

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.

**РАСПАРАЛЛЕЛИВАНИЕ АЛГОРИТМА СОРТИРОВКИ СЛИЯНИЕМ (MERGESORT)**

ОТЧЕТ

по лабораторной работе №5

|  |  |  |
| --- | --- | --- |
| Студент | Черняков Матвей Сергеевич | ФИО студента |

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

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

Екатеринбург

2025

**ОГЛАВЛЕНИЕ**

[ОПИСАНИЕ ЗАДАЧИ 3](#_Toc211277100)

[ПРИНЦИП ПАРАЛЛЕЛИЗАЦИИ 4](#_Toc211277101)

[РЕШЕНИЕ ЗАДАЧИ 6](#_Toc211277102)

[ОЦЕНКА ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛИЗАЦИИ 9](#_Toc211277103)

[ПРИЛОЖЕНИЕ А – ЛИСТИНГ КОДА 10](#_Toc211277104)

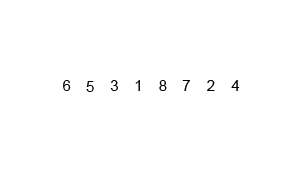
[ПРИЛОЖЕНИЕ Б – ЭКСПЕРИМЕНТ 1 14](#_Toc211277108)

ОПИСАНИЕ ЗАДАЧИ

Целью данной лабораторной работы является изучение возможностей параллельного программирования на примере алгоритма сортировки слиянием (Mergesort). В рамках работы реализуется классический последовательный алгоритм Mergesort и его параллелизация с использованием модуля multiprocessing в Python. Основная задача заключается в исследовании зависимости времени выполнения алгоритма от количества используемых процессов.

Для проведения эксперимента генерируется массив случайных целых чисел размером 5 000 000 элементов, что обеспечивает время выполнения около 20 секунд при использовании одного процесса. Затем производится серия запусков программы с количеством процессов от 1 до 24, при этом для каждого запуска фиксируется реальное время выполнения. Полученные результаты сравниваются с теоретически идеальным временем, которое вычисляется как отношение времени работы одного процесса к количеству процессов. По результатам экспериментов строится график, демонстрирующий отклонение реального времени выполнения от идеального.

Алгоритм сортировки слиянием (MergeSort) представляет собой классический пример рекурсивного подхода по принципу "разделяй и властвуй". Временная сложность алгоритма Mergesort составляет O(n log n) в худшем, среднем и лучшем случаях. Требует дополнительной памяти размером O(n) для временного хранения элементов во время слияния.



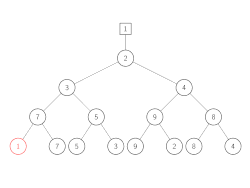
GIF 1 – Алгоритм MergeSort

ПРИНЦИП ПАРАЛЛЕЛИЗАЦИИ

Параллелизация алгоритма сортировки слиянием основана на принципе разделения исходного массива данных на несколько независимых частей, каждая из которых обрабатывается отдельным процессом. При запуске программы с M процессами исходный массив делится на M приблизительно равных сегментов, причем последний сегмент может быть немного больше для учета остатка от деления. Каждый процесс получает границы своего сегмента в виде индексов начала и конца, а также копию исходного массива данных.

Воркер-функция каждого процесса извлекает свою часть массива и выполняет над ней классическую сортировку слиянием независимо от других процессов. После завершения работы всех процессов получается M отсортированных подмассивов, которые необходимо объединить в единый отсортированный массив. Для этого используется алгоритм k-way merge, который эффективно сливает несколько отсортированных массивов в один с помощью структуры данных min-heap.

Ключевое преимущество использования min-heap заключается в том, что операции вставки и извлечения минимального элемента выполняются за логарифмическое время O(log k), где k - количество подмассивов. Таким образом, общая временная сложность k-way merge составляет O(n log k), где n - суммарное количество элементов во всех подмассивах.



GIF 2 – K-way merge

На начальном этапе алгоритма создается пустая куча, и в нее добавляются первые элементы из каждого отсортированного подмассива. Основной цикл алгоритма работает до тех пор, пока куча не станет пустой. На каждой итерации из кучи извлекается минимальный элемент, который по определению является наименьшим среди всех первых неиспользованных элементов всех подмассивов. Этот элемент добавляется в результирующий массив. После извлечения элемента алгоритм проверяет, остались ли еще элементы в том подмассиве, откуда был взят только что извлеченный элемент. Если да, то следующий элемент из этого подмассива добавляется в кучу вместе с обновленной позицией.

Сортировки полностью параллелизуются, поскольку процессы работают независимо друг от друга, но объединение результатов выполняется последовательно. Время выполнения всей программы определяется самым медленным процессом (воркером), что при неравномерном распределении нагрузки или различной загруженности процессорных ядер может приводить к дополнительным задержкам. Это объясняет отклонение реального времени выполнения от теоретически идеального.

РЕШЕНИЕ ЗАДАЧИ

Функция mergesort(arr) – сортировка массивов.

def mergesort(arr):

"""  
Возвращает отсортированный массив.  
"""  
if len(arr) <= 1:  
 return arr  
  
 mid = len(arr) // 2  
 left = mergesort(arr[:mid])  
 right = mergesort(arr[mid:])  
  
 return merge(left, right)

Функция merge(left, right) – слияние отсортированных массивов в единый.

def merge(left, right):  
 *"""  
 Слияние двух отсортированных массивов в один отсортированный.  
 """* result = []  
 i = j = 0  
  
 while i < len(left) and j < len(right):  
 if left[i] <= right[j]:  
 result.append(left[i])  
 i += 1  
 else:  
 result.append(right[j])  
 j += 1  
  
 result.extend(left[i:])  
 result.extend(right[j:])  
  
 return result

Функция worker\_sort(start, end, data) – воркер, запускающий mergesort.

def worker\_sort(start, end, data):  
 *"""  
 Воркер-функция для сортировки части массива.  
 Принимает индексы start, end и общий массив data.  
 Возвращает отсортированную часть массива и время работы.  
 """* t0 = time.perf\_counter()  
  
 *# Извлекаем часть массива* chunk = data[start:end]  
  
 *# Сортируем часть* sorted\_chunk = mergesort(chunk)  
  
 t1 = time.perf\_counter()  
 elapsed = t1 - t0  
  
 return sorted\_chunk, elapsed

Функция k\_way\_merge(sorted\_chunks) – объединение отсортированных массивов с помощью min-heap.

def k\_way\_merge(sorted\_chunks):  
 *"""  
 Объединяет K отсортированных массивов в один отсортированный массив.  
 Использует min-heap для эффективного слияния.  
 """  
 # Создаём кучу с первыми элементами каждого массива* heap = []  
 for chunk\_idx, chunk in enumerate(sorted\_chunks):  
 if len(chunk) > 0:  
 heapq.heappush(heap, (chunk[0], chunk\_idx, 0)) *# (значение, индекс массива, индекс в массиве)* result = []  
  
 while heap:  
 value, chunk\_idx, elem\_idx = heapq.heappop(heap)  
 result.append(value)  
  
 *# Добавляем следующий элемент из того же массива, если он есть* if elem\_idx + 1 < len(sorted\_chunks[chunk\_idx]):  
 next\_value = sorted\_chunks[chunk\_idx][elem\_idx + 1]  
 heapq.heappush(heap, (next\_value, chunk\_idx, elem\_idx + 1))  
  
 return result

Далее для эксперимента используется вызов этих функций и функций воркеров с помощью pool.starmap из библиотеки multiprocessing.

for M in range(1, 25):  
 *# Разделяем массив на M частей* chunk\_size = N\_test // M  
 ranges = []  
 for i in range(M):  
 start = i \* chunk\_size  
 end = (i + 1) \* chunk\_size if i < M - 1 else N\_test  
 ranges.append((start, end, data\_test))  
  
 *# Запускаем параллельную сортировку* t\_start = time.perf\_counter()  
  
 with multiprocessing.Pool(processes=M) as pool:  
 results = pool.starmap(worker\_sort, ranges)

ОЦЕНКА ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛИЗАЦИИ

Ключевыми параметрами является количество ядер (12) и количество потоков (24). Для данных характеристик было проведено тестирование сравнения идеального и условного времени сортировки при M от 1 до 24. Результаты тестирования представлены на графике (рисунок 1). Точные значения представлены в приложении Б;

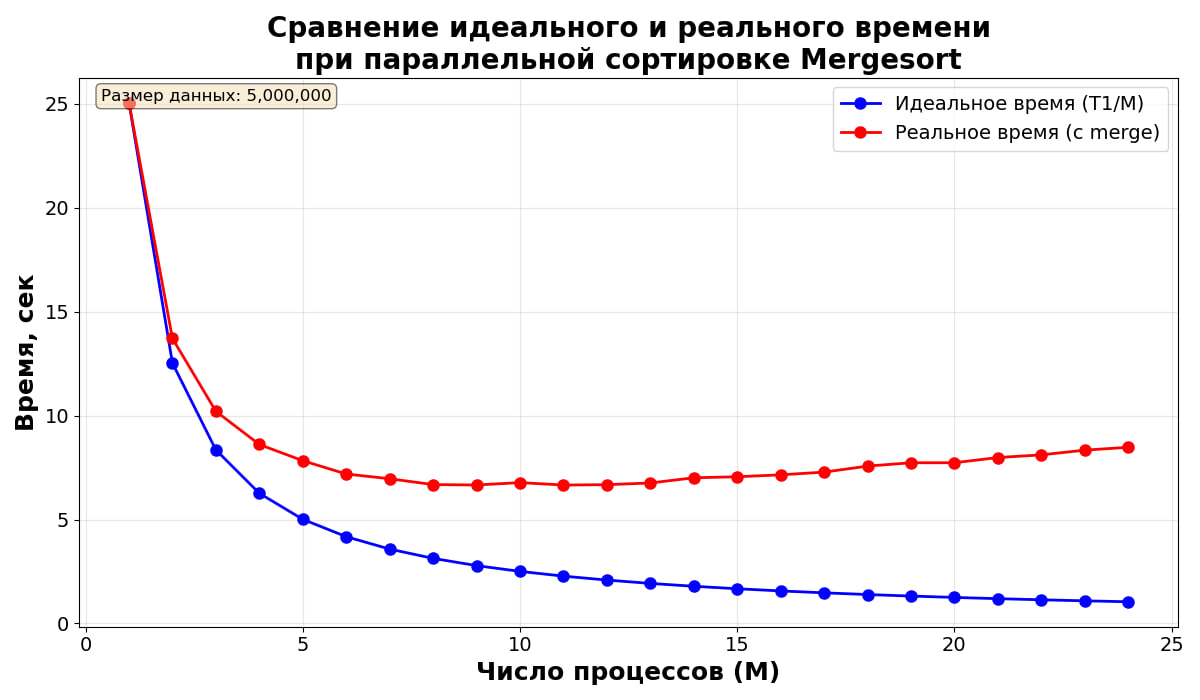


Рисунок 1 – График эксперимента 1

По графику для эксперимента 1 видно, что реальное время достаточно близко к идеальному и отклоняется на большом количестве потоков меньше, чем на 2-3 секунды. Отсюда можно понять, что можно выбрать оптимальное количество потоков для распараллеливания данного алгоритма сортировки (здесь это примерно 11 потоков).

ПРИЛОЖЕНИЕ А – ЛИСТИНГ КОДА

import heapq  
import multiprocessing  
import time  
  
import matplotlib.pyplot as plt  
import numpy as np  
  
  
def mergesort(arr):  
 *"""  
 Классическая рекурсивная реализация сортировки слиянием.  
 Возвращает отсортированный массив.  
 """* if len(arr) <= 1:  
 return arr  
  
 mid = len(arr) // 2  
 left = mergesort(arr[:mid])  
 right = mergesort(arr[mid:])  
  
 return merge(left, right)  
  
  
def merge(left, right):  
 *"""  
 Слияние двух отсортированных массивов в один отсортированный.  
 """* result = []  
 i = j = 0  
  
 while i < len(left) and j < len(right):  
 if left[i] <= right[j]:  
 result.append(left[i])  
 i += 1  
 else:  
 result.append(right[j])  
 j += 1  
  
 result.extend(left[i:])  
 result.extend(right[j:])  
  
 return result  
  
  
def worker\_sort(start, end, data):  
 *"""  
 Воркер-функция для сортировки части массива.  
 Принимает индексы start, end и общий массив data.  
 Возвращает отсортированную часть массива и время работы.  
 """* t0 = time.perf\_counter()  
  
 *# Извлекаем часть массива* chunk = data[start:end]  
  
 *# Сортируем часть* sorted\_chunk = mergesort(chunk)

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

t1 = time.perf\_counter()  
 elapsed = t1 - t0  
  
 return sorted\_chunk, elapsed  
  
  
def k\_way\_merge(sorted\_chunks):  
 *"""  
 Объединяет K отсортированных массивов в один отсортированный массив.  
 Использует min-heap для эффективного слияния.  
 """  
 # Создаём кучу с первыми элементами каждого массива* heap = []  
 for chunk\_idx, chunk in enumerate(sorted\_chunks):  
 if len(chunk) > 0:  
 heapq.heappush(heap, (chunk[0], chunk\_idx, 0)) *# (значение, индекс массива, индекс в массиве)* result = []  
  
 while heap:  
 value, chunk\_idx, elem\_idx = heapq.heappop(heap)  
 result.append(value)  
  
 *# Добавляем следующий элемент из того же массива, если он есть* if elem\_idx + 1 < len(sorted\_chunks[chunk\_idx]):  
 next\_value = sorted\_chunks[chunk\_idx][elem\_idx + 1]  
 heapq.heappush(heap, (next\_value, chunk\_idx, elem\_idx + 1))  
  
 return result  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("=" \* 60)  
 print("ЛАБОРАТОРНАЯ РАБОТА: Параллельный Mergesort")  
 print("=" \* 60)  
  
 *# Параметры эксперимента* N\_test = 5\_000\_000 *# Размер массива для основного эксперимента* print(f"\nГенерация случайного массива размером N = {N\_test:,}...")  
 np.random.seed(42) *# Для воспроизводимости* data\_test = np.random.randint(-1\_000\_000, 1\_000\_000, size=N\_test).tolist()  
  
 print("\nЗамер времени для M процессов от 1 до 24:")  
 print("-" \* 60)  
  
 M\_values = []  
 real\_times = []  
 ideal\_times = []  
  
 T1 = None *# Время одного процесса* for M in range(1, 25):  
 *# Разделяем массив на M частей* chunk\_size = N\_test // M  
 ranges = []

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

for i in range(M):  
 start = i \* chunk\_size  
 end = (i + 1) \* chunk\_size if i < M - 1 else N\_test  
 ranges.append((start, end, data\_test))  
  
 *# Запускаем параллельную сортировку* t\_start = time.perf\_counter()  
  
 with multiprocessing.Pool(processes=M) as pool:  
 results = pool.starmap(worker\_sort, ranges)  
  
 *# Получаем отсортированные части и время работы каждого воркера* sorted\_chunks = [r[0] for r in results]  
 worker\_times = [r[1] for r in results]  
 max\_worker\_time = max(worker\_times)  
  
 *# Объединяем отсортированные части (k-way merge)* t\_merge\_start = time.perf\_counter()  
 final\_sorted = k\_way\_merge(sorted\_chunks)  
 t\_merge\_end = time.perf\_counter()  
 merge\_time = t\_merge\_end - t\_merge\_start  
  
 t\_end = time.perf\_counter()  
 total\_time = t\_end - t\_start  
  
 *# Время для одного процесса* if M == 1:  
 T1 = total\_time  
  
 M\_values.append(M)  
 real\_times.append(total\_time)  
 ideal\_times.append(T1 / M if T1 else 0)  
  
 print(f"M={M:2d} | Total={total\_time:7.4f}s | Workers={max\_worker\_time:7.4f}s | "  
 f"Merge={merge\_time:6.4f}s | Ideal={T1 / M if T1 else 0:7.4f}s")  
  
 print("-" \* 60)  
 print(f"\nБазовое время T1 (1 процесс): {T1:.4f} секунд")  
 print(f"Ускорение при 24 процессах: {T1 / real\_times[-1]:.2f}x")  
  
 *# ---------------- ГРАФИК ----------------* print("\nПостроение графика...")  
  
 fig, ax = plt.subplots(figsize=(12, 7))  
  
 ax.plot(M\_values, ideal\_times, "bo-", label="Идеальное время (T1/M)",  
 linewidth=2, markersize=8)  
 ax.plot(M\_values, real\_times, "ro-", label="Реальное время (с merge)",  
 linewidth=2, markersize=8)

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

ax.set\_title("Сравнение идеального и реального времени\nпри параллельной сортировке Mergesort",  
 fontsize=20, fontweight='bold')  
 ax.set\_xlabel("Число процессов (M)", fontsize=18, fontweight='bold')  
 ax.set\_ylabel("Время, сек", fontsize=18, fontweight='bold')  
  
 ax.tick\_params(axis='both', which='major', labelsize=14)  
 ax.legend(fontsize=14, loc='upper right')  
 ax.grid(True, alpha=0.3)  
  
 *# Добавляем аннотацию с ускорением* ax.text(0.02, 0.98, f'Размер данных: {N\_test:,}',  
 transform=ax.transAxes, fontsize=12, verticalalignment='top',  
 bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))  
  
 plt.tight\_layout()  
 plt.show()  
  
 print("\nГотово! График отображён.")

ПРИЛОЖЕНИЕ Б – ЭКСПЕРИМЕНТ 1

Генерация случайного массива размером N = 5,000,000...

Замер времени для M процессов от 1 до 24:

------------------------------------------------------------

M= 1 | Total=25.0592s | Workers=22.0867s | Merge=1.3754s | Ideal=25.0592s

M= 2 | Total=13.7156s | Workers=10.6284s | Merge=1.6166s | Ideal=12.5296s

M= 3 | Total=10.2092s | Workers= 7.0070s | Merge=1.6625s | Ideal= 8.3531s

M= 4 | Total= 8.6134s | Workers= 5.1941s | Merge=1.8095s | Ideal= 6.2648s

M= 5 | Total= 7.8316s | Workers= 4.1755s | Merge=1.8696s | Ideal= 5.0118s

M= 6 | Total= 7.1926s | Workers= 3.3681s | Merge=1.8632s | Ideal= 4.1765s

M= 7 | Total= 6.9636s | Workers= 2.9679s | Merge=1.8930s | Ideal= 3.5799s

M= 8 | Total= 6.6834s | Workers= 2.5909s | Merge=1.9417s | Ideal= 3.1324s

M= 9 | Total= 6.6632s | Workers= 2.2749s | Merge=2.0145s | Ideal= 2.7844s

M=10 | Total= 6.7787s | Workers= 2.1080s | Merge=2.0847s | Ideal= 2.5059s

M=11 | Total= 6.6611s | Workers= 1.8745s | Merge=2.0620s | Ideal= 2.2781s

M=12 | Total= 6.6782s | Workers= 1.8415s | Merge=2.0159s | Ideal= 2.0883s

M=13 | Total= 6.7605s | Workers= 1.7103s | Merge=2.0540s | Ideal= 1.9276s

M=14 | Total= 7.0102s | Workers= 1.6682s | Merge=2.0950s | Ideal= 1.7899s

M=15 | Total= 7.0596s | Workers= 1.4923s | Merge=2.0916s | Ideal= 1.6706s

M=16 | Total= 7.1520s | Workers= 1.4295s | Merge=2.1087s | Ideal= 1.5662s

M=17 | Total= 7.2829s | Workers= 1.3738s | Merge=2.1395s | Ideal= 1.4741s

M=18 | Total= 7.5712s | Workers= 1.2607s | Merge=2.2317s | Ideal= 1.3922s

M=19 | Total= 7.7336s | Workers= 1.1134s | Merge=2.1882s | Ideal= 1.3189s

M=20 | Total= 7.7341s | Workers= 1.0971s | Merge=2.1682s | Ideal= 1.2530s

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ Б

M=21 | Total= 7.9841s | Workers= 1.0058s | Merge=2.2656s | Ideal= 1.1933s

M=22 | Total= 8.1125s | Workers= 1.0119s | Merge=2.2145s | Ideal= 1.1391s

M=23 | Total= 8.3402s | Workers= 0.9244s | Merge=2.2216s | Ideal= 1.0895s

M=24 | Total= 8.4772s | Workers= 0.9038s | Merge=2.2544s | Ideal= 1.0441s

------------------------------------------------------------

Базовое время T1 (1 процесс): 25.0592 секунд

Ускорение при 24 процессах: 2.96x

Построение графика...

Готово! График отображён.