**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет информационных технологий**

**Кафедра параллельных вычислений**

**ОТЧЕТ О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ**

# Параллельная реализация алгоритма Quick Sort с помощью OpenMP

студента 2 курса, группы 23201

Смирнова Гордея Андреевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
А.С. Матвеев

Новосибирск 2025

# СОДЕРЖАНИЕ

# ЦЕЛЬ

Изучить особенности параллельной реализации алгоритма быстрой сортировки с использованием директив OpenMP, а также оценить эффективность распараллеливания за счёт создания независимых задач.

# ЗАДАНИЕ

1. **Изучить возможности OpenMP:**

- Проанализировать работу директив ***task*** и ***taskwait*** для создания и управления параллельными задачами.
- Определить порядок создания задач при использовании директив внутри параллельной секции.

2. **Реализовать параллельную версию алгоритма *Quick sort:***

- Разработать программу, реализующую параллельную быструю сортировку с использованием OpenMP.
- Внедрить условие создания задачи (с помощью параметра ***if(expr)***), чтобы избежать накладных расходов на создание задач для небольших подзадач.

3. **Экспериментальное исследование:**

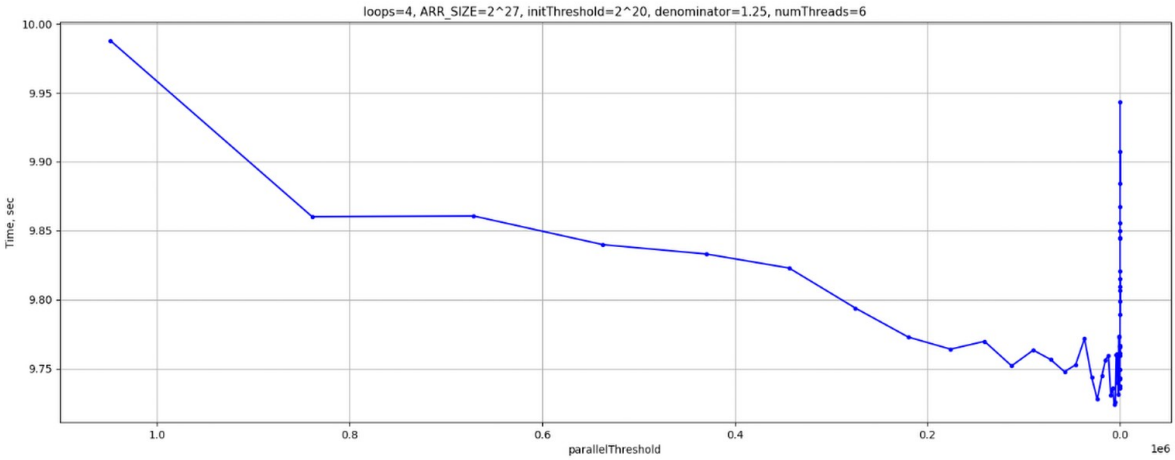
- Экспериментально определить минимальный размер массива, начиная с которого выполнение рекурсивных вызовов в отдельных задачах становится выгодным.
- Провести измерения времени сортировки массива при использовании различного количества потоков.

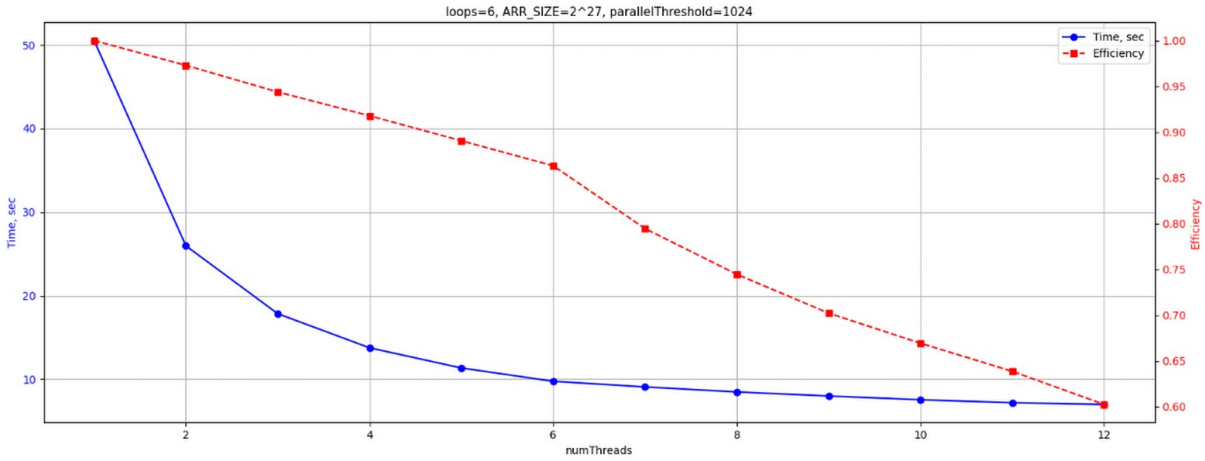
4. **Анализ и визуализация результатов:**

- Построить график зависимости времени сортировки массива от числа используемых потоков, оценить эффективность распараллеливания.

# ОПИСАНИЕ РАБОТЫ

На языке C++ была написана OpenMP-программа для параллельной сортировки массива из float. Были проведены замеры зависимости производительности от parallelThreshold (пороговый размер массива, начиная с которого задача рекурсивной сортировки перестает распараллеливаться) и numThreads (кол-во используемых потоков процессора). Результаты замеров находятся ниже, исходный код программы находится в приложениях.




По результатам замеров было установлено, что минимальный пригодный для распараллеливания размер массива находится в промежутке от 1000 до 2000.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения практической работы были изучены функции библиотеки OpenMP, позволяющие распараллеливать вычисления с помощью создания новых задач.

# ПРИЛОЖЕНИЯ

### Приложение 1: *SortableArray.h*

```
#pragma once

#include <vector>
#include <string>

class SortableArray {
 public:
    SortableArray();
    void testParallelThreshold();
    void testNumThreads();

 private:
    static std::vector<float> fillRandom();
    void quickSortOMP(size_t beginIdx, size_t sortSize);
    size_t partition(size_t beginIdx, size_t sortSize);
    double qSortMeasureTime(int loops);
    void validate() const;

    const std::vector<float> unsorted;
    std::vector<float> sorted;
    size_t parallelThreshold;

    static constexpr size_t ARR_SIZE = 128 * 1024 * 1024;

    static inline const std::string TEST_PT_FILE_NAME = "testPT.csv";
    static constexpr int TEST_PT_NUM_LOOPS = 4;
    static constexpr size_t TEST_PT_INIT_PARALLEL_THRESHOLD = ARR_SIZE /
128;
    static constexpr float TEST_PT_DENOMINATOR = 1.25;
    const int TEST_PT_NUM_THREADS;

    static inline const std::string TEST_NT_FILE_NAME = "testNT.csv";
    static constexpr int TEST_NT_NUM_LOOPS = 8;
    static constexpr size_t TEST_NT_PARALLEL_THRESHOLD = 1024;
    const int TEST_NT_MAX_THREADS;
};
```

### Приложение 2: *SortableArray.cpp*

```
#include "SortableArray.h"

#include <omp.h>
#include <random>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <stdexcept>
```

```
#include <chrono>
#include <vector>
#include <utility>

SortableArray::SortableArray()
: unsorted(fillRandom()), parallelThreshold(0),
TEST_PT_NUM_THREADS(omp_get_max_threads() / 2),
TEST_NT_MAX_THREADS(omp_get_max_threads()) {
    std::cout << "Created array of length " << ARR_SIZE << std::endl;
}

void SortableArray::testParallelThreshold() {
    std::cout << "Testing parallelThreshold..." << std::endl;

    std::ofstream file(TEST_PT_FILE_NAME);
    if (!file.is_open()) throw std::runtime_error(
        "Could not open file " + TEST_PT_FILE_NAME);

    omp_set_num_threads(TEST_PT_NUM_THREADS);
    file << "parallelThreshold,Time (sec)" << std::endl;
    for (parallelThreshold = TEST_PT_INIT_PARALLEL_THRESHOLD;
         parallelThreshold > 0; parallelThreshold /= TEST_PT_DENOMINATOR) {
        const double duration = qSortMeasureTime(TEST_PT_NUM_LOOPS);
        validate();
        file << parallelThreshold << ',' << duration << std::endl;
    }

    file.close();
    std::cout << "Data written in " << TEST_PT_FILE_NAME << std::endl;
}

void SortableArray::testNumThreads() {
    std::cout << "Testing numThreads..." << std::endl;

    std::ofstream file(TEST_NT_FILE_NAME);
    if (!file.is_open()) throw std::runtime_error(
        "Could not open file " + TEST_NT_FILE_NAME);

    parallelThreshold = TEST_NT_PARALLEL_THRESHOLD;
    file << "numThreads,Time (sec)" << std::endl;
    for (int numThreads = 1; numThreads <= TEST_NT_MAX_THREADS; +
+numThreads) {
        omp_set_num_threads(numThreads);
        const double duration = qSortMeasureTime(TEST_NT_NUM_LOOPS);
        validate();
        file << numThreads << ',' << duration << std::endl;
    }

    file.close();
    std::cout << "Data written in " << TEST_NT_FILE_NAME << std::endl;
}

void SortableArray::quickSortOMP(size_t beginIdx, size_t sortSize) {
    if (sortSize <= 1) return;
```

```
    const size_t pivot = partition(beginIdx, sortSize) + 1;
    const size_t sizeL = pivot - beginIdx;
    const size_t sizeR = sortSize - sizeL;

    #pragma omp task default(none) shared(sorted, beginIdx, sizeL) \
        if (sizeL > parallelThreshold)
    quickSortOMP(beginIdx, sizeL);
    #pragma omp task default(none) shared(sorted, pivot, sizeR) \
        if (sizeR > parallelThreshold)
    quickSortOMP(pivot, sizeR);
}

size_t SortableArray::partition(size_t beginIdx, size_t sortSize) {
    const float pivot = sorted[beginIdx];
    size_t l = beginIdx;
    size_t r = beginIdx + sortSize - 1;

    while (true) {
        while (l < beginIdx + sortSize && sorted[l] < pivot) ++l;
        while (r > beginIdx && sorted[r] > pivot) --r;

        if (l ≥ r) return r;
        std::swap(sorted[l], sorted[r]);
        ++l;
        --r;
    }
}

double SortableArray::qSortMeasureTime(int loops) {
    auto minDuration = std::chrono::high_resolution_clock::duration::max();
    for (int i = 0; i < loops; ++i) {
        sorted = unsorted;
        auto start = std::chrono::high_resolution_clock::now();
        #pragma omp parallel default(shared)
        {
            #pragma omp single
            {
                #pragma omp task default(none) shared(sorted)
                quickSortOMP(0, ARR_SIZE);
                #pragma omp taskwait
            }
        }
        auto end = std::chrono::high_resolution_clock::now();
        minDuration = std::min(minDuration, end - start);
    }
    return std::chrono::duration_cast<
        std::chrono::duration<double>>(minDuration).count();
}

void SortableArray::validate() const {
    for (size_t i = 1; i < ARR_SIZE; ++i) {
        if (sorted[i - 1] > sorted[i]) {
            throw std::runtime_error("Array sorted incorrectly!");
```

```
        }
    }
}

std::vector<float> SortableArray::fillRandom() {
    auto arr = std::vector<float>(ARR_SIZE);
    std::mt19937 gen(23201);
    std::uniform_real_distribution dist(0.0f, 1000.0f);
    std::generate(arr.begin(), arr.end(), [&] { return dist(gen); });
    return arr;
}
```

**Приложение 3:** ***main.cpp***

```
#include <iostream>
#include "SortableArray.h"

int main() {
    try {
        SortableArray arr;
        arr.testParallelThreshold();
        arr.testNumThreads();
        std::cout << "Done! Exiting..." << std::endl;
    } catch (const std::runtime_error& e) {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

**Приложение 4*: CMakeLists.txt***

```
cmake_minimum_required(VERSION 3.29)
project(lab2_5)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_FLAGS "-O3 -march=native")

find_package(OpenMP REQUIRED)

set(SRC
        src/SortableArray.cpp
)

add_executable(${PROJECT_NAME} ${SRC} src/main.cpp)
target_include_directories(${PROJECT_NAME} PRIVATE
        src/include
)
target_link_libraries(${PROJECT_NAME} PRIVATE OpenMP::OpenMP_CXX)
```