Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет безопасности информационных технологий

Дисциплина:

«Алгоритмы и структуры данных»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

«PSRS-сортировка»

Выполнили:

Суханкулиев М.,

студент группы N3246

_____
(подпись)

Проверил:

Ерофеев С. А.

_____
(отметка о выполнении)

_____
(подпись)

Санкт-Петербург

2025 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы – Разработать программу сортировки методом PSRS (Parallel Sorting by Regular Sampling) для чисел из файла, которые считываются в статический или динамический массив по выбору пользователя.

Для достижения поставленной цели необходимо решить следующие **задачи**:

- Разработать блок-схему алгоритма;
- Составить спецификацию всех переменных;
- Реализовать программу на языке С;
- Провести тестирование программы.

1 АЛГОРИТМ PSRS

1.1 Описание алгоритма

Начало.

Шаг 1. Исходный массив `numbers` из `n` элементов разделим поровну между `p` потоками. Каждый поток получит подмассив размера `base_size` (с учетом возможного остатка `remainder`).

Шаг 2. На каждом потоке запускаем быструю сортировку (`qsort`) для соответствующего подмассива (`thread_data[i].array`).

Шаг 3. Формируем вспомогательный массив (`sample`) из элементов каждого подмассива `sub_arrays`, взятых под индексами $0, \frac{n}{p^2}, \frac{2n}{p^2}, \dots, \frac{(p-1)n}{p^2}$.

Шаг 4. Сортируем вспомогательный массив `sample` с помощью быстрой сортировки (`qsort`).

Шаг 5. Формируем массив разделителей (`splitters`) из элементов `sample`, взятых под индексами $p + \left[\frac{p}{2}\right] - 1, 2p + \left[\frac{p}{2}\right] - 1, \dots, (p-1)p + \left[\frac{p}{2}\right] - 1$.

Шаг 6. Делим данные в подмассивах `sub_arrays` с помощью массива `splitters`. Пусть $a_1, a_2, \dots, a_j$ — разделители. Тогда данные в каждом подмассиве `sub_arrays[i]` разбиваются на группы элементов, попадающие в соответствующие полуинтервалы $(-\infty, a_1], (a_1, a_2], \dots, (a_j, +\infty)$. Число элементов в каждой группе фиксируется в массиве `counts`.

Шаг 7. Производим многопутевое слияние (`p_way_merge`) всех подмассивов `sub_arrays` в массив `result`. Слияние выполняется поочередно: сначала объединяется первая группа со второй, затем результат с третьей и так далее. В итоге получаем отсортированный массив `result`.

Шаг 8. Записываем отсортированные данные из `result` в выходной файл (`write_result_to_file`).

Конец.

1.2 Блок-схема алгоритма

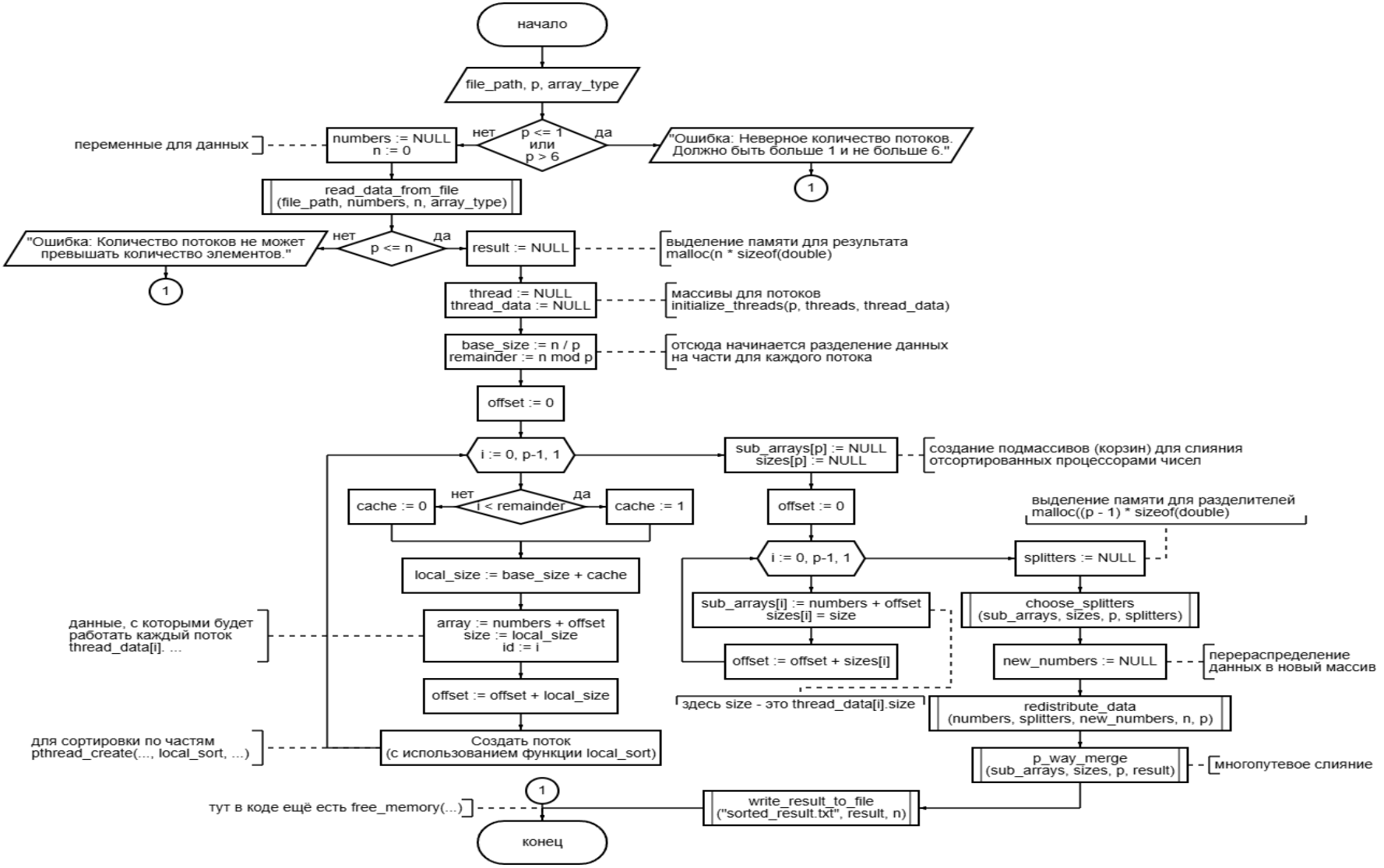

Рисунок 1 – Основная схема алгоритма

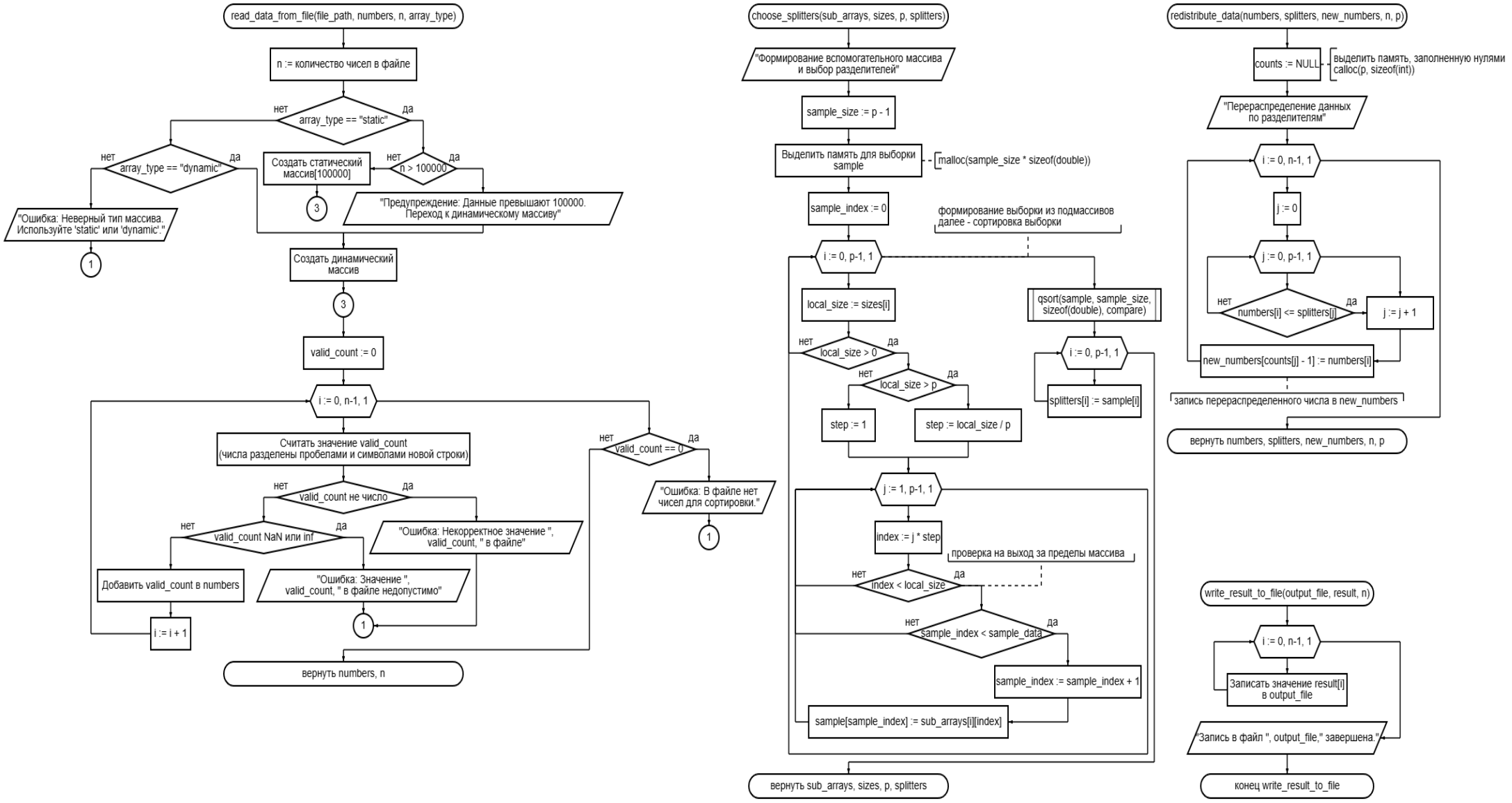

Рисунок 2 – Подпрограммы (I часть)

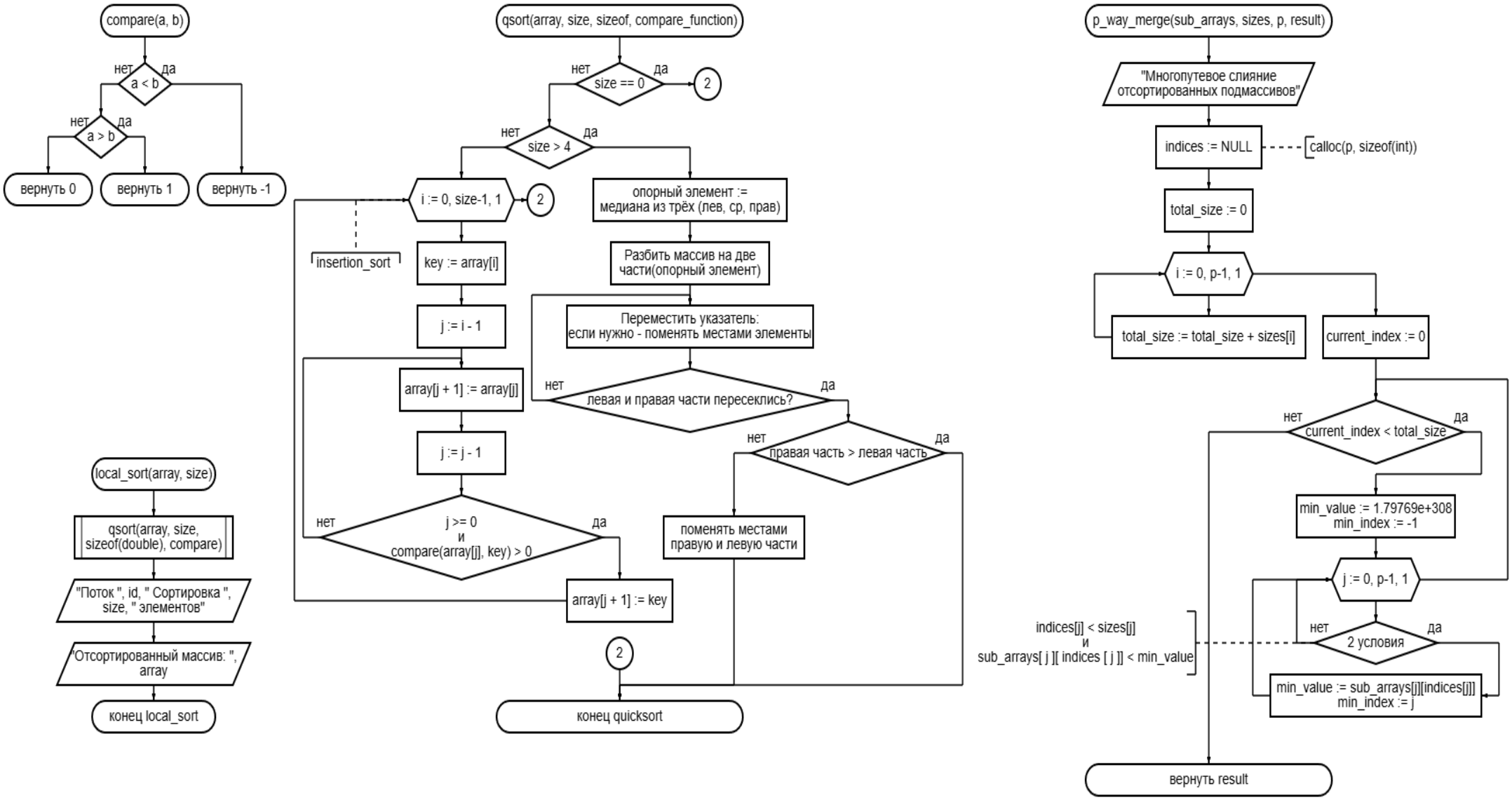

Рисунок 3 – Подпрограммы (II часть)

2 СПЕЦИФИКАЦИЯ ПЕРЕМЕННЫХ

Переменная	Описание	Тип использования	Тип	Размер (байт)	Диапазон значений
`file_path`	Путь к файлу, содержащему данные для сортировки	Входная	const char	8	Путь к файлу в файловой системе (строка)
`p`	Количество потоков	Входная	char	8	[2, 6]
`array_type`	Тип массива (static/dynamic)	Входная	int	4	“static” или “dynamic”
`numbers`	Массив чисел для сортировки	Входная/Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
`n`	Количество чисел в массиве	Промежуточная	int	4	[0, 2 147 483 647]
`threads`	Массив потоков	Промежуточная	pthread_t	8	Системный указатель на поток (зависит от платформы)
`thread_data`	Массив структур с данными для каждого потока	Промежуточная	ThreadData	8	Указатель на структуры с информацией о каждом потоке
`splitters`	Массив разделителей для многопутевого слияния	Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
`new_numbers`	Массив для перераспределенных данных	Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
`sub_arrays`	Массив указателей на подмассивы для слияния	Промежуточная	double	8	Указатели на отсортированные подмассивы
`sizes`	Массив размеров подмассивов для каждого потока	Промежуточная	int	4	[0, 357 913 942]
`indices`	Массив индексов для слияния данных из подмассивов	Промежуточная	int	4	[0, 2 147 483 647]
`sample`	Вспомогательный массив для выборки разделителей	Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
`line`	Буфер для чтения строки из файла	Промежуточная	char[256]	256	Строка с ASCII-символами, длина до 255 символов + \0
`token`	Буфер для токенов строки при разборе данных из файла	Промежуточная	char	8	Указатель на текущий токен
`temp`	Временная переменная для хранения числовых значений	Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
`valid_count`	Количество корректно считанных чисел	Промежуточная	int	4	[0, 2 147 483 647]
`file`	Указатель на файл	Промежуточная	FILE	8	Указатель на открытый файл
`sample_size`	Размер выборки для разделителей	Промежуточная	int	4	[0, 5]
`offset`	Смещение для перераспределения данных	Промежуточная	int	4	[0, 2 147 483 647]
`remainder`	Остаток элементов при делении на количество потоков	Промежуточная	int	4	[0, 2 147 483 647]
`step`	Шаг для формирования выборки разделителей	Промежуточная	int	4	[1, 2 147 483 647]
`min_value`	Минимальное значение для слияния данных	Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
`min_index`	Индекс минимального значения при слиянии данных	Промежуточная	int	4	[0, 5]
`result`	Массив для хранения отсортированных данных	Выходная	double	8	[-1.7977e + 308, 1.7977e + 308]

Примечание: размер переменных в памяти указан для стандартных платформ x86-64.

3 РЕАЛИЗАЦИЯ ПРОГРАММЫ

3.1 Описание программы

Программа написана на языке C и состоит из трех основных файлов:

1) Заголовочный файл (`psrs.h`) – содержит объявления функций и структур данных, которые используются в реализации программы.

2) Основной файл (`main.c`) – включает функцию `main()`, которая управляет процессом выполнения программы.

3) Файл с реализациями функций (`psrs.c`) – содержит определения всех функций, объявленных в заголовочном файле `psrs.h`.

3.2 Код программы

3.2.1 psrs.h

```
#ifndef PSRS_H
#define PSRS_H

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>
#include <string.h>

#define MAX_NUMBERS 100000

typedef struct {
    double *array;
    int size;
    int id;
} ThreadData;

int read_data_from_file(const char *file_path, double **numbers, int *n, const char *array_type);
int initialize_threads(int p, pthread_t **threads, ThreadData **thread_data);
void *local_sort(void *arg);
void choose_splitters(double **sub_arrays, int *sizes, int p, double *splitters);
void redistribute_data(double *numbers, double *splitters, double *new_numbers, int n, int p);
void p_way_merge(double **sub_arrays, int *sizes, int p, double *result);
void write_result_to_file(const char *output_file, double *result, int n);
void free_memory(double *numbers, double *result, double *splitters, double *new_numbers, pthread_t *threads, ThreadData *thread_data, const char *array_type);

#endif // PSRS_H
```

3.2.2 psrs.c

```
#include "psrs.h"

// Функция сортировки
int compare(const void *a, const void *b) {
```

```
            if (*(double*)a < *(double*)b) return -1;
            if (*(double*)a > *(double*)b) return 1;
            return 0;
        }

        // Локальная сортировка для каждого потока
        void *local_sort(void *arg) {
            ThreadData *data = (ThreadData*)arg;
            qsort(data->array, data->size, sizeof(double), compare);
            printf("Поток %d: Сортировка %d элементов...\nОтсортированный подмассив: \n", data-
>id, data->size);
            for (int i = 0; i < data->size; ++i) {
                printf("%.15g ", data->array[i]);
            }
            printf("\n");
            return NULL;
        }

        // Чтение данных из файла и заполнение массива
        int read_data_from_file(const char *file_path, double **numbers, int *n, const char
*array_type) {
            FILE *file = fopen(file_path, "r");
            if (!file) {
                printf("Ошибка: Не удалось открыть файл %s\n", file_path);
                return 0;
            }
            double temp;
            *n = 0;
            // Считываем количество чисел в файле
            while (fscanf(file, "%lf", &temp) == 1) {
                (*n)++;
            }
            rewind(file);
            // Выбор типа массива
            if (strcmp(array_type, "static") == 0) {
                if (*n > MAX_NUMBERS) {
                    printf("Предупреждение: Данные превышают MAX_NUMBERS. Переход к динамическому
массиву.\n");
                    *numbers = (double*)malloc(*n * sizeof(double));  // Динамическая память, если
превышен лимит
                } else {
                    static double static_numbers[MAX_NUMBERS];  // Статический массив
                    *numbers = static_numbers;  // Указываем на начало статического массива
                }
            } else if (strcmp(array_type, "dynamic") == 0) {
                *numbers = (double*)malloc(*n * sizeof(double));  // Динамическая память
            } else {
                printf("Ошибка: Неверный тип массива. Используйте 'static' или 'dynamic'.\n");
                fclose(file);
                return 0;
            }
            if (!*numbers) {
                printf("Ошибка: Не удалось выделить память для чисел\n");
                fclose(file);
                return 0;
            }
            // Считываем данные из файла с проверкой на ошибки
            int valid_count = 0;
            char line[256];  // Буфер для строки
            while (fgets(line, sizeof(line), file)) {
                char *token = strtok(line, " \n\r\t");  // Разделяем строку на токены по пробелам
и символам новой строки
                while (token) {
                    // Проверка, что строка является корректным числом
                    char *endptr;
                    temp = strtod(token, &endptr);
                    // Если strtod не смог преобразовать строку в число
                    if (*endptr != '\0') {
```

```
                        printf("Ошибка: Некорректное значение \"%s\" в файле\n", token);
                        if (array_type && strcmp(array_type, "dynamic") == 0) {  // Освобождаем
только динамически выделенную память
                            free(*numbers);
                        }
                        fclose(file);
                        return 0;
                    }
                    // Проверка на NaN и Infinity
                    if (isnan(temp) || isinf(temp)) {
                        printf("Ошибка: Значение \"%s\" в файле недопустимо\n", token);
                        if (array_type && strcmp(array_type, "dynamic") == 0) {  // Освобождаем
только динамически выделенную память
                            free(*numbers);
                        }
                        fclose(file);
                        return 0;
                    }
                    if (valid_count < *n) {
                        (*numbers)[valid_count++] = temp;
                    }
                    token = strtok(NULL, " \n\r\t");  // Переход к следующему токену
                }
            }
            if (valid_count == 0) {
                printf("Ошибка: В файле нет чисел для сортировки.\n");
                if (array_type && strcmp(array_type, "dynamic") == 0) {  // Освобождаем только
динамически выделенную память
                    free(*numbers);
                }
                fclose(file);
                return 0;
            }
            fclose(file);
            return 1;
        }

        // Функция для выделения памяти для потоков и данных потока
        int initialize_threads(int p, pthread_t **threads, ThreadData **thread_data) {
            *threads = (pthread_t*)malloc(p * sizeof(pthread_t));
            *thread_data = (ThreadData*)malloc(p * sizeof(ThreadData));
            if (!*threads || !*thread_data) {
                printf("Ошибка: Не удалось выделить память для потоков или данных потока\n");
                return 0;
            }
            return 1;
        }

        // Выбор регулярных образцов для разделителей
        void choose_splitters(double **sub_arrays, int *sizes, int p, double *splitters) {
            printf("Формирование вспомогательного массива и выбор разделителей...\n");
            // Выделение памяти для выборки
            int sample_size = p - 1;
            double *sample = (double*)malloc(sample_size * sizeof(double));
            if (!sample) {
                printf("Ошибка: Не удалось выделить память для выборки\n");
                return;
            }
            int sample_index = 0;
            // Формирование выборки из подмассивов
            for (int i = 0; i < p; ++i) {
                int local_size = sizes[i];
                if (local_size > 0) {
                    int step = (local_size > p) ? (local_size / p) : 1; // Защита от деления на 0
                    for (int j = 1; j < p; ++j) {
                        int index = j * step;
                        if (index < local_size) {
                            // Убедиться, что индекс не выходит за пределы массива
```

```
                            if (sample_index < sample_size) {
                                sample[sample_index++] = sub_arrays[i][index];
                            } else {
                                break;
                            }
                        }
                    }
                }
            }
            // Сортировка выборки
            qsort(sample, sample_size, sizeof(double), compare);
            // Выбор разделителей из отсортированной выборки
            for (int i = 0; i < p - 1; ++i) {
                splitters[i] = sample[i];
            }
            free(sample);
        }

        // Перераспределение данных по разделителям
        void redistribute_data(double *numbers, double *splitters, double *new_numbers, int n, int
p) {
            int *counts = (int*)calloc(p, sizeof(int));
            if (!counts) {
                printf("Ошибка: Не удалось выделить память для counts\n");
                return;
            }
            memset(counts, 0, p * sizeof(int));
            printf("Перераспределение данных по разделителям...\n");
            for (int i = 0; i < n; ++i) {
                int j;
                for (j = 0; j < p - 1; ++j) {
                    if (numbers[i] <= splitters[j]) {
                        break;
                    }
                }
                counts[j]++;
                new_numbers[counts[j] - 1] = numbers[i];  // Запись перераспределенного числа в
new_numbers
            }
            free(counts);
        }

        // Многопутевое слияние отсортированных подмассивов
        void p_way_merge(double **sub_arrays, int *sizes, int p, double *result) {
            printf("Многопутевое слияние отсортированных подмассивов...\n");
            int *indices = (int*)calloc(p, sizeof(int));
            if (!indices) {
                printf("Ошибка: Не удалось выделить память для индексов\n");
                return;
            }
            memset(indices, 0, p * sizeof(int));
            int total_size = 0;
            for (int i = 0; i < p; ++i) {
                total_size += sizes[i];
            }
            int current_index = 0;
            while (current_index < total_size) {
                double min_value = __DBL_MAX__;
                int min_index = -1;

                for (int j = 0; j < p; ++j) {
                    if (indices[j] < sizes[j] && sub_arrays[j][indices[j]] < min_value) {
                        min_value = sub_arrays[j][indices[j]];
                        min_index = j;
                    }
                }
                result[current_index] = min_value;
                indices[min_index]++;
```

```
                current_index++;
            }
            free(indices);
        }

        // Запись результата в файл
        void write_result_to_file(const char *output_file, double *result, int n) {
            FILE *file = fopen(output_file, "w");
            if (!file) {
                printf("Ошибка: Не удалось открыть выходной файл %s\n", output_file);
                return;
            }
            for (int i = 0; i < n; i++) {
                fprintf(file, "%.15g ", result[i]);
            }
            fclose(file);
            printf("Запись в файл \"%s\" завершена.\n", output_file);
        }

        // Освобождение памяти
        void free_memory(double *numbers, double *result, double *splitters, double *new_numbers,
pthread_t *threads, ThreadData *thread_data, const char *array_type) {
            if (array_type && strcmp(array_type, "dynamic") == 0) {
                free(numbers);  // Освобождаем только если это динамический массив
            }
            free(result);
            free(splitters);
            free(new_numbers);
            free(threads);
            free(thread_data);
        }
```

3.2.3 main.c

```
#include "psrs.h"

// Главная функция
int main(int argc, char *argv[]) {
    if (argc < 4) {
        printf("Использование:  %s  <путь  к  файлу>  <количество  потоков>  <тип  массива:
        static/dynamic>\n", argv[0]);
        return 1;
    }
    const char *file_path = argv[1];
    int p = atoi(argv[2]);
    if (p <= 1 || p > 6 ) {
        printf("Ошибка: Неверное количество потоков. Должно быть больше 1 и не больше 6.\n");
        return 1;
    }
    const char *array_type = argv[3];
    // Переменные для данных
    double *numbers = NULL;
    int n = 0;
    if (!read_data_from_file(file_path, &numbers, &n, array_type)) {
        printf("Ошибка при чтении файла\n");
        return 1;
    }
    if (p > n) {
        printf("Ошибка: Количество потоков не может превышать количество элементов.\n");
        free(numbers);
        return 1;
    }
    // Резервирование памяти для результата
    double *result = (double*)malloc(n * sizeof(double));
    if (!result) {
        printf("Ошибка: Не удалось выделить память для результата\n");
        free(numbers);
```

```
        return 1;
    }
    // Массивы для потоков
    pthread_t *threads;
    ThreadData *thread_data;
    if (!initialize_threads(p, &threads, &thread_data)) {
        printf("Ошибка: Не удалось создать потоки threads\n");
        free(numbers);
        free(result);
        return 1;
    }
    // Разделение данных на части для каждого потока
    int base_size = n / p;
    int remainder = n % p;
    int offset = 0;
    for (int i = 0; i < p; ++i) {
        int local_size = base_size + (i < remainder ? 1 : 0);
        thread_data[i].array = numbers + offset;
        thread_data[i].size = local_size;
        thread_data[i].id = i;
        offset += local_size;
        pthread_create(&threads[i], NULL, local_sort, &thread_data[i]);
    }
    // Ожидание завершения всех потоков
    for (int i = 0; i < p; ++i) {
        pthread_join(threads[i], NULL);
    }
    // Создание подмассивов для слияния
    double *sub_arrays[p];
    int sizes[p];
    offset = 0;
    for (int i = 0; i < p; ++i) {
        sub_arrays[i] = numbers + offset;
        sizes[i] = thread_data[i].size;
        offset += sizes[i];
    }
    // Выбор разделителей
    double *splitters = (double*)malloc((p - 1) * sizeof(double));
    if (!splitters) {
        printf("Ошибка: Не удалось выделить память для разделителей\n");
        free_memory(numbers, result, NULL, NULL, threads, thread_data, array_type);
        return 1;
    }
    choose_splitters(sub_arrays, sizes, p, splitters);
    // Перераспределение данных
    double *new_numbers = (double*)malloc(n * sizeof(double));
    if (!new_numbers) {
        printf("Ошибка: Не удалось выделить память для new_numbers\n");
        free_memory(numbers, result, splitters, NULL, threads, thread_data, array_type);
        return 1;
    }
    redistribute_data(numbers, splitters, new_numbers, n, p);
    // Многопутевое слияние
    p_way_merge(sub_arrays, sizes, p, result);
    // Запись результата в файл
    write_result_to_file("sorted_result.txt", result, n);
    // Освобождение памяти
    free_memory(numbers, result, splitters, new_numbers, threads, thread_data, array_type);
    return 0;
}
```

4 ТЕСТИРОВАНИЕ ПРОГРАММЫ

Программа скомпилирована с использованием следующих флагов:

```
gcc -O3 main.c psrs.c -o psrs -lpthread -lm -Wall -Wextra -Werror
```

Для проверки на наличие утечек памяти и других ошибок использовалась утилита `Valgrind`.

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./psrs…
```

В процессе тестирования не было обнаружено утечек памяти или других проблем (примеры ниже).

4.1 Скриншоты тестирования

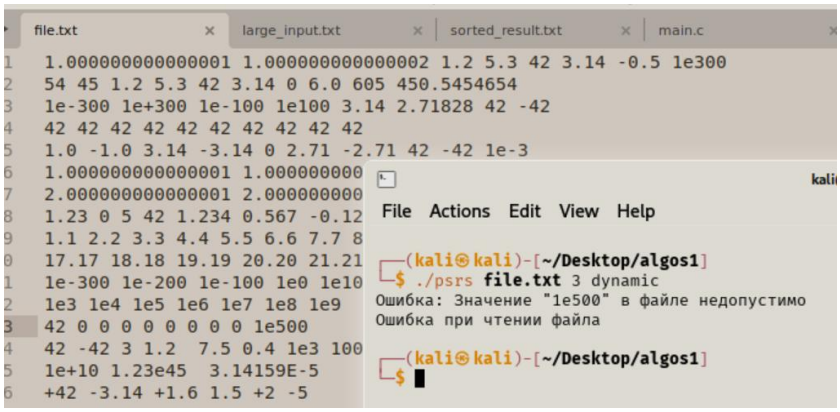

Рисунок 4 – Неподдерживаемые значения

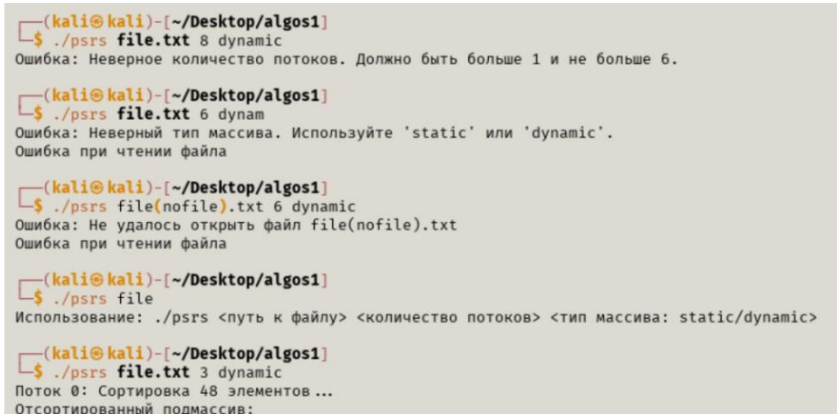

Рисунок 5 – Обработка вводимых аргументов

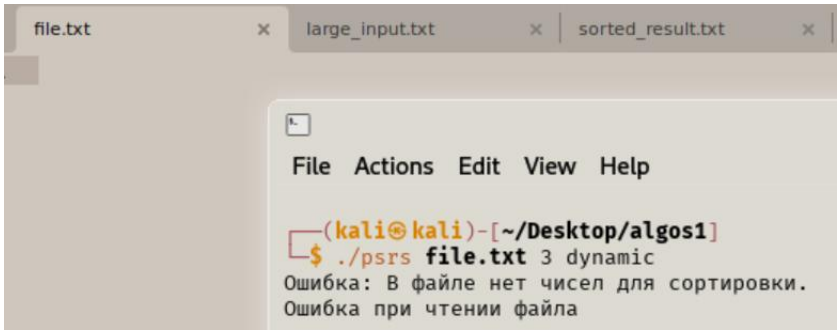

Рисунок 6 – Пустой файл или файл без чисел

Рисунок 7 – Все поддерживаемые значения и вывод

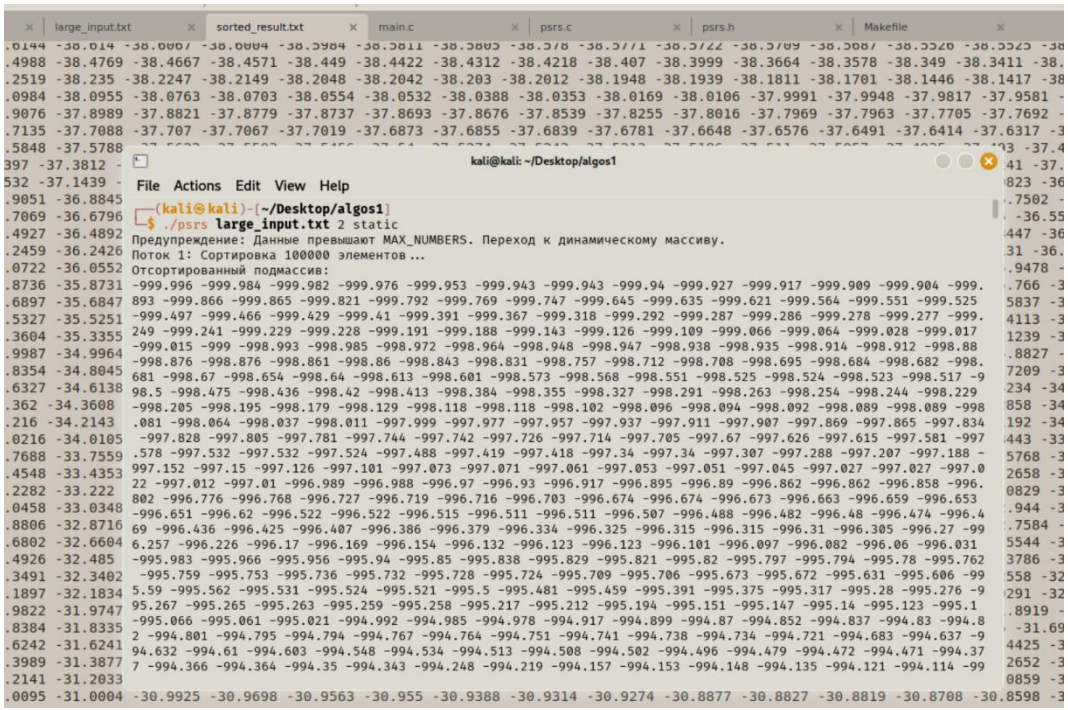

Рисунок 8 – Очень большое количество чисел

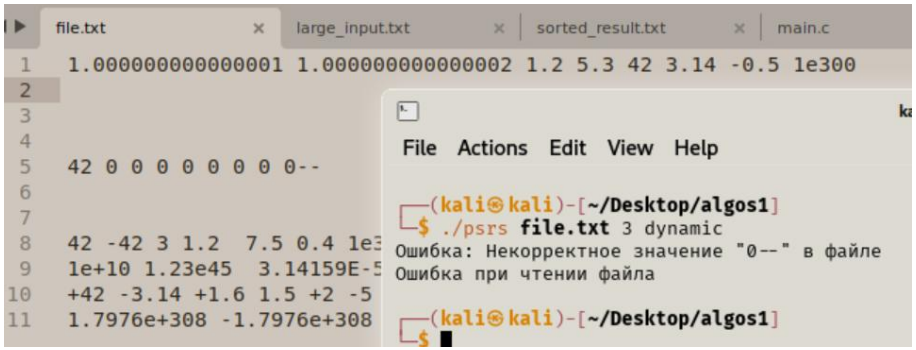

Рисунок 9 – Некорректные знаки

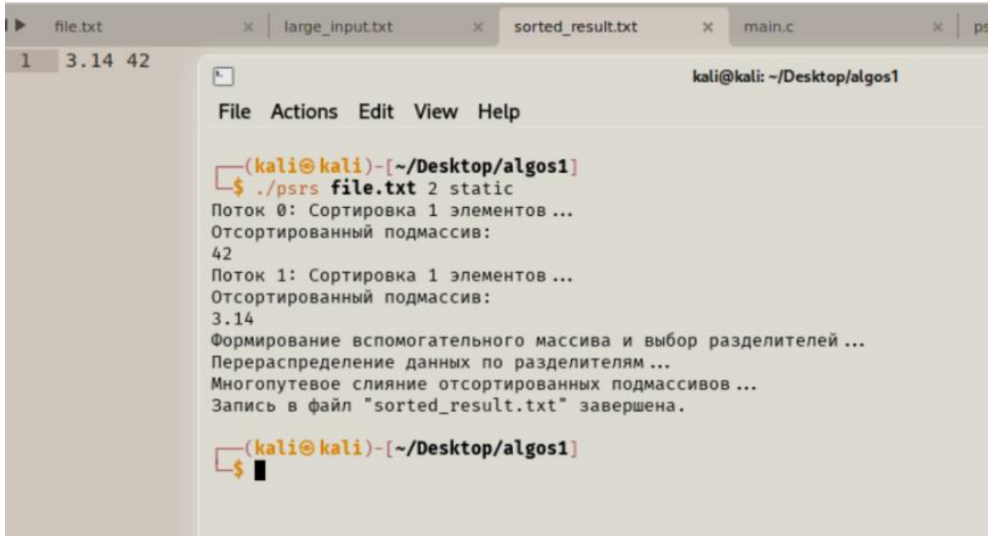

Рисунок 10 – Минимальное количество чисел

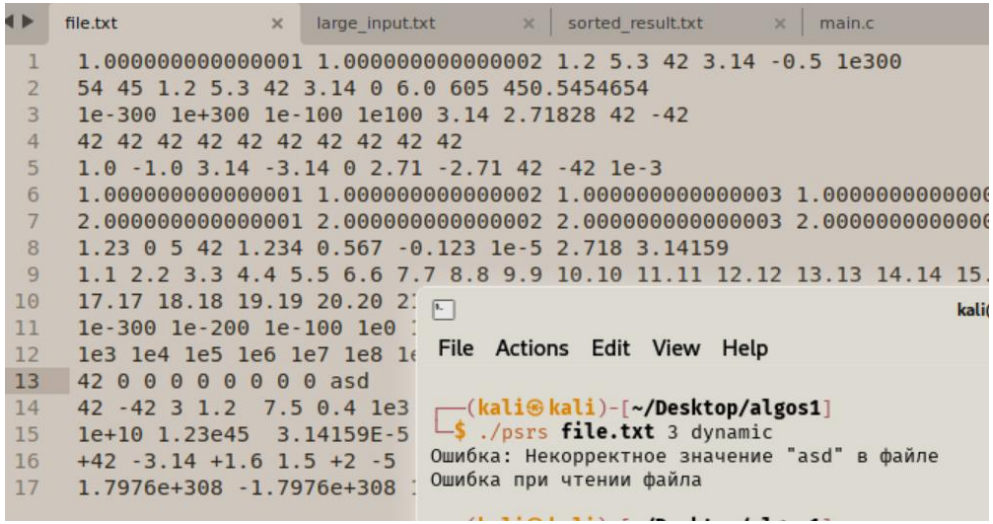

Рисунок 11 – Строка в файле

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы был разработан и реализован алгоритм параллельной сортировки методом PSRS. Для проверки работоспособности программы проведено тестирование с различными входными данными, включая крайние случаи и большие массивы чисел. Анализ результатов показал корректность работы алгоритма и отсутствие утечек памяти.

Выполненная работа позволила закрепить навыки многопоточного программирования на языке C, работы с динамическими структурами данных и анализа алгоритмов сортировки.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Сортировки — Викиконспекты – 2022.
2. PSRS-сортировка — Викиконспекты – 2022.
3. Быстрая сортировка — Викиконспекты – 2022.
4. H. Shi and J. Schaeffer. "Parallel Sorting by Regular Sampling. Journal of Parallel and Distributed Computing," 14(4):361--372, 1992. – URL : psrs1.pdf - Yandex Documents
5. Parallel and Distributed Computing - Department of Computer Science and Engineering (DEI) Instituto Superior Tencico – 2012. – URL : parallelsort.pdf - Yandex Documents
6. Sathish Vadhiyar Parallel Sorting – 2012. – URL : ParallelSorting.pdf - Yandex Documents
7. .Stack Overflow – 2022. – URL : Which parallel sorting algorithm has the best average case performance? - Stack Overflow