**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(национальный исследовательский университет)»

**Институт (Филиал)** №8 “Компьютерные науки и прикладная математика” **Кафедра** 806

**Группа** М80-407Б-18  **Направление подготовки** 01.03.02 Прикладная математика и информатика

**Профиль** Информатика

**Квалификация бакалавр**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**БАКАЛАВРА**

На тему: Алгоритмы факторизации больших чисел

Автор ВКРБ Гамов Павел Антонович\_\_\_\_\_\_\_\_\_\_ (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Руководитель Ухов Петр Александрович (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Консультант (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Консультант (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Рецензент (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

**К защите допустить**

Заведующий кафедрой 806 Крылов Сергей Сергеевич (\_\_\_\_\_\_\_\_\_)

(№ каф) (фамилия, имя, отчество полностью)

\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_\_г.

Москва 2022

РЕФЕРАТ

Выпускная квалификационная работа бакалавра содержит: 24 страницы, 6 рисунков, 10 использованных источников, 1 приложение.

ФАКТОРИЗАЦИЯ, МЕТОД ПОЛЛАРДА, КВАДРАТИЧНОЕ РЕШЕТО, КРИПТОГРАФИЯ

Цель работы: исследование и анализ различных методов факторизации чисел. Программная реализация ρ-метода Полларда и алгоритма квадратичного решета, сравнение их между собой, а также поиск возможных улучшений каждого из алгоритмов.

В работе рассматривается программная реализация ρ-метода Полларда, его возможных улучшений, программная реализация метода квадратичного решета, его улучшений, а также сравнения данных методов между собой.

Разработаны программные последовательная и параллельная реализации ρ-метода Полларда, программная реализация алгоритма квадратичного решета и улучшения этапов его работы.

СОДЕРЖАНИЕ

Стр.

ВВЕДЕНИЕ………………………………………………………………..……...4

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ
   1. Основные обозначения и понятия……………...………….………….6
   2. Алгоритм Полларда………………………………..…………………..7
   3. Алгоритм квадратичного решета……………………….……………..8
2. ПРАКТИЧЕСКАЯ ЧАСТЬ
   1. Реализация алгоритма Полларда……………………….…………….15
   2. Реализация алгоритма квадратичного решета………………………16
   3. Ускорение квадратичного решета…………………………………...19

ЗАКЛЮЧЕНИЕ………………………………………………………………….22

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ…..………….…………….23

ПРИЛОЖЕНИЕ А Код проекта……………………………………..………….24

ВВЕДЕНИЕ

В настоящее время в любых сетях используется шифрование информации для защиты от вмешательства третьих лиц. Для шифрования используются различные криптосистемы, такие как RSA. Тот факт, что задача факторизации больших чисел есть задача предполагаемой большой вычислительной сложности, лежит в основе различных криптосистем. Факторизация или разложение на множители целых чисел – одна из древнейших проблем теории чисел. Методы факторизации целых чисел затрагивают такие области математики, как теория чисел, модульная арифметика, решение матричных уравнений. В настоящей работе рассматриваются и анализируются некоторые известные методы факторизации. Приводится их сравнение между собой, рассматриваются области применения. При решении задачи факторизации затрагивается ряд других задач, непосредственно связанных с ней, таких как тесты на простоту и генерация больших простых чисел.

Тема работы: алгоритмы факторизация больших чисел.

Цель работы: рассмотреть алгоритмы факторизации целых натуральных чисел, показать области применения каждого алгоритма, сравнить их скорость работы между собой. Так же, модернизировать алгоритмы с целью их ускорения, показать возможные пути дальнейшего улучшения, привести математическую оценку скорости работы тех или иных улучшений.

Актуальность: алгоритмы факторизации чисел – достаточно узконаправленная сфера математики. На данный момент, когда в методах криптографии используются достаточно длинные открытые и закрытые ключи шифрования, не каждый алгоритм способен дешифровать их самостоятельно, потребуется огромная вычислительная мощность, ресурсы многих вычислительных машин, а также много компьютерного времени работы. Но тем не менее, работа в данной области нацелена на улучшение защиты современных методов шифрования. Никто не может сказать наверняка, получится ли найти такой алгоритм, который станет прорывным в области факторизации, с помощью которого можно будет вскрывать самые защищенные системы шифрования данных. Но если об этом говорить, популяризировать данные алгоритмы и все что с ними связано, есть шанс зацепиться за те или иные математические выкладки, найти новые пути и возможности решения данного типа задач, что приблизит человечество к поиску новых путей защиты информации.

В данной дипломной работе были рассмотрены основные методы факторизации целых чисел. Было произведено сравнение работы алгоритмов факторизации, а также были представлены улучшения, которые позволили улучшить временные затраты работы.

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ
   1. **Основные понятия**

Пусть число N целое и составное, под термином факторизация будем иметь в виду нахождение чисел отличных от единицы и самого числа, произведение которых будет давать нам искомое число N. В случае, когда таких чисел нет, число N будет считаться простым, имеющим в качестве делителя само себя и единицу. Сами по себе, делители могут быть разного вида, они могут быть длинными или короткими, но как правило, для задач факторизации, делители числа состоят из примерно одинакового количества десятичных чисел. Данное условие не дает нам просто начав с нуля, методом перебора, пройтись по цепочке в надежде получить самый малый из факторов, что облегчит поиск остальных. Так же факторизируемое число может состоять из более чем двух факторов, но в данной работе условимся, что число состоит из произведения двух факторов, которые являются простыми числами.

Гладкое число – понятие, которое будет встречаться относительно часто в разборе алгоритма квадратичного решета. Основная теорема арифметики гласит, что любое число можно разложить на множество простых чисел. Но что если множество простых чисел ограничить сверху константой, получается ограниченный набор простых чисел, так вот, если число можно разложить на множители используя простые числа из ограниченного набора, такое число называется B – гладким, где константа B – ограничение по размеру факторной базы, заданное пользователем. Возникает удобный способ записи гладких чисел, каждое такое число можно представить вектором степеней простых чисел, используемых в разложении, такой способ хранения чисел понадобится нам далее при работе с матрицами в алгоритме квадратичного решета.

* 1. **Алгоритм Полларда**

ρ - алгоритм — предложенный Джоном Поллардом в 1975 году алгоритм, служащий для факторизации целых чисел. Данный алгоритм основывается на алгоритме Флойда поиска длины цикла в последовательности и некоторых следствиях из парадокса дней рождения. Алгоритм наиболее эффективен при факторизации составных чисел с достаточно малыми множителями в разложении. Сложность алгоритма оценивается как O(N^(1/4)).

ρ - алгоритм Полларда строит числовую последовательность, элементы которой образуют цикл, начиная с некоторого номера n, что может быть проиллюстрировано, расположением чисел в виде греческой буквы ρ, что послужило названием семейству алгоритмов.

Алгоритм, реализующий данный метод состоит из следующих шагов.

1. Выбирается натуральное число x0 < n и строится последовательность чисел {xn}, n = 0, 1, 2,…, где каждое следующее число определяется по формуле xm+1 = (xm2 + 1) mod n.
2. На каждом i-м шаге вычисляется d = НОД(n, |xi - xj|), где j < i. Можно ограничиваться парами вида (xi - xj), где j = 2k, где k = 1, 2, 3, …, i [2^k + 1, 2^(k+1)].
3. Когда будет найдено d не равное 1, вычисления заканчиваются и d – делитель числа n (не обязательно простой), если n / d – составное, то можно проверить алгоритм для числа n / d.

Таким образом ρ – алгоритм Полларда является вероятностным методом, для которого сложность вычисления, без учета логарифмической сложности нахождения НОД, оценивается как O(q^(1/2)) <= O(n^(1/4)). То есть можно сказать, что сложность зависит от размера делителя, а не от размеров самого числа, что может быть полезно для некоторых форматов чисел. Отсюда вытекает первый минус, алгоритм непредсказуем и для какого-то числа будет работать очень быстро, в силу своей вероятности, а для других чисел, потребует излишне много ресурсов и времени. Возникает идея оптимизации данного алгоритма, запуск нескольких потоков с различными стартовыми условиями, что, конечно, не дает нам линейного выигрыша, зато позволяет получить большие шансы найти делитель числа. Вторая идея - использовать другие генерирующие многочлены.

Существует смысл использования алгоритма, зная, что факторизируемое число состоит из относительно маленьких множителей, скорость метода зависит от длинны такого числа, но что делать если число состоит из примерно равных по длине чисел, например, как числа RSA, в таком случае алгоритм может и не выдать ответ, сколько бы мы ни ждали. Нам нужна надежная оценка скорости, которая бы хоть и зависела бы от размеров числа, но для которой можно было бы найти примерное время факторизации.

* 1. **Алгоритм квадратичного решета**

Метод квадратичного решета - метод факторизации больших чисел, разработанный Померанцем в 1981 году. Долгое время превосходил другие методы факторизации целых чисел общего вида, не имеющих простых делителей, порядок которых значительно меньше корня из n. Этот метод считается вторым по быстроте (после общего метода решета числового поля). И до сих пор является самым быстрым для целых чисел до 100 десятичных цифр и устроен значительно проще чем общий метод решета числового поля. Это универсальный алгоритм факторизации, так как время его выполнения исключительно зависит от размера факторизуемого числа, а не от его особой структуры и свойств.

Алгоритм пытается найти такие квадраты чисел, которые равны по модулю n, что часто приводит к факторизации n. Алгоритм работает в два этапа: этап сбора данных, где он собирает информацию, которая может привести к равенству квадратов; и этап обработки данных, где он помещает всю собранную информацию в матрицу и обрабатывает её для получения равенства квадратов. Первый этап может быть легко распараллелен на много процессов, но второй этап требует большие объемы памяти и его трудно распараллелить.

В данной работе стоит разделить алгоритм не на два этапа, а на 3: формирование факторной базы, нахождение гладких чисел (просеивание), решение матрицы за которыми следует формирование ответа. Связано это с особенностями гиперпараметра B, который определяет сколько простых чисел будет задействовано на втором этапе просеивания. В дальнейшем будет рассказано о данной оптимизации поподробнее.

Первый этап алгоритма использует гиперпараметр B, который обозначает верхнюю границу факторной базы, которая будет состоять из простых чисел меньше заданного числа B. Данное множество простых чисел называется факторной базой для числа n с гиперпараметром B. Как известно, любое число можно представить в виде произведения простых чисел, числа, которые можно разложить в виде произведений простых чисел из факторной базы, называются гладкими числами по этому множеству.

Итак, у нас есть задача найти все простые числа до некой верхней планки, сразу на ум приходит алгоритм решета Эратосфена. Является базовым, но от этого не менее эффективным способом нахождения всех простых чисел до некой верхней границы, решето Эратосфена прекрасно справляется с нашей задачей. Так как в дальнейшем, нам возможно придется возвращаться к первому этапу, нам важно, чтобы алгоритм работал крайне быстро. Сложность алгоритма составляет O(n log log n).

Небольшое улучшение алгоритма поиска простых чисел, придуманное А. О. Л. Аткином и Д. Ю. Бернштайном, называется решетом Аткина. Скорость работы алгоритма соответствует скорости ранее известных алгоритмов просеивания, но требует меньше памяти и может работать на интервале. Хитрость алгоритма сводится к решению уравнений вида .

Все числа, равные (по модулю 60) 1, 13, 17, 29, 37, 41, 49 или 53, имеют остаток от деления на 4, равный 1. Эти числа являются простыми тогда и только тогда, когда количество решений уравнения 4x2 + y2 = n нечётно и само число не кратно никакому квадрату простого числа.

Числа, равные (по модулю 60) 7, 19, 31, или 43, имеют остаток от деления на 6, равный 1. Эти числа являются простыми тогда и только тогда, когда количество решений уравнения 3x2 + y2 = n нечётно и само число не кратно никакому квадрату простого.

Числа, равные (по модулю 60) 11, 23, 47, или 59, имеют остаток от деления на 12, равный 11. Эти числа являются простыми тогда и только тогда, когда количество решений уравнения 3x2 − y2 = n (для x > y) нечётно и само число n не кратно никакому квадрату простого.

Отдельный шаг алгоритма вычёркивает числа, кратные квадратам простых чисел. Так как ни одно из рассматриваемых чисел не делится на 2, 3, или 5, то, соответственно, они не делятся и на их квадраты. Поэтому проверка, что число не кратно квадрату простого числа, не включает 22, 32, и 52.

Для уменьшения требования к памяти, просеивание строится на некотором участке, равным примерно корню из верхней границы.

По оценке авторов алгоритм имеет асимптотическую сложность O(N / (log log N)) и требует O().

Конечно, можно найти все простые числа до некоторой границы и использовать их для разложения всех гладких чисел, которые мы будем встречать в дальнейшем, но это не так эффективно. Мною был разработан алгоритм нахождения специальных простых чисел, которые будут давать нам прирост скорости при просеивании значений, полученных генерирующим полиномом.

Символ Лежандра – функция, используемая в теории чисел. Является частным случаем символа Якоби. Определение его звучит примерно так: пусть а – целое число и p – простое число, отличное от 2. Символ Лежандра определяется следующим образом: , если а делится на p. , если а является квадратичным вычетом по модулю p (то есть существует такое целое x, что , но при этом а не делится на p. , если а является квадратичным невычетом по модулю p.

Существуют различные способы нахождения символа Лежандра, я использую самый простой их них, так как встроенные функции работы с длинными числами в языке Python позволяют брать модуль и целить на очень большие числа. Однако существуют и другие более быстрые способы вычисления данного символа, рекурсивные, или с помощью разложения символа Якоби.

Теперь перейдем к реализации моего ускорения, используя символ Лежандра получаем только те простые числа, для которых существует решение уравнения, то есть они являются квадратичным вычетом нашему факторизируемому числу.

Входные данные: p – нечетное простое число, n – целое число, являющееся квадратичным вычетом по модулю p.

Результат работы алгоритма: вычет R, удовлетворяющий сравнению .

1. Выделим степени двойки из p – 1, то есть пусть p – 1 = 2SQ, где Q нечетно, S больше или равняется 1.
2. Выберем произвольный квадратичный невычет z, то есть символ Лежандра для него равен -1, положим .
3. Пусть также , , M = S.
4. Выполняем цикл:
   * 1. Если , то алгоритм возвращает R.
   1. В противном случае в цикле находим наименьшее i, 0 < i < M, такое что , с помощью итерирования возведем в квадрат.
   2. Пусть и положим , , , , возвращаемся к началу цикла.

После нахождения решения сравнения R второе решение сравнения находится как p – R.

Сам код алгоритма будет приведен в приложении. Нам необходимо просто получить корни уравнения, они пригодятся в дальнейшем при просеивании. Дело в том, что данные корни дают возможность не проверять лишний раз числа, которые не делятся на какое-либо число из факторной базы, работает это так. При просеивании генерирующий многочлен дает значения из ограниченной сетки, корни, полученные для каждого из простых чисел, позволяют перескакивать с числа, которое делится, к следующему такому числу, минуя все внутренние, которые не делятся на данное простое число. Улучшение дает существенный прирост скорости, так как плотность гладких чисел может быть довольно низка. Небольшая предобработка на всех простых числах позволяет пропускать сразу множество регионов, в которых просто не существует гладких чисел.

Этап просеивания начинается с генерирующего полинома. В этом алгоритме он представляет собой полином второй степени, подставляя в него число, мы получаем возможно гладкое число, которое в дальнейшем требуется проверить.

Как правило генерирующий полином выглядит как , где m – корень из числа, которое требуется факторизовать, округленное вниз, а n – число, которое требуется факторизовать. Сразу видно недостаток данного полинома, чем дальше уходит число x в положительную или отрицательную сторону, тем более большими становятся числа и, следовательно, тяжелее найти в них те, которые являются гладкими на нашем множестве простых чисел. Как правило, результат генерирующего полинома ищется по модулю числа n, чтобы избежать очень больших чисел. Но нам может не хватить чисел, требуемых в дальнейшем для решения матрицы. В таком случае выбирается другой гиперпараметр B, ограничивающий факторную базу. Это влечет за собой последствия, гладкие числа будет найти гораздо проще, но их потребуется в разы больше. При маленькой плотности гладких чисел мы никогда не сможем найти необходимый минимум.

Второй этап алгоритма квадратичного решета – этап просеивания. Заключается в формировании интервала из чисел, сформированный с помощью генерирующего многочлена. Далее используя ускорение из теоремы Тоннели – Шенкса, мы находим смещения для каждого простого числа из факторной базы, и начинаем делить. Если какое-либо из чисел после процесса деления на все степени каждого числа из факторной базы окажется равным единице, тогда данное число оказалось гладким, и мы вносим его в отдельный массив гладких чисел. Моя реализация после просеивания участка сразу формирует матрицу гладких чисел с их разложениями. Так как каждый отдельный фрагмент интервала считается независимо от других, данный этап прекрасно можно распараллелить, в моем случае используя модуль мультипроцессинга, я выделяю рабочих по размеру доступных ядер, в моем случае 4 ядра. Фактически, данное улучшение дает существенный прирост в скорости просеивания.

Ранее было упомянуто, что нам потребуется иногда возвращаться к первому этапу, связано это с тем, что имея данные о плотности и скорости нахождения гладких чисел, мы сможем примерно оценить время, которое нам потребуется на нахождение минимального объема гладких чисел. В случае если данное время очень велико, нам следует вернутся к прошлому этапу формирования факторной базы и поменять гиперпараметр, в надежде получить лучшую оценку по времени.

В коде нет реализации возврата к первому пункту, однако все данные выводятся в консоль, и пользователь может сам оценить время и скорость работы. В случае, если время не устраивает пользователя, он в праве начать менять гиперпараметр.

Финальный третий этап работы алгоритма осуществляет формирование матрицы по модулю два разложения всех найденных гладких чисел. Каждая строка является вектором разложения найденного числа.

Решение матрицы сводится к нахождению линейно зависимых строк, путем прямого прохода алгоритмом Гаусса. Как только будет найдена совокупность чисел, которая дает полный квадрат (значения вектора будет состоять из нулей и только), алгоритм переходит к проверке полученного решения. Используя первичное разложение, формируется правая часть из критерия Ферма. Полученный вектор дает квадрат числа, чтобы получить его корень необходимо просто поделить все степени в разложении на два. Таким образом получено одно из чисел, дальнейшие операции над которым могут дать ответ факторизации.

Так как сложность алгоритма Гаусса стремится к кубу, каких-то возможных улучшений для данного этапа найти очень сложно. Изначально, решение матрицы выполнялось в среде Python, но было принято решение использовать отдельный написанный модуль на языке C++, который дал существенный прирост в скорости. Основным улучшением можно назвать использование битовых срезов. Идея такого улучшения выводится из того, что при решении матрицы, числа не превосходят единицу, таким образом, строки матрицы можно представить в виде битого представления других чисел. Нам больше не требуется суммировать каждое число отдельно, применяя битовые операции над элементами матрицы, которые мы представили как битовые маски, мы сокращаем количество операций в разы. Решение спорное, так как тратится время на перевод изначальной матрицы в другой формат, а после формирования ответа нам необходимо привести матрицу к изначальному формату.

1. ПРАКТИЧЕСКАЯ ЧАСТЬ
   1. **Реализация алгоритма Полларда**

Реализация алгоритма Полларда является чисто символической, сравнение его с квадратичным решетом не совсем корректно, мы лишь можем убедиться, что это два принципиально разных алгоритма, разного характера. Первый является экспоненциальным, легким в реализации и содержащим под собой не так много математических выкладок. Однако при сравнении на относительно малых числах, от 10 до 20 десятичных знаков, разница скорости работы не так уж и огромна.

В данной работе реализуется линейная и параллельная реализация метода Полларда на языке Python. Так как алгоритм больше является вероятностным, оценка скорости берется как среднее время множества попыток факторизовать число, так как существует вероятность того, что данный алгоритм закончит работу через малое время, обогнав квадратичное решето, что плохо отразится в статистике. Для факторизации были использованы машины с процессорами Intel core i9 3.5 ГГц 4 ядра, а также Intel core i5 2.7 ГГц 4 ядра. Было бы интересно посмотреть, как ускорится работа, если запустить алгоритм на большем количестве ядер.

Рисунок 2.1 – Время работы алгоритма Полларда

На рисунке 2.1 представлен график сравнения скорости работы алгоритма Полларда на числах от 10 до 20 десятичных знаков. График показывает, что прирост скорости составляет в среднем от 6 до 10 секунд на одну операцию факторизации.

Сравнение на большей длине чисел будет расти по экспоненте, на 40 числах можно прождать около 4 часов и так и не получить результат, все зависит от скорости работы машины и нашей удачи.

* 1. **Реализация алгоритма квадратичного решета**

Изначально данный алгоритм был реализован на курсе криптографии в качестве лабораторной работы. Опора шла на одноименную статью в Википедии, где алгоритм был приведен в виде математических выкладках, объясняющих основные этапы работы алгоритма. Уже потом, реализовав основные этапы, стало ясно видно основные недостатки каждого из шагов. Были реализованы флаги работы, позволяющие сохранять каждый из этапов в бинарный файл, так как тестирование шло на достаточно больших числах, от 60 до 80 десятичных знаков. Каждый этап в такой работе может занимать от нескольких минут до нескольких часов и бывает полезно вовремя сохранить полученные значения в файл, для того чтобы, поменяв пару параметров не приходилось начинать алгоритм с нуля.

Много раз было пересмотрен подход к организации кода, так как алгоритм требовал постоянных внедрений и оптимизаций, возможность расширения и оптимизация кода должна быть по крайней мере возможной и не требующей переписывания половины написанного кода. Таким образом код разделился на 3 части, формирование факторной базы включающий в себя поиск простых чисел и формирование корней алгоритмом Тоннели-Шенкса, просеивания и нахождения разложения гладких чисел, а также работа с матрицей и решение матричного уравнения.

На рисунке 2.2 представлена блок схема алгоритма квадратичного решета.

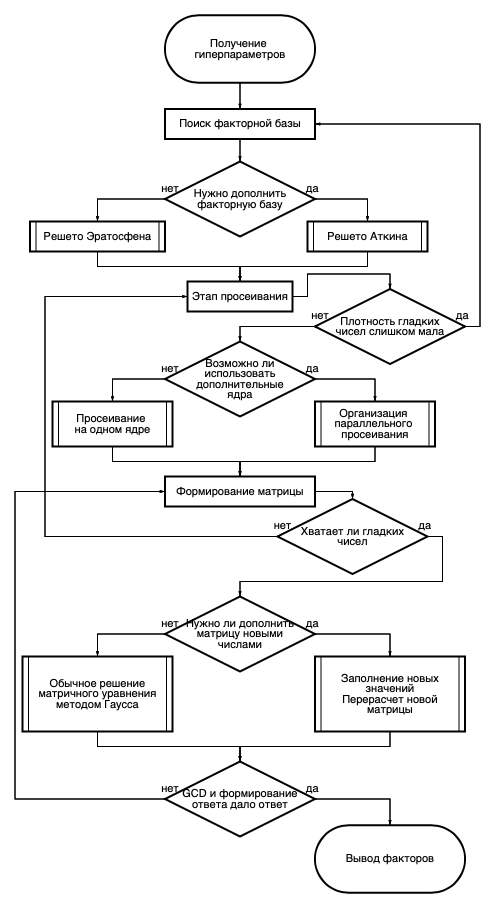


Рисунок 2.2 – блок схема работы квадратичного решета

* 1. **Ускорение квадратичного решета**

Первое ускорение достигается благодаря решету Аткина. Главное достоинство данного алгоритма проявляется, когда требуется найти простые числа не с начала, а от какой-то левой границы. Для простой генерации факторной базы можно спокойно брать алгоритм решета Эратосфена

На рисунке 2.3 приводится график скорости работ блока нахождения простых чисел для факторной базы. Сравниваются алгоритм решета Эратосфена и решета Аткина.

Рисунок 2.3 – Время работы алгоритмов поиска факторной базы

Самым долгим этапом считается просеивание и нахождение гладких чисел. Данный этап прекрасно можно считать на многих машинах параллельно, организовав такой метод работы достигается линейный прирост к скорости работы алгоритма.

На рисунке 2.4 приводится график скорости работы алгоритмов просеивания. Представлено как обычное линейное просеивание на 1 ядре процессора, а также просеивание с использованием всех доступных ядер.

Рисунок 2.4 – Скорость работы алгоритмов просеивания

И финальный этап алгоритма квадратичного решета, формирование и решение матричного уравнения. Согласно блок схеме он может закончиться неудачей, таким образом мы по цепочке должны возвращаться обратно к предыдущим этапам алгоритма.

На рисунке 2.5 приводится график скорости подходов к решению матрицы. Использование битовых срезов и организация кода на языке C++ дает потенциальный выигрыш в скорости работы.

Рисунок 2.5 – Время работы алгоритмов решения матрицы

Подводя итоги работы, удалось достигнуть прогресса в скорости работы алгоритма факторизации. Рассмотренные алгоритмы имеют свой спектр применения, но именно квадратичное решето имеет под собой больший потенциал для развития, что было показано в данной работе.

На рисунке 2.6 представлен график, на котором явно показано улучшение времени работы алгоритма.

Рисунок 2.6 – Итоговое время

ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе бакалавра были рассмотрены алгоритмы факторизации целых чисел, произведены сравнение скорости работы, а также области применения каждого из алгоритмов. Также было описаны и реализованы некоторые возможные улучшения алгоритма квадратичного решета. Сравнивая его с алгоритмом Полларда, мы получили быстрый, субэкспотенциальный, не вероятностный алгоритм, который имеет простор для дальнейшего улучшения.

В представленной реализации много, что можно сделать по-другому, написать весь код на более быстром языке программирования, например на языке C++, использовать более быстрые алгоритмы нахождения простых чисел, ускорить этап просеивания, путем распределенных вычислений или иным методом работы. Использовать Cuda ядра и перенести основные вычисления на видеокарты. Для ускорения решения матрицы в конечных полях существует алгоритм Видемана и прочие возможности для улучшения скорости работы алгоритма над разряженными матрицами.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Шнайер Б. Прикладная криптография. —Москва: Триумф, 2013.
2. Коблиц Н. Курс теории чисел и криптографии. —Москва: Научное издательство «ТВП», 2001.
3. Василенко О.Н. Теоретико – числовые алгоритмы в криптографии. —М.: МЦНМО, 2003.
4. Ишмухаметов Ш.Т. Методы факторизации натуральных чисел: учебное пособие —Казань: Казан. Ун., 2011.
5. Brent R. P.Some parallel algorithms for integer factorization—Proc. Fifth International Euro-Par Conference (Toulouse, France, 1-3 Sept 1999), Lecture Notes in Computer Science 1685, Springer, 1999.
6. Koblitz N.A. Course in number theory and cryptography — Springer-Verlag, 1987. — С. 180—182.
7. Gerver J.L. Factoring large numbers with a quadratic sieve. — Math. Comp., 1983. — Vol. 41. — P. 287—294.
8. Коэн А. A Course in Computational Algebraic Number Theory — 4th Print Edition — Берлин, Гейдельберг, Нью-Йорк: Springer, 2000. — 550 с. — (Graduate Texts in Mathematics).
9. Montgomery P., Silverman R. D. An FFT extension to the P-1 factoring algorithm Math. Comp.— AMS, 1990.
10. Pollard J. M. Theorems on factorization and primality testing Mathematical Proceedings of the Cambridge Philosophical Society B. J. Green — Cambridge University Press.

ПРИЛОЖЕНИЕ А

**Код проекта**

Ссылка на проект на GitHub:

https://github.com/pagamov/VKR

QR код со ссылкой на проект на GitHub представлен на рисунке A.1



Рисунок A.1 – QR код со ссылкой на проект на GitHub