RESEARCH ARTICLE

Optimizing Static Data Partitioning for Cost-Aware Parallel Sorting in Heterogeneous Distributed Systems

Joeniño Cainday and Junar Landicho

Department of Computer Science, University of Science and Technology of Southern Philippines, Cagayan De Oro City, Philippines

ARTICLE HISTORY
Compiled June 4, 2025

1. Introduction

The widespread adoption of cloud computing has heightened the importance of efficient resource allocation in distributed systems. Traditional data partitioning methods—such as uniform or round-robin—often fall short in heterogeneous environments, leading to load imbalance and poor utilization of resources. These challenges are amplified when data is distributed across geographically distant nodes, where differences in hardware capabilities and increased communication overhead introduce latency and reduce overall system throughput [48].

In addition to performance concerns, cost has become a central factor in partitioning decisions with the rise of pay-as-you-go cloud platforms [5]. While premium instances may offer better performance, this is not guaranteed across providers. Consequently, partitioning strategies must account for both execution time and monetary cost. Ideally, partitioning decisions should fall on the Pareto front—where no improvement in one metric, such as speed or cost, is possible without compromising the other [48]. Inefficient strategies can result in inflated expenses or system failures, such as memory overflows. Thus, effective solutions must balance these competing objectives.

This study addresses the problem of static data partitioning for parallel sorting in cost- and resource-constrained distributed environments. To capture the heterogeneity of computing nodes, we consider variations in usage costs, memory capacities, and processing capabilities. Specifically, we abstract the combined effects of communication latency, bandwidth, and computation speed into a unified processing throughput metric, defined as the relative rate at which a node can complete its assigned tasks. For example, a node with twice the throughput of another can process and communicate data approximately twice as fast. In practice, this throughput ($\mathrm{Perf}[i]$) could be estimated through benchmarking specific instance types or by analyzing historical performance data. To address this multifaceted optimization problem, we propose a linear programming (LP)-based approach that produces cost-aware and load-balanced data partitioning schemes for parallel sorting. The model seeks to minimize execution time (makespan) while also reducing overall resource costs, without incurring significant performance penalties. In contrast to heuristic or sampling-based approaches, the LP formulation guarantees globally optimal solutions under linear constraints. Furthermore, its polynomial-time solvability [57] makes it well-suited for deployment in real-world heterogeneous cloud environments.

2. Related Work

In the domain of parallel sorting, many algorithms operate under the assumption of a homogeneous computing environment. For instance, Parallel Sorting by Regular Sampling (PSRS) selects pivots to divide the dataset into equally sized partitions [51], and commonly used benchmarks are based on synthetically generated data with uniform or similarly idealized distributions. However, such methods do not account for the heterogeneity inherent in real-world distributed systems. Addressing this limitation, Monga and Lodhi [53] introduced a modified version of PSRS designed for heterogeneous environments. Their approach improves load balancing by allocating data in proportion to each node's relative throughput. Nevertheless, their method does not consider other crucial system attributes. Additionally, the reliance on regular sampling introduces computational overhead associated with global sampling and all-to-all redistribution. Their procedure also entails initial local sorting and sampling prior to defining data ranges, which may result in uneven memory usage and load imbalance during the redistribution phase.

Moreover, the effectiveness of their partitioning strategy may be sensitive to the characteristics of the input dataset. For example, their use of custom dataset construction—potentially employing techniques such as least common multiple (LCM)-based schemes—may not guarantee optimal or balanced partitioning for arbitrary or non-uniform data distributions, potentially leading to suboptimal resource utilization. While a commonly used strategy for approximating ideal continuous allocations in integer-constrained systems is the largest remainder method (also known as Hamilton's method), which is widely applied in apportionment and resource allocation problems, its application in H-PSRS would still be subject to the limitations of sampling-based pivot selection.

In the context of distributed systems, this research distinguishes itself from existing works such as Yoon and Kamal's "Optimal Dataset Allocation in Distributed Heterogeneous Clouds" [48] by adopting a more focused approach for parallel sorting. While Yoon and Kamal broadly tackle optimal dataset allocation across geographically distributed cloud data centers using a multi-objective linear programming (LP) model to find a Pareto front that simultaneously minimizes processing time and monetary cost, this research employs a single, scalarized LP objective for makespan minimization with cost as weighted secondary consideration. This allows for a direct and computationally efficient solution tailored to data partitioning in heterogeneous nodes, simplifying complex inter-node considerations like detailed communication and data transfer speeds into a generalized "throughput" metric, which encapsulates latency, bandwidth, and compute speeds, for practical application in parallel sorting tasks.

In big-data systems like Spark, dynamic partitioning and scheduling algorithms have been proposed. For example,[52] developed a dynamic partitioning strategy for intermediate Spark data to mitigate skew, and a greedy scheduling method that considers node speed. They find that balanced partitioning significantly lowers completion time. Our work differs by focusing on static initial partitioning with explicit metrics, rather than in-job rebalancing.

3. Methodology

3.1. Baseline Approach: H-PSRS

As established in Section 2, existing heterogeneous sorting approaches, particularly H-PSRS by Monga and Lodhi [**?**], provide a foundation for load-balanced sorting but have significant limitations in cost optimization and global optimality. We use H-PSRS as our baseline comparison since it represents the current state-of-the-art in heterogeneous parallel sorting.

H-PSRS operates through four sequential phases: local sorting and sampling, pivot selection via coordinator, all-to-all data redistribution, and final merge. While this approach successfully adapts PSRS for heterogeneous environments by proportional workload allocation, our analysis identified key areas for improvement that directly inform our design decisions.

3.1.1. H-PSRS Complexity Analysis

We analyze the computational and space complexity of H-PSRS across its four phases:

Phase I (Local Sorting and Sampling):

- *Time Complexity:* Each node i sorts its proportionally allocated data chunk of size $n_i =$

$n \cdot \frac{\text{Perf}[i]}{\sum_{j=0}^{p-1} \text{Perf}[j]}$, requiring $O(n_i \log n_i)$ time. Sample extraction takes $O((p-1) \cdot \text{Perf}[i])$ time per node. Since operations occur in parallel, the bottleneck is $\max_i(n_i \log n_i + (p-1) \cdot \text{Perf}[i])$.
- *Space Complexity:* Each node requires $O(n_i)$ space for its data chunk and $O(p \cdot \text{Perf}[i])$ space for samples.

Phase II (Pivot Selection):

- *Time Complexity:* The coordinator collects $O(p^2)$ total samples and sorts them in $O(p^2 \log p)$ time, creating a sequential bottleneck that cannot be parallelized.
- *Space Complexity:* The coordinator requires $O(p^2)$ space to store and process all samples from nodes.

Phase III (Data Redistribution):

- *Time Complexity:* Each node partitions its data using $(p-1)$ pivots in $O(n_i \log p)$ time using binary search. All-to-all communication involves $O(n)$ total data movement across the network, with each node potentially sending and receiving $O(n/p)$ data on average.
- *Space Complexity:* Each node requires temporary buffers of size $O(n_i)$ for partitioning and $O(n/p)$ for receiving data from other nodes.

Phase IV (Final Merge and Sort):

- *Time Complexity:* Each node merges p sorted sublists (one from each node) in $O(n_i \log p)$ time, then performs final local sorting in $O(n_i \log n_i)$ time. Coordinator concatenation requires $O(n)$ time.
- *Space Complexity:* Each node uses $O(n_i)$ space for merged data, and the coordinator requires $O(n)$ space for the final result.

Overall Complexity:

- *Total Time Complexity:* $O(\max_i(n_i \log n_i) + p^2 \log p + n)$
  - The $O(p^2 \log p)$ pivot selection creates a sequential bottleneck that limits scalability
  - Communication overhead $O(n)$ becomes significant in distributed environments
  - Under balanced allocation, $\max_i(n_i \log n_i) \approx O(\frac{n \log n}{p})$
- *Total Space Complexity:* $O(n + p^2)$, where $O(n)$ is distributed across nodes and $O(p^2)$ is concentrated at the coordinator

3.2. Proposed Approach: H-PSLP

Building upon the limitations identified in Section 2, we propose Heterogeneous Parallel Sorting by Linear Programming (H-PSLP). Our approach fundamentally redesigns the resource allocation strategy by replacing heuristic sampling with mathematical optimization, directly addressing the cost-awareness and global optimality gaps in existing methods.

3.2.1. Linear Programming Formulation

Our LP model optimizes workload allocation by minimizing a scalarized objective that combines makespan and cost. This approach assumes linearity in performance and cost metrics; while this is a common simplification for LP, it may not fully capture complex non-linear behaviors (e.g., network congestion, tiered pricing).

Algorithm 1 Heterogeneous Parallel Sorting by Linear Programming (H-PSLP)

```
Require: Dataset $S$ of size $n$, nodes $N_0, \ldots, N_{p-1}$ with performance Perf, memory Mem, cost Cost
Ensure: Sorted dataset $S'$ with optimized makespan and cost
Phase I: Linear Optimization
Solve LP model (Section 3.2.1) to obtain optimal workload allocation $x[i]$

Phase II: Data Partitioning
Apply largest remainder method to convert fractional LP solution to integer allocation

Phase III: Parallel Local Sorting
for each node $N_i$ in parallel do
    Sort assigned chunk $C_i$ locally: $O(|C_i| \log |C_i|)$
end for

Phase IV: Multi-way Merge
Coordinator performs $p$-way merge using min-heap: $O(n \log p)$
return Sorted dataset $S'$
```

$$\begin{aligned}
\text{minimize} \quad & T + \lambda \sum_{i=0}^{p-1} \mathrm{Cost}[i] \cdot \frac{x[i] \cdot \beta}{\mathrm{Perf}[i]} && (1)\\
\text{subject to} \quad & \sum_{i=0}^{p-1} x[i] = n && (2)\\
& x[i] \leq \mathrm{Mem}[i] \quad \forall i \in \{0, \ldots, p-1\} && (3)\\
& T \geq \frac{x[i] \cdot \beta}{\mathrm{Perf}[i]} \quad \forall i \in \{0, \ldots, p-1\} && (4)\\
& x[i] \geq 0 \quad \forall i \in \{0, \ldots, p-1\} && (5)
\end{aligned}$$

where T represents the makespan, $x[i]$ is the workload assigned to node i, β is a computational complexity factor, λ is the cost-performance trade-off parameter, and p is the number of nodes. The factor β normalizes the workload $x[i]$ to a time unit, effectively encapsulating the constant factors associated with the sorting algorithm's per-element operations (e.g., the log factor in $O(N \log N)$ complexity). The parameter λ allows for adjusting the relative importance of execution time versus monetary cost in the optimization; a higher λ places greater emphasis on reducing cost, potentially at the expense of a slightly increased makespan, thus enabling exploration of the Pareto front.

3.2.2. H-PSLP Complexity Analysis

We analyze the computational and space complexity of H-PSLP across all four phases:

Phase I (Linear Optimization):

- *Time Complexity:* The LP formulation contains p variables and $2p + 1$ constraints. Using interior-point methods, the solver requires $O(p^3)$ time in the worst case. Modern specialized LP solvers can achieve better performance, with some reaching $O(p^{2.5})$ for well-conditioned problems.
- *Space Complexity:* The constraint matrix requires $O(p^2)$ space, with additional $O(p)$ space for variables and dual solutions.

Phase II (Data Partitioning):

- *Time Complexity:* Converting fractional LP solutions to integer allocations using the largest remainder method requires $O(p \log p)$ time for sorting fractional parts, plus $O(p)$ for remainder distribution.
- *Space Complexity:* Temporary arrays for fractional parts and sorted indices require $O(p)$ space.

Phase III (Parallel Local Sorting):

- *Time Complexity:* Each node i sorts its assigned chunk C_i of size n_i in $O(n_i \log n_i)$ time. Since sorting occurs in parallel, the bottleneck is $\max_i(n_i \log n_i)$. Under optimal LP allocation, this approaches $O(\frac{n \log n}{p})$ per node.
- *Space Complexity:* Each node requires $O(n_i)$ space for its data chunk. Total distributed space is $O(n)$.

Phase IV (Multi-way Merge):

- *Time Complexity:* The p-way merge using a min-heap processes n elements with $O(\log p)$ operations per element, resulting in $O(n \log p)$ total time. This phase is centralized at the coordinator, which could become a bottleneck for extremely large datasets if the coordinator node's resources are insufficient.
- *Space Complexity:* The coordinator requires $O(n)$ space for the final sorted array and $O(p)$ space for the min-heap structure.

Overall Complexity:

- *Total Time Complexity:* $O(p^3 + p \log p + \frac{n \log n}{p} + n \log p)$
  - For large datasets where $n \gg p^3$, this simplifies to $O(\frac{n \log n}{p} + n \log p)$
  - The LP overhead $O(p^3)$ becomes negligible as n grows, making the algorithm scale well with data size
- *Total Space Complexity:* $O(n + p^2)$, where $O(n)$ is for data storage and $O(p^2)$ is for LP formulation

Comparison with H-PSRS:

- *Time Improvement:* H-PSLP eliminates the $O(p^2 \log p)$ sampling bottleneck and $O(n)$ all-to-all communication overhead present in H-PSRS
- *Space Improvement:* H-PSLP requires $O(p^2)$ space for LP formulation vs. H-PSRS's $O(p^2)$ for sample storage, but eliminates temporary redistribution buffers
- *Scalability:* H-PSLP scales better with increasing node count p due to the elimination of quadratic sampling overhead

3.3. Key Advantages of H-PSLP

Compared to H-PSRS, our approach offers several advantages:

(1) **Global Optimality:** LP formulation guarantees globally optimal solutions under linear constraints, unlike heuristic sampling approaches.
(2) **Cost Awareness:** Explicit incorporation of monetary costs in resource allocation decisions, crucial for cloud environments.
(3) **Memory Constraint Handling:** Direct enforcement of memory limits prevents system failures due to resource exhaustion.
(4) **Reduced Communication Overhead:** Elimination of all-to-all redistribution phase significantly reduces network traffic.
(5) **Robustness to Input Data Characteristics:** The workload allocation strategy is determined by node capabilities and costs, making the algorithm's performance less sensitive to the specific statistical distribution of the input data, providing consistent optimization across diverse datasets.

4. Experimental Setup

4.1. Simulation Environment

We implement a comprehensive simulation framework to evaluate the performance of H-PSLP against the baseline H-PSRS approach in heterogeneous distributed environments. The simulation is designed to model realistic cloud computing scenarios with varying node capabilities, costs, and workload characteristics.

4.1.1. Cluster Configuration Generation

Our experimental framework generates heterogeneous cluster configurations using a systematic approach that ensures realistic diversity in node capabilities:

- **Node Count:** Experiments are conducted with cluster sizes ranging from 3 to 8 nodes
- **Throughput Distribution:** Relative node throughput values are randomly sampled from the range [1, 10], representing the performance ratio compared to a baseline node
- **Memory Allocation:** Total cluster memory is set to 150% of the dataset size to simulate realistic overprovisioning scenarios. Individual node memory is allocated using random fractions that sum to unity, ensuring diverse memory constraints
- **Cost Structure:** Node billing costs are sampled uniformly from [0.1, 1.0] cost units per time unit, reflecting the heterogeneous pricing models in cloud environments

Dataset Size Constraints: Following Monga and Lodhi's approach, we use LCM-based synthetic dataset sizing to ensure exact proportional allocation for H-PSRS. The dataset size is calculated as:

$$\text{DataSize} = k \times \sum_{i=0}^{p-1} \text{Perf}[i] \times \text{LCM}(\text{Perf}[0], \ldots, \text{Perf}[p-1]) \tag{6}$$

where k is a scaling factor. While H-PSLP uses the largest remainder method and doesn't require this constraint, we apply it consistently for fair comparison.

4.1.2. Dataset Generation and Characteristics

To evaluate algorithm robustness across different data distributions, we generate datasets with varying statistical properties:

- **Uniform Distribution:** Random integers uniformly distributed across the range [0, n]
- **Zipf Distribution:** Power-law distribution with parameter $\alpha = 2.0$, simulating real-world data skewness
- **Exponential Distribution:** Exponentially distributed values with scale parameter $n/5$
- **Gaussian Distribution:** Normal distribution with configurable mean and standard deviation

Dataset sizes range from 10,000 to 100,000 elements, all conforming to the LCM-based sizing requirement described above to ensure compatibility with both algorithms. This provides sufficient scale to observe performance differences while maintaining reasonable simulation runtime.

4.1.3. Performance Metrics and Measurement

The simulation framework measures the following key performance indicators:

- **Makespan:** Total execution time from start to completion, measured in microseconds
- **Total Cost:** Cumulative monetary cost across all nodes, calculated as the sum of individual node costs

- **Load Balance:** Distribution of workload across nodes, measured as the coefficient of variation of assigned data sizes
- **Memory Utilization:** Percentage of allocated memory used by each node during execution
- **Throughput Efficiency:** Ratio of theoretical optimal throughput to observed throughput

4.1.4. Simulation Methodology

The simulation employs Python's multiprocessing framework to model parallel execution accurately:

(1) **Process Isolation:** Each node is simulated as a separate process with dedicated memory space and processing capabilities
(2) **Communication Modeling:** Inter-node communication is modeled using message queues, introducing realistic latency and synchronization overhead
(3) **Performance Scaling:** Node performance differences are simulated by scaling execution time inversely proportional to throughput values
(4) **Resource Constraints:** Memory limits are enforced to prevent allocation beyond node capacity

4.2. Algorithm Implementations

Both algorithms are implemented as described in their respective methodology sections. The H-PSRS baseline strictly adheres to Monga and Lodhi's original specification, while H-PSLP implements our proposed LP-based approach using OR-Tools CBC solver for optimization.

4.3. Experimental Design

4.3.1. Parameter Variations

We conduct experiments across multiple parameter dimensions:

- **Cluster Size:** 3, 4, 6, and 8 nodes
- **Dataset Size:** 10K, 25K, 50K, and 100K elements (all LCM-adjusted for fair comparison)
- **Data Distribution:** Uniform, Zipf, Exponential, and Gaussian
- **Cost-Performance Trade-off:** $\lambda \in \{10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}\}$
- **Memory Overprovision Factor:** 1.2, 1.5, and 2.0

4.3.2. Experimental Procedure

Each experimental configuration is repeated 10 times with different random seeds to ensure statistical significance. The simulation procedure follows these steps:

(1) Generate cluster configuration with specified parameters
(2) Create dataset with chosen distribution and LCM-adjusted size for algorithm compatibility
(3) Execute H-PSRS baseline algorithm using proportional allocation
(4) Execute H-PSLP algorithm with multiple λ values using LP-optimized allocation
(5) Record performance metrics for all approaches
(6) Analyze results for statistical significance using paired t-tests

4.4. Validation and Correctness

To ensure simulation accuracy, we implement several validation mechanisms:

- **Correctness Verification:** All sorting results are validated against Python's built-in sorted() function

- **Resource Constraint Validation:** Memory usage is monitored to ensure no node exceeds its allocated capacity
- **Load Balance Verification:** Workload distribution is verified against theoretical allocations for both proportional (H-PSRS) and LP-optimized (H-PSLP) approaches
- **Performance Consistency:** Multiple runs with identical parameters show consistent results within expected variance
- **Algorithm Compatibility:** Dataset sizes are verified to satisfy LCM requirements for H-PSRS while confirming H-PSLP handles arbitrary sizes correctly