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

Чемагина Александра Сергеевича

Тема работы:

РАЗРАБОТКА СРЕДСТВ АВТОМАТИЧЕСКОГО ПОДБОРА ПАРАМЕТРОВ ПАРАЛЛЕЛЬНОЙ ПРОГРАММЫ НА ОСНОВЕ ТЕСТИРОВАНИЯ ДЛЯ ОПТИМИЗАЦИИ ЕЁ НЕФУНКЦИОНАЛЬНЫХ СВОЙСТВ

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

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

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ

Киреев С.Е./……..
(ФИО) / (подпись)
«…»……20…г.

Новосибирск, 2022

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)

«……»……2022г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Чемагину Александру Сергеевичу 18209
(фамилия, имя, отчество, номер группы)

Тема Разработка средств автоматического подбора параметров параллельной программы на основе тестирования для оптимизации её нефункциональных свойств
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 20.01.2022 №0012

Срок сдачи студентом готовой работы 20 мая 2022 г.

Исходные данные (или цель работы):

Разработка средства оптимизации параллельной программы численного моделирования.

Структурные части работы:

Обзор литературы и анализ существующих решений, разработка программы и проведение экспериментов.

Руководитель ВКР
к.т.н., доцент
доцент каф. ПВ ФИТ НГУ
Маркова В.П. /……..
(ФИО) / (подпись)

«…»……2022г.

Задание принял к исполнению
Чемагин А.С. /……..
(ФИО студента) / (подпись)

«…»……2022г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ

Киреев С.Е. /……..
(ФИО) / (подпись)

«…»……2022г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Большинство программ, призванных упростить человеку жизнь, сильно зависят от входных параметров. Чем хуже используемый набор значений параметров, тем менее эффективно (тут и далее эффективность понимается в смысле времени выполнения, расхода памяти и т.п.) будет работать программа. Из этого следует, что подбор параметров - важная часть реализации любой программы, в том числе и параллельной. На практике найти оптимальные наборы параметров возможно только перебором всех возможных комбинаций их значений, что для большинства программ невозможно за приемлемое время. Для параллельных программ эта проблема стоит наиболее остро. Параллельная программа дополнительно требует решать вопросы декомпозиции данных, логики их обработки, взаимодействия между рабочими потоками/процессами и другие.

Полный перебор для больших задач занимает большое количество времени, помимо этого он не может быть применен в случаях, если хотя бы один из параметров вещественный. Из-за этого часто параметры подбираются с помощью некоторых алгоритмов. Для многих задач подобного рода для работы используют наборы параметров, которые алгоритм посчитал лучшими. Получить понимание об оптимальности того или иного набора можно на основе тестовых запусков и их профилирования - сбора информации о разных характеристиках исполнения.

Если известны какие-либо детали реализации программы, то можно подобрать алгоритм, который будет их учитывать, что значительно упростит процесс поиска. Но периодически приходится иметь дело с программами, реализация которых может быть скрыта или сильно запутанна, из-за этого приходится взаимодействовать с ней методом чёрного ящика - подавать входные данные и получать некоторый выход без возможности оценить промежуточные

состояния и значения. Алгоритмы для такого взаимодействия должны быть гибкими и отказоустойчивыми для случаев, когда программа не может правильно работать при определенных входных значениях.

Цель данной работы — разработать средство настройки параметров реализации параллельной программы на основе профилирования. Программа должна рассматриваться как "черный ящик".

Задачи:

1. Провести анализ нескольких методов оптимизации с точки зрения цели работы.
2. Реализовать прототип системы автоматизированного подбора параметров, использующий рассмотренные методы оптимизации.
3. Сделать выводы о эффективности работы реализованных методов оптимизации для достижения поставленной цели.

Новизна данной работы работы состоит в применении методов подбора параметров для оптимизации параллельных численных программ для суперкомпьютеров.

Практическая ценность состоит в возможности использования данного средства в системах автоматического конструирования параллельных программ численного моделирования, таких как LuNA [1].

Структура и объем работы. Работа состоит из введения, 4 глав и заключения. В первой главе рассматриваются существующие средства оптимизации программ и алгоритмы оптимизации функций, который могут быть использованы в нашем средстве. Во второй главе рассматриваются свойственные предметной области особенности и с их учётом выбираются алгоритмы для реализации в разрабатываемом средстве. В третьей главе описаны используемые технологии, разработанный прототип средства и реализация алгоритмов. Четвертая глава посвящена проведению экспериментов

и выбору алгоритма, который может быть использован в итоговом средстве поиска.

1 Постановка проблемы и обзор существующих решений

1.1 Определение параметров

Прежде, чем рассматривать алгоритмы нахождения реализационных параметров, стоит понять, что именно входит в это понятие. Под параметром в программировании подразумевается принятый подпрограммой (функцией или другой запущенной в ходе работы программой) аргумент. В нашем случае реализацией функции является тестируемая программа.

Обычно программа принимает на вход два вида параметров: реализационные и функциональные. Реализационные параметры программы – это те параметры, которые влияют только на нефункциональные свойства исполнения (потребление памяти, время счёта, …), но не влияют на результат вычислений. Что касается непосредственно параллельных программ, большинство из них рассчитаны на работу с большими массивами данных в нескольких процессах или потоках. Принцип их разбиения может сильно повлиять на скорость работы программы. Под принципом может подразумеваться как сама логика распределения данных, так и размер участка.

Функциональными параметрами называют те параметры, которые характеризуют выполняемую программой функцию. Иными словами, они влияют непосредственно на результат.

Помимо значений, которые программа получает на вход, на поведение программы сильное влияние оказывают используемый компилятор и ключи компиляции. Статья [2] посвящена сравнению нескольких компиляторов для языков C и C++. Изучив её, можно сделать вывод, что правильно выбранный компилятор может положительно повлиять на эффективность работы программы. Помимо этого, могут существовать компиляторы, предназначенные для определённых систем или процессоров. Одними из таких являются Intel oneAPI DPC++/C++ Compiler [3] и x86 Open64 Compiler Suite [4], которые, по

заверениям разработчиков, будут хорошо работать на Intel и AMD процессорах соответственно.

Для анализа степени влияния ключей компиляции на эффективность работы программы рассмотрим компилятор MinGW(gcc) [5] для языков C и C++. Он имеет несколько ключей компиляции, влияющих на оптимизацию. Самыми часто используемыми из них являются -O0, -O1, -O2/-O и -O3. Программа, полученная с использованием ключа -O3 в большинстве случаев будет работать быстрее, чем при использовании других. Данный результат вызван тем, что за этими ключами на самом деле скрывается целый набор ключей, которые делают уже определённые преобразования, и при -O3 используются все ключи, нацеленные на уменьшение длительности работы программы. Эти конкретные ключи можно использовать как вместе, так и по отдельности, или же исключить для данной конкретной компиляции. Другие компиляторы могут использовать те же наборы ключей, но их инструкции или реализация может отличаться.

Основываясь на этих знаниях о компиляторах и их ключах компиляции, можно сделать вывод, что они тоже являются реализационными параметрами, хоть и не такими очевидными.

Каждый параметр оказывает своё влияние на процесс исполнения программы. Помимо этого могут существовать случаи, когда при определённых значениях какого-то параметра программа не будет работать по самым разным причинам. Из-за этого для подбора параметров следует использовать алгоритмы, способные эффективно работать в такой ситуации.

1.2 Обзор методов подбора параметров

Чтобы качественно и эффективно находить параметры для программы, взаимодействие с которой происходит методом чёрного ящика, алгоритм поиска параметров должен отвечать некоторым требованиям:

1. Не накладывать на целевую функцию строгих ограничений (существование производной, неразрывность функции и подобное).
2. Эффективно работать в ситуации существования не оказывающих или оказывающих малое влияния параметров.
3. Предусматривать ситуации, когда при определенных наборах параметров значение функции не определено.

Для выполнения этих требований алгоритм должен использовать методы, отвечающие тем же требованиям хотя бы частично. Несколько таких методов рассмотрено ниже.

1.2.1 Метод полного перебора

Метод полного перебора [6] является математическим методом поиска решения исчерпыванием всех возможных вариантов. Сам метод довольно прост, мы рассматриваем абсолютно все возможные значения наборов параметров, после чего получаем полную картину поведения нашей программы и можем однозначно определить оптимальный набор значений параметров.

Данный алгоритм отвечает всем требованиям, приведённым выше. Но у него есть явный недостаток, перебор всех возможных значений требует большого количества времени и ресурсов. Использование этого метода будет целесообразно при малых диапазонах значений всех параметров, когда и количество самих параметров достаточно мало. В иных случаях стоит использовать другие методы для более эффективного нахождения подходящего набора параметров.

1.2.2 Метод покоординатного спуска

Используя метод покоординатного спуска [7] мы пошагово оптимизируем все параметры. Каждый следующий шаг такой оптимизации рассчитывается на основе фиксации всех параметров, кроме одного. Для этого одного выбранного параметра происходит отдельная оптимизация на основе одного из методов одномерной оптимизации.

У данного метода есть два ярко выраженных недостатка. Во-первых, могут существовать параметры, значения которых слабо влияет или не влияет вовсе на работу программы. Во-вторых, некоторые параметры могут сильно зависеть друг от друга, из-за чего изменение одного с фиксацией другого не даст полной картины.

Помимо этого, метод покоординатного спуска требует ещё какого-либо метода подбора параметров для одномерного поиска. Это значит, что помимо собственных недостатков, покоординатный спуск будет иметь все недостатки второго используемого метода, из-за чего в конечном итоге данный метод может не отвечать требуемым параметрам.

1.2.3 Поиск по сетке

Поиск по сетке, или по-другому метод равномерного поиска [8], является довольно простым методом поиска оптимума. Заключается он в том, что весь диапазон значений разбивается на равные (или близкие к равным) части точками деления. После этого производится анализ значений в этих точках, и выбирается диапазон (или несколько диапазонов), в которых будет продолжаться поиск. Алгоритм заканчивает свою работу, когда не может больше разбить отрезок.

Этот метод тоже не лишён недостатков. Первое, на что нужно обратить внимание - каким образом будут получены точки деления (узлы сетки). Плохой выбор узлов может может повлиять на то, что области с “хорошими” параметрами не будут рассматриваться. Данный алгоритм будет отвечать всем нужным требованиям, если определить его поведение в точках, где значение функции не определено.

1.2.4 Случайный поиск

Случайный поиск заменяет полный перебор всех комбинаций на выборку их случайным образом, тем самым оставляет нас без необходимости использовать какие-либо методы для одномерной оптимизации. Случайный поиск может превзойти поиск по решётке, особенно в случае, если только малое число параметров оказывает влияние на производительность целевой функции, представленной в нашем случае временем работы программы.

Из недостатков можно выделить то, что случайный поиск не гарантирует, что хоть одна из точек будет хорошей.

Случайный поиск имеет много реализаций [9]. Грубый случайный поиск (метод Монте-Карло) на каждом шаге выбирает случайные точки из доступного диапазона, является очень быстрым, но менее эффективным, чем другие.

Алгоритм с парной пробой делает движение в сторону от точки, расстояние движения равно единичному вектору, так же делается шаг и в обратную сторону. После анализа двух получившихся точек, алгоритм выбирает, какую из них сделать новой стартовой или остановиться. Недостатком данного является склонность ходить в некоторой окрестности хороших точек в поисках лучшей из них.

Алгоритм с возвращением при неудачном шаге на каждом своём шаге смещается от точки на некоторое значение. Если новая точка лучше

предыдущей, то на следующем шаге алгоритм начинает работу от неё, иначе возвращается к стартовой точке этого шага. Как и предыдущий алгоритм, данная реализация случайного поиска склонна ходить в некоторой окрестности хороших точек.

Любая из реализаций данного метода отвечает всем требованиям.

1.2.5 Генетические алгоритмы

Генетический алгоритм [10] - алгоритм поиска, используемый для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе. Отличительной особенностью генетического алгоритма является акцент на использование оператора «скрещивания», который производит операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе.

Алгоритм состоит из нескольких шагов. Первый - выбор “начальных представителей”, после чего на них считается функция приспособленности, насколько хорошо они решают поставленную задачу. Вторым шагом мы каким-либо способом выбираем лучших представителей (двух или больше, в зависимости от используемых далее алгоритмов). После этого производится “скрещивание”, суть которого в комбинировании родителей каким-то образом, с целью получить потомка с лучшим результатом согласно функции приспособленности. После этого алгоритм возвращается на второй шаг и так продолжается, пока алгоритм не решит остановиться, обычно это зависит от ограничений на время работы или количество поколений. Помимо этого к потомкам может применяться “мутация”, некоторое дополнительное изменение потомка, которое может дать как положительный, так и отрицательных эффект.

Недостатки тоже довольно существенны. Алгоритм склонен сходится к локальному оптимуму, а не к глобальному, особенно остро это встаёт при сложных функциях с большим количеством локальных оптимумов. Помимо этого, алгоритм требует частой оценки функцией приспособленности, что тратит много ресурсов и времени, при этом не гарантирует результат. Будет ли отвечать данный алгоритм всем требованиям зависит от его реализации.

1.2.6 Градиентный спуск

Градиентный спуск [11] - метод нахождения локального минимума или максимума функции с помощью движения вдоль градиента. Градиент - вектор, который указывает направление наибольшего возрастания некоторой функции.

Данный метод имеет большой недостаток. В случае наличия большого количества оптимумов градиентный спуск может долгое время проводить поиск в их окрестности или вовсе остановить свой поиск, рассмотрев только локальную область. Помимо этого данный метод накладывает на функцию строгое ограничение - существование градиента.

1.2.7 Баевская оптимизация

Байесовская оптимизация [12] - стратегия последовательного проектирования для глобальной оптимизации функций черного ящика, поэтому соответствует всем приведенным выше требованиям. Обычно данный метод используется для оптимизации дорогостоящих в оценке функций. Именно к такому типу относится наша функция времени от параметров. Суть метода заключается в том, что на основе полученных значений для нашей оптимизируемой функции строится другая, аппроксимирующая её функция, анализ которой менее дорогостоящий.

Хоть Баевская оптимизация и дает хорошие результаты, но имеет ряд недостатков. При большом количестве параметров время выбора тестового набора может быть сопоставимо с временем работы программы. Но большое время работы может быть и в случаях малого количества параметров, так как при получении значений аппроксимирующая функция может потребовать сильной корректировки.

1.3 Обзор средств подбора параметров

1.3.1 Система автоматизированного распараллеливания Ratio

Одним из средств для автоматической оптимизации является система для автоматизированного распараллеливания Ratio [13]. Данная система ставит перед собой цель как можно более эффективно распределить данные между рабочими потоками, учитывая скорость работы потока, памяти и сетевых соединений нужных для обмена данными. Для эффективной работы данная система определяет схему выполнения программы и распределение работы между потоками, производит настройку выполнения параллельной программы. Процессы оптимизации в данной системе производятся на основе оценки времени работы параллельных инструкций, связанных с взаимодействием потоков между собой. Параллельные программы, которые могут быть оптимизированы данной системой, реализуют механизм взаимодействия master/worker. Механизм работы данной системы заключается в распределении ролей между потоками. Один из них получает роль master, его задача во время выполнения программы распределять данные между остальными worker потоками. Помимо этого master поток производит сбор обработанных данных. Worker процесс выполняет три функции. Первая - получение данных. Вторая - их обработка. И третья - пересылка обработанных данных master потоку. Вторая функция не подлежит оптимизации в рамках подбора параметров. Система

Ratio оценивает время этих взаимодействий, чтобы минимизировать время простоя, пока master занят отправкой/получением данных от другого worker процесса. Оценка времени в данной системе производится сразу тремя способами. Первый - предсказания времени работы, основываясь на анализе программного кода и характеристик используемой системы. Второй - профилирование кода. Третий - указание пользователем ключевых переменных и времени выполнения участков программы.

Данная реализация имеет недостатки. Предсказание времени, основываясь на профилировании работы на меньшем объёме данных, может скрыть важные нюансы, что может повлиять на увеличение времени простоя процессов из-за увеличения объемов передаваемых данных. Предсказание времени человеком довольно сложно назвать точным, но при этом оно требует от человека определенных знаний, чтобы предсказание не было случайным.

1.3.2 Библиотека Intel MPI

Библиотека Intel MPI [14] тоже имеет возможность оптимизировать связанные с её использованием функции. Данная библиотека реализует MPI — интерфейс передачи сообщений между процессами, выполняющими одну задачу. Он предназначен, в первую очередь, для систем с распределенной памятью. Данная библиотека сама автоматически определяет выбор алгоритма для коллективных операций и требуемые для него параметры.

1.3.3 Система Active Harmony

Ещё одним оптимизирующим средством является система Active Harmony [15], являющееся средством подбора параметров, которое осуществляет подбор во время исполнения программы. Эта система, по словам разработчиков, вносит в код минимальные изменения. Получая на вход тип и список параметров,

система производит запуски с целью подобрать лучшую комбинацию. Active Harmony учитывает важную деталь, что использование времени как основного параметра оптимизации — не самый лучший вариант. Active Harmony больше сосредотачивается на поведении непосредственно системы при выполнении функций (эффективность работы процессов, используемая память и тому подобное). Помимо этого, данная оптимизационная система рассматривает вопрос наличия локальных оптимумов. Также, Active Harmony не ставит своей целью поиск усреднённого решения, которое бы показывало себя достаточно хорошо при разных условиях. Данная система ищет оптимальные или наиболее близкие к ним параметры при данных ей в момент исполнения условиях. В своей работе данное средство использует метод на основе метода Нелдера-Мида.

Active Harmony хоть и решает аналогичную нашей задачу, ища оптимальные параметры на данной машине, но важным моментом является то, что программа всё-таки изменяет код. Любое изменение кода в параллельных программах влечёт за собой изменение в работе этой самой программы, поэтому хотелось бы избежать подобного подхода в своей работе.

1.3.4 Вывод

Хоть проблема оптимизации параметров давно извесна, но на данный момент было найдено мало средств, способных решать задачу оптимизации программы чёрного ящика. Большинство из них сосредотачивается на определенном аспекте (оптимизация памяти, оптимизация взаимодействий потоков и другое) или оптимизации программ конкретной направленности.

Из рассмотренных средств только система Active Harmony может быть использована для решения задачи оптимизации программы чёрного ящика, но

изменение кода, которое производит система, является её существенным недостатком.

2 Основные особенности и выбор алгоритмов

2.1 Основные особенности оптимизируемых программ

Процесс оптимизации любой программы изначально является не самой простой задачей для любого программиста, даже при знании всех аспектов её работы. Но задача оптимизации параллельной программы в разы сложнее оптимизации однопоточной. У параллельных программ есть свои особенности, которые могут влиять на выбор способа оптимизации параметров. Рассмотрим эти особенности ниже.

2.1.1 Декомпозиция данных

Декомпозиция [16] - это разбиение задачи на более простые, относительно независимые друг от друга подзадачи. Декомпозиция может быть проведена по заданиям, по данным или по информационным потокам.

Декомпозиция по задачам или функциональная декомпозиция разбивает работу приложения на подзадачи таким образом, чтобы каждая подзадача была связана с отдельной функцией. Но не все операции могут выполняться параллельно, и при разбиении это нужно учитывать.

Декомпозиция по информационным потокам служит для задач, когда работа одного потока зависит от результата выполнения другого. Например, во время чтения из файла поток — обработчик считанных данных не может начать работу, пока поток-читатель не считает определенный объем данных.

При декомпозиции по данным каждая подзадача выполняет одни и те же действия, но с разными данными. Во многих задачах, параллельных по данным, существует несколько способов их разделения. Например, задача матричного умножения допускает разделение по строчкам – каждый поток обрабатывает одну или несколько строк матрицы, по столбцам – каждый поток обрабатывает один или несколько столбцов, а также по блокам заданного размера. Как можно

видеть из примера, одни и те же данные могут быть распределены разными способами, каждый из которых будет по-своему влиять на эффективность работы.

В большинстве случаев одна программа реализует только один тип разбиения данных. В качестве параметрической величины обычно рассматривается именно размер данных, передаваемый на шаге работы программы. Неэффективное разбиение может пагубно сказаться на эффективности работы. К примеру, разбиение данных на слишком малые объёмы может привести к тому, что основной операцией программы станет не обработка данных, а обмен их между рабочими потоками. В таком случае многие потоки могут долгое время оставаться в состоянии ожидания, из-за чего будут тратить предоставленные им системой ресурсы неэффективно. Другой проблемой может быть неравномерная нагрузка процессов, из-за чего полезные мощности уже закончивших свою работу будут простаивать в ожидании завершения работы более нагруженных процессов.

Чтобы избежать таких проблем, нужно выбрать такое разбиение, при котором все рабочие процессы будут работать максимально эффективно. К несчастью, выбрать такой параметр может быть довольно сложно. Помимо того, что оптимальный размер данных сильно зависит от системы, на которой запущена программа, не все данные могут быть равномерно разбиты, из-за чего нельзя будет избежать простоя процессов, можно лишь попытаться сократить это время. Для этого потребуется провести тестовые запуски при разных значениях параметра.

2.2.1 Зависимость от системы

Эффективность работы любой программы сильно зависит от системы, на которой она была запущена. Большое значение, особенно для параллельных

программ, имеет процессор или процессоры, используемые в системе. Именно их параметры ограничивают количество допустимых потоков и определяют эффективность работы процессов.

Характеристики памяти системы сильно влияют на скорость работы программы. Память любой ЭВМ обычно подчиняется иерархии [17]:

1. Внутренняя память процессора (регистры организованные в регистровый файл и кэш процессора).
2. ОЗУ системы и вспомогательных карт памяти.
3. Вторичная память компьютера - жесткие диски и твердотельные накопители, не требующие длительных (секунды и больше) действий для начала получения данных.
4. Третичная память - память, требующая переключения носителей или ощутимого времени (больше секунды) для получения информации.

Положение типа памяти в иерархии зависит от скорости обращения к ней, но чемы выше память в иерархии, тем меньше её размер. К примеру, обращение к регистрам процессора происходит за время сопоставимое с одним тактом, но размер такой памяти редко превосходит несколько тысяч байт. L1 уровень кэша процессора в это же время имеет размер в несколько десятков килобайт, но время обращения к нему уже сопоставимо с несколькими тактами. Другие уровни обладают большим размером памяти и заметно большим временем обращения, количество этих уровней тоже зависит от процессора.

Часто для программ параметры ответственные за память (размер массивов) подаются в виде реализационных параметров. Как можно понять, для эффективной работы лучше всего использовать память, находящуюся выше в иерархии, тогда это сильно уменьшит затраты на обращение в память.

2.2.3 Недетерминированность исполнения

Для параллельных программ свойственна недетерминированность исполнения. При каждом новом исполнении потоки будут работать с разной скоростью, поэтому одни и те же события будут происходить в разное время. На то, в каком порядке будут работать процессы, влияет планировщик процессов системы, и повлиять на его работу довольно сложно. Помимо этого загрузка системы посторонними процессами оказывает сильное влияние, так как они тоже учитываются планировщиком. Также посторонние процессы претендуют на такие ресурсы, как память, доступ к разделяемым ресурсам и сеть, что замедляет работы других программ.

Из-за этого единственный запуск программы может не дать полной картины, поэтому лучше провести несколько запусков и получить усредненный результат, чтобы оценить работу программы при разном возможном поведении планировщика на текущей машине при текущей нагрузке.

2.3 Выбор алгоритмов подбора параметров

Не все приведенные в обзоре алгоритмы могут быть использованы для работы с программой “чёрным ящиком”. К примеру, метод градиентного спуска требует лишние вычисления функции для вычисления производной в точке, что может занять очень много времени. Помимо этого, оптимизируемая функция может иметь большое количество локальных экстремумов, поэтому нам требуются методы глобальной оптимизации.

Метод полного перебора является простым и самым точным методом поиска значений, но его существенным недостатком является время работы. Метод полного перебора в данной работе не рассматривается как возможный метод поиска для итогового алгоритма, а нужен для получения полной картины для оценки итогов работы других методов.

Грубый случайный поиск не подвержен проблеме поиска значений в области локального оптимума, благодаря чему может дать хороший результат. Помимо этого, данный алгоритм легко справляется со случаями, когда часть параметров слабо влияют на исполнение программы.

Поиск по сетке является простым алгоритмом, стоит сравнить результаты его работы с другими. Стоит рассмотреть несколько его реализаций, так как данный алгоритм сильно зависит от принципа разбиения области.

Байесовская оптимизация была специально создана для анализа функций, взаимодействие с которыми производится методом чёрного ящика, поэтому обязательно стоит её рассмотреть.

3 Реализация прототипа средства оптимизации

3.1 Реализация прототипа оптимизирующего средства

Для создания алгоритма, способного работать с программами, представленными “чёрным ящиком”, требуется рассмотреть несколько методов поиска и сравнить их эффективность по отношению друг к другу. Для этих целей на языке Python была написана программа, реализующая несколько методов.

В начале работы программа предлагает выбрать конфигурационный файл из списка файлов хранящихся в специальной папке. Пример конфигурационного файла представлен в приложении А. К параметрам, задающимся в конфигурационном файле, относятся:

1. Путь до программы или скрипта. Требуется указать полный путь до программы.
2. Путь до файла, куда будут сохранены итоги всех запусков. Указывается полный путь до файла формата txt.
3. Строка, которую программа будет искать в выводе программы.
4. Номер алгоритма.
5. Данные для алгоритмов, а именно число, строка или массив. Правильность этих данных должен обеспечить программист.
6. Строка, которую программа будет искать в выводе программы.
7. Ограничение по времени для оптимизации. Значение 0 - ограничений не накладывается.
8. Количество запусков при одних условиях (для получения среднего более точного результата).
9. Количество параметров.
10. Список параметров.

В списке параметров для каждого их них задаётся тип (1 - целое число, 2 - строка, 3 - вещественное число) и диапазон значений для числовых параметров и массив строк для строковых параметров.

Конфигурационный файл проверяется на валидность (содержит ли он все нужные значения, и совпадает ли реальный размер списка параметров с указанным). Дальше на основе этого файла создаётся объект со всеми нужными для запусков значениями и передаётся в класс с выбранным методом.

Так как в языке Python нет интерфейсов, то вместо этого любой класс метода поиска имеет одинаковую сигнатуру, а именно конструктор без параметров и основной рабочий метод, требующий объект со всеми данными для запусков. Все итоги запусков сохраняются в специальный словарь, сопоставляющий набор параметров и массив с полученными значениями. Логика каждого метода поиска параметров своя, но имеет схожую структуру.

Прежде всего внутри класса поиска создается вспомогательный класс для выбора значений параметров. Все такие классы имеют одинаковую сигнатуру. В конструктор передаются данные для алгоритма, в рабочий метод передаётся диапазон параметров. Метод сам определяет тип параметра и выдаёт для него значения, основываясь на этой особенности. По итогу своей работы предоставляет массив со значениями для параметра, которые нужно проверить на данном шаге. Такой поиск ведётся для всех параметров.

После подготовки значений для параметров, массив значений и данные для запусков передаются в другой класс, который запускает тестируемую программу с полученными наборами значений в отдельном потоке, после чего анализирует вывод. Если в выводе не было найдено строка указанная в конфигурационном файле, то результатом запуска считается значение MAXINT. После запусков массив полученных значений возвращается в класс метода, там он сохраняется в специальный словарь, после чего алгоритм анализирует

результаты и, если считает нужным, продолжает свою работу с этапа поиска новых значений для параметров. Новые диапазоны выбирает сам алгоритм на основе параметров.

В случае ограничения по времени после каждого запуска проверяется, не работает ли программа дольше заданного времени. Если да, то запуски прекращаются и в класс метода поиска передаётся информация только об уже проведённых запусках. Далее класс алгоритма тоже сверяется со временем и останавливает свою работу, проводя лишь итоговый анализ (выбор лучшего результата). После этого результаты всех запусков записываются в текстовый файл а в командную строку выводится лучший найденный набор и время работы при его использовании.

Отдельно стоит сказать про строковые параметры. Работа со строковыми параметрами строится следующим образом. Во время работы массив строк превращается в диапазон от 1 до длины массива строк. Алгоритмы работают с ним как с целочисленным значением, но при запусках извлекается значение, имеющее нужный номер в массиве строк.

Ниже представлены используемые в работе алгоритмы с описанием реализации.

3.2 Используемые языки и технологии

3.2.1 Python 3

Тестирующая программа была реализована на языке Python 3 [18]. Данный язык был выбран в силу своего удобства и наличия большого количество вспомогательных библиотек в открытом доступе.

3.2.2 C++

Тестируемые программы были написаны на языке программирования C++ [19]. Данный язык был выбран, так как является распространенным, и на нём реализовано множество библиотек для параллельного программирования.

3.2.3 Subprocess

Для запуска тестируемых программ в нашей тестирующей Python программе использована библиотека subprocess [20]. Она используется для порождения новых процессов, соединения с потоками стандартного ввода, стандартного вывода, стандартного вывода сообщений об ошибках и получения кодов возврата от этих процессов. В нашей реализации в этих процессах запускается тестируемая программа.

3.2.4 Itertools

Библиотека Itertools [21] языка Python позволяет упростить работу с данными. Предоставляемые библиотекой итераторы позволяют проще анализировать, комбинировать и инициализировать данные. Помимо этого они эффективно используют память и обладают большой скоростью работы.

3.2.5 Байесовская оптимизация

Для байесовской оптимизации использована библиотека Bayesian Optimization [22]. Оная является простой в использовании и не накладывает ограничений на целевую функцию.

3.3 Реализация методов

3.3.1 Метод полного перебора

Реализация полного перебора в данной задаче довольно проста. Мы получаем все возможные комбинации значений параметров, после чего рассматриваем результат работы нашей тестируемой программы для всех наборов. В данной работе данный метод будет рассматриваться только в задачах, где в качестве параметров не используются вещественные числа, так как в таком случае его реализация не является возможной.

3.3.2 Грубый случайный поиск (метод Монте-Карло)

В нашей реализации алгоритм проводит запуски со случайными наборами значений до тех пор, пока за определенное количество запусков лучший результат не изменялся. Это число по умолчанию задано равным пятидесяти, но оно может быть изменено пользователем в конфигурационном файле.

3.3.3 Поиск по сетке

В нашей реализации алгоритм будет выбирать значения по всему диапазону с некоторым начальным шагом, максимальное значение которого задаётся пользователем в конфигурационном файле, после чего для каждого параметра выбирается свой начальный шаг. Начальный шаг получается уменьшением максимального шага в несколько раз до тех пор, пока шаг не станет приемлемым.

К примеру, пусть задано два диапазона: от 1 до 10000 и от 1 до 5. Если начальный шаг равен 10, то для второго диапазона будет выбрано только значение 1, что сразу сильно уменьшит количество вариантов. Предположим, что на каждом шаге алгоритма шаг выбора параметров уменьшается в 5 раз.

Тогда для первого интервала будут выбраны все значения из диапазона с шагом 10, а для второго все значения с шагом 2.

В нашем случае на каждом шаге алгоритма шаг поиска будет уменьшаться в 2 раза.

3.3.4 Байесовская оптимизация

Для реализации данного метода была использована библиотечная функция [14]. Для работы алгоритм принимает в качестве параметров два числа. Первое число определяет, сколько запусков будут проводится с использованием алгоритмов случайного поиска для составления начального понимания о программе. Второе число определяет, сколько запусков будет проводится с целью составить более точную аппроксимирующую функцию и найти лучший набор параметров. В нашем алгоритме два этих числа равны. По умолчанию их значение равно тридцати. В случае надобности, это число можно увеличить в конфигурационном файле.

4 Проведение тестов

4.1 Первый тест. Перемножение матриц

4.1.1 Описание программы и цели тестирования

Рассмотрим классическую задачу перемножения двух матриц на C++, использующую MPI (матрицы 1300 на 1300). У нас будет 4 параметра. В качестве первого параметра у нас будут выступать ключи компиляции: -O1, -O2, -O3, -Os и -O0. Вторым параметром будет целое число от 1 до 8, количество процессов. Третий и четвёртый параметр не будут оказывать влияние на процесс исполнения и будут задаваться диапазоном от 1 до 100, они нужны чтобы диапазон поиска не был мал. Сама программа приведена в приложении Б.

Первое, что мы рассмотрим, это влияние недетерминированности исполнения программы на результаты работы рассматриваемых алгоритмов.

4.1.2 Ожидаемые результаты

Для данной программы был запущен полный перебор. По его итогу лучшим набором значений параметров является (-O3, 4, 2, 3). Последние два значения могут быть любыми, так как они не влияют на нефункциональные свойства.

Заранее стоит сказать, что тесты проводились на машине, которая способна выделить только 4 параллельных потока. При значениях от 5 до 8 результаты будут аналогичны, как при 4.

4.1.3 Базовый тест

Прежде всего был рассмотрен случай, когда для каждого рассматриваемого набора данных программа была запущена один раз. Для поиска по сетке были рассмотрены два разных начальных шага 30 и 20. Также рассмотрена ситуация, когда ключи в массиве идут в порядке “-O1, -O2, -O3,

-Os, -O0” (случай 1), “-O1, -O0, -Os, -O2, -O3” (случай 2) и “-O3, -O0, -Os, -O2, -O1” (случай 3). Для двух других алгоритмов проверки для трёх этих случаев были проведены и не дали примечательных данных.

По итогу данного теста байесовская оптимизация показала лучшее время работы. Обусловлено это было тем, что она проводит фиксированное количество запусков. Из-за особенностей параллельных программ, один и тот же запуск мог давать разное итоговое время, из-за чего случайный поиск в нашей реализации в стремлении найти более оптимальные значения проводил существенно большее количество запусков.

Что касается поиска по сетке, его большее по сравнению с остальными время обусловлено деталями реализации. Если сделать начальный шаг больше или увеличить его сокращение на каждый шаг алгоритма, то время уменьшается. Но даже если отсеять проблему большего времени, данный алгоритм не всегда находит оптимальные значения из-за плохого начального шага. Помимо этого, как и ожидалось, поиск по сетке плохо работает со строковыми значениями, так как при перестановке строк его результаты улучшились. Результаты тестирования представлены в таблице 1.

Таблица 1 - результаты оптимизации первой программы без подсчёта среднего

Алгоритм	Результаты	Время работы (секунды)	Вывод
Случайный поиск	(-O3, 4, 54, 57), (-O3, 4, 34, 81), (-O3, 4, 41, 84), (-O3, 4, 69, 61), (-O3, 4, 42, 84)	92, 207,133, 270, 164	Результат всегда верный

Алгоритм	Результаты	Время работы (секунды)	Вывод
Байесовская оптимизация	(-O3, 4, 91, 53), (-O3, 4, 84, 72), (-O3, 4, 33, 51), (-O3, 4, 27, 89), (-O3, 4, 64, 5)	147, 132, 161, 138, 142	Результат всегда верный
Поиск по сетке (начальный шаг 30, случай 1)	(-O3, 4, 53, 54)	1938	Результат верный
Поиск по сетке (начальный шаг 30, случай 2)	(-Os, 4, 42, 58)	1894	Результат неверный
Поиск по сетке (начальный шаг 30, случай 3)	(-O3, 4, 27, 68)	1167	Результат верный
Поиск по сетке (начальный шаг 20, случай 1)	(-O3, 4, 3, 17)	2776	Результат верный
Поиск по сетке (начальный шаг 20, случай 2)	(-O3, 4, 78, 62)	2438	Результат верный
Поиск по сетке (начальный шаг 20, случай 3)	(-O3, 4, 40, 86)	2512	Результат верный

4.1.4 Тест с подсчетом среднего

Были проведены те же запуски, но теперь для каждого набора полученных параметров программа запускалась три раза для получения среднего значения, так как при запуске при одном наборе значений разница между результатами достигала десятой секунды при среднем времени исполнения меньше двух. Для нашей байесовской оптимизации это значит, что программа будет запускаться не 60 раз, а 180, то есть можно было заранее предположить, что время работы вырастет примерно в 3 раза. Для случайного поиска всё не так однозначно.

Результаты поиска по сетке здесь не приводятся, так как он дал те же результаты, что и в предыдущем тесте.

Как видно из данного эксперимента, случайный поиск лучше работает со средними значениями, чем для случаев единичного запуска (среднее увеличение времени работы меньше, чем в 3 раза). С учётом того, что многократные запуски увеличивают точность, в чём их основное предназначение, возможно, что для более сложных программ случайный поиск будет лучше себя показывать. В это же время, иногда, из-за рассмотрения плохих значений, случайный поиск дольше работает, что было заметно в этом эксперименте.

По итогу этого теста можно сделать вывод, что байесовская оптимизация работает более предсказуемое время, что может стать плюсом в определённых ситуациях. Все результаты представлены в таблице 2.

Таблица 2 - результаты оптимизации первой программы с подсчетом среднего

Алгоритм	Результаты	Время работы (секунды)	Вывод
Случайный поиск	(-O3, 4, 14, 1), (-O3, 5, 23, 78), (-O3, 5, 41, 84), (-O3, 4, 34, 2), (-O3, 4, 45, 67)	620, 302, 479, 475, 728	Результат всегда верный
Байесовская оптимизация	(-O3, 4, 71, 5), (-O3, 6, 48, 79), (-O3, 7, 21, 84), (-O3, 5, 87, 96), (-O3, 7, 55, 69)	401, 438, 492, 369, 457	Результат всегда верный

4.1.5 Итог

После анализа полученных ранее данных становятся понятны многие вещи:

1. Поиск по сетке - не самый лучший алгоритм для задач типа “чёрный ящик”. Данный метод очень зависим от начального шага, который сложно выбрать, не имея знаний о программе.
2. Случайный поиск не так хорошо показывает себя без повторных запусков при одинаковых параметрах.
3. При наличии повторных запусков случайный поиск наоборот может показать себя лучше, чем байесовская оптимизация. Возможно это зависит от уровня шума программы, разброса времени исполнения появившегося из-за недетерминированности исполнения.

4.2 Второй тест. Перемножение матриц блочным методом

4.2.1 Описание программы и цели тестирования

Рассмотрим ещё одну реализацию перемножения двух матриц 1300 на 1300. На этот раз параллельность у нас будет обеспечена с помощью OpenMP. Помимо этого будем использовать блочное умножение. В качестве параметров у нас будут: ключи компиляции, размеры блоков и количество потоков. В качестве ключей у нас будут: “-O1”, “-O2”, “-O3”, “-Os”,”-Ofast”, “-O1 -march=native”, “-O2 -march=native”, “-O3 -march=native”, “-Os -march=native”, “-Ofast -march=native”. Ключ “-march=native” [23] говорит компилятору, что все оптимизации нужно проводить с учетом особенностей системы, поэтому данный ключ, в теории, увеличивает производительность. Размеры блоков для каждой из трех матриц будут задаваться тремя параметрами в диапазоне от 2 до 60. Число потоков будет задаваться параметром в диапазоне от 1 до 8. Сама программа приведена в приложении В.

Проводя данный тест, мы преследуем несколько целей:

1. Убедиться в неэффективности поиска по сетке. В прошлом тесте он показывал себя плохо, но быть может были выбраны неудачные для него условия.
2. Сравнить эффективность работы случайного поиска и байесовской оптимизации на реальной задаче с большим диапазоном возможных значений при сопоставимом времени работы.

4.2.2 Ожидаемые результаты

В отличии от первой серии тестов, здесь определить лучшие параметры так просто не получится. Методу полного перебора для получения данных понадобилось несколько дней, что ещё раз подтверждает его неэффективность на реальных задачах. Итоги его работы были неоднозначны. По ряду причин

точно выделить один оптимальный набор набор не получилось. Причины этого будут описаны дальше. С помощью данного метода мы смогли лишь определить оптимальные области значений параметров или их комбинаций.

Для ключей компиляции лучшими значениями были - “-O2 -march=native”, “-O3 -march=native”, “-Ofast -march=native”, “-O3” и “-Ofast”. Остальные ключи, при прочих равных, уступали. Из этого можно сделать вывод, что ключ “-march=native” действительно даёт преимущество в оптимизации, а разница между ключами “-O3” и “-Ofast” для данной задачи несущественна.

Для количества потоков, как и ожидалось, лучшее значение было от 4. Как и в первом тесте, вычисления будут производиться на машине с 4 ядерным процессором, которая способна выделить только 4 параллельных потока, при значениях от 5 до 8 результаты будут аналогичны, как при 4.

Значения параметров, ответственных за размеры блоков, стоит рассмотреть вместе, так как они претендуют на один и тот же ресурс - память. Для оптимальной работы все нужные для расчетов данные должны помещаться, при возможности, в процессорную память как можно более высокого уровня. Это значит, что размеры блоков не должны быть большими. Для машины, на которой проводились тесты, размер уровне кэш памяти процессора представлен ниже. Лучше всего использовать L1d уровень памяти, в данном случае нам доступно 32 КБайт, или если учесть размер одного значения типа double в C++ равное 8 байт, то мы можем разместить на этом уровне 4096 значений. Но на практике нам будет доступно меньше места. Если использовать слишком малые блоки, то основной операцией станет не вычисление, а организация циклов, эту теорию подтвердили и тестовые запуски. При размерах всех блоков меньше 15 программа работала медленнее, чем при больших значениях. Что касается больших размеров блоков, программа давала меньшую производительность при

значении размера всех блоков больше 35, или хотя бы одного блока больше 40. Из этого всего мы можем сделать вывод, что оптимальными значениями этих параметров можно считать диапазон от 15 до 30 с небольшими смещениями (к примеру набор 7, 25, 38 тоже будет оптимальным).

4.2.3 Базовый тест

Как и в прошлом тесте, сначала был рассмотрен случай, когда для каждого рассматриваемого набора данных программа была запущена один раз. Для поиска по сетке были рассмотрены три разных начальных шага 30, 20 и 10.

По итогу проведенных запусков поиск по сетке снова плохо показал себя, его время работы было значительно больше, а найденные наборы параметров не всегда можно было назвать оптимальными.

Байесовская оптимизация и случайный поиск показали себя хорошо, при этом оба алгоритма работали за сравнительно одинаковое время, несмотря на разницу в реализации. Это хорошо видно по результатам, представленным в таблице 3.

Таблица 3 - результаты оптимизации второй программы без подсчёта среднего

Алгоритм	Найденные значения	Время работы(секунды)	Вывод
Поиск по сетке (начальный шаг 10)	(-O2, 24, 27, 31, 4)	2064	Результат неверный
Поиск по сетке (начальный шаг 20)	(-Ofast, 12, 5, 16, 4)	1467	Результат можно считать верным

Алгоритм	Найденные значения	Время работы(секунды)	Вывод
Поиск по сетке (начальный шаг 30)	(-O3 -march=native, 2, 31, 31, 8)	829	Результат верный
Случайный поиск	(-O2 -march=native, 11, 23, 16, 4) (-O3 -march=native, 23, 35, 34, 4) (-O3, 19, 38, 32, 7) (-Ofast, 9, 36, 15, 4) (-O3 -march=native, 5, 35, 31, 4)	50, 78, 76, 60, 125	Результат верный
Байесовская оптимизация	(-O2 -march=native, 7, 38, 22, 5), (-O2 -march=native, 5, 37, 29, 8), (-O3 -march=native, 7, 38, 22, 4), (-O3 -march=native, 7, 38, 22, 4), (-O2 -march=native, 7, 38, 22, 4),	52, 59, 57, 72, 61	Результат верный

4.2.4 Тест с подсчетом среднего

Снова будем проводить три запуска и считать среднее время.

В отличие от предыдущего теста, в этом поиск по сетке дал разные результаты при единичном запуске и при подсчёте среднего. Но его время работы было сильно больше, чем у двух других алгоритмов. При этом, как и раньше, не всегда удавалось найти оптимальные параметры. Этот момент более подробно показан в таблице 4.

Случайный поиск и байесовская оптимизация снова хорошо себя показали и работали сравнимое время.

Таблица 4 - результаты оптимизации второй программы с подсчётом среднего

Алгоритм	Найденные значения	Время работы (секунды)	Вывод
Поиск по сетке (начальный шаг 10)	(-Ofast, 7, 28, 28, 4)	8083	Результат верный
Поиск по сетке (начальный шаг 20)	(-O1 -march=native, 13, 17, 22, 4),	4277	Результат неверным
Поиск по сетке (начальный шаг 30)	(-O3 -march=native, 5, 30, 28, 4)	3684	Результат верный
Алгоритм	Найденные значения	Время работы (секунды)	Вывод
Случайный поиск	(-O3 -march=native, 24, 29, 38, 7), (-Ofast, 35, 14, 23, 8), (-O3, 17, 31, 19, 8), (-O2 -march=native, 6, 33, 16, 7), (-O3 -march=native, 32, 16, 26, 7)	290, 171, 135, 235, 163	Результат верный
Байесовская оптимизация	(-Ofast -march=native, 14, 35, 37, 8), (-O2 -march=native, 7, 38, 22, 4), (-O3 -march=native, 21, 38, 26, 4), (-O2 -march=native, 10, 39, 26, 7), (-Ofast -march=native, 9, 30, 31, 8)	184, 176, 177, 189, 172	Результат верный

4.2.5 Итог

По итогу данного теста стало очевидно, что поиск по сетке, по крайней мере, в данной реализации является неэффективным. Ему не всегда удается найти подходящие наборы, при этом его время работы заметно больше. Дальнейшее его рассмотрение нецелесообразно.

В данном тесте байесовская оптимизация и случайный поиск показали себя так же, как и в предыдущем.

4.3 Специальное тестирование для оценки влияния шума

4.3.1 Описание программы и цели тестирования

В предыдущих тестах в случае получении среднего значения и при наличии существенного шума случайный поиск был сопоставим, а иногда даже превосходил байесовскую оптимизацию. Но диапазон значений, оказывающих значительное влияние на производительность тестируемой программы, был мал. Следует увеличить диапазон значимых значений, чтобы ни один из двух этих алгоритмов не мог получить оптимальные параметры полным перебором.

Помимо этого значительное влияние на процесс поиска параметров мог оказать шум. Стоит рассмотреть разные уровни шума, и как они повлияют на решения методов.

Используем модельную программу. Она будет получать на вход два параметра. Первый - целое число от 1 до 1000 (параметр A). Вторым параметром выступает вещественное число от 10 до 20 (параметр B). Помимо этого в программе присутствует шум (N) - случайное число от 0 до n, n задается внутри самой программы, в нашем случае это будет 5, 15 и 30 для каждого теста ниже соответственно. Программа считает значение функции F = 40 + |100 -A%200| + B + N. Дополнительно если B>=15, оно становится отрицательным - случай сложной зависимости параметров. И, если A равно 300, то B после

предыдущего преобразования увеличивается в два раза. В конце своей работы программа ожидает столько времени, какой был получен ответ. Полный код программы приведен в приложении Г.

Эта программа хорошо нам подходит, так как имеет однозначное оптимальное значение (A = 300, B = 20), большой диапазон значений для целого числа и вещественный диапазон с бесконечным количеством значений. На данной программе мы сможем оценить влияние разного уровня шума на работу алгоритмов. Проведём три теста с максимальным шумом 1, 10 и 25. Тесты будут проведены с подсчётом среднего значения трёх запусков.

Помимо этого поставим алгоритмам ограничение на время исполнения в 30000 секунд. Это нужно, так как в противном случае случайный поиск будет работать значительно дольше, чем байесовская оптимизация с фиксированным количеством запусков. В таком случае данный тест позволит нам оценить точность алгоритмов при одинаковом времени работы.

4.3.2 Тест с минимальным уровнем шума

По итогу тестов байесовская оптимизация показала себя лучше, что хорошо видно в таблице 5, предложенные ею наборы параметров были ближе к точкам оптимума, чем у случайного поиска. Вместе с тем, случайному поиску удалось найти оптимальную точку для всей программы, но скорее всего это просто случайность, возможная в данном методе.

Из-за того, что шум мал по сравнению со средним временем работы программы, закономерности, выявленные в первом тесте при подсчёте среднего, не проявились.

Таблица 5 - результаты оптимизации специальной программы с малым уровнем шума

Алгоритм	Найденные значения	Время работы(секунды)
Случайный поиск	(508, 17.4507875612940), (699, 18.36070315182939), (901, 19.58880028540661), (902, 16.13184164761533), (300, 15.62070029527079)	30340, 30081,30621, 30304,28076
Байесовская оптимизация	(501, 19.9134011295167), (502, 20),	24906, 25265, 24925, 24915, 24772

4.3.3 Тест со средним уровнем шума

Случайный поиск в данном эксперименте показал себя хуже, чем в предыдущем. Это заставляет усомниться в высказанном предположении, что случайный поиск дает лучше результаты для зашумленных программ. Более подробные результаты показаны в таблице 6.

Таблица 6 - результаты оптимизации специальной программы со средним уровнем шума

Алгоритм	Найденные значения	Время работы(секунды)
Случайный поиск	(500, 19.5687406699145), (105, 17.0904529567840), (292, 15.7229156481985), (694, 18.8364800333046), (100, 17.2313750337073)	27929, 30133, 30466, 30027,30051
Байесовская оптимизация	(502, 20), (501, 19.9134011295167),	24734, 24729, 25331, 24981

4.3.4 Тест с высоким уровнем шума

По итогу запусков, приведенных в таблице 7, даже при большом уровне шума случайный поиск уступает в своей точности байесовской оптимизации.

Таблица 7 - результаты оптимизации специальной программы с высоким уровнем шума

Алгоритм	Найденные значения	Время работы(секунды)
Случайный поиск	(901, 17.7479926991529), (101, 15.1423044737220), (106, 18.6891536568414), (900, 17.9972754883852), (495, 17.5835605509233)	30243, 28803, 30412, 30360, 30569
Байесовская оптимизация	(502, 20), (501, 19.9134011295167),	24934, 29824, 26345, 27687

4.3.5 Итог

Одной из причин, почему полученные в предыдущих тестах закономерности не повторились, может быть большой диапазон возможного времени исполнения программы (от 10 до 270 секунд).

На основе полученных в этой серии результатов можно сделать определённые выводы:

1. Случайный поиск не превосходит байесовскую оптимизацию ни в точности, ни в скорости.
2. Байесовская оптимизация лучше справляется с зашумленными программами, по крайней мере при большом диапазоне возможного времени исполнения тестируемой программы.

Что касается интересных значений набора параметров для байесовской оптимизации (они совпадали при разных запусках и при разном уровне шума), скорее всего это связано с устройством библиотеки, реализующей этот метод.

ЗАКЛЮЧЕНИЕ

Был проведен анализ методов, которые могли быть применимы для оптимизации параллельных программ.

В рамках работы было разработано программное средство, способное использовать разные методы поиска параметров для программ, взаимодействие с которыми проводятся методом чёрного ящика. На практических задачах методы были протестированы.

По итогу проведённого тестирования лучше всего себя показала байесовская оптимизация. Из этого можно сделать вывод, что она лучше подходит для поиска параметров.

Разработанное средство может быть использовано для поиска параметров для параллельных программ, что было продемонстрировано на тестах.

В рамках последующей работы над данным средством планируется улучшение взаимодействия между программой и пользователем, а именно добавления возможности составить конфигурационный файл, используя само средство (диалоговое окно), и добавление возможности прервать выполнение программы до окончания времени выполнения, заданного в конфигурационном файле.

Помимо этого планируется рассмотреть более сложные методы поиска, такие как генетические алгоритмы, уже упомянутые в этой работе.

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

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

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

_____ _____

ФИО студента Подпись студента

« ____ »_____20 __г. (заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Перепёлкин В. А. Язык и система LuNA для автоматического конструирования программ численного моделирования // Проблемы информатики. - 2020. - №1. - С. 66-74.
2. A Performance-Based Comparison of C/C++ Compilers : сайт - URL: https://colfaxresearch.com/ (дата обращения: 22.11.2021). Текст - электронный
3. Intel® oneAPI DPC++/C++ Compiler : сайт - URL: https://www.intel.ru/content/www/ru/ru/homepage.html (дата обращения: 22.11.2021). Текст - электронный
4. x86 Open64 Compiler Suite : сайт - URL: https://developer.amd.com/ (дата обращения: 22.11.2021). Текст - электронный
5. Mingw-w64 : сайт - URL: https://www.mingw-w64.org/ (дата обращения: 22.11.2021). Текст - электронный
6. Полный перебор : сайт - URL: https://dic.academic.ru/dic.nsf/ruwiki/201612 (дата обращения: 11.12.2021). Текст - электронный
7. Метод покоординатного спуска : сайт - URL: https://studref.com/ (дата обращения: 11.12.2021). Текст - электронный
8. М. Ю. Черняк, М. С. Эльберг Методы оптимизации в технике. - Красноярск: Издательский дом «Красноярский университет», 2017. - 95 с.
9. И. Ф. Минаков Сравнительный анализ некоторых методов случайного поиска // Известия Самарского научного центра РАН. - 1999. - №2. - С. 286-293.
10. Т.В. Панченко Генетические алгоритмы. - Астрахань: Издательский дом «Астраханский университет», 2007. - 88 с.

11. Метод градиентного спуска : сайт - URL: http://www.machinelearning.ru/wiki/index.php?title=Метод_градиентного_спуска (дата обращения: 11.12.2021). Текст - электронный
12. Байесовский подход : сайт - URL: https://dyakonov.org/2018/07/30/байесовский-подход/ (дата обращения: 11.12.2021). Текст - электронный
13. Телегин П.Н Настройка выполнения параллельных программ //Настройка выполнения параллельных программ. - 2012. - №4. - С. 25-30
14. MPI Tuning : сайт - URL: https://www.intel.ru/content/www/ru/ru/homepage.html (дата обращения: 17.10.2021). Текст - электронный
15. Jeffrey K. Hollingsworth. Parallel Parameter Tuning for Applications with Performance Variability / Jeffrey K. Hollingsworth, Vahid Tabatabaee, Ananta Tiwari // IEEE : Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, 2005. - 1-59593-061-2. - 10.1109/SC.2005.52.
16. Проблемы разработки параллельных приложений : сайт - URL: https://intuit.ru/studies/courses/5938/1074/lecture/16445 (дата обращения: 21.03.2022). Текст - электронный
17. Иерархия памяти в персональном компьютере : сайт - URL: https://studopedia.ru/7_79938_ierarhiya-pamyati-v-personalnom-kompyutere.html (дата обращения: 21.03.2022). Текст - электронный
18. Python 3.8.0 : сайт - URL: https://www.python.org/downloads/release/python-380/ (дата обращения: 17.10.2021). Текст - электронный
19. Discussion about Standard C++ : сайт - URL: https://isocpp.org/ (дата обращения: 17.10.2021). Текст - электронный

20. Subprocess management : сайт - URL: https://docs.python.org/3/library/subprocess.html (дата обращения: 17.10.2021). Текст - электронный
21. Functions creating iterators for efficient looping : сайт - URL: https://docs.python.org/3/library/itertools.html (дата обращения: 11.12.2021). Текст - электронный
22. Bayesian Optimization : сайт - URL: https://github.com/fmfn/BayesianOptimization (дата обращения: 15.03.2022). Текст - электронный
23. 3.19.60 x86 Options : сайт - URL: https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html (дата обращения: 20.04.2022). Текст - электронный

ПРИЛОЖЕНИЕ А

Пример конфигурационного файла


```
{
  "program_path": "some_script_path",
  "save_path": "some_save_path/file.txt",
  "searching_word": "time",
  "find_type": 3,
  "find_condition": 30,
  "time_limit": 0,
  "trails": 1,
  "data_set_len": 2,
  "data_set": [
    {
      "type": 2,
      "range": [
        "-O0","-O1","-O2","-O3","-Os"
      ]
    },
    {
      "type": 1,
      "range": [
        1,
        100
      ]
    }
  ]
}
```

ПРИЛОЖЕНИЕ Б

Программа перемножения матриц

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<chrono>
#include<time.h>
 int main(int argc,char *argv[])
{
        if(argc<3){
                return 0;
        }
   double start, stop;
   int maxa = atoi(argv[1]);
   int maxb = atoi(argv[1]);
   int i, j, k, l;
   int *a, *b, *c, *buffer, *ans;
   int size = 1300;
   int rank, numprocs, line;
   auto t0 = std::chrono::high_resolution_clock::now();
   MPI_Init(NULL,NULL);//MPI Initialize
     MPI_Comm_rank(MPI_COMM_WORLD,&rank);// Получить текущий номер процесса
     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);// Получить количество процессов
```

line = size/numprocs;// Делим данные на блоки (количество процессов), и основной процесс также должен обрабатывать данные

```
a = (int*)malloc(sizeof(int)*size*size);
b = (int*)malloc(sizeof(int)*size*size);
c = (int*)malloc(sizeof(int)*size*size);
```

// Размер кеша больше или равен размеру обрабатываемых данных, когда он больше, чем фактическая часть данных

```
buffer = (int*)malloc(sizeof(int)*size*line);// Размер пакета данных
```

ans = (int*)malloc(sizeof(int)*size*line);// Сохраняем результат расчета блока данных

// Основной процесс присваивает матрице начальное значение и передает матрицу N каждому процессу, а матрицу M передает каждому процессу в группах.

```
if (rank==0)
{
    for(i=0;i<size;i++) // Чтение данных
        for(j=0;j<size;j++)
            a[i*size+j]=(i+j)%maxa;
    for(i=0;i<size;i++)
        for(j=0;j<size;j++)
            b[i*size+j]=(i+j)%maxb;
    // Отправить матрицу N другим подчиненным процессам
    for (i=1;i<numprocs;i++)
    {
            MPI_Send(b,size*size,MPI_INT,i,0,MPI_COMM_WORLD);
    }
```

// Отправляем каждую строку а каждому подчиненному процессу по очереди

```
for (l=1; l<numprocs; l++)
{
   MPI_Send(a+(l-1)*line*size,size*line,MPI_INT,l,1,MPI_COMM_WORLD);
}
// Получаем результат, рассчитанный по процессу
for (k=1;k<numprocs;k++)
{

MPI_Recv(ans,line*size,MPI_INT,k,3,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   // Передаем результат в массив c
   for (i=0;i<line;i++)
   {
      for (j=0;j<size;j++)
      {
         c[((k-1)*line+i)*size+j] = ans[i*size+j];
      }
   }
}
// Рассчитать оставшиеся данные
for (i=(numprocs-1)*line;i<size;i++)
{
   for (j=0;j<size;j++)
   {
      int temp=0;
```

```
            for (k=0;k<size;k++)
                temp += a[i*size+k]*b[k*size+j];
            c[i*size+j] = temp;
        }
    }
    // Результат теста
    // Статистика по времени
   auto t1 = std::chrono::high_resolution_clock::now();
      double dt_normal = 1.0e-3 *
std::chrono::duration_cast<std::chrono::milliseconds>(t1 - t0).count();
      printf("Time:%lf\n",dt_normal);
    free(a);
    free(b);
    free(c);
    free(buffer);
    free(ans);
  }
     // Другие процессы получают данные и после вычисления результата
отправляют их в основной процесс
  else
  {
      //printf("Here \n");
    // Получаем широковещательные данные (матрица b)

MPI_Recv(b,size*size,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNOR
E);
```

```
MPI_Recv(buffer,size*line,MPI_INT,0,1,MPI_COMM_WORLD,MPI_STATUS_IG
NORE);
        // Рассчитать результат продукта и отправить результат в основной процесс
        for (i=0;i<line;i++)
        {
            for (j=0;j<size;j++)
            {
                int temp=0;
                for(k=0;k<size;k++)
                    temp += buffer[i*size+k]*b[k*size+j];
                ans[i*size+j]=temp;
            }
        }
        // Отправить результат расчета в основной процесс
        MPI_Send(ans,line*size,MPI_INT,0,3,MPI_COMM_WORLD);
    }
    MPI_Finalize();//Конец
    return 0;
}
```

ПРИЛОЖЕНИЕ В

Перемножение матриц блочным методом

```
#include<algorithm> // for std::min()
#include<cstdio>    // for printf
#include<chrono>    // for time measurement
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void my_dgemm5(int n,int si,int sj,int sk, int num,const double *a,const double
*b,double *c) {
   #pragma omp parallel for num_threads(num) collapse(2)
   for (int bi = 0; bi < n; bi += si)
      for (int bj = 0; bj < n; bj += sk)
         for (int bk = 0; bk < n; bk += sj)
            for (int i = bi; i < std::min(n,bi+si); i++)
               for (int k = bj; k < std::min(n,bj+sj); k++)
                  for (int j = bk; j < std::min(n,bk+sk); j++)
                     c[i*n + j] += a[i*n + k] * b[k*n + j];
}

int main(int argc, char* argv[]) {
   int n = 1300;
   double *c = new double [n*n];
   double *a = new double [n*n];
   double *b = new double [n*n];
   int si = atoi(argv[1]);
```

```
    int sj = atoi(argv[2]);
    int sk = atoi(argv[3]);
    int num = atoi(argv[4]);
    auto t2 = std::chrono::high_resolution_clock::now();
    my_dgemm5(n, si, sj,sk, num, a,b,c);
    auto t3 = std::chrono::high_resolution_clock::now();
    double dt_blocked = 1.0e-3 * std::chrono::duration_cast<std::chrono::milliseconds>(t3 - t2).count();

    printf("Time:%lf\n",dt_blocked);
}
```

ПРИЛОЖЕНИЕ Г

Модельная программа

```
#include <stdio.h>
#include <stdlib.h>
#include<cstdio>
#include <unistd.h> // для Unix систем
#include <ctime>

int main(int argc, char* argv[]) {
    int Noise = 25; // уровень шума (1, 10, 25)

    if (argc!=3){
        return 0;
    }
    srand( time(NULL) );
    int a = atoi(argv[1]);
    float b = atof(argv[2]);
    if (b>=15){
        b=-b;
    }
    if (a==300){
        b*=2;
    }
    a = 100 - a%200;
    if(a<0){
        a*=-1;
    }
```

```
    a+=40;
    float ret = rand();
    ret = ret/RAND_MAX;
    ret+=a+b;

    printf("Time:%lf\n",a+b+ret);
    return 0;

}
```