МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки)

Компьютерные технологии и математическое моделирование
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему Реализация и анализ эффективности многопоточного алгоритма сжатия

Обучающийся М.Ю. Пересыпкин
(Инициалы Фамилия) (личная подпись)

Руководитель к.ф.-м. н., доцент, Г.А. Тырыгина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант О.А. Головач
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2023

Аннотация

Тема выпускной квалификационной работы - "Реализация и анализ эффективности многопоточного алгоритма сжатия".

Актуальность работы заключается в повсеместности вопроса оптимального хранения и передачи данных в сфере обработки информации. В то время как современные алгоритмы достигают высокого уровня сжатия за счёт сложных методов кодировании данных, не менее важно рассмотреть вопрос повышения скорости их работы.

Цель работы - анализ эффективности многопоточной реализации алгоритма сжатия.

Объектом исследования бакалаврской работы являются алгоритмы сжатия информация.

Предмет исследования бакалаврской работы является параллельный комплексный алгоритм сжатия.

Структура бакалаврской работы представлена введением, тремя главами, заключением, списком литературы, и составляет 45 страниц текста, содержит 18 рисунков, 5 формул, 1 таблицу, и 20 источников, из которых 19 на иностранном языке.

В первой главе описывается общая теория в сфере алгоритмов сжатия информации и эффективности алгоритмов.

Во второй главе происходит разработка конкретного комплексного алгоритма сжатия, его параллелизация, и описание правил проведения анализа.

В третьей главе показаны стандартная и многопоточная реализации разработанного алгоритма на языке программирования Rust, а также реализация средств тестирования и оценки реализованных алгоритмов. В конце проведены эксперименты для и анализа полученного повышения эффективности.

Abstract

The topic of the graduation work is "Implementation and performance analysis of a multithreaded compression algorithm".

Relevancy of this work stems from the problem of optimizing storage and transmission of data. While modern compression algorithms reach high levels of compression with complex methods of encoding data, of no less importance is the question of optimizing their performance.

The goal of this research is to analyze the effectiveness of a multithreaded implementation of a compression algorithm.

The object of this research is data compression algorithms.

The subject of this research is parallelized complex compression algorithm.

The graduation work consists of an explanatory note on 45 pages, 18 figures, 5 formulas, 1 table and 20 references, of which 19 are from foreign sources.

The first chapter of the thesis explores the general theory of data compression and algorithm efficiency.

The second chapter of the thesis develops a complex compression algorithm, consisting of multiple sequenced compression algorithms, and analyzes ways to improve and measure its computational efficiency.

The third chapter of the thesis implements the developed algorithms in the Rust programming language, as well as algorithms for testing and benchmarking them. In the end experiments are conducted to measure the gained efficiency.

Overall, the results suggest that creation of an effective and relevant algorithm is a multi-stage process, that depends not only on a proper theoretical planning, but also a sane choice of tools and methods during actual development.

Содержание

Введение

В связи с постоянным процессом внедрения цифровых технологий во все сферы жизни и отрасли производства, огромное количество информации было перенесено в электронный вид, и сейчас большая часть начинает свое существование в электронном виде. По окончанию процессов генерации и обработки, информация обычно сохраняется в том формате, который максимально упрощает ее дальнейшее применение. Такой формат вовсе не обязательно будет являться максимально компактным. С расширением масштаба хранимых данных, лишний объем будет постепенно накапливаться, и вызывать новые проблемы, как в процессе хранения, так и в процессе загрузки и передачи информации.

Для предупреждения данной проблемы применяются алгоритмы сжатия информации. Цель алгоритма сжатия - представить информацию в виде меньшего количества битов, чем оригинальное представление [11]. В основе алгоритмов лежит тщательный анализ данных на обнаружение и сокращение избыточности - повторяющихся блоков данных. Но в то время как дополнительная сложность анализа позволяет повысить сокращение избыточности, это несёт за собой дополнительные вычислительные затраты. Поэтому в данной работе будет рассмотрен вопрос повышения эффективности классических алгоритмов сжатия.

Вопрос оптимизации процесса сжатия по любым из показателей актуален в настоящее время, и научная сфера имеет поддержку со стороны крупных компаний. В пример можно привести Global Data Compression Competition (GDCC) – всемирное соревнование по инновации и оптимизации в сфере сжатии информации [4].

Задачи бакалаврской работы:

– описать теоретические основы в сфере оптимизации алгоритмов сжатия информации;

– разработать эффективный алгоритм сжатия;
– реализовать, протестировать и проанализировать разработанный алгоритм сжатия.

Объектом исследования бакалаврской работы являются алгоритмы сжатия информация.

Предмет исследования бакалаврской работы является многопоточный комплексный алгоритм сжатия.

Цель работы - анализ эффективности многопоточной реализации алгоритма сжатия.

Бакалаврская работа содержит три главы.

В первой главе описывается общая теория в сжатия информации, их классификация и общая теория в сфере повышения вычислительной эффективности алгоритмов.

Во второй главе происходит разработка комплексного алгоритма сжатия, предлагается многопоточная реализация алгоритма, и ставятся цели анализа.

В третьей главе определяется выбор технической составляющей реализации, показаны стандартная и многопоточная реализации разработанного алгоритма сжатия на языке программирования Rust, а также реализация средств оценки реализованных алгоритмов и результаты анализа.

1 Теоретические аспекты повышения эффективности алгоритмов сжатия

1.1 Основная теория и классификация алгоритмов сжатия

Сжатие информации - процесс преобразования данных, используя меньшее количество бит, чем в исходном представлении. Это позволяет снизить затраты таких критических системных ресурсов, как место хранения и пропускную способность каналов передач.

Дать оценку эффективности алгоритма сжатия можно по нескольким параметрам. Самым важным параметром является коэффициент сжатия. Его можно описать формулой (1):

$$compression\ ratio = \frac{uncompressed}{compressed} \quad (1)$$

, где $uncompressed$ - количество битов до сжатия;

$compressed$ - количество битов после сжатия.

Таким образом, если, к примеру, коэффициент сжатия будет равен 2-м, значит алгоритм смог в два раза сжать исходный файл.

Алгоритмы сжатия можно разделить на две категории.

Самой распространенной категорией являются алгоритмы сжатия без потерь. Сжатые таким образом файлы могут быть позднее восстановлены в форму, идентичную начальной. Такой метод подходит для хранения текстовых, бинарных, и сильно структурированных файлов (таблиц, записи базы данных), где любые изменения могут привести к потере смысла, или, в случае исполняемых файлов, нарушения целостности и невозможности применения.

В 1948 Клод Элвуд Шеннон в своей научной работе "Математическая теория коммуникации" писал, что существует фундаментальный предел для

алгоритмов сжатия без потерь [17]. Этот предел, названный энтропийной скоростью H, зависит от статистического поведения исходных данных. Можно, каким либо алгоритмом сжатия без потерь, сжать данные с коэффициентов сжатия близким к H, но получить больший коэффициент математически невозможно.

Информационное содержание (в битах) каждого символа алфавита можно описать формулой (2):

$$h(a_i) = \log_2 \frac{1}{w_i} \tag{2}$$

, где w_i - вероятность появления символа.

Тогда общая энтропия для алфавита A будет равна взвешенной сумме всех символов по формуле (3):

$$H(A) = \sum_{w_i>0} w_i h(a_i) = \sum_{w_i>0} w_i \log_2 \frac{1}{w_i} \tag{3}$$

Для упрощения формулы, символы с нулевым шансом появления просто отбрасываются.

В то же время, Шеннон разработал теорию сжатия с потерями, которое более известна как теория соотношения коэффициент-искажение. При сжатия с потерями, восстановленные данные могут не полностью соответствовать оригинальным. Наоборот, допустимо некоторое количество искажения D. Тогда для исходных данных и определенном коэффициенте искажения существует функция R(D), которая называется функция коэффициент-искажение, где, если D - допустимое количество искажения, тогда R(D) - наилучший возможный уровень сжатия.

Это теория проложила путь категории алгоритмов сжатия с потерями. Такие алгоритмы находят своё применение в медиа сферах (аудио, видео, и

изображениях), где, в идеальном случае, потеря данных будет либо минимальна, либо вовсе незаметна для человеческого восприятия. В отличии от текстовых или бинарных файлов, медиа файлы не требуют, чтобы восстановленный вариант был идентичен оригиналу, особенно если потерянные данные не играют критической роли. Ещё одно применение методы сжатия с потерями часто находят в системах сжатия на лету.

В данной работе будут рассматриваться алгоритмы из первой категории - алгоритмы сжатия без потерь. Подробнее разберем их историю и вариации.

1.2 Методы сжатия информации без потерь

С развитием методов сжатия формировались различные подходы к анализу и конвертации данных.

Самый простой подход - кодирование длин серий (run-length encoding). Он заключается в замене групп повторяющихся символов на один соответствующий символ и число повторений [15].

Сам по себе такой метод кодирования имеет ограниченную эффективность, так как не проводит глубокий анализ данных, а опирается лишь на одно предположение - последовательно повторение символов. Полностью лексикографически отсортированные наборы данных будут сжаты с высоким коэффициентом, но в реальных информационных системах данные не являются настолько предсказуемыми. Ручная сортировка данных не является доступной альтернативной, так как они потеряют свой смысл. При полном отсутствии повторяющихся групп символов в входном файле метод кодирования длин серий имеет нулевую эффективность.

Более совершенный метод кодирования заключается в формировании в процессе обработки файла словаря, содержащего последовательности символов определенной максимальной величины, которые затем кодируются

в кодовые фразы заданной длины. Алгоритмы, реализующие такой подход, называются словарными алгоритмами. Анализ данных на повторения групп символов позволяет таким методам избежать главный недостаток кодирования длин серий. Данный метод хорошо справляется с непредсказуемостью данных, и применяется в популярных современных программах для компрессии, таких как ZIP [16].

Другой подход работает с входными данными посимвольно. Компьютеры обычно кодируют символы с использование таблицы ASCII, которая задаёт 8-битное значение каждому символу. Одинаковый размер символов позволяет легко отделить их друг от друга при чтении, и последовательность битов для каждого символа закреплена и универсальна.

Но практически в любом наборе данных, одни символы будут встречаться чаще, чем другие. Проведя статистический эксперимент и подтвердив это свойство, можно отказаться от единой длины кода, и присваивать более частым символам более короткие коды. Задача разделения символов решается соблюдением при генерации кодов свойства префиксного кода. Определение префиксного кода - если в код входит слово a, то для любой непустой строки b слова ab в коде не существует [10].

Так как в основе получения высокого коэффициента данным подходом лежит проведение статистического анализа, он носит название статистической компрессии. Процесс работы статистической компрессии состоит из двух главных шагов:

- формирование статистической модели - модель описывает вероятность появления каждого символа при последовательном чтении входного файла;
- кодирование исходных данных - алгоритм кодировки, применяя рассчитанную модель, генерирует таблицу кодов, и создает сжатый файл.

Процессы моделирования многообразны и имеют большую историю. Стремление описать поведение данных с наименьшими отклонениями приводит к росту сложности и размера модели [18]. В сфере алгоритмов сжатия это является критическим недостатком. Одно из ключевых преимуществ методов сжатия без потерь - возможность восстановить файл в его оригинальное состояние. Для того, чтобы восстановить файл, сжатый аналитическим методом, необходимо сначала восстановить статистическую модель, по которой кодировались данные. Восстановление, в случае со сложным, многоитеративным построением модели, потребует огромные затраты на этапе распаковки файла. Если скорость является критическим параметром в системе, то шаг восстановления можно пропустить, если архиватор запишет статистическую модель в выходной файл. Очевидно, это нанесет удар по итоговому коэффициенту сжатия, тем самым аннулируя преимущества применения сложной статистической модели.

Самые простые статистические модели очень малы. Таблицу подсчета количеств появления каждого символа можно уместить всего лишь в 256 байт, если после подсчета сжать диапазон количеств в промежуток от 0 до 255, включительно. Это не будет критической величиной, за исключением случаев с крайне малыми исходными файлами, или с очень специфическими наборами данных, к примеру миллиону одинаковых символов. Такой подход совершает лишь один проход по исходному файлу, что, с одной стороны, является крайне быстрой операций, но с другой стороны, имеет ограниченную эффективность. Чтобы добиться более высоких коэффициентов сжатия таким подходом, алгоритмы будут несколько раз проходить по файлу, в поисках дополнительных закономерностей. В результате растет вычислительная сложность, и размер модели, которую, после формирования, потребуется записать в файл.

Существует другой подход, где модель генерируется адаптивно, во время первого и единственного прохода по файлу [8]. В таком случае нет

необходимости записи модели в выходной файл, так как алгоритм распаковки сам постепенно восстановит модель тем же методом, которым она была собрана. Являясь хорошим решением как в плане в вычислительной эффективности, так и в эффективности сжатия, адаптивные методы серьёзно ограничивают работу следующего за ними этапа кодирования.

Еще одним подходом к повышению уровня сжатия является предобработка. Алгоритмы предобработки сами по себе не являются алгоритмами сжатия, так как после конвертации файл не становится меньшего размера. Их цель - обратимая конвертация данных в формат, более подходящий для эффективной работы последующих алгоритмов сжатия. Примером конвертации может быть расстановка близлежащих символов в группы, имеющие больше статистических закономерностей между собой. Известным алгоритмом предобработки, работающим с расстановкой символов, является преобразование Барроуза-Уилера. Другим методом предобработки является "Перемещение к началу", которое заменяет символы на индексы, соответствующие давности последнего появления символа.

Так как эти преобразования обратимые, никаких лишних данных в выходной файл не записывается, и единственным минусом остаются лишь дополнительные затраты на их выполнение. Поэтому высокая эффективность здесь также является весомым фактором. Подробнее разберем сферу оптимизации алгоритмов.

1.3 Теория эффективности алгоритмов

Эффективность алгоритма - свойство, относящееся к количеству вычислительных ресурсов, используемых алгоритмом. Если вычислительные затраты (потребление вычислительных ресурсов) алгоритма находятся в каком-либо допустимо пределе, алгоритм считается эффективным [3].

К базовым вычислительным ресурсам относятся:

- время выполнения;
- размер памяти - максимальное количество памяти, которое может потребовать алгоритм в любой момент выполнения;
- пропускная способность используемых каналов.

Скорость выполнения является ключевым параметром в оценке эффективности алгоритма.

Одним из подходов к повышению скорости выполнения является параллелизация. Параллелизация заключается в разбиение задачи на несколько независимых частей для их одновременного выполнения. Некоторые алгоритмы можно разделить на более узкие, независимые задачи.

Минимальное использование оперативной памяти необходимо при применении алгоритмов во встроенных системах, а также при одновременной работе многих процессов сжатия [20]. Также нужно помнить, что в многопоточных алгоритмах ожидаемое использование памяти вырастает в количество раз, соответствующее количеству одновременно выполняемых потоках. Большая часть оптимизация использования памяти происходит на этапе реализации алгоритма, при выборе оптимальных для решения задачи структур данных. Не менее важно аккуратное обращение с жизненными циклами объектов, так как их создание и очистка являются дорогими операциями в типичной системе.

Оптимизация использования пропускной способности каналов передачи позволяет повысить пропускную способность сервера и ускорить обратный отклик. Для минимизации объема передаваемых данных по каналу активно применяются алгоритмы сжатия. Другой стороной оптимизации является сокращение периодов неактивности каналов. Проблема заключается в том, что сам по себе этот процесс является блокирующим - пока данные не будут прочитаны полностью (или в заданном количестве), процесс обработки простаивает. Разрешение этой проблемы будет предложено далее, в главе 1.4.

Реальное повышение эффективности алгоритмов (или вариаций реализации одного алгоритма) измеряется проведением сравнительного анализа. При проведении сравнительного анализа необходимо избегать вариаций в среде тестирования, которые могут привести к нарушению целостности анализа. К минимальным требованиям можно отнести:

- неизменность физических и программных компонентов;
- минимальная сторонняя нагрузка на систему;
- неизменность параметров, влияющих на сборку программы (если работа алгоритмов с разными параметрами не являются частью анализа);

Многие алгоритмы не являются в полной мере статическими, и зачастую имеют различные параметры, которые тем или иным образом влияют на последовательность и подход к обработке данных. Современные компиляторы также имеют большое количество параметров, которые способны оказать большое влияние на работу программы, как в положительную сторону, так и в негативную. Поэтому проведение экспериментов необходимо для оптимизации финальной сборки, которая может являться критическими компонентами какой-либо востребованной архитектуры.

Параллельные алгоритмы - одна из таких сфер алгоритмов, где результативность реализаций в большей степени зависит от подобранных параметров и сбалансированного распределения вычислительного процесса. Разберем теорию разработки параллельных алгоритмов.

1.4 Теория параллелизации алгоритмов

Параллельная обработка - процесс выполнения нескольких задач одновременно на нескольких процессорных потоках [19].

Существует несколько парадигм, которые позволяют реализовать параллельные вычисления на вычислительных устройствах. Формирование различных подходов тесно связано с развитием вычислительной техники, особенно в плане обращения с оперативной памятью. Определим две из них:

- Message Passing Interface (MPI, интерфейс передачи сообщений) - реализация параллельного программирования для кластеров, суперкомпьютеров, и вычислительных машин с разделенной памятью; позволяет нескольким машинам работать над одной задачей путем стандартизированного протокола обмена сообщениями и данными [13];
- Многопоточность - ведущий поток порождает некоторое количество дополнительных потоков, и система распределяет задачу между ними; подход предназначен для вычислительных машин с общей памятью; среда выполнения сама назначает потоки доступным процессорам; процессоры работают на общей памяти [9];

Для эксперимента, проводимого в данной работе, будем использовать вторую парадигму - распределение обработки между несколькими потоками на одной устройстве.

Разберем методы разработки многопоточных алгоритмов на основе существующих последовательных алгоритмов.

Первым шагом является проведение теоретического анализа для определения участков, для которых применение параллелизации может быть допустимо. Теоретический анализ заключается в обнаружении повторяющихся логических цепочек, выполняющихся независимо друг от друга. В первую очередь это однотипная обработка большого количества данных, где использование результатов обработки предыдущих частей для обработки следующих не является ключевым свойством. Параллельное выполнение обработки независимых частей файла приносит почти линейное повышение эффективности относительно количества задействованных

потоков. Теоретический анализ комплексных алгоритмов происходит путем разделения их на подзадачи.

Второй шаг разработки параллельного алгоритма - практический анализ рабочей программы, сконструированной на основе последовательного алгоритма. Путем проведения профилирования отдельных компонентов программы можно определить самые тяжелые участки кода, и исследовать возможности их параллелизации, или, в случае, если это невозможно, исследовать возможность замены их на другие алгоритмы, более подходящие к параллельным вычислениям.

Еще один параметр, на который следует обратить внимание - эффективность использования доступных вычислительных ресурсов системы. Типичный процесс обработки данных состоит из трех этапов - чтение данных с какого-либо физического источника, их обработка, и запись/передача преобразованных данных.

В последовательном алгоритме, одновременно будет выполняться лишь одно из этих действий. Применение многопоточности позволяет разделить их на три параллельных процесса, и осуществлять их совместную работу через два буфера - один для чтения, другой для записи. Поток обработки будет постепенно забирать данные из первого буфера, обрабатывать их, и передавать во второй буфер. Тем временем поток чтения будет своевременно пополнять буфер, а поток записи будет отправлять обработанные данные по месту финального назначения.

Данный паттерн многопоточного программирования носит название "производитель-потребитель". Визуальное представление такой системы представлено на рисунке 1:

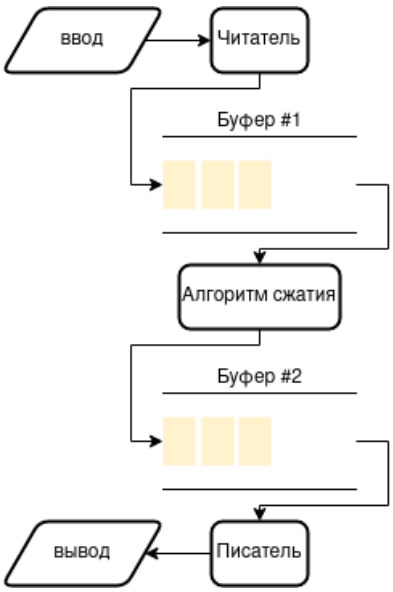


Рисунок 1 – Паттерн «Производитель-Потребитель»

Визуализация повышения эффективности представлена на рисунке 2:

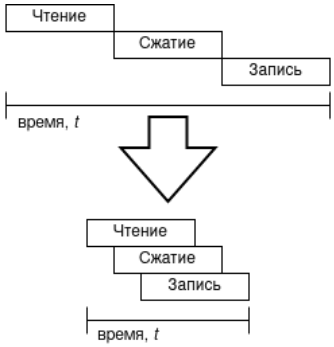


Рисунок 2 – Эффект применения параллельных вычислений

Получив три отдельных процесса, проанализируем их. Параллельное чтение из разных мест устройства хранения, в отличии от последовательного чтения большими блоками, имеет сильную зависимость от архитектуры устройства хранения, поэтому не приносит однозначного повышения эффективности. Запись может происходить лишь в последовательной форме, поэтому также не является предметом параллелизации.

Так как процесс обработки может быть достаточно сложным, могут возникнуть ситуации, когда буфер прочитанных данных будет переполнен, и

процесс обработки не будет успевать их использовать. Создание нескольких потоков обработки минимизирует данный риск, позволяя последовательным процессам чтения и записи постоянно быть в активном состоянии. Количество потоков, являющееся оптимальным для конкретной задачи, зависит от количества разделяемой работы. Общим правилом является минимизация порождения потоков и перераспределения работы между ними, так как противное приводит к большим затратам вычислительных ресурсов со стороны системы.

Для успешной совместной работы нескольких потоков применяется синхронизация. В задаче производителя-потребителя поток обработок, чтобы получить свою часть данных, использует блокировку, запрещая остальным потокам использовать буфер, пока он не прочитает свою долю. В отдельном процессе обработки, когда несколько потоков поровну разделяют между собой данные, могут применяться барьеры, препятствующие продолжению работы, пока все потоки не завершат свою работу.

Ожидаемое повышение эффективности от параллелизации можно описать законом Густафсона [5], который выражается формулами (4) и (5):

$$S = s + p \times N \tag{4}$$

$$s + p = 1 \tag{5}$$

где n - количество задействованных процессоров;

s - доля последовательных расчетов в программе;

p - доля параллельных расчетов в программе.

Закон Густафсона описывает теоретическое ускорение при масштабировании системы и увеличения сложности решаемых задач, что дает оценку эффективности организации параллельных вычислений.

Практический анализ эффективности (профилирование) многопоточных алгоритмов гораздо более сложен. Для оценки их

производительности обычно применяется практический анализ. Алгоритм воспроизводится несколько раз, и вычисляется среднее время его выполнения.

Так как на время выполнения программы может повлиять множество сторонних факторов (случайная сторонняя загрузка системы, функции энергосбережения, и т.п.), получить более точное время помогает инструментарий замера используемого процессорного времени на выполнение программы, который даёт время, которое использовало именно выполнение программы.

В то же время, при выполнении программы на нескольких потоках, каждый из них использует свое процессорное время, которое затем суммируется для получения финального результата. При подведении результатов вероятно можно будет заметить, что последовательный алгоритм будет использовать меньше процессорного времени, чем параллельный, так как происходили дополнительные вычисления для создания и синхронизации потоков, но при этом параллельный алгоритм закончит работу за меньший промежуток реального времени.

Поэтому для анализа эффективности разрабатываемого многопоточного алгоритма относительно последовательного будет использоваться именно реальное время.

Выводы по 1-й главе:

В данной главе была обобщена теория в сфере сжатия данных - классификация, применение, методы, ограничения, и характеристики. Также были разобраны базовые вычислительные ресурсы, по которым можно оценить эффективность разрабатываемых алгоритмов, и подходы к минимизации их затрат. Наконец, разобраны базовые вопросы разработки параллельного алгоритма.

2 Разработка эффективного алгоритма сжатия

2.1 Алгоритм сжатия

В рамках данной работы будет рассмотрен комплексный метод сжатия без потерь. Метод является комплексным, так как будет состоять из комбинации алгоритмов сжатия, применяемых над данными в определенном порядке. Будет задействовано два алгоритма предобработки, и затем метод статистической компрессии.

На вход алгоритм будет получать файл, который будет необходимо сжать. Обрабатываться файл будет поблочно, независимыми блоками заданной максимальной длины. Процесс обработки блока состоит из 3-х последовательных шагов:

- чтение блока в память;
- выполнение алгоритмов сжатия над блоком;
- запись сжатого блока на диск.

Перед каждым записанным сжатым блоком будет идти заголовок, содержащий размер блока, и отдельные данные, необходимые для работы некоторых методов декодирования. Работа алгоритма заканчивается, когда последний блок в файле будет обработан и записан.

Разберём первый метод предобработки - преобразование Барроуза-Уилера (Burrows-Wheeler transform, BWT). Преобразование заключается в генерации таблицы, содержащей все возможные битовые сдвиги строки, отсортированных в лексикографическом порядке. Последним столбцом таблицы будет являться преобразованная строка [12]. Пример работы метода на слове GOOGOL представлен на рисунке 3:

Рисунок 3 – Процесс работы преобразования BWT

Генерация в памяти огромной таблицы с каждым состоянием не является самым эффективным подходом. Для корректной сортировки достаточно представить строку в виде суффиксного массива.

Суффиксный массив - лексикографически отсортированный массив всех суффиксов строки. Для сортировки таким методом нет необходимости создавать новые массивы данных, так как программа может сравнивать два участка исходной строки по указателям на их начала. Результат сортировки применим на набор индексов каждого символа, что даст нам индексы строк в финальной трансформации таблицы. Другими словами, это будет первый столбец таблицы. Для получения последнего столбца, возьмем символы исходной строки с индексами $i - 1$ для каждого индекса в отсортированном списке.

Для восстановления исходных данных алгоритму декомпрессии потребуется, помимо самих данных в сжатом виде, местоположение первого символа исходного текста. Запишем его в виде индекса в заголовок блока.

Алгоритм обратного преобразования заключается в итеративном восстановлении исходной строки по тому же принципу, что и построение таблицы в алгоритме преобразования. Соответственно, и в этот раз, вместо итеративной генерации таблицы и ее многократной сортировки, возьмем индексы символов отсортированного сжатого блока, и, начиная с первого

символа, индекс которого будет считан из заголовка, обратным ходом восстановим исходную строку.

Второй метод предобработки - перемещение к началу (move-to-front, MTF). Основная идея преобразования состоит в замене каждого исходного символа на число, соответствующее его местоположению в стеке последних использовавшихся символов [2]. Таким образом, длинные последовательности символов будут заменены на последовательности нулей, а редко появляющиеся символы будут обозначаться большим числом. Большее количество последовательностей и перераспределение частот символов делает данные более предсказуемыми.

Процесс работы алгоритма представлен в виде таблицы на рисунке 4:

Шаг	Ввод	Вывод	Стэк
1	**1**114655544	1	1023456
2	1**1**14655544	10	1023456
3	11**1**4655544	100	1023456
4	1114**6**55544	1004	4102356
5	1114**6**55544	10046	6410235
6	11146**5**5544	100466	5641023
7	11146**5**5544	1004660	5641023
8	1114655**5**44	10046600	5641023
9	1114655**5**44	100466002	4561023
10	1114655**5**44	1004660020	4561023

Рисунок 4 – Процесс работы преобразования MTF

Восстановление происходит аналогичным образом, но в обратном порядке - индексы заменяются на символ с соответствующим индексом.

За конкретный метод статистического сжатия возьмем код Хаффмана. Выходные данные алгоритма можно представить в виде таблицы кодов разных длин для посимвольного кодирования исходной строки. Таблица

генерируется, исходя из статистической модели, описывающую вероятность появления (или общую частоту) каждого возможного символа исходной строки. Более частым символам задаются более короткие коды. Если сгенерированную таблицу отсортировать по частотам, то временная сложность процесса кодирования становится $O(n)$. Код Хаффмана является оптимальным методом для кодирования символов по отдельности [7].

Полный алгоритм работы следующий:

Сначала рассчитывается вероятность каждого уникального символа в полученной строке, и сохраняются в виде единой таблицы. Полученная таблица сортируется от самого частого символа до самого редкого.

Затем по таблице строится бинарное дерево. Построенное дерево затем обходится, и на каждом узле левой дуге задается значение "0", а правой дуге - "1", и генерируется словарь бинарных кодов, где каждому уникальному символу присваивается бинарный код, собранный из значений пройденных по пути к нему дуг.

Последним шагом является кодирование входных данных, что является простой последовательной конвертаций символов в бинарные коды по сгенерированному ранее словарю.

Пример таблицы вероятностей, бинарного дерева и готовой таблицы представлены на рисунке 5:

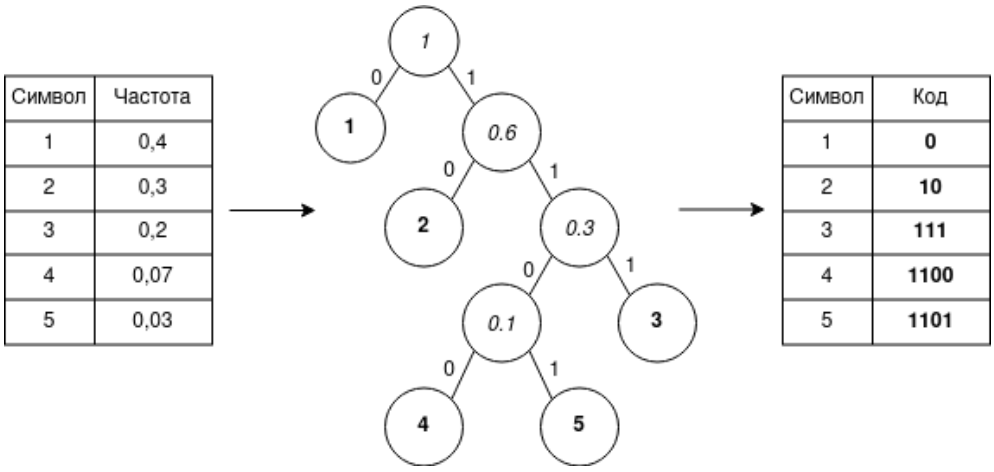


Рисунок 5 – Процесс работы кода Хаффмана

Чтобы допустить последующее восстановление данных, перед ними запишем бинарное дерево. Тогда алгоритм восстановления, после чтения дерева, путем побитового чтения будет итеративно проходить по дереву, выбирая дуги в соответствии с каждым прочитанным битом, и записывать каждое попадание в один из листов.

Таким образом построен процесс последовательного сжатия данных. Выбранные алгоритмы помогут добиться высокой эффективности сжатия для большого контингента возможных данных.

2.2 Параллелизация алгоритма

Рассмотрим возможность повышения производительности разработанного метода с помощью параллелизации.

При разработке параллельного процесса, необходимо помнить, что создание и удаление потоков - дорогостоящая операция. Поэтому для решения задач будут применяться группы потоков, созданных заранее в определенном количестве.

Главный процесс, требующий оптимизации - последовательность чтение-обработка-запись. Как было предложено ранее, применим буферизацию для минимизации времени, когда процессы обработки будут ожидать новые данные.

В то время, как процессы обработки запустятся одновременно, длительность их выполнения может относительно сильно варьироваться. Из этого вытекает проблема, что обработанные блоки данных будут поступать на выход в непредсказуемом порядке. Дополнение логики процесса, записывающего обработанные данные, строгим соблюдением порядка записи несет большой риск переполнения буфера, если процесс с нужным блоком сильно опоздает. Это приведет к полной остановке работы программы.

Альтернативой является добавление префикса к каждому блоку, который будет содержать номер блока. Это позволит записывать их в любом порядке, в каком процессы обработки будут их возвращать. Процесс восстановления, в свою очередь, будет считывать индексы, и будет откладывать в памяти блоки данных с неожиданными индексами. Здесь соблюдение порядка не приведет к переполнению буфера для процессов обработки, так как будет использоваться отдельный буфер.

Очевидным недостатком является незначительное повышение в размере выходного, сжатого файла. И таким образом, существование данной альтернативы еще раз акцентирует внимание на характере проблемы алгоритмов сжатия - постоянной балансировке достоинств и недостатков в зависимости от контекста поставленной задачи.

Другой важной деталью является небольшое изменение формата сжатых данных, так как добавляется необходимость передачи индекса блока. Автоматическое решение этой проблемы, не требующей ручного указания, каким методом был сжат архив, будет описано в 3-й главе, при планировании реализации программы.

Таким образом разработан алгоритм параллельного комплексного сжатия. Далее рассмотрим способы подтверждения эффективности применения именно параллельной версии алгоритма.

2.3 Практическая оценка эффективности алгоритма сжатия

Для того, чтобы сделать вывод о наличии повышения эффективности в разработанных параллельных алгоритмах, необходимо, как было рассмотрено ранее, провести практический сравнительный анализ. Требование использования именно практического анализа исходит из того, что асимптотический анализ параллельных процессов неоднозначен.

Для получения полноценных результатов, нужно провести эксперимент для нескольких параметров, которые можно изначально задать в разработанном параллельном алгоритме. Список таких параметров следующий:

- размер блока, на который будет разбит файл;
- количество потоков, выделенных для процессов обработки;

Также опишем критерии, по которым будет даваться оценка алгоритмам:

- коэффициент сжатия - во сколько раз был сжат файл;
- время компрессии - реальное время, ушедшее на выполнение процесса сжатия;
- время декомпрессии - реальное время, ушедшее на выполнение процесса восстановления.

Помимо верификации эффективности параллелизации, необходимо доказать преимущества использования методов предобработки, проведя сравнительный анализ с использованием разных наборов методов.

Выводы по 2-й главе:

В данной главе был рассмотрен комплексный алгоритм сжатия. Были описаны структура работы каждой части алгоритма в последовательной форме. Затем алгоритмы были проанализированы на возможности параллелизации, и были предложены новые, параллельные варианты их работы. Наконец, описаны параметры и критерии, по которым будет проводиться сравнительный анализ, подтверждающий полученное повышение эффективности от применения многопоточности.

3 Реализация, тестирование и анализ разработанных алгоритмов сжатия

3.1 Средства и структура программы

Реальная производительность высокоскоростных алгоритмов, по своей природе, зависит не только от их структуры, но и от эффективной реализации. Первый шаг к обеспечению релевантности разрабатываемого продукта - осмысленный выбор средств, на которых он будет построен. Для нашего алгоритма выберем язык программирования Rust.

Rust - язык программирования с открытым кодом, ключевыми приоритетами которого являются безопасность, скорость и полный функционал для создания параллельных программ в базовой комплектации. Rust является быстро развивающимся языком, в частности в плане применения в системном программировании и разработки ядер операционных систем.

Rust обеспечивает безопасность доступа к памяти, включая защиту от утечек, на этапе компиляции, отслеживая жизненные циклы всех ссылок на объекты через систему “borrow checking” ("проверка владения") [20]. Тем самым Rust избегает использования сборщика мусора и учета количества ссылок во время выполнения программы, что позволяет избежать неожиданных скачков в задержке в ключевых моментах и достичь более стабильной и эффективной работе программы.

Строгий учет и ограничения существования ссылок на объекты позволяет также позволяет исключить целый ряд ошибок в обращении с памятью, часто встречающихся в коде, написанном на других языках. Одной из таких ошибок является конкуренция (race condition), при которой работа системы зависит от порядка выполнения неконтролируемых частей кода. Гарантия соблюдения условия, что на объект одновременно может

существовать лишь одна ссылка, по которой можно произвести запись, позволяет исключить возможность конкуренции. Данные гарантии также сильно упрощают попытки формальной верификации, что позволяет добиться стабильно растущего применения в академических работах.

Определим главные структуры, которые будут использоваться в программе.

Первая из них - главный заголовок, с которого будет начинаться сжатый файл. Цель заголовка - дать алгоритму восстановления всю необходимую ему информацию для начала работы. Содержание заголовка следующее:

- сигнатура - состоит из 3 байтов, и является опознавательным знаком, по которому программа определит, имеет ли она дело с совместимым с работой файлом;
- код использованных параметров при сжатии - содержит в себе 1 бит, определяющий, использовалась ли параллелизация при сжатии.

Второй структурой является сжатый блок. Каждый блок также будет иметь свой заголовок, без которого его восстановление не является разумной задачей. Содержание блока следующее:

- длина блока в битах (4 байта);
- индекс первого символа в отсортированной таблице в методе BWT (4 байта);
- сжатые данные (n бит).

Третьей структурой является циклический буфер, который будет использоваться для связи потоков, работающих с файлами, и потоков, сжимающих поступающие данные.

Циклический буфер - структура данных, использующая единственный буфер фиксированного размера, за последним элементом которого следует его первый элемент. По форме доступа к данным буфер подобен очереди. Помимо данных, буфер хранит в себе два указателя - один на последний

прочитанный элемент, другой на последний записанный элемент. Указатель записи всегда будет впереди указателя чтения. Такая структура данных позволит избежать создания и передачи отдельных буферов для каждого потока.

На рисунке 6 представлена визуализация структуры циклического буфера:

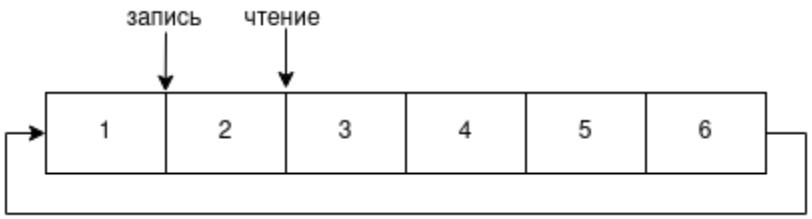


Рисунок 6 – Структура циклического буфера

Последней используемой структурой являются каналы. Каналы будут применены для распределения работы между потоками в процессе восстановления файла. Чтобы алгоритм восстановления отработал без ошибок, на вход должен поступить целый блок. Применение здесь циклического буфера не гарантирует, что один поток полностью прочитает свой блок, лишь если не применять большое количество синхронизации, что несомненно нанесет большой удар по производительности. Каналы позволяют передавать блок единой структурой, минуя описанную проблему.

Во время разработки, помимо базового функционала языка Rust редакции 2021 года и его стандартной библиотеки, будут применены отдельные готовые решения для выполнения различных подзадач, не относящихся непосредственно к теме работы. В экосистеме языка Rust эти решения носят название крейтов (crate). Перечислим нам необходимые:

- bitreader - побитовое чтение данных;
- bitvec - работа с данным в бинарном представлении;
- byteorder - чтение из файла значений определенного размера;

– clap - парсинг аргументов при запуске программы;
– crossbeam - манипуляция потоков;
– num - отдельные математические операции;
– ringbuf - безопасная для многопоточной работы реализация кольцевого буфера.

На вход программа будет получать входной файл, и количество потоков, выделенных для параллельной обработки. По умолчанию программа будет выполнять процесс сжатия, если не получит на вход специальный флаг.

Перейдем к реализации разработанных алгоритмов и последовательного процесса обработки.

3.2 Реализация последовательного алгоритма

Работа последовательного процесса сжатия состоит из открытия входного и выходных файлов, инициализации обычных статических буферов, и поблочного цикла чтения, обработки и записи данных вместе с соответствующим заголовком. На рисунке 7 представлена оптимизированная реализация данного процесса:

```rust
pub fn seq_encode(file_name: &str) {
    let mut reader: BufReader<File> = open_input_file(file_name);
    let mut writer: BufWriter<File> = open_output_file(file_name, ext: "bmh");

    let mut buf: [u8; _] = [0u8; BLOCK_SIZE];
    let mut res: [u8; _] = [0u8; BLOCK_SIZE];

    while let Ok(block_size: usize) = reader.read(&mut buf) {
        if block_size == 0 {
            break;
        }

        let block_out: CompressedBlock = encode_block(buf, block_size, &mut res);

        writer.write_u32::<LittleEndian>(block_out.length).unwrap();
        writer.write_u32::<LittleEndian>(block_out.s0_idx).unwrap();
        writer.write_all(&block_out.buf).unwrap();
    }
}
```

Рисунок 7 – Листинг функции последовательного сжатия файла

На рисунке 8 представлена функция encode_block, которая, используя два буфера для избежания излишнего копирования, сжимает данные, и возвращает структуру, содержащую сам блок и его заголовок:

```rust
pub fn encode_block(
    mut buf: [u8; BLOCK_SIZE],
    block_size: usize,
    res: &mut [u8; BLOCK_SIZE],
) -> CompressedBlock {
    let mut s0_idx: usize = 0;
    bwt(&buf, block_size, res, &mut s0_idx);
    buf = *res;

    mtf(&buf, block_size, res);
    buf = *res;

    let mut bv: BitVec<u8, Msb0> = huffman(&buf, block_size);
    bv.set_uninitialized(false);

    CompressedBlock {
        length: bv.len().try_into().unwrap(),
        s0_idx: s0_idx.try_into().unwrap(),
        buf: bv.into_vec(),
    }
}
```

Рисунок 8 – Листинг функции сжатии блока

Рассмотрим отдельные детали реализаций функций преобразования и сжатия.

В алгоритме BWT функция генерации и сравнения суффиксных массивов применяет функционал срезов. Срез - ссылка на участок массива. Используя их, удается полностью пропустить шаг генерации таблиц как сдвигов, так и суффиксов, что дает массивный плюс к вычислительной эффективности программы. На рисунке 9 представлена функция, возвращающая индексы отсортированных символов.

```rust
fn argsort(data: &[u8], block_size: usize) -> Vec<usize> {
    let mut indices: Vec<usize> = (0..block_size).collect::<Vec<_>>();
    indices.sort_by(compare: |&a: usize, &b: usize| data[a..block_size].cmp(&data[b..block_size]));
    indices
}
```

Рисунок 9 – Листинг функции сортировки суффиксных массивов

Реализация алгоритма перемещения к началу для сдвига соответствующего элемента на первый слот использует функционал быстрого копирования срезов массивов. В своей основе функция copy_within

использует системный вызов memmove, который работает напрямую с памятью. Полный код преобразования перемещения к началу представлен на рисунке 10:

```rust
pub fn mtf(buf: &[u8; BLOCK_SIZE], block_size: usize, res: &mut [u8; BLOCK_SIZE]) {
    let mut alphabet: [u8; 256] = [0; 256];
    for (i: usize, a: &mut u8) in alphabet.iter_mut().enumerate() {
        *a = i as u8;
    }

    for i: usize in 0..block_size {
        let index: usize = alphabet.iter().position(|&a: u8| a == buf[i]).unwrap();
        res[i] = index as u8;
        alphabet.copy_within(src: 0..index, dest: 1);
        alphabet[0] = buf[i];
    }
}
```

Рисунок 10 – Листинг функции преобразования MTF

Реализация алгоритма составления кода Хаффмана использует несколько структур, упрощающих процесс генерации бинарного дерева и работы с ним. Переходным элементом между таблицей частот и сгенерированным бинарным деревом была реализация приоритетной очереди BinaryHeap. Применение приоритетной очереди позволяло сразу расставить листы дерева в соответствии с таблицей частот, и затем рекурсивно сгенерировать бинарное дерево.

Структура узла дерева реализована через тип данных Enum, которая в языке Rust способна не только обозначать одно конкретное состояние, но и хранить различные данные для каждого возможного состояния. Реализация узла дерева представлена на рисунке 11:

```rust
#[derive(Debug, Eq, PartialEq)]
3 implementations
enum NodeKind {
    Internal(Box<Node>, Box<Node>),
    Leaf(u8),
}

You, 2 days ago | 1 author (You)
#[derive(Debug, Eq, PartialEq)]
5 implementations
struct Node {
    frequency: usize,
    kind: NodeKind,
}
```

Рисунок 11 – Листинг структуры узла дерева

Общий процесс восстановления файла не сильно отличается от процесса сжатия. Ключевым отличием является порядок чтения - сначала читается заголовок блока, а затем блок данных. Над прочитанным блок открывается побитовый читатель, и с помощью него выполняют свою работы алгоритмы восстановления, после чего восстановленный блок записывается в новый файл.

На рисунке 12 представлена реализация функции decode_block:

```rust
pub fn decode_block(block: CompressedBlock) -> Vec<u8> {
    let bit_reader: BitReader = BitReader::new(bytes: &block.buf);
    let mut res: Vec<u8> = huffman_reverse(buf: &mut bit_reader.relative_reader_atmost(len: block.length.into()));

    let mut buf: Vec<u8> = res.clone();
    let buf_n: usize = buf.len();
    mtf_reverse(&buf, block_size: buf_n, &mut res);

    buf = res.clone();
    let buf_n: usize = buf.len();
    bwt_reverse(&buf, block_size: buf_n, &mut res, s0_idx: &mut (block.s0_idx as usize));

    res
}
```

Рисунок 12 – Листинг функции восстановления блока

Далее, приступим к дополнению программы элементами параллельных вычислений.

3.3 Реализация параллельного алгоритма

При реализации многопоточных алгоритмов всегда важно иметь ввиду, в каких случаях работа потоков может пересечься. Для обработки таких случаев и общения потоков между собой существуют стандартизированные подходы. Разберем те, которые будут использоваться в разрабатываемой программе, а именно атомарные переменные и мьютексы.

Атомарные переменные - переменные, гарантирующие выполнения над ними записи и чтения за минимальный отрезок времени, во время которого другие потоки не успеют воспользоваться переменной. [6] Их использование позволяет избежать части необходимой синхронизации.

Мьютексы - объекты синхронизации, позволяющие нескольким потокам безопасно использовать общие ресурсы: перед обращением к разделяемым ресурсам поток блокирует мьютекс, выполняет какие-либо операции и после этого разблокирует его. [1] Обеспечение доступа к мьютексам через распределенную переменную позволяет обеспечить последовательные записи в буфер между несколькими потоками.

Использование этих элементов дает гарантию корректной работы нашей программы уже со стадии компиляции. Перейдем к описанию самой программы.

Для синхронизация потоков в процессе сжатия были использованы следующие три атомарные переменные:

- WORKERS_ACTIVE - количество активных потоков обработки; используется потоком записи для окончания работы;
- BLOCK_INDEX - счетчик блоков; используется процессами для записи в файл;
- END_OF_FILE - флаг окончания чтения файла; выставляется процессом чтения, и используется остальными процессами для окончания работы.

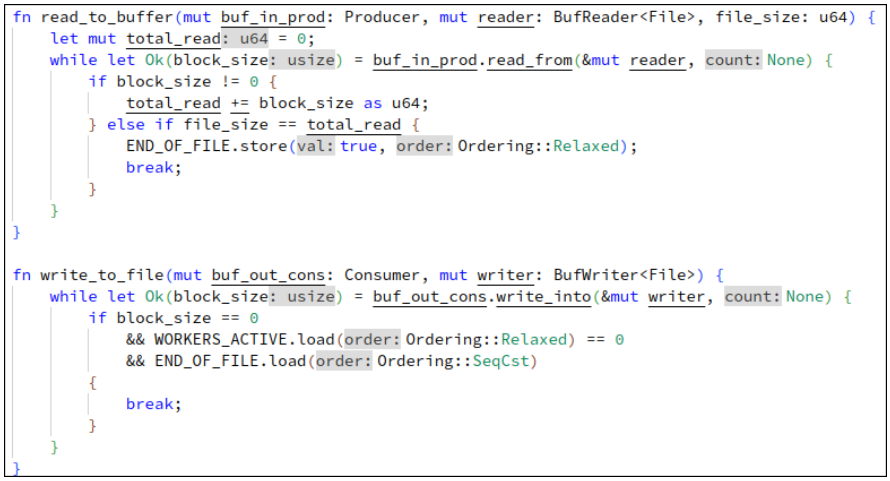
Процесс сжатия начинается с открытия файлов и инициализации двух циклических буферов. Затем создается три группы потоков:

- один поток для чтения;
- N потоков для обработки;
- один поток для записи.

Поток чтения получает на вход открытый файл, вход первого циклического буфера, и размер открытого файла. Цель потока чтения - своевременно пополнять буфер. Как только будет прочитан весь открытый файл, поток выставляет статическую булеву переменную END_OF_FILE в положительное состояние, чтобы оповестить остальные потоки о завершении своей работы.

Поток записи получает на вход открытый выходной файл, и выход второго циклического буфера. Его цель - перенаправлять данные из буфера в файл, задействуя устройство записи, освобождая процессы обработки от . Если в момент очередного неуспешного чтения из пустого буфера, поток обнаруживает, что все процессы обработки закончили свою работу, то он также завершает свою работу.

На рисунке 13 представлен процессы работы потоков чтения и записи:



```
fn read_to_buffer(mut buf_in_prod: Producer, mut reader: BufReader<File>, file_size: u64) {
    let mut total_read: u64 = 0;
    while let Ok(block_size: usize) = buf_in_prod.read_from(&mut reader, count: None) {
        if block_size != 0 {
            total_read += block_size as u64;
        } else if file_size == total_read {
            END_OF_FILE.store(val: true, order: Ordering::Relaxed);
            break;
        }
    }
}

fn write_to_file(mut buf_out_cons: Consumer, mut writer: BufWriter<File>) {
    while let Ok(block_size: usize) = buf_out_cons.write_into(&mut writer, count: None) {
        if block_size == 0
            && WORKERS_ACTIVE.load(order: Ordering::Relaxed) == 0
            && END_OF_FILE.load(order: Ordering::SeqCst)
        {
            break;
        }
    }
}
```

Рисунок 13 – Листинг функций потоков чтения и записи

Потоки обработки, тем же временем, читают блоки максимальной длины из первого буфера, обрабатывают их, и передают во второй буфер. Каждый поток имеет свои локальные статические буферы, и копии указателей на мьютексы, содержащих указатели на кольцевые буфера.

Перед записью очередного обработанного блока, поток инкрементирует статическую атомарную переменную BLOCK_INDEX, и добавляет ее прошлое значение в заголовок блока. Это необходимо для сохранения порядка блоков при восстановление файла.

Если поток обработки обнаруживает, что буфер пуст, и переменная END_OF_FILE положительна, то он завершает свою работу.

На рисунке 14 представлена реализация потока обработки:

Рисунок 14 – Листинг потока обработки

Процесс восстановления, как было разобрано ранее, использует каналы для связи потоков. Это упрощает .

Так как процесс сжатия, в целях эффективности, не занимался сортировкой блоков предварительно их записи на диск, эта задача переходит на процесс восстановления. Процессы чтения и обработки работают довольно прямо, соответственно считывая и обрабатывая данные, и передавая их дальше по цепочке. Поэтому задача сортировки остается за потоком записи.

Поток записи получает по каналу сообщения в виде восстановленного блока данных и его индекса. Поток сравнивает индекс нового блока со своим внутренним счетчиком, и если новый блок оказался вне очереди, то поток откладывает его в свой отдельный локальный буфер, реализованный в виде словаря, где ключом является номер блока. После каждой успешной записи очередного блока, поток проходит по буферу отложенных блоков, и записывает те, которых очередь наступила.

На рисунке 15 представлена реализация потока записи:

```rust
fn write_to_file(channel: Receiver<UncompressedBlock>, mut writer: BufWriter<File>) {
    let mut block_n: u32 = 1;
    let mut deferred_blocks: HashMap<u32, Vec<u8>> = HashMap::new();
    while let Ok(block: UncompressedBlock) = channel.recv() {
        if block.n == block_n {
            writer.write_all(&block.buf).unwrap();
            block_n += 1;
            while let Some(b: Vec<u8>) = deferred_blocks.remove(&block_n) {
                writer.write_all(buf: &b).unwrap();
                block_n += 1;
            }
        } else {
            deferred_blocks.insert(k: block.n, v: block.buf);
        }
    }
}
```

Рисунок 15 – Листинг функции записи восстановленного файла

Таким образом были реализованы методы параллельного сжатия и восстановления данных. Наконец, проведем эксперимент для доказательства получения выгоды от улучшения алгоритма.

3.4 Тестирование и сравнительный анализ эффективности

Все эксперименты ставились на программе, скомпилированной с использованием стандартных оптимизаций. Были учтены требования проведения сравнительного анализа, описанные в главе 1.3. Время засекалось с использованием встроенной библиотекой работы со временем, без учета времени, уходящего на открытие/закрытие файлов. Эксперименты проведены на вычислительной машине с 16-ю процессорными ядрами, и в качестве устройства хранения использовался SSD вида M.2.

Эксперимент будет поставлен с использованием трех входных файлов, каждый из которых находится в открытом доступе в целях использования в сфере исследований алгоритмов сжатия без потерь. Разберем релевантные свойства данных файлов:

– enwik8 - первые 10^8 байт снимка базы данных википедии, датированных 3-м марта 2006 года; файл имеют форму XML

документа, в основном состоящего из английского текста; размер - 100 мегабайт;

– nci - фрагмент базы данных, содержащей химические составы структур, их компоненты, координаты, свойства и прочее; файл имеет форму SDF, которые является специфичным форматом для хранения молекулярных структур; размер - 32 мегабайта;
– samba - упакованный исходный код имплементации сетевого протокола SMB; кроме кода содержит документацию и графику; размер - 21.6 мегабайта.

Как планировалось ранее, проведем несколько экспериментов, чтобы с нескольких сторон доказать эффективность выбранного в работе подхода.

Первым оценим выбор самого комплексного алгоритма сжатия, доказав эффективность методов предобработки, путем измерения коэффициента сжатия. В таблице 1 представлены результаты эксперимента:

Входной файл	Без предобработки	BWT	MTF	BWT+MTF
enwik8	1.57	1.57	1.51	2.97
nci	3.29	3.30	2.77	6.34
samba	1.44	1.44	1.48	3.42

Таблица 1 – Коэффициенты сжатия файлов для разных наборов алгоритмов

Как можно увидеть, использование одного из методов предобработки имеет сомнительную пользу, показывая неоднозначные результаты для разных видов данных. В то время как использование обоих методов приносит почти двукратное повышение эффективности сжатия. Поэтому для дальнейших экспериментов будет использовать только полный набор разработанных алгоритмов.

Далее, докажем повышение эффективности в связи с использованием разработанной модели параллелизации. Во всех последующих графиках будет использоваться следующая нотация - синим цветом обозначается процесс сжатия, красным - процесс восстановления. Ось абсцисс указывает на затраченное на выполнение время (в секундах), ось ординат - на количество используемых потоков.

Рассмотрим скорость работы программы для разного количества потоков.

На рисунке 16 представлен график скорости выполнения сжатия и восстановления для первого файла, enwik8:

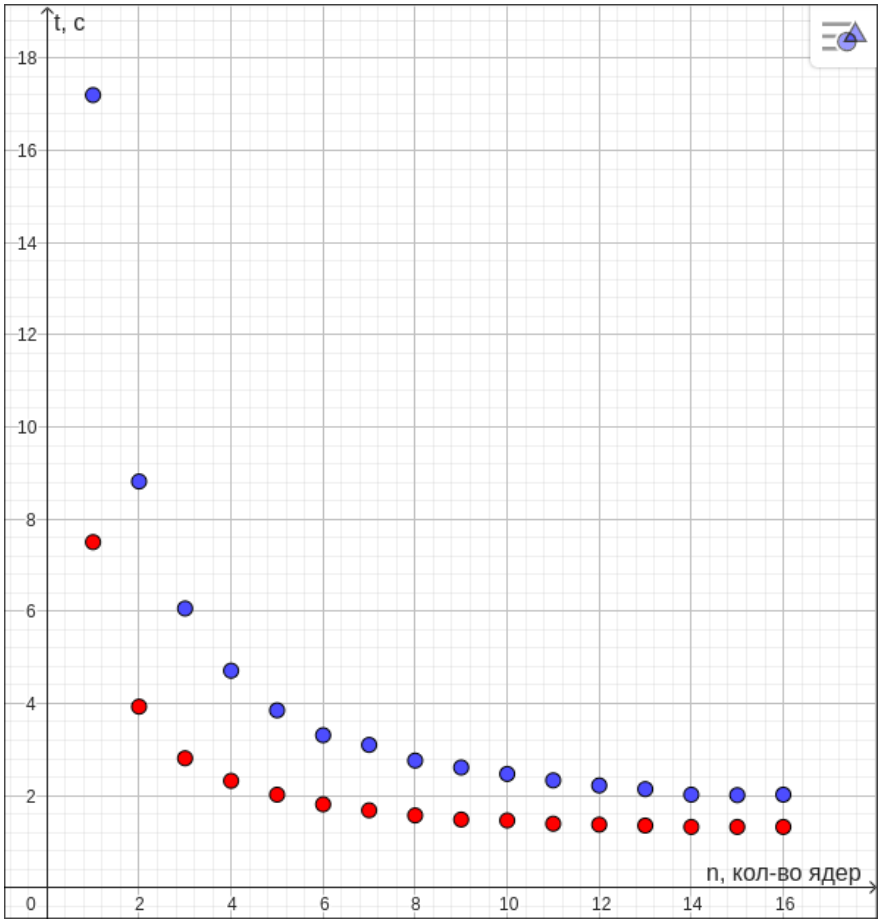


Рисунок 16 - График отношения скорости к количеству выбранных потоков при работе с файлом enwik8

На рисунке 17 представлен график скорости выполнения сжатия и восстановления для второго файла, nci:

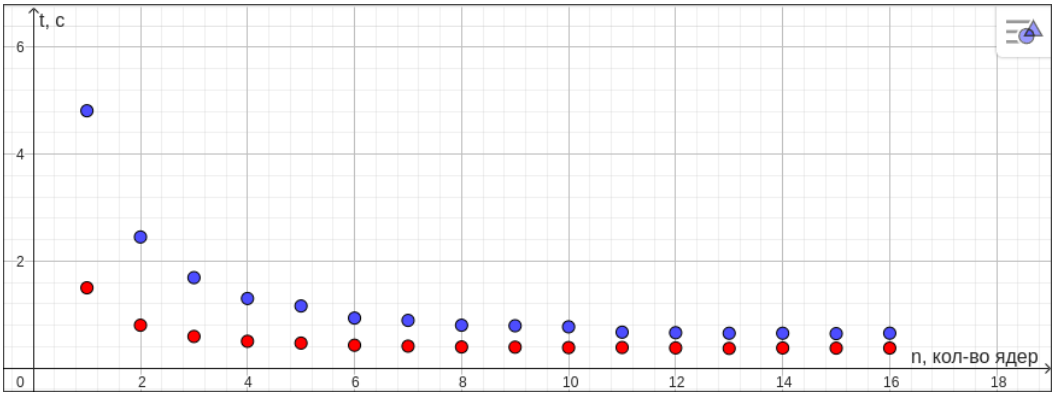


Рисунок 17 - График отношения скорости к количеству выбранных потоков при работе с файлом nci

На рисунке 18 представлен график скорости выполнения сжатия и восстановления для третьего файла, samba:

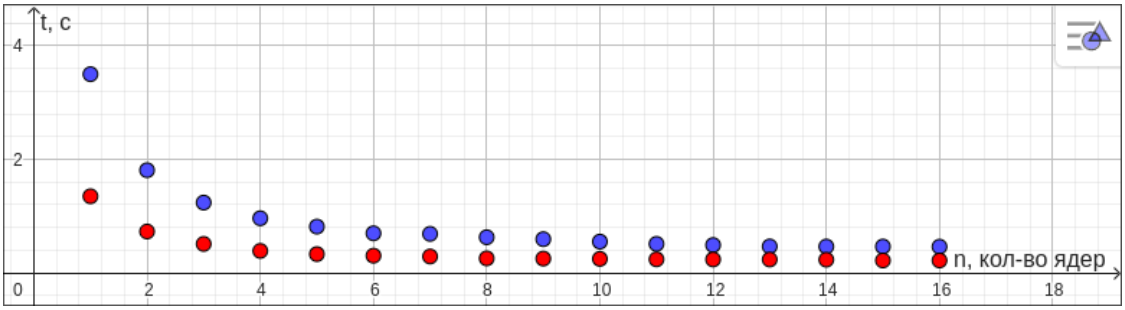


Рисунок 18 - График отношения скорости к количеству выбранных потоков при работе с файлом samba

Таким образом, результаты эксперимента подтверждают, что разработка и реализация параллельного алгоритма позволила получить многократное ускорение в скорости как сжатия, так и восстановления файла. Процесс восстановления файла, являясь менее трудоемким в плане использования вычислительных ресурсов в сравнении с процессом сжатия, получает в меньшей степени , но тем не менее в порядке 3-х раз.

Предел дополнительной эффективности от повышения количества потоков достигается при приближении к физическому количеству доступных потоков. Следовательно, при дальнейшем развитии и распространении данного продукта, имеет место добавить функционал для автоматического выбора количества потоков на основе доступной информации о системе.

Выводы по 3-й главе

В данной главе был подробно разобран процесс реализации программного компонента для разработанных во 2-й главе алгоритмов. Были выбраны средства и программные подходы, гарантирующие высокую эффективность алгоритма, и были описаны ключевые моменты в реализации каждого модуля.

Наконец, было поставлено несколько сравнительных экспериментов для разрешения следующих вопросов, заданных во время описания сферы задачи:

- имеет ли смысл использовать методы предобработки для улучшения работы методов сжатия;
- какой степени повышение эффективности можно ожидать от применения параллельных вычислений.

Результаты указали на наличие существенных преимуществ как от применения методов предобработки, так и от параллельных вычислений.

Заключение

В ходе выполнения выпускной квалификационной работы было проведено исследование, объектом которого являлись алгоритмы сжатия информации.

Целью работы был анализ эффективности параллелизации комплексного алгоритма сжатия. В ходе работы были поставлены и выполнены следующие задачи:

- изучены и проанализированы теоретические основы в сфере оптимизации вычислительных алгоритмов и сфере сжатия информации;
- разработаны комплексный алгоритм сжатия, состоящий из двух методов предобработки и метода статистической компрессии, многопоточный вариант алгоритма, и описаны условия проводимого анализа;
- реализованы разработанные алгоритмы и проведено исследование на степень повышения его эффективности от параллелизации.

В результате проделанной работы были сделаны следующие выводы

- большая часть эффективности алгоритмов сжатия исходит из корректного совмещения нескольких подходов;
- язык программирования Rust является идеальным инструментов для разработки критически важных компонентов;
- эффективность программы зависит насколько от грамотного подхода к решению задачи, так и от эффективной реализации с использованием соответствующих средств.

Список используемой литературы

1. Мютексы и семафоры. // МГТУ имени Н.Э.Баумана [Электронный ресурс]. URL: https://e-learning.bmstu.ru/iu6/mod/page/view.php?id=87 (дата обращения: 15.04.2023).
2. Arnavut, Z. Move-to-Front and inversion coding / Z. Arnavut Data Compression Conference, Snowbird, UT, USA - 2000.
3. Daintith J., Wright E. A Dictionary of Computing (6 ed.) / J. Daintith, E. Wright Oxford University Press - 2008.
4. GDCC - 3rd Edition of Global Data Compression Competition // GDCC Portal [Электронный ресурс]. URL: https://gdcc.tech/ (дата обращения 03.04.2023).
5. Gustafson, J.L. Gustafson’s Law / J.L. Gustafson Communications of the ACM - 2011.
6. Hesselink, W.H. An assertional proof for a construction of an atomic variable / W.H. Hesselink Formal Aspects of Computing - 2004.
7. Huffman, D.A. A Method for the Construction of Minimum-Redundancy Codes / D.A. Huffman Resonance - 1952.
8. Javed M.Y., Nadeem A. Data compression through adaptive Huffman coding schemes / M.Y. Javed, A. Nadeem Intelligent Systems and Technologies for the New Millennium - 2000.
9. Krishna K. Multithreading Implementations / K. Krishna The University of Texas at Arlington - 1998.
10. Long D., Jia W., Li M. Optimal Prefix Codes And Huffman Codes / D. Long, W. Jia, M. Li International Journal of Computer Mathematics - 2003.
11. Mahdi O., Mohammed M., Mohamed A. Implementing a Novel Approach an Convert Audio Compression to Text Coding via Hybrid Technique / O. Mahdi, M. Mohammed, A. Mohamed, IJCSI International Journal of Computer Science Issues - 2013.

12. Manzini, G. The Burrows-Wheeler Transform: Theory and Practice / G. Manzini Mathematical Foundations of Computer Science - 1999.
13. MPI: A Message-Passing Interface Standard / Message P Forum - 1994.
14. Reed, E. Patina: A Formalization of the Rust Programming Language / E. Reed University of Washington - 2015.
15. Robinson A.H., Cherry C. Results of a prototype television bandwidth compression scheme / A.H. Robinson, Cherry C., Proceedings of the IEEE - 1967.
16. Salomon, D. Data compression: the complete reference / D. Salomon Springer Science & Business Media – 2004.
17. Shannon, C.E. A Mathematical Theory of Communication / C.E. Shannon The Bell System Technical Journal - 1948.
18. Tan L.S., Lau S.P., Tan C.E. Optimizing LZW Text Compression Algorithm via Multithreading Programming / L.S. Tan, S.P. Lau, C.E. Tan IEEE 9th Malaysia Internation Coneference of Communications - 2009.
19. Wilkinson B., Michael A. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers / B. Wilkinson, A. Michael Pearson, 2nd edition - 1999.
20. Yemliha T. Performance and Memory Space Optimizations for Embedded Systems / T. Yemliha Syracuse University - 2011.

Приложение А

Листинг программного кода алгоритмов сжатия

consts.rs:

```
// 900 kbytes
pub const BLOCK_SIZE: usize = 900 * 1024;
pub const BUFFER_SIZE: usize = BLOCK_SIZE * 2;
```

compression/bwt.rs:

```
use crate::consts::BLOCK_SIZE;
fn argsort(data: &[u8], block_size: usize) -> Vec<usize> {
    let mut indices = (0..block_size).collect::<Vec<_>>();
    indices.sort_by(|&a, &b| data[a..block_size].cmp(&data[b..block_size]));
    indices
}
pub fn bwt(buf: &[u8; BLOCK_SIZE], block_size: usize, res: &mut [u8; BLOCK_SIZE], s0_idx: &mut usize) {
    let indices = argsort(buf, block_size);
    for i in 0..block_size {
        let ind = indices[i];
        if ind != 0usize {
            res[i] = buf[ind - 1];
        } else {
            *s0_idx = i;
            res[i] = buf[block_size - 1];
        }
    }
}
fn argsort2<T: Ord>(data: &[T], block_size: usize) -> Vec<usize> {
```

```
    let mut indices = (0..block_size).collect::<Vec<_>>();
    indices.sort_by_key(|&i| &data[i]);
    indices
}
pub fn bwt_reverse(buf: &[u8], block_size: usize, res: &mut [u8], s0_idx: &mut usize) {
    let indices = argsort2(buf, block_size);
    let s0 = buf[*s0_idx];
    let mut j = *s0_idx;
    for r in &mut res[0..block_size - 1] {
        if buf[indices[j]] == s0 && indices[j] <= *s0_idx {
            j -= 1;
        }
        j = indices[j];
        *r = buf[j];
    }
    res[block_size - 1] = s0;
}
```

mtf.rs:

```
use crate::consts::BLOCK_SIZE;

pub fn mtf(buf: &[u8; BLOCK_SIZE], block_size: usize, res: &mut [u8; BLOCK_SIZE]) {
    let mut alphabet: [u8; 256] = [0; 256];
    for (i, a) in alphabet.iter_mut().enumerate() {
        *a = i as u8;
    }
    for i in 0..block_size {
```

```
        let index = alphabet.iter().position(|&a| a == buf[i]).unwrap();
        res[i] = index as u8;
        alphabet.copy_within(0..index, 1);
        alphabet[0] = buf[i];
    }
}
pub fn mtf_reverse(buf: &[u8], block_size: usize, res: &mut [u8]) {
    let mut alphabet: [u8; 256] = [0; 256];
    for (i, a) in alphabet.iter_mut().enumerate() {
        *a = i as u8;
    }
    for i in 0..block_size {
        let index = buf[i] as usize;
        res[i] = alphabet[index];
        alphabet.copy_within(0..index, 1);
        alphabet[0] = res[i];
    }
}
```

huffman.rs:

```
use bitreader::BitReader;
use bitvec::prelude::Msb0;
use bitvec::vec::BitVec;
use bitvec::view::BitView;
use core::panic;
use std::collections::binary_heap::BinaryHeap;
use std::collections::BTreeMap;
use crate::consts::BLOCK_SIZE;
```

```
#[derive(Debug, Eq, PartialEq)]
enum NodeKind {
    Internal(Box<Node>, Box<Node>),
    Leaf(u8),
}
#[derive(Debug, Eq, PartialEq)]
struct Node {
    frequency: usize,
    kind: NodeKind,
}
impl Ord for Node {
    fn cmp(&self, rhs: &Self) -> std::cmp::Ordering {
        rhs.frequency.cmp(&self.frequency)
    }
}
impl PartialOrd for Node {
    fn partial_cmp(&self, rhs: &Self) -> Option<std::cmp::Ordering> {
        Some(self.cmp(rhs))
    }
}
type HuffmanCodeMap = BTreeMap<u8, BitVec>;
pub fn huffman(buf: &[u8; BLOCK_SIZE], block_size: usize) -> BitVec<u8, Msb0> {
    let mut count = [0usize; 256];
    for i in 0..block_size {
        count[buf[i] as usize] += 1;
    }
    let mut prio_queue = BinaryHeap::new();
    for (i, freq) in count.iter().enumerate() {
```

```
            if *freq > 0 {
                prio_queue.push(Node {
                    frequency: *freq,
                    kind: NodeKind::Leaf(i as u8),
                });
            }
        }
        while prio_queue.len() > 1 {
            let left_child = prio_queue.pop().unwrap();
            let right_child = prio_queue.pop().unwrap();
            prio_queue.push(Node {
                frequency: right_child.frequency + left_child.frequency,
                kind: NodeKind::Internal(Box::new(left_child),
Box::new(right_child)),
            });
        }
        let mut codes = HuffmanCodeMap::new();
        generate_codes(prio_queue.peek().unwrap(), BitVec::new(), &mut
codes);
        let mut res = BitVec::new();
        encode_node(prio_queue.peek().unwrap(), &mut res);
        (0..block_size).for_each(|i| {
            let c = codes.get(&buf[i]).unwrap();
            res.extend(c);
        });
        res.set_uninitialized(false);
        res.force_align();
        res
    }
```

```
    fn encode_node(node: &Node, res: &mut BitVec<u8, Msb0>) {
        match &node.kind {
            NodeKind::Internal(l, r) => {
                res.push(false);
                encode_node(l, res);
                encode_node(r, res);
            }
            NodeKind::Leaf(c) => {
                res.push(true);
                let mut symbol: BitVec<u8> = (*c).view_bits().into();
                symbol.reverse();
                // println!("{} - {}", c, symbol);
                res.append(&mut symbol);
            }
        }
    }
    fn generate_codes(node: &Node, prefix: BitVec, out_codes: &mut HuffmanCodeMap) {
        match node.kind {
            NodeKind::Internal(ref l, ref r) => {
                let mut left_prefix = prefix.clone();
                left_prefix.push(false);
                generate_codes(l, left_prefix, out_codes);

                let mut right_prefix = prefix;
                right_prefix.push(true);
                generate_codes(r, right_prefix, out_codes);
            }
            NodeKind::Leaf(ch) => {
```

```
                out_codes.insert(ch, prefix);
            }
        }
    }
    fn read_node(r: &mut BitReader) -> Node {
        if let Ok(b) = r.read_bool() {
            match b {
                false => {
                    let left_child = read_node(r);
                    let right_child = read_node(r);
                    return Node {
                        frequency: 0,
                        kind:                    NodeKind::Internal(Box::new(left_child),
Box::new(right_child)),
                    };
                }
                true => {
                    let val = r.read_u8(8).unwrap();
                    return Node {
                        frequency: 0,
                        kind: NodeKind::Leaf(val),
                    };
                }
            }
        };
        panic!("Could not read tree");
    }
    pub fn huffman_reverse(buf: &mut BitReader) -> Vec<u8> {
        let tree = Box::new(read_node(buf));
```

```
    let mut current_node = &tree;
    let mut res = Vec::new();
    let mut tempstr = String::new();
    while let Ok(b) = buf.read_bool() {
        if let NodeKind::Internal(l, r) = &current_node.kind {
            match b {
                true => {
                    current_node = r;
                    tempstr += "1";
                }
                false => {
                    current_node = l;
                    tempstr += "0";
                }
            }
        }
        if let NodeKind::Leaf(c) = &current_node.kind {
            res.push(*c);
            tempstr.clear();
            current_node = &tree;
        }
    }
    res
}
```

Приложение Б

Листинг программного кода многопоточного процесса сжатия

par/encode.rs:

```
use std::{
    fs::File,
    io::{BufReader, BufWriter, Write},
    sync::{
        atomic::{AtomicBool, AtomicU32, Ordering},
        Arc, Mutex,
    },
    time::Instant,
};
use byteorder::{LittleEndian, WriteBytesExt};
use crate::{compression::encode_block, consts::BLOCK_SIZE};
use super::{initialize_ring_buffers, open_input_file, open_output_file, Consumer, Producer};
static END_OF_FILE: AtomicBool = AtomicBool::new(false);
static BLOCK_INDEX: AtomicU32 = AtomicU32::new(1);
static WORKERS_ACTIVE: AtomicU32 = AtomicU32::new(0);
pub fn par_encode(file_name: &str, n_workers: u8) {
    // Open files
    let (file_size, reader) = open_input_file(file_name);
    let writer = open_output_file(file_name, "bmh");
    let (buf_in_prod, buf_in_cons) = initialize_ring_buffers();
    let (buf_out_prod, buf_out_cons) = initialize_ring_buffers();
    let start = Instant::now();
    crossbeam::scope(|s| {
        s.spawn(|_| {
```

```
            read_to_buffer(buf_in_prod, reader, file_size);
        });
        s.spawn(|_| {
            write_to_file(buf_out_cons, writer);
        });
        let shared_buf_in_cons = Arc::new(Mutex::new(buf_in_cons));
        let shared_buf_out_prod = Arc::new(Mutex::new(buf_out_prod));
        for _ in 0..n_workers {
            let cons = Arc::clone(&shared_buf_in_cons);
            let prod = Arc::clone(&shared_buf_out_prod);
            s.builder()
                .stack_size(4 * 1024 * 1024)
                .spawn(move |_| {
                    WORKERS_ACTIVE.fetch_add(1, Ordering::Relaxed);
                    let mut local_buf = [0u8; BLOCK_SIZE];
                    let mut local_res = [0u8; BLOCK_SIZE];
                    let mut block_size = 0;
                    let mut block_n = 0;
                    loop {
                        if let Ok(mut lock) = cons.lock() {
                            block_size = lock.pop_slice(&mut local_buf);
                            if block_size != 0 {
                                block_n = BLOCK_INDEX.fetch_add(1,
Ordering::SeqCst);
                            } else if END_OF_FILE.load(Ordering::SeqCst) {
                                WORKERS_ACTIVE.fetch_sub(1, Ordering::Relaxed);
                                break;
                            }
                        }
```

```
                    if block_size != 0 {
                        let block_out = encode_block(local_buf, block_size, &mut
local_res);
                        let w = &mut prod.lock().unwrap();
                        w.write_u32::<LittleEndian>(block_n).unwrap();
                        w.write_u32::<LittleEndian>(block_out.length).unwrap();
                        w.write_u32::<LittleEndian>(block_out.s0_idx).unwrap();
                        w.write_all(&block_out.buf).unwrap();
                    }
                }
            })
            .unwrap();
        }
    })
    .unwrap();
    let duration = start.elapsed();
    println!("File encoded in {:?}", duration);
}
fn read_to_buffer(mut buf_in_prod: Producer, mut reader:
BufReader<File>, file_size: u64) {
    let mut total_read = 0;
    while let Ok(block_size) = buf_in_prod.read_from(&mut reader, None) {
        if block_size != 0 {
            total_read += block_size as u64;
        } else if file_size == total_read {
            END_OF_FILE.store(true, Ordering::Relaxed);
            break;
        }
    }
```

```
    }
    fn write_to_file(mut buf_out_cons: Consumer, mut writer: BufWriter<File>)
{
        while let Ok(block_size) = buf_out_cons.write_into(&mut writer, None) {
            if block_size == 0
                && WORKERS_ACTIVE.load(Ordering::Relaxed) == 0
                && END_OF_FILE.load(Ordering::SeqCst)
            {
                break;
            }
        }
    }

    decode.rs:
    use std::{
        collections::HashMap,
        fs::File,
        io::{BufReader, BufWriter, Read, Write},
        time::Instant,
    };
    use byteorder::{LittleEndian, ReadBytesExt};
    use crossbeam::channel::{bounded, unbounded, Receiver, Sender};
    use num::integer::div_ceil;
    use crate::compression::{decode_block, CompressedBlock,
UncompressedBlock};
    use super::{open_input_file, open_output_file};
    pub fn par_decode(file_name: &str, n_workers: u8) {
        // Open files
        let (_, reader) = open_input_file(file_name);
```

```
let writer = open_output_file(file_name, "out");
let (sender_in, receiver_in) = bounded(4);
let (sender_out, receiver_out) = bounded(4);
let start = Instant::now();
crossbeam::scope(|s| {
    s.spawn(|_| {
        read_to_buffer(sender_in, reader);
    });
    s.spawn(|_| {
        write_to_file(receiver_out, writer);
    });
    for _ in 0..n_workers {
        let local_rec = receiver_in.clone();
        let local_snd = sender_out.clone();
        s.builder()
            .stack_size(4 * 1024 * 1024)
            .spawn(move |_| loop {
                match local_rec.recv() {
                    Ok(block) => {
                        let n = block.0;
                        let decoded_block = decode_block(block.1);
                        local_snd
                            .send(UncompressedBlock {
                                n,
                                buf: decoded_block,
                            })
                            .unwrap();
                    }
                    Err(_) => {
```

```
                    drop(local_snd);
                    break;
                }
            }
        })
        .unwrap();
    }
    drop(sender_out);
  })
  .unwrap();
  let duration = start.elapsed();
  println!("File decoded in {:?}", duration);
}
fn read_to_buffer(channel: Sender<(u32, CompressedBlock)>, mut reader: BufReader<File>) {
  while let Ok(block_n) = reader.read_u32::<LittleEndian>() {
    let mut buf = Vec::new();
    let length = reader.read_u32::<LittleEndian>().unwrap();
    let s0_idx = reader.read_u32::<LittleEndian>().unwrap();
    let block_size = div_ceil(length, 8);
    (&mut reader)
      .take(block_size.into())
      .read_to_end(&mut buf)
      .unwrap();

    let block = CompressedBlock {
      length,
      s0_idx,
      buf,
```

```
            };

            channel.send((block_n, block)).unwrap();
        }

        drop(channel);
    }
    fn write_to_file(channel: Receiver<UncompressedBlock>, mut writer: BufWriter<File>) {
        let mut block_n = 1;
        let mut deferred_blocks: HashMap<u32, Vec<u8>> = HashMap::new();
        while let Ok(block) = channel.recv() {
            if block.n == block_n {
                writer.write_all(&block.buf).unwrap();
                block_n += 1;
                while let Some(b) = deferred_blocks.remove(&block_n) {
                    writer.write_all(&b).unwrap();
                    block_n += 1;
                }
            } else {
                deferred_blocks.insert(block.n, block.buf);
            }
        }
    }
```