МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

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

Образовательная программа: 09.03.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

РАЗРАБОТКА АЛГОРИТМОВ АНАЛИЗА ПРОИЗВОДИТЕЛЬНОСТИ ФРАГМЕНТИРОВАННЫХ ПРОГРАММ

утверждена распоряжением проректора по учебной работе №309 от «06» декабря 2016 г.

Литвинов Василий Сергеевич, группа 13201 ________________
(подпись студента)

«К защите допущена»	**Руководитель ВКР**
Заведующий кафедрой Параллельных вычислений ФИТ НГУ,	д.т.н., профессор,
д.т.н., профессор	зав. кафедрой Параллельных вычислений ФИТ НГУ, зав. лабораторией ИВМиМГ СО РАН
Малышкин Виктор Эммануилович / …….. (подпись)	Малышкин Виктор Эммануилович / .... (подпись)
«……»………………20…г.	«……»………………20…г.

Дата защиты: «21» июня 2017 г.

Новосибирск, 2017г.

МИНОБРНАУКИ РОССИИ

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

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В.Э.

…………………
(подпись)
«……»…………………20…г.

ЗАДАНИЕ

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Литвинову Василию Сергеевичу, группы 13201

Тема «Разработка алгоритмов анализа производительности фрагментированных программ»

утверждена распоряжением проректора по учебной работе от……………………..№………

Срок сдачи студентом готовой работы………………………20.. г.

Исходные данные (или цель работы): разработка алгоритмов анализа исполнения фрагментированных программ и создание инструмента, позволяющего получить характеристики исполнения программ в системе LuNA, определение которых позволит программисту сделать выводы о вероятных причинах проблем с производительностью.

Структурные части работы: изучение технологии фрагментированного программирования и системы фрагментированного программирования LuNA, обзор средств профилирования параллельных программ, выбор целевых характеристик исполнения фрагментированных программ, разработка алгоритма получения характеристик, реализация системы анализа исполнения фрагментированных программ для системы LuNA.

Руководитель ВКР
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э. /…………..

«…»…………………20…г.

Задание принял к исполнению
Литвинов В.С. /…………..

«…»…………………20…г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **4**

1 Обзор средств профилирования параллельных программ **6**

1.1 MPE 6

1.2 Extrae / Paraver 7

1.3 Intel Trace Analyzer and Collector 7

1.4 Scalasca (KOJAK) 8

1.5 LuNA Evlog Analyze 9

1.6 Результаты обзора 9

2 Постановка задачи **11**

2.1 Цели и задачи работы 11

2.2 Выбор представления фрагментированного алгоритма 11

2.3 Выбор характеристик исполнения 13

3 Разработка алгоритма получения характеристик исполнения **15**

3.1 Построение модели исполнения фрагментированной программы 15

3.2 Алгоритм получения характеристик исполнения 17

4 Реализация системы анализа исполнения фрагментированных программ **21**

4.1 Описание реализации 21

4.2 Тестирование системы анализа исполнения 23

4.2.1 Тест для Starvation 23

4.2.2 Тест для Overhead 23

4.2.3 Тест для Latency 25

4.2.4 Анализ LuNA-реализации метода частиц-в-ячейках 26

ЗАКЛЮЧЕНИЕ **28**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ **29**

ВВЕДЕНИЕ

В современной науке, технике, медицине часто возникают задачи, решение для которых представляется в виде численного алгоритма. Их реализация требует большого количества вычислений, за приемлемое время осуществимых лишь с помощью высокопроизводительных вычислительных систем. Это влечет за собой необходимость распараллеливания алгоритмов и программ, их реализующих.

Основная проблема параллельного подхода в случае реализации сложных численных алгоритмов заключается в том, что на плечи программиста ложится задача по распределению данных и вычислений по узлам вычислителя, и сложность этих задач может превосходить квалификацию программиста. Вдобавок к этому идет сложность отладки параллельных программ по сравнению с последовательными, обусловленная недетерминизмом исполнения и необходимостью следить сразу за несколькими потоками управления. Таким образом, программисту приходится реализовывать динамические системные функции, которые напрямую не относятся к исходной задаче.

Разработанные в ИВМиМГ СО РАН технология фрагментированного программирования [1, 2] и система фрагментированного программирования LuNA [3, 4] позволяют спрятать от программиста часть описанных выше проблем и, таким образом, упростить процесс создания параллельных программ. Смысл упомянутой технологии заключается в автоматическом обеспечении динамических свойств параллельной программы, таких как настройка на имеющиеся аппаратные ресурсы мультикомпьютера, динамическая балансировка нагрузки, осуществление коммуникационных взаимодействий на фоне вычислений. В технологии фрагментированного программирования параллельная программа представлена как множества фрагментов данных и фрагментов вычислений с явно определёнными связями. Такое фрагментированное представление сохраняется вплоть до момента исполнения программы. Система исполнения обеспечивает удовлетворение информационных зависимостей между фрагментами вычислений и их работу на множестве вычислителей.

Критерием, позволяющим сравнить качество программ, направленных на решение одной задачи, помимо корректности результатов их работы, может служить также и время их работы при одинаковых входных данных. В случае параллельных программ оно зависит от времени, затраченного на исполнение последовательного кода и от масштабируемости программ.

Обнаружение причин плохой производительности и масштабируемости параллельной программы – трудоемкая задача из-за обилия возможных факторов, требующая проверки многих гипотез. В частности, ее нужно решать и для фрагментированных программ. В настоящее время при использовании системы LuNA для решения различных прикладных задач [5, 6] остаются невыясненными причины получаемых характеристик производительности. Решение этой задачи можно упростить, если указать начальную область поиска. Для этого создаются профилировщики – инструменты, собирающие характеристики исполнения программы в профиль, затем анализирующие его и дающие представление об использовании процессора, памяти, сетевой подсистемы, о временных затратах на выполнение того или иного участка программы и т.д. на разных уровнях абстракции: аппаратная среда, операционная система, язык или система программирования и т.п. Также с помощью профилировщиков может быть получено множество интегральных характеристик: для определенного промежутка времени, для множества ресурсов, для частей программы на уровне абстракций самой системы параллельного программирования, позволяющее оценить вклад факторов, влияющих на производительность и масштабируемость.

Полезными для программиста являются такие характеристики исполнения, которые

- объясняют полученную производительность, указывают на возможные проблемы, ограничивающие производительность программы;
- дают информацию в терминах, которыми пользуется программист.

Будем называть такие характеристики высокоуровневыми - по сравнению с характеристиками в терминах вычислительной системы (например, машинные команды, операции коммуникационной подсистемы) или с характеристиками, не позволяющими оценить качество исполнения (например, количество выполненных операций, количество межузловых обменов).

1 Обзор средств профилирования параллельных программ

Рассмотрим существующие средства для профилирования параллельных программ, проанализируем их с точки зрения применимости к анализу фрагментированных программ на языке LuNA и получению высокоуровневых характеристик исполнения.

1.1 MPE

MPE (MPI Parallel Environment) – это свободно распространяемый пакет для профилирования MPI-программ, основанный на анализе трасс исполнения (post-mortem analyzer) [7]. Он включает в себя:

- Библиотеки с подпрограммами для сбора данных о ходе выполнения (профилей) программы и их обработки.
- Скрипты для компиляции программы с поддержкой MPE, утилиты для конвертации профилей в разные форматы, для их фильтрации, распечатки и т.д.
- Графический инструмент для отображения результатов профилирования Jumpshot.

Запись трасс исполнения в MPE может работать в двух режимах: автоматическом, когда собирается информация о каждой вызванной MPI-подпрограммы, и в ручном, в котором программист описывает модель программы в виде конечного автомата, и в таком случае записи подлежат события, соответствующие моменту смены состояния автомата. С помощью утилиты Jumpshot (Рисунок 1) можно отобразить статистику различных событий, детализацию профиля в заданной временной области, в которой представлены моменты начала или конца работы MPI-подпрограммы, моменты смены состояния автомата, пути пересылки данных между узлами.

Разноцветными прямоугольниками на рисунке показаны временные интервалы, соответствующие полезным вычислениям и работе MPI-подпрограмм в MPI-процессах. Желтым отрезкам соответствуют межпроцессные коммуникации.

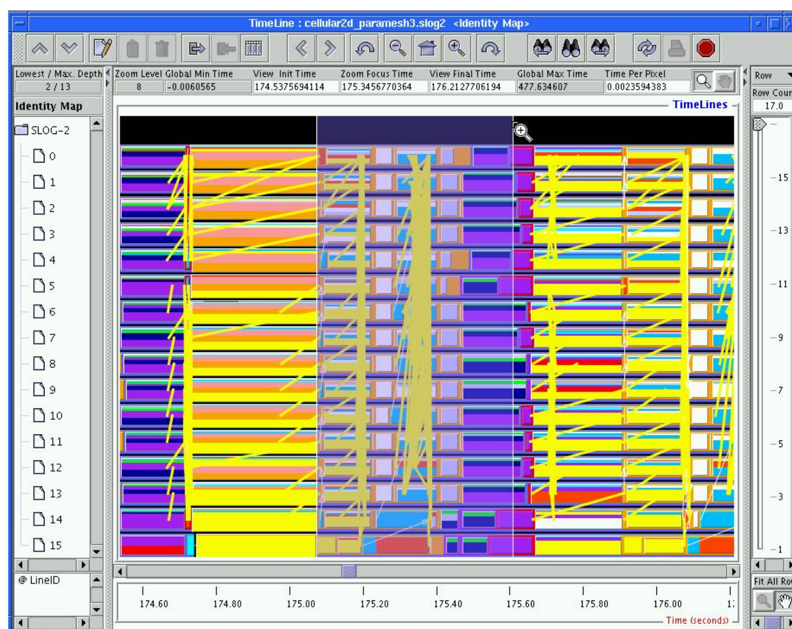

Рисунок 1 – Графический интерфейс утилиты Jumpshot

1.2 Extrae / Paraver

Эта пара инструментов предназначена для анализа работы MPI-программ: Extrae используется для записи профиля [8], Paraver – для его визуализации в виде временных диаграмм в нескольких режимах [9]:

- отображение поведения MPI-программы аналогично Jumpshot,
- отображение производительности в flops для заданного процесса,
- отображение количества инструкций в секунду (IPS) на вычислителе для заданного процесса.

Интерфейс программы поддерживает открытие нескольких видовых окон, визуализирующих профиль в разных согласованных по времени режимах. Тем не менее, рассматриваемые характеристики выполнения программы относятся к низкоуровневым, их интерпретация трудна и в большинстве случаев может занять значительное время.

1.3 Intel Trace Analyzer and Collector

Этот проприетарный программный пакет нацелен на сбор и анализ профиля выполнения MPI-программы. Основу составляют утилиты со следующими возможностями [13]:

- Визуализация поведения MPI программы, аналогичная представленной в Jumpshot.

- Вычисление статистики по событиям, оценка сбалансированности загрузки вычислителя.
- Анализ времени работы отдельных подпрограмм.
- Выделение участков программы, активно использующих подпрограммы для коммуникации между узлами, выявление в этих участках шаблонов взаимодействия процессов, негативно сказывающихся на производительности.
- Получение т.н. “MPI performance snapshot” – нескольких чисел, характеризующих выполнение параллельной программы в целом, а также оценка времени, затраченного на полезный счет, на выполнение подпрограмм параллельного API.
- Симуляция исполнения программы с нулевыми задержками и бесконечной пропускной способностью в сетевой подсистеме. Сравнивая полученный в результате симуляции профиль и первоначальный, Intel Trace Analyzer может указать на сетевую подсистему как на одну из причин низкой производительности.
- Проверка корректности вызовов MPI-подпрограмм и обнаружение таких ошибок программирования, как выход за границы буфера, возможные ситуации взаимной блокировки процессов.

1.4 Scalasca (KOJAK)

Система Scalasca [11] (и ее предшественник KOJAK [10]) направлена на анализ производительности программ, использующих MPI, OpenMP, POSIX threads [10]. Ее отличительная черта – возможность автоматически находить причины неэффективного исполнения программ на высоком уровне абстракции. Это осуществляется с помощью параллельной схемы анализа трасс исполнения, состоящей из двух этапов:

1. Редукции больших по объему профилей с информацией об элементарных событиях в более компактное высокоуровневое промежуточное представление, позволяющее абстрагироваться от конкретного программного интерфейса для параллельного программирования.
2. Анализ созданного представления, поиск в нем заранее известных шаблонов взаимодействия процессов, снижающих производительность программы, разделение их по категориям, упорядочивание по важности и привязка к исходному коду программы.

Примеры обнаруживаемых шаблонов: неправильный порядок приема-передачи информационных зависимостей, слишком ранний вызов блокирующих подпрограмм

редукции на некоторых узлах, поздний вызов блокирующих подпрограмм широковещательной коммуникации и т.д. Также существует расширение, позволяющее проводить анализ с учетом топологии вычислителя: судить о ходе исполнения программы в контексте выбранной схемы распараллеливания, учитывать локальность взаимодействий.

1.5 LuNA Evlog Analyze

Данный инструмент разработан в рамках проекта по разработке системы LuNA и предназначен для анализа генерируемого системой исполнения профиля LuNA-программ в JSON-формате, содержащего информацию о событиях, связанных с:

- управлением фрагментами данных (создание, удаление, передача по сети, ...),
- управлением и исполнением фрагментов вычислений,
- динамической балансировкой нагрузки,
- другими действиями системы исполнения.

LuNA Evlog Analyze позволяет извлечь из полученного профиля некоторые интегральные характеристики, построить зависимости этих характеристик от времени работы программы с заданным шагом по времени, с заданной функцией для редукции множества значений в каждом временном интервале. Также возможно задание предиката для выборки определенных событий, учитываемых при анализе, например, по времени, номеру процесса, рабочего потока, по имени фрагмента. Примеры извлекаемых характеристик: объем оперативной памяти, занимаемой фрагментами данных, объем передаваемых по сети фрагментов данных, темп передачи фрагментов данных или фрагментов вычислений по сети, количество фрагментов вычислений, готовых к выполнению, количество одновременно выполняющихся фрагментов вычислений и т.п.

1.6 Результаты обзора

Рассмотренные выше инструменты MPE, Extrae/Paraver, Intel Trace Analyzer and Collector, Scalasca (KOJAK) имеют развитые средства для анализа полученной производительности, но работают с общей моделью параллельных программ, основанной на низкоуровневых сущностях операционной системы – процессах и потоках, поэтому их применение для профилирования фрагментированных программ затруднено из-за более низкого уровня абстракции.

Существующая в системе LuNA утилита Evlog Analyze учитывает специфику фрагментированного программирования и может предоставлять информацию в терминах фрагментов данных и вычислений, но информация на ее выходе все еще слишком низкоуровневая - на ее основе нельзя сделать предположения о причинах полученной производительности и масштабируемости, о причинах простоев вычислительных ресурсов.

По результатам обзора можно сделать вывод, что пока нет инструмента, способного указать на недостатки фрагментированной программы и ее окружения, ставшими причинами простоев и, возможно, ухудшающих масштабируемость, в терминах, специфичных для технологии фрагментированного программирования.

2 Постановка задачи

2.1 Цели и задачи работы

Целью данной работы является разработка алгоритмов анализа исполнения фрагментированных программ и создание инструмента, позволяющего получить высокоуровневые характеристики исполнения LuNA-программ, определение которых позволит программисту сделать выводы о возможных проблемах с производительностью фрагментированных программ.

В рамках данной работы было предложено осуществить выбор характеристик на основании факторов, снижающих масштабируемость параллельных программ. Множество факторов предложено взять из работы [12]. В качестве характеристик было предложено принять вклад этих факторов в формирование простоев вычислительных ресурсов.

Для достижения цели работы ставятся следующие задачи:

1. Оценить, можно ли на основе предложенных характеристик сделать выводы о возможных проблемах с производительностью фрагментированной программы. Сформулировать возможные выводы для различных значений характеристик.
2. Построить упрощенную модель исполнения фрагментированного алгоритма, достаточную для определения значений выбранных характеристик. Данная модель должна быть универсальной в том смысле, что она не должна зависеть от конкретной реализации исполнительной системы.
3. Построить алгоритм получения выбранных характеристик по заданному исполнению в соответствии с построенной моделью.
4. Реализовать полученный алгоритм применительно к LuNA-программам.
5. Проверить работоспособность выбранного подхода к оценке производительности фрагментированных программ с помощью тестирования.

2.2 Выбор представления фрагментированного алгоритма

Программа на языке LuNA - это фрагментированная программа (ФП), в основе которой лежит представление алгоритма в виде потенциально бесконечного ориентированного графа (назовем его “граф алгоритма”) с фрагментами данных, фрагментами вычислений и информационными зависимостями - отношениями “вход” и “выход”. Фрагменты данных в этом представлении являются переменными единственного

присваивания, фрагменты вычислений являются операциями однократного срабатывания. Кроме того, это представление включает набор средств, позволяющих представить любую вычислимую функцию, а также обеспечивающих удобство записи численных алгоритмов (операторы типа if, for, while, массивы, подпрограммы). Эти средства позволяют представить потенциально бесконечный граф алгоритма в виде записи конечного размера. Исполнительная система (ИС, runtime-система) системы фрагментированного программирования LuNA реализует ФП на мультикомпьютере и обеспечивает динамические свойства ФП. В ходе исполнения запускаются те ФВ, для которых все входные ФД получили значения. После выполнения ФВ значения получают его выходные ФД. Таким образом, ход выполнения ФП определяется информационными зависимостями.

Исполнительная система, исполняя фрагментированную программу, реализует некоторый конкретный граф алгоритма конечного размера. В этом графе уже сделаны все выборы пути исполнения, раскручены все циклы и т.п. Для целей профилирования достаточно будет рассмотреть только этот конечный граф, полученный в результате исполнения фрагментированной программы. Будем далее его называть “граф исполнения”. В рамках работы можно считать, что исполнение фрагментированной программы есть исполнение алгоритма, представленного в виде графа исполнения, игнорируя получение этого графа из фрагментированной программы.

Будем использовать следующее представление фрагментированного алгоритма. Фрагментированный алгоритм - это четверка *(X, F, in, out)*, где:

X – конечное множество фрагментов вычислений (ФВ),

F – конечное множество фрагментов данных (ФД),

$in \subset X \times F$ – отношение “вход”,

$out \subset X \times F$ – отношение “выход”.

Будем говорить, что “ФВ зависит от ФД” или “ФД является входным для ФВ”, если они находятся в отношении in. Будем говорить, что “ФВ вычисляет (инициализирует) ФД”, или “ФД вычисляется (инициализируется) ФВ”, или “ФД является выходным для ФВ”, если они находятся в отношении out.

Отношение out обладает следующим свойством: $\forall u = (c, d) \in out\ \forall v = (c_1, d_1) \in out$ выполняется $d = d_1 \Rightarrow c = c_1$. Другими словами, любой фрагмент данных вычисляется не более чем одним фрагментом вычислений.

Другим образом фрагментированный алгоритм можно представить как ориентированный ациклический двудольный граф — граф исполнения, вершинами которого

являются ФВ и ФД, дуги которого соединяют ФВ с ФД либо ФД с ФВ, и входящая степень всех ФД-вершин равна 1.

2.3 Выбор характеристик исполнения

В соответствии с работой [12], факторами, ограничивающими масштабируемость параллельных программ, являются:

1. **S**tarvation – отсутствие работы на данном этапе исполнения вследствие низкой степени параллелизма в алгоритме, из-за чего не полностью используются доступные вычислительные ресурсы.
2. **L**atency – задержки, обусловленные доступом к данным, находящимся на другом узле вычислителя.
3. **O**verhead – работа по решению системных задач параллельного программирования (выбор управления и распределения ресурсов), которой не возникает в последовательной программе.
4. **W**aiting for contention resolution – задержки при синхронизации доступа к общим данным.

Далее будем для обозначения этих факторов использовать аббревиатуру SLOW.

На основе перечисленных факторов введем следующие характеристики исполнения фрагментированной программы:

1. TStarvation (TS) в фрагментированной программе – доля времени простоя ресурсов, связанная с отсутствием на узле готовых к исполнению ФВ.
2. TLatency (TL) в фрагментированной программе – доля времени простоя ресурсов при ожидании готовых ФД, являющимися входными для ФВ.
3. TOverhead (TO) – доля времени простоя ресурсов, вызванного работой исполнительной системы.
4. Фактор “Waiting for contention resolution” применительно к технологии фрагментированного программирования не имеет смысла, так как ФД неизменяемы, получают значение только от одного фрагмента вычислений, и при доступе к ним нет конкуренции.

Будем использовать аббревиатуру SLO для обозначения указанных характеристик.

Если определить значения данных характеристик, то можно будет судить о следующих возможных недостатках алгоритма или системы исполнения:

1. Преобладает TS: низкая степень параллелизма в алгоритме, крупная фрагментация алгоритма, плохое распределение ФВ по узлам вычислителя. В таких случаях, если позволяет алгоритм, можно уменьшить размеры ФД, увеличив их количество, и занять простаивающие ресурсы их выполнением.
2. Преобладает TL: медленная коммуникационная среда, плохое распределение ФД относительно ФВ по процессам, из-за которого нарушается принцип локальности данных.
3. Преобладает TO: мелкая фрагментация алгоритма (большое количество мелких фрагментов), неэффективная работа системы исполнения.

Таким образом, характеристики SLO, соответствующие вкладу каждого из факторов SLOW в интервалы простоя, позволяют сделать выводы о возможных проблемах с производительностью фрагментированной программы.

3 Разработка алгоритма получения характеристик исполнения

3.1 Построение модели исполнения фрагментированной программы

Для построения универсальной модели, не зависящей от конкретной реализации исполнительной системы, опишем упрощенную виртуальную машину, работа которой будет моделировать работу реальной исполнительной системы. Виртуальная машина выполняет вычисления в соответствии с графом исполнения на некоторых ресурсах, частично повторяющих реальные вычислительные ресурсы.

Считаем, что среда исполнения состоит из N вычислительных узлов $r_1, \ldots, r_N$, на каждом i-ом узле работают T_i рабочих потоков $t_i^1, \ldots, t_i^{Ti}$ (по числу T_i процессорных ядер на узле), в которых исполняются фрагменты вычислений. Кроме того, на каждом узле работают сервисные потоки системы исполнения и другие процессы и потоки операционной системы, которые работают в режиме разделения времени с рабочими потоками.

Каждому фрагменту данных или фрагменту вычислений в исполнительной системе соответствует уникальный номер. Для каждого фрагмента вычислений m известны множества входных и выходных фрагментов данных (находящихся с m в отношениях in, out) - *inputs(m), outputs(m)*. На основе этих множеств строится множество $cinputs(m) = \{\text{ФВ } d \mid outputs(d) \subseteq inputs(m)\}$. Жизненный цикл фрагментов в процессе вычислений удобно описать с помощью конечных автоматов (КА). Состояние конечного автомата будет соответствовать состоянию фрагмента данных или фрагмента вычислений в исполнительной системе, а поступающие на вход символы - это события, происходящие в процессе исполнения, с отмеченными началом и концом.

Жизненный цикл некоторого ФД x будет описываться конечным автоматом $M_x = (V, Q, q0, F, \delta)$. Входной алфавит V (множество событий исполнительной системы) состоит из символа O_a^{finish} и символов $\{O_r^{Vsend} \mid r \in \text{подмножеству узлов}\}$. Появление символа O_a^{finish} на входной ленте автомата соответствует появлению события завершения исполнения ФВ a. Появление любого из символов O_r^{Vsend} на входной ленте автомата соответствует появлению события отправки ФД x на узел r. Множество состояний Q (состояния ФД) состоит из *{Не вычислен, Вычислен},* первое из которых является начальным состоянием q_0, а второе является единственным представителем множества завершающих состояний F. Правила переходов устроены таким образом, что при завершении ФВ a автомат переходит в

состояние “Вычислен”. В момент этого перехода фрагмент данных получает значение, инициализируется фрагментом вычислений *a*. ФД в состоянии “Вычислен” может быть отправлен на другие узлы. Диаграмма состояний конечного автомата M_x представлена на рисунке 2.

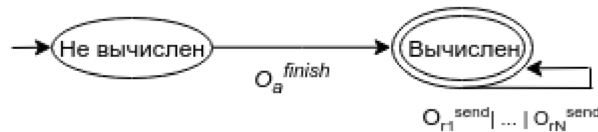

Рисунок 2 — Конечный автомат, описывающий жизненный цикл фрагмента данных

Жизненный цикл некоторого ФВ *a* будет описываться конечным автоматом $M_a = (V, Q, q_0, F, \delta)$. Входной алфавит V (множество событий исполнительной системы) состоит из символов $\{O^{V\,ready},\ O_a^{start},\ O_a^{finish}\} \cup \{O_d^{V\,recv} |\ d \in$ подмножеству $inputs(a)\}$. Множество состояний Q (состояния ФВ) состоит из *{Не готов к исполнению, Ожидание входных ФД, Готов к исполнению, Исполнение, Исполнен}*. Начальным состоянием q_0 является “Не готов к исполнению”. Множество завершающих состояний F состоит из единственного состояния “Исполнен”. Множество переходов по символу O^{Vready} соответствуют любому из переходов *{* O_b^{finish} *| b – ФВ, т. ч. существует ФД, являющийся для ФВ b выходным, а для ФВ a входным}*. В случае, если множество входных ФД для данного ФВ пусто, или последний из входных ФД был вычислен, то осуществляется переход в состояние “Ожидание входных ФД”. Множество переходов по любому из символов O_d^{Vrecv} соответствуют событию приема входного ФД *d* для ФВ *a*. В случае, если множество входных ФД пусто, или последний из входных ФД был передан на узел, то осуществляется переход в состояние “Готов к исполнению”. Появление символа O_a^{start} на входной ленте автомата соответствует появлению события извлечения ФВ *a* из очереди готовых ФВ и началу его исполнения. Появление символа O_a^{finish} на входной ленте автомата соответствует появлению события завершения исполнения ФВ *a*. Переходам O_a^{start} и O_a^{finish} однозначно ставится в соответствие номер рабочего потока t_k^m, в котором произошли соответствующие события системы исполнения. Также определяются отображения $wt(m) = t_k^m,\ rk(m) = k$. Диаграмма состояний конечного автомата M_a представлена на рисунке 3.

Рисунок 3 — КА, описывающий жизненный цикл фрагмента вычислений

Исполнение всей фрагментированной программы описывается работой конечного автомата, полученного путем объединения конечных автоматов всех ФД и ФВ данного графа исполнения. Считаем событиями начала и завершения исполнения фрагментированной программы P^{start} и P^{finish}, соответственно.

Каждой смене состояния в построенном конечном автомате фрагментированной программы ставятся в соответствие моменты времени, соответствующие начале смены состояния (началу события в исполнительной системе) и окончанию смены состояния (окончанию события в исполнительной системе). Указанные моменты времени измеряются по двум разным часам, когда реальная исполнительная система проходит через соответствующий этап исполнения. Первые часы – часы реального (системного) времени, вторые – часы потока, измеряющие время, затраченное процессором на выполнение определенного потока системы исполнения.

Рабочие потоки узла i находятся в цикле исполнения ФВ и ожидания события P^{finish} либо появления на узле r_i фрагментов вычислений, готовых к исполнению. Интервал времени, в течение которого рабочий поток t_i^j находится в ожидании, назовем простоем. Начало этого интервала соответствует событию P^{start} либо событию, соответствующему переходу O_k^{finish} в данном потоке, конец – событию P^{finish} либо выборке готового к исполнению ФВ m, т.е. переходу O_m^{start}, в данном потоке.

3.2 Алгоритм получения характеристик исполнения

В алгоритме получения характеристик исполнения SLO для фрагментированной программы сначала производится получение характеристик для каждого интервала простоя, затем полученные результаты суммируются для каждой из характеристик. Опишем данный алгоритм:

1. Введем переменные - ассоциативные массивы:
    a. s_1 : ФВ → интервал времени,
    b. $ovhSI$: ФВ → множество интервалов времени,
    c. $oUniCf$, $oUniStarv$, TO, TL, TS : ФВ → длительность.
2. Определим множество интервалов простоя $\{(g^1, g^2) \mid g^1$ соответствует окончанию перехода КА для d в состояние “Исполнение”, g^2 соответствует началу перехода КА для d в состояние “Исполнен”, а ФВ $d \in$ множеству ФВ анализируемой программы$\}$.
3. Если не осталось не проанализированных интервалов простоя, то осуществляется переход на шаг 16.

4. Выберем еще не проанализированный интервал простоя $g=(g^1,g^2)$ с минимальным g^2 среди всех не проанализированных интервалов, т.е. завершившегося раньше других из-за перехода КА некоторого фрагмента вычислений *m* в состояние “Исполнен”.
5. Для *g* определим подинтервал s_1, соответствующий простою, формирующему характеристики *TS* и *TO* внутри *g*. Это достигается за счет выполнения следующих шагов:
   a. Примем s_1^2 = *max({* g_1 *}* ∪ *{ момент начала перехода КА для n в состояние “Исполнен” | n* ∈ *cinputs(m) }).* Это время будет грубо обозначать момент окончания задержки типа “Starvation” внутри интервала простоя.
   b. $s_1 = (g^1,s_1^2)$. В этот интервал входит время счета входных ФД для ФВ *m* и, возможно, ФД, являющихся входными для других фрагментов вычислений, а также часть времени на накладные расходы.
6. Определим $s_1[m] = s_1$
7. Если s_1 не пуст:
   a. Определим ФВ *n* такой, что *n* ∈ *cinputs(m) и* ∀*c* ∈ *cinputs(m) момент начала перехода КА для c в состояние “Исполнен”* ≤ *моменту начала перехода КА для n в состояние “Исполнен”.* (момент начала перехода КА для *n* в состояние “Исполнен” будет совпадать с s_1^2)
   b. Примем *D = { ФВ d | wt(d)=wt(n) и момент окончания перехода КА для d в состояние “Исполнение”* ∈ s_1 *}*, т.е. множество фрагментов, исполнявшихся в том же рабочем потоке, что и *n*, в течение интервала s_1.
   c. Определим накладные расходы, повлиявшие на простой *g*, для каждого ФВ *d* из *D* и интервала простоя, предшествующего исполнению d:
      i. Примем p_2 = интервалу времени, в течение которого КА для d находился в состоянии “Исполнение”.
      ii. Примем $p_1 = s_1 \cap p_2$.
      iii. Примем $q = s_1 \cap s_1[d]$.
      iv. Примем $oIsect[m,d] = \bigcup_{ob \in OvhSI[d]} (s_1 \cap ob)$.
      v. Примем $oUniCf[m,d] = oUniCf[d] * |p_1| / |p_2|$.
      vi. Примем $oUniStarv[m,d] = oUniStarv[d] * |q| / |s_1[d]|$.
      vii. *Комментарий к приведенным выше шагам*: Задача отслеживания границ простоев, вызванных накладными расходами, возникающих из-за

вытеснения рабочих потоков, занятых исполнением ФВ, -- сложная задача, так как для ее решения нужно анализировать ход исполнения ФВ, неделимый с точки зрения исполнительной системы. Но возможно определение суммарной длительности указанных простоев. Поэтому введено упрощение: считается, что эта длительность равномерно распределена по всему интервалу исполнения ФВ, и переменные oUniCf, oUniStarv принимают значения с учетом этого упрощения.

d. Определим $oUniStarv[m] = \sum_{d \in D} (oUniCf[m,d] + oUniStarv[m,d])$. Эта величина является оценочной суммарной длительностью простоев в течение s_1 в рабочем потоке *wt(m),* вызванных накладными расходами, связанными с вытеснением рабочих потоков во время исполнения ФВ.

e. Примем $oIsect = \bigcup_{d \in D} oIsect[m,d]$. Это есть объединение интервалов простоев внутри s_1, вызванных накладными расходами, границы которых удалось измерить.

8. Если s_1 пуст, то примем $oUniStarv[m] = 0;\ oIsect = \{\}$.
9. Для *g* определим подинтервал *l,* соответствующий простою, формирующему характеристику *TL* внутри *g*. Это достигается за счет выполнения следующих шагов:
   a. Примем $l^2 = max(\{$ *конец* $s_1 \} \cup \{$ *момент начала перехода* O_d^{Vrecv} *в КА для m* | $d \in inputs(m) \})$.
   b. Если l^2 = концу s_1, то примем $l^1 = l^2$ = концу s_1.
   c. Иначе известен ФД *d* такой, что момент начала перехода O_d^{Vrecv} в КА для $m = l^2$. Примем l^1 = моменту окончания перехода O_r^{Vsend}, где $r = rk(m)$.
   d. Определим $l = (l^1,\ l^2)$. Данный интервал соответствует задержке, вызванной передачей по сети входных ФД для *m* на узел *rk*(*m*) сверх времени счета его входных ФД.
10. Определим $OvhSI[m] = (g \setminus s_1 \setminus l) \cup oIsect$ -- множество интервалов простоев, обусловленных работой исполнительной системы и посторонних процессов операционной системы, внутри *g*.
11. Определим *OUniCf*[*m*] как разницу между длительностями исполнения *m* по системным часам и по часам потока *wt(m).* Эта разница будет соответствовать длительности времени, в течение которого рабочий поток *wt(m)* был вытеснен другими потоками во время исполнения ФВ.

12. Определим $TO[m] = \sum_{ob \in OvhSI[m]} |ob| + oUniStarv[m]$.
13. Определим $TL[m] = |l|$.
14. Определим $TS[m] = |s_1| - oUniStarv[m] - \sum_{oi \in oIsect} |oi|$.
15. Перейдем к шагу 3.
16. Просуммируем характеристики *TO[i]*, *TL[i]*, *TS[i]* для каждого интервала простоя *i*, и получим, соответственно, характеристики *TO*, *TL*, *TS* исполнения фрагментированной программы.

Представленный алгоритм позволяет определить характеристики исполнения фрагментированной программы TS, TL и TO по заданному графу исполнения и списку событий, соответствующих описанной в виде конечного автомата модели исполнения. При построении алгоритма пришлось использовать ряд упрощений, вызванных трудностью определения моментов взаимного вытеснения потоков, в результате чего определяемые значения характеристик могут незначительно отличаться от реальных.

4 Реализация системы анализа исполнения фрагментированных программ

4.1 Описание реализации

Приведенный в предыдущем разделе алгоритм получения характеристик исполнения SLO для фрагментированной программы был реализован для системы фрагментированного программирования LuNA в виде системы анализа исполнения. Система состоит из двух частей:

1. Модуля исполнительной системы LuNA для сбора информации о ходе исполнения фрагментированной программы.
2. Инструмента для обработки полученной информации.

Необходимость хранить данные для анализа не в оперативной памяти, а на диске следует из того, что объем памяти, необходимый для анализа одного интервала, может быть неприемлемо большим. Это следует из того, что для анализа одного интервала простоя нужно копить информацию о всех фрагментах вычислений, исполнявшихся в этом интервале, до тех пор, пока не станет известен ФВ, начало исполнения которого послужило окончанием простоя.

При том, что время простоя ресурса теоретически может быть сравнимо с временем выполнения всей программы, а количество ФВ может быть сколь угодно большим, объем оперативной памяти, выделенной под хранение нужной для анализа информации, может быть критическим фактором, мешающим нормальной работе фрагментированной программы в системе исполнения LuNA.

Объем памяти, необходимой для анализа одного интервала простоя, обычно значительно превосходит объем памяти, необходимый для хранения результатов анализа данного простоя. Таким образом, если на диск записывать исходную информацию для *всех* интервалов простоя, то это повлечет за собой более частый вызов системных вызовов для вывода данных, что плохо скажется на производительности разрабатываемого инструмента.

Для некоторых же интервалов простоя все данные, необходимые для их анализа, можно уместить в оперативной памяти в буфере, приемлемый размер которого может быть задан при запуске фрагментированной программы, и провести анализ непосредственно во время ее исполнения. В случае, если размер буфера оказался мал, можно сбросить его на диск и вычислить вклад каждого фактора SLOW позже.

Было принято решение использовать для хранения данных профилирования буфер в оперативной памяти с выгрузкой на диск в случае переполнения буфера.

Модуль для сбора моментов времени, соответствующих описанным в модели исполнения событиям, написан на языке C++ и встроен в систему исполнения LuNA. В соответствии с выбранным подходом, он состоит из следующих основных компонент:

1. *IDelayLogger* – интерфейс, содержащий методы, вызываемые из других модулей исполнительной системы в моменты, соответствующие возникновению событий, необходимых для проведения анализа.
2. *IdentDelayLogger* – примитивная реализация интерфейса *IDelayLogger*, записывающая в файлы всю информацию о событиях и не проводящая никакого анализа. Данный класс использовался на ранних стадиях разработки, так как всю записанную на диск информацию легко можно перенести в реляционную базу данных, что позволяет описанный алгоритм анализа реализовать посредством нескольких запросов к базе данных. Но такой подход годен только для создания прототипа инструмента, т.к. обладает низкой производительностью.
3. *DelayLoggerAnalyzer* – абстрактный класс, реализующий интерфейс *IDelayLogger*, преобразующий информацию, поступающую через методы *IDelayLogger,* в удобные для анализа структуры, и передающий их в свои абстрактные методы.
4. *Analyzer* – класс, используемый для анализа интервалов простоя на основе структур, создаваемых *DelayLoggerAnalyzer.*
5. *RamLoggerAnalyzer* – класс, реализующий *DelayLoggerAnalyzer*. Его предназначение – по возможности анализировать данные о задержках в ходе исполнения. В конструктор ему передается приемлемый размер буфера в оперативной памяти для хранения информации об исполнении, указатель на объект класса Analyzer, а также указатель на объект класса, реализующий *DelayLoggerAnalyzer,* которому в случае надобности будут переданы не поместившиеся в буфер данные.
6. *DiskLoggerAnalyzer* – класс, реализующий *DelayLoggerAnalyzer.* Его предназначение – сохранять данные для последующего анализа на диск. В конструктор ему передается путь в файловой системе, по которому будут храниться данные.

Инструмент для обработки информации, получаемой с помощью реализованного модуля для системы LuNA, использует те же классы, но уже получает информацию об исполнении фрагментированной программы с диска.

4.2 Тестирование системы анализа исполнения

Была проведена серия тестовых испытаний созданного инструмента. Первые три теста являются синтетическими и направлены на проверку правильности определения характеристик TS, TL и TO. Для этого были созданы тестовые программы на языке LuNA, в результате анализа исполнения которых ожидалось изменение одной из характеристик SLO. В последнем тесте с помощью разработанного инструмента выполняется анализ реализованной в системе LuNA задачи моделирования динамики самогравитирующего пылевого облака методом частиц-в-ячейках [14]. По результатам анализа сделаны соответствующие выводы о причинах полученной производительности.

4.2.1 Тест для Starvation

Цель теста - продемонстрировать влияние степени параллелизма фрагментированного алгоритма на характеристику TS. Запуски теста проводились на одном многоядерном узле с 8 рабочими потоками. В этом тесте фрагментированная программа представляла из себя N (для разных запусков это количество менялось от 1 до 16) независимых цепочек ФВ-ФД-ФВ-ФД… постоянной длины (100 ФВ в одной цепочке). ФВ внутри каждой цепочки исполняются последовательно из-за информационных зависимостей. Время выполнения одного ФВ было постоянным, обратно пропорциональным N. От данного теста ожидалось уменьшение TS с ростом N от 1 до 8 до нуля (связанное с увеличением степени параллелизма программы и занятием простаивающих потоков), при N от 8 до 16 ожидалось пренебрежимо малое значение TS (накапливаемое в момент завершения работы программы).

Результаты теста, представленные ниже на рисунке 4, соответствуют ожиданиям — TS уменьшается при увеличении количества независимых цепочек до 8, затем находится на постоянном пренебрежимо малом уровне. WallT — суммарное время работы программы по системным часам, умноженное на общее количество рабочих потоков. ThreadT — суммарное процессорное время исполнения для всех ФВ (полезная работа).

4.2.2 Тест для Overhead

Цель данного теста — продемонстрировать изменение доли времени, которое тратится на работу исполнительной системы, при изменении степени фрагментации алгоритма. Запуски теста проводились на одном узле с 1 рабочим потоком.

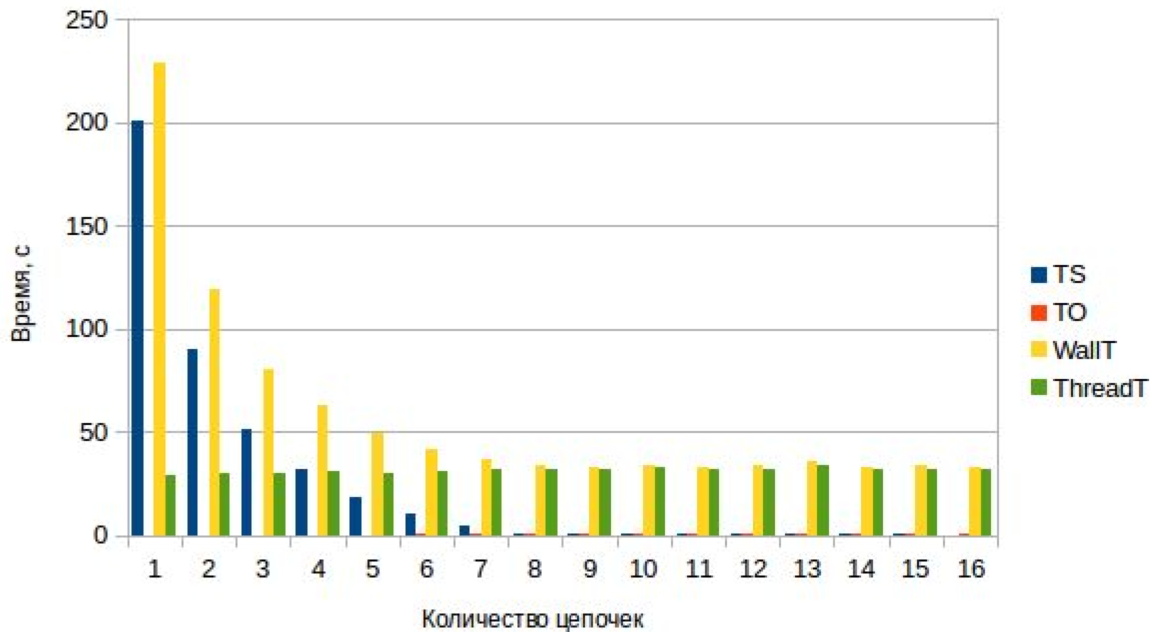

Рисунок 4 — результаты теста для Starvation

В этом тесте фрагментированная программа представляет из себя одну цепочку ФВ-ФД-ФВ-ФД... Время выполнения одного ФВ для разных запусков менялось пропорционально 1,2,4,...,128. Длина цепочки менялась обратно пропорционально времени выполнения одного ФВ. Таким образом, объем полезной работы должен был оставаться неизменным. От данного теста ожидалось уменьшение Overhead с ростом времени работы одного ФВ. Результаты теста, представленные на рисунке 5, соответствуют ожиданиям — значение TO уменьшается с увеличением времени работы одного ФВ.

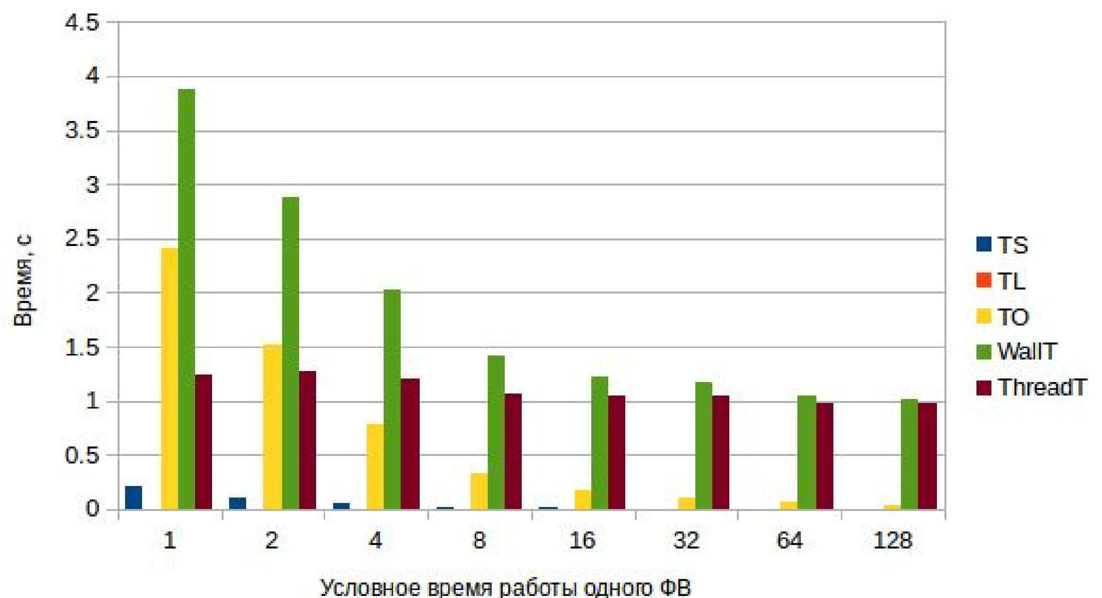

Рисунок 5 — результаты теста для Overhead

4.2.3 Тест для Latency

Цель данного теста — вызвать изменение характеристики TL путем изменения объема передаваемых между узлами данных. Для этого тесты запускаются на двух узлах, на каждом из которых запускается по одному рабочему потоку. LuNA программа состоит из двух цепочек *A* и *B* вида ФВ-ФД-ФВ-ФД… равной длины. Каждый четный ФВ из *A* выполняется на первом узле, каждый нечетный ФВ из *A* выполняется на втором узле. И наоборот, каждый четный ФВ из *B* выполняется на втором узле, каждый нечетный ФВ из *B* выполняется на первом узле. Таким образом, каждый вычисленный на одном узле ФД должен быть передан на другой узел. Характерная длительность выполнения для каждого ФВ была выбрана одинаковой для того, чтобы уменьшить значение характеристики *TS* в предположении, что интервалы времени, в течение которых работают *i*-ые ФВ из разных цепочек, примерно совпадают. Условный размер всех ФД в пределах одного запуска постоянен, для разных запусков этот размер меняется. Предполагается, что TL будет увеличиваться при увеличении размера ФД. Результаты теста, представленные на рисунке 6, соответствуют ожиданиям — TL растет при увеличении условного размера ФД.

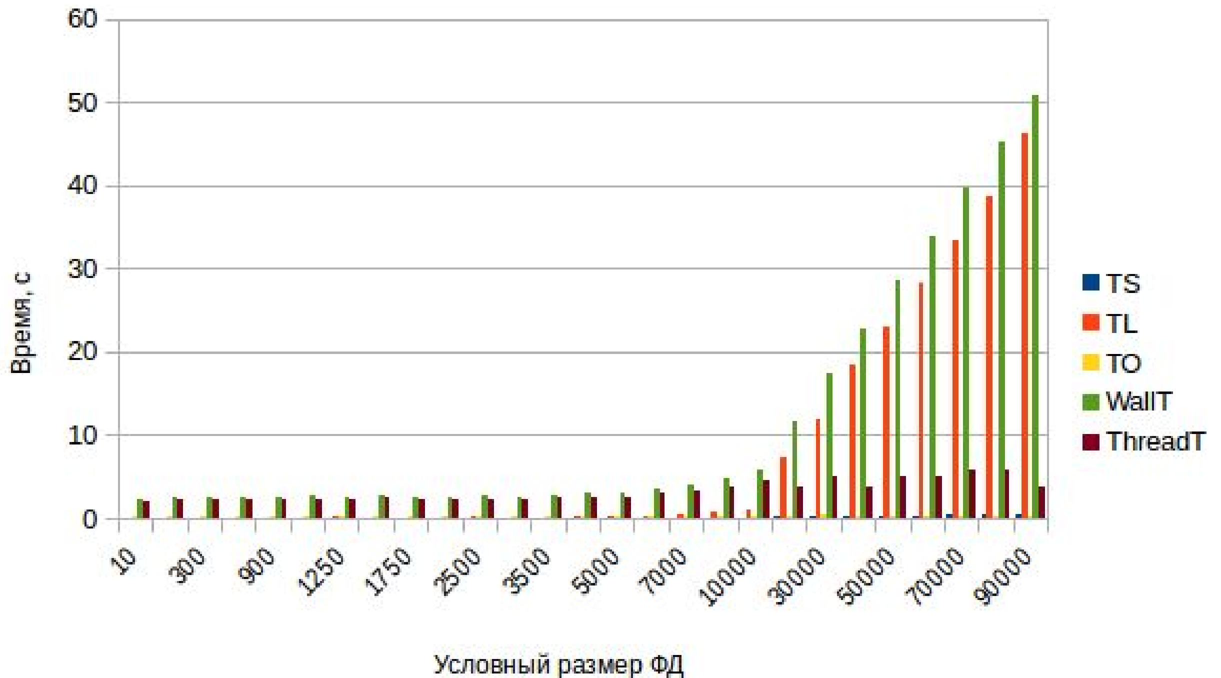

Рисунок 6 — результаты теста для Latency

4.2.4 Анализ LuNA-реализации метода частиц-в-ячейках

Цель теста — проверить работоспособность разработанной системы анализа исполнения фрагментированных программ на “реальной” задаче. В качестве “реальной” задачи использовалась реализованная в системе LuNA задача трехмерного моделирования самогравитирующего пылевого облака методом частиц-в-ячейках [14]. Основными параметрами задачи являются:

- размеры пространственной сетки: NX × NY × NZ,
- количество модельных частиц: NP,
- число шагов по времени: NT.

Параметрами реализации являются:

- степень фрагментации пространства моделирования - количество подобластей, на которые оно разбивается: FX × FY × FZ,
- ресурсы, использованные для расчета - количество узлов кластера, ядер и рабочих потоков в одном узле.

Для исполнения программы, реализующей задачу с параметрами NX = NY = NZ = 100, NP = 1 000 000, NT = 10, FX = FY = FZ = 4, на 1 узле с 16 рабочими потоками, полученные значения характеристик были следующими:

- TS = 30,1 с
- TL = 0 с
- TO = 110,6 с
- ThreadT = 13,1 с
- WallT = 159,8 с

Для исполнения программы, реализующей задачу с параметрами NX = NY = NZ = 100, NP = 1 000 000, NT = 10, FX = FY = FZ = 4, на 4 узлах с 16 рабочими потоками на каждом, полученные значения характеристик были следующими:

- TS = 7669 с
- TL = 1556 с
- TO = 1234 с
- ThreadT =12 с
- WallT = 10499 с

Во время второго запуска происходил обмен фрагментами данных между узлами, из-за чего характеристика *TL* приняла ненулевое значение.

Доля *TO* от общего времени работы в первом запуске составляет около 69 процентов, что свидетельствует о высоких накладных расходах на работу исполнительной системы. При использовании 4 узлов во втором запуске значение TO еще увеличилось.

Полезное время работы ThreadT для двух запусков незначительно отличается. Это возможно из-за того, что в него помимо непосредственно полезных вычислений входит время на динамическое выделение памяти под фрагменты данных. Также это возможно из-за того, что общий объем активно используемой памяти на одном узле снизился, что привело к более эффективному использованию кэш-памяти.

При сравнении значений WallT для двух разных запусков задачи видно, что несмотря на увеличение общего количества доступных ядер в 4 раза, время счета задачи увеличивается почти в 16 раз. Это известная [5] проблема исполнительной системы LuNA, возникающая при запуске задач более, чем на одном вычислительном узле.

Во втором запуске доля TS от общего времени работы составляет около 73 процентов, что свидетельствует о низкой степени параллелизма в задаче.

По результатам анализа можно сделать следующие выводы:

1. Высокая доля TO в первом запуске свидетельствует о высоких накладных расходах на работу исполнительной системы LuNA. Для ее снижения при использовании данных ресурсов, возможно, потребуется модификация самой исполнительной системы или уменьшение степени фрагментации алгоритма.
2. Высокая доля TS во втором запуске свидетельствует о низкой степени параллелизме, что не дает эффективно использовать доступные вычислительные ресурсы. Возможным решением, позволяющим снизить значение этой характеристики, является увеличение степени фрагментации алгоритма.

ЗАКЛЮЧЕНИЕ

В ходе работы был проведен обзор и анализ существующих средств профилирования параллельных программ. На основе предложенного набора факторов SLOW был сформулирован набор характеристик, позволяющих определить узкие места в исполнении фрагментированных программ в терминах технологии фрагментированного программирования. Построена модель исполнения фрагментированного алгоритма в виде конечного автомата, достаточная для определения значений выбранных характеристик, не зависящая от конкретной реализации исполнительной системы. Построен алгоритм получения выбранных характеристик по заданному исполнению в соответствии с построенной моделью. Построенный алгоритм был реализован для системы LuNA в виде подсистемы анализа исполнения LuNA-программам. Работоспособность выбранного подхода к оценке производительности фрагментированных программ, а также его реализации, была проверена с помощью тестирования.

Таким образом, цель работы была достигнута. На защиту выносится:

- набор характеристик исполнения фрагментированных программ и их интерпретация,
- модель исполнения фрагментированного алгоритма в виде конечного автомата,
- алгоритм получения выбранных характеристик по заданному исполнению,
- реализация алгоритма для системы LuNA.

Результаты работы были приняты на международную конференцию “Вычислительная и прикладная математика 2017” в качестве устного доклада и на XXIX Летнюю международную молодежную Школу-конференцию по параллельному программированию в качестве устного доклада.

В дальнейшем возможна работа по следующим направлениям:

1. Получение характеристик исполнения для отдельных частей LuNA-программы.
2. Автоматизация определения возможных проблем с производительностью на основе анализа получаемых характеристик.

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

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием

для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

___________________________	______________________________
ФИО студента	Подпись студента

« ____ »___________20 __г.
(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. В.А.Вальковский, В.Э.Малышкин. Синтез параллельных программ и систем на вычислительных моделях. – Наука, Новосибирск, 1988, 128 стр.
2. Kraeva, M.A., Malyshkin, V.E.: Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. In the Int. Journal on Future Generation Computer Systems, Elsevier Science, NH. Vol. 17, No. 6, 2001. P. 755-765.
3. V.E. Malyshkin, V.A. Perepelkin. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem. // Proceedings of the 11th International Conference on Parallel Computing Technologies (PaCT-2011), LNCS, Vol. 6873. 2011. P. 53-61.
4. Malyshkin V., Perepelkin V. Optimization methods of parallel execution of numerical programs in the LuNA fragmented programming system // The Journal of Supercomputing. Vol. 61, № 1. 2012. P. 235-248.
5. Darkhan Akhmed-Zaki, Danil Lebedev, and Vladislav A. Perepelkin. Implementation of a Three-Phase Fluid Flow (“Oil-Water-Gas”) Numerical Model in the LuNA Fragmented Programming System // Proceedings of the 13th International Conference on Parallel Computing Technologies (PaCT-2015), LNCS, Vol. 9251. 2015. P. 80-85.
6. Malyshkin, V.E., Perepelkin, V.A. The PIC implementation in LuNA system of fragmented programming // Journal of Supercomputing. 2014. Vol. 69, Iss. 1. P. 89-97.
7. MPE - MPI Parallel Environment. http://www.mcs.anl.gov/research/projects/perfvis/ [Электронный ресурс]
8. Extrae instrumentation package. https://tools.bsc.es/extrae [Электронный ресурс]
9. Paraver performance analysis tool. https://tools.bsc.es/paraver [Электронный ресурс]
10. Bernd Mohr, Felix Wolf, KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs // Euro-Par 2003, LNCS 2790, Springer-Verlag Berlin Heidelberg, 2003, P. 1301-1304.
11. Scalasca performance analysis tool. http://www.scalasca.org [Электронный ресурс]
12. H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, HPX: A Task Based Programming Model in a Global Address Space // Proc. of the 8th International Conference on PGAS Programming Models, 2014.
13. Intel Trace Analyzer and Collector. https://software.intel.com/en-us/intel-trace-analyzer [Электронный ресурс]

14. С.Е. Киреев Использование системы фрагментированного программирования LuNA для параллельной реализация PIC-метода // Параллельные вычислительные технологии - XI международная конференция, ПаВТ'2017, г. Казань, 3-7 апреля 2017 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2017, с. 519.