

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ  
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»**  
**(НИУ «БелГУ»)**

ИНСТИТУТ ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК  
КАФЕДРА МАТЕМАТИЧЕСКОГО И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ  
ИНФОРМАЦИОННЫХ СИСТЕМ

**ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ РАСПРЕДЕЛЕНИЯ КВАДРАТОВ В  
ПРОИЗВЕДЕНИЯХ РАЗБИЕНИЙ НАТУРАЛЬНЫХ ЧИСЕЛ**

Выпускная квалификационная работа  
обучающегося по направлению подготовки 010200.62  
«Математика и компьютерные науки»  
очной формы обучения  
группы 07001303

Базарова Владимира Алексеевича

Научный руководитель:

к.т.н. доцент

Михелев Владимир Михайлович

БЕЛГОРОД, 2017

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	3
ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ.....	6
1.1. Анализ предметной области и обзор существующих алгоритмов.....	6
1.2. Разработка основной части последовательного алгоритма.....	9
1.3. Разработка функции проверки квадратов.....	16
ГЛАВА 2. РЕАЛИЗАЦИОННАЯ ЧАСТЬ .....	23
2.1. Разработка параллельного алгоритма.....	23
2.2. Описание программы.....	26
ГЛАВА 3. ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ.....	34
3.1. Тестирование программы и результаты её работы .....	34
3.2. Расчёт ускорения и эффективности.....	36
3.3. Анализ полученных результатов.....	38
ЗАКЛЮЧЕНИЕ.....	48
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	50
ПРИЛОЖЕНИЕ 1.....	51

## ВВЕДЕНИЕ

*Разбиением* натурального числа  $n$  называется представление  $n$  в виде суммы положительных целых чисел – *частей* или *элементов* разбиения. При этом их порядок, в отличие от композиций, не учитывается: два разбиения, отличающиеся только порядком слагаемых, являются равными. Разбиение принято записывать в порядке невозрастания его частей. Таким образом, разбиение с элементами  $a_1 \geq a_2 \geq \dots \geq a_k$  записывается в каноническом виде как  $\{a_1, a_2, \dots, a_k\}$ .

Общее количество разбиений натурального числа  $n$  обозначается  $p(n)$ . Все возможные разбиения небольших  $n$  легко найти простым перебором. Например, для  $n = 6$  имеем:

$\{6\}, \{5, 1\}, \{4, 2\}, \{4, 1, 1\}, \{3, 3\}, \{3, 2, 1\}, \{3, 1, 1, 1\}, \{2, 2, 2\}, \{2, 2, 1, 1\}, \{2, 1, 1, 1, 1\}, \{1, 1, 1, 1, 1, 1\}$ .

Таким образом,  $p(6) = 11$ . Однако при возрастании  $n$  перебор вручную становится неэффективным: получение  $p(24) = 1575$  или  $p(45) = 89134$  разбиений без помощи компьютера требует расхода значительного количества времени и усилий. Вместе с тем задачи, связанные с разбиениями, регулярно встречаются в различных областях математики, в первую очередь в комбинаторике и теории чисел.

Одной из актуальных задач, связанных с разбиениями чисел, является вычисление рангов групп центральных единиц целочисленных групповых колец знакопеременных групп, используя разбиения числа. Известно [6], что ранг группы центральных единиц целочисленного группового кольца знакопеременной группы  $A_n$  равен количеству разбиений числа  $n$ , удовлетворяющих следующим свойствам:

- 1) Все части разбиения нечётны;
- 2) Части разбиения попарно различны;
- 3)  $(n - k)$  делится на 4, где  $k$  – количество частей разбиения;

4) Произведение всех частей разбиения не является квадратом натурального числа.

Введём следующие функции для натурального аргумента  $n$ :

- $r(n)$  – количество разбиений числа  $n$ , удовлетворяющих условиям 1) – 2);
- $r_4(n)$  – количество разбиений числа  $n$ , удовлетворяющих условиям 1) – 3);
- $rank(n)$  – количество разбиений числа  $n$ , удовлетворяющих условиям 1) – 4);
- $sqrs(n)$  – количество разбиений числа  $n$ , удовлетворяющих условиям 1) – 3), но не удовлетворяющих условию 4).

На данный момент существуют алгоритмы для нахождения значений функции  $rank(n)$ , но пока нет такого, который выполняет это действие за приемлемое время. Вместе с тем, очевидно, что  $rank(n) = r_4(n) - sqrs(n)$ , то есть для расчёта рангов достаточно уметь находить значений функций  $r_4(n)$  и  $sqrs(n)$ . В данной работе будет решена задача расчёта этих функций, причём особое внимание будет уделяться функции  $sqrs(n)$  в силу того, что условие 4 самое сложное с практической точки зрения.

Основной целью данной работы является разработка параллельного алгоритма для вычисления значений функции  $sqrs(n)$ , то есть нахождение количества разбиений натурального числа  $n$  из различных нечётных натуральных слагаемых, таких, что их произведение является квадратом натурального числа, а разность между их количеством и  $n$  делится на 4.

Для достижения данной цели необходимо решить следующие задачи:

- Изучить предметную область;
- Исследовать существующие алгоритмы;
- Разработать эффективный алгоритм вычисления значений функции, определяющей количество разбиений натурального числа, состоящего из

различных нечётных частей, количество которых даёт тот же остаток при делении на 4, что и их сумма, а произведение является точным квадратом;

- Распараллелить созданный алгоритм;
- На основе данного алгоритма разработать программу для определения значений функции;
- Вычислить значения функции для достаточно малых аргументов;
- Проанализировать полученные результаты и понять характер функции.

Работа содержит введение, три главы, заключение и список использованных источников.

В первой части происходит обзор предметной области и разбираются теоретические основы для достижения поставленной цели, а именно разрабатывается эффективный последовательный алгоритм и с обоснованием правильности его работы.

Во второй части происходит распараллеливание данного алгоритма и описание программы, которая создаётся на его основе.

В третьей части производится тестирование полученной программы, фиксируются результаты её работы для некоторого множества входных данных и производится анализ полученных результатов с целью понять характер поведения функции.

В заключении подводятся итоги всей работы.

Объём работы составляет 50 страниц, объём использованной литературы – 10 источников. Работа содержит 7 таблиц, 23 рисунка, 4 формулы и 3 листинга.

## ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

### 1.1. Анализ предметной области и обзор существующих алгоритмов

Разбиение натурального числа – это его запись в виде суммы натуральных слагаемых, которые называются его частями или элементами [1]. Разбиения, совпадающие с точностью до порядка слагаемых, считаются одинаковыми. Следовательно, можно считать, что части разбиения следуют в порядке невозрастания.

Существуют различные способы точного или приближительного вычисления значений функции  $p(n)$ . Так, в 1740 году Леонард Эйлер открыл формулу производящей функции числа разбиений:

$$\sum_{n=1}^{\infty} p(n) x^n = \prod_{k=1}^{\infty} 1/(1 - x^k) \quad (1.1)$$

Для вычисления знаменателя правой части используется Пентагональная теорема Эйлера [5]:

$$\prod_{k=1}^{\infty} (1 - x^k) = \sum_{q=-\infty}^{\infty} (-1)^q x^{(3q^2+q)/2} \quad (1.2)$$

Таким образом, через эти формулы можно находить значения функции  $p(n)$  путём деления формальных степенных рядов. Рассмотрим некоторые точные значения этой функции. Они приведены в таблице 1.1.

Таблица 1. 1

Точные значения функции  $p(n)$  для некоторых  $n$

$n$	$p(n)$
1	1
2	2

Таблица 1.1. Продолжение

$n$	$p(n)$
4	5
8	22
16	231
32	8 349
50	204 226
100	190 569 292
1000	24 061 467 864 032 622 473 692 149 727 991

Для оценки количества разбиений используется формула, полученная Харди и Рамануджаном [10]:

$$p(n) \sim \frac{\exp\left(\pi\sqrt{\frac{2}{3}}\sqrt{n - \frac{1}{24}}\right)}{4n\sqrt{3}} \text{ при } n \rightarrow \infty \quad (1.3)$$

Так,  $p(1000) \approx 2.44 * 10^{31}$ . Кроме того, из этой формулы следует, что функция  $p(n)$  растёт экспоненциально.

Таким образом, нахождение самого количества разбиений не представляет проблемы. Основная сложность возникает при переборе всех разбиений числа  $n$ , который в силу экспоненциальности функции  $p(n)$  уже для относительно небольших  $n$  становится слишком ресурсоёмким.

Для полного перебора всех разбиений используется алгоритм Липского [2]. Он изображён на рис. 1.1.

```

1. begin
2.   S[1] := n; R[1] := 1; d := 1; (*первое разбиение из одного числа n*)
3.   (*Вывод первого разбиения*)
4.   while S[1] > 1 do begin
5.     sum := 0; (*sum – сумма устранённых слагаемых*)
6.     if S[d] = 1 then begin (*удаление единиц из хвоста*)
7.       sum := sum + R[d]; dec (d);
8.     end;
9.     sum := sum + S[d]; dec (R[d]); l := S[d] - 1;
10.    if (R[d] > 0 then inc (d);
11.    S[d] := l; R[d] := sum div l;
12.    l := sum mod l;
13.    if l ≠ 0 then begin
14.      inc (d); S[d] := l; R[d] := 1;
15.    end;
16.    {Вывод очередного разбиения;}
17.  end;
18. end

```

Рис. 1.1. Алгоритм Липского для разбиений натурального числа

Предложенный им алгоритм позволяет получить последовательность разбиений натурального числа  $N$  в порядке, обратном лексикографическому. При этом разбиение представлено переменными  $S[1], \dots, S[d]$ , содержащими попарно различные части разбиения ( $S[1] > \dots > S[d]$ ), а также переменными  $R[1], \dots, R[d]$ :  $R[i]$  равно количеству раз, которое  $i$ -е слагаемое появляется в разбиении, и  $R[i] \neq 0$ .

Очевидным недостатком алгоритма Липского для решения задачи о нахождении значений функции  $sgrs(n)$  является перебор слишком большого числа заведомо неподходящих разбиений. Проиллюстрируем сказанное рассмотрением разбиений для числа 25, которые указаны в таблице 1.2.



Разбиения числа 25

#	$r(n)$	$r_4(n)$	$sgrs(n)$
1	{25}	{25}	{25}
2	{21, 3, 1}	{9, 7, 5, 3, 1}	–
3	{19, 5, 1}	–	–
4	{17, 7, 1}	–	–
5	{17, 5, 3}	–	–
6	{15, 9, 1}	–	–
7	{15, 7, 3}	–	–
8	{13, 11, 1}	–	–
9	{13, 9, 3}	–	–
10	{13, 7, 5}	–	–
11	{9, 7, 5, 3, 1}	–	–

Известно, что  $p(25) = 1958$ . Таким образом, из 1958 разбиений числа 25 нашлось всего 11, удовлетворяющих условиям 1-2, два, удовлетворяющих условиям 1-3, и одно, удовлетворяющее условиям 1-3, но не удовлетворяющее условию 4. Стало быть, время работы алгоритма уменьшится на несколько порядков, если перебирать не все разбиения числа  $n$ , а только те, которые удовлетворяют условиям 1-3.

Следовательно, необходимо разработать алгоритм, который будет формировать все разбиения, удовлетворяющие условиям 1-3, и функцию, которая для каждого разбиения определяет, удовлетворяет ли она условию 4. Также будет объявлена переменная для подсчёта количества благоприятных разбиений, которая, в случае выполнения необходимых условий, будет увеличиваться на 1.

Разработка этого алгоритма рассматривается далее.

## 1.2. Разработка основной части последовательного алгоритма

Как было оговорено ранее, алгоритм для получения значений функции  $sqr_s(N)$  будет состоять из двух частей: алгоритма, перебирающего все разбиения, удовлетворяющие условиям 1-3, и функции, которая для каждого разбиения проверяет выполнимость условия 4.

Множество всех возможных разбиений натурального числа  $N$  на  $k$  различных нечётных натуральных слагаемых обозначим через  $A(N, k)$ . Сами разбиения будем обозначать строчными латинскими буквами, например,  $a(N, k)$ . Поскольку для задания такого разбиения достаточно указать лишь все его элементы в порядке убывания, будем считать, что разбиение  $a(N, k)$  равно  $\{a_1, a_2, \dots, a_k\}$ , причём  $a_i > a_j$  при всех  $1 \leq i < j \leq k$ .

Введём операцию сравнения для разбиений одного числа на одинаковое количество слагаемых.

**Определение 1.** Разбиение  $a(N, k)$  больше разбиения  $b(N, k)$  (обозначается  $a(N, k) > b(N, k)$ ), если существует такое  $1 \leq i < k$ , что  $a_i > b_i$ , а  $a_j = b_j$  при всех  $j = 1, 2, \dots, i - 1$ . Аналогично,  $a(N, k) < b(N, k)$ , если существует такое  $1 \leq i < k$ , что  $a_i < b_i$ , а  $a_j = b_j$  при всех  $j = 1, 2, \dots, i - 1$ . Если  $a_i = b_i$  при всех  $i = 1, 2, \dots, k$ , то разбиения  $a(N, k)$  и  $b(N, k)$  являются равными.

Нетрудно убедиться, что операция сравнения разбиений транзитивна, то есть для любых трёх попарно различных разбиений  $a(N, k)$ ,  $b(N, k)$  и  $c(N, k)$  из  $a(N, k) > b(N, k)$  и  $b(N, k) > c(N, k)$  следует  $a(N, k) > c(N, k)$ . (Это следует непосредственно из определения сравнения разбиений). Таким образом, все разбиения множества  $A(N, k)$  могут быть упорядочены по возрастанию.

Говоря формально, примем следующие определения.

**Определение 2.** Разбиение  $a(N, k)$  называется *минимальным*, если любое другое разбиение  $b(N, k)$  больше него. Аналогично, если  $a(N, k) > b$

$(N, k)$  для любого  $b(N, k) \neq a(N, k)$ , то разбиение  $a(N, k)$  называется *максимальным*.

**Определение 3.** Разбиение  $b(N, k)$  назовём *следующим* за разбиением  $a(N, k)$  (или для разбиения  $a(N, k)$ ), если  $b(N, k) > a(N, k)$  и не существует разбиения  $c(N, k)$ , такого, что  $b(N, k) > c(N, k) > a(N, k)$ .

Таким образом, получаем план алгоритма для нахождения и записи всех разбиений множества  $A(N, k)$ . Он представлен на рис. 1.2.

1. begin
2. Ввод  $N, k$
3. Получить минимальное разбиение
4. Выписать разбиение
5. Пока не получено максимальное разбиение
6. begin
7. Получить следующее разбиение
8. Выписать разбиение
9. end
10. end

Рис. 1.2. План алгоритма для записи всех разбиений множества  $A(N, k)$

Как видно из приведённого плана, для того, чтобы использовать алгоритм, необходимо уметь получать минимальное разбиение при заданных  $N$  и  $k$ , а также находить следующее разбиение для заданного  $a(N, k)$ . Нетрудно видеть, что в любом разбиении в силу нечётности, различности и убывания его элементов будет выполняться неравенство  $a_i \geq 2(k - i) + 1$  для всех  $i = 1, 2, \dots, k$ . Поскольку  $a_1 + a_2 + \dots + a_k = N$ , суммируя неравенства по всем  $i$ , получаем  $N \geq k^2$ . Итак, если  $N < k^2$ , то разбиений, удовлетворяющих условию, не существует.

Обозначим неполное частное и остаток числа  $(N - k^2)/2$  при делении на  $k$  через  $q$  и  $r$  соответственно, то есть  $(N - k^2)/2 = qk + r$ , где числа  $q, r$  — целые неотрицательные и  $r < k$ . Докажем, что верна следующая лемма.

**Лемма 1.** Минимальное разбиение  $a(N, k)$  определяется следующим образом:  $a_i = 2(k - i) + 1 + 2q$  для  $i = r + 1, r + 2, \dots, k$  и  $a_i = 2(k - i) + 1 + 2(q + 1)$  для  $i = 1, 2, \dots, r$ .

**Доказательство.** Вначале покажем, что данное разбиение удовлетворяет условию. Очевидно, все его элементы нечётны и  $a_i > a_j$  при  $i < j$ . Сумма всех элементов равна  $\sum_{i=1}^r (2(k - i) + 1 + 2(q + 1)) + \sum_{i=r+1}^k (2(k - i) + 1 + 2q) = \sum_{i=1}^r k(2k - i + 1 + 2q) + 2r = \sum_{i=1}^r k(2k - i + 1) + 2qk + 2r = k^2 + 2(qk + r) = k^2 + 2(N - k^2)/2 = N$ . Итак, данное разбиение действительно входит в множество  $A(N, k)$ .

Теперь предположим, что существует разбиение  $b(N, k) < a(N, k)$ . Из этого по определению следует, что существует такое  $1 \leq i < k$ , что  $b_i < a_i$ , а  $a_j = b_j$  при всех  $j = 1, 2, \dots, i - 1$ . Это значит, что  $\sum_{j=1}^i b_j < \sum_{j=1}^i a_j$ . Однако  $\sum_{j=1}^k b_j = \sum_{j=1}^k a_j = N$ , то есть найдётся  $j > i$  такое, что  $b_j > a_j$ . Далее, поскольку все элементы разбиения нечётны, то  $b_j \geq a_j + 2$  и  $b_i \leq a_i - 2$ . Отсюда  $a_i \geq b_i + 2$  и  $-a_j \geq -b_j + 2$ . Стало быть,  $a_i - a_j \geq b_i - b_j + 4$ .

С другой стороны, из определения разбиения  $a(N, k)$  следует, что  $a_i - a_j$  равно  $2(j - i)$  или  $2(j - i) + 2$ . Значит,  $b_i - b_j + 4 \leq a_i - a_j \leq 2(j - i) + 2$ , то есть  $b_i - b_j \leq 2(j - i) - 2$ . Противоречие. Значит, предположение неверно и **лемма доказана**.

Таким образом, используя эту лемму, несложно составить алгоритм для нахождения минимального разбиения числа  $N$  на  $k$  различных нечётных слагаемых.

**Определение 4.** Назовём элемент  $a_i$  *уменьшаемым*, если  $a_i > 1$  при  $i = k$  или  $a_i > a_{i+1} + 2$  при  $i \neq k$ . Если элемент  $a_i$  уменьшаемый, а  $a_j$  при любом  $j > i$  не является уменьшаемым, то такой элемент  $a_i$  назовём *правым уменьшаемым*.

Очевидно, правый уменьшаемый элемент существует в любом разбиении, в котором есть хотя бы один уменьшаемый элемент. В свою очередь, из индукции следует, что при  $k \geq 2$  уменьшаемый элемент

существует в любом разбиении, кроме разбиения, для которого  $a_i = 2*(k - i) + 1$  при всех  $i = 1, 2, \dots, k$ , то есть единственного разбиения из множества  $A(k^2, k)$ . Однако в таком множестве это разбиение, в силу своей единственности, является также и максимальным. При  $k = 1$  разбиение также единственно и, стало быть, также является максимальным.

**Определение 5.** Назовём элемент  $a_i$  *увеличиваемым*, если  $a_i < N - (k - 1)^2$  при  $i = 1$  или  $a_i < a_{i-1} - 2$  при  $i \neq 1$ . Увеличиваемый элемент  $a_i$  назовём *правым увеличиваемым*, если выполняются два свойства;

- 1)  $i < j$ , где  $a_j$  – правый уменьшаемый элемент;
- 2) Не существует такого натурального  $l$ , что  $i < l < j$  и  $a_l$  также является увеличиваемым элементом.

Аналогично, легко заметить, что для любого разбиения, не являющегося максимальным, существует ровно один правый уменьшаемый элемент.

**Лемма 2.** Пусть  $a(N, k)$  – разбиение, не являющееся максимальным. Разбиение  $b(N, k)$ , следующее за разбиением  $a(N, k)$ , задаётся следующими условиями:

- 1)  $b_i = a_i$  при всех  $i = 1, 2, \dots, l - 1$ , где  $a_l$  – правый увеличиваемый элемент;
- 2)  $b_l = a_l + 2$ ;
- 3) Разбиение  $\{b_{l+1}, b_{l+2}, \dots, b_k\}$  является минимальным разбиением числа  $N - \sum_{i=1}^l a_i$  на  $k - l$  частей.

**Доказательство.** Докажем вначале, что получаемое таким образом разбиение входит в множество  $A(N, k)$ . Действительно, все его  $k$  частей нечётны, а их сумма равна  $N$ . Для того чтобы доказать, что все его части различны, достаточно доказать, что  $b_l > b_{l+1}$ , поскольку части разбиения с  $b_1$  по  $b_l$  и с  $b_{l+1}$  по  $b_k$  упорядочены по убыванию.

По определению правого увеличиваемого элемента, найдется такое  $f > l$ , что элемент  $a_f$  – правый уменьшаемый. Рассмотрим разбиение  $\{d_1, d_2, \dots, d_k\}$ , такое, что  $d_l = a_l + 2$ ,  $d_f = a_f - 2$ , а при всех остальных  $i$  соответствующие

элементы равны ( $a_i = d_i$ ). Очевидно, что  $d_1 > d_2 > \dots > d_k$  и это разбиение входит в множество  $A(N, k)$ . Заметим, что  $d_i = b_i$  при всех  $i = 1, 2, \dots, l$ , а значит, разбиение  $\{d_{l+1}, d_{l+2}, \dots, d_k\}$  является разбиением того же числа, что и разбиение  $\{b_{l+1}, b_{l+2}, \dots, b_k\}$ . Стало быть,  $d_{l+1} \geq b_{l+1}$ , что следует из определения минимального разбиения. Итак, получаем неравенство  $b_l = d_l > d_{l+1} \geq b_{l+1}$ , то есть  $b_l > b_{l+1}$ .

Осталось доказать, что не существует такого разбиения  $c(N, k)$ , что  $a(N, k) < c(N, k) < b(N, k)$ . Предположим противное; тогда, по определению,  $c_i = b_i = a_i$  при  $i = 1, 2, \dots, l-1$ , а  $c_l$  равно либо  $a_l$ , либо  $a_l + 2$ . Если  $c_l = a_l + 2 = b_l$ , то  $\sum_{i=l+1}^k c_i = N - \sum_{i=1}^l c_i = N - \sum_{i=1}^l b_i$ , то есть разбиение  $\{c_{l+1}, c_{l+2}, \dots, c_k\}$  меньше  $\{b_{l+1}, b_{l+2}, \dots, b_k\}$  – минимального разбиения числа  $N - \sum_{i=1}^l a_i$  на  $k-l$  частей. Противоречие. Значит,  $c_l = a_l$ .

Но в таком случае, поскольку  $c(N, k) > a(N, k)$ , должно найтись такое  $j > l$ , что  $c_j > a_j$  и  $c_i = a_i$  при всех  $i = 1, 2, \dots, j-1$ . Поскольку  $\sum_{i=j}^k c_i = \sum_{i=j}^k a_i = N - \sum_{i=1}^{j-1} a_i$  и  $c_j > a_j$ , то найдется  $m > j$ , такое, что  $c_m < a_m$ . Очевидно, что в таком случае либо  $a_m$  – правый уменьшаемый элемент, либо правый уменьшаемый элемент в разбиении  $a(N, k)$  имеет номер больше  $m$ . С другой стороны, элемент  $a_j$  по определению является увеличиваемым:  $j \neq 1$  и  $a_j < a_j + 2 \leq c_j < c_{j-1} = a_{j-1}$ . Но тогда элемент  $a_l$  не является правым увеличиваемым, поскольку не выполняется свойство 2 из определения 5. Противоречие. Следовательно, исходное предположение неверно и **лемма доказана**.

Теперь, используя леммы 1 и 2, составим алгоритмы для функций *MinPart* и *NextPart*. Блок-схема первого из этих алгоритмов представлена на рис. 1.3.

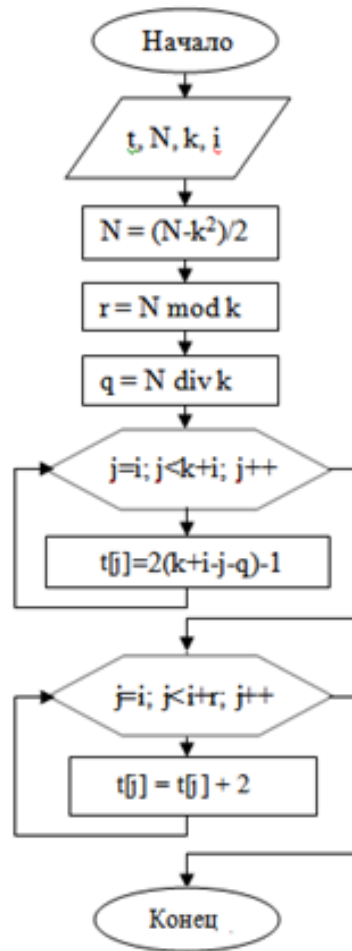


Рис. 1.3. Блок-схема функции *MinPart*

Здесь  $t$  – массив, в который записывается разбиение,  $N$  и  $k$  – число и количество частей, на которые оно разбивается, а  $i$  – номер элемента массива, с которого начинается запись разбиения (то есть части разбиения хранятся в массиве под номерами с  $i$  по  $i+k-1$ ). Функция *MinPart* принимает на вход эти 4 значения и ничего не возвращает. Результатом работы функции является присваивание элементам массива  $t[1]$ ,  $t[2]$ , ...,  $t[k]$  значений частей минимального разбиения числа  $N$  на  $k$  частей.

Следующая функция принимает на вход числа  $N$  и  $k$  и массив  $t$ , в котором содержится разбиение числа  $N$  на  $k$  слагаемых. В результате своей работы функция заменяет разбиение следующим за ним, согласно определению 3. Для этого объявляется переменная *label*, которая находит вначале индекс правого уменьшаемого элемента, а затем – индекс правого увеличиваемого элемента. После этого данный элемент массива

увеличивается на 2, а для нахождения оставшихся используется функция *MinPart*, блок-схема которой представлена на рисунке 1.3.

На рис. 1.4. представлена блок-схема функции *NextPart*.

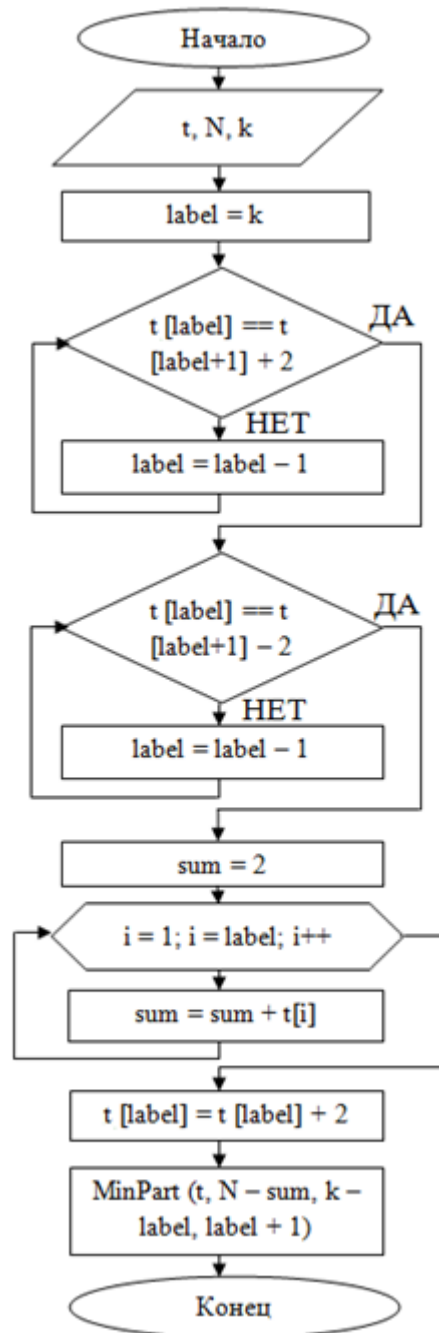


Рис. 1.4. Блок-схема функции *NextPart*

Теперь запишем алгоритм полного перебора и вычисления количества разбиений из множества  $A(N, k)$ . Он представлен на рис. 1.5.



```

1. begin
2.   Ввод N, k;
3.   Part [0] := N - (k - 1)2 + 2;
4.   Part [k+1] := -1;
5.   MinPart(Part, N, k, 1);
6.   inc(kolic);
7.   while Part [1] < N - (k - 1)2 do begin
8.       NextPart(Part, N, k);
9.       inc(kolic);
10.  end;
11. end

```

Рис. 1.5. Алгоритм расчёта количества разбиений  $A(N, k)$

В этом алгоритме для учёта количества разбиений используется переменная *kolic*.

Чтобы получить алгоритм для всех возможных  $k$ , необходимо повторить алгоритм на рис. 1.4 в цикле при изменении  $k$  с шагом 4 от минимального значения, равного  $(N-1) \bmod 4 + 1$ , до максимального, не превосходящего  $N^{0.5}$ . Последнее утверждение верно, поскольку сумма  $k$  различных нечётных слагаемых больше или  $k^2$  (что очевидным образом следует из индукции).

### 1.3. Разработка функции проверки квадратов

Полученный алгоритм позволяет перебрать все разбиения числа  $N$ , удовлетворяющие условиям 1-3. Для того чтобы получить значения функции *sqrs* ( $N$ ), необходимо разработать функцию, которая по заданным частям разбиения определяет, является ли их произведение квадратом натурального числа. Обозначим эту функцию *IsSquare*.

Самый простой способ узнать, является ли произведение натуральных чисел полным квадратом – вычислить его, а затем проверить на истинность условие  $[\text{sqr}(P)] = \text{sqr}(P)$ , где  $P$  – данное произведение,  $\text{sqr}(P)$  – квадратный корень из  $P$ , а квадратные скобки обозначают целую часть числа, то есть наибольшее целое, не превосходящее  $P$ . Однако этот метод в данной задаче не работает ввиду ограниченности максимального значения целочисленных типов данных. Так, максимальное значение типа `int` равно 2 147 483 648, а для типа `long long` это число равняется  $9\,223\,372\,036\,854\,775\,807 \approx 9 \cdot 10^{19}$ . В то же время произведение даже 20 частей разбиения, больших 40, даст результат, больший, чем  $2^{40} * 10^{20} > 10^{32}$ . Стало быть, необходимо действовать по-другому.

Будем использовать метод факторизации: каждая часть разбиения раскладывается на простые множители, после чего для каждого из этих множителей вычисляется чётность количества раз, которое он встречается.

Воспользуемся известной теоремой теории чисел: если  $n$  – составное число, то оно имеет простой делитель, не превосходящий  $\text{sqr}(n)$ . Создадим массив, в который занесём несколько первых простых чисел. Поскольку разрабатываемая программа будет вычислять значения для  $N \leq 1500$ , будет достаточно занести в массив 12 простых чисел (13-е простое число равно 41, а  $41^2 > 1500$ ). Обозначим этот массив *primes*.

Для факторизации будем использовать другой массив (обозначим его *MainArray*). Основная идея такова: вначале запишем все части разбиения в этот массив, начиная с 13-го места (с первого по 12-е места заполним нулями), а затем для всех возможных пар  $i$  и  $j$  будем нацело делить  $i$ -ю часть разбиения (записанную в массив *MainArray*) на  $j$ -й элемент массива *primes* до тех пор, пока это возможно. При этом всякий раз, когда операция деления проходит успешно и число в массиве уменьшается, будем инкрементировать  $j$ -й элемент массива *MainArray*. Очевидно, любой элемент массива *MainArray* (начиная с 13-го) после завершения над ним всех операций деления будет равен либо 1, либо простому числу, поскольку он не будет делиться ни на

одно из первых 12 простых чисел. Стало быть, после того, когда все операции завершатся, все отличные от 1 элементы массива *MainArray*, начиная с 13-го, будут являться простыми числами.

С другой стороны, при  $i \leq 12$  в  $i$ -м элементе массива будет записан показатель степени, в которой число  $p_i$  входит в разложение числа  $N$ . Если хотя бы одно из этих чисел будет нечётным, то  $N$  заведомо не является полным квадратом. Проведём соответствующую проверку, и в случае подтверждения передаём в качестве возвращаемого значения функции *false*. В противном случае необходимо проверить, чётное ли количество раз встречается каждое из оставшихся чисел, входящих в массив и не равных 1. Если это верно, возвращаем *true*, иначе – *false*.

Один из способов осуществить эту проверку состоит в следующем. В двойном цикле осуществляется перебор все пар индексов  $(i, j)$  для элементов массива, начиная с 13-го, и происходит сравниваем элементов этого массива с соответствующими индексами. В том случае, когда они равны и больше 1, присваиваем каждому из них значение 1. Если по завершении этого цикла в этой части массива нет элементов, отличных от 1, то каждое из чисел, отличных от 1, встречалось чётное количество раз. В противном случае хотя бы одно из простых чисел встречалось нечётное количество раз.

Таким образом, был описан план по созданию алгоритма для функции *IsSquare*. Теперь запишем полный алгоритм этой функции. Он представлен на рис. 1.6.

```

1. begin
2.  Ввод t, k;
3.  p := true;
4.  for (i := 0; i < 12; inc (i)) do begin
5.    fact[i] := 0;
6.  end;
7.  for (i := 12; i < 12+k; inc (i)) do begin
8.    fact[i] := t[i-11];
9.  end;
10. for (i := 0; i < 12; inc (i)) do begin
11.   for (j := 12; j < k+12; inc (j)) do begin
12.    while fact[j] mod primes[i] = 0 do begin
13.     fact[j] := fact[j]/primes[i];
14.     inc(fact[i]);
15.    end;
16.   end;
17. end;
18. for (i := 0; i < 12; inc (i)) do begin
19.   if fact[i] mod 2 = 1 then begin
20.     p := false;
21.   end;
22. end;
23. if (p = true) then begin
24.   for (i := 13; i < 12+k; inc (i)) do begin
25.     for (j := 12; j < i; inc (j)) do begin
26.       if fact[j] = fact[i] then begin
27.         fact[j] := 1;
28.         fact[i] := 1;
29.       end;
30.     end;
31.   end;
32.   for (i := 12; i < 12+k; inc (i)) do begin
33.     if fact[i] ≠ 1 then begin
34.       p := false;
35.     end;
36.   end;
37. end;
38. Вывод p;
39.end

```

Рис. 1.6. Алгоритм функции *IsSquare*

Поскольку алгоритм функции основан на трудоёмкой операции факторизации, имеет смысл проводить предварительную проверку для отсеивания разбиений, заведомо удовлетворяющих условию 4. Воспользуемся известным свойством полных квадратов: если  $p$  – нечётное число, то  $p^2 \equiv 1 \pmod{8}$ . Обозначим через  $q_1$ ,  $q_3$ ,  $q_5$  и  $q_7$  количество частей

разбиения, дающих при делении на 8 остатки 1, 3, 5 и 7 соответственно, а через  $c_1, c_3, c_5$  и  $c_7$  – соответствующие остатки этих чисел при делении на 2. Тогда, обозначая произведение частей разбиения через  $P$ , получаем

$$P \equiv 1^{q_1} * 3^{q_3} * 5^{q_5} * 7^{q_7} \equiv 3^{c_3} * 5^{c_5} * 7^{c_7} \equiv 1 \pmod{8} \quad (1.3)$$

Предпоследнее равенство следует из того, что числа 3, 5 и 7 в любой чётной степени дают остаток 1 при делении на 8. В самом деле, пусть  $r_3, r_5$  и  $r_7$  – такие числа, что  $q_3 = 2 * r_3 + c_3$ ,  $q_5 = 2 * r_5 + c_5$  и  $q_7 = 2 * r_7 + c_7$ . Тогда получаем следующее:

$$1) 3^{q_3} = 3^{(2 * r_3 + c_3)} = (3^2)^{r_3} * 3^{c_3} = 9^{r_3} * 3^{c_3} \equiv 1^{r_3} * 3^{c_3} \equiv 3^{c_3} \pmod{8}.$$

$$2) 5^{q_5} = 5^{(2 * r_5 + c_5)} = (5^2)^{r_5} * 5^{c_5} = 25^{r_5} * 5^{c_5} \equiv 1^{r_5} * 5^{c_5} \equiv 5^{c_5} \pmod{8}.$$

$$3) 7^{q_7} = 7^{(2 * r_7 + c_7)} = (7^2)^{r_7} * 7^{c_7} = 49^{r_7} * 7^{c_7} \equiv 1^{r_7} * 7^{c_7} \equiv 7^{c_7} \pmod{8}.$$

После перемножения получаем этих трёх сравнений и получаем требуемое.

Поскольку числа  $c_3, c_5$  и  $c_7$  могут равняться лишь 0 или 1, то простым перебором убеждаемся, что сравнение (1.3) истинно тогда и только тогда, когда  $c_3 = c_5 = c_7$ . В самом деле, рассмотрим все возможные случаи.

$$1) c_3 = c_5 = c_7 = 0. \text{ Тогда } 3^{c_3} * 5^{c_5} * 7^{c_7} \equiv 1 \pmod{8}.$$

$$2) c_3 = c_5 = 0, c_7 = 1. \text{ Тогда } 3^{c_3} * 5^{c_5} * 7^{c_7} \equiv 7 \pmod{8}.$$

$$3) c_3 = c_7 = 0, c_5 = 1. \text{ Тогда } 3^{c_3} * 5^{c_5} * 7^{c_7} \equiv 5 \pmod{8}.$$

$$4) c_5 = c_7 = 0, c_3 = 1. \text{ Тогда } 3^{c_3} * 5^{c_5} * 7^{c_7} \equiv 3 \pmod{8}.$$

$$5) c_3 = c_5 = c_7 = 1. \text{ Тогда } 3^{c_3} * 5^{c_5} * 7^{c_7} \equiv 1 \pmod{8}.$$

$$6) c_3 = c_5 = 1, c_7 = 0. \text{ Тогда } 3^{c_3} * 5^{c_5} * 7^{c_7} \equiv 15 \equiv 7 \pmod{8}.$$

$$7) c_3 = c_7 = 1, c_5 = 0. \text{ Тогда } 3^{c_3} * 5^{c_5} * 7^{c_7} \equiv 21 \equiv 5 \pmod{8}.$$

$$8) c_5 = c_7 = 1, c_3 = 0. \text{ Тогда } 3^{c_3} * 5^{c_5} * 7^{c_7} \equiv 35 \equiv 3 \pmod{8}.$$

Таким образом, из  $3^{c_3} * 5^{c_5} * 7^{c_7} \equiv 1$ , действительно, следует  $c_3 = c_5 = c_7$ .

Стало быть, в ходе алгоритма, после получения очередного разбиения, необходимо проверять это равенство и использовать функцию *IsSquare*

только в том случае, когда оно не выполняется. Это позволит получить достаточный выигрыш во времени при больших  $N$ .

Окончательный последовательный алгоритм приведён на рис. 1.7.

```

1. begin
2.  Ввод N;
3.   $k := (N - 1) \bmod 4 + 1$ ;
4.  while  $k^2 \leq N$  do begin
5.     $\text{Part}[0] := N - (k - 1)^2 + 2$ ;  $\text{Part}[k+1] := -1$ ;
6.    MinPart (Part, N, k, 1);
7.    inc (kolic);
8.    if IsSquare (Part, k) then begin
9.      inc (kolicsqr);
10.    end;
11.    while  $t[1] \leq N - (k - 1)^2$  do begin
12.      NextPart (Part, N, k);
13.      inc (kolic);
14.       $c3 := 0$ ;  $c5 := 0$ ;  $c7 := 0$ ;
15.      for (index := 1; index  $\leq$  k; index++) do begin
16.        if  $t[\text{index}] \bmod 8 = 3$  then begin
17.           $c3 := (c3 + 1) \bmod 2$ ;
18.        end;
19.        if  $t[\text{index}] \bmod 8 = 5$  then begin
20.           $c5 := (c5 + 1) \bmod 2$ ;
21.        end;
22.        if  $t[\text{index}] \bmod 8 = 7$  then begin
23.           $c7 := (c7 + 1) \bmod 2$ ;
24.        end;
25.      end;
26.      if  $c3 = c5 \ \& \ c7 = c5$  then begin
27.        if IsSquare (Part, k) then begin
28.          inc (kolicsqr);
29.        end;
30.      end;
31.    end;
32.     $k := k + 4$ ;
33.  end;
34. end

```

Рис. 1.7. Последовательный алгоритм

## ГЛАВА 2. РЕАЛИЗАЦИОННАЯ ЧАСТЬ

### 2.1. Разработка параллельного алгоритма

Следующим шагом после получения последовательного алгоритма является его распараллеливание. Для этого создадим алгоритм, перебирающий часть разбиений числа  $N$  на  $k$  натуральных слагаемых, в зависимости от количества частей (*size*) и номера части (*rank*).

Суть работы алгоритма в следующем. На вход подаются переменные  $N$ ,  $k$ , *size* и *rank*. После этого для каждого процесса определяются два числа, которые мы назовём *верхней* и *нижней* границами. Алгоритм работает так же, как и последовательный, но каждый процесс перебирает лишь часть разбиений, начиная с минимального из тех, максимальная часть которых не меньше нижней границы, и заканчивая самым большим из тех, максимальная часть которых не превосходит верхней границы. Таким образом, каждый процесс находит значения  $r_4(n)$  и  $sgrs(n)$  для своей части разбиений, после чего применяется операция редукции для получения окончательного ответа. Основной проблемой при таком подходе является точный подбор верхней и нижней границ. В идеальном случае они должны делить множество разбиений таким образом, чтобы в каждую из частей попадало примерно одинаковое количество разбиений.

Пусть *min* – максимальная часть минимального разбиения, а *max* – максимальная часть максимального разбиения. Очевидно, верхняя граница каждого процесса, кроме последнего, является нижней границей для следующего процесса. Таким образом, для двух процессов неизвестная граница всего одна. Обозначим её *middle*.

В данной работе опытным путём было установлено, что множество разбиений  $A(N, k)$  будет делиться примерно на две равные части, если принять  $middle = ((min*max)^{0.5} + 2*min*max/(min+max))/2$ , то есть среднее

арифметическое между средним геометрическим и средним гармоническим чисел  $min$  и  $max$ . Обозначим эту функцию через  $mid (min, max)$ . Таким образом, для двух процессов решение найдено.

В целях оптимизации эффективности работы программы было решено создать такой параллельный алгоритм, который будет работать для количества процессов, равных степени двойки (то есть  $2^n$ , где  $n$  – целое неотрицательное). Для хранения границ будет использоваться массив  $bound$  с размерностью, на 1 превышающей количество процессов. В нулевую ячейку массива записывается  $min$ , в последнюю ( $n$ -ю) –  $max$ . Число  $mid (min, max)$  записывается в ячейку ровно посередине между ними, то есть в ячейку с номером  $n/2$ . На следующем этапе заполняются ячейки ровно посередине между заполненными – то есть ячейки с номерами  $n/4$  и  $3n/4$ . Эти действия продолжаются до тех пор, пока не будут заполнены все ячейки массива. При этом нижняя и верхняя границы для процесса с номером  $i$ , очевидно, будут равны  $bound [i]$  и  $bound [i+1]$  соответственно.

Остаётся найти оптимальный алгоритм для заполнения ячеек массива  $bound$ , начиная со второго этапа, поскольку функция  $mid$  вычисляет эффективную границу только на первом шаге. Для этого была определена функция  $avg (x, y, a, b) = (a * x + b * y) / (a + b)$ . При этом для установления границы в случае  $x = min$  коэффициенты используются коэффициенты  $a = 1$  и  $b = e$ , где  $e \approx 2.71828459045$  – число Эйлера. Если  $y = max$ , то коэффициенты принимают значения  $a = 9$  и  $b = 1$  соответственно. Во всех остальных случаях  $a = b = 1$ , то есть граница устанавливается посередине.

Очевидно, что элементы массива  $bound$  зависят лишь от трёх чисел – числа  $N$ , количество разбиений которого вычисляется в программе, количества частей разбиений  $k$  и количества процессов  $size$ .

Рассмотрим алгоритм, заполняющий этот массив. Его блок-схема изображена на рис. 2.1.



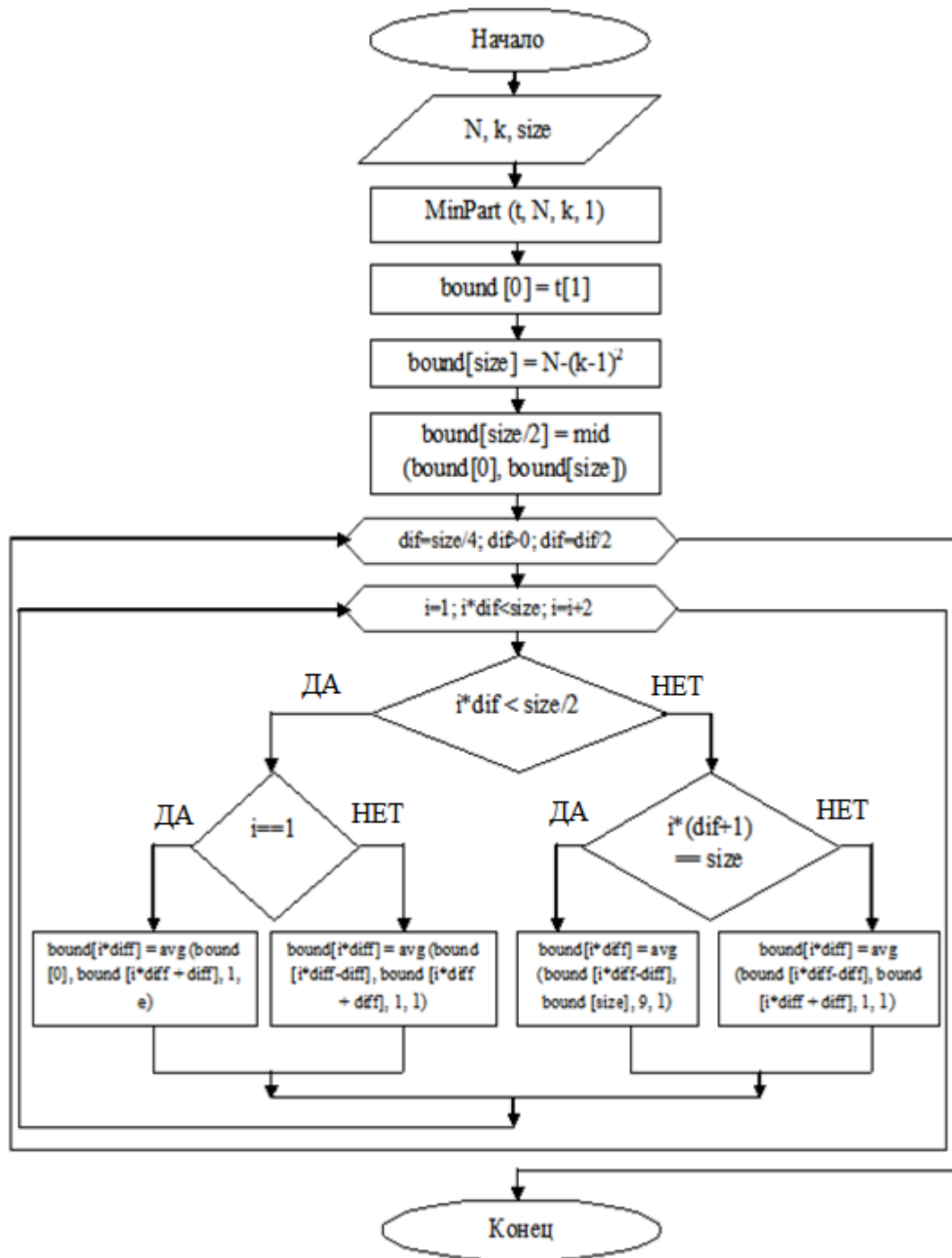


Рис. 2.1. Блок-схема алгоритма для заполнения массива *bound*

Обозначим функцию для заполнения массива *bound* (то есть для определения всех границ) через *fill*, а её параметрами являются числа *N*, *k* и количество процессов *size*. Тогда алгоритм, перебирающий часть разбиений числа *N* на *k* натуральных слагаемых, в зависимости от количества частей (*size*) и номера части (*rank*) работает следующим образом (рис. 2.2):

```

1. begin
2.   Ввод N, k, size, rank;
3.   fill (N, k, size);
4.   for (i := 0; i < size; inc (i)) do begin
5.     if rank = i then begin
6.       Part[1] := ceil (bound[i]);
7.       if Part[1] mod 2 = 0 then begin
8.         inc (Part[1]);
9.       end;
7.       MinPart (Part, N - bound [i], k - 1, 2);
8.       inc (kolic);
9.       while Part [1] < bound [i+1] do begin
10.        NextPart (Part, N, k);
11.        inc (kolic);
12.       end;
13.     end;
14.   end;
15. end

```

Рис. 2.2. Параллельный алгоритм для перебора части разбиений

Здесь *ceil* – функция, возвращающая число, равное минимальному целому, которое не меньше аргумента функции.

Чтобы получить окончательный параллельный алгоритм, необходимо инициализировать  $k$  значением  $(N-1) \bmod 4 + 1$ , а затем в цикле при изменении  $k$  с шагом 4 до тех пор, пока  $k^2$  не превосходит  $N$ , использовать алгоритм, описанный выше, в который, по аналогии с алгоритмом на рис. 1.6, нужно добавить функцию проверки квадратов.

## 2.2. Описание программы

Разработку программы было решено проводить на языке программирования C, поскольку к нему существуют стандартные «привязки» MPI, в котором будет реализована параллельная программа.

В программе используется несколько пользовательских функций, алгоритм которых описан в части 1 данной работы:

- *MinPart* – функция для получения минимального разбиения;
- *NextPart* – функция для получения следующего разбиения;
- *IsSquare* – функция для проверки данного разбиения на выполнение условия 4 (является ли произведение его всех его частей полным квадратом).

Функции *MinPart* и *NextPart* имеют тип `void`, поскольку они не возвращают значений, а лишь формируют необходимые разбиения и вычисляют их количество.

Далее представлены листинги функций *MinPart* и *NextPart*.

#### Листинг 2.1. Функция *MinPart*

```
void MinPart(int t[40], int N, int k, int i) {
N=(N-k*k)/2;
int r=N%k;
int q=N/k;
for (int j=i; j<k+i; j++) {t[j]=2*(k+i-j+q)-1;}
for (int j=i; j<i+r; j++) {t[j]+=2;}
}
```

Конец листинга

#### Листинг 2.2. Функция *NextPart*

```
void NextPart(int t[40], int N,int k) {
int metka=k, sum;
while (t[metka]==t[metka+1]+2) {metka--;}
metka--;
while (t[metka]==t[metka-1]-2) {metka--;}
sum=2;
for (int i=1; i<=metka; i++) {sum+=t[i];}
t[metka]+=2;
MinPart(t, N-sum,k-metka,metka+1);
}
```

Конец листинга

В листинге 2.3 приведен код функции *IsSquare*.

### Листинг 2.3. Функция *IsSquare*

```
bool IsSquare (int t[40], int k) {
int i, j;
bool p=true;
for (i=0; i<12; i++) {MainArray[i]=0;}
for (i=12; i<12+k; i++) {MainArray[i]=t[i-11];}
for (i=0; i<12; i++) {
    for (j=12; j<k+12; j++) {
        while (MainArray[j]%primes[i]==0) {
            MainArray[j]/=primes[i];
            MainArray[i]++;}
    }
}
for (i=0; i<12; i++) {
if (MainArray[i]%2==1) {p=false;}}
if (p) {
    for (i=13; i<12+k; i++) {
        for (j=12; j<i; j++) {
            if (MainArray[j]==MainArray[i]) {
                MainArray[j]=1;
                MainArray[i]=1;
            }
        }
    }
    for (i=12; i<12+k; i++) {
        if (MainArray[i]!=1) {p=false;}
    }
}
return p;
}
```

Конец листинга

Поскольку функция *IsSquare* даёт информацию о том, является ли произведение всех частей разбиения полным квадратом, и возвращает значения *true* или *false*, для неё выбран тип *bool*. В работе функции также используются массив *primes*, содержащий первые 12 простых чисел, и массив

*MainArray*, в котором непосредственно и происходит факторизация. Оба массива в данной программе объявлены глобально.

С целью упрощения понимания программы также была введена функция *GetPart*, которая по числам  $N$ ,  $k$ ,  $size$  и  $rank$  перебирает все разбиения  $N$  на  $k$  частей, удовлетворяющие условиям 1-3, начиная от нижней границы процесса с номером  $rank$  и заканчивая верхней границей. Также эта функция вычисляет число всех таких разбиений и количество тех из них, которые не удовлетворяют условию 4. Таким образом, она рассчитывает значения  $r_4(n)$  и  $sqr_s(n)$  и возвращает последнее из них.

Код функции *GetPart* в составе разработанной программы содержится в приложении 1.

В главной функции программы – `main` – происходит инициализация MPI с помощью `MPI_Init`. Затем устанавливается количество процессов (`size`) и каждому из них присваивается номер (`rank`) с помощью функций `MPI_Comm_size` `MPI_Comm_rank` соответственно. Функцией `MPI_Bcast` число  $N$ , введённое при запуске программы, пересылается всем процессам, после чего каждый процесс при помощи функций `MinPart` и `NextPart` перебирает свою часть разбиений и находит свою часть  $r_4(n)$  и  $sqr_s(n)$ . После этого применяется функция `MPI_Reduce`, которая собирает значения со всех процессов в нулевой и суммирует их, получая окончательный результат.

Для ввода и вывода данных используются функции форматированного ввода и вывода – `scanf` и `printf` соответственно. Функция `int printf (const char *format, arglist)` записывает в `stdout` аргументы из списка `arglist` под управлением строки, на которую указывает аргумент `format`. Эта строка включает в себя объекты, выполняющие две цели: символы, которые сами должны быть выведены на экран, и спецификаторы формата, которые определяют вид для вывода аргументов из списка. Функция `int scanf (const char *format, arglist)`, используемая для ввода, считывает данные из потока `stdin`. По форматам она аналогична функции `printf`.

Спецификаторы формата состоят из символа %, за которым следует код формата. Количество аргументов должно точно соответствовать количеству спецификаторов формата, причем следовать они должны в одинаковом порядке.

Список спецификаторов форматов для различных типов данных:

- %c – символ типа char;
- %d, %i – число типа int со знаком;
- %f – число типа double;
- %o – восьмеричное число без знака;
- %u – число типа unsigned int;
- %x – 16-ричное целое число без знака (строчные буквы);
- %X – 16-ричное целое число без знака (заглавные буквы);
- %li – тип длинного целого числа со знаком (long, long int);
- %lu – тип длинного целого числа без знака (unsigned long);
- %lli – тип двойного длинного целого числа со знаком (long long);
- %llu – тип двойного длинного целого числа без знака (unsigned long long).

С целью измерения времени в программе используется функция `MPI_Wtime`. Это функция возвращает астрономическое время в секундах, которое прошло с некоторого фиксированного момента в прошлом. Чтобы получить время работы программы, функция применяется дважды: перед блоком кода, длительность выполнения которого необходимо узнать, и сразу после него. Разность между переменными, в которые сохраняется время (из последней вычитается первая) и даст необходимый результат.

Таким образом, в программе используются следующие MPI-функции:

- `MPI_Init (&argc, &argv)` – инициализация MPI;
- `int MPI_Comm_size (MPI_Comm comm, int * size)` – определение количества процессов;
- `int MPI_Comm_rank (MPI_Comm comm, int * rank)` – задаёт каждому процессу номер;

- `int MPI_Bcast (void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)` – функция для широковещательной рассылки данных. Процесс с номером `root` рассылает сообщение типа `datatype` из `count` элементов всем процессам области связи коммутатора `comm`. Адресом начала расположения в памяти рассылаемых данных является `buffer`.

- `double MPI_Wtime()` – возвращает астрономическое время в секундах, прошедшее с некоторого фиксированного момента времени.

- `int MPI_Reduce (void *sendbuf, type *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)` – главный процесс с номером `root` собирает данные типа `datatype` из буфера `sendbuf` размерности `count` остальных процессов и проводит над ними операцию `op`. Результат возвращается в буфер `recvbuf`, который имеет те же тип данных и размерность, что и `sendbuf`.

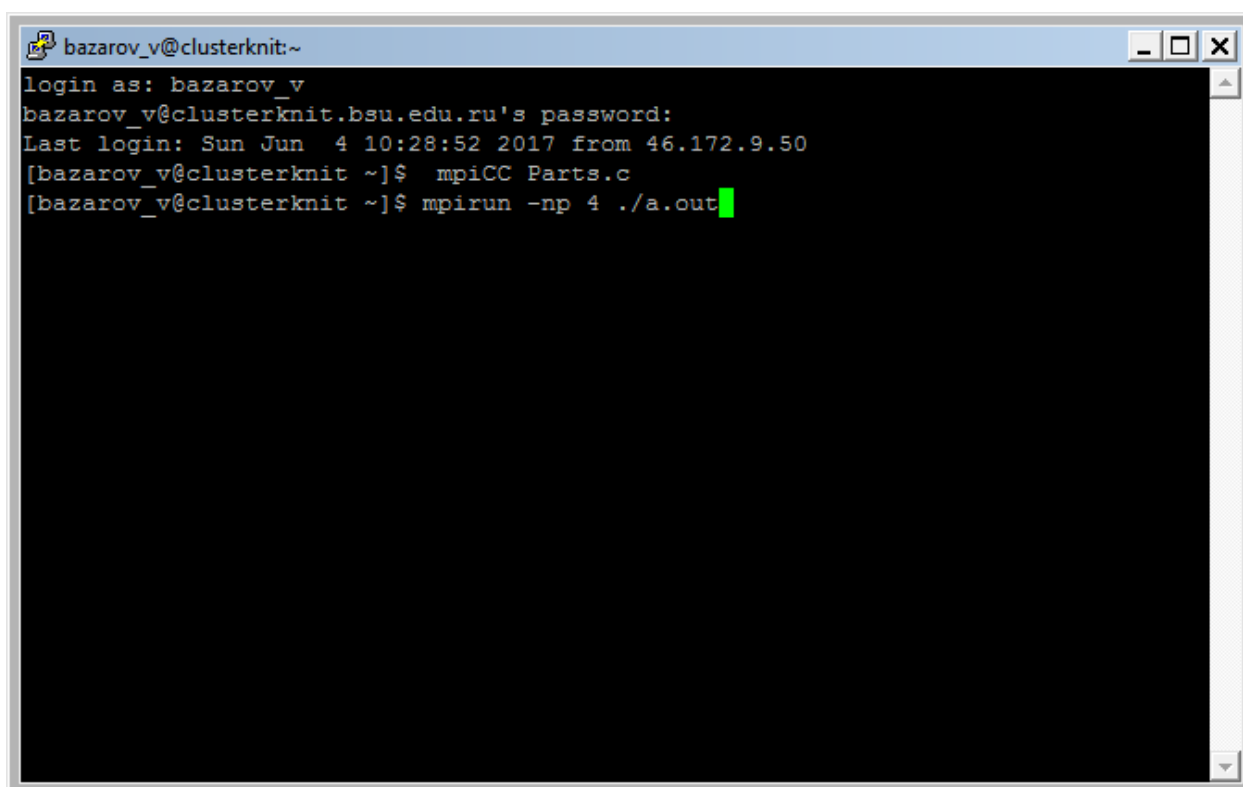
- `MPI_Finalize()` – завершает работу с MPI [8].

Переменные `kolic` и `kolicsqr`, использующиеся в программе для хранения количества разбиений  $r_4(n)$  и  $sqr_s(n)$ , имеют тип `unsigned long long`, ввиду того, что они могут принимать значения, превышающие верхнюю границу диапазона `int`. Функция `GetPart`, возвращающая значение  $sqr_s(n)$ , имеет этот же тип.

Несмотря на то, что основная программа выводит лишь значения функции  $sqr_s(n)$ , её можно использовать для получения информации о функции  $r_4(n)$ . Для этого достаточно изменить возвращаемое значение в этой функции с `kolicsqr` на `kolic`. Для получения значений функции  $r(n)$  достаточно изменить инициализацию  $k$  на  $(N-1) \bmod 2 + 1$  и усечь шаг с 4 до 2.

Поскольку программа предназначена для запуска на кластере, графический интерфейс у неё отсутствует. Перед запуском программу необходимо откомпилировать (рис. 2.3), для чего используется следующая команда:

```
- mpiCC Parts.c
```

A terminal window titled 'bazarov\_v@clusterkmit:~' with standard window controls. The terminal text shows a login sequence for 'bazarov\_v' at 'clusterkmit.bsu.edu.ru', followed by the execution of 'mpiCC Parts.c' and 'mpirun -np 4 ./a.out'.

```
bazarov_v@clusterkmit:~  
login as: bazarov_v  
bazarov_v@clusterkmit.bsu.edu.ru's password:  
Last login: Sun Jun  4 10:28:52 2017 from 46.172.9.50  
[bazarov_v@clusterkmit ~]$ mpiCC Parts.c  
[bazarov_v@clusterkmit ~]$ mpirun -np 4 ./a.out
```

Рис. 2.3. Компиляция и запуск программы

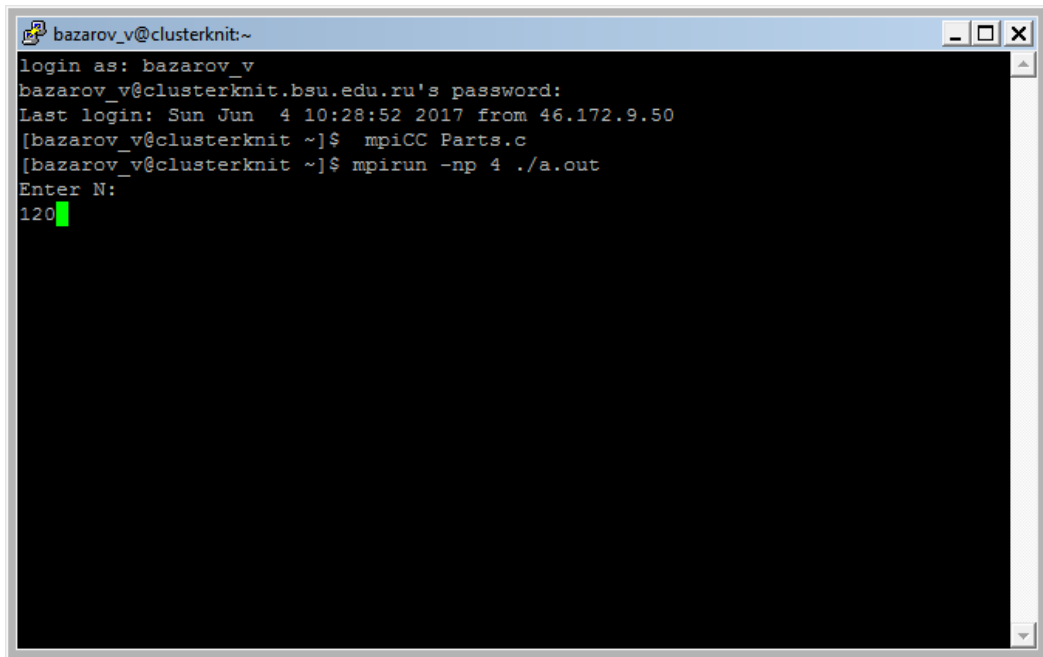
Здесь `Parts.c` – название файла программы. Для запуска программы применяется команда `mpirun`, в число параметров которой входит переменная, равная количеству процессов, на которых будет исполняться программа. Например, для запуска на 4 процессах необходимо ввести следующее:

- `mpirun -np 4 ./a.out`

Где `a.out` – имя выходного файла.

На рисунке 2.4 изображён ввод данных в программу.



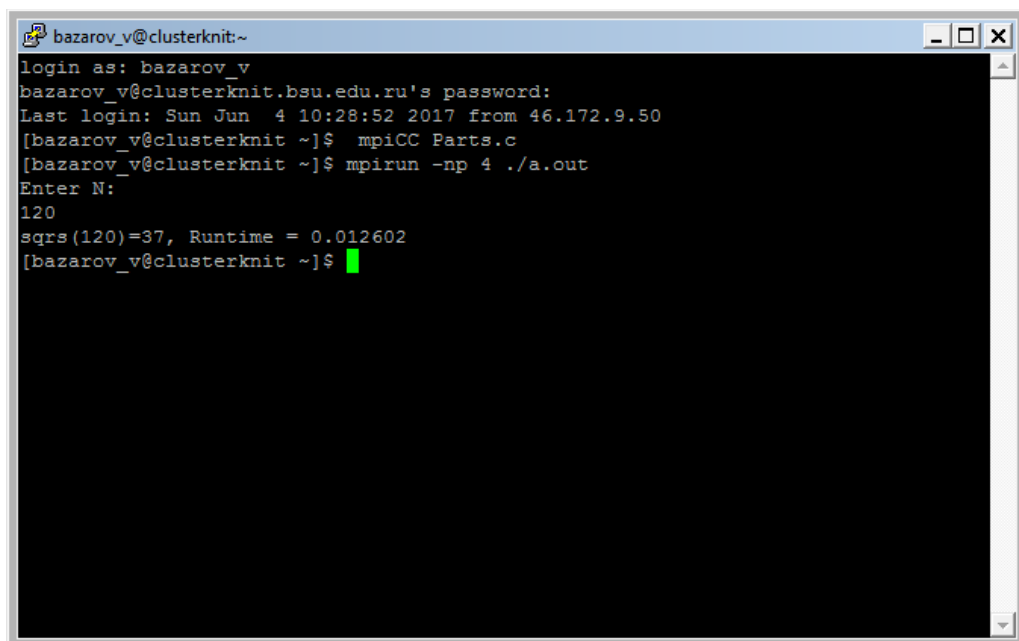


```
bazarov_v@clusterkmit:~  
login as: bazarov_v  
bazarov_v@clusterkmit.bsue.edu.ru's password:  
Last login: Sun Jun 4 10:28:52 2017 from 46.172.9.50  
[bazarov_v@clusterkmit ~]$ mpiCC Parts.c  
[bazarov_v@clusterkmit ~]$ mpirun -np 4 ./a.out  
Enter N:  
120
```

Рис. 2.4. Ввод данных в программу

После запуска программы предлагается ввести натуральное число  $N$ , являющееся аргументом функции квадратов. В данном примере рассматривается работа программы для случая  $N = 120$ .

Программа вычисляет и выводит значение функции  $sgrs(N)$ , а также время в секундах, которое было затрачено для получения результата с момента ввода данных. Результаты её работы представлены на рисунке 2.5.



```
bazarov_v@clusterkmit:~  
login as: bazarov_v  
bazarov_v@clusterkmit.bsue.edu.ru's password:  
Last login: Sun Jun 4 10:28:52 2017 from 46.172.9.50  
[bazarov_v@clusterkmit ~]$ mpiCC Parts.c  
[bazarov_v@clusterkmit ~]$ mpirun -np 4 ./a.out  
Enter N:  
120  
sgrs(120)=37, Runtime = 0.012602  
[bazarov_v@clusterkmit ~]$
```

Рис. 2.5. Результаты работы программы при  $N=120$

## ГЛАВА 3. ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

### 3.1. Тестирование программы и результаты её работы

Правильность алгоритма, на основе которого работает программа, была доказана в первой части данной работы. Тем не менее, было решено проверить её работу на достаточно малых значениях аргумента  $n$ .

Для этого расчёт функций  $r(n)$ ,  $r_4(n)$ ,  $sqrs(n)$  при всех  $n \leq 32$  был произведён вручную. Результаты, полученные программой, прошли проверку на предмет соответствия теоретически рассчитанным значениям. Результаты представлены в таблице 3.1.

Таблица 3.1

Тестирование программы при  $N \leq 32$

$N$	Теоретически рассчитанные значения			Результаты работы программы		
	$r(N)$	$r_4(N)$	$sqrs(N)$	$r(N)$	$r_4(N)$	$sqrs(N)$
<b>1</b>	1	1	1	1	1	1
<b>2</b>	0	0	0	0	0	0
<b>3</b>	1	0	0	1	0	0
<b>4</b>	1	0	0	1	0	0
<b>5</b>	1	1	0	1	1	0
<b>6</b>	1	1	0	1	1	0
<b>7</b>	1	0	0	1	0	0
<b>8</b>	2	0	0	2	0	0
<b>9</b>	2	1	1	2	1	1
<b>10</b>	2	2	1	2	2	1
<b>11</b>	2	1	0	2	1	0
<b>12</b>	3	0	0	3	0	0
<b>13</b>	3	1	0	3	1	0
<b>14</b>	3	3	0	3	3	0
<b>15</b>	4	3	0	4	3	0
<b>16</b>	5	1	0	5	1	0
<b>17</b>	5	1	0	5	1	0
<b>18</b>	5	4	0	5	4	0
<b>19</b>	6	5	0	6	5	0
<b>20</b>	7	2	0	7	2	0
<b>21</b>	8	1	0	8	1	0
<b>22</b>	8	5	0	8	5	0

Таблица 3.1. Продолжение

<i>N</i>	Теоретически рассчитанные значения			Результаты работы программы		
	$r(N)$	$r_4(N)$	$sqr(N)$	$r(N)$	$r_4(N)$	$sqr(N)$
<b>23</b>	9	8	1	9	8	1
<b>24</b>	11	5	1	11	5	1
<b>25</b>	12	2	1	12	2	1
<b>26</b>	12	6	1	12	6	1
<b>27</b>	14	12	0	14	12	0
<b>28</b>	16	9	0	16	9	0
<b>29</b>	17	3	0	17	3	0
<b>30</b>	18	7	1	18	7	1
<b>31</b>	20	16	2	20	16	2
<b>32</b>	23	15	2	23	15	2

Сравнение столбцов таблицы с результатами расчётов вручную и работы программы подтвердили, что на заданном наборе аргументов функций программа даёт верный результат.

В ходе исследования были получены значения функций  $r(n)$ ,  $r_4(n)$ ,  $sqr(n)$  для всех  $n \leq 600$ . Результаты вычислений представлены в таблице 3.2 для  $n = 1$  и  $n$ , кратных 50.

Таблица 3.2

## Значения функций

<i>N</i>	$r(N)$	$r_4(N)$	$sqr(N)$
<b>1</b>	1	1	1
<b>50</b>	98	26	2
<b>100</b>	2 574	1 008	2
<b>150</b>	34 037	17 436	55
<b>200</b>	312 928	172 441	453
<b>250</b>	2 257 009	1 108 211	1 267
<b>300</b>	13 663 248	6 524 506	2 588
<b>350</b>	72 267 480	37 245 270	10 410
<b>400</b>	343 112 116	172 512 720	43 862
<b>450</b>	1 490 170 896	729 111 366	112 617
<b>500</b>	6 003 604 571	3 044 709 903	220 569
<b>550</b>	22 676 522 718	11 338 825 972	639 196
<b>600</b>	80 974 290 498	40 129 477 067	2 073 662

Из таблицы видно, что каждая из функций возрастает при изменении  $n$

с шагом, равным 50. Кроме того, заметно, что при больших  $n$  выполняется приближённое равенство  $r(n) \approx 2r_4(n)$ , а  $sgrs(n)$  много меньше обеих функций. Для того чтобы оценить скорость роста функций  $r_4(N)$  и  $sgrs(N)$ , рассмотрим их значения для каждого  $n$  от 1 до 75.

Таблица 3.3

Значения функций при всех  $N \leq 75$ 

$N$	$r(N)$	$r_4(N)$	$sgrs(N)$	$N$	$r(N)$	$r_4(N)$	$sgrs(N)$	$N$	$r(N)$	$r_4(N)$	$sgrs(N)$
<b>1</b>	1	1	1	<b>26</b>	12	6	1	<b>51</b>	107	49	0
<b>2</b>	0	0	0	<b>27</b>	14	12	0	<b>52</b>	117	84	0
<b>3</b>	1	0	0	<b>28</b>	16	9	0	<b>53</b>	125	71	0
<b>4</b>	1	0	0	<b>29</b>	17	3	0	<b>54</b>	133	39	3
<b>5</b>	1	1	0	<b>30</b>	18	7	1	<b>55</b>	144	59	3
<b>6</b>	1	1	0	<b>31</b>	20	16	2	<b>56</b>	157	108	7
<b>7</b>	1	0	0	<b>32</b>	23	15	2	<b>57</b>	168	102	6
<b>8</b>	2	0	0	<b>33</b>	25	6	1	<b>58</b>	178	58	3
<b>9</b>	2	1	1	<b>34</b>	26	8	1	<b>59</b>	192	72	2
<b>10</b>	2	2	1	<b>35</b>	29	21	1	<b>60</b>	209	136	2
<b>11</b>	2	1	0	<b>36</b>	33	23	0	<b>61</b>	223	142	1
<b>12</b>	3	0	0	<b>37</b>	35	11	0	<b>62</b>	236	86	0
<b>13</b>	3	1	0	<b>38</b>	37	10	1	<b>63</b>	255	90	1
<b>14</b>	3	3	0	<b>39</b>	41	27	2	<b>64</b>	276	170	6
<b>15</b>	4	3	0	<b>40</b>	46	34	1	<b>65</b>	294	193	6
<b>16</b>	5	1	0	<b>41</b>	49	19	0	<b>66</b>	312	126	1
<b>17</b>	5	1	0	<b>42</b>	52	13	0	<b>67</b>	335	113	0
<b>18</b>	5	4	0	<b>43</b>	57	33	0	<b>68</b>	361	208	0
<b>19</b>	6	5	0	<b>44</b>	63	47	0	<b>69</b>	385	256	1
<b>20</b>	7	2	0	<b>45</b>	68	31	0	<b>70</b>	408	180	2
<b>21</b>	8	1	0	<b>46</b>	72	18	3	<b>71</b>	437	145	5
<b>22</b>	8	5	0	<b>47</b>	78	40	5	<b>72</b>	471	254	11
<b>23</b>	9	8	1	<b>48</b>	87	64	3	<b>73</b>	501	334	10
<b>24</b>	11	5	1	<b>49</b>	93	48	2	<b>74</b>	530	253	4
<b>25</b>	12	2	1	<b>50</b>	98	26	1	<b>75</b>	568	190	2

Полученные результаты подробно рассмотрены в 3.3.

### 3.2. Расчёт ускорения и эффективности

В ходе выполнения работы были произведены замеры времени работы последовательной и параллельной программ для различного количества

процессов и разных начальных данных. Также в каждом случае было вычислено отношение первой величины – длительности выполнения последовательной программы - ко второй.

Для того чтобы избежать большого влияния статистической погрешности на время выполнения программы, были выбраны значения  $N \geq 288$ . С другой стороны, максимальное значение  $N$ , которое использовалось для подсчётов, равно 500, поскольку при достаточно больших  $N$  время выполнения последовательной программы вызывает затруднения.

Результаты измерений помещены в таблицу 3.4.

Таблица 3.4

Время выполнения последовательной и параллельной программ

$N$	1 процесс	2 процесса		4 процесса		8 процессов	
	$T_1, c$	$T_2, c$	$T_1/T_2$	$T_4, c$	$T_1/T_4$	$T_8, c$	$T_1/T_8$
288	2.85	1.45	1.97	0.83	3.43	0.46	6.20
300	3.14	1.65	1.94	0.84	3.74	0.47	6.68
320	8.83	4.46	1.97	2.52	3.50	1.32	6.69
360	33.57	17.18	1.95	9.10	3.69	5.16	6.50
400	122.35	64.50	1.90	34.12	3.59	19.52	6.26
450	544.90	279.80	1.95	142.7	3.82	87.92	6.20
480	1310.56	670.17	1.96	393.4	3.33	219.1	5.98
500	2000.8	1042.1	1.92	564.7	3.54	327.6	6.11

Одним из показателей успешности параллельной программы является её ускорение, то есть отношение времени выполнения последовательной версии к времени выполнения параллельной версии. Другой показатель – эффективность, равный, по определению, отношению ускорения программы к количеству используемых процессов.

В таблице 3.4 даны ускорения программы для каждого случая. Рассмотрим среднее ускорение для каждого  $size$ , где  $size$  – количество

процессов, и вычислим среднюю эффективность. Эти данные приведены в таблице 3.5.

Таблица 3.5

## Ускорение и эффективность программы

Количество процессов	Ускорение	Эффективность
1	1	1
2	1.945	0.972
4	3.58	0.895
8	6.328	0.791

Таким образом, из таблицы видно, что ускорение возрастает с увеличением числа процессов, но максимальная эффективность достигается при  $size = 2$ . Это связано с тем, что разбиения множества  $A(N, k)$  разделить на две примерно равные группы можно с большей точностью, чем на 4, 8 и более.

### 3.3. Анализ полученных результатов

Из таблицы 3.3 видно, что функция  $r_4(n)$  не является монотонной. Однако легко доказать, что при любом натуральном  $n$  выполняется неравенство

$$r_4(n + 4) \geq r_4(n) \quad (3.1)$$

причём равенство достигается только при  $n = 1, 3, 4, 5, 8, 9, 13, 17, 21$ .

В самом деле, любое разбиение для числа  $n + 4$  можно получить из разбиения числа  $n$  увеличением максимальной части разбиения на 4. При этом оно, очевидно, будет также обладать свойствами нечётности и различности частей, а количество частей не поменяется, то есть и остаток при делении на 4 будет тем же, что у числа  $n + 4$ . Таким образом, любому

разбиению числа  $n$ , удовлетворяющему свойствам 1-3, взаимно однозначно соответствует некоторое разбиение числа  $n + 4$  с теми же свойствами, откуда и следует выполнение неравенства.

Легко также заметить, что при  $k \geq 2$  можно установить взаимно однозначное соответствие между всеми нужными разбиениями числа  $n$  и некоторыми из нужных разбиений числа  $n + 4$  и другим способом – увеличением первой и второй частей разбиения на 2. При этом будет существовать по крайней мере одно из нужных разбиений числа  $n + 4$ , которому ничего не поставлено в соответствие – разбиение, в котором первая часть принимает максимально допустимое значение. Это значит, что неравенство будет строгим. Следовательно, равенство возможно только в тех случаях, когда число  $n$  можно разложить только на одну часть (то есть, в силу свойства 3,  $n \equiv 1 \pmod{4}$ ), либо ни одного разбиения не существует ( $r_4(n) = 0$ ). Число 25 уже можно разложить на 5 частей, поэтому в первом случае получаем  $n = 1, 5, 9, 13, 17$ . Во втором случае  $n$  заведомо не превосходит 16, поэтому простым перебором с использованием таблицы 3.3 находим оставшиеся значения.

Таким образом, функция  $r_4(n)$  при изменении  $n$  с шагом 4 является монотонной и строго возрастает на всей числовой оси, кроме некоторых достаточно малых значений.

Также из таблицы 3.3 следует, что функция  $r(n)$  является монотонной при  $n \geq 2$ , причём при  $n \geq 26$  она строго возрастает.

Приведём строгое доказательство этого факта, то есть докажем, что  $r(n + 1) \geq r(n)$  при всех натуральных  $n \neq 1$ . Для этого каждому разбиению числа  $n$  из нечётных различных частей будем ставить в соответствие разбиение числа  $n + 1$  с аналогичными свойствами по следующему правилу:

- Если минимальная часть разбиения числа  $n$  не равна 1, добавляем часть, равную 1;
- Если минимальная часть разбиения числа  $n$  равна 1, то исключаем её из разбиения, а максимальную часть увеличиваем на 2.

Очевидно, что все части нового разбиения будут нечётными и попарно различными. Кроме того, разбиение числа  $n$  будет содержать часть, равную 1, тогда и только тогда, когда соответствующее ему разбиение числа  $n + 1$  такой части не содержит.

Из этого, в свою очередь, следует, что два разбиения числа  $n + 1$ , соответствующие разбиениям числа  $n$ , одно из которых содержит часть, равную 1, а другое – нет, не могут быть равными. Из правила, по которому устанавливается соответствие, также следует, что это равенство невозможно также и в случае, когда оба исходных разбиения содержат часть, равную 1, или же оба её не содержат.

Стало быть, данное правило различным разбиениям числа  $n$  ставит в соответствие различные разбиения числа  $n + 1$ . Кроме того, единственным разбиением, для которого не найдётся соответствие, очевидно, является то, в котором лишь одна часть и она равна 1, то есть при  $n = 1$ . Это верно, поскольку в остальных случаях часть, равную 1, можно исключить из разбиения, или же исключать части вообще не нужно.

Таким образом, если  $n > 1$ , то из вышесказанного следует, что каждому разбиению числа  $n$  из нечётных различных чисел будет поставлено в соответствие какое-то разбиение числа  $n + 1$  из нечётных различных чисел, причём разным разбиениям соответствуют разные. Это и означает, что  $r(n + 1) \geq r(n)$ , что и требовалось.

Чтобы лучше понять поведение функций  $r(n)$  и  $r_4(n)$ , в том числе относительно друг друга, рассмотрим их графики в одной системе координат. Они изображены на рис. 3.1.



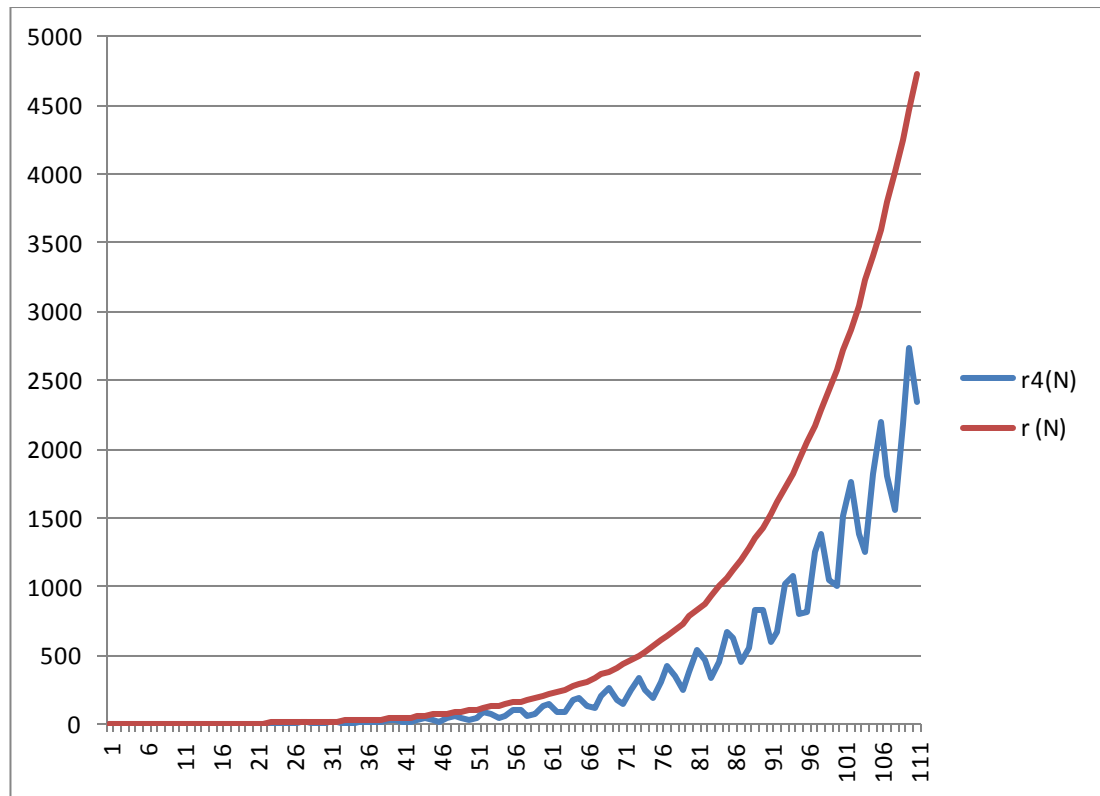
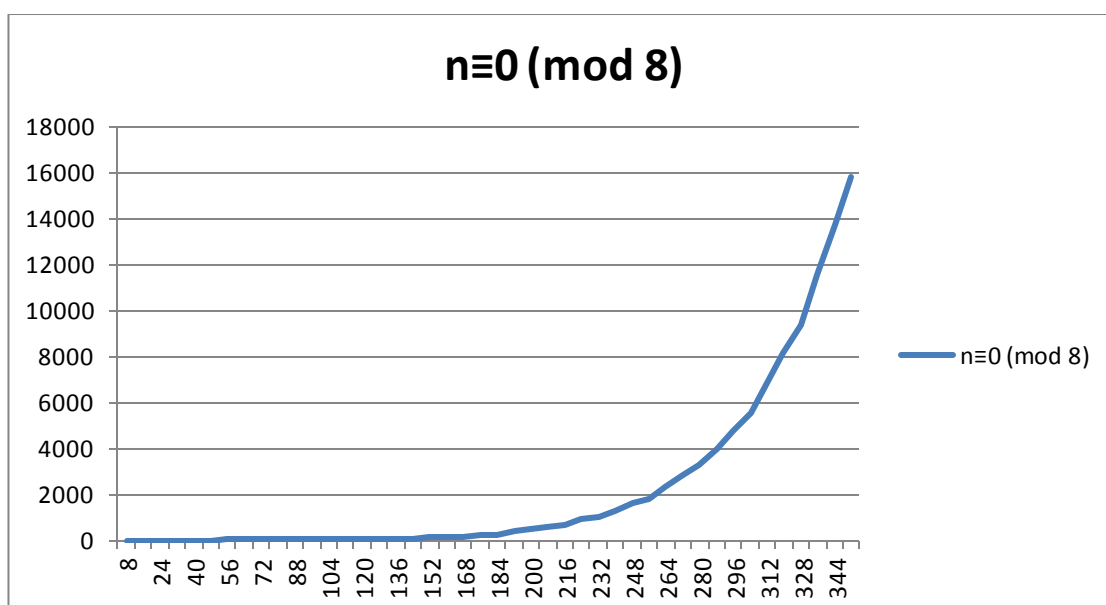


Рис. 3.1. Графики функций  $r(n)$  и  $r_4(n)$

Из графиков видно, что в среднем функция  $r_4(n)$  принимает значения, в два раза меньшие, чем  $r(n)$ .

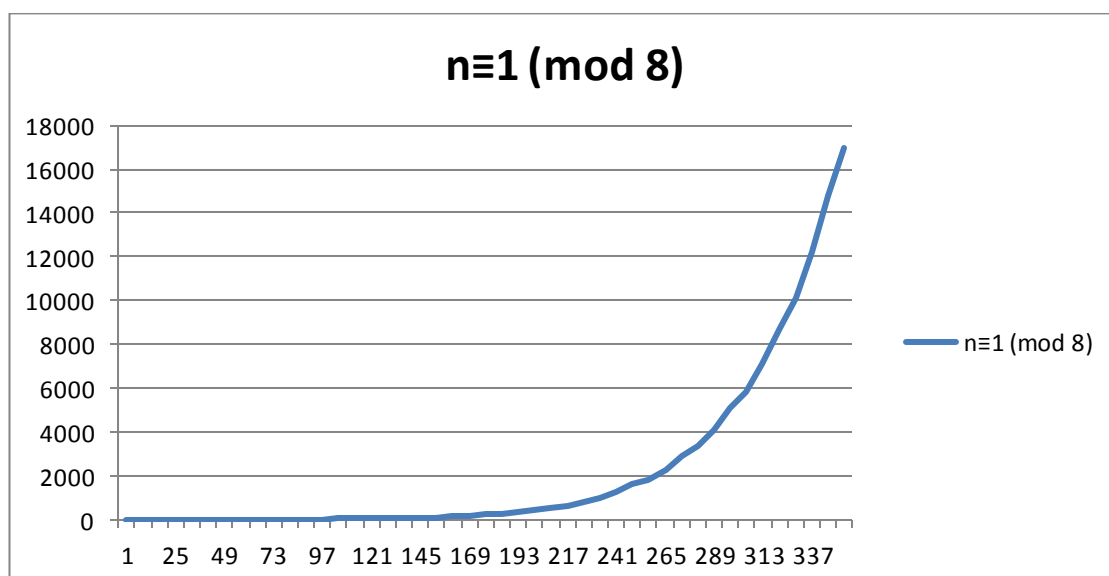
Перейдём к функции  $sqrs(n)$ . Как видно из таблицы 3.3, эта функция не является монотонной при изменении  $n$  с шагом 1, однако, по аналогии с функцией  $r_4(n)$ , будем пытаться менять  $n$  с определённым шагом для получения монотонности. Детальный анализ показывает, что такой шаг должен быть равным 8.

Рассмотрим различные графики функции  $sqrs(n)$ , соответствующих различным случаям  $n$ :  $i$ -й график отражает значения функции при  $n \equiv i - 1 \pmod{8}$ .

Рис. 3.2.  $n \equiv 0 \pmod{8}$ 

На рис. 3.2 рассмотрен случай, когда  $n$  делится на 8. Аргументы функции при этом изменяются в диапазоне от 8 до 352.

Теперь обратимся к рассмотрению случая  $n \equiv 1 \pmod{8}$  (рис. 3.3).

Рис. 3.3.  $n \equiv 1 \pmod{8}$ 

В этом случае максимальное значение, которое принимает  $n$ , равно 353. Сравнение этого графика с предыдущим позволяет сделать вывод о том, что при  $n \equiv 0 \pmod{8}$  и  $n \equiv 1 \pmod{8}$  функция  $sgrs(n)$  принимает близкие друг к другу значения.

Обратимся к случаю  $n \equiv 2 \pmod{8}$  (рис. 3.4).

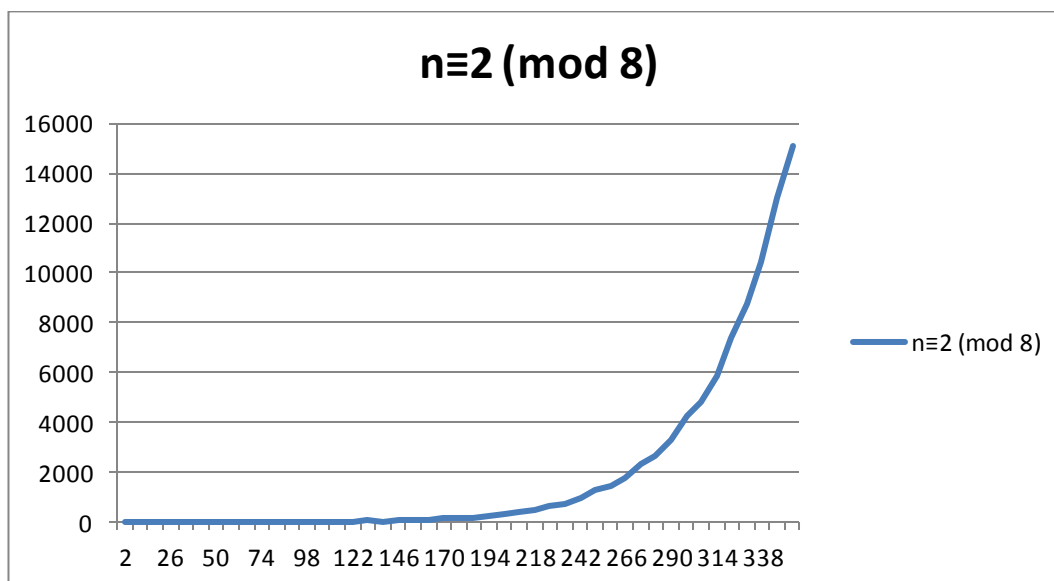


Рис. 3.4.  $n \equiv 2 \pmod{8}$

На этом графике видно, что значения функции стабильно меньше, чем в первых двух случаях, однако она по-прежнему монотонно возрастает, начиная с определённого момента.

На рис. 3.5 изображён график для  $n \equiv 3 \pmod{8}$ .

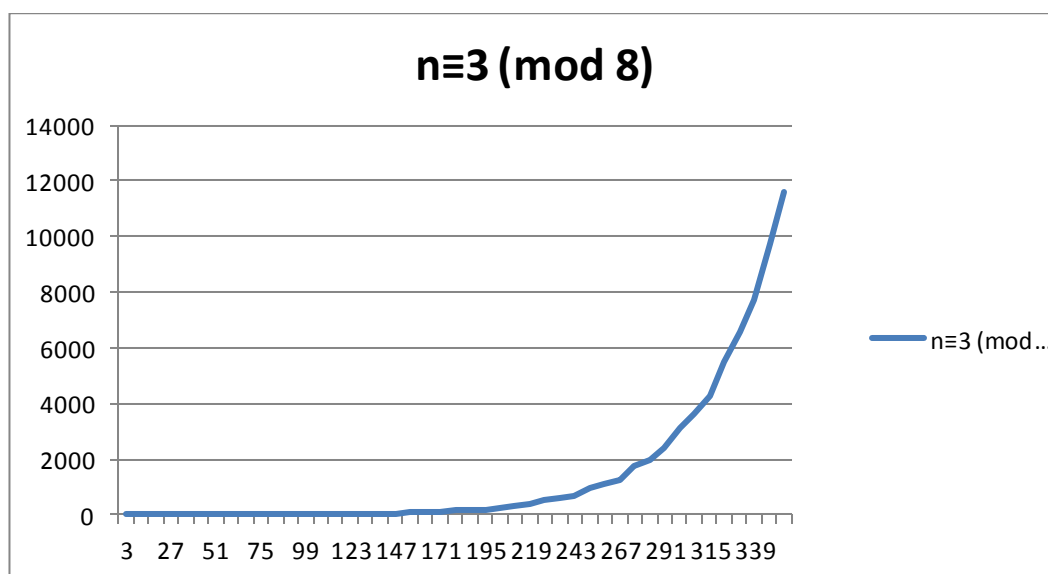


Рис. 3.5.  $n \equiv 3 \pmod{8}$

На этих двух графиках максимальное значение  $n$  равно 346 и 347 соответственно.

Перейдём к случаю  $n \equiv 4 \pmod{8}$ . Соответствующий график рассмотрен на рис. 3.6.

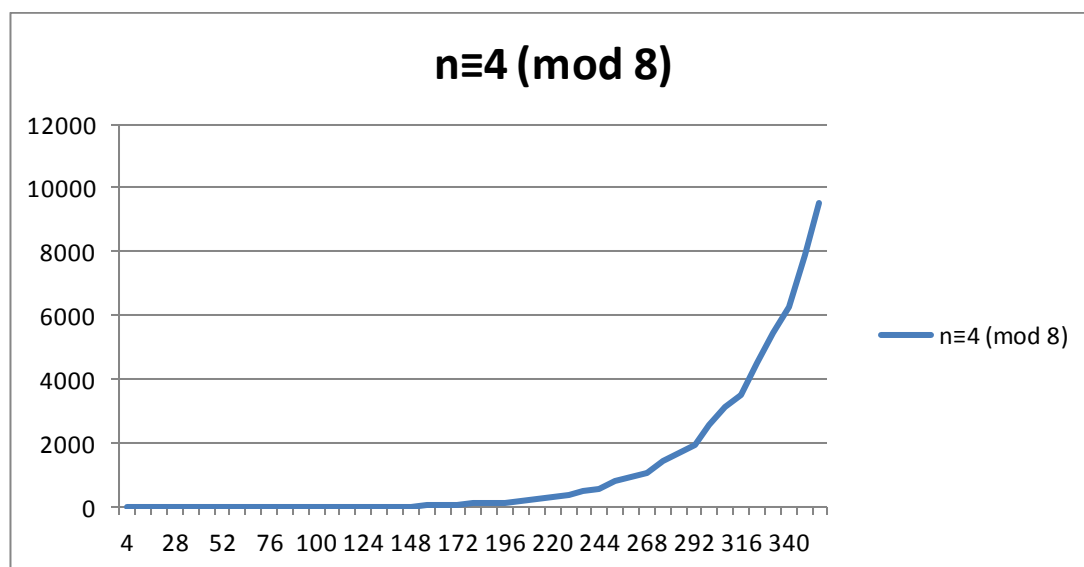


Рис. 3.6.  $n \equiv 4 \pmod{8}$

В этом случае функция принимает наименьшие значения по сравнению со всеми предыдущими ситуациями.

На рис. 3.7 приведён график для  $n \equiv 5 \pmod{8}$ . При этом максимальное значение аргумента функции уменьшено до 249.

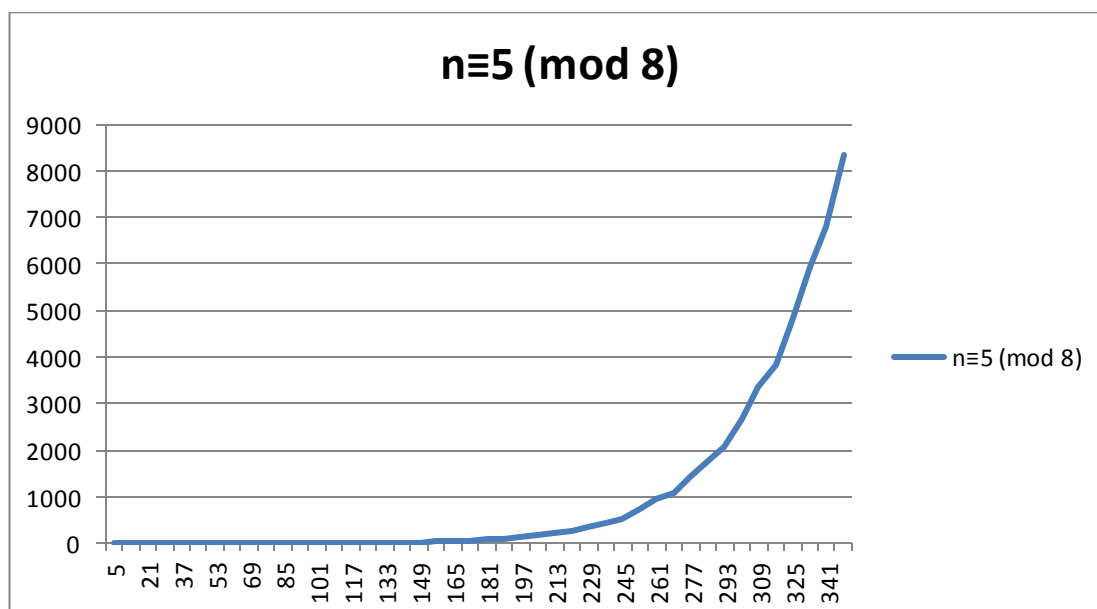


Рис. 3.7.  $n \equiv 5 \pmod{8}$

Далее на очереди  $n \equiv 6 \pmod{8}$ . Соответствующий график показан на рис. 3.8.

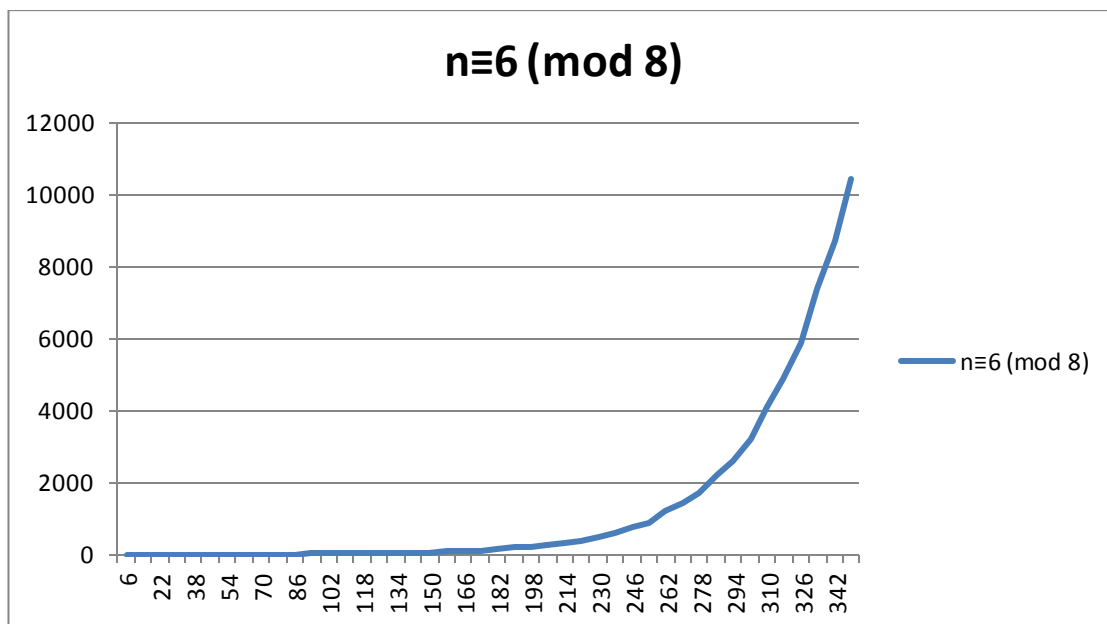


Рис. 3.8.  $n \equiv 6 \pmod{8}$

Последний график, для случая  $n \equiv 7 \pmod{8}$ , представлен на рис. 3.9.

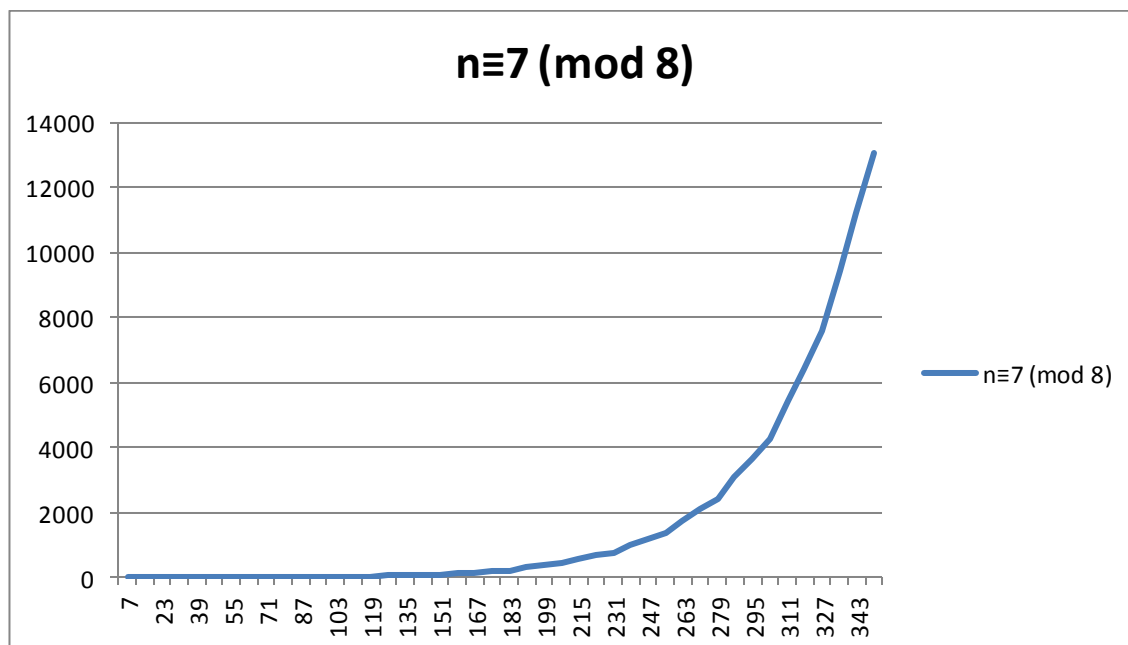


Рис. 3.9.  $n \equiv 7 \pmod{8}$

Из приведённых графиков видно, что, начиная с некоторого момента, функция строго возрастает, причём её возрастание носит экспоненциальный

характер. Точнее, функция экспоненциально возрастает при  $n \geq 173$  при изменении  $n$  с шагом 8 для любых  $n$ .

Рассмотрим теперь все 8 графиков в одной системе координат. Они представлены на рис. 3.10.

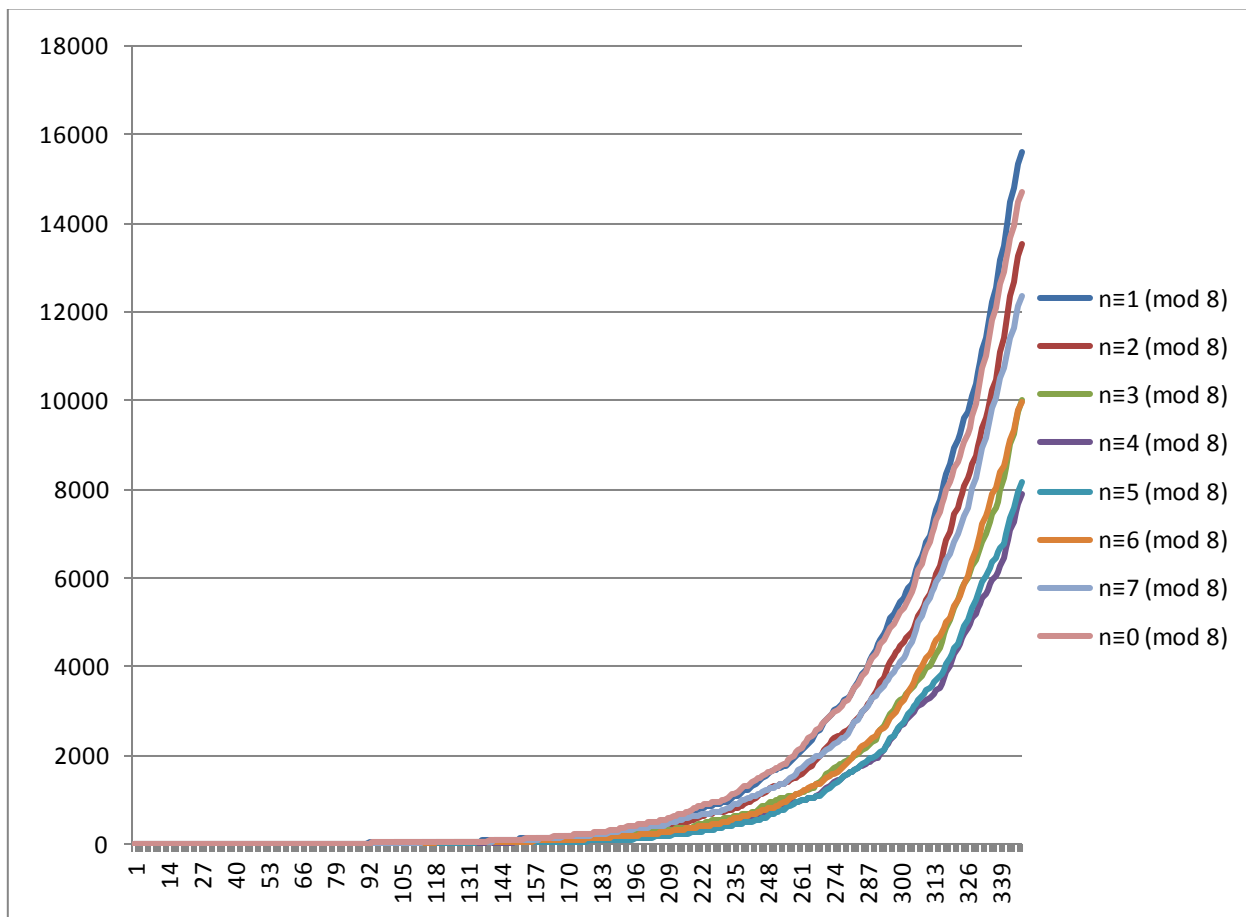


Рис. 3.10. Графики в общей системе координат

Из рисунка видно подтверждение вышесказанному: графики функций для  $n \equiv i \pmod{8}$  и  $n \equiv j \pmod{8}$  практически совпадают при таких  $i$  и  $j$ , что  $i + j \equiv 1 \pmod{8}$ . Максимальные значения функция принимает при  $i = 0$  и  $j = 1$ , минимальные – при  $i = 4$  и  $j = 5$ , а значения при  $i = 2$  и  $j = 7$  больше значений при  $i = 3$  и  $j = 6$ .

Теперь выясним, как ведёт себя функция при изменении  $n$  с шагом 1. Её график приведён на рисунке 3.11. При этом  $n$  меняется в пределах от 1 до 357.

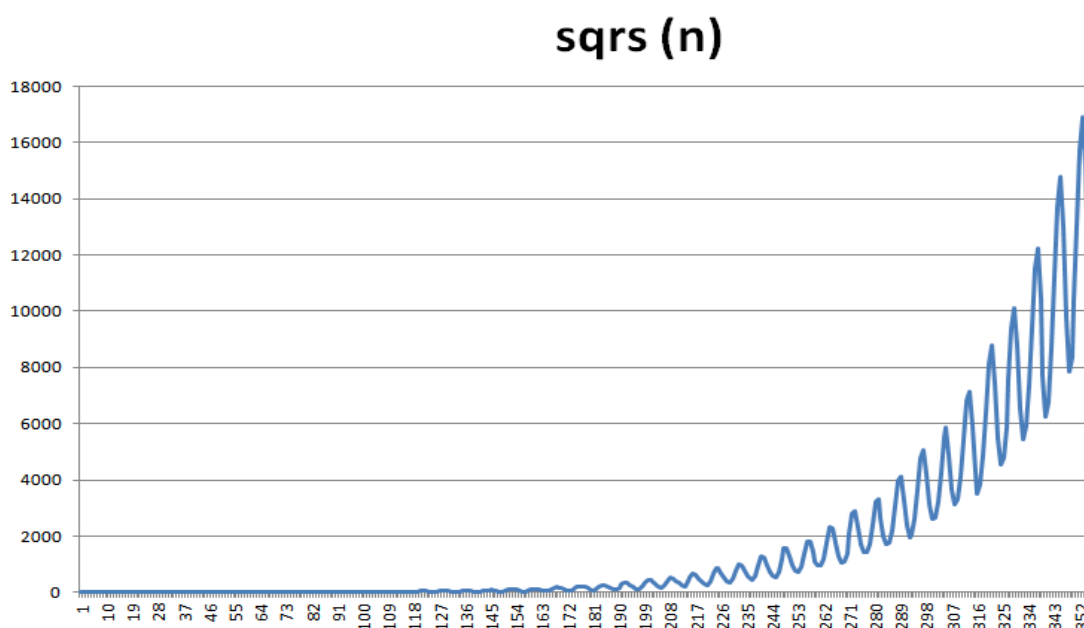


Рис. 3.11. График функции  $sqrs(n)$

Из графика видно, что отрезки возрастания и отрезки убывания чередуются у функции примерно на равных интервалах. При подробном рассмотрении выясняется, что длина отрезков монотонности функции равна от 3 до 5 для достаточно больших  $n$ , причём на отрезках  $[8k + 1; 8k + 4]$  функция строго убывает, а на отрезках  $[8k + 5; 8(k + 1)]$  – строго возрастает.

Таким образом, для функции  $sqrs(n)$  были сформулированы гипотезы о её свойствах на основе анализа значений. Для подтверждения или опровержения этих гипотез необходимы дальнейшие исследования.

## ЗАКЛЮЧЕНИЕ

В данной работе была затронута тема разбиения натуральных чисел на различные нечётные слагаемые. Особое внимание уделялось разбиениям, количество частей которых даёт тот же остаток от деления на 4, что и их сумма, а произведение частей является квадратом натурального числа. Были определены функции для количества разбиений, удовлетворяющих всем или некоторым из данных свойств.

В первой части данной работы была рассмотрена предметная область, в том числе – алгоритм Липского для порождения всех возможных разбиений натурального числа. Ввиду его непригодности для достижения цели работы возникла необходимость в разработке собственного алгоритма. Этапы разработки с необходимыми доказательствами также содержатся в теоретической части.

Во второй части был разобран один из возможных способов распараллеливания данного алгоритма, а также разработана и описана программа по данному алгоритму.

В третьей части было произведено тестирование программы и сделан окончательный вывод о правильности её работы, после чего были найдены ускорение и эффективность, в зависимости от количества процессов и аргументов функции.

Результатом работы являются разработанный параллельный алгоритм для расчёта количества квадратов в произведениях разбиений натуральных чисел, а также эффективная программа, созданная по этому алгоритму. Таким образом, основная цель работы достигнута. Также выполнены и другие задачи работы:

- Изучена предметная область;
- Исследованы существующие алгоритмы;



- Вычислены значения функции для достаточно малых аргументов;
- Проанализированные полученные результаты с целью понять характер функции.

Несмотря на полное решение всех задач работы, данные о характере поведения функции остаются гипотезами, нуждающимися в строгих математических доказательствах. Также сохраняется возможность найти более быстрый алгоритм вычисления значений функции  $sqs(n)$ , путём, например, перебора только тех разбиений, которые удовлетворяют условиям 1-3 и не удовлетворяют условию 4. Таким образом, по итогам данной работы остаются открытые вопросы, которые требуют дальнейшего изучения.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Эндрюс, Г. Теория разбиений. / Эндрюс Г. – М.: Наука, 1982 – 256 с.
2. Липский, В. Комбинаторика для программистов. / Липский, В. – М.: Мир, 1988 – 200 с.
3. Фултон, У. Таблицы Юнга и их приложения к теории представлений и геометрии. / Фултон, У. – МЦНМО, 2006 – 328 с.
4. Джеймс, Г. Теория представлений симметрических групп. / Джеймс, Г. – М.: Мир, 1982 – 214 с.
5. Вайнштейн, Ф. В. Разбиение чисел. / Вайнштейн, Ф. В. // Квант – М.: Наука, 1988 - №11, с. 19-25.
6. Ferraz, R. A. Simple components and central units in group rings. / Ferraz, R. A. // Journal of Algebra, 2004. – No 1, p. 191-203.
7. Макдональд, И. Симметрические функции и многочлены Холла. / Макдональд, И. — М.: Мир, 1985. — 224 с.
8. Корнеев В. Д. Параллельное программирование в MPI. / Корнеев В. Д. – Новосибирск, 2002 – 220 с.
9. Алгоритмы: построение и анализ. / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М.: Вильямс, 2005 – 1296 с.
10. Hardy G. H. An Introduction to the Theory of Numbers, 6<sup>th</sup> edition. / G. H. Hardy, E. M. Wright. – UK: Oxford University Press, 2008. – 656 p.

## ПРИЛОЖЕНИЕ 1

## Полный код программы

```

#include <mpi.h>
#include <stdio.h>
#include <math.h>

int MainArray[52];
int primes[12] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};

#define exp 2.71828459045

void MinPart(int t[40], int N, int k, int i) {
    N=(N-k*k)/2;
    int r=N%k;
    int q=N/k;
    for (int j=i; j<k+i; j++) {t[j]=2*(k+i-j+q)-1;}
    for (int j=i; j<i+r; j++) {t[j]+=2;}
}

void NextPart(int t[40], int N,int k) {
    int metka=k, sum;
    while (t[metka]==t[metka+1]+2) {metka--;}
    metka--;
    while (t[metka]==t[metka-1]-2) {metka--;}
    sum=2;
    for (int i=1; i<=metka; i++) {sum+=t[i];}
    t[metka]+=2;
    MinPart(t, N-sum,k-metka,metka+1);
}

bool IsSquare (int t[40], int k) {
    int i, j;
    bool p=true;
    for (i=0; i<12; i++) {MainArray[i]=0;}
    for (i=12; i<12+k; i++) {MainArray[i]=t[i-11];}
    for (i=0; i<12; i++) {
        for (j=12; j<k+12; j++) {
            while (MainArray[j]%primes[i]==0) {
                MainArray[j]/=primes[i];
                MainArray[i]++;
            }
        }
    }
    for (i=0; i<12; i++) {
        if (MainArray[i]%2==1) {p=false;}
    }
    if (p) {
        for (i=13; i<12+k; i++) {
            for (j=12; j<i; j++) {
                if (MainArray[j]==MainArray[i]) {
                    MainArray[j]=1;
                    MainArray[i]=1;
                }
            }
        }
    }
}

```

```

        for (i=12; i<12+k; i++) {
            if (MainArray[i]!=1) {p=false;}
        }
    }
    return p;
}

float mid (float min, float max) {
    float middle1 = 2*min*max/(min+max);
    float middle2 = sqrt(min*max);
    float middle = (middle1 + middle2)/2;
    return middle;
}

float avg (float min, float max, float a, float b) {
    float middle = (a*min + b*max)/(a+b);
    return middle;
}

unsigned long long GetPart (int t[40], int N, int k, int rank, int size) {
    int c3=0, c5=0, c7=0;
    unsigned long long kolic=0, kolicsqr=0;
    t[0]=N-(k-1)*(k-1)+2;
    t[k+1]=-1;
    MinPart(t, N, k, 1);
    float bound[17];
    bound[0] = t[1];
    bound[size] = N-(k-1)*(k-1);
    bound[size/2]=mid(bound[0], bound[size]);
    for (int diff=size/4; diff>0; diff/=2) {
        for (int i=1; i*diff<size; i+=2) {
            if (i*diff<size/2) {
                if (i==1) {bound[i*diff]=avg(bound[i*diff-diff], bound [i*diff+diff], 1, exp);}
                else {bound[i*diff]=avg(bound[i*diff-diff], bound [i*diff+diff], 1, 1);}
            }
            else {
                if (i*diff+diff==size) {bound[i*diff]=avg(bound[i*diff-diff], bound [i*diff+diff], 9, 1);}
                else {bound[i*diff]=avg(bound[i*diff-diff], bound [i*diff+diff], 1, 1);}
            }
        }
    }
    int minr=ceil(bound[rank]);
    if (minr%2==0) {minr++;}
    if (minr<bound[rank+1]) {
        t[1]=minr;
        MinPart(t, N-minr, k-1, 2);

        if (rank != 0) {NextPart(t, N, k);}
        c3=0;
        c5=0;
        c7=0;
        kolic++;
        for (int ind = 1; ind <= k; ind++) {
            if (t[ind]%8==3) {c3=(c3+1)%2;}
            if (t[ind]%8==5) {c5=(c5+1)%2;}
            if (t[ind]%8==7) {c7=(c7+1)%2;}
        }
        if (c3==c5 && c5==c7) {
            if (IsSquare (t, k)) {kolicsqr++;}
        }
    }
}

```

```

    }

    while (t[1] < bound[rank+1]) {
    NextPart(t, N, k);
    c3=0;
    c5=0;
    c7=0;
    kolic++;
    for (int ind = 1; ind <= k; ind++) {
        if (t[ind]%8==3) {c3=(c3+1)%2;}
        if (t[ind]%8==5) {c5=(c5+1)%2;}
        if (t[ind]%8==7) {c7=(c7+1)%2;}
    }

    if (c3==c5 && c5==c7) {
    if (IsSquare (t, k)) {kolicsqr++;}
    }
    }

return kolicsqr;
}

int main(int argc, char *argv[]) {
int myid, size, rank, N, i;

double starttime, endtime;
int Part[40];
unsigned long long kolic=0, allkolic;

MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &size);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank==0) {
printf("Enter N:\n");
scanf("%d", &N);
}
starttime = MPI_Wtime();
MPI_Bcast(&N,1,MPI_INT,0,MPI_COMM_WORLD);
for (i=0; i<size; i++) {
if (rank==i) {
for (int k=(N-1)%4+1; k*k <= N; k+=4) {
kolic += GetPart (Part, N, k, i, size);}
}
}
MPI_Reduce(&kolic, &allkolic, 1, MPI_UNSIGNED_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
endtime=MPI_Wtime();
if (rank==0) {
printf("sqs(%d)=%llu, Runtime = %f\n", N, allkolic, endtime-starttime);
}
MPI_Finalize();
return 0;
}

```

Выпускная квалификационная работа выполнена мной совершенно самостоятельно. Все использованные в работе материалы и концепции из опубликованной научной литературы и других источников имеют ссылки на них.

«\_\_» \_\_\_\_\_ г.

---

*(подпись)*

---

*(Ф.И.О.)*