Operating Systems Assignment 4

High-Performance Parallel Sorting with Pthreads

Name: Teng Lei
Student ID: 999019730

December 26, 2025

1 Overview

This assignment implements `psort`, a parallel sorting program designed to sort large binary files containing fixed-size records (128 bytes). The primary goal is to leverage multicore architectures using pthreads to achieve performance gains over single-threaded sorting.

The solution adopts a **"Map-Reduce"** style parallelism:

1. **Map Phase:** The input data is partitioned into disjoint chunks, which are sorted independently by multiple threads in parallel.
2. **Reduce Phase:** The sorted chunks are merged into a single output file using a K-Way Merge algorithm powered by a Min-Heap.

2 Implementation Strategy

My implementation maximizes system resource utilization and ensures data consistency through the following key strategies:

2.1 Memory Mapping (mmap) & Double Buffering

Instead of standard file I/O (`read`/`write`), I utilized `mmap` to map files directly into the virtual address space.

- **Input:** Mapped as `PROT_READ`.
- **Buffer:** A buffer is used as an intermediate workspace. Data is copied from the input map to this buffer to prevent race conditions and allow in-place sorting without modifying the source. I used `memcpy()` to copy from input-map to buffer.

- **Output:** Mapped as `PROT_WRITE | PROT_READ`. The final merged results are written directly to this memory region.

Inaddition, before using `mmap()`, I used `fstat()` to get the information of the input file, and use `ftruncate()` to initialize the size of the output file.

2.2 Parallel Sorting

The workload is distributed based on the number of available CPU cores (`get_nprocs()`). The data range is split into N segments. Each thread runs `qsort` on its assigned segment within the buffer. This phase is parallel with no inter-thread communication required. I use the fucntion: `my_compare` to compare two records base on the first 4-byte

2.3 K-Way Merge with Min-Heap

After all threads complete local sorting, we have N sorted subarrays. Merging them efficiently is critical.

- **Data Structure:** A Min-Heap of size K (where K is the number of threads).
- **Algorithm:** The heap is initialized with the first element of each chunk. In each iteration, the global minimum (root) is extracted and written to the output. The root is then replaced by the next element from the same chunk, followed by a `heapify` operation.
- **Complexity:** This ensures the merge phase runs in $O(M \log K)$ time, where M is the total number of records.

```
while (heap.heap_size > 0) {
    // 1. Extract minimum (root) to output
    Node min_node = heap.nodes[0];
    output_map[output_index++] = min_node.record;

    // 2. Refill from the same chunk
    int chunk_idx = min_node.index;
    if (curr_idx[chunk_idx] < threads[chunk_idx].nmemb) {
        heap.nodes[0].record = base_addr[curr_idx[chunk_idx]];
        curr_idx[chunk_idx]++;
    } else {
        // Chunk exhausted, shrink heap
        heap.nodes[0] = heap.nodes[heap.heap_size - 1];
        heap.heap_size--;
    }
    // 3. Restore heap property
    if (heap.heap_size > 0) heapify(&heap, 0);
}
```

Listing 1: K-Way Merge Core Logic

3 Work Division Strategy

To handle cases where the number of records is not perfectly divisible by the thread count, I implemented a remainder distribution logic.

- **Base Count:** $\lfloor \text{TotalRecords}/\text{NumThreads} \rfloor$
- **Remainder:** $\text{TotalRecords}\%\text{NumThreads}$

The first *Remainder* threads receive one extra record. This ensures the load imbalance between any two threads is at most 1 record.

4 Correctness Verification

Correctness was rigorously tested using the following methods:

1. **Record Size Validation:** The program enforces the 128-byte record size constraint.
2. **Full Scan Verification:** I implemented a separate C program, `check_sorted.c`, to verify the output. This program maps the output file and iterates through all records to ensure the sorting invariant ($Key_i \leq Key_{i+1}$) holds for the entire file.

```
    // Core logic from check_sorted.c
    for (size_t i = 0; i < num_records - 1; i++) {
        if (map[i].key > map[i+1].key) {
            fprintf(stderr, "Error: Key %u > Key %u\n", ...);
            return 1;
        }
    }

```

3. **Size Integrity:** Verified that `input_size == output_size`.
4. **Persistence:** `fsync(fd_out)` is called before exit to flush the page cache to disk.

5 Performance Measurement & Analysis

Performance was measured using GNU `time` on a Linux environment. I compared the multi-threaded implementation against a single-threaded baseline on two dataset sizes. I also implement a similar C program: `sort_singular.c` which just use the main thread to sort.

5.1 Test Case 1: Large Dataset (1 Million Records, ≈ 128MB)

Metric	Single-Thread	Multi-Thread	Result
Real Time	1.402s	1.218s	**1.15x Speedup**
User Time	0.543s	0.917s	Increased (Overhead)
Sys Time	0.771s	0.877s	I/O Bound

Table 1: Performance on 1 Million Records

Analysis: While the 1.15x speedup seems modest, a deeper analysis reveals the impact of Disk I/O.

- **I/O Bound:** Both versions spent $\approx 0.8s$ in System time (paging 128MB from disk). Parallelism cannot accelerate disk bandwidth.
- **Compute Acceleration:** If we isolate the computing phase ($Real - Sys$):
  - Single-thread compute: $1.40s - 0.77s = 0.63s$
  - Multi-thread compute: $1.21s - 0.88s = 0.33s$
- **Conclusion:** The parallel sorting logic achieved a **nearly 2x speedup** in the computation phase, demonstrating true algorithmic efficiency masked by I/O latency.

5.2 Test Case 2: Small Dataset (100,000 Records, ≈ 12MB)

Metric	Single-Thread	Multi-Thread	Result
Real Time	0.142s	0.156s	**Slightly Slower**

Table 2: Performance on 100k Records

Analysis (Parallel Overhead): This result illustrates the cost of parallelism. For small datasets, the overhead of creating threads, context switching, and managing the merge heap outweighs the benefits of parallel sorting. This confirms that multi-threading is most effective for computationally intensive tasks where the workload is large enough to amortize the startup costs.

5.3 Test Case 3: Very Large Dataset (Approx. 68 Million Records, ≈ 8.17 GB)

To test the system under heavy load, I ran a test with a dataset significantly larger than physical memory cache, utilizing 16 CPU cores.

Metric	Single-Thread	Multi-Thread	Result
Real Time	2m 19.185s (139.2s)	1m 16.432s (76.4s)	**1.82x Speedup**
User Time	1m 19.969s (80.0s)	1m 25.021s (85.0s)	Similar Workload
Sys Time	0m 58.151s (58.2s)	0m 59.903s (59.9s)	I/O Bound

Table 3: Performance on 8 GB Dataset (16 Threads)

Analysis: This test case clearly demonstrates the scalability of the parallel implementation.

- **Significant Speedup:** The multi-threaded version reduced the total execution time by nearly 50%, achieving a speedup of **1.82x**.
- **Compute Bound Analysis:** When subtracting the I/O overhead (Sys Time $\approx 60s$):
  - Single-thread compute: $139.2s - 58.2s = 81.0s$
  - Multi-thread compute: $76.4s - 59.9s = 16.5s$
- **Massive Compute Efficiency:** The pure sorting/merging phase achieved a **4.9x speedup**. This indicates that while disk I/O remains a bottleneck, the parallel algorithm effectively utilizes the 16 cores to crush the computational part of the task.

6 Known Issues and Limitations

- **Merge Cost:** In this approach, I sorted the data parallel but merge the chunks only used the main thread. So maybe we can also use multi-thread to do the merge part.
- **Memory Constraint:** The program requires loading the entire file into memory. It cannot handle files larger than physical RAM (would cause thrashing). An external sorting algorithm would be needed for larger datasets.
- **Disk I/O Bottleneck:** As observed, execution time is dominated by disk I/O. Future optimizations could explore Asynchronous I/O (AIO) to overlap computation with disk reads/writes.