы их вычислительная сложность превышала бы накладные расходы на организацию их выполнения; достаточно мелкими, чтобы в процессе выполнения можно было бы ожидать возникновения такого количества таких фрагментов, которое будет достаточным для загрузки всех узлов суперкомпьютера.

Задание размера гранулы параллелизма является способом, которым можно влиять на поведения программы для параллельной системы, позволяет обеспечивать переносимость прикладной программы в зависимости от параметров доступных ресурсов для вычисления (таких как размер КЭШа, пропускная способность шины).

1.6. Результаты проведенного обзора

Проведенный обзор языков и систем программирования показывает, что для управления поведением программы в эти системы и языки введены некоторые дополнительные императивные средства. При этом выбор тех или иных средств управления зачастую связан с особенностями организации исполнения прикладного алгоритма.

Глава 2. Теоретическая часть

2.1. Модель фрагментированного алгоритма в системе LuNA

Фрагментированный алгоритм (ФА) представляет собой множество фрагментов кода (ФК).

$$ФА := \{ФК_i\},\ i = 1, N$$

Фрагменты кода состоят из двух наборов параметров называемых входным и выходным и тела ФК (Body).

$$ФК := \langle \langle in_1, in_2, \ldots, in_n\rangle, \langle out_1, out_2, \ldots, out_m\rangle, Body \rangle,\ n\geq 0,\ m\geq 0$$

Тело ФК это либо ссылка на атомарный ФК (внешний объект по отношению к модели), либо структурированный ФК.

$$Body\ := atomic\ ФК \ ||\ structured\ ФК$$

Структурированный ФК представляет собой множество локальных фрагментов данных (ФД) и множество локальных фрагментов вычислений ФВ.

ФД может быть либо «атомарным», т.е. блоком памяти фиксированного размера, либо «кортежём» – массивом, состоящим из других ФД, и размер которого вычисляется динамически.

ФВ делятся на 3 типа: single, for или while. При этом в данном случае не подразумевается задание последовательного порядка исполнения, а только декларативное описание. Порядок исполнения определяется только информационными зависимостями.

ФВ single – это применение некоторого ФК, при котором формальным параметрам ФК в соответствие ставятся фактические. Входным параметрам ставится в соответствие т.н. RValue, выходным – LValue. ФВ single имеет параметр – условие, в который должен быть подставлен RValue. Если условие не тождественно истинное, то ФВ называется условным.

ФВ for – это массовое применение некоторого ФК, определяющее множество ФВ, полностью аналогичных ФВ single, которые различаются значением параметра цикла – индексной переменной. ФВ for имеет два входных параметра (которым в соответствие ставится RValue), которые определяют диапазон значений индексной переменной

ФВ while – это массовое применение некоторого ФК, определяющее потенциально бесконечное множество ФВ, аналогичных ФВ single, которые различаются значением параметра цикла – индексной переменной. ФВ while имеет входной параметр (ему в соответствие ставится RValue), задающий начальное значение индексной переменной, и выходной параметр (ему в соответствие ставится LValue), в который будет занесён номер

первой итерации, при которой условие при ФВ обратится в ложь (если такое вообще произойдёт).

LValue – это либо некоторый ФД, либо размер ФД-кортежа.

RValue – это либо LValue, либо константа, либо выражение (+, −, *, /, %, >, <, ==, !=, >=, <=, &&, ||) над RValue. Семантика выражений соответствует таковой для этих операций в C.

2.1.1. Семантика исполнения ФА

1. Исполнение ФА определяется, как исполнение одного ФВ, который является применением одного из ФК, не имеющего ни входных, ни выходных параметров.

2. Исполнение ФВ определяется следующим образом.

a. Входные параметры ФК (если они есть) задают константы и ФД, значения которых вычислены на момент начала исполнения ФК (что такое значение вычислено, см. ниже). Локальные ФД и выходные параметры – это объекты, значения которых требуется вычислять.

b. Среди множества всех ФВ выделяются те, чьи входные параметры означены. Они называются готовыми к исполнению ФВ. Остальные ФВ называются неготовыми к исполнению.

c. Из множества готовых к исполнению ФВ извлекается один ФВ и исполняется.

d. По мере исполнения ФВ какие-то ФД (или псевдо-ФД «размер кортежа») получают свои значения. Если при этом для каких-то ФВ все входные параметры оказались означенными, то эти ФВ переходят в множество готовых к исполнению.

e. Процесс (шаги c-d) продолжается до тех пор, пока множество готовых к исполнению ФВ не окажется пустым притом, что ни один другой ФВ больше не исполняется.

3. Значение ФД считается вычисленным, если:

a. Для атомарного ФД – его значение выработано некоторым ФВ

b. Для кортежа – если вычислены его размер и значения всех его элементов.

2.2. Описание модели управления на основе Сетей Петри

Предлагается следующая модель представления ФА, которая, в отличие от ФА, императивно задает порядок исполнения ФВ, которая была названа RuSh-программой (от Runtime Shell — оболочка времени исполнения).

Она основывается на модели сетей Петри, в которой срабатывания переходов отожествляются с фрагментами вычисления (ФВ), все места необходимые для срабатывания некоторого перехода *t* пронумеровываются, и в дальнейшем их будем называть входными портами перехода *t*. Каждой дуге, выходящей из перехода, ставится в соответствие объект, называемый выходной порт и выходные порты, принадлежащие некоторому переходу, пронумерованы. Меткам соответствует доступность значения некоторого ФД для вычисления, т.е. в отведенном буфере памяти в данный момент хранится значение этого ФД. Каждому выходному порту поставлен в соответствие некоторый входной порт. Данное описание представляет некоторый фрагментированный алгоритм (ФА), но порядок выполнения ФВ будет определяться функционированием сети.

Далее опишем предлагаемую модель управления формально.

RuSh-программа представляет собой набор $\langle E \quad P_i \quad P_o \quad B \quad M \quad f \quad w \rangle$. E - множество переходов, называемых Executor'ами. Срабатывание перехода соответствует запуску некоторого фрагмента вычислений из ФА. Отметим, что в отличие от ФВ, каждый из которых исполняется однократно, Executor может исполняться многократно. P_i - множество всех входных портов RuSh-программы. P_o - множество всех выходных портов RuSh-программы. Когда некоторый Executor вырабатывает значение некоторого ФД в некоторый выходной порт, то это значение в виде сообщения передаётся в определённый входной порт. M - множество сообщений, которые передаются между Executor'ами через порты. Отношение $B \subseteq P_o \times P_i$ определяет, из какого выходного порта, в какой входной порт будет передано сообщение. А именно, если *(p1, p2)*$\in$*B*, то сообщение из выходного порта *p1* будет передано во входной порт *p2*. Отношение $f \subseteq E \times (P_i \cup P_o)$ определяет, какие входные и выходные порты принадлежат Executor'у. Функция $w : M \to P_i$ определяет какое сообщение в какой порт записывается.

Описанная модификация сети Петри должна быть безопасной, т.е. число меток в любом месте не превышает числа 1. Это позволяет ограничиться 1 буфером для реализации принятия сообщений во входные порты.

Определяются два варианта исполнения Executor'а.

1. Если во всех входных портах некоторого Executor'а есть сообщение, то такой Executor считается готовым к исполнению и должен будет исполниться. Порядок исполнения готовых к исполнению Executor'ов может быть любым.
2. Перед началом исполнения RuSh-программы Executor может быть помечен как готовый к исполнению. Тогда он будет исполнен однократно, так же как и

любой другой готовый к исполнению Executor. При этом не требуется, чтобы в его входных портах были сообщения. Если в момент исполнения в каких-то из его входных портов имеются сообщения, то они не потребляются и могут быть потреблены позже.

После исполнения Executor'а не обязательно, что в каждом выходном порте данного Executor'а будет метка.

2.2.1. Исполнение RuSh-программы на параллельном вычислителе

RuSh-программа предназначена для исполнения на мультикомпьютере, то есть параллельном компьютере с распределённой памятью, вычислительным узлами которого могут быть мультипроцессоры с общей памятью. В ходе исполнения RuSh-программы Executor'ы распределяются среди доступных узлов вычислителя. При этом выходные порты будут разделены на два вида:

- Локальные, для коммуникаций между Executor'ами внутри одного узла.
- Прокси, для коммуникации Executor'ов между разными узлами.

Сообщения в локальные порты могут быть реализованы простым копированием в памяти или даже передачей указателя, в то время как сообщения в прокси порте будут реализованы с помощью коммуникаций мультикомпьютера.

2.3. Отображение фрагментированного алгоритма в представления модели управления RuSh

Покажем, как RuSh-программа может быть использована для представления ФА и задания управления в нем.

Известно, что сети Петри позволяют описывать управление для широкого круга систем. Так как предложенная модель основывается на сетях Петри, то область её применения тоже ожидается достаточно широкой. Для того чтобы показать применимость предложенной модели, рассмотрим несколько примеров численных алгоритмов, чтобы продемонстрировать её применимость для, как минимум, сходных по структуре алгоритмов. Более детальное исследование области применения предложенной модели представляет собой отдельную задачу и выходит за рамки работы.

Далее в этом разделе рассматриваются некоторые численные алгоритмы. Приводится их ФА и предлагаются RuSh-программы.

2.3.1 Алгоритм Кэннона (умножение матриц)

2.3.1.1. Постановка задачи

При построении параллельных способов выполнения матричного умножения широко используется блочное представление матриц. Алгоритм Кэннона – блочный алгоритм умножения матриц, предназначенный для исполнения на регулярных мультикомпьютерах со структурой «решетка» или «2D-тор»[10].

Пусть поставлена задача умножения матриц $C = A * B$, где C,A,B – квадратные матрицы размером $m*m$. Разобьем каждую из матриц A и B на p квадратных блоков и зададим нумерацию процессоров $P(i,j)$, где $i = 1,m-1$, $j= 1,m-1$ и умножение блоков $A(i,j)$ и $B(i,j)$ привяжем к процессору $P(i,j)$ соответственно.

Разбиение матриц на блоки в алгоритме Кэннона осуществляется таким образом, чтобы располагаемые блоки в процессорах могли быть умножены без каких-либо дополнительных коммуникаций между процессорами, а так же чтобы передача данных между процессорами в процессе выполнения алгоритма могла быть осуществлена с помощью простых коммуникационных операций, таких как циклический сдвиг.

Сам алгоритм состоит из двух больших операций - это инициализация матриц, то есть подготовка к умножению, и основной цикл алгоритма.

Инициализация матриц:

- для каждой строки i, кроме первой, циклический сдвиг блоков на i - 1 позиций влево;
- для каждой строки i, кроме первой, циклический сдвиг блоков на i - 1 позиций влево;

Основной цикл:

Формально определим операции: (*) - циклический сдвиг влево блоков строки на 1 позицию (**) - циклический сдвиг вверх блоков столбца на 1 позицию.

Тогда алгоритм выглядит следующим образом:

1. Для всех строк кроме первой производим коммуникационную операцию (*)
2. Для всех столбцов кроме первого производим коммуникационную операцию (**)
3. Вычисляем C(i,j) = C(i,j) + A(i,j) * B(i,j)

```
for (int i = 0; i <= m - 1; i++)
{
    Для всех строк производим коммуникационную операцию (*).
    Для всех столбцов производим коммуникационную операцию (**).
    Вычисляем C(i,j) = C(i,j) + A(i,j) * B(i,j)
```

}

2.3.1.2. RuSh-программа

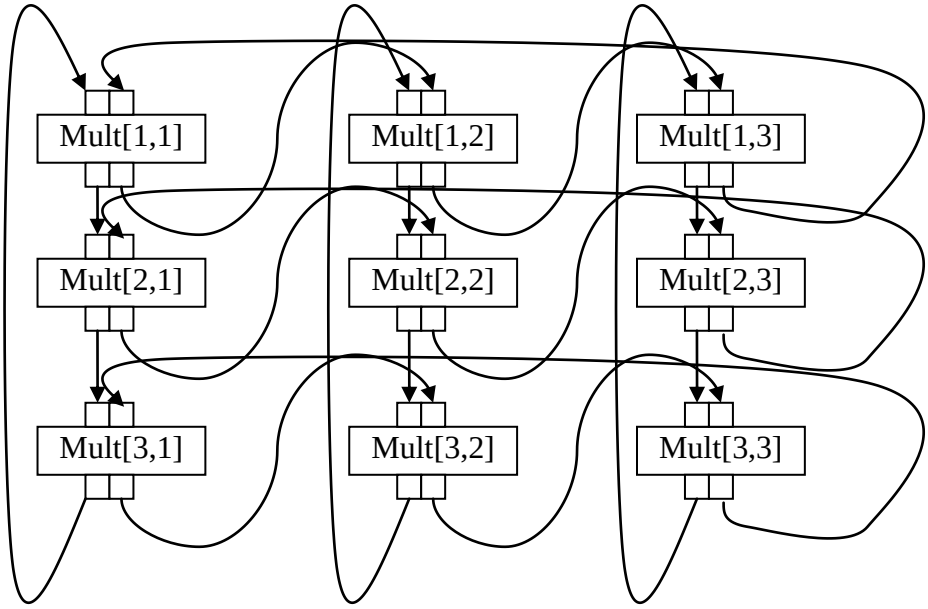

Рис. 1 – Схема RuSh-программы алгоритма Кэннона

На Рис. 1 приведена схема RuSh-программы алгоритма Кэннона. Каждый Executor $Mult[i, j]$ отвечает за расчет своей результирующей подматрицы $C_{i,j}$. При этом в начале все Executor'ы помечаются как готовые к исполнению, и этот этап соответствует инициализации матриц.

2.3.2. Решение уравнение Пуассона явным методом

Решение уравнение Пуассона является одним из этапов моделирования методом «частиц в ячейках», который будет рассматриваться в разделе 2.3.3.. Этот этап можно рассматривать как самостоятельную задачу, так как в ходе моделирования он составляет существенную часть всех вычислений.

2.3.2.1. Постановка задачи

Уравнение Пуассона имеет вид (1)

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} + \frac{\partial^2 \Phi}{\partial z^2} = \rho \quad (1).$$

При этом оно аппроксимируется, следующей 7-точечной разностной схемой второго порядка

$$\frac{\Phi_{i+1,k,l} - 2\Phi_{i,k,l} + \Phi_{i-1,k,l}}{h_x^2} + \frac{\Phi_{i,k+1,l} - 2\Phi_{i,k,l} + \Phi_{i,k-1,l}}{h_y^2} + \frac{\Phi_{i,k,l+1} - 2\Phi_{i,k,l} + \Phi_{i,k,l-1}}{h_z^2} = \rho_{i,k,l} \quad (2)$$

Для описания и реализации были выбраны 7-точечная разностная схема и метод Якоби, так как они использовались в работе [11]. с которой производилось сравнение результатов моделирования.

Метод Якоби является итерационным методом, в нем на каждой итерации происходит уточнение решения относительно некоторого начального. Формула вычисления значений на новой итерации получается, если выразить $\Phi_{i,k,l}$ из схемы (2). В методе Якоби для вычисления сеточного значения на новой итерации все используемые в формуле значения берутся с предыдущей итерации. Итерационный процесс продолжаются до сходимости: $\max_{i,j,k} \left| \Phi_{i,j,k}^{m+1} - \Phi_{i,j,k}^{m} \right| < \varepsilon$. Здесь индекс m – номер итерации.

2.3.2.2. RuSh-программа

В основе параллельной реализации лежит принцип пространственной декомпозиции. Пространство моделирования разбивается на слои (или решетки), и каждый слой выполняется отдельным фрагментом в случае RuSh. Так как для расчета $\Phi_{i,k,l}$ требуется значения $\Phi_{i-1,k,l}$ и $\Phi_{i+1,k,l}$, то для параллельной реализации решения уравнения Пуассона требуется обмен теневыми гранями на каждой итерации.

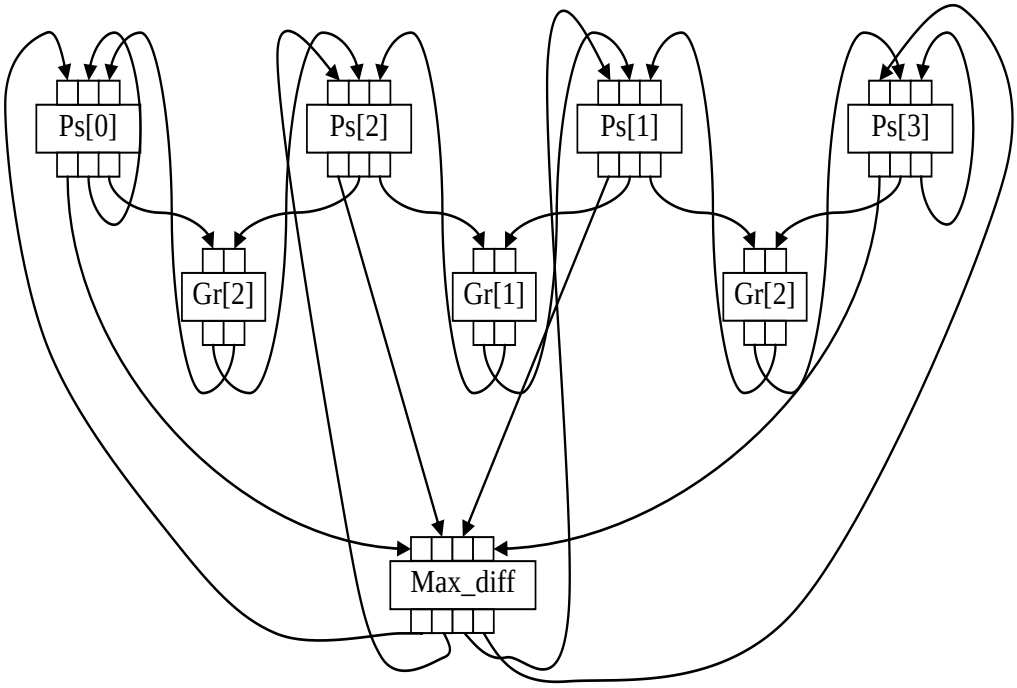

Рис. 2 - Схема RuSh-программы метода Якоби для решения уравнения Пуассона (область разбита на 4 фрагмента)

В приведенной схеме RuSh-программы область разбивается вдоль оси X на 4 подобласти, при этом количество подобластей является задаваемым параметром. Далее опишем Executor'ы, использующиеся в схеме:

FR - количество подобластей (задаваемый параметр);

Ps[i], i = [0, FR-1] –Executor, вычисляющий 1 итерацию решения уравнения Пуассона;

Gr[i], i = [0, FR-2] – Executor, отвечающий за обмен теневыми гранями;

Max_diff – Executor, вычисляющий максимальную невязку.

2.3.3. Моделирование самогравитирующего вещества методом «частиц в ячейках»

2.3.3.1 Постановка задачи

Подробно задача моделирования самогравитирующегося вещества методом «частиц в ячейках» описана в работе [11], а здесь приведем только основные этапы моделирования.

Движение вещества под действием сил гравитации описывает следующая система уравнений:

$$\frac{\partial f}{\partial t} + \vec{u}\,\frac{\partial f}{\partial \vec{x}} + \vec{a}\,\frac{\partial f}{\partial \vec{u}} = 0\,, \qquad (1)$$

$$\vec{a} = -\nabla\Phi\,, \qquad (2)$$

$$\frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2} + \frac{\partial^2\Phi}{\partial z^2} = 4\pi\rho\,, \qquad (3)$$

$$\rho(t,\vec{x}) = \int f d\vec{u}\,. \qquad (4)$$

Здесь уравнение (1) – бесстолкновительное уравнение Больцмана (в физике плазмы известно как уравнение Власова), $f(t,\vec{x},\vec{u})$ - функция распределения вещества во времени t по координатам $\vec{x} = (x, y, z)$ и скоростям $\vec{u} = (u, v, w)$, $\vec{a}(t,\vec{x})$ – ускорение, $\rho(t,\vec{x})$ – распределение плотности вещества, Φ – гравитационный потенциал, распределение которого удовлетворяет уравнению Пуассона (3).

Алгоритм метода «частиц в ячейках» состоит из четырех чередующихся этапов.

1. *Лагранжев этап. Сдвиг частиц под действием сил гравитации.*

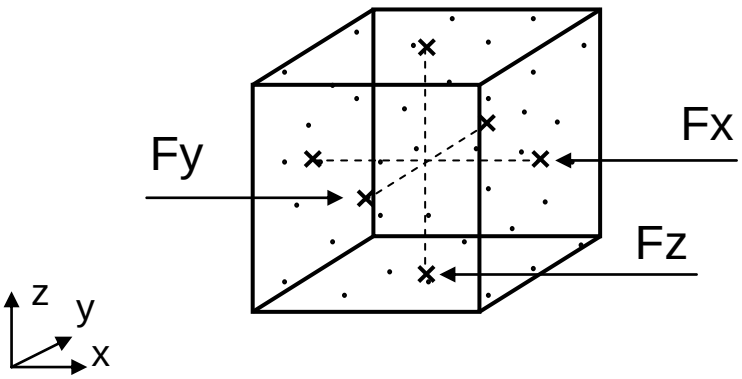

Рис. 3- Схема расположения точек (обозначены крестиками), в которых заданы значения гравитационных сил, в отдельной ячейке сетки.

На этом этапе частицы сдвигаются под действием гравитационных сил, заданных на сетке (Рис. 3). Новые скорости и координаты частиц вычисляются по формулам (5). При этом силы $\vec{F} = (F_X, F_Y, F_Z)$, действующие на частицу, задаются путем линейной интерполяции из ближайших узлов сетки по формулам (6).

$$\vec{u}_j^{\,n+\frac{1}{2}} = \vec{u}_j^{\,n-\frac{1}{2}} + \tau \frac{\vec{F}_j^{\,n}}{m_j}, \qquad \vec{x}_j^{\,n+1} = \vec{x}_j^{\,n} + \tau \vec{u}_j^{\,n+\frac{1}{2}}. \qquad (5)$$

$$\begin{aligned} F_X &= (1-s'_x)\Big[(1-s_y)\big((1-s_z)F_{X\,i',k,l} + s_z F_{X\,i',k,l+1}\big) + s_y\big((1-s_z)F_{X\,i',k+1,l} + s_z F_{X\,i',k+1,l+1}\big)\Big] + \\ &+ s'_x\Big[(1-s_y)\big((1-s_z)F_{X\,i'+1,k,l} + s_z F_{X\,i'+1,k,l+1}\big) + s_y\big((1-s_z)F_{X\,i'+1,k+1,l} + s_z F_{X\,i'+1,k+1,l+1}\big)\Big], \end{aligned}$$

$$\begin{aligned} F_Y &= (1-s_x)\Big[(1-s'_y)\big((1-s_z)F_{Y\,i,k',l} + s_z F_{Y\,i,k',l+1}\big) + s'_y\big((1-s_z)F_{Y\,i,k'+1,l} + s_z F_{Y\,i,k'+1,l+1}\big)\Big] + \\ &+ s_x\Big[(1-s'_y)\big((1-s_z)F_{Y\,i+1,k',l} + s_z F_{Y\,i+1,k',l+1}\big) + s'_y\big((1-s_z)F_{Y\,i+1,k'+1,l} + s_z F_{Y\,i+1,k'+1,l+1}\big)\Big] \end{aligned}, \qquad (6)$$

$$\begin{aligned} F_Z &= (1-s_x)\Big[(1-s_y)\big((1-s'_z)F_{Z\,i,k,l'} + s'_z F_{Z\,i,k,l'+1}\big) + s_y\big((1-s'_z)F_{Z\,i,k+1,l'} + s'_z F_{Z\,i,k+1,l+1'}\big)\Big] + \\ &+ s_x\Big[(1-s_y)\big((1-s'_z)F_{Z\,i+1,k,l'} + s'_z F_{Z\,i+1,k,l'+1}\big) + s_y\big((1-s'_z)F_{Z\,i+1,k+1,l'} + s'_z F_{Z\,i+1,k+1,l+1'}\big)\Big] \end{aligned}.$$

Здесь значения сил F_X, F_Y, F_Z с индексами i,k,l без штрихов соответствуют целочисленным точкам (в центрах ячеек) по соответствующему направлению, а индексы со штрихами соответствуют полуцелым точкам (на границах ячеек) по соответствующему направлению. Таким же образом значения коэффициентов s_x, s_y, s_z без штрихов соответствуют доле шага сетки, на который отстоит частица от ближайшей меньшей целочисленной точки, а значения коэффициентов со штрихами – доле шага сетки, на который отстоит частица от ближайшей меньшей полуцелой точки по соответствующему направлению.

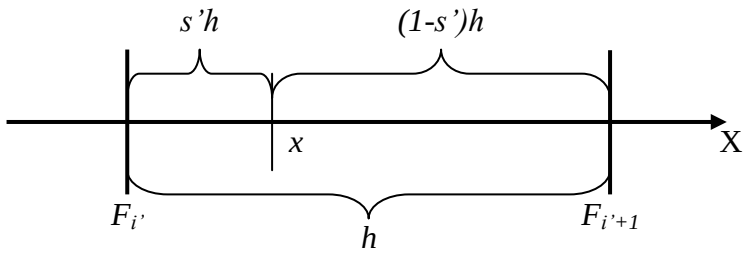

Рис. 4 - Расположение сил в ячейке сетки для одномерного случая.

На рис.4 приведен пример для одномерного случая. Значения сил $F_{i'}$ и $F_{i'+1}$ заданы на границах ячейки. x – расположение некоторой j-й частицы в ячейке. Сила для этой частицы тогда будет вычисляться по формуле: $F = (1 - s')F_{i'} + s'F_{i'+1}$.

2. *Вычисление распределения плотности.*

Для перехода от лагранжевого этапа к эйлерову необходимо распределение плотности, заданное в виде лагранжевой сетки из частиц, перевести на фиксированную эйлерову сетку. Значения плотности будут задаваться в центрах ячеек (в точках с целочисленными индексами).

Вклад каждой частицы в распределение плотности определяется ядром метода частиц. В трехмерном случае ядро выглядит следующим образом:

$$\tilde{R}(x, y, z) = \begin{cases} \dfrac{1}{h_x h_y h_z}\left(1 - \dfrac{x}{h_x}\right)\left(1 - \dfrac{y}{h_y}\right)\left(1 - \dfrac{z}{h_z}\right), & |x| \le h_x \ \text{и} \ |y| \le h_y \ \text{и} \ |z| \le h_z, \\ 0, & \text{иначе.} \end{cases}$$

Вклады одной частицы в ближайшие узлы сетки при использовании ядра определяются соотношениями (7).

$$\begin{aligned} &\rho_{i,k,l} = (1 - s_x)(1 - s_y)(1 - s_z)m && \rho_{i,k,l+1} = (1 - s_x)(1 - s_y)s_z m \\ &\rho_{i,k+1,l} = (1 - s_x)s_y(1 - s_z)m && \rho_{i,k+1,l+1} = (1 - s_x)s_y s_z m \\ &\rho_{i+1,k,l} = s_x(1 - s_y)(1 - s_z)m && \rho_{i+1,k,l+1} = s_x(1 - s_y)s_z m \\ &\rho_{i+1,k+1,l} = s_x s_y(1 - s_z)m && \rho_{i+1,k+1,l+1} = s_x s_y s_z m \end{aligned} \qquad (7)$$

Здесь индексы i, k и l определяют узел сетки, ближайший к частице слева по соответствующему направлению, а индексы $i+1$, $k+1$ и $l+1$ – ближайший узел справа. m – масса частицы. Коэффициенты s_x, s_y, s_z соответствуют доле шага сетки, на который отстоит частица от ближайшего узла слева (аналогично предыдущему шагу алгоритма).

Таким образом, вычисления на данном этапе представляют собой накопление вкладов от всех частиц в сеточное распределение плотности путем применения формул (7) для каждой частицы.

3. Эйлеров этап. Вычисление распределения гравитационного потенциала.

На эйлеровом этапе решается уравнение Пуассона (3): по распределению плотности вещества ρ на фиксированной сетке вычисляется распределение гравитационного потенциала Φ. Трехмерное уравнение Пуассона аппроксимирутся следующей 7-точечной разностной схемой второго порядка:

$$\frac{\Phi_{i+1,k,l} - 2\Phi_{i,k,l} + \Phi_{i-1,k,l}}{h_x^2} + \frac{\Phi_{i,k+1,l} - 2\Phi_{i,k,l} + \Phi_{i,k-1,l}}{h_y^2} + \frac{\Phi_{i,k,l+1} - 2\Phi_{i,k,l} + \Phi_{i,k,l-1}}{h_z^2} = 4\pi\rho_{i,k,l} \quad (8)$$

Для решения уравнения Пуассона используется метод Якоби. Это итерационный метод, в котором на каждой итерации происходит уточнение решения относительно некоторого начального. Формула вычисления значений на новой итерации получается, если выразить $\Phi_{i,k,l}$ из схемы (8). В методе Якоби для вычисления сеточного значения на новой итерации все используемые в формуле значения берутся с предыдущей итерации. Итерационный процесс продолжаются до сходимости: $\max\limits_{i,j,k} \left| \Phi_{i,j,k}^{m+1} - \Phi_{i,j,k}^{m} \right| < \varepsilon$. Здесь индекс m – номер итерации.

Можно существенно ускорить процесс сходимости, если на каждом шаге по времени в качестве начального приближения брать значения потенциала с предыдущего шага по времени.

Граничные условия для уравнения Пуассона задаются приближенно по формуле:

$$\Phi\big|_{\Gamma} = -\frac{M}{r}. \qquad (9)$$

Здесь M – масса всего вещества в системе, r – расстояние от центра масс до соответствующей точки на границе. Эта формула справедлива для сферически симметричного распределения вещества и является хорошим приближением, если большая часть вещества расположена вблизи центра масс, а граница расположена достаточно далеко от него.

4. *Вычисление гравитационных сил.*

На завершающем этапе из распределения гравитационного потенциала Φ, заданного в центрах ячеек сетки (точки с целочисленными индексами), необходимо вычислить значения гравитационных сил $\vec{F} = (F_X, F_Y, F_Z)$, заданных в центрах граней сетки.

Сила, действующая на частицу единичной массы, задается соотношением $\vec{F} = -\nabla\Phi$. Аппроксимация этого соотношения с помощью центральных разностей имеет следующий вид:

$$F_{X\,i+\frac{1}{2},k,l} = -\frac{\Phi_{i+1,k,l} - \Phi_{i,k,l}}{h},\ F_{Y\,i,k+\frac{1}{2},l} = -\frac{\Phi_{i,k+1,l} - \Phi_{i,k,l}}{h},\ F_{Z\,i,k,l+\frac{1}{2}} = -\frac{\Phi_{i,k,l+1} - \Phi_{i,k,l}}{h}. \quad (10)$$

Графически взаимное расположение точек на сетке, в которых заданы силы и потенциал, показаны на Рис. 3.

По завершении последнего этапа все готово для выполнения лагранжевого этапа на следующем шаге по времени.

2.3.3.2. RuSh-программа метода «частиц в ячейках»

Фрагментация метода «частиц в ячейках» основана на декомпозиции области. В нашем случае область разбивалась вдоль оси X.

На этапе вычисления потенциала (решения уравнения Пуассона) для расчета $\Phi_{i,k,l}$ требуется значения $\Phi_{i-1,k,l}$ и $\Phi_{i+1,k,l}$, поэтому необходимо осуществлять обмен теневыми гранями потенциала на каждой итерации. Так же на каждой итерации происходил сдвиг частиц, что повлекло необходимость обмена частицами между Executor'ами.

При этом на этапе вычисления потенциала (решения уравнения Пуассона) необходимо было осуществлять обмен теневыми гранями потенциала. Так как при вычислении плотности частица вносит вклад сразу в 4 узла сетки, то также был обмен теневыми гранями плотности.

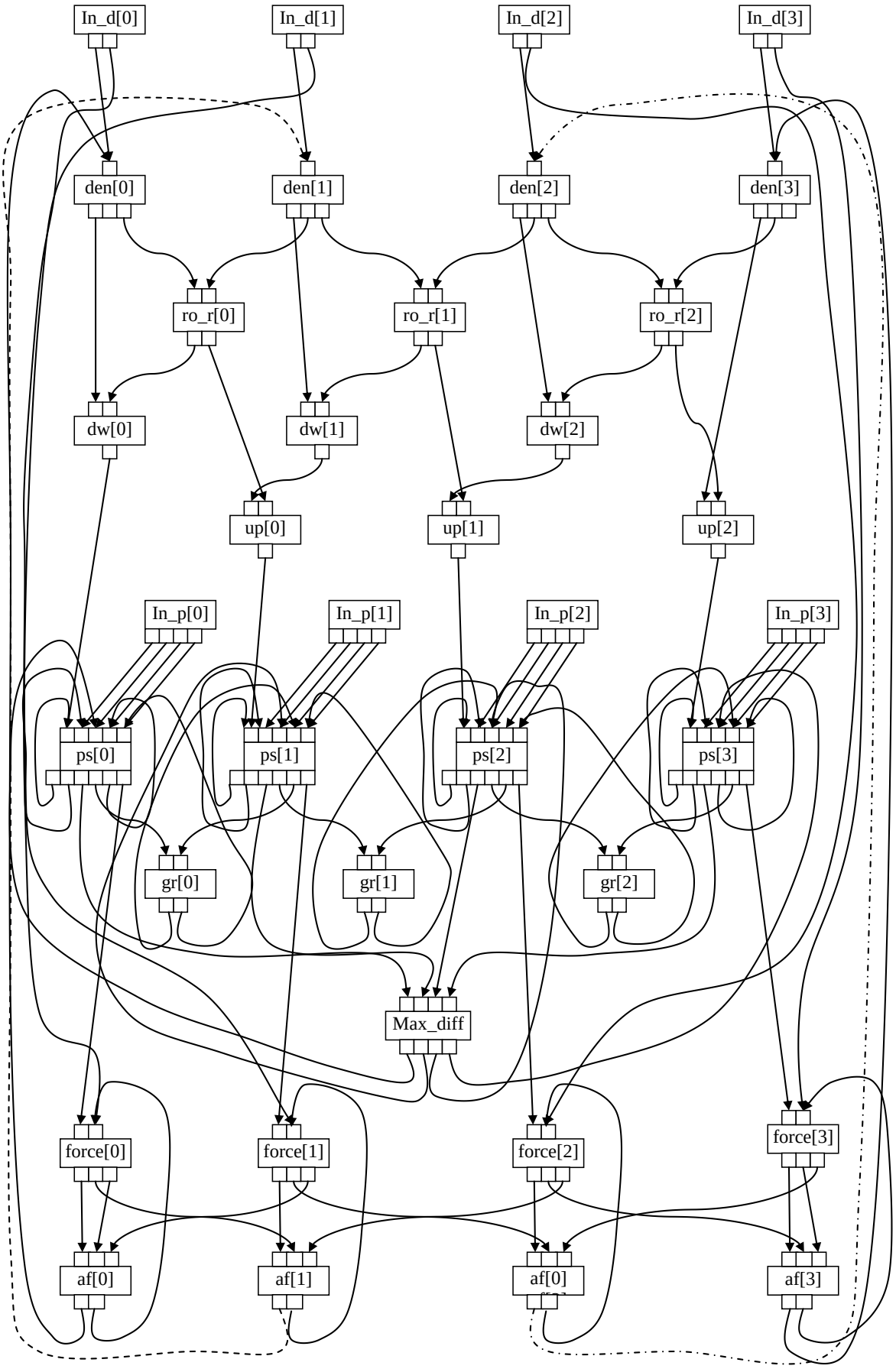

Рис. 5 - Схема RuSh-программы метода «частиц в ячейках» (область разбита на 4 фрагмента)

В приведенной схеме RuSh-программы область моделирования разбита на 4 подобласти для примера, а в общем случае количество подобластей является задаваемым параметром. При этом различный шаблон дуги используется для наглядности. Далее приведем описание используемых обозначений:

FR - количество подобластей (задаваемый параметр);

Задание частиц в области моделирования

In_d[i], i = [0, FR-1] - Executor, инициализирующий частицы в заданной подобласти

Вычисление плотности

den[i], i = [0, FR-1] – Executor, вычисляющий плотность в заданной подобласти

Некоторые грани плотности, вычисляются в нескольких Executor'ах, то используются дополнительные Executor'ы для вычисления плотности, такие как ro_r[i], i = [0, FR-2], dw[i], i = [0, FR-2], up[i], i = [0, FR-2].

Вычисление потенциала

In_p[i], i = [0, FR-1] – Executor, инициализация данных для решения уравнения Пуассона на 1 временном слое.

Ps[i], i = [0, FR-1] – Executor, вычисляющий одну итерацию решения уравнения Пуассона;

Gr[i], i = [0, FR-2] – Executor, отвечающий за обмен теневыми гранями;

Max_diff – Executor, вычисляющий максимальную невязку.

Вычисление сил действующих на частицы и сдвиг частиц внутри Executor'а

force[i], i = [0, FR-1]

Сдвиг частиц между Executor'ами

af[i], i = [0, FR-1]

2.3.4. Реакция окисления CO на поверхности палладия

2.3.4.1. Постановка задачи

В работах [12-14] представлена реакция окисления CO на поверхности палладия, моделируемая блочно-синхронным клеточным автоматом, здесь же опишем основные этапы моделирования.

Каталитическое окисление CO на платиновых металлах сопровождается колебаниями скорости образования CO_2 и концентраций адсорбированных веществ. В работе [12] был предложен следующий детальный механизм реакции:

1) $O_{2(газ)} + 2* \rightarrow 2O_{ads}$; 2) $CO_{газ} + * \leftrightarrow CO_{ads}$;

3) $CO_{ads} + O_{ads} \rightarrow CO_{2(газ)} + 2*$; 4) $O_{ads} + *v \rightarrow [*O_{sub}]$;

5) $CO_{ads} + [*O_{sub}] \rightarrow CO_{2(газ)} + 2* + *v$; 6) $CO_{газ} + [*O_{sub}] \leftrightarrow [CO_{ads}*O_{sub}]$;

7) $[CO_{ads}*O_{sub}] \rightarrow CO_{2(газ)} + * + *v$

где $*$ и $*v$ – активные центры поверхности и приповерхностного слоя, соответственно. Первая стадия описывает необратимую адсорбцию кислорода, вторая – адсорбцию и десорбцию CO. Третья стадия соответствует реакции между молекулами CO_{ads} и атомами кислорода O_{ads}. Образование слоя растворенного кислорода O_{sub} отображена стадией 4. Реакция между молекулами CO_{ads} и приповерхностным кислородом описывается стадией 5. Образование комплекса молекул CO_{ads} на растворенном кислороде в виде $[CO_{ads}*O_{sub}]$ происходит через прямую адсорбцию CO из газовой фазы (стадия 6). Разрушение комплекса $[CO_{ads}*O_{sub}]$ сопровождается образованием молекул CO_2 с освобождением адсорбционных мест $*$ и $*v$ (стадия 7).

Поверхностная диффузия молекул CO_{ads} описывается следующими уравнениями:

8) $CO_{ads} + * \rightarrow * + CO_{ads}$; 9) $CO_{ads} + [*O_{sub}] \rightarrow * + [CO_{ads}*O_{sub}]$;

10) $[CO_{ads}*O_{sub}] + [*O_{sub}] \rightarrow [*O_{sub}] + [CO_{ads}*O_{sub}]$;

Стадия 9 описывает образование комплекса молекул CO_{ads} на растворенном кислороде в виде $[CO_{ads}*O_{sub}]$ вследствие диффузии CO по поверхности. Стадии 8 и 10 описывают диффузию частиц CO_{ads} и $[CO_{ads}*O_{sub}]$. Диффузия на поверхности Pd катализатора является анизотропной, коэффициент интенсивности диффузии для разных направлений различен.

Диффузия CO_{ads} на свободные участки и участки с приповерхностным кислородом важна для синхронизации процессов возникающих на отдельных участках моделируемой поверхности.

Состояния ячеек матрицы устанавливаются согласно правилам, определенными подробным механизмом реакции: каждая клетка решетки может существовать в одном из пяти состояний: *, CO_{ads}, O_{ads}, $[*O_{sub}]$, и $[CO_{ads}*O_{sub}]$.

Каждый раз после выбора одного из процессов и попытки его реализовать, программа рассматривала внутренний цикл распространения, который включал M_{diff} попыток диффузии для молекул CO_{ads} (обычно M_{diff} = 20-100). Коэффициент реакции окисления CO и доля поверхности, занятая реагентами, вычислялись после каждого МС

такта как отношение количества образовавшихся молекул CO_2 (или число клеток решетки в соответствующем состоянии) к полному количеству клеток N x M.

Вероятность реализации каждой стадии для процессов адсорбции, десорбции и реакции определялась отношением константы скорости данной стадии k_i к сумме констант скоростей всех стадий: Σk_i . В константы скоростей адсорбции k_1, k_2, k_6 включены парциальные давления реагирующих веществ и концентрация активных центров поверхности палладия.

В работе [13] для стадий 1- 7 определены значения величин k_i, при которых в системе возникают колебательные процессы.

Таблица значений k_i, при которых возникают колебательные процессы							
$\mathbf{k_1}$	$\mathbf{k_2}$	$\mathbf{k_{-2}}$	$\mathbf{k_4}$	$\mathbf{k_5}$	$\mathbf{k_6}$	$\mathbf{k_{-6}}$	$\mathbf{k_7}$
1	1	0.2	0.03	0.01	1	0.5	0.02

Случайным образом (используя генератор равномерного распределения на отрезке [0,1]) выбирается один из возможных процессов в соответствие с отношением $k_i/\Sigma k_i$. Затем также случайным образом выбираются координаты ячейки решетки (или двух, в зависимости от выбранного процесса). Состояние клетки определяется выбранным процессом и возможностью его реализации.

После каждой попытки адсорбции CO или O_2 , а также после диффузии CO_{ads} , проверялась возможность осуществления реакции образования диоксида углерода. В модели предполагается, что эта реакция происходит немедленно.

2.3.4.1.1. Моделирования реакции окисления с помощью блочно-синхронного КА

Алгоритм построения блочно-синхронного КА

Подробный алгоритм построения блочно-синхронного КА и его сравнение с асинхронным описано в работе [14] .

Приведем только основные этапы.

Шаг 1. – Выбор шаблона моделирования в соответствии с особенностями реакцией окисления. В данном случае шаблон состоит из 13 клеток и представлен на Рис. 6.

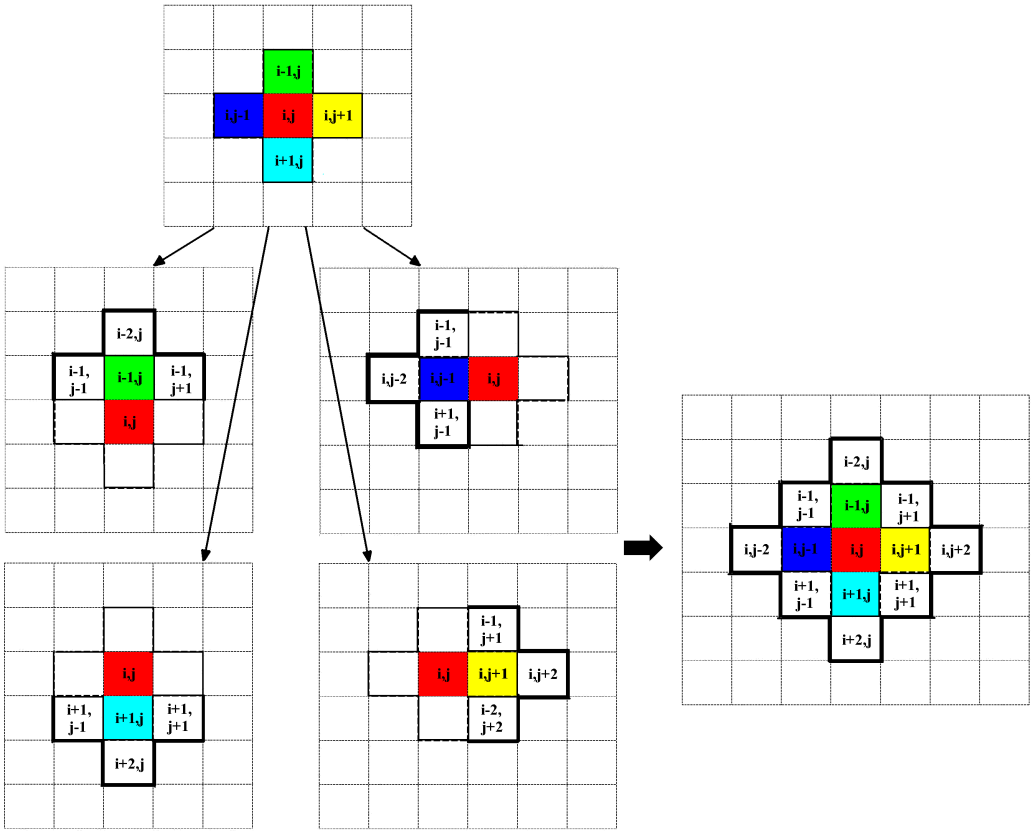

Рис. 6 – Размер шаблона для моделирования задачи реакции окисления CO на поверхности палладия КА

Шаг 2. На множестве имен M, блок $B_к$ определяет множество разбиений множества, где каждая итерация состоит из 13 шагов. На каждой стадии подстановки применяются синхронно к k-ой клетке всех блоках принадлежащих k-му разбиению.

2.3.4.2. RuSh-программа блочно-синхронного КА

Распараллеливания блочно-синхронного КА основано на декомпозиции области. Разные подобласти области моделирования распределяются между процессорными элементами, поэтому возникает необходимость обмениваться значениями граничных клеток в ходе моделирования.

Описанная задача, по сути, похожа на явную схему и в этом смысле аналогична методу Якоби. Отличием является то, что не все граничные клетки меняют свои значения, а только те которые входят в шаблон. Поэтому в данном случае обмен теневыми гранями заключается в обмене только тех граничных клеток, значения которых изменились. Для выбранного шаблона моделирования теневая грань имеет ширину 2, а не 1, как в методе Якоби. Так же для того чтобы при наложении шаблоны не пересекались, накладывается ограничение на размер области, она должна быть кратна 13 и N должно быть меньше либо равно размеру области моделирования.

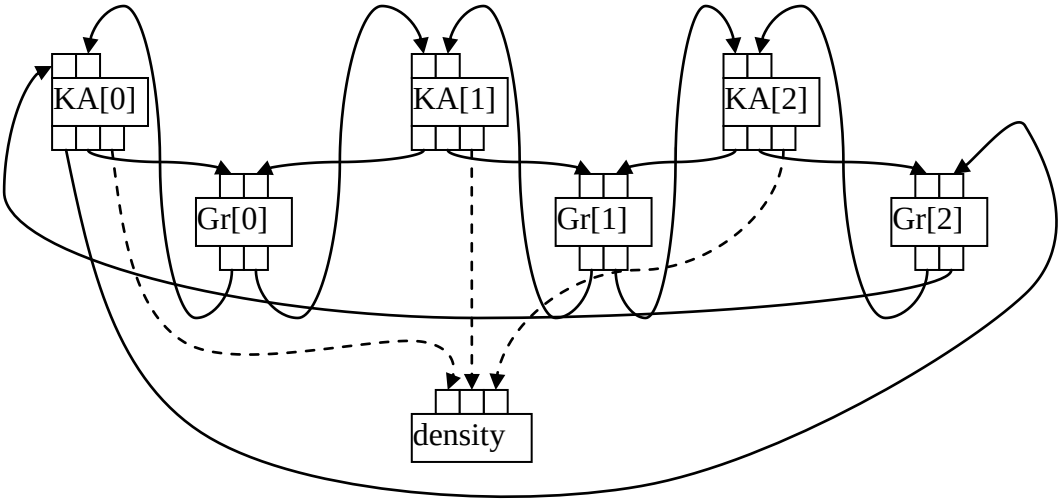

Рис. 7 – Схема RuSh-программы блочно-синхронного КА (область разбита на 3 подобласти)

В описании схемы RuSh-программы используются следующие обозначения:

N – количество подобластей (задаваемый параметр)

KA[i], i = 1,N – Executor, который отвечает за расчет доменов, принадлежащих i-подобласти

Gr[i], i = 1,N – Executor, отвечающий за обмен значениями граничных клеток

Density – Executor, считающий плотность на каждой K итерации, где K – шаг моделирования на котором выводятся значения плотностей веществ для визуализации.

Пунктирной линией обозначены передачи сообщений при выполнение некоторого условия (в данном случае условие номер итерации кратен K).

2.4. Взаимодействие с модулем динамической балансировки

Во фрагментированном программировании одна из функций runtime-системы – это автоматическая динамическая балансировка нагрузки. Чтобы rush-программы могла совместно работать с этим модулем, в системе RuSh необходимо обеспечивать возможность миграции Executor'а с узла на узел по запросу модуля, который следит за балансом загрузки доступных ресурсов вычислителя. В тоже время мы должны гарантировать корректность миграции. Для этого предложим алгоритм миграции и проверим корректность его работы.

Будем говорить, что миграция выполняется корректно, если выполняются следующие требования:

1. Если сообщение от одного Executor'а послано другому, то оно должно быть доставлено, т.е. гарантируется его доставка.
2. Для конкретного набора данных Executor срабатывает ровно один раз.

2.4.1. Алгоритм миграции

Допустим системе RuSh поступил запрос от модуля динамической балансировки о миграции Executor'а *A* с узла *p1* на узел *p2*.

Сначала рассмотрим основную идею алгоритма миграции неформально.

Алгоритм миграции можно разбить на следующие этапы:

Этап 1. Блокировка Executor'а *A*. Т.е. если Executor *A* в данный момент исполняется, то необходимо дождаться окончания его исполнения, а затем запретить всякое исполнение, даже если есть сообщения во входных портах.

Этап 2. Чтобы выполнялось требование 1 корректности работы системы, а именно предотвратить потерю сообщений, которые могут быть посланы мигрирующему Executor'у, необходимо заблокировать исполнение Executor'ов, которые пишут сообщения в порты Executor'у *A*.

Этап 3. После этого создаем копию мигрирующего Executor'а *A* на узле *p2* с тем же состоянием, что и исходный. Для того чтобы можно было переносить Executor с узла на узел, он должен поддерживать стандартные процедуры сериализации – сохранение состояния в буфер и восстановление состояния из буфера.

Этап 4. Необходимо, чтобы все Executor'ы, которые посылаю сообщения мигрирующему Executor'у *A,* теперь посылали сообщение его копии на узле *p2,* а также чтобы все Executor'ы, которые получают сообщения от Executor'а *A,* получали сообщения от его копии.

Этап 5. Удаление Executor'а *A* с узла *p1*

Этап 6. Разблокирование Executor'ов, которые были заблокированы на Этапе 2.

Теперь опишем алгоритм миграции более формально.

На каждом узле есть непересекающиеся множества

E_R - множество, содержащее Executor'ы, которые сейчас работают

E_{WB} - множество, содержащее Executor'ы, которые хотят заблокироваться, но пока не могут.

E_B - множество, содержащее заблокированные Executor'ы.

E_M - множество, содержащее мигрирующие Executor'ы.

E_A - множество, содержащее мигрировавшие Executor'ы.

$f \subseteq E_R \times E_R$ отношение, определяющее связь между Executor'ами.

В начальный момент времени множество E_R и f задается пользователем, а E_{WB} ,E_B , E_M , E_A пусты.

Переход из множества во множество осуществляется следующим образом. Если *Executor* должен мигрировать с текущего узла на другой, то, если он в данный момент исполняется, *Executor* переходит во множество E_{WB} , если же нет, то в множество E_B . Затем для всех *Executor*'ов, которые с ним связаны через отношение f , проделывается тоже самое. Если *Executor* и все *Executor*'ы, связанные с ним через отношение f , принадлежат множеству E_B , то *Executor* переходит в множество E_M . После этого его копия создается на другом узле в множестве E_A . Для копии *Executor*'а определяются отношения такие же, как и для его оригинала. После этого оригинал уничтожается, а копия переходит во множество E_R на своем узле, в тоже время, в множество E_R переходят все *Executor*'ы, связанные с ним через отношение f .

Формально эти правила изменение состояния алгоритма во время миграции *ex1* на узел *Node* будут представлены следующим образом:

1) Если ($ex1 \to state \neq 0$), то из $E_R = E_R / ex1$ и $E_{WB} = E_{WB} \cup ex1$, иначе $E_R = E_R / ex1$ и $E_B = E_B \cup ex1$
2) Если ($ex1 \to state = 0$) и $ex1 \in E_{WB}$, то $E_{WB} = E_{WB} / ex1$ и $E_B = E_B \cup ex1$
3) $\forall e \in ex1 fe \;\; e \in E_R$ Если ($e \to state \neq 0$), то из $E_R = E_R / e$ и $E_{WB} = E_{WB} \cup e$, иначе $E_R = E_R / e$ и $E_B = E_B \cup e$
4) $\forall e \in ex1 fe \;\; e \in E_R$ Если ($e \to state = 0$) и $e \in E_{WB}$, то $E_{WB} = E_{WB} / e$ и $E_B = E_B \cup e$
5) Если $ex1 \in E_B$ и $\forall e \in ex1 fe \;\; e \in E_B$, то $E_B = E_B / ex1$ и $E_M = E_M \cup ex1$

6) Если $ex1 \in E_W$, то на узле *Node* $ex2 = ex1$, $E_A = E_A \cup ex2$
7) $\eta(x) = \{ex \mid \exists exfx\}$ и $\mu(x) = \{ex \mid \exists xfex\}$

Если $\eta(ex2) = \eta(ex1)$ и $\mu(ex2) = \mu(ex1)$, то $E_A = E_A / ex2$ и $E_R = E_R \cup ex2$ и $E_M = E_M / ex1$ и $\forall e \in f(ex1) \;\; E_B = E_B / e \;\; E_R = E_R \cup e$.

Нетрудно показать, что предлагаемый алгоритм полностью удовлетворяет всем требованиям. Т.к. блокируются все Executor'ы, которые могут отправить сообщения в момент миграции, то первое свойство выполняется. Так же вместе с состоянием Executor'а мигрируют его сообщения из входных портов, вследствие чего сообщения не теряются. В тоже время при этом негде в момент миграции не происходит дублирования сообщений и выполнение второго свойства, очевидно. Таким образом, предлагаемый алгоритм корректен.

Глава 3. Реализация

3.1. Описание реализации предлагаемой модели управления

В ходе магистерской работы был реализован отдельный программный модуль RuSh, который обеспечивает параллельное исполнение прикладного алгоритма в общей памяти. При этом управление в прикладном алгоритме было задано предложенными средствами.

Языком реализации является C++, для обеспечения параллельного исполнения в общей памяти использовалась библиотека POSIX Threads.

Далее рассмотрим основные структуры, классы и их методы. Надо отметить, что приведенное ниже описание является не полным и более подробную информацию можно получить в Приложении 1.

```
class Fragment {
  public:
        virtual size_t getInputsCount()=0;
        virtual size_t getOutputsCount()=0;
        virtual void execute (Message* outputs);
        virtual void execute (Message* inputs, Message* outputs);
  };
```

Класс Fragment является общим классом для всех Executor’ов. Поэтому для создания Executor’а необходимо создать некоторый класс и определить его как наследника некоторого общего класса Fragment и для него переопределить следующие виртуальные функции.

virtual size_t getInputsCount() - функция, возвращающая количество входных портов.

virtual size_t getOutputsCount() функция возвращающая количество выходных портов.

virtual void execute (Message* imports, Message* outports) - функция запускающая вычисления когда получены сообщения во все порты

virtual void execute (Message* outports) - функция запускающая вычисления в первоначальный момент времени.

```
struct Message
{
        Void* Data;     // указатель на передаваемые данные
        Size_t Size;    // размер передаваемых данных
        Bool IsSet;     // флаг готовности данных
};
```

Для коммуникации между Executor'ами необходимо определять структуру Message. При этом флаг готовности данных является важной составляющей, показывающей что данное сообщение готово к передаче к другому Executor'у.

```
Class RuSh
{
        public:
                void addFragment(Fragment*, bool initially_ready=false);
                void bind(Port src, Port dest);
                void runThreads (size_t threads_num);
};
```

Класс Rush является главным управляющим вычислениями классом. Для инициализации вычислений в него необходимо добавить Executor'ы с помощью функции addFragment , при этом флаг initially_ready определяет первоначальный запуск фрагмента (либо начальный запуск без параметров, либо запуск при получении сообщений во все порты). По умолчанию, у Executor'а задается запуск при получении сообщений. Далее необходимо определить связи между портами для Executor'ов с помощью функции void bind(Port src, Port dest). При этом структура Port содержит указатель на фрагмент и номер порта.

Для запуска параллельных вычислений в общей памяти необходимо запустить функцию void runThreads (size_t threads_num), для которой параметром является количество потоков.

Реализованная версия модуля работает только в общей памяти, а версия для распределенной памяти еще не реализована и её реализация планируется в дальнейшем.

3.2. Описание реализации модели миграции

Для представления миграции в процессе исполнения RuSh-программы будем использовались следующие структуры данных на каждом процессорном элементе.

Таблица *ID: map <int exec_id, Executor* exec>* содержит указатель и соответствующий идентификатор для каждого Executor'а, находящегося на данном узле.

Таблица *Locate: map<int exec_id, int node>* - содержит идентификатор Executor'а и номер узла для тех и только тех Executor'ов, которые либо находятся на данном узле, либо получают/пишут в Executor, находящийся на этом узле.

Таблица *OutExec: table<int ex1, int outport, int inport, int ex2>* - содержит Executor'ы, в которые пишут Executor'ы, находящиеся на данном узле. Поле *ex1* – идентификатор локального Executor'а, поле *ex2* – идентификатор Executor'а, в который пишет *ex1*, поле *outport* – номер выходного порта локального Executor'а, поле *inport* – номер входного порта для *ex2*.

Таблица *InExec: table<int ex1, set<int ex2, int port>>* - содержит информацию для локальных Executor'ов, о том какие Executor'ы в какой порт ему пишут.

Таблица *Blockade <int ex1, int metka, int migrant>* содержит информацию о блокируемых Executor'ах. Поле *ex1* – идентификатор блокируемого Executor'а, поле *metka* – место, с которого начинать после блокировки (этап миграции), поле *migrant* - идентификатор Executor'а, для миграции которого блокируется *ex1* (если *ex1* мигрирует, то *migrant = -1*).

Таблица *RegPort: map <<Executor* , int N>, tag>* .

В начальный момент времени таблица *Blockade* пуста, все остальные заполняются пользователем, исходя из начального распределения.

Изменение состояния структур данных при определенных событиях

А) Событие: миграция $ex1$ на узел $Node$

1) $ex1 \to blocked = 1$

Если ($ex1 \to state \neq 0$) $Blockade.push(ex1,1,-1)$, иначе переход на 2

2) отправляем сообщение на $Node$ о том, что надо создать копию $ex1$.

3)

i) $\forall e': e' \in InExec[ex1]\ P += ID[e']$

ii) $\forall e' \in P$

Если $Locate[e'] = Locate[ex1]$, то

$e' \to blocked = 1$

Если ($e' \to state \neq 0$) $Blockade.push(e',2,ex1)$,

иначе переход на iii

Иначе сообщение на узел $Locate\ [e']$ о блокировке e' для миграции $ex1$

iii) $\forall q : q \in OutPort\ [e', m, q, ex1]\ Port += q$
$\forall q \in Port\ \ q \to blocked\ = 1$

Если $(\forall p : p \in ex1 \to InputPorts\ \ n = 0\ \ \forall m \in InExec\ [ex1, e', p]\ n = n + 1$ $p \to blocked\ = n)$, то переходим на пункт Е).2)

б) Событие $ex1$ завершил исполнение $(ex1 \to state\ = 0)\ metka\ .Blockade\ [ex1] = 1$ $Blockade\ .delete\ (ex1)$ переход на пункт А).2

В) Событие: сообщение о блокировке e' для миграции $ex2$

1) $e' \to blocked\ = 1$

Если $(e' \to state\ \neq 0)\ Blockade\ .push\ (e', 3, ex1)$,

иначе переход на 2)

2) $\forall q : q \in OutPort\ [e', m, q, ex1]\ Port += q$

$\forall q \in Port$ сообщение на узел $Locate\ [ex1]$ о блокировке порта q

Г) Событие: e' завершил исполнение $(e' \to state\ = 0)\ metka\ .Blockade\ [e'] = 2$

$Blockade\ .delete\ (e')$ переход на пункт А).3).iii)

Д) Событие: e' завершил исполнение $(e' \to state\ = 0)\ metka\ .Blockade\ [e'] = 3$

$Blockade\ .delete\ (e')$ переход на пункт В).2)

Е) Событие: блокировка порта q у $ex1$

1) $q \to blocked\ = 1$

Если $(\forall p : p \in ex1 \to InputPorts\ \ n = 0\ \ \forall m \in InExec\ [ex1, e', p]\ n = n + 1$

$p \to blocked\ = n)$, то переходим на пункт 2)

2)Отправляем сообщение на $Node$ о создании копии Maker'а и $\forall b \in Q\ \ b \to exexutor\ = ex1$ отправляем b на $Node$.

3) $\forall ex2 :\ Locate\ [ex2] \neq Node$ отослать на $Node$ $Locate\ [ex2, n]$

Ж) Событие: сообщение о необходимости создать копию $ex1$

$ID.push\ (ex1)$ и *Locate.push(ex1,Node)* и создаем копию $ex1$

З) Событие: пришли Bundle $ex1$

$Bundle\ \to executor\ = ID[ex1]\ Queue\ ->\ push\ (Bundle\ )$

И) Событие: сообщение о создании копии Maker'а $ex1$

$Ma\ker \rightarrow executor = ID[ex1]$ $Ma\ker \rightarrow queue = Queue$

Отправка сообщение, что создался *Maker* для миграции $ex1$

К) Событие: сообщение создался *Maker* для миграции $ex1$

1) $\forall p \in OutExec\ [ex1, m, q, ex2]$ отправить *p* на $Node$ для миграции $ex1$

2) $\forall e' \in InExec\ [ex1, e', p]$

Если $Locate\ [e'] = Locate\ [ex1]$, то $Locate\ [ex1] = Node$ и переход на пункт 3), иначе сообщение на $Locate\ [e']$ для e' о перенаправлении порта p для миграции $ex1$ на $Node$

3) $\forall m \in OutExec\ [e', m, q, ex1]$ создаем $delete\ m, m = new\ Port\ \Pr oxy$ отправляем запрос о регистрации порта на $Node$, и $e' \rightarrow blocked = 0$

4) $\forall e' \in InExec\ [ex1, e', p] \cup OutExec\ [ex1, m, q, e']$

Если $\forall ex \in ID / ex1\ InExec\ [ex] \neq e'$ и $OutExec\ [ex, m, q] \neq e'$, то $Locate.delete\ (e')$ и $OutExec.delete\ (ex1)$

5) $\forall m \in InExec\ [ex1, e', p]$ отправить *m* на $Node$ для миграции $ex1$

$InExec.delete\ (ex1, e', p)$ и $ID.delete\ (ex1)$

$\forall r \in ex1 \rightarrow InputPorts\ \operatorname{Re} gPort.delete\ (ex1, r)$

Удалить $ex1$, *Maker*

Л) Событие: сообщение для e' о перенаправить порта *p* для миграции $ex1$ на $Node$

Отправить сообщение на $Node$ о регистрации в таблицу *RegPort* (e',*p*)

М) Событие: получение записи таблицы $Locate\ [ex2, n]$

Если $\forall e' \in Locate\ [e'] : e' \neq ex2$,то $Locate.push\ (ex2, n)$.

Н) Событие: получение записи таблицы $OutExec$

$\forall p \in OutExec\ [ex1, m, q, ex2]$

Если $Locate\ [ex2] = Node$, то $delete\ m,\ m = new\ Port$, $\operatorname{Re} gPort.delete\ (ex2, q)$

Иначе отправляем сообщение о перерегистрации порта *p* для $ex1$

Для тестирования реализации механизма миграции для поддержки динамической балансировки загрузки был разработан прототип системы RuSh в распределенной памяти, который не обеспечивал необходимую производительность при исполнении RuSh-программ, но вполне годился для тестирования механизма миграции. Тестирование механизма миграции проводилось на синтетическом наборе задач, что служит еще одним аргументом, что при доказательстве корректности не было допущено ошибок.

Глава 4. Тестирование и сравнительный анализ производительности

Тестирование производилось на одном из двух параллельных вычислителях:

Вычислитель 1

Компьютер с 6 ядрами (Intel® Xeon® CPU X5600 2.8 Ghz)

Вычислитель 2

Кластер, принадлежащий ССКЦ СО РАН, г. Новосибирска, NKS-160 со 64 двойным блейд-сервером HP BL2x220 G6 и характеристиками, приведенными в Таблице 1.

Таблица 1 – Характеристики кластера NKS-160

	Количество	Характеристики
Вычислительный модуль	128	ОП модуля 16 Гбайт
Процессор (Intel Xeon E5540)	256 (1024 ядер)	2.53 Ггц (Nehalem), 8 ядер
Производительность		10,36 Тфлопс

Так как реализации RuSh-программ исполняются в общей памяти, а MPI-реализации в распределенной памяти, то для объективного сравнения при тестировании с MPI-реализациями все MPI процессы запускались на одном узле, чтобы свести к минимуму накладные расходы, связанные с доступом к памяти между процессами.

4.1. Метод Якоби для решения уравнения Пуассона

Тестирование производилось на Вычислителе 1. Сетка 200x200.

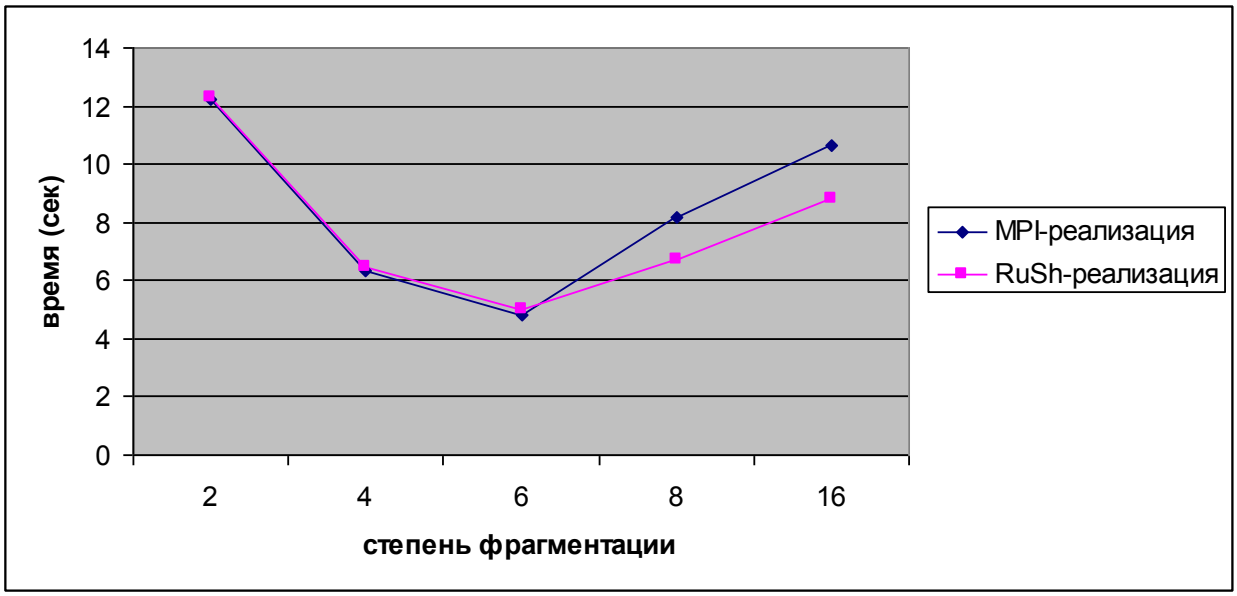

Рис. 8 - Сравнение зависимости времени исполнения от степени фрагментации для MPI и RuSh реализаций

Как видно из приведенного Рис. 8, коэффициент распараллеливания для MPI и RuSh реализаций практически совпадает, что говорит о хорошей эффективности Rush-реализации. Можно заметить, что после степени фрагментации 6 время исполнения не уменьшается (это связанно с тем, что тестирование проводилось на 6-ядерном компьютере и дальнейшая фрагментация не помогает, т.к. уже нет свободных процессоров или ядер), а даже начинает увеличиваться из-за увеличения накладных расходов связанных с фрагментацией алгоритма. При этом время работы MPI-реализаций увеличивается быстрее. Вероятно, это связано с тем, что переключение MPI-процессов является более накладным, чем переключение потоков.

4.2. Реакция окисления CO на поверхности палладия

Тестирование проводилось на Вычислителе 2. Размер сетки 4992x4992, 1000 итераций моделирования. Такие параметры являются близкими к реально используемым [14] и поэтому позволяют судить о характеристиках программ в условиях, приближенных к реальным.

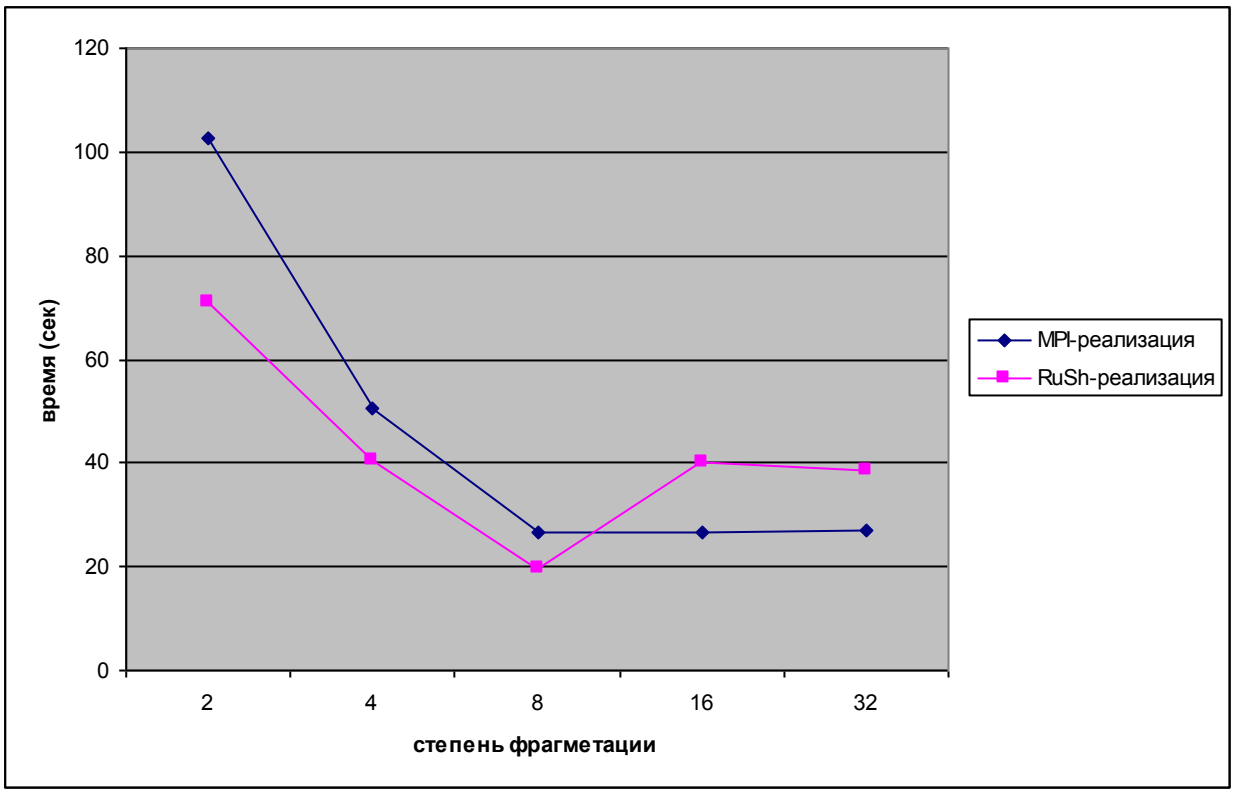

Рис. 9 – Сравнение зависимости времени исполнения от степени фрагментации MPI и RuSh реализаций

Из Рис. 9 можно сделать вывод, что время исполнения RuSh-реализаций сопоставимо с временем MPI-реализации. Прекращение уменьшения времени исполнения MPI и RuSh реализаций, начиная со степени фрагментации 16, связано с исполнением на 8 ядерном компьютере. Разница времени исполнения MPI- и RuSh- реализациями на 16 фрагментах и выше, может быть, связана с накладными расходами системы RuSh.

4.3. Метод «частиц в ячейках»

Для тестирования метода частиц в ячейках использовались два разных теста:

Данные А. Размер сетки 64x64x64, количество 1000000 частиц, 1000 временных итераций.

Данные В. Размер сетки 300x300x300, количество частиц 1000000, число итераций моделирования 100

Такие параметры являются близкими к реально используемым [11] и поэтому позволяют судить о характеристиках программ в условиях, приближенных к реальным.

Необходимо было провести следующие тесты.

1) Чтобы оценить накладные расходы, связанные с фрагментацией алгоритма, нужно было сравнить время работы последовательной реализации и реализации RuSh-программы. А затем найти оптимальное потоков и

разбиение области, при котором достигается максимальное ускорение и оценить полученный выигрыш в производительности по сравнению с последовательной реализацией.

2) Провести исследование масштабируемости: для разной степени фрагментации, найти оптимальное количество потоков и сравнить время работы программы. Под масштабируемостью понимается способность системы сохранять относительно низкий уровень накладных расходов при увеличении количества вычислительных ресурсов. Масштабируемость является важной характеристикой RuSh-программы.

3) Так как в требованиях к модели управления оговаривается эффективность исполнения на суперкомпьютере, а именно производительность сопоставимая с реализацией задачи с использованием MPI, то необходимо было провести сравнительное тестирование времени исполнения и эффективности распараллеливания RuSh-реализации и MPI-реализации метода «частиц в ячейках» в зависимости от степени фрагментации.

4.3.1. Тест 1

Данный тест выполнялся на Вычислителе 2 с использованием Данных А.

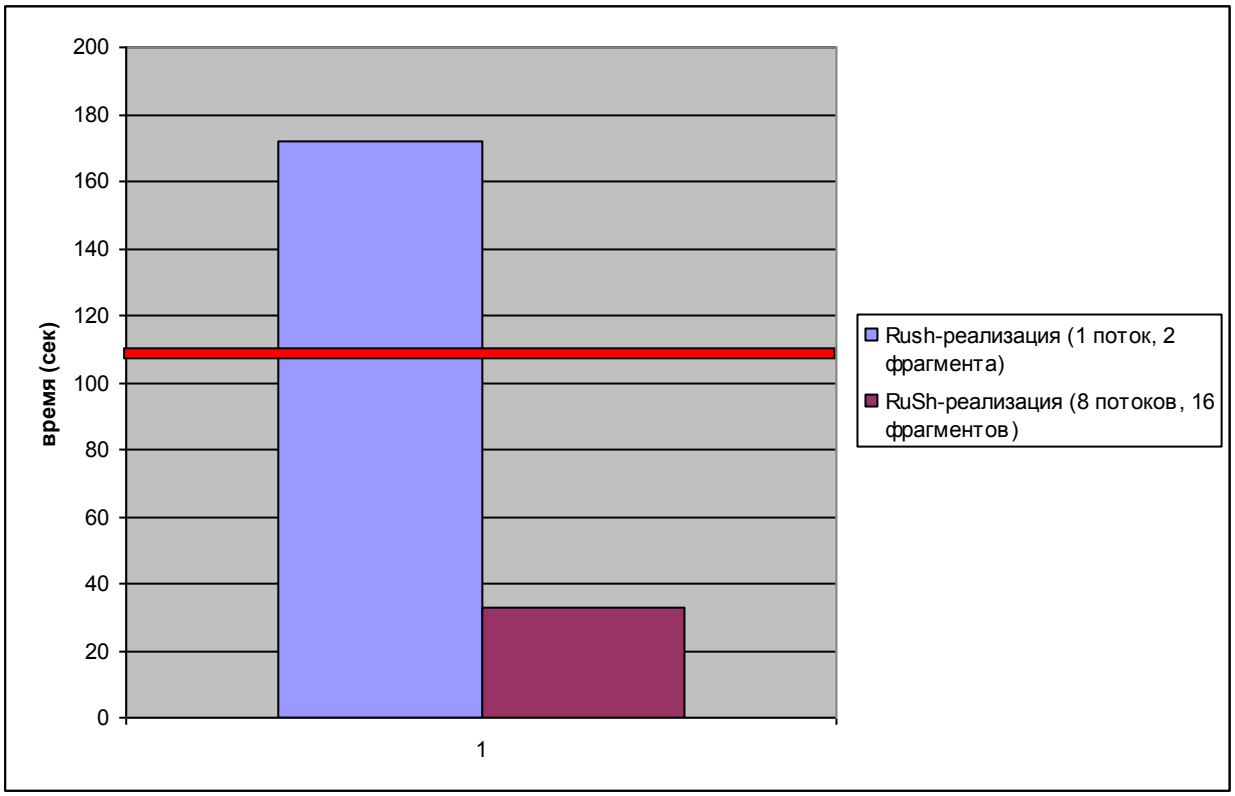

Рис. 10 - Сравнение последовательной и реализаций RuSh-программы (красная горизонтальная линия – время последовательной реализации)

На Рис. 10 изображено время работы реализации RuSh-программы при использовании 1 потока и минимальным разбиением области. Нижний столбец – это время при оптимальных параметрах. Ими оказались 8 потоков и 16 фрагментов. А горизонтальной чертой показано время работы последовательной программы.

Видно, что даже на 1 потоке RuSh-реализация не сильно уступает последовательной. Она работает медленнее из-за усложнения алгоритма после его фрагментации. Параллельная же реализация работает быстрее, чем последовательная, что позволяет сделать вывод о приемлемой производительности нашего модуля RuSh, а значит и применимости предлагаемой модели управления.

4.3.2. Тест 2

Данный тест выполнялся на Вычислителе 2 с использованием Данных А.

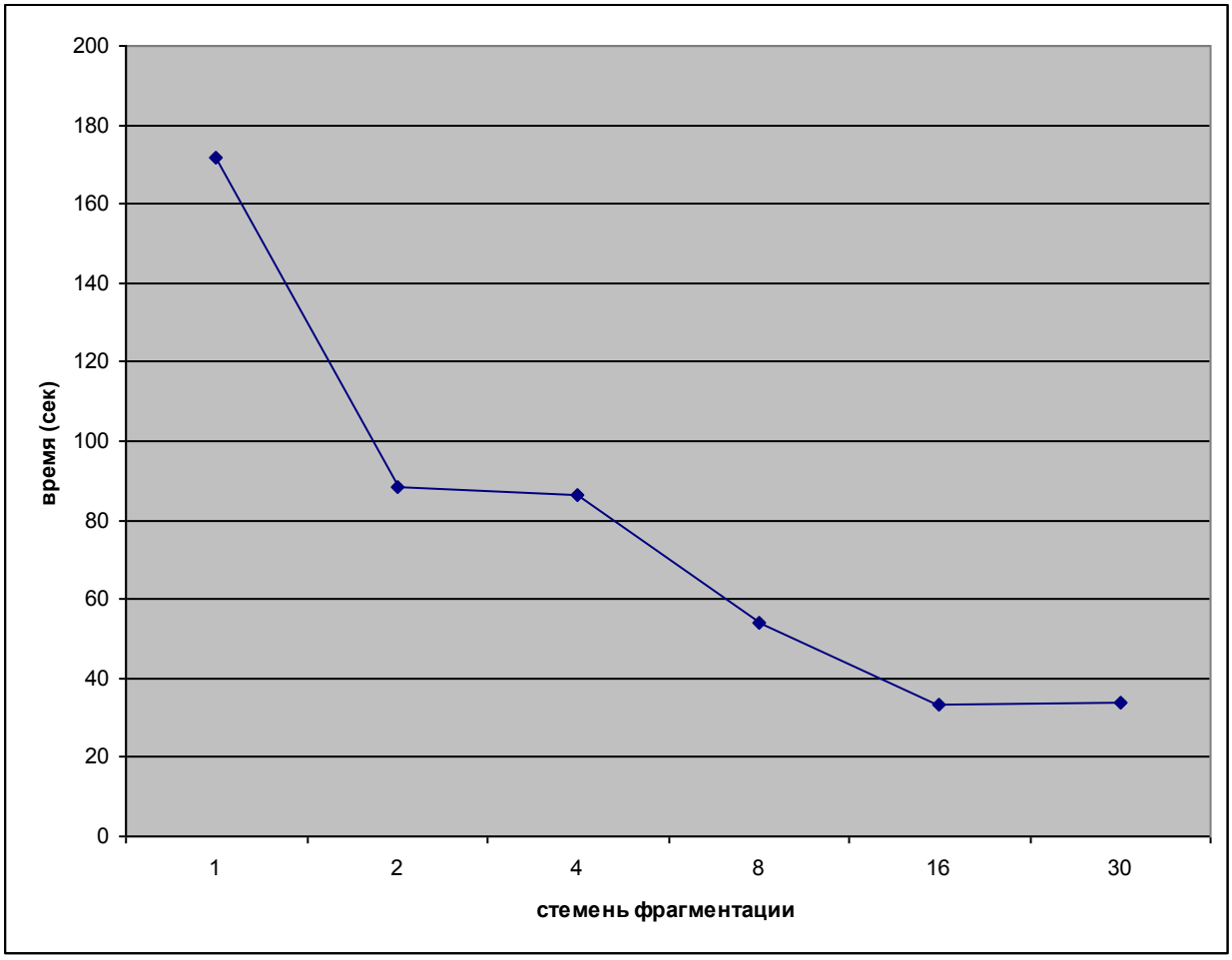

Рис. 11 - Сравнение зависимости времени исполнения RuSh-программы метода «частиц в ячейках» от степени фрагментации

На приведенном графике для степени фрагментации равной 1 взято значение при использовании минимального разбиения области и 1 потока как приближение к последовательной реализации.

На 8-ядерном компьютере в идеале можно ожидать 8-кратного ускорения. В нашем случае RuSh-реализация ускоряется максимум в 5.24 раза. Это может быть объяснено двумя причинами. Во-первых, в тестах используется сетка 64 по разрезаемому направлению, и это ограничивает максимально возможную степень фрагментации, а также долю накладных расходов, связанных с фрагментацией. Во-вторых, решаемая задача относится к задачам с большим количеством обращений к памяти и небольшим количеством вычислений, поэтому узким местом для данной реализации является, скорее всего, пропускная способность шины памяти. Поэтому ускорение в 5,24 раза является вполне приемлемым в таких обстоятельствах результатом.

4.3.3. Тест 3

Данный тест выполнялся на Вычислителе 2 с использованием Данных B.

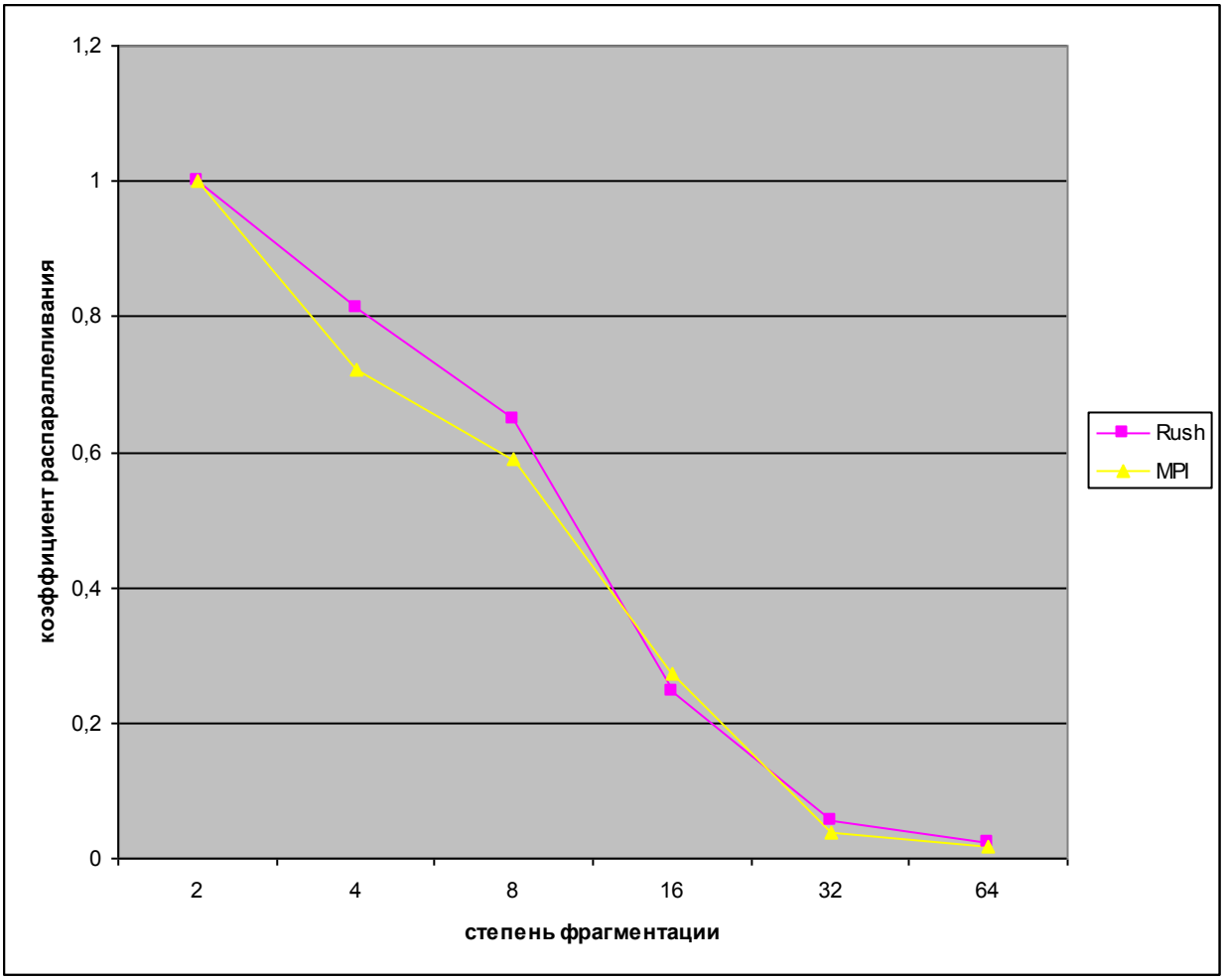

Рис. 12 – Сравнение коэффициента распараллеливания в зависимости от степени фрагментации для MPI и RuSh реализаций

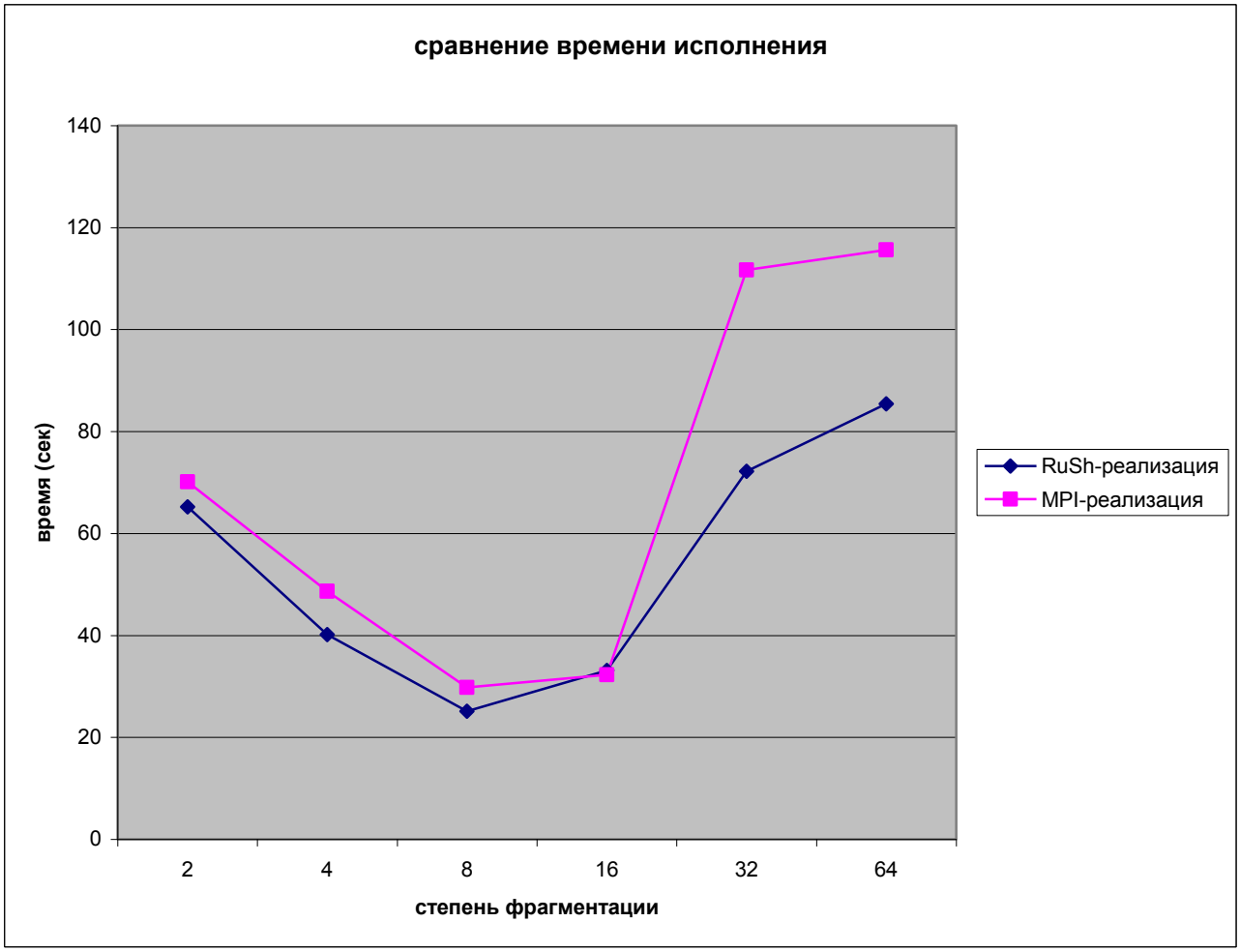

Рис. 13 – Сравнение зависимости времени исполнения MPI и RuSh реализаций метода «частиц в ячейках» от степени фрагментации

Как видно из приведенных данных коэффициент распараллеливания MPI-реализации и RuSh-реализации практически совпадает и производительность RuSh-реализаций сравнима с производительностью MPI-реализации, что является хорошим результатом и говорит о том, что предлагаемая модель управления и разработанный модуль позволяют эффективное исполнение RuSh-программы на мультикомпьютере.

Заключение

В результате магистерской работы была предложена модель представления фрагментированного алгоритма на основе Сетей Петри, включающая средства для задания императивного управления, называемая RuSh-программа. реализован отдельный программный модуль RuSh, который обеспечивает параллельное исполнение RuSh-программы в общей памяти.

Был разработан и реализован алгоритм миграции частей RuSh-программы между узлами вычислителя, доказана его корректность и также проведено тестирование работоспособности на синтетическом наборе тестов.

Были разработаны RuSh-программы для следующих численных задач: решение уравнения Пуассона методом Якоби, моделирование самогравитирующегося вещества методом «частиц в ячейках», реакция окисления *CO* на поверхности палладия блочно-синхронным клеточным автоматом. Проведено исследование производительности мультикомпьютера при реализации этих RuSh-программ и сделан сравнительный анализ производительности RuSh-программ с аналогичными MPI-реализациями этих задач. Производительность реализаций в модуле RuSh для рассматриваемых численных задач оказалась сравнима с производительностью MPI-реализации, что говорит о том, что предлагаемая модель управления и разработанный модуль позволяют эффективное исполнение RuSh-программы на мультикомпьютере.

Работа была представлена на конференциях МНСК “Студент и научно-технический прогресс”(г.Новосибирск) 2012 и 2013 годов, а также на XIII Всероссийской конференции молодых ученых по математическому моделированию и информационным технологиям”(г.Новосибирск).

На защиту выносятся следующие результаты:

- Модель для описания императивного управления во фрагментированных алгоритмах;
- алгоритм миграции частей RuSh-программы для поддержки динамической балансировки загрузки процессоров с доказательством его корректности;
- исследование эффективности реализации модуля RuSh на ряде численных алгоритмов;

В дальнейшем планируется реализация модуля RuSh для распределенной памяти и исследование предложенной модели на примере других численных алгоритмах и конфигурациях вычислителей.

Список используемой литературы

1. Seibel P. Practical Common LISP, APRESS, 2005
2. Стерлинг Л. Шапиро Э. Искусство программирования на языке Пролог, Москва: МИР, 1990 – 235
3. Malyshkin V.E., Perepelkin V.A. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem.— In the Proceedings of the 11-th Conference on Parallel Computing Technologis, LNCS 6873, Springer, 2011. — pp. 53–61.
4. SMP Superscalar [Электронный ресурс] http://www.bsc.es/computer-sciences/programming-models/smp-superscalar
5. Grid Superscalar [Электронный ресурс] http://www.bsc.es/computer-sciences/grid-computing/grid-superscalar
6. Cell Superscalar [Электронный ресурс] http://www.bsc.es/computer-sciences/programming-models/cell-superscalar
7. Charm++ [Электронный ресурс] http://www.charm.cs.uiuc.edu
8. OpenTS [Электронный ресурс] http://www.opents.net/index.php/ru
9. Абрамов С.М., Загоровский И.М., Коваленко М.Р., Матвеев Г.А., Роганов В.А. Миграция от MPI к платформе OpenTS: эксперимент с приложениями PovRay и ALCMD – Программные системы: теория и приложения, Переславль-залесскии, 2006
10. Алгоритм Кэннона [Электронный ресурс]http://www.hpcc.unn.ru/?dir=828
11. С.Е. Киреев. Параллельная реализация метода частиц в ячейках для моделирования задач гравитационной космодинамики. “Автометрия”, СО РАН, 2006, т. 42, № 3, с. 32-39
12. E.I. Latkin, V.I. Elokhin, A.V. Matveev, V.V. Gorodetskii, The role of subsurface oxygen in oscillatory behaviour of $CO+O_2$ reaction over Pd metal catalysts: Monte Carlo model, Journal of Molecular Catalysis A: Chemical 158, 2000, pp. 161–166
13. V.V. Gorodetskii, V.I. Elokhin, J.W. Bakker, B.E. Nieuwenhuys, Field electron and field ion microscopy studies of chemical wave propagation in oscillatory reactions on platinum group metals, Catalysis Today 105, 2005, pp 183–205
14. Маркова В.П., Шарифулина А.Е. Параллельная реализация кинетического клеточного автомата, моделирующего реакцию окисления CO на PD – “Прикладная дискретная математика” №1(11), 2011.