

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное образовательное учреждение высшего образования «Уральский федеральный университет имени первого Президента России Б.Н. Ельцина» (УрФУ)

Физико-технологический институт

Кафедра «Технической физики»

Оценка

Преподаватель

Кашин И.В.

**АЛГОРИТМ БЫСТРОЙ СОРТИРОВКИ С ИСПОЛЬЗОВАНИЕМ МНОГОПОТОЧНОСТИ В PYTHON**

ОТЧЕТ

по лабораторной работе №5

|  |  |  |
| --- | --- | --- |
| Студент | Маркова Алёна Денисовна | ФИО студента |

|  |
| --- |
| Специальность (направление подготовки) |
| 09.03.02 Информационные системы и технологии | |

|  |  |
| --- | --- |
| Группа | Фт-420008 |

Екатеринбург

2025

**СОДЕРЖАНИЕ**

[ОПИСАНИЕ ЗАДАЧИ 3](#_Toc209471666)

[ПРИНЦИП ПАРАЛЛЕЛИЗАЦИИ 7](#_Toc209471667)

[РЕШЕНИЕ ЗАДАЧИ 8](#_Toc209471668)

[ОЦЕНКА ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛИЗАЦИИ 11](#_Toc209471669)

[ПРИЛОЖЕНИЕ А – ЛИСТИНГ КОДА 13](#_Toc209471670)

ОПИСАНИЕ ЗАДАЧИ

В данной задаче нам предстояло реализовать какой-либо алгоритм для эффективной сортировки массива чисел. Изучив виды сортировки для задачи, я выбрала алгоритм быстрой сортировки.

Быстрая сортировка – это один из самых эффективных алгоритмов сортировки, который работает по принципу "Разделяй и властвуй". Он был разработан Тони Хоаром в МГУ в 1960 году и до сих пор остается одним из самых популярных алгоритмов.

Общая идея алгоритма состоит в следующем:

* Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.
* Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие».
* Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.



Рисунок 1 – Визуализация алгоритма быстрой сортировки

Временная сложность алгоритма определяет, как быстро растет время выполнения при увеличении размера входных данных. Для быстрой сортировки существует три сценария:

1. Лучший случай – O(n log n)

Лучший случай достигается, когда опорный элемент каждый раз делит массив примерно пополам. Это означает, что обе части (левая и правая) получаются приблизительно равными по размеру.

1. Средний случай – O(n log n)

(по факту реальная производительность алгоритма)

Средний случай – типичная ситуация для случайных данных. Даже если опорный элемент не всегда делит массив ровно пополам, он делит его достаточно хорошо (например, 30/70 или 40/60).

1. Худший случай – O(n2)

Худший случай наступает, когда опорный элемент постоянно выбирается неудачно и делит массив крайне неравномерно (например, 1 элемент и n-1 элемент).

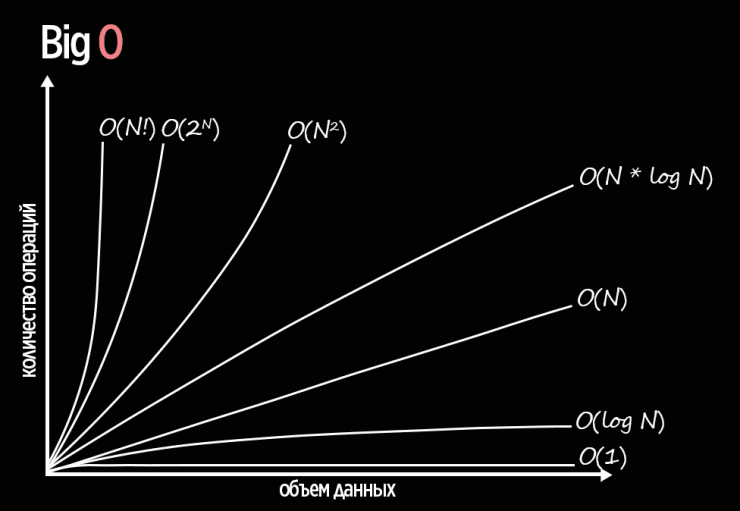


Рисунок 2 – График алгоритмической сложности

ПРИНЦИП ПАРАЛЛЕЛИЗАЦИИ

Основная идея параллелизации заключается в том, что сортировка больших массивов данных является вычислительно затратной задачей. Для массива из N элементов классическая быстрая сортировка требует в среднем O(n log n) операций. При выбранном N = 5.000.000 это составляет примерно 111 миллионов операций сравнения и перестановок, что на однопроцессорной системе занимает значительное время. Использование многопроцессорной обработки позволяет распределить вычислительную нагрузку между несколькими ядрами процессора и существенно ускорить выполнение.

Для параллелизации воспользуемся пакетом multiprocessing, который позволяет запускать несколько процессов одновременно и распределять между ними задачи.

**Принцип работы следующий:**

1. **Разделение исходного массива на независимые части (chunks)**

Массив из N элементов делится на P равных частей, где P — количество используемых процессов. Размер каждой части определяется как chunk\_size = N / P. Последний chunk может содержать дополнительные элементы, если N не делится нацело на P.

Каждому chunk присваивается уникальный идентификатор для сохранения правильного порядка после параллельной обработки.

1. **Создание пула процессов и распределение задач**

С помощью конструкции:

with mp.Pool(processes=processes) as pool:

results = pool.map(worker\_inplace, chunks)

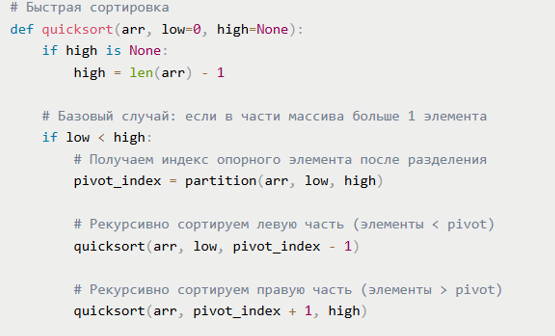
создаётся пул из P процессов, и каждый процесс независимо выполняет сортировку своей части массива. Функция pool.map() автоматически распределяет задачи между доступными процессами и ожидает завершения всех вычислений.

1. **Параллельная сортировка независимых частей**

Каждый процесс получает свою часть массива и независимо выполняет быструю сортировку с использованием функции partition и quicksort.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.



1. **Синхронизация и упорядочивание результатов**

После завершения всех процессов результаты возвращаются в виде списка кортежей:

results = [(sorted\_chunk\_1, time\_1, id\_1), (sorted\_chunk\_2, time\_2, id\_2), ...]

Результаты сортируются по идентификаторам chunk для восстановления правильного порядка:

results.sort(key=lambda x: x[2])

Это необходимо, так как процессы могут завершаться в произвольном порядке, не совпадающем с исходным расположением chunks в массиве.

1. **Итеративное попарное слияние отсортированных частей**

На этом этапе выполняется объединение P отсортированных частей в единый массив. Используется алгоритм итеративного попарного слияния (аналогичный подходу в сортировке слиянием):

Итерация 1: Chunks сливаются попарно: (1,2), (3,4), (5,6), ..., получаем P/2 больших chunks;

Итерация 2: Полученные chunks снова сливаются попарно: (1-2, 3-4), (5-6, 7-8), ..., получаем P/4 chunks;

Итерация k: Процесс повторяется, пока не останется один полностью отсортированный массив.

Количество итераций слияния составляет log₂(P). Каждая итерация обрабатывает все N элементов, поэтому сложность этапа слияния — O(n log P).

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

1. **Слияние двух отсортированных массивов**

Для объединения двух отсортированных chunks используется классический алгоритм "два указателя":

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

РЕШЕНИЕ ЗАДАЧИ

Рисунок 3 демонстрирует результаты выполнения программы в консоли. Программа последовательно тестировала сортировку с различным количеством процессов

Изображение выглядит как текст, Шрифт, снимок экрана, алгебра

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 3 – Результаты выполнения программы в консоли

После выполнения всех тестов программа вывела сообщение: "Массив отсортирован правильно!" – это подтверждает, что параллельная реализация алгоритма работает корректно и не нарушает порядок элементов.

Наилучший результат достигается при использовании 6 процессов – время выполнения составляет ~7.22 секунды. Это означает, что задача решается почти в 3 раза быстрее.

Для каждого числа процессов программа измеряет реальное время выполнения и строит график зависимости времени работы от количества процессов. На графике показаны две кривые: идеальная производительность, которая уменьшается линейно с ростом числа процессов, и реальная производительность, измеренная экспериментально. Это позволяет оценить, насколько эффективно распараллеливание ускоряет вычисления.

ОЦЕНКА ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛИЗАЦИИ

Мной было подобрано число N=5.000.000, при котором время выполнения программы будет превышать 20 секунд.

Если мы будем брать в качестве количества потоков число, равное количеству логических процессоров, то заметим, что после 6 потоков рост производительности практически прекращается. Это связано с тем, что мой процессор имеет 6 физических ядер, а использование 12 логических потоков (за счёт технологии Hyper-Threading) не даёт двукратного увеличения скорости, а лишь частично улучшает распределение нагрузки.

Оптимальное количество потоков для данной задачи соответствует числу физических ядер процессора. Увеличение числа потоков сверх этого значения не даёт существенного выигрыша и может даже ухудшить результат. График, показывающий использование 6 процессов показан на рисунке 4.

Изображение выглядит как текст, линия, График, диаграмма

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 4 – График зависимости времени работы от количества использованных процессов с использованием 6 процессов

Реальное время: [21.174751800019294, 12.230333599960431, 9.693543700035661, 8.53597249998711, 7.6933295000344515, 7.218865999951959]

Идеальное время: [21.174751800019294, 10.587375900009647, 7.058250600006431, 5.2936879500048235, 4.234950360003859, 3.5291253000032157]

ПРИЛОЖЕНИЕ А – ЛИСТИНГ КОДА

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, документ, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, документ, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, документ, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.