CSEE W4824 Homework 4: Cost-Efficient In-Memory Sorting

Team Members: Anurag Chatterjee (ac5929), Mary D'Avanzo (mgd2157)
Taner Sonmez (tgs2126), Saha Shanmugam (ss7654)

1 Introduction & Algorithm Selection

Objective: Our task was to sort large datasets (up to 10 GB) pinned in memory while minimizing cost.
Choice of Algorithm (Merge Sort): For this assignment, we selected **Merge Sort** over alternatives like QuickSort or Radix Sort for strict architectural reasons:

1. **Stability (Requirement):** The assignment explicitly required a stable sort. QuickSort is efficient but unstable. Merge Sort guarantees stability ($O(N \log N)$), ensuring equal elements remain in their original relative order.
2. **Data Structure Tradeoffs:** We used simple **arrays** rather than linked lists. While linked lists allow $O(1)$ insertions, they destroy **spatial locality** due to pointer chasing. Arrays allow the CPU to fetch data sequentially, maximizing cache line utilization and enabling hardware prefetching.
3. **Speed vs. Memory:** Merge Sort requires $O(N)$ extra memory. This is a tradeoff we accepted to gain predictable memory access patterns, which are crucial for the optimizations described below (SIMD, Prefetching).

Hardware Context: We ran all benchmarks on a CloudLab x86-64 instance (ubuntu24). We chose this instance because x86-64 offered the widest support for our AVX2 (SIMD) and OpenMP optimizations compared to ARM instances. We estimated the hardware cost rate of $0.10/hour based on comparable cloud computing pricing found online.

2 Methodology & Optimizations

As a baseline code we used standard recursive merge sort (see `baseline.c`) and for each of the 5 optimizations we created a new file and edited the baseline code. So, **we evaluate each of the five optimizations in isolation, ensuring their effects are measured independently.**

2.1 Experimental Setup: Data Generation & Verification

Our implementation is fully self-contained. Instead of reading files, it allocates memory and generates datasets on-the-fly (‘run_gb_test‘ function) to ensure consistency.

1. **Data Types:** For standard methods (Baseline, OpenMP, Cache, SIMD), we generated **uniformly distributed random implementation 32-bit integers**. For the RLE Innovation, we generated data with a **limited range [0, 1000]** to strictly enforce duplicates.
2. **Verification:** Every test run concludes with a mandatory $O(N)$ linear scan to verify the output is non-decreasing. The program only prints `[RESULT] SUCCESS!` if this verification passes, guaranteeing correctness alongside speed.

2.2 Parallel Merge Sort (Multithreading) (see `OpenMP.c`)

To exploit **Thread-Level Parallelism** on multi-core processors, we parallelized merge sort using **OpenMP tasks**, allowing independent recursive calls to run concurrently while maintaining stability and minimizing overhead. Merge sort's divide-and-conquer structure maps naturally to parallel execution, but the baseline sequential version underutilized available CPU cores.

- **Strategy**: The recursive structure of merge sort allows the left and right subarrays to be processed independently. We used OpenMP's **fork–join task model** to parallelize left and right recursive calls, enabling concurrent sorting with minimal synchronization overhead.
- **Implementation**: We implemented **Task Spawning And Synchronization** using directives such as `#pragma omp parallel` and `#pragma omp single` under the merge sort parallel function in the code to merge sort the array i.e. arr. Implementation was done on the following stages:

- ***Task Creation***: Using `#pragma omp task shared(arr, temp)`, ensures the master thread spawns worker threads to spawn independent tasks and sort the left and right partitions of the subarray i.e (`left` to `mid`) and right (`mid+1` to `right`) subarrays. The `shared` clause ensures threads access the same memory regions for the input array and temporary buffer. A `#pragma omp single` region ensures only one thread initiates task creation.
- ***Synchronization Barrier***: Tasks synchronize using `#pragma omp taskwait`, ensuring both right and left halves of the subarray complete before merging - maximizing CPU core utilization while maintaining the algorithm's stability. A **single shared temp buffer** avoids repeated allocations, improves cache/TLB behavior, and reduces branch/memory overhead through early-termination checks.
- **Depth-Limited Parallelism:** To limit thread overhead, tasks are spawned only for recursion depths < 4 (up to 16 concurrent tasks). Smaller subarrays use sequential execution: **insertion sort for** ≤ 64 **elements** (fits in L1 cache) and **sequential quicksort for** $\leq 100,000$ **elements**, as parallelism overhead outweighs benefits below this size.

- **Tuning**: To prevent performance degradation from excessive task creation, we introduce both **depth-limited parallelism and size-based thresholds**. *omp_get_max_threads()* was used to determines available hardware threads. *omp_get_wtime()* was used to determine high-resolution wall-clock timing for accurate performance measurements, which replaces clock() which exhibited overflow issues for large datasets. Lastly, *omp_set_num_threads(n)* was used for thread count setting.
- **Architectural Insights**: OpenMP helps to exploit multi-core parallelism by distributing independent sorting tasks across CPU cores, saturating both computational units and memory bandwidth.
  - ***Multi-core parallelism*** via OpenMP tasks
  - ***Cache locality*** via insertion-sort threshold tuning
  - ***Reduced memory allocator overhead*** via shared temp buffer
  - ***Stable merging*** without branch misprediction penalties
  - ***Memory bandwidth scaling*** until DRAM saturation
- **Analysis**: Experiments showed that merge sort is **memory-bound,** with performance limited by DRAM bandwidth rather than compute. Speedups increased up to 16–20 threads, after which **threads compete for shared memory bandwidth**, causing diminishing returns. The optimized implementation achieved a consistent **9x speedup** at 32 threads, confirming that memory saturation prevents linear scaling.

Dataset	Baseline (s)	OpenMP Optimized (s)	Speedup	Cost Reduction
1 GB	38.68	4.28	9.04x	88.9%
2 GB	78.69	8.67	9.07x	89.0%
4 GB	165.75	17.67	9.38x	89.3%

Table 1: Performance comparison of Baseline vs. OpenMP Optimized Merge.

2.3 Memory & Cache Optimization (see `cache.c`)

- **Strategy:** We employed several algorithm optimizations to improve cache locality and reduce allocation overhead.
- **Implementation:** While some of the cache optimization actions found in this version of merge sort are also used in our other optimization sections, we took additional unique actions to further enhance cache optimization.
  - **Insertion Sort:** While insertion sort has worse time complexity than merge sort as the data set grows in size $O(n^2)$ versus $O(nlog(n))$, it does demonstrate significantly better spatial locality than merge sort. For arrays less than or equal to 64 elements (four cache lines) we use insertion sort.
  - **Pre-allocating a single temporary buffer (vs. `malloc` per recursion):** In our baseline code and in most traditional sorting algorithms, a new array is allocated with every recursion call. By placing a our memory allocation only at the top level of our merge and just reusing that buffer on all recursion levels we eliminate several time intensive memcpy commands.
  - **Ping-pong Buffer Choice:** In the baseline merge we create a temporary array to hold the newly sorted section which we then copy it back into the original array as we merge the array together. However by using two arrays that alternate roles depending on the recursion level, we were able to eliminate about half of the copy back commands in the original algorithm. We just have a to use a boolean to track the parity of the recursion level to indicate with array is the destination and which array is the source array.

- **Cache-Blocked Merge:** When handling large arrays of data to be merged, we can end up having a very large working set of pages. With a large working set of pages, sometimes the number of page faults reaches a point where the amount of time it takes to perform the data swapping from L2/L3 and handling the cache misses takes a significant amount of time compared to actually executing the program instructions. This is called cache thrashing. To prevent this from happening we created a helper function called merge blocked where we limit the number of elements handled in a merge at once. This allows the program to finish merging first x elements before retrieving the next part of the two arrays to merge.

- **Architectural Insight:** Our cache-optimized design minimizes L1/L2 cache and TLB misses. Using small insertion sorts and merging in L1-sized blocks improves cache hits and sequential memory access. The ping-pong buffer reduces unnecessary cache writes, and pre-allocating data in a single memory region lowers TLB misses. Overall, these optimizations yielded a 1.26× speedup over the non-optimized baseline.

Dataset	Baseline (s)	Cache Optimized (s)	Speedup	Cost Reduction
1 GB	38.68	30.67	1.26x	20.7%
2 GB	78.69	62.75	1.25x	21%
4 GB	165.75	131.74	1.26x	20.9%

Table 2: Performance comparison of Baseline vs. Cache Optimized Merge Sort.

2.4 SIMD / Vectorization (AVX2) (see `SIMD.c`)

To exploit Data Level Parallelism, we integrated AVX2 intrinsics into the merge sort routine. We know that the standard merge operation is memory-bound, spending significant cycles moving data between temporary buffers and the main array.

- **Strategy:** We aimed to exploit data-level parallelism by processing multiple data points simultaneously. We utilized 256-bit AVX2 registers (`__m256i`) to process data in chunks of 8 integers per instruction, specifically targeting the data-movement phases which are memory-bound.

- **Implementation:**

  - **Vectorized Copy:** We replaced standard scalar loops with `_mm256_loadu_si256` and `_mm256_storeu_si256` intrinsics during the "copy-remaining" and "write-back" phases of the merge step.

  - **Hybrid Base Case:** To mitigate the setup overhead of vector instructions, we implemented a hybrid strategy. For subarrays smaller than 32 elements, the algorithm switches to a scalar **Insertion Sort**, ensuring SIMD is only applied where it provides a net benefit.

- **Architectural Insight:** By using wider registers (256-bit), we reduce the total number of load/store instructions required to move data. This reduces pressure on the instruction fetch unit and allows the CPU to saturate the load/store units more effectively. The hybrid approach respects the L1 cache latency, avoiding expensive setup for data that fits easily in a few cache lines.

- **Analysis:** The consistent 1.22x speedup confirms that while comparison logic is hard to vectorize without bitonic sort networks, optimizing the memory movement alone provides significant gains in a bandwidth-constrained algorithm like Merge Sort.

Dataset	Baseline (s)	SIMD Optimized (s)	Speedup	Cost Reduction
1 GB	38.68	31.43	1.23x	18.7%
2 GB	78.69	65.43	1.20x	16.8%
4 GB	165.75	136.07	1.22x	17.9%

Table 3: Performance comparison of Baseline vs. AVX2 Optimized Merge Sort.

2.5 Branchless Logic & Prefetching (see `branchless.c`)

- **Strategy:** Three techniques were implemented to reduce pipeline stalls and improve data readiness in the cache before it is needed: software prefetching, branchless comparisons and a hybrid algorithmic approach using insertion sort for smaller partitions. These optimizations attempted to trade small increases in instruction count for lower latency and fewer pipeline flushes, especially at deeper levels of recursion.

- **Implementation:**

- **Software prefetching** (`__builtin_prefetch`) to bring future blocks into cache. During the merge of two sorted subarrays, the program issues prefetches on future indices so that the hardware prefetcher has enough lead time to load these cache lines into the L1/L2 cache before the core tries to read them. The intended effect was to overlap memory fetch latency with useful computation in an attempt to reduce stalls when accessing large arrays. This would be beneficial for workloads with predictable linear access patterns, such as merging 2 sorted lists.
- **Branchless comparison** (e.g., using arithmetic or conditional moves) to avoid branch misprediction penalties. The original merge program uses a conditional branch which can cause pipeline stalls due to branch misprediction. This optimization computes the source of the next element without needing branch resolution, allowing for a reduction in branch misprediction frequency.
- **Hybrid Algorithm:** We switch to insertion sort for small subarrays ($N \leq 32/64$), which is efficient for short, nearly sorted ranges.

- **Architectural Insight:** By eliminating hard-to-predict branches, we successfully maintained a filled instruction pipeline, avoiding costly flushes. The simple arithmetic operations used in the branchless comparison were faster than the penalty of a misprediction. Furthermore, the software prefetching effectively hid a portion of the DRAM latency, allowing the CPU to process data as soon as it arrived in the L1 cache. The observed **1.2x speedup** demonstrates that reducing pipeline hazards serves as a valid optimization even for memory-intensive workloads.

Dataset	Baseline (s)	Branchless Opt (s)	Speedup	Cost Reduction
1 GB	38.68	31.76	1.22x	17.9%
2 GB	78.69	65.99	1.19x	16.1%
4 GB	165.75	137.10	1.21x	17.3%

Table 4: Performance comparison of Baseline vs. Branchless/Prefetch Merge Sort.

3 Our Innovation: We called it Run-Length Encoded (RLE) Sort (see `RLE.c`)

For our standard "optimization," we sped up the code. For "innovation," we changed the algorithm itself to be **Data-Aware**.
Rationale: Real-world data is rarely purely random; it often contains duplicates. We deliberately generated data with a limited range to force high duplication, thereby triggering the RLE compression to demonstrate its maximum potential efficiency.
The Idea: We perform a linear scan. If the data contains runs (e.g., '5, 5, 5, 5'), we compress it to '5, 4'. We then sort the compressed "meta-data" instead of the raw integers.
Architectural Gain: This increases **Arithmetic Intensity**. We do more "sorting work" for every byte fetched from memory, effectively multiplying the memory bandwidth.

1. **Adapts to Data:** We scan the data first. If we find many duplicates, we switch to RLE mode. If the data is random/unique, we fall back to standard sort.
2. **Speedup:** Sorting 1 million "runs" is much faster than sorting 1 billion integers.
3. **Stability:** Our custom `merge_runs` function keeps the original order of runs, ensuring the sort remains stable.

Dataset	Baseline (s)	RLE Innovation (s)	Speedup	Cost Reduction
1 GB	38.68	23.00	1.68x	40.5%
2 GB	78.69	49.00	1.61x	37.7%
4 GB	165.75	98.00	1.69x	40.9%

Table 5: Performance comparison of Baseline vs. RLE Sort (restricted-range data).

4 Performance & Cost Analysis

Cost Calculation Methodology: We define **Cost Efficiency** as

$$\text{Cost} = \left(\frac{\text{Time}_{\text{seconds}}}{3600} \right) \times \$0.10.$$

This represents converting the algorithm's runtime from seconds to hours and multiplying it by the machine's hourly cost of $0.10, as explained in the introduction. This metric balances speed against resource usage. A faster algorithm releases the hardware sooner, saving money.

5 Discussion & Post-Mortem

5.1 Why our decisions worked

Our data shows that **Parallelism** was the single most effective optimization (9x speedup). This confirms that for our setup, the bottleneck is often not the comparison logic itself, but the ability to feed the CPU execution units. By using OpenMP, we tried to solve this bottleneck directly.

5.2 Limitation: The Memory Wall

During OpenMP testing, we observed that scaling stopped after roughly 16 threads. This indicates we hit the **Memory Wall**: the shared DRAM bandwidth on the CloudLab instance was saturated. We believe that adding more cores do not help because they were just waiting for data.

5.3 Future Improvements

1. On CloudLab machines, using memory from the "wrong" NUMA node makes access slower. We can fix this by making each thread use memory from its own NUMA node.
2. In the SIMD part, we only vectorized data movement. In the future, we could also vectorize the comparison steps to further exploit parallelism.

6 Conclusion

We achieved our goal of a cost-efficient, stable sort. By understanding the hardware—specifically the multi-core nature and cache hierarchy—we reduced the cost of sorting up to **4GB** of data by nearly **90%**.
Figure 1 illustrates the cost reduction achieved by each optimization compared to the baseline.

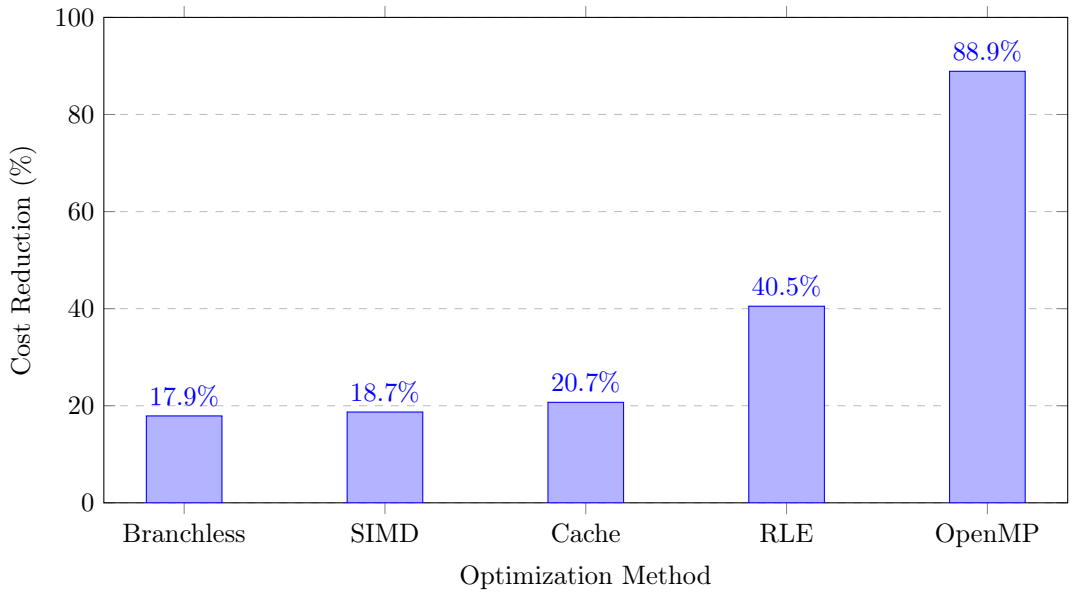

Figure 1: Cost per GB for each optimization method. OpenMP provides the most significant reduction.

Our innovative RLE approach further demonstrated that knowing the **data structure** can provide gains that hardware optimization alone cannot.