МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

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

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Валитова Андрея Александровича

Тема работы:

РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМА ДИНАМИЧЕСКОЙ АДАПТАЦИИ КОЛИЧЕСТВА РАБОЧИХ ПОТОКОВ В СИСТЕМЕ LUNA

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

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

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

Новосибирск, 2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий

Кафедра Параллельных вычислений
(название кафедры)

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

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В. Э.
(фамилия, И., О.)
…….
(подпись)
«16» января 2023 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студент*у(ке)* Валитову Андрею Александровичу, группы 19205
(фамилия, имя, отчество, номер группы)

Тема Разработка и реализация алгоритма динамической адаптации количества рабочих потоков в системе LuNA
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 07 ноября 2022 г. № 0330, скорректирована распоряжением проректора по учебной работе от 16 января 2023 г. №0006

Срок сдачи студентом готовой работы 20 мая 2023 г.

Исходные данные (или цель работы):

разработать и реализовать в виде программного модуля системы фрагментированного программирования LuNA алгоритм динамической адаптации количества рабочих потоков

Структурные части работы:

содержание, введение, обзор литературы, постановка задачи, разработка и реализация алгоритма, тестирование

Руководитель ВКР
доцент каф. ПВ ФИТ НГУ,
к.т.н.
Власенко А. Ю. /……..
(ФИО) / (подпись)
«16» января 2023г.

Задание принял к исполнению
Валитов А.А. /……..
(ФИО студента) / (подпись)
«16» января 2023г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ,
Перепёлкин В. А. /……..
(ФИО) / (подпись)
«16» января 2023г.

СОДЕРЖАНИЕ

Введение 5
1 Анализ предметной области 9
1.1 Обзор систем автоматического конструирования параллельных программ и средств параллельного программирования 9
1.2 Результаты обзора 13
2 Алгоритм динамической адаптации количества рабочих потоков .. 14
2.1 Необходимые термины и определения 14
2.2 Постановка задачи 15
2.2.1 Неформальная постановка задачи 15
2.2.2 Математическая постановка задачи 16
2.2.3 Требования, предъявляемые к алгоритму 18
2.3 Описание предлагаемого решения 19
2.3.1 Идея предлагаемого решения 19
2.3.2 Сбор информации о выполнении фрагментов кода 21
2.3.3 Определение границ скользящего окна и границ анализа собранных данных 22
2.3.4 Определение общей загрузки процессора 25
2.3.5 Определение полезной загрузки процессора 26
2.3.6 Правило изменения количества потоков исполнения 27
2.4 Характеристика предложенного алгоритма 28
3 Программная реализация алгоритма 30
3.1 Особенности реализации системы фрагментированного программирования LuNA 30
3.2. Программная реализация алгоритма динамической адаптации количества потоков 31
3.2.1 Встраивание алгоритма в исполнительную систему LuNA 31
3.2.2 Передача временных меток на вход алгоритму 31
3.2.3 Выбор потока исполнения, выполняющего анализ временных меток 32

3.2.4 Особенности хранения временных меток 32
3.2.5 Особенности определения скользящего окна и интервала анализа временных меток 34
3.2.6 Особенности определения общей загрузки процессора 35
3.2.7 Особенности определения полезной загрузки процессора 36
3.2.8 Изменение количества рабочих потоков 37
3.2.9 Дополнительная функциональность по логированию работы алгоритма 38
3.3 Характеристика реализованного модуля 39
4 Тестирование 41
4.1 Тестирование алгоритма на задаче блочного умножения матриц 41
4.2 Тестирование алгоритма на итерационной задаче решения уравнения Пуассона 46
4.3 Тестирование алгоритма на решателе модельного трехмерного уравнения теплопроводности в единичном кубе 48
4.4 Итоги тестирования 51
Заключение 52
Список используемых источников и литературы 54
Приложение А 54
Приложение Б 68

ВВЕДЕНИЕ

В настоящее время создаются новые и продолжают развиваться уже существующие системы автоматического конструирования и исполнения программ. Такие системы позволяют создавать программы, описывая на некотором высокоуровневом языке логику ее работы, не вдаваясь в подробности низкоуровневого программирования, сетевого взаимодействия и др., а также исполнять эти программы. Использование систем автоматического конструирования и исполнения программ, в отличие от подхода традиционного программирования, так или иначе решает сложности, связанные с созданием эффективных и корректных программ. Это составляет одно из их главных преимуществ, ведь тем самым на программиста накладываются менее высокие требования к умению создавать такие программы.

Одним из важных классов таких систем является класс систем автоматического конструирования и исполнения параллельных программ. Данный класс систем сложен, поскольку при их создании необходимо обеспечить корректное взаимодействие не только между потоками исполнения, но и между отдельными процессами и даже отдельными узлами мультикомпьютера.

Поскольку исполнение параллельной программы происходит в распределенной многопоточной среде, потоки исполнения играют в нем важную роль: от них зависит такая ключевая характеристика, как скорость работы программы.

В программах, созданных системами автоматического конструирования и исполнения параллельных программ, вообще говоря, можно использовать разное количество потоков. Их количество может быть и статическим (количество потоков задается на этапе подготовки к исполнению программы), и динамическим (количество потоков изменяется во время исполнения программы на основе какого-либо правила или алгоритма). Одновременно с этим, установить оптимальное количество потоков (то есть такое, при котором

программа работает быстрее всего) в общей постановке задачи сложно в силу большого количества факторов, прямо или косвенно влияющих на время выполнения. Поэтому важно исследовать разные алгоритмы и методы, которые в той или иной степени решают данную проблему.

Одним из представителей класса систем автоматического конструирования и исполнения параллельных программ является система фрагментированного программирования LuNA [1], разрабатываемая в ИВМиМГ СО РАН. Исполнительная система LuNA использует task-based модель. Задачами в LuNA являются информационно связанные единицы программы, выполнение которых позволяет системе вычислить конечные целевые переменные. Выполнение задач осуществляется потоками исполнения, называемых рабочими потоками.

Разные LuNA программы в ходе выполнения формируют разное количество задач для исполнения. Помимо этого, задачи могут иметь разный характер по объему машинных инструкций. Таким образом, количество задач и их свойства меняются динамически, и поэтому в разный момент времени может быть лучше иметь разное количество потоков, причем их количество должно быть как можно ближе к оптимальному. При этом изменение количества потоков должно происходить в автоматическом режиме, поскольку иначе теряется бóльшая часть выигрыша этих изменений.

С точки зрения оптимизации исполнения LuNA программ исполнение с оптимальным или близким к нему количеством рабочих потоков может привести к уменьшению общего времени работы программы. Зачастую именно данная характеристика является определяющей для конечного пользователя.

Для управления количеством потоков автором не было найдено ни одной работы, в которой бы использовался в автоматическом режиме динамический подход к выбору количества потоков в процессе исполнения. Таким образом, проблема отсутствия такого подхода при его необходимости остается не решенной, что подчеркивает его актуальность.

В связи с вышеперечисленным, возникает необходимость в новом алгоритме динамической адаптации количества рабочих потоков в исполнительной системе, который позволит в ходе выполнения программы в автоматическом режиме анализировать нефункциональные характеристики рабочих потоков и принимать решение об изменении их количества.

Цель работы – разработка и реализация алгоритма динамической адаптации количества рабочих потоков в системе LuNA.

Для достижения данной цели были поставлены следующие **задачи**:

1. Проанализировать особенности проблемы динамической адаптации количества рабочих потоков в системе LuNA;
2. Сформировать требования к разрабатываемому алгоритму;
3. Разработать алгоритм динамической адаптации количества рабочих потоков, удовлетворяющий требованиям из п. 2;
4. Реализовать разработанный алгоритм и встроить его в систему LuNA;
5. Провести экспериментальное исследование времени выполнения практически значимых вычислительных задач при активированном алгоритме динамической адаптации количества рабочих потоков.

Научная новизна работы состоит в том, что был разработан и реализован алгоритм, который позволил динамически адаптировать количество рабочих потоков.

Практическая ценность работы заключается в том, что новый алгоритм динамической адаптации количества рабочих потоков системы LuNA позволил сократить время исполнения различных прикладных и модельных задач. С помощью данного алгоритма было достигнуто ускорение работы некоторых апробированных LuNA программ по сравнению с их выполнением со статическим количеством потоков.

Настоящая работа состоит из введения, четырех глав и заключения. Во введении описывается общая проблема, которую призван решить алгоритм,

актуальность проблемы, цель работы, практическая значимость и научная новизна. В первой главе рассматриваются родственные системы параллельного программирования, организация рабочих потоков в них, управление их количеством, фиксируется результат обзора и формулируются требования к разрабатываемому алгоритму. Во второй главе описывается модель нового алгоритма динамической адаптации количеств потоков. В третьей главе описываются детали реализации. В четвертой главе приведены результаты экспериментального исследования производительности алгоритма на выбранном наборе вычислительных задач.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

Поскольку в разных системах как автоматического конструирования параллельных программ, так и в целом параллельного программирования используются различные совокупные подходы к управлению потоками и повышению производительности за счет них, необходимо проанализировать существующие решения в этой области. Такой анализ позволит учесть существующий опыт разработки алгоритмов управления потоками, установить, как с помощью потоков системы повышают производительность программ, выявить незакрытые в области проблемы и наконец сформировать требования к разрабатываемому алгоритму.

Нижеизложенный обзор систем автоматического конструирования параллельных программ и средств параллельного программирования был произведен по следующим критериям:

1. Как производится разбиение задач по потокам;
2. Какие алгоритмы используются для выбора количества потоков как на старте программы, так и в ходе ее исполнения;
3. Как потоки отображаются на ядра узла и предоставляется ли механизм управления отображением;
4. Реализован ли алгоритм динамического выбора количества потоков в системе.

1.1 Обзор систем автоматического конструирования параллельных программ и средств параллельного программирования

Система автоматического конструирования параллельных программ Charm++ [2], разрабатываемая в Иллинойском университете в Урбане-Шампейне, США, представляет программу как набор взаимодействующих объектов, называемых в системе «чарами». Программа в системе запускается на нескольких отдельных процессах, может работать не только на одном узле, но и на кластере. Исполнение программы происходит в исполнительной системе

Charm++. Исполнительная система задает ход выполнения программы, управляет порядком исполнения чаров, выстраивает коммуникацию между процессами системы. Каждый процесс имеет свою очередь задач, в которую эти задачи посылает исполнительная система. Задачи на каждом процессе выполняются рабочими потоками. Количество рабочих потоков [3] на процессе является статическим параметром при запуске системы, равному количеству ядер узла, однако система предоставляет опцию запуска, при включении которой в ходе исполнения работает и балансировщик нагрузки. Балансировщик нагрузки [4] в ходе исполнения анализирует загрузку всех процессов в системе, и может “перекинуть” рабочий поток с более загруженного процесса на менее загруженный. Также, система предоставляет другую важную для настоящей работы опцию запуска как привязка рабочих потоков к ядрам узла. По умолчанию, система отключает данную опцию, однако, как отмечают разработчики системы Charm++ в официальной документации [5], включение опции может ускорить некоторые программы. Более того, ими же отмечено, что лучшее статическое количество рабочих потоков на процесс является неоднозначной характеристикой, зависящей от специфики выполняемой программы. Таким образом, система Charm++ контролирует производительность выполнения программы посредством балансировщика нагрузки и не предоставляет возможность изменения совокупного количества рабочих потоков в ходе выполнения программы.

Другая система автоматического конструирования параллельных программ PaRSEC [6] также оперирует задачами и рабочими потоками. Исполнение программы происходит в исполнительной системе PaRSEC. Задачи могут попадать в различные очереди согласно параметру запуска планировщика задач. По умолчанию используется локальная очередь для каждого потока, так что задачи для выполнения в очередь потока попадают от планировщика [7]. Система также предоставляет гибкий механизм привязки рабочих потоков к ядрам узла, причем, в отличие от Charm++, использует по умолчанию правило

“один привязанный рабочий поток на одно ядро”. Однако, в PaRSEC не существует отдельного модуля балансировки нагрузки, а более высокая производительность выполняемой программы зачастую достигается путем удачного выбора опций запуска программы [8]. Таким образом, система PaRSEC не использует внутренние алгоритмы для адаптации исполнительной системы к текущей загрузки узла и не поддерживает динамическое изменение количества рабочих потоков.

Еще одна система параллельного исполнения программ Legion [9], разрабатываемая в Стэнфордском университете, США, использует подход разделения всей программы на две составляющие: описание самой выполняемой программы и описание способа отображения программы на вычислительные ресурсы узла. Поскольку такой подход предполагает отсутствие автоматизации в конструировании способа отображения программы, алгоритмы управления количеством потоков в ходе параллельного исполнения также отсутствуют. Более высокая производительность достигается за счет правильного описания отображения программы на ресурсы узла, что зачастую требует в команде программиста по системному параллельному программированию.

Одно из самых популярных средств параллельного программирования OpenMP [10] позволяет программисту самостоятельно указать параллельные секции кода, при этом имея возможности гибкого настраивания исполнения этой параллельной секции. OpenMP использует специальный алгоритм [11] для выбора диапазона количества потоков параллельной секции, если не было указано точное число потоков. Но сам алгоритм оставляет право выбрать итоговое количество потоков компилятору, как и их способ отображения на ядра узла: например, компилятор GCC, согласно документации [12], определяет итоговое количество потоков равное количеству ядер. Балансировка нагрузки на потоки может быть активирована путем использования планировщика с типом dynamic, однако такой тип лишь распределяет более оптимальным способом задачи между потоками. Таким образом, система параллельного

программирования OpenMP не предоставляет механизмов динамической адаптации количества потоков во время исполнения.

Система фрагментированного программирования LuNA [1], разрабатываемая в ИВМиМГ СО РАН, представляет выполняемую программу в виде множества фрагментов вычислений – объектов исполнения, и множества фрагментов данных – чисел, массивов, структур и др., а также связей между ними [13]. В системе LuNA описание пользовательского кода разделено с описанием программы на специализированном языке программирования LuNA, а выполнением занимается исполнительная система [14]. Система LuNA позволяет автоматически генерировать исполняемый параллельный код, влиять на исполнение которого пользователь системы может с помощью т. н. рекомендаций – специальной мета информации, указание которой изменяет специфику исполняемой LuNA программы [15].

В последней версии системы LuNA используется мультиагентный подход [16]. Он сопоставляет каждому агенту некоторый фрагмент вычислений и набор соответствующих задач, которые должен выполнить этот агент. Агент фрагмента вычислений использует общую очередь задач и конкурирует с другими агентами за право исполнить свои задачи рабочими потоками. Количество рабочих потоков является статическим параметром, задаваемым по умолчанию системой без учета особенностей вычислительного узла или задаваемым через параметр запуска программы.

Состояния программы LuNA в разные моменты исполнения разнородны, из-за чего в очереди задач в разное время находится разное количество задач. Более того, влияет и фактор готовности входных фрагментов данных для соответствующих задач, который также со временем имеет свойство меняться. Таким образом, системе с ее постоянно изменяющимися состояниями требуется динамическая адаптация под состояние в данный промежуток времени. Это можно достичь с помощью алгоритма динамической адаптации количества рабочих потоков в системе LuNA.

1.2 Результаты обзора

В результате обзора существующих работ в области систем автоматического конструирования параллельных программ и средств параллельного программирования было установлено, что ни одна из систем не предоставляет алгоритмы динамической адаптации количества потоков во время исполнения. Количество потоков выбирается либо непосредственно пользователем, либо как константа исполнения. Также лишь некоторые из рассмотренных систем предоставляют механизмы отображения потоков на ядра узла.

Система фрагментированного программирования LuNA достаточно подходит в качестве базы для исследования алгоритма динамической адаптации количества рабочих потоков, поскольку, во-первых, благодаря своей модульной структуре система позволяет легко интегрировать новые алгоритмы, и во-вторых, фрагментированная структура LuNA программ позволяет отслеживать различные необходимые характеристики над множеством фрагментов вычислений, выделяя при этом накладные расходы самой исполнительной системы.

2 АЛГОРИТМ ДИНАМИЧЕСКОЙ АДАПТАЦИИ КОЛИЧЕСТВА РАБОЧИХ ПОТОКОВ

2.1 Необходимые термины и определения

Перед определением алгоритма динамической адаптации количества рабочих потоков необходимо ввести следующие термины и определения:

- Агент фрагмента вычислений – агент в смысле мультиагентной системы, реализующий соответствующий фрагмент вычислений;
- Атомарный блок кода агента – корутина, составляющая с аналогичными корутинами программы агентов фрагментов вычислений;
- Вычислительный узел – узел мультикомпьютера;
- Общая загрузка процессора – отношение совокупного времени работы всех потоков исполнения данного процесса к длине рассматриваемого промежутка времени, умноженного на количество ядер вычислительного узла;
- Полезная загрузка процессора – отношение совокупного времени выполнения фрагментов кода всеми рабочими потоками к длине рассматриваемого промежутка времени, умноженного на количество ядер вычислительного узла;
- Потоковое время – процессорное время, полученное данным потоком исполнения;
- Процессное время – время, вычисленное по отдельному таймеру для соответствующего процесса;
- Реальное время – время, вычисленное по общесистемным часам реального времени;
- Скользящее окно – смещающийся по временной последовательности на заданную единицу сдвига интервал, содержащий набор значений, которые будут использоваться на следующей итерации алгоритма;

- Фрагментированная программа – описание рекурсивно перечислимого множества фрагментов вычислений и их входных/выходных фрагментов данных [17];
- Фрагмент данных – агрегат из переменных [1];
- Фрагмент вычислений – независимая единица программы, содержащая описание входных и выходных фрагментов данных и кода (модуля или процедуры) фрагмента [17];
- Фрагмент кода – единица программы, предоставленная пользователем в виде программного модуля в определенном формате (например, процедура на традиционном языке программирования).

2.2 Постановка задачи

2.2.1 Неформальная постановка задачи

Фрагментированная программа состоит из единиц данных (фрагментов данных) и единиц исполнения (фрагментов вычислений). Под исполнением программы понимается исполнение фрагментов вычислений, на входе и выходе которых потребляются и вычисляются фрагменты данных соответственно. Фрагмент вычислений предполагает вызов фрагмента кода, а также других процедур исполнительной системы. Фрагмент вычислений есть вызов фрагмента кода. Фрагменты кода выполняют операции ввода/вывода данных, вычисления выходных данных, поэтому их выполнение составляет интерес пользователя системы LuNA. Для них во время использования системы LuNA важно иметь как можно меньше накладных расходов системы и как можно большей их доли работы в общем времени работы исполняемой программы. Поэтому будем считать выполнение любого фрагмента кода как полезную часть работы, а все остальное – накладные расходы системы.

В системе LuNA каждый фрагмент вычислений выполняется одним из рабочим потоков исполнительной системы. Одна из важных характеристик в

этом вопросе, на которую можно повлиять “со стороны”, является количество данных рабочих потоков. Количество рабочих потоков существенно влияет на производительность исполняемой LuNA программы в целом. Таким образом, изменение количества рабочих потоков является механизмом, с помощью которого можно попытаться уменьшить общее время работы программы и повысить долю полезной части ее работы.

В итоге, под созданием решения для уменьшения общего времени работы LuNA программы подразумевается разработка и реализация алгоритма автоматического изменения количества рабочих потоков. Такой алгоритм может рассматриваться как алгоритм динамической адаптации количества рабочих потоков под текущие условия на вычислительном узле.

2.2.2 Математическая постановка задачи

Количество рабочих потоков является изменяемой во времени величиной. Далее установим формально, относительно чего и как она будет определяться. Для этого введем функцию $f(t)$ – оптимальное количество потоков в момент времени t для исполняемой программы S на узле N. Функция $f(t)$ из своего определения является дискретной функцией. Область значений данной функции – это множество целых чисел от 1 до M, где M – некоторое максимальное количество потоков, допускаемое, например, операционной системой на узле запуска программы S. Область определения $f(t)$ – множество вещественных чисел от 0 до T_{total}, где T_{total} – общее время работы программы S.

На практике же, если в каждый момент времени t находить значение $f(t)$ и менять количество потоков согласно ему, то время, затраченное на создание рабочего потока, превышало бы потенциальный выигрыш изменения. Это связано прежде всего с тем, что изменение количества потоков требует немалое реальное время вследствие накладных расходов. Все эти рассуждения приводят к тому, что рассматривать функцию $f(t)$ на непрерывном отрезке $[0, T_{total}]$ на практике нецелесообразно.

Альтернативный и рабочий на практике подход для таких ситуаций является разбиение вещественной оси времени на промежутки относительно равных длин L. В таком случае область определения функции меняется – теперь аргументом функции является конечный отрезок времени, а значением функции – оптимальное статическое количество потоков на данном отрезке. При таком подходе функция является отображением дискретной величины на дискретную величину. Это позволяет выбрать приемлемую частоту обновления количества потоков, а также открывает возможность проводить периодический анализ функции на отрезках.

Функция $f(T)$ – оптимальное количество потоков на отрезке времени T длины L – имеет место лишь в абстрактном представлении. С ней можно работать в теории, но на практике работа с ней представляется сложной задачей. Это связано с тем, что существует большое количество независимых от программы S факторов, влияющих на нее. Учесть все эти факторы на практике очень сложно, поэтому будет далее удобно работать не напрямую с функцией $f(T)$, а с аппроксимацией для данной функции. Для этого построим целевую функцию $g(T)$ – приближенное оптимальное количество потоков на отрезке времени T для программы S на узле N.

Будем говорить, что $g(T)$ хорошо аппроксимирует $f(T)$, если для программы S на узле M время работы S, полученное с использованием функций изменения количества потоков $g(T)$, сопоставимо с той же величиной, полученной согласно функции $f(T)$. Проблема определения критерия сопоставимости времен выполнения, полученных по функциям $f(T)$ и $g(T)$, выходит за рамки данной ВКР.

На практике будем считать, что функция $f(T)$ есть константа, равная оптимальному статическому количеству рабочих потоков для программы S. Такую константу можно найти обычным перебором, поскольку множество допустимого количества потоков конечно. Критерий сопоставимости определим тогда следующим образом: время работы программ, полученных с использованием функций изменения количества потоков $g(T)$ и $f(T)$,

сопоставимо, если они отличаются не более чем в *1,5* раз. Таким образом, изначальная задача сводится к нахождению функции *g(T)*, хорошо аппроксимирующей функцию *f(T)*.

Функций *g(T)*, согласно данному определению, вообще говоря, существует бесконечно много. Для работы далее необходима любая из них, поэтому необходим алгоритм, с помощью которого будет возможно найти одну из таких функций. Такой алгоритм может использовать прогностический подход для поиска нового значения функции на следующем отрезке времени, используя информацию об уже пройденных отрезках. Таким образом, алгоритм будет динамически адаптировать значение функции *g(T)* – приближенное оптимальное количество рабочих потоков – на отрезках времени на основе прогностического подхода.

2.2.3 Требования, предъявляемые к алгоритму

Для того, чтобы алгоритм динамической адаптации количества рабочих потоков возможно было интегрировать в систему LuNA, а также для того, чтобы с помощью алгоритма стало возможно достичь хороших показателей производительности, выдвинем следующие требования к алгоритму:

1. Алгоритм должен быть реализован в виде отдельного слабо связного модуля;
2. Алгоритм должен требовать от исполнительной системы LuNA только два реализованных интерфейсных метода – удаление и добавление некоторого количества рабочих потоков;
3. Итоговое время работы LuNA программы с активированным алгоритмом должно быть сопоставимо или лучше со временем работы той же программы с оптимальным статическим количеством потоков (без активированного алгоритма) на некоторых практически значимых вычислительных задач;

4. Время работы LuNA программы с учетом всех совокупных накладных расходов на работу алгоритма (в том числе накладных расходов на возможную синхронизацию анализируемых данных) должно быть меньше или сопоставимо, чем время той же программы с количеством потоков, не близкому к оптимальному.

2.3 Описание предлагаемого решения

2.3.1 Идея предлагаемого решения

Ключевая идея алгоритма динамической адаптации рабочих потоков заключается в изменении количества потоков в том направлении, в котором происходит рост общей и полезной загрузки процессора. Алгоритм запоминает значения данных характеристик на предыдущих отрезках времени, вычисляет полезную и общую загрузку процессора на текущем отрезке X и на основе правила R изменяет количество потоков исполнения в пуле потоков исполнительной системы LuNA. Для вычисления общей и полезной загрузки процессора используется информация о времени работы фрагментов кода в различных единицах: реальное время, время работы процесса и время работы отдельного потока исполнения (далее – временные метки). Для оптимизации накладных расходов алгоритма по памяти используется механизм скользящего окна, который позволяет сохранить в памяти актуальные временные метки и при этом удалить уже невостребованные.

Рассмотрим далее основные части алгоритма (см. рисунок 1). Основными частями предлагаемого алгоритма являются:

1. Сбор информации о выполнении фрагментов кода, на основе которой будет вычисляться функция $g(T)$ во время исполнения программы. Данная информация будет поступать в представляющий алгоритм программный модуль из исполнительной системы в определенные события – завершения исполнения фрагментов кода. Таким образом, вместе с результатами вычислений фрагмента кода

система будет получать и информацию о времени его выполнения. Собранную информацию предлагается хранить в хранилище с табличной структурой данных;

2. Определение временных границ скользящего окна и границ анализа собранных данных. Алгоритм накапливает информацию о выполнении фрагментов кода и периодически запускает процесс анализа собранных данных. Поскольку на временной шкале алгоритм обладает информацией только в точках, соответствующих началу и концу выполнения фрагментов кода, длина каждого отрезка времени T разбиения не является постоянной величиной. Поэтому необходимо в процессе работы алгоритма находить границы отрезков анализа собранных данных и границы скользящего окна, используемого для оптимизации работы по памяти;
3. Определение общей загрузки процессора и полезной загрузки процессора. Данные характеристики являются целевыми для алгоритма: на основе их значений алгоритм выбирает количество потоков, которое будет работать на следующем отрезке времени;
4. Изменение количества рабочих потоков по зафиксированному правилу R на основе общей и полезной загрузок процессора. Правило R определяет то, как алгоритм будет интерпретировать полученные значения общей и полезной загрузок процессора: на вход правилу R подаются данные характеристики, а на выходе – количество потоков к добавлению или удалению.

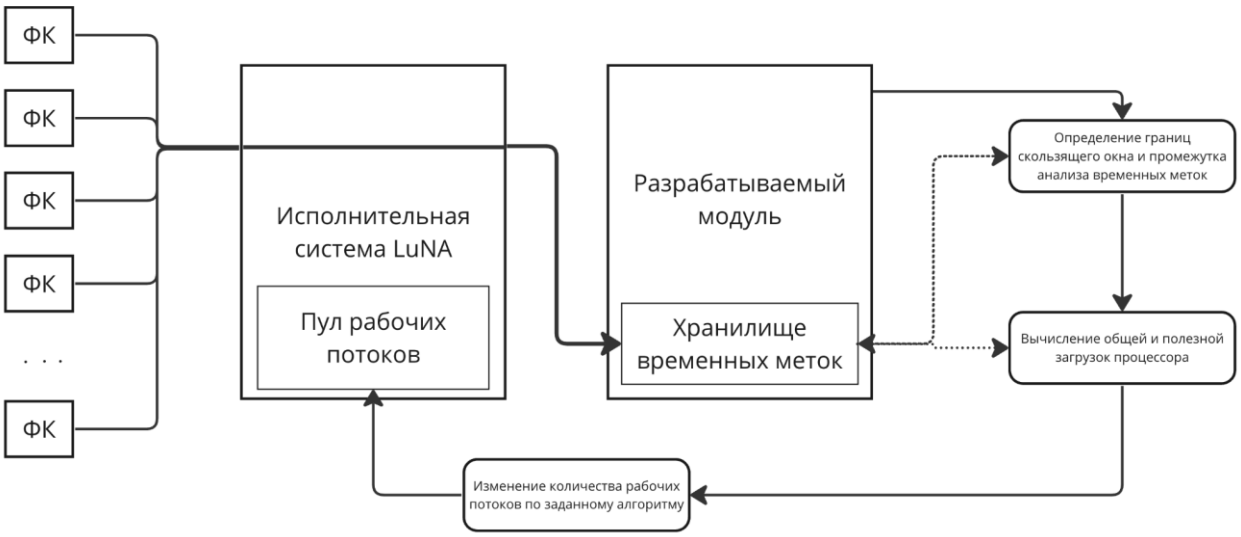

Рисунок 1 – Иллюстрация взаимодействия основных частей предлагаемого алгоритма

2.3.2 Сбор информации о выполнении фрагментов кода

Основной информацией, с которой работает алгоритм, являются временные метки о работе фрагментов кода. Алгоритм оперирует тремя типами времени:

1. Реальное время – монотонное время, пройденное с некоторой начальной точки отсчета;
2. Время работы процесса – суммарное время работы всех потоков исполнения данного процесса от момента запуска процесса;
3. Время работы потока исполнения – суммарное время работы данного потока исполнения от момента запуска процесса.

Для их получения вычисляются шесть значений: реальное время, время работы процесса и время работы потока исполнения непосредственно перед вызовом фрагмента кода, а также непосредственно после.

После вычисления данных временных меток они попадают в хранилище временных меток по определенному индексу так, чтобы все временные метки заданного потока исполнения однозначно определялись по некоторой формуле. Такую организацию хранения удобно разработать, используя операции по

модулю: если потоку исполнения присвоить некоторое уникальное значение I, то тогда будем считать, что он работает на ядре процессора с номером n и является k-м по счету созданным потоком на данном ядре, где $n = I \% c$ и $k = I / c$, а c – общее количество ядер на узле N. Использование данного правила позволяет однозначно сопоставить потоку с уникальным значением I ячейку в хранилище под индексом $[n][k]$.

Предложенный способ хранения временных меток позволяет в рамках одного общего хранилища без высоких накладных расходов организовать удобный доступ до ячеек хранилища каждому из рабочих потоков.

2.3.3 Определение границ скользящего окна и границ анализа собранных данных

С одной стороны, каждый поток исполнения записывает информацию о временных метках выполненных фрагментов кода. Поскольку время работы фрагментов кода заранее неизвестно, как неизвестно и количество выполненных потоком фрагментов вычислений, нельзя заранее сказать о размере хранилища временных меток. С другой стороны, вопрос разбиения вещественной оси времени на отрезки все еще остался не решенным.

Для того, чтобы в процессе работы память расходовалась рационально, а также для разбиения оси времени на отрезки, алгоритм использует механизм скользящего окна (см. рисунок 2). Его использование позволяет отслеживать и сохранять в памяти только те временные метки, которые как минимум один раз будут использованы далее в вычислении полезной и общей загрузок процессора. Границы же скользящего окна определяют очередной отрезок времени, причем можно задать смещение окна так, чтобы некоторые временные метки использовались в нескольких отрезках разбиения.

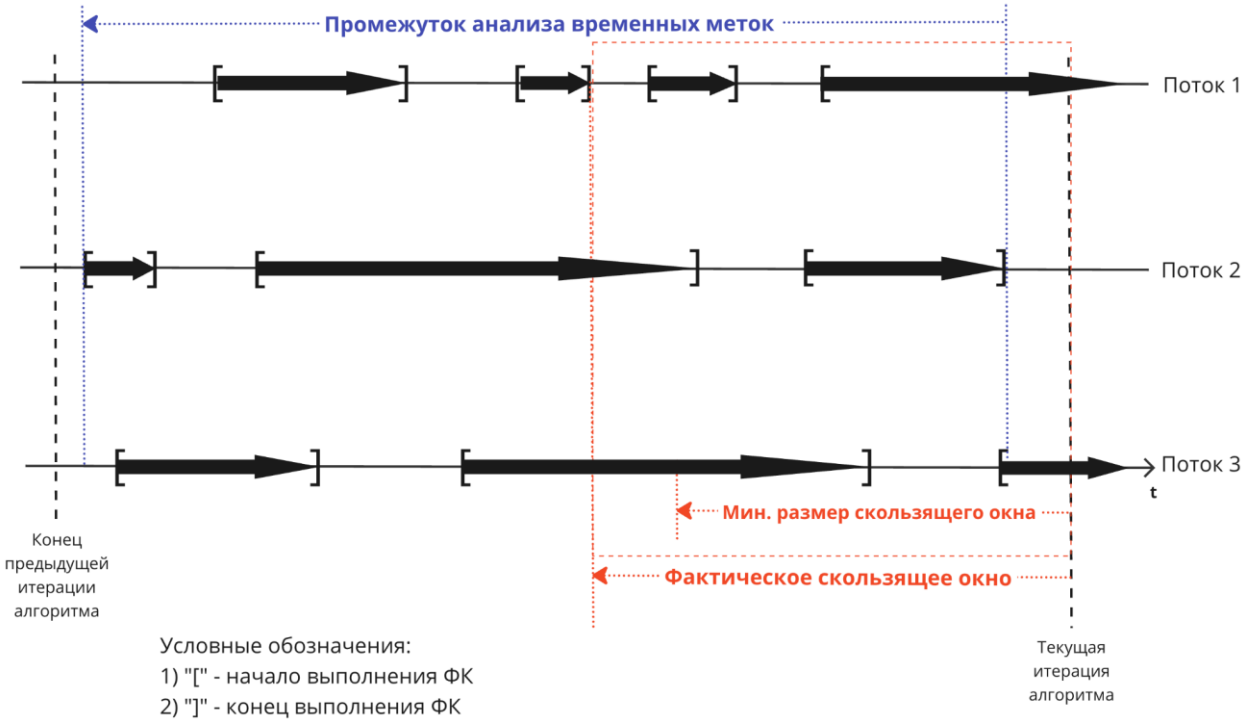

Рисунок 2 – Пример получения границ промежутка анализа временных меток и скользящего окна

Границы скользящего окна во время исполнения предлагается определять следующим образом: пусть программа S запущена на узле N с общим количеством ядер c. Пусть $\{I_1, \ldots, I_r\}$ – это последовательность созданных потоков исполнения на момент времени T_b. Зафиксируем минимальный размер скользящего окна S_w и отступим на шкале времени от значения T_b назад на величину S_w. В итоге получится новая величина $T_{in} := T_b - S_w$. Тогда заведомо длина любого отрезка $[t, T_b]$, где $t \leq T_{in}$, будет не меньше чем минимальный размер скользящего окна.

Заметим, что информация на непрерывной шкале времени есть только в точках начала и конца выполнения фрагментов кода. В таком случае значение t будем определять как самое позднее реальное время завершения фрагмента кода, при котором данное время все еще не больше T_{in}. Строго говоря, пусть $\{T_{1,j1}, \ldots, T_{r,jr}\}$ – это последовательность из неотрицательных вещественных значений, где $T_{i,jk}$ – это реальное время завершения фрагмента кода, выполненного потоком

исполнения с идентификатором i, причем $T_{i,jk} \leq T_{in}$, а $T_{i,\, j(k+1)} > T_{in}$. Тогда левая граница скользящего окна W_{left} будет определяться как $max(\{T_{1,j1}, \ldots, T_{r,jr}\})$.

Правая граница скользящего окна W_{right} будет определяться следующим образом: если в данный момент времени T_b множество $\{T_{1,j1}, \ldots, T_{r,jr}\}$ – это последовательность реальных времен последних завершенных фрагментов кода потоками $1,\ldots,r$ соответственно, то $W_{right} = min(\{T_{1,j1}, \ldots, T_{r,jr}\})$.

Таким образом, все временные метки фрагментов кода, реальное время завершения которых меньше W_{left}, можно удалить из памяти, считая их более не актуальными.

Смещение скользящего окна в процессе исполнения не является постоянной величиной и зависит от нового значения W_{left}. Действительно, пусть U – период проведения анализа временных меток, тогда в силу определения величины $W_{left} \neq U + W_{left_old}$. На практике, данная величина будет тем ближе к $U + W_{left_old}$, чем с большей частотой в хранилище будут попадать новые временные метки.

Границы анализа собранных временных меток на шкале времени определяются после применения операции удаления более не актуальных временных меток на предыдущей итерации алгоритма и до аналогичной операции на текущей итерации. На практике это означает, что на текущей итерации алгоритма (когда будут вычисляться общая и полезная загрузки процессора и меняться количество потоков) будут учитываться все временные метки, которые присутствуют в хранилище.

Таким образом, используя скользящее окно, становится возможным экономить оперативную память узла за счет поддержания только актуальных временных меток. Определение промежутка анализа собранных временных меток позволяет однозначно установить множество временных меток, которые учтутся в вычислении текущей общей и полезной загрузок процессора. Более того, если задать минимальный размер скользящего окна S_w больше, чем период проведения анализа временных меток U, то некоторые временные метки учтутся

в нескольких итерациях алгоритма. Это может быть полезно, поскольку в таком случае алгоритм с большой вероятностью не будет подвержен влиянию резких локальных изменений загрузки в текущий период. В текущей работе предлагается эмпирическим путем определить минимальный размер скользящего окна и период проведения анализа временных меток, исходя из результатов тестирования некоторых практически значимых LuNA программ.

2.3.4 Определение общей загрузки процессора

Общая загрузка процессора в контексте данной ВКР определяется как величина, равная отношению суммы времени работы всех потоков исполнения программы S за промежуток времени T к длине данного промежутка T, умноженного на общее число ядер процессора на узле N. Промежуток времени T есть промежуток анализа временных меток, определенный в разделе 2.3.3.

Общая загрузка процессора показывает, какую долю из максимально возможной суммы времени работы потоков, потоки программы S работали в реальности на промежутке T. Действительно, если на узле N количество ядер c, то за промежуток времени T потоки в идеале могли бы занять все ядра и непрерывно работать на них. В реальности же такого достичь сложно, поэтому при высокой загрузке всех ядер даже только одной программой S возникают разные накладные расходы (например, переключение контекста потоков исполнения, если их больше, чем количество ядер, либо если потоки могут перемещаться с ядра на ядро), что приводит к уменьшению общей загрузки процессора. Запущенные параллельно на узле N другие процессы также будут снижать введенный показатель. Данные факты необходимо учитывать при дальнейшей интерпретации данной полученной общей загрузки процессора (например, путем учета показателя загрузки процессора остальными процессами в диспетчере задач).

При условии, что на узле N пользователь, помимо служб операционной системы, запускает только программу S, общая загрузка процессора может быть

использована для того, чтобы с увеличением количества потоков попробовать увеличить долю времени, занимаемую потоками программы S на процессоре узла. Данная характеристика может быть особенно полезна на первых этапах выполнения программы S, когда количество потоков меньше или близко к количеству ядер. В таком случае можно увеличить количество потоков исполнения без вычисления полезной загрузки процессора, заняв некоторые простаивающие ядра узла.

2.3.5 Определение полезной загрузки процессора

В данной работе под полезной загрузкой процессора понимается величина, равная отношению суммы времени работы всех рабочих потоков только над фрагментами кода за промежуток времени T к длине данного промежутка T, умноженного на общее число ядер процессора на узле N. Промежуток времени T, как и ранее, есть промежуток анализа временных меток.

Полезная загрузка процессора учитывает не все потоки исполнения программы S, а только рабочие потоки, занимающиеся выполнением фрагментов вычислений. Она показывает, сколько суммарно времени все рабочие потоки потратили на выполнение фрагментов кода по отношению к максимально возможной величине. Поскольку в разделе “2.2.1. Общая постановка задачи” одной из главной задач являлось повышение доли полезной работы в общем времени выполнения программы S, показатель полезной загрузки процессора здесь играет очень важную роль. На его основе можно говорить о том, насколько эффективно исполнительная система позволяет выполнять фрагменты кода. С повышением доли полезной загрузки программа S быстрее выполнит все необходимые фрагменты кода, и это с большой вероятностью может привести к уменьшению времени работы всей программы S.

2.3.6 Правило изменения количества потоков исполнения

После получения таких показателей, как полезная и общая загрузки процессора, необходимо ими воспользоваться для изменения количества рабочих потоков. Задача поиска множества функции, на входе которых будут показатели общей и полезной загрузки процессора, а на выходе – число, показывающее насколько нужно изменить количество потоков (отрицательное, если их количество нужно уменьшить), имеет множество решений и поэтому требует дополнительного исследования. В рамках данной ВКР будет использоваться одно из решений, выведенное из эмпирических соображений.

Изменять количество рабочих потоков в общем случае предлагается следующим способом. Пусть на предыдущей итерации алгоритма показатель полезной загрузки процессора равнялся P_1 и количество потоков изменилось на k_1 единиц (k_1 – целое, не обязательно положительное). Пусть на текущей итерации алгоритма имеется полезная загрузка процессора P_2. Зафиксируем две постоянные величины – доля C (в процентах), изменение полезной загрузки на которую мы будем считать существенным, и целую положительную величину F – максимальное подряд количество итерации алгоритма, при которых не было существенного изменения полезной загрузки процессора.

Правило изменения количества рабочих потоков работает следующим образом:

1. Если $P_2 - P_1 \geq C$, то количество потоков на текущей итерации алгоритма необходимо изменить на $k_1 + sign(k_1)$. Данная формула прироста количества потоков представляет собой арифметическую прогрессию. Применение арифметической прогрессии позволяет за несколько итераций алгоритма значительно изменить количество рабочих потоков;
2. Если $P_1 - P_2 \geq C$, то количество потоков на текущей итерации алгоритма необходимо изменить на $-sign(k_1)$. Другими словами,

если полезная загрузка процессора на предыдущей итерации была больше, чем на текущей итерации на C, то направление изменения количества потоков следует поменять на противоположное;

3. Если $abs(P_2 - P_1) < C$, то изменение полезной загрузки процессора оказалось несущественным. В таком случае мы будем запоминать количество подряд итераций, на которых выполняется данное условие. Если же их количество оказалось равным F, то направление изменения количества рабочих потоков выбирается случайным образом. В любом случае количество потоков либо увеличится, либо уменьшится на единицу. Наличие такого условия необходимо для того, чтобы количество рабочих потоков не стало в какой-то момент постоянной величиной. Поскольку мы не знаем при таком условии, в каком направлении увеличится полезная загрузка процессора, остается только предположить это направление. Даже если предположение окажется неверным, то алгоритм рано или поздно все равно выйдет из плато по количеству потоков и продолжит искать их оптимум.

2.4 Характеристика предложенного алгоритма

Разработанный алгоритм позволяет адаптировать количество рабочих потоков под текущую загрузку узла на основе таких характеристик, как общая и полезная загрузки процессора. Алгоритм занимается сбором, накоплением и анализом временных меток, характеризующих выполнение фрагментов кода. Также алгоритм способен разделять актуальные и неактуальные метки с помощью механизма скользящего окна, удалять неактуальные метки, что создает меньше накладных расходов по памяти.

Изменяя количество рабочих потоков раз в период, алгоритм пытается приблизиться к их оптимальному количеству. Для того, чтобы это сделать по возможности быстрее, алгоритм использует арифметическую прогрессию в

вычислении числа потоков, которые необходимо добавить или удалить. Алгоритм также гарантирует, что адаптация будет происходить всегда и что количество рабочих потоков никогда не станет постоянной величиной с какого-то момента и до конца выполнения программы.

Алгоритм имеет несколько настраиваемых параметров, ключевой из которых – это период обновления количества потоков, который требуется выставлять эмпирическим путем. Стоит отметить, что хоть и частота обновления потоков остается постоянной величиной на протяжении всего выполнения программы, подобрать ее приемлемой в достаточной степени для большего класса задач проще, чем подбирать статическое количество потоков с той же целью.

Алгоритм не гарантирует, что время работы любой программы увеличится или уменьшится по сравнению со статическим количеством потоков. Алгоритм ставит своей целью приблизиться к такому количеству потоков, при которых достигаются высокие показатели общей и полезной загрузки процессора, пытаясь повысить таким образом производительность программы.

Алгоритм предполагает реализацию в виде отдельного слабо связанного с системой модуля, поэтому он может применяться и в других системах, отличных от LuNA. Такая возможность использования достигается путем низких требований к системе (всего две реализованные интерфейсные функции – удаление и добавление заданного количества потоков).

Таким образом, алгоритм полностью соответствует требованиям, предъявленным к нему в разделе 2.2.3.

3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА

Предложенный алгоритм динамической адаптации количества рабочих потоков был реализован на языке программирования C++ 11-го стандарта. Программная реализация была выполнена на базе системы фрагментированного программирования LuNA в виде отдельного модуля, реализующего разработанный алгоритм. Для данного модуля были написаны Руководство программиста (см. Приложение А) и Описание программы (см. Приложение Б).

В Разделе 3.1 приведены необходимые особенности реализации системы фрагментированного программирования LuNA. В Разделе 3.2 последовательно описаны шаги выполнения операций по изменению количества потоков реализованным модулем.

3.1 Особенности реализации системы фрагментированного программирования LuNA

Для того, чтобы встроить алгоритм динамической адаптации количества рабочих потоков в систему фрагментированного программирования LuNA, рассмотрим предварительно процесс компиляции LuNA программы.

Как уже упоминалось ранее в Разделе 1.1, в реализации системы фрагментированного программирования LuNA используется мультиагентный подход. Каждому фрагменту вычислений соответствует его один агент, который имеет доступ к исполнительной системе. Цель агента во время его существования – это реализация соответствующего фрагмента вычислений. Каждый агент имеет свою программу, которая обеспечивает достижение этой цели. Программы агентов состоят из набора атомарных блоков кода агента (см. раздел 2.1). Вызов фрагмента кода происходит в одном из таких атомарных блоков кода.

Компиляция программы на языке LuNA происходит посредством работы нескольких взаимосвязанных модулей, которые в конечном счете преобразуют код программы на языке LuNA в набор программ агентов на языке C++. Текст

LuNA программы в первую очередь переводится в абстрактное синтаксическое дерево (АСТ) на основе формата JSON, которое затем, после некоторых модификаций, транслируется в набор программ агентов.

3.2. Программная реализация алгоритма динамической адаптации количества потоков

3.2.1 Встраивание алгоритма в исполнительную систему LuNA

На этапе трансляции АСТ в набор программ агентов имеется возможность встраивать в них различную функциональность. На данном этапе происходит также и вставка вызовов фрагментов кода в конечные программы агентов. Поскольку алгоритму необходимо фиксировать время начала и время завершения фрагментов кода, достаточно воспользоваться снятием показаний с системных часов. Для этого предлагается добавить в генерируемый C++ код вызовы процедуры *clock_gettime()* [18] с различными параметрами до и после вызова фрагмента кода. Это позволит получить реальное пройденное время, время работы процесса и время работы потока, выполняющего фрагмент кода.

3.2.2 Передача временных меток на вход алгоритму

Исполнительная система LuNA хранит для каждого фрагмента вычислений объект класса *CF*, который соответствует состоянию исполняемого фрагмента вычислений. Передача собранных временных меток, полученных в результате вызовов *clock_gettime()*, в исполнительную систему LuNA осуществляется через процедуру *save_timestamps()*, добавленную в класс *CF* (в блоке вызова фрагмента кода есть указатель на объект класса CF). С помощью него же возможно передать в управляющий исполнительной системой класс *RTS* все необходимые алгоритму данные. Храниться эти данные будут непосредственно в реализуемом модуле, связанным с классом *RTS* в отношении “Композиция” (*RTS* включает реализуемый модуль, время жизни модуля прямо связано со временем жизни объекта *RTS*).

3.2.3 Выбор потока исполнения, выполняющего анализ временных меток

Рабочий поток после вызова фрагмента кода передает полученную временную метку в хранилище временных меток. Напомним, что анализ временных меток происходит раз в период (см. Раздел 2.3.3). В соответствии с этим, выбор потока исполнения, выполняющего анализ временных меток, был реализован следующим образом: пусть каждый рабочий поток после сохранения очередной временной метки проверяет, прошел ли данный период обновления количества потоков. Если с момента последнего обновления прошел зафиксированный период, то рабочий поток начинает выполнять анализ временных меток в рамках текущей задачи. При этом выставляется флаг в атомарную переменную (*std::atomic<bool> [19]*), сообщающий другим рабочим потокам о том, что в данный момент идет анализ. Это необходимо, поскольку в соответствии с алгоритмом только один поток может производить анализ временных меток (одновременный анализ временных меток несколькими потоками в текущей реализации мог бы привести к неопределенному поведению программы).

3.2.4 Особенности хранения временных меток

В процессе анализа временных меток есть, с одной стороны, рабочий потоки, который накапливает в отведенном ему месте хранилища временные метки, и, с другой стороны, поток, который данные временные метки анализирует. Поэтому может возникнуть ситуация одновременного доступа к хранилищу, когда один поток добавляет новый элемент, а другой их читает. Чтобы избежать последствий в виде неопределенного поведения программы от данной гонки за ресурсы, необходимо обеспечить синхронизацию доступа к хранилищу.

Также, поскольку количество рабочих потоков будет меняться динамически, необходима и динамическая структура данных. Помимо этого,

необходимо учесть особенности адресации, диктуемые модулем – обращаться модуль к хранилищу будет по идентификационному номеру рабочего потока, составленного из номера его процессорного ядра и его порядкового номера на этом ядре.

С учетом вышеперечисленных особенностей, в качестве хранилища временных меток предлагается использовать двумерный динамический массив, элементами которого будут выступать структуры данных типа дек, защищенные от одновременного доступа мьютексами (*std::mutex*). Введение мьютекса необходимо, поскольку дек из стандартной библиотеки C++ (*std::deque<T>*) не обеспечивает потокобезопасность [19]. Таким образом, “пишущий” рабочий поток будет добавлять временные метки в конец дека, а “читающий” поток – читать и удалять при необходимости с начала дека.

Хранилище временных меток можно было реализовать и другим способом (например, в виде односвязного списка для каждого рабочего потока). Однако, как показало отдельное тестирование, решение с использованием *std::mutex* для каждого рабочего потока не приводит к высокому повышению накладных расходов на время работы программы (отдельно измерялась доля времени в общем времени работы некоторых LuNA программ, затраченная на захват/освобождение и ожидание мьютекса: доля составила менее 2% для каждой программы. В общем случае доля зависит от конкретной задачи и может быть как больше, так и меньше 2%).

Предложенная реализация хранилища временных меток (представлена классом *AtomicDeque* разрабатываемого модуля) полностью соответствует описанной в Разделе “2.3.2. Сбор информации о выполнении фрагментов кода” модели хранения временных меток.

3.2.5 Особенности определения скользящего окна и интервала анализа временных меток

Вычисление общей и полезной загрузок процессора осуществляется после получения границ скользящего окна и интервала анализа временных меток, описанных в Разделе “2.3.3 Определение границ скользящего окна и границ анализа собранных данных”. Конец скользящего окна и конец интервала анализа временных меток совпадают, поэтому необходимо вычислить три величины – начало скользящего окна, начало интервала анализа временных меток и их общий конец.

Конец скользящего окна и конец интервала анализа временных меток вычисляется в процедуре *get_window_end_point()* реализованного модуля (везде далее упомянутые процедуры являются процедурами реализованного модуля) посредством линейного поиска по хранилищам временных меток всех рабочих потоков. При данном поиске модуль просматривает последние в деках временные метки потоков и находит самое раннее реальное время конца выполнения фрагмента кода. Эта величина и будет задавать общий конец данных промежутков.

Начало интервала анализа временных меток вычисляется в процедуре *get_processing_start_point()* также посредством линейного поиска. При данном поиске модуль просматривает первые в деках временные метки потоков и находит самое раннее реальное время начала выполнения фрагмента кода. Данная величина будет задавать начало интервала анализа на шкале времени.

Процедура вычисления начала скользящего окна *get_window_start_point()*, как и для величин ранее, использует линейный проход по хранилищам временных меток всех рабочих потоков. Она итерируется по всем временным меткам, у которых реальное время завершения фрагмента кода меньше либо равно текущему реальному времени за минусом минимального размера скользящего окна. Среди таких временных меток процедура ищет ту, у которой

реальное время завершения фрагмента кода более позднее. Последняя величина и определяет начало скользящего окна.

Стоит сделать несколько замечаний относительно алгоритмической сложности работы процедур *get_window_end_point(), get_processing_start_point()* и *get_window_start_point()*. Процедура *get_processing_start_point()* на шаге N алгоритма определяет временную метку, которая находится на нулевой позиции хранилища некоторого рабочего потока. Однако, информация об этой метке также есть и на шаге N-1, $N > 0$, поскольку при вычислении начала скользящего окна цикл по временным меткам потока останавливается на ней. Таким образом, если при вычислении начала скользящего окна запоминать необходимую метку, то *get_processing_start_point()* может работать за $O(1)$. Процедура *get_window_end_point()* прямо зависит от новых временных меток, поэтому возвращаемая из нее метка никак не может быть вычислена заранее и быстрее чем за $O(k)$, k – количество потоков, найти данную метку нельзя. Процедура *get_window_start_point()* также зависит от новых временных меток, поэтому поиск необходимой метки возможен за не константное время. В работе предлагается использовать линейный поиск по всем хранилищам потоков, то есть поиск с алгоритмической сложностью $O(kM)$, где k – количество потоков и M – максимальное количество временных меток среди всех потоков, накопленных за период, однако данную алгоритмическую сложность можно улучшить. Поиск идет для каждого потока, временные метки которого образуют упорядоченное множество пар <количество секунд, количество наносекунд>. Для упорядоченных множеств применим алгоритм бинарного поиска, поэтому итоговая алгоритмическая сложность может быть улучшена до $O(klogM)$.

3.2.6 Особенности определения общей загрузки процессора

Показатель общей загрузки процессора вычисляется в процедуре *get_cpu_load()* по двум точкам на шкале времени, в которых известно время работы процесса и реальное время. Этими самыми точками и являются

начальная и конечная точки промежутка анализа временных меток. В соответствии с Разделом 2.3.4, разница процессного времени в данных двух точках покажет совокупное время работы всех потоков данного процесса, то есть исполняемой программы.

3.2.7 Особенности определения полезной загрузки процессора

За реальное время *T*, прошедшее между точками начала и конца промежутка анализа временных меток, рабочие потоки выполняли задачи из очереди. Из всех задач, выполняемых рабочими потоками, для полезной загрузки процессора представляют интерес задачи, связанные с выполнением фрагментов кода. Полезная загрузка процессора призвана показать, какую долю от максимально возможного процессорного времени занимало выполнение фрагментов кода.

Каждый рабочий поток сохраняет в своем хранилище временные метки, в которых есть также информация о том, сколько времени рабочий поток реально выполнял тот или иной фрагмент кода – потоковое время работы. В таком случае возможно вычислить совокупное время, затраченное рабочими потоками на выполнение фрагментов кода. Для этого необходимо проитерироваться по всем временным меткам каждого рабочего потока и сложить все потоковые времена работы. При этом в сумму должны входить только те временные метки, которые попадают на промежуток анализа временных меток (процедура *process_payload()*).

Для оптимизации работы рассматриваемого модуля предлагается одновременно с вычислением полезной загрузки процессора определять неактуальные более временные метки и удалять их. Для этого временная метка сначала проверяется на принадлежность к промежутку анализа временных меток, далее если она ему принадлежит, то в общей сумме учитывается ее потоковое время работы, и потом проверяется, попадает ли временная метка в скользящее окно. Если она не попадает в скользящее окно, то по ранее данному

определению временная метка является неактуальной, и ее можно удалить из памяти. Такое решение позволит сократить время вычислений примерно в *2* раза, поскольку при нем не требуется еще одна итерация по всем временным меткам.

3.2.8 Изменение количества рабочих потоков

Изменение количества рабочих потоков происходит по алгоритму, описанному в разделе “2.3.6 Правило изменения количества потоков исполнения”. Алгоритм реализован в виде процедуры *process_load_stamps()* разрабатываемого модуля.

В начальный момент времени, когда отсутствует предыдущее значение полезной загрузки процессора, предлагается добавить 1 рабочий поток. Далее, когда на новом шаге работы алгоритма уже будет возможность сравнить текущую полезную загрузку процессора с предыдущей, изменять количество рабочих потоков становится возможным по описанному алгоритму.

Для изменения количества рабочих потоков модулю требуется доступ до пула рабочих потоков исполнительной системы. Именно поэтому необходимо было выставить требование к наличию реализованных интерфейсных процедур пула потоков – добавление (процедура *add_thread()*) и удаление (процедура *delete_thread()*) заданного количества потоков.

При удалении рабочего потока в стеке сохраняется его идентификатор, который может быть использован в будущем при создании рабочего потока. При этом важно, чтобы рабочий поток, выполняющий анализ, не запросил удаление самого себя. Поэтому при удалении необходима проверка на самоудаление (рабочий поток, зная свой идентификатор, выбирает идентификатор удаляемого потока; если идентификаторы совпали, то выбирается следующий поток по циклу и удаляется).

При создании рабочего потока ему выдается идентификатор либо ранее использованный другим, уже удаленным рабочим потоком, либо новый. Для последнего случая предлагается поддерживать количество созданных потоков

threads_count, и если освободившихся идентификаторов нет, то выдавать рабочему потоку идентификатор, равный *threads_count*, а саму величину *threads_count* инкрементировать.

3.2.9 Дополнительная функциональность по логированию работы алгоритма

Для отслеживания работы алгоритма динамической адаптации количества рабочих потоков, его отладке были реализованы два вида логирования хода работы – отслеживание текущего состояния в стандартный поток вывода и отслеживание общих трендов изменения количества потоков в файл формата CSV. Данный формат позволяет быстро импортировать логи алгоритма в современные программы по работе с таблицами и визуализировать данные из них.

Первый тип логирования (см. рисунок 3) позволяет во время исполнения программы наблюдать, потоки с какими идентификаторами удаляются или добавляются на какое ядро. Также выводится информация об общем количестве рабочих потоков, количестве задач в очереди, количестве временных меток у каждого из рабочих потоков.

```
INFO:   New thread [7] mapped on cpu [7]
INFO:   New thread [8] mapped on cpu [0]
INFO:   New thread [9] mapped on cpu [1]
INFO:   10 Th, 17394 Jb, 7 RJ
Cpu [0], thread no [0] has size = 607
Cpu [1], thread no [0] has size = 604
Cpu [2], thread no [0] has size = 601
Cpu [3], thread no [0] has size = 572
Cpu [4], thread no [0] has size = 486
Cpu [5], thread no [0] has size = 203
Cpu [6], thread no [0] has size = 205
```

Рисунок 3 – Пример логирования первого типа

Логирование данного типа срабатывает в каждый период обновления количества потоков. Оно может быть полезно при отладке, выявлении периодов высокой нагрузки, оптимизации фрагментированной программы, подбора других параметров системы.

Логирование второго типа (см. рисунок 4) также раз в период обновления количества потоков выводит в файл .csv следующую информацию: реальное время создания записи в лог-файле, общая загрузка процессора, полезная загрузка процессора, количество рабочих потоков, количество задач в очереди в момент создания записи, а также величину последнего изменения количества потоков.

```
time, cpuload, payload, threads, jobs, trend, status
1680872471.82, 58.248338, 45.537552, 5, 20396, 2, Growth
1680872479.82, 66.586555, 53.930929, 7, 18831, 3, Growth
1680872487.82, 77.609286, 65.004397, 10, 17394, 4, Growth
1680872495.82, 85.766096, 74.905049, 14, 16137, 5, Growth
1680872503.82, 91.142514, 82.068153, 19, 14983, 6, Growth
1680872511.82, 88.295997, 80.854370, 18, 13980, -2, Single fall
1680872519.82, 82.043321, 75.362502, 19, 13057, 2, Single growth
1680872527.82, 78.157566, 71.484704, 18, 11955, -2, Single fall
1680872535.82, 75.571546, 68.618821, 19, 10937, 2, Single growth
1680872543.82, 75.771230, 69.044296, 19, 9933, 2, Single growth
1680872551.82, 75.759276, 69.026246, 19, 8880, 2, Single growth
1680872559.82, 74.443303, 67.633185, 18, 0, -2, Single fall
```

Рисунок 4 – Пример логирования второго типа

Второй тип логирования работы алгоритма может быть полезен при визуализации изменения количества потоков, полезной и общей загрузки процессора. Именно данный тип был использован во время тестирования работы модуля на некоторых практически значимых задачах.

3.3 Характеристика реализованного модуля

Таким образом, описанный модуль полностью реализует предложенный в Главе 2 алгоритм динамической адаптации количества потоков. Модуль выставляет лишь одно требование к исполнительной системе – наличие

реализованных процедур добавления и удаления заданного количества рабочих потоков в пуле потоков. Вычисление нового количества потоков происходит на основе анализа собранных временных меток, соответствующих исполненным фрагментам кода. Во время анализа временных меток вычисляются две введенные характеристики: общая и полезная загрузки процессора, которые в конечном счете применяются в правиле изменения количества потоков.

4 ТЕСТИРОВАНИЕ

Для проверки работоспособности и эффективности модуля алгоритма динамической адаптации количества рабочих потоков было проведено тестирование на различных LuNA программах, представляющих практический интерес. Тестирование всех программ производилось на восьмиядерном узле с 16 Гб ОЗУ. Тестирование задачи блочного умножения матриц было проведено также на кластере ИВЦ НГУ [20]: запрашивались два 12-ядерных вычислительных узла с 16 Гб ОЗУ на каждом.

Общая идея тестирования заключается в сравнении времени работы тестируемых фрагментированных программ, исполненных со статическим количеством рабочим потоком и с динамическим количеством рабочих потоков. Для статического количества рабочих потоков у всех тестируемых программ были выбраны значения 4 (значение в системе LuNA по умолчанию), 8 (значение, равное количеству ядер на компьютере) и статическое оптимальное для каждой задачи по отдельности (значение, полученное перебором), если это значение не совпало с 4 и 8. Начальное количество рабочих потоков для динамического режима – половина количества ядер, то есть 4. Период обновления количества потоков выставлен 4 секунды (данный период был установлен эмпирическим путем как достаточно приемлемый на заданном наборе задач). На кластере ИВЦ НГУ были выбраны соответствующие значения статического количества потоков с поправкой на количество ядер на каждом из узлов. Параметры тестируемого алгоритма динамической адаптации количества потоков при запуске на кластере не изменялись.

4.1 Тестирование алгоритма на задаче блочного умножения матриц

Работоспособность и эффективность алгоритма динамической адаптации количества рабочих потоков были протестированы на задаче блочного умножения матриц, реализованной в виде фрагментированной программы на языке LuNA.

Для блочного умножения матриц характерно большое количество однотипных операций умножения и равномерное изменение количества задачи в очереди. Исполнительная система генерирует для каждых подматриц свои задачи в очереди, далее по мере возможностей рабочие потоки выполняют их.

Для задач с равномерным изменением количества задач в очереди ожидается, что алгоритм динамической адаптации количества рабочих потоков сможет относительно близко приблизиться к оптимальному количеству потоков, но при этом не даст большого прироста по сравнению со статическим количеством потоком, равному количеству ядер узла. Для таких задач динамичность изменений оказывается менее эффективна в силу отсутствия большой необходимости в адаптации количества потоков.

Результаты тестирования эффективности алгоритма динамической адаптации количества рабочих потоков на задаче блочного умножения матриц представлены в таблице 1, а также на рисунках 5 и 6.

Таблица 1 – Сравнение эффективности алгоритма на задаче блочного умножения матриц

Количество рабочих потоков	Время работы, сек	Ускорение
Статическое со значением 4	130,11	0,604
Статическое со значением 8	78,57	1,000
Статическое со значением 10	86,57	0,908
Статическое со значением 14	91,17	0,862
Динамическое с начальным значением 4	79,32	0,991

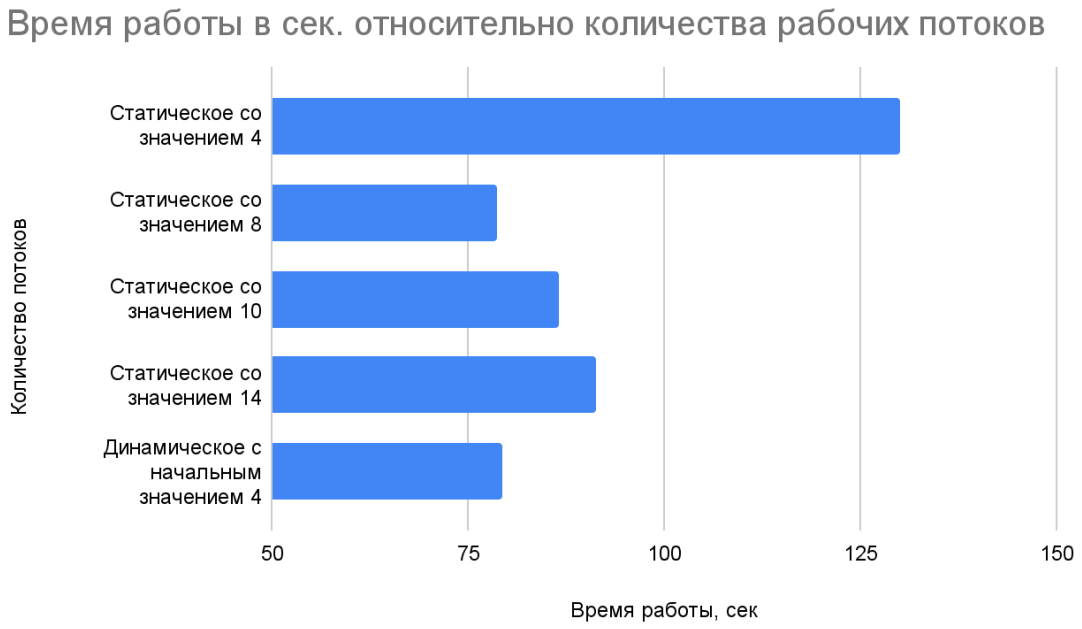

Рисунок 5 – Сравнение времени работы задачи блочного умножения матриц при различном количестве рабочих потоков

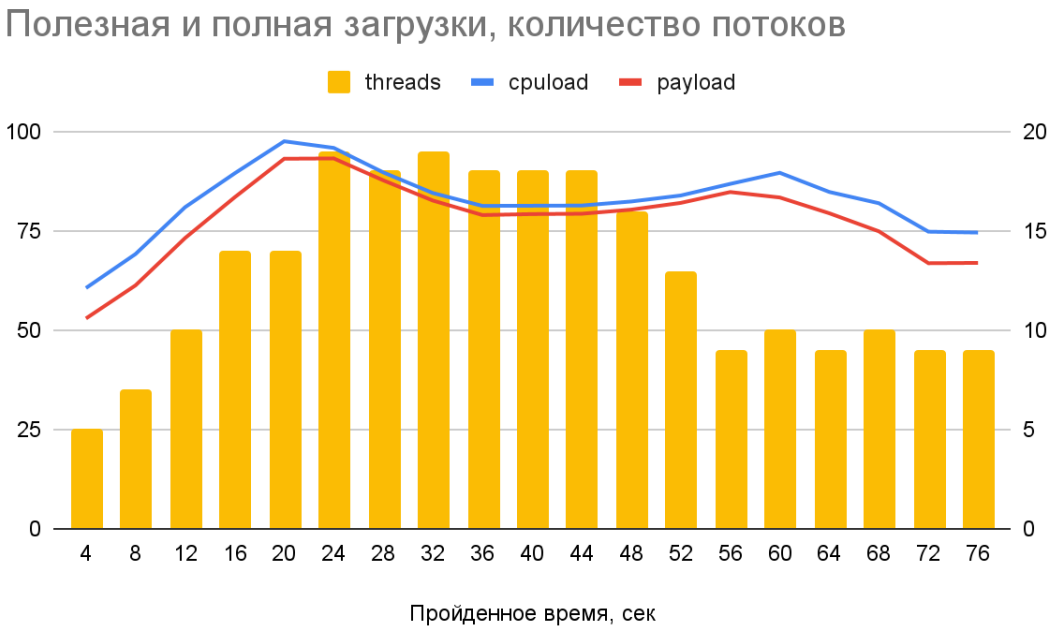

Рисунок 6 – Показатели полезной и полной загрузки процессора и количества потоков для задачи блочного умножения матриц

Результаты тестирования подтверждают ожидаемое: заняв все ядра на узле, рабочие потоки выполняют задачи и постепенно уменьшают размер очереди. Количество потоков при этом некоторое время было примерно в 2 раза

больше, чем количество ядер, но далее алгоритм распознал данную избыточность и уменьшил количество потоков до 7-9, что почти совпадает с оптимальным количеством (8 потоков) и тем самым дает более высокую производительность.

Результаты тестирования эффективности алгоритма динамической адаптации количества рабочих потоков на задаче блочного умножения матриц на кластере ИВЦ НГУ представлены в таблице 2, а также на рисунке 7.

Таблица 2 – Сравнение эффективности алгоритма на задаче блочного умножения матриц (кластер ИВЦ НГУ)

Количество рабочих потоков	Время работы, сек	Ускорение
Статическое со значением 8	169,07	0,816
Статическое со значением 12	137,94	1,000
Статическое со значением 14	136,4	1,011
Статическое со значением 16	139,28	0,990
Статическое со значением 24	141,55	0,974
Динамическое с начальным значением 4	136,79	1,008

Рисунок 7 – Сравнение времени работы задачи блочного умножения матриц при различном количестве рабочих потоков (кластер ИВЦ НГУ)

Результаты тестирования задачи блочного умножения матриц несколько отличаются от результатов на одном узле: на кластере статическое оптимальное количество потоков оказалось чуть большим, чем количество ядер на узлах. Это связано с тем, что при количестве узлов больше одного начинает принимать участие коммуникационная подсистема LuNA. Рабочие потоки могут простаивать в силу ожидания фрагментов данных с другого узла по сети. В таком случае они освобождают ядра, позволяя другим потокам на них выполняться. Поэтому становится более выгодно иметь количество потоков чуть больше, чем количество ядер. Именно эти рабочие потоки могут занять свободные ядра и тем самым ускорить программу.

Таким образом, тестирование на задаче блочного умножения матриц показало, что алгоритм удовлетворяет его нефункциональным требованиям. Время работы с динамической адаптацией количества рабочих потоков почти аналогично времени со статическим количеством, равному количеству ядер (оптимальное статическое количество для данной задачи).

4.2 Тестирование алгоритма на итерационной задаче решения уравнения Пуассона

Эффективность алгоритма динамической адаптации количества рабочих потоков была также протестирована на задаче решения уравнения Пуассона итерационным методом, реализованной в системе фрагментированного программирования LuNA [21].

Для данной задачи свойственно малое количество задач в очереди почти на всем протяжении времени работы. Сначала формируется большое количество задач, связанных с инициализацией данных, которые выполняются в течение нескольких секунд. После большая часть вычислений выполняется в рамках нескольких задач. Очередь задач довольно часто остается пустой, поскольку рабочие потоки успевают выполнять задачи быстрее, чем они формируются в очереди.

Для задач с малыми размерами очередей ожидается, что алгоритм динамической адаптации количества рабочих потоков подстроится под частоту появления новых задач в очереди. Результаты тестирования эффективности алгоритма на задаче решения уравнения Пуассона итерационным методом отражены в таблице 3, а также на рисунках 8 и 9.

Таблица 3 – Сравнение эффективности алгоритма на задаче решения уравнения Пуассона итерационным методом

Количество рабочих потоков	Время работы, сек	Ускорение
Статическое со значением 4	237,7	0,940
Статическое со значением 8	223,42	1,000
Статическое со значением 9	220,28	1,014
Статическое со значением 10	218,53	1,022
Статическое со значением 12	222,95	1,002
Статическое со значением 16	226,04	0,988
Динамическое с начальным значением 4	222,28	1,005

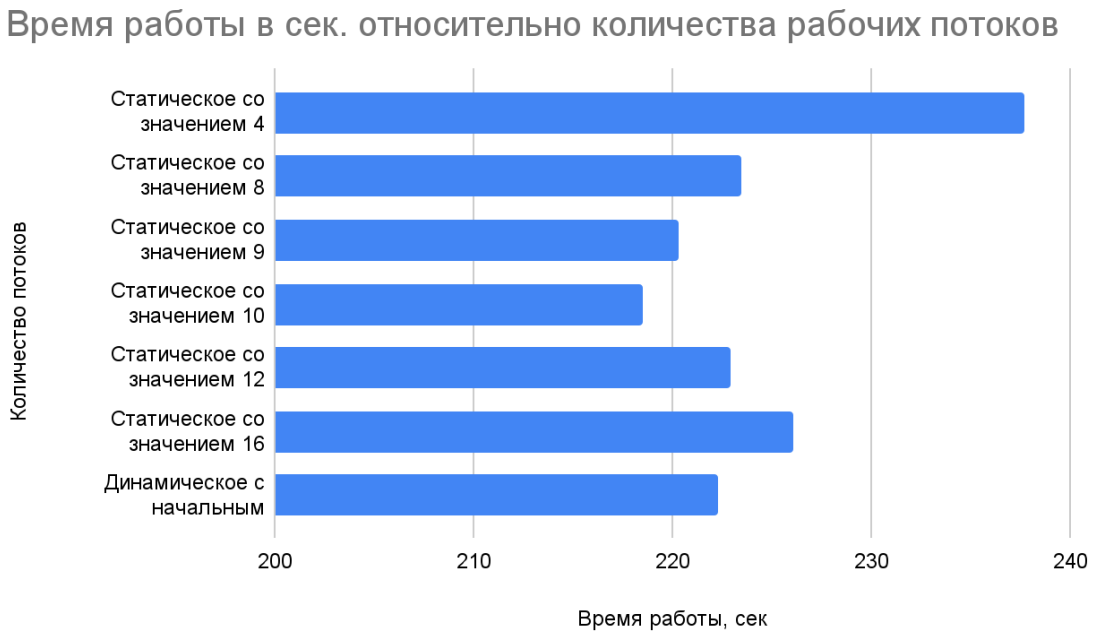

Рисунок 8 – Сравнение времени работы задачи решения уравнения Пуассона итерационным методом при различном количестве рабочих потоков

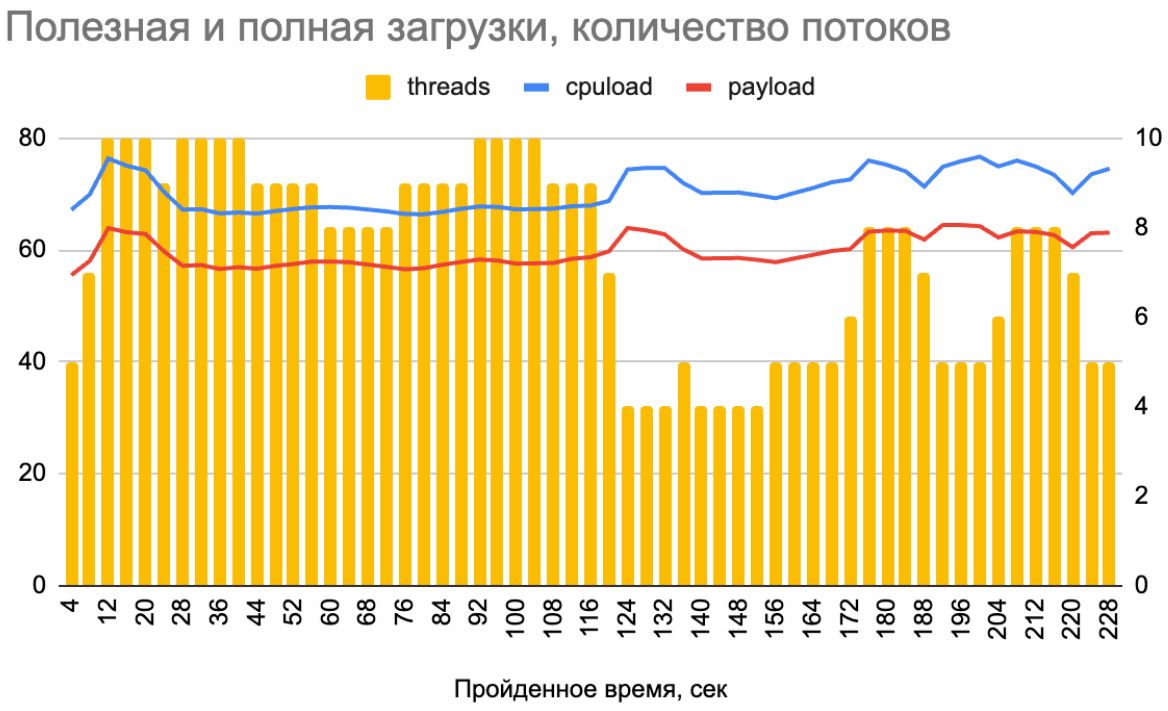

Рисунок 9 – Показатели полезной и полной загрузки процессора и количества потоков для задачи решения уравнения Пуассона итерационным методом

Результаты тестирования на задаче решения уравнения Пуассона итерационным методом показывают, что время работы фрагментированной программы с динамическим количеством потоков не всегда меньше, чем с некоторым статическим неоптимальным. Особенно это свойственно задачам с малыми размерами очередей. Поэтому также необходимо предоставить пользователю возможность запускать LuNA программы как с заданным статическим, так и с динамическим количеством потоков.

Результаты тестирования на задаче решения уравнения Пуассона итерационным методом также показывают и то, что заранее поставить заведомо большое статическое количество потоков – это неудовлетворительная стратегия. Действительно, как видно из результатов, начиная с некоторого статического количества потоков общее время работы программы начинает расти. Алгоритм же динамической адаптации количества потоков способен избежать данный случай.

4.3 Тестирование алгоритма на решателе модельного трехмерного уравнения теплопроводности в единичном кубе

Последней практически значимой задачей, на которой проводилось тестирование эффективности алгоритма динамической адаптации количества рабочих потоков, был решатель модельного трехмерного уравнения теплопроводности в единичном кубе. Программное решение было реализовано в виде фрагментированной LuNA программы совместно с Казахским национальным университетом [22].

Для решателя модельного трехмерного уравнения теплопроводности в единичном кубе характерно неравномерное изменение размера очереди задач в ходе работы программы. Агенты порождают новые задачи порциями, поэтому в разные моменты времени размер очереди бывает то небольшим (порядка 10-100 задач), то относительно большим (1000-5000 задач). Ожидается, что именно для таких задач динамическая адаптация количества рабочих потоков может

показать себя с лучшей стороны – при достаточно высокой частоте обновления (2-6 секунд) количества потоков алгоритм позволяет подстроиться под текущие изменения, быстро находя оптимальные показатели количества потоков в данный промежуток.

Результаты тестирования эффективности алгоритма на решателе модельного трехмерного уравнения теплопроводности в единичном кубе отражены в таблице 4, а также на рисунках 10 и 11.

Таблица 4 – Сравнение эффективности алгоритма на решателе модельного трехмерного уравнения теплопроводности в единичном кубе

Количество рабочих потоков	Время работы, сек	Ускорение
Статическое со значением 4	228,16	0,985
Статическое со значением 8	224,84	1,000
Статическое со значением 11	215,05	1,046
Статическое со значением 14	219,97	1,022
Статическое со значением 18	216,74	1,037
Статическое со значением 23	226,29	0,994
Динамическое с начальным значением 4	214,32	1,049

Рисунок 10 – Сравнение времени работы решателя модельного трехмерного уравнения теплопроводности в единичном кубе при различном количестве рабочих потоков

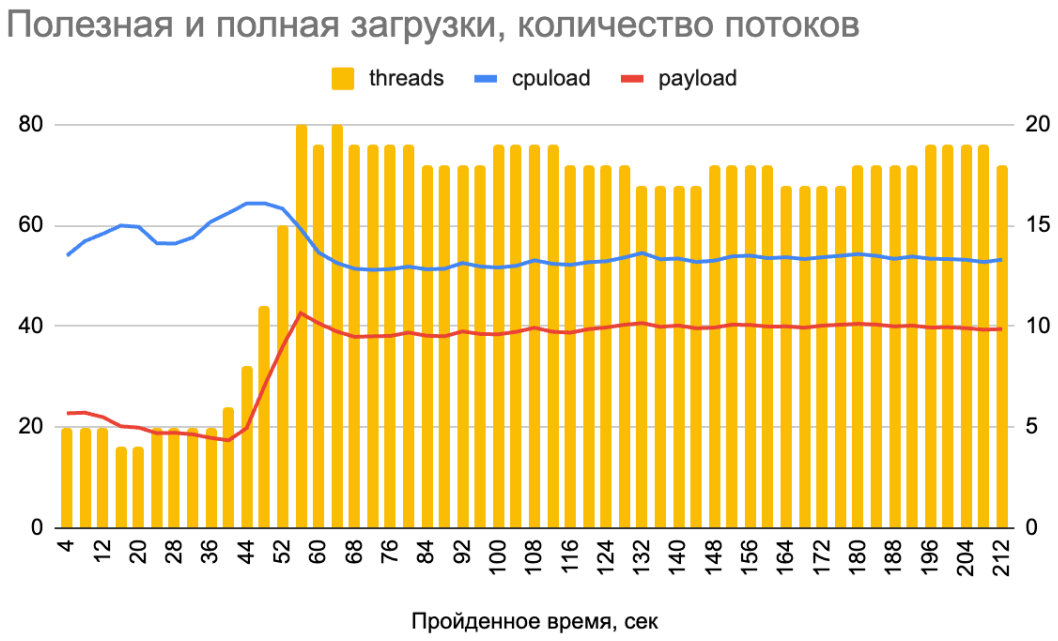

Рисунок 11 – Показатели полезной и полной загрузки процессора и количества потоков для решателя модельного трехмерного уравнения теплопроводности в единичном кубе

Результаты тестирования алгоритма на решателе модельного трехмерного уравнения теплопроводности в единичном кубе показали, что, действительно, динамическая адаптация количества потоков на “неравномерной” задаче дает преимущества перед статическим количеством – программа исполнилась быстрее. При этом из графиков полной загрузки процессора также видно, что не все ядра компьютера были постоянно загружены. Это связано с тем, что порой для создания новых задач требуется завершение некоторых предыдущих. Поэтому рабочие потоки иногда простаивали, ожидая появления новых задач, что и отразилось на полезной загрузке процессора.

4.4 Итоги тестирования

Результаты тестирования подтверждают работоспособность алгоритма динамической адаптации количества рабочих потоков и его эффективность для своего класса задач. По результатам можно увидеть, что динамическая адаптация количества потоков позволила уменьшить время работы фрагментированной программы по сравнению со статическим количеством, заданным в системе LuNA по умолчанию. Более того, алгоритм показал, что для некоторых фрагментированных программ статическое количество потоков, равное количеству ядер на узле, не всегда является оптимальным.

Алгоритм динамической адаптации количества рабочих потоков хоть и не гарантирует ускорение фрагментированной программы, но для задач, которым характерны большие изменения размеров очередей, использование алгоритма имеет смысл. Для других задач алгоритм дает сопоставимые результаты со статическим количеством потоков, равному количеству ядер. Таким образом, использование алгоритма позволяет полностью освободить пользователя от установки количества рабочих потоков, а для своего класса задач и вовсе дать ускорение.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы был разработан и реализован алгоритм динамической адаптации количества рабочих потоков на основе введенных характеристик полезной и общей загрузок процессора. Реализованный алгоритм был встроен в систему фрагментированного программирования LuNA в качестве модуля исполнительной системы LuNA. Основной целью алгоритма является ускорение работы фрагментированных программ.

Было проведено тестирование разработанного алгоритма, в ходе которого сравнивалась производительность LuNA программ с динамическим и статическими количествами рабочих потоков. Результаты тестирования продемонстрировали работоспособность алгоритма и его эффективность. С помощью алгоритма удалось ускорить некоторые LuNA программы, решающие практически значимые вычислительные задачи.

Данная работа была представлена на 61-й Международной научно-студенческой конференции, г. Новосибирск, 2023 г. [23]

Защищаемые положения:

1. разработан алгоритм динамической адаптации количества рабочих потоков;
2. разработанный алгоритм был реализован и встроен в систему фрагментированного программирования LuNA в качестве модуля исполнительной системы LuNA;
3. проведено экспериментальное исследование работоспособности и эффективности реализованного алгоритма.

В дальнейшем планируется продолжить работу над модулем динамической адаптацией количества потоков в системе фрагментированного программирования LuNA. Возможными направлениями для развития являются:

1. Усовершенствование алгоритмов, по которым выбирается количество рабочих потоков, исходя из общей и полезной загрузок процессора;

2. Расширение модуля динамической адаптации количества потоков с целью учета всех видов потоков исполнения (сетевых, возможных новых системных) в системе LuNA;
3. Интеграция алгоритма динамической адаптации количества потоков с алгоритмами балансировки нагрузки с целью возможности работы не только в рамках одного узла, но и всего мультикомпьютера.

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

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

_____ _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.
(заполняется от руки)

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Malyshkin, V. E. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem / V. E. Malyshkin, V. A. Perepelkin // Proceedings of the 11th Conference on Parallel Computing Technologies. – 2011. – Vol. 6873. – P. 53–61

2. Laxmikant, V. Kale. Charm++: A Portable Concurrent Object Oriented System Based on C++ / Laxmikant V. Kale, Sanjeev Krishnan // 1993 OOPSLA. – 1993. – 18 P.

3. Seonmyeong, B. Integrating OpenMP into the Charm++ Programming Model / Seonmyeong Bak, Harshitha Menon, Sam White, Matthias Diener, Laxmikant Kale // 2017 ESPM2. – 2017. – 7 P.

4. Gengbin, Z. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers / Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, Laxmikant V. Kale // 2010 P2S2. – 2010. – 9 P.

5. The Charm++ Parallel Programming System. Официальная документация Charm++ [Электронный ресурс]. – Режим доступа: https://charm.readthedocs.io/en/latest/charm++/manual.html (дата обращения 15.02.2023).

6. Bosilca, G. PaRSEC: Exploiting Heterogeneity to Enhance Scalability / George Bosilca, Anthony Danalis // Institute of Electrical and Electronics Engineers: Computing in Science and Engineering – 2013. – Article N15 – P. 36 – 45.

7. PaRSEC. Официальный репозиторий Bitbucket [Электронный ресурс]. – Режим доступа: https://bitbucket.org/icldistcomp/parsec/wiki/browse/ (дата обращения 15.12.2022)

8. Bosilca, G. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. – 2013. – Article – 99 P.

9. Bauer, M. Legion: Expressing Locality and Independence with Logical Regions / Michael Bauer, Sean Treichler, Elliott Slaughter, Alex Aiken // 2012 International

Conference for High Performance Computing, Networking, Storage and Analysis. – 2012. – 11 P.

10. Using OpenMP. Portable Shared Memory Parallel Programming / Barbara Chapman, Gabriele Jost, Ruud van der Pas. – Cambridge, The MIT Press. – 2007. – 384 P.

11. Official OpenMP API Specification. Официальная спецификация API стандарта OpenMP [Электронный ресурс]. – Режим доступа: https://www.openmp.org/spec-html/5.0/openmpsu35.html (дата обращения 27.11.2022)

12. Official GNU Compiler Collection documentation. Официальная документация компилятора GCC [Электронный ресурс]. – Режим доступа: https://gcc.gnu.org/onlinedocs/gcc-4.6.4/libgomp/index.html (дата обращения 25.01.2023)

13. Малышкин В. Э. Синтез параллельных программ и систем на вычислительных моделях. / Вальковский В.А., Малышкин В. Э. // Издательство "Наука", Сибирское отд-е, Новосибирск – 1988. – 129 стр.

14. Malyshkin, V. E. Active Knowledge, LuNA and Literacy for Oncoming Centuries / Malyshkin V. E. // Springer, LNCS, Vol. 9465. – pp. 292-303

15. Валитов, А. А. Управление распределенными фрагментированными данными в системе LuNA. / Валитов А.А., Перепелкин В. А. // Всероссийская летняя XXXVII молодежная Школа-конференция по параллельному программированию с международным участием (г. Новосибирск, 5 – 16.07.2021 г.). Всероссийская летняя XXXIX молодежная Школа-конференция по параллельному программированию с международным участием (г. Новосибирск, 4 – 15.07.2022 г.): Тез. докл. / отв. ред. д-р техн. наук, проф. В. Э. Малышкин. Новосиб. гос. унт. – Новосибирск: ИПЦ НГУ, 2023 (в печати).

16. Akhmed-Zaki, D., Lebedev, D., Malyshkin, V., Perepelkin, V. Automated construction of high performance distributed programs in LuNA system // 15th

International Conference on Parallel Computing Technologies, PaCT 2019; Almaty; Kazakhstan. LNCS 11657. Springer, 2019. pp. 3 – 9

17. Малышкин, В. Э. Технология фрагментированного программирования / В. Э. Малышкин // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. – 2012. – N 46 (305). – С. 45 – 55.

18. Ubuntu man pages. Справочные материалы по системным вызовам ОС Ubuntu [Электронный ресурс]. – Режим доступа: https://manpages.ubuntu.com/manpages (дата обращения 01.04.2023).

19. C++ programming language documentation. Документация на язык программирования C++ [Электронный ресурс]. – Режим доступа: https://en.cppreference.com/w/ (дата обращения 03.04.2023).

20. Информационно-вычислительный центр Новосибирского государственного университета [Электронный ресурс]. – Режим доступа: http://nusc.nsu.ru/ (дата обращения 12.04.2023).

21. Киреев, С. Е. Фрагментация метода частиц-в-ячейках // Мат. 9-й Росс. конф. с междунар. участием "Новые информационные технологии в исследовании сложных структур". – Томск: Изд-во НТЛ. - 2012 - С. 8.

22. Akhmed-Zaki, D. Zh. Automation of distributed numerical programs construction in LuNA system on a model application example / D. Zh. Akhmed-Zaki, D.V. Lebedev, V.E. Malyshkin, V.A. Perepelkin // Problems of Informatics – 2019. – Article N3 – 12 P.

23. Валитов, А. А. Разработка и реализация алгоритма динамической адаптации количества рабочих потоков в системе LuNA / А. А. Валитов // Информационные технологии: Материалы 61-й Междунар. науч. студ. конф. 17–26 апреля 2023 г. / Новосиб. гос. ун-т. — Новосибирск: ИПЦ НГУ, 2023 (в печати)

ПРИЛОЖЕНИЕ А

Модуль динамической адаптации количества рабочих потоков в системе LuNA

Руководство программиста

Листов 11

Новосибирск, 2023

Содержание

Аннотация 59
1. Назначение и условия применения программы 60
1.1 Назначение программы 60
1.2 Функции, выполняемые программой 60
1.2.1 Внешние функции, выполняемые программой 60
1.2.2 Внутренние функции, выполняемые программой 61
1.3 Условия, необходимые для выполнения программы 61
2. Характеристика программы 63
3. Обращение к программе 64
4. Входные и выходные данные 65
4.1 Организация используемой входной информации 65
4.2 Организация используемой выходной информации 65
5. Сообщения 66
6. Лист регистрации изменений 67
6. Входные данные 78
7. Выходные данные 79
8. Лист регистрации изменений 80

АННОТАЦИЯ

В данном документе приведено руководство программиста для модуля динамической адаптации количества рабочих потоков для системы фрагментированного программирования LuNA.

Функцией модуля является автоматическая адаптация количества рабочих потоков с учетом текущей загрузки системы LuNA с целью повышения производительности исполняемой программы.

Исходным языком программы является C++. Рекомендуемое средство разработки – среда разработки CLion от компании JetBrains.

Оформление программного документа «Руководство программиста» произведено по требованиям ГОСТ 19.504-79 «ЕСПД. Руководство программиста» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

1. НАЗНАЧЕНИЕ И УСЛОВИЯ ПРИМЕНЕНИЯ ПРОГРАММЫ

1.1 Назначение программы

Модуль динамической адаптации количества рабочих потоков является частью исполнительной системы LuNA (система LuNA предназначена для исполнения фрагментированных программ на одном или нескольких вычислительных устройств) и предназначен для повышения производительности исполняющихся LuNA программ за счет поиска функции-аппроксимации оптимального количества рабочих потоков в реальном времени. Для этого данный модуль использует такие понятия, как полезная и полная загрузки процессора. Помимо увеличения или уменьшения количества рабочих потоков при достаточном изменении полезной загрузки, модуль также способен выполнять изменения в случайно выбранном направлении при малых изменениях полезной загрузки с целью поиска лучшего количества рабочих потоков. Периодический анализ временных меток и удаление более не актуальных из них позволяет модулю достаточно эффективно расходовать память ОЗУ. Использование примитивов синхронизации позволяет модулю организовать корректный доступ до разделяемых между рабочими потоками ресурсов.

1.2 Функции, выполняемые программой

1.2.1 Внешние функции, выполняемые программой

Функциональность модуля для пользователя включает в себя:

- активация модуля;
- изменение константных параметров, прямо влияющих на внутренние алгоритмы модуля;
- вывод состояния модуля в файл формата CSV.

Модуль также предоставляет следующий набор внешних функций, используемых самой исполнительной системой LuNA:

- вычисление и сохранение временных меток различных типов (время

работы потока исполнения, время работы процесса и общее затраченное реальное время), соответствующим функциям вызова пользовательского кода из LuNA программ;

- увеличение, уменьшение или поддержание неизменного количества рабочих потоков на основе общей и полезной загрузки процессора.

1.2.2 Внутренние функции, выполняемые программой

Модуль использует следующий набор внутренних функций:

- проверка времени, пройденного с момента последнего обновления количества рабочих потоков, и запуск процесса анализа временных меток и обновления при истечении зафиксированного периода;
- проверка времени, пройденного с момента последней проверки рабочих потоков на активность, и запуск процесса удаления неактивных рабочих потоков при истечении зафиксированного периода;
- поиск временной метки, соответствующей началу скользящего окна;
- поиск временной метки, соответствующей концу скользящего окна и одновременно концу интервала анализа временных меток;
- поиск временной метки, соответствующей началу интервала анализа временных меток;
- вычисление общей загрузки процессора на найденном интервале анализа временных меток;
- вычисление полезной загрузки процессора на найденном интервале анализа временных меток;
- удаление более не актуальных временных меток из хранилища меток.

1.3 Условия, необходимые для выполнения программы

Требования к составу и содержанию дистрибутивного носителя данных полностью соответствуют требованиям к составу и содержанию дистрибутивного носителя данных системы LuNA. Модуль разработан как часть исполнительной

системы LuNA, поэтому для использования модуля никаких дополнительных действий по установке вспомогательного ПО либо настройке не требуется.

Условия, необходимые для выполнения данного модуля, соответствуют условиям, необходимым для выполнения LuNA программ. Для возможности исполнения функций модуля необходимо указать при компиляции LuNA программы флаг компиляции *--enable-threads-controller*.

2. ХАРАКТЕРИСТИКА ПРОГРАММЫ

Исходный программный код данного модуля расположен в одном месте с исходными кодами исполнительной системы LuNA. Модуль реализован и внедрен в систему LuNA по принципу слабо связной архитектуры: он выдвигает к системе лишь требование к наличию заданных интерфейсных функций, остальные же вычисления и анализ модуль проводит самостоятельно.

В текущей реализации модуль требует от системы наличие следующих реализованных функций: добавление и удаление из пула потоков системы заданное положительное количество потоков, получение текущего размера пула потоков, а также получение флагов активности всех потоков из пула потоков системы (потоки, которые за пройденный промежуток времени фиксированной длины не выполнили ни одну задачу из очереди задач, считаются неактивными – модуль в праве их удалить для оптимизации расходуемой памяти).

Для поиска требуемых модулю для анализа временных меток используются алгоритмы с линейной сложностью по количеству задач из пула потоков. Самая трудоемкая часть модуля – это вычисление полезной загрузки процессора с одновременной проверкой актуальности временных меток.

Для синхронизации временных меток, генерируемых каждым отдельным потоком исполнения, используется структура данных дек с мьютексом. Поток, выполнивший задачу из очереди, захватывает мьютекс и сохраняет соответствующие временные метки в конец дека. Поток, выполняющий вычисление загрузки процессора, захватывает мьютекс и извлекает временные метки из начала дека. Все остальные свойства модуля примитивных типов, требующие синхронизации, используют std::atomic<?> в виде средства синхронизации.

3. ОБРАЩЕНИЕ К ПРОГРАММЕ

Обращение к модулю производится только при условии, что компиляция LuNA программы была осуществлена с флагом компиляции --enable-threads-controller. Активация модуля происходит посредством указания флага компиляции --enable-threads-controller в момент запуска LuNA программы. При активированном алгоритме в генерируемый код добавляются вызовы функций модуля.

Сохранение вычисленных временных меток осуществляется через метод update_state класса ThreadsController, отвечающего за всю работу модуля. При вызове этого метода осуществляется проверка пройденного времени после момента последнего обновления количества рабочих потоков. Также, при вызове данного метода при истечении зафиксированного на момент компиляции периода происходит обновление количества потоков одним из случайных существующих рабочих потоков. Поскольку модуль изменяет состояние пула рабочих потоков, для работы модуля необходима реализация интерфейса удаления и добавления заданного количества рабочих потоков. В текущей версии модуля данные интерфейсы реализованы в виде функций delete_thread и add_thread класса ThreadPool соответственно.

Результатом единичной работы алгоритма анализа временных меток является возможное изменение количества рабочих потоков (которое может увеличиться, уменьшится или остаться неизменным).

4. ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ

4.1 Организация используемой входной информации

Входной информацией модуля являются флаг активации модуля --enable-threads-controller (указываемый при запуске LuNA программы), период проведения анализа и последующего обновления количества потоков (в секундах), период проверки активности рабочих потоков (в секундах). Изменение данных параметров при необходимости осуществляется посредством задания новых значений константам THREADS_ACTIVITIES_CHECKING_PERIOD_SECS и UPDATING_PERIOD_SECS, находящихся в файле threads_controller.cpp исполнительной системы LuNA. Также входными данными модуля являются начальное количество рабочих потоков, минимально допустимое количество рабочих потоков и общее количество доступных ядер на вычислительном узле (все они вычисляются автоматически модулем во время вызова конструктора).

4.2 Организация используемой выходной информации

На этапе исполнения выходной информацией модуля является количество рабочих потоков, требуемых к увеличению или к удалению. При равенстве данной величины нулю количество потоков остается прежним. Величина вычисляется функцией process_load_stamps класса ThreadsController и используется пулом потоков (класс ThreadPool) для изменения количества потоков в нем.

5. СООБЩЕНИЯ

При включенной опции OUT_TO_CSV (включение осуществляется через соответствующую директиву компиляции #define в файле common.h из исходных кодов исполнительной системы LuNA) модуль в момент обновления количества рабочих потоков записывает в файл свое текущее состояние: время обновления, текущая полезная и общая загрузки процессора, текущее количество потоков, тренд обновления (рост, падение или постоянство), а также на сколько было изменено количество потоков в текущем обновлении.

При включенной опции INFO (включение осуществляется через соответствующую директиву компиляции #define в файле common.h из исходных кодов исполнительной системы LuNA) модуль в момент обновления количества рабочих потоков выводит в стандартный поток вывода для каждого нового рабочего потока и для каждого предназначенного к удалению потока его идентификатор и номер ядра, к которому этот поток был привязан.

6. ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

Таблица 5 – Лист регистрации изменении руководства программиста

Лист регистрации изменений									
Номера листов (страниц)					**Всего листов (страниц) в докум.**	**№ докумен-та**	**Входящий № сопроводит. докум. и дата**	**Подп.**	**Дата**
Изм.	**изменен-ных**	**заменен-ных**	**новых**	**аннули-рованных**					

ПРИЛОЖЕНИЕ Б

Модуль динамической адаптации количества рабочих потоков в системе LuNA

Описание программы

Листов 13

Новосибирск, 2023

Содержание

Аннотация 70
1. Общие сведения 71
1.1 Обозначение и наименование программы 71
1.2 Программное обеспечение, необходимое для функционирования программы 71
1.3 Языки программирования 71
2. Функциональное назначение 72
2.1 Назначение программы 72
2.2 Сведения о функциональных ограничениях на применение 72
3. Описание логической структуры 73
3.1 Алгоритм программы 73
3.2 Используемые методы 73
3.3 Структура программы 73
3.4 Связь программы с другими программами 75
4. Используемые технические средства 76
5. Вызов и загрузка 77
6. Входные данные 78
7. Выходные данные 79
8. Лист регистрации изменений 80

АННОТАЦИЯ

В данном документе приведено описание модуля динамической адаптации количества рабочих потоков в системе LuNA.

Для пользователя доступна опция запуска LuNA программы, активирующая модуль динамической адаптации количества потоков (--enable-threads-controller), а также возможность получения лог-файла о ходе работы модуля.

Поскольку модуль является частью исполнительной системы LuNA, для возможности его работы необходимо иметь предустановленные все программы и модули, указанные в требованиях системы LuNA.

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

1. ОБЩИЕ СВЕДЕНИЯ

1.1 Обозначение и наименование программы

Полное наименование программы – Модуль динамической адаптации количества рабочих потоков в системе LuNA.

1.2 Программное обеспечение, необходимое для функционирования программы

Для функционирования программы необходимо установить все программы, библиотеки и модули, указанные в требованиях к системе LuNA. Никакого дополнительного ПО для функционирования более не требуется.

1.3 Языки программирования

Исходным языком программы является C++ 11-го стандарта. Средство разработки – среда разработки CLion от компании JetBrains.

2. ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ

2.1 Назначение программы

Программа предназначена для динамического изменения количества рабочих потоков в процессе исполнения LuNA программы на основе анализа временных меток выполнения фрагментов кода с целью эффективного использования ресурсов вычислительного узла и получения возможного ускорения.

2.2 Сведения о функциональных ограничениях на применение

Программа не предназначена для использования на платформах, не поддерживающих систему LuNA. Поддерживающая платформа определяется наличием всех программ, указанных к требованиям к системе LuNA.

3. ОПИСАНИЕ ЛОГИЧЕСКОЙ СТРУКТУРЫ

3.1 Алгоритм программы

Программа использует такие новые характеристики, как общая загрузка процессора и полезная загрузка процессора. Раз в зафиксированный период один из рабочих потоков вычисляет данные величины на основе собранных временных меток о выполнении пользовательских функций и по заранее установленному алгоритму изменяет количество рабочих потоков (как в большую, так и в меньшую сторону). Изменение количества рабочих потоков происходит при этом так, чтобы в итоге при соответствующих показателях общей и полезной загрузок процессора достигалась большая производительность LuNA программы.

3.2 Используемые методы

Модуль динамической адаптации количества рабочих потоков осуществляет изменение количества рабочих потоков, выполняя раз в период последовательно следующие операции:

1. Определение границ скользящего окна;
2. Определение границ промежутка анализа временных меток;
3. Вычисление общей загрузки процессора;
4. Вычисление полезной загрузки процессора;
5. Очистка хранилища временных меток;
6. Применение алгоритма изменения количества потоков.

3.3 Структура программы

Структура программы состоит из одного модуля динамической адаптации количества рабочих потоков, реализованного на языке программирования C++. Каждая из операций, указанных в разделе “3.2 Используемые методы”, выделена в отдельный блок функций модуля. Исключение составляют две операции: вычисление полезной загрузки процессора и очистка хранилища временных меток объединены в одну функцию с целью оптимизации работы модуля. Набор функций

каждой операции приведен в таблице 6.

Таблица 6 – Набор функций модуля

Наименование	Описание
Определение границ скользящего окна	Функции *get_window_start_point()* и *get_window_end_point()* вычисляют левую и правую границы скользящего окна соответственно. Используются далее для вычисления полезной загрузки процессора.
Определение границ промежутка анализа временных меток	Функции *get_processing_start_point()* и *get_window_end_point()* вычисляют начало и конец промежутка анализа временных меток соответственно. Используются для вычисления как общей, так и для полезной загрузки процессора.
Вычисление общей загрузки процессора	Функция *get_cpu_load()* вычисляет общую загрузку процессора (в процентах) по переданным в нее временным меткам. Используется далее при изменении количества рабочих потоков.
Вычисление полезной загрузки процессора и очистка хранилища временных меток	Функция *process_payload()* вычисляет полезную загрузку процессора и, вместе с этим, очищает хранилище временных меток от тех, которые не попали в текущее скользящее окно.
Применение алгоритма изменения количества потоков	Функция *process_load_stamps()* применяет алгоритм изменения количества потоков по переданным в нее полезной и общей загрузок процессора. Использует реализованные интерфейсные функции *add_thread()* и *delete_thread()* пула потоков исполнительной системы LuNA. Сохраняет текущие значения загрузок процессора.

3.4 Связь программы с другими программами

Для возможности использования модуля динамической адаптации количества рабочих потоков необходимо наличия установленной системы фрагментированного программирования LuNA.

4. ИСПОЛЬЗУЕМЫЕ ТЕХНИЧЕСКИЕ СРЕДСТВА

Режим работы программы – составная часть (модуль) исполнительной системы LuNA, активирующаяся при наличии соответствующей опции компиляции LuNA программы.

Требования модуля к используемым техническим средствам соответствуют требованиям к используемым техническим средствам системы LuNA.

5. ВЫЗОВ И ЗАГРУЗКА

Запуск модуля осуществляется посредством указания опции компиляции –enable-threads-controller LuNA программы.

6. ВХОДНЫЕ ДАННЫЕ

Модуль использует временные метки с информацией о времени работы пользовательских функций для анализа и вычисления нового количества рабочих потоков. Данные временные метки передаются в модуль исполнительной системой LuNA после завершения каждой пользовательской функции.

7. ВЫХОДНЫЕ ДАННЫЕ

В каждый период обновления количества рабочих потоков модуль выдает новое количество рабочих потоков и выставляет его пулу потоков исполнительной системы LuNA посредством добавления или удаления заданного количества потоков.

Доступны в виде вывода в файл также специальные возможности по отслеживанию работы модуля (выводится следующая информация: время обновления; текущая полезная и общая загрузки процессора; текущее количество потоков; тренд обновления: рост, падение или постоянство; на сколько было изменено количество потоков в текущем обновлении). Активация данной возможности осуществляется включением директивы компиляции OUT_TO_CSV.

8. ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

Таблица 7 – Лист регистрации изменении описания программы

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц) в докум.	№ докумен-та	Входящий № сопроводит. докум. и дата	Подп.	Дата
Изм.	изменен-ных	заменен-ных	новых	аннули-рованных					