RESEARCH ARTICLE

Optimized Data Partitioning for Parallel Sorting on Heterogeneous Distributed Systems using Linear Programming

Joeniño Cainday and Dr. Junar Landicho

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

ARTICLE HISTORY
Compiled May 6, 2025

ABSTRACT
Do not read this as its just a placeholder. This research addresses the optimization of data partitioning for parallel computations in heterogeneous distributed systems, with the dual objective of minimizing financial expenditure and execution time. We develop a Mixed Integer Programming (MIP) model that incorporates critical system constraints including CPU speed, memory capacity, memory latency, network latency, and budget limitations. Our model replaces the heuristic partitioning phase of a parallel quicksort algorithm, which serves as our performance benchmark. Utilizing synthetic datasets representing both workload and node specifications, our simulations focus on pre-execution data partitioning decisions. The goal is to provide a framework for financial and time optimization within cloud computing environments, where efficient resource allocation and budget management are essential. Results demonstrate the effectiveness of our MIP-based approach in achieving improved load distribution and reduced runtime compared to traditional heuristics.

KEYWORDS
Data Partitioning; Linear Programming; Heterogeneous Distributed Systems; Cost Optimization; Makespan Minimization

1. Introduction

The widespread adoption of cloud computing and serverless architectures has increased the importance of efficient resource allocation in distributed systems. This research addresses the challenge of statically partitioning data across heterogeneous computing environments, where nodes vary in processing speed, memory capacity, network latency, and monetary implications.

We formulate data partitioning as a Linear Programming (LP) problem aimed at minimizing execution time (makespan) while adhering to defined constraints. Our approach replaces traditional heuristic-based partitioning in parallel sorting algorithms with a novel LP model that captures comprehensive system characteristics to produce balanced, cost-aware data distributions. Our linear formulation allows polynomial-time solvability, providing optimal partitioning under linear constraints without incurring computational intractability.

1.1. Background and Motivation

Distributed computing enables parallel processing of large-scale datasets across multiple nodes. However, in heterogeneous systems with diverse computational capabilities, simplistic data partitioning strategies often result in load imbalances and inefficient resource utilization[48]. Common approaches like uniform distribution or partitioning based solely on heuristics typically fail to consider system compositions in cloud environments.

Cloud computing introduces new complexities to data partitioning. These environments offer flexibility with pay-as-you-go pricing models where computing resources are available on demand[5]. This necessitates re-evaluating traditional partitioning methodologies to explicitly consider economic factors alongside performance metrics. Effective strategies must balance high performance with budget limitations—minimizing both computational time and overall expenditure for cloud resource utilization.

While Monga and Lodhi[53] demonstrated benefits of heterogeneity-aware allocation, their model primarily considered CPU speed alone. Our work introduces a more sophisticated LP-based model that integrates multiple system attributes to achieve globally optimized data distributions, providing a more realistic and budget-conscious approach to optimized data processing.

1.2. Cost Considerations in Heterogeneous Distributed Systems

In this context, cost refers to the pricing of cloud compute nodes. While higher-priced instances often imply better performance, this is not guaranteed, as pricing and performance can vary across vendors. Each node has attributes including processing speed, memory capacity, network characteristics, and pricing models inspired by real cloud offerings, reflecting typical variations in heterogeneous cloud deployments.

Allocating data across heterogeneous resources inherently involves performance-cost tradeoffs[48]. Balancing processing time with financial implications when assigning datasets to different machines is critical. Effective data partitioning can significantly reduce both runtime and monetary cost, while suboptimal partitioning leads to inflated expenses or resource limitations like memory overflows.

This paper specifically addresses statically partitioning large synthetic datasets before executing parallel sorting across simulated heterogeneous environments. Our primary objective is to minimize the overall execution time (makespan) while ensuring we satisfy the defined constraints. As a secondary objective, we aim to identify the most cost-effective solution with an equivalent makespan, using a lexicographic optimization approach. The controlled synthetic environment enables repeatable experiments independent of real-world cloud variability.

Linear programming offers a powerful and tractable mathematical framework for such optimization problems, especially when:

- Data volumes are represented as continuous variables;
- Constraints and objectives remain linear;
- No integer or combinatorial decisions are involved;

This allows the use of efficient solvers that provide globally optimal solutions in polynomial time [57]. By leveraging LP, we avoid the complexity of NP-hard integer-based scheduling, while still capturing key system characteristics through continuous modeling of load, speed, and cost.

1.3. Research Contributions

This paper contributes the following:

(1) A linear programming (LP) formulation for initial data partitioning in parallel quicksort on heterogeneous systems;
(2) An integration of the LP model into an existing heterogeneous sorting framework;
(3) A performance evaluation using synthetic datasets with extended metrics;
(4) A reproducible experimental setup and discussion of directions for future work.

2. Related Work

Heterogeneous scheduling and data allocation have been studied in various contexts. For instance,[48] considered dataset allocation across geo-distributed clouds with two objectives (processing time and cost). They formulated a linear program to place data blocks on VMs to minimize a weighted sum of time and cost, demonstrating Pareto trade-offs. Similarly,[49] used a linear model for data assignment in a hybrid heterogeneous processing environment. These works focus on resource assignment rather than within-job data partitioning, but they highlight the utility of mathematical programming for cloud cost-performance optimization.

Classic scheduling theory demonstrates that minimizing makespan on unrelated machines is NP-hard when tasks must be assigned discretely[50]. However, in our case, we model data volumes as continuous variables without requiring integer decision variables. This places our problem within the class of linear programs, which are solvable in polynomial time using interior-point methods or simplex-based solvers [57]. Thus, our static data partitioning model, while inspired by hard scheduling problems, does not inherit their computational complexity due to the absence of discrete allocation and the use of continuous variables.

In the domain of parallel sorting, many algorithms assume a homogeneous machine model. For instance, Parallel Sorting by Regular Sampling (PSRS) chooses pivots to create equally-sized partitions[51], and typical benchmarks use randomly-generated data with uniform or other distributions. These benchmarks (e.g., uniform 32-bit integer inputs) guide our synthetic data choices. However, PSRS and related methods do not account for cost or heterogeneous speeds.

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 cost metrics, rather than in-job rebalancing. To the best of our knowledge, prior work has not explicitly applied linear programming to cost-aware static data partitioning in heterogeneous cloud sort workloads.

We build upon the model proposed by Monga and Lodhi[53], which partitions data across heterogeneous nodes based on CPU performance to balance load during parallel sorting. However, their approach does not consider critical real-world factors such as network latency or resource cost—key limitations in cloud and serverless settings. Moreover, their method performs initial local sorting and sampling before defining data ranges, which can lead to uneven memory usage and load imbalance during the redistribution phase.

3. Problem Formulation

We now formalize the partitioning problem based on the system characteristics introduced in Section 1.3. To ensure full control over system parameters and facilitate reproducibility, we operate within a synthetic environment. Each node is defined by parameters such as processing speed, memory capacity, network latency, bandwidth, and cost, inspired by real-world cloud configurations (e.g., AWS, GCP). This abstraction enables rigorous evaluation of partitioning strategies without relying on live infrastructure.

3.1. System Model

We partition the dataset into contiguous blocks of size d, assigning each block to a unique node. Consider a heterogeneous cluster with N nodes, where each node i is characterized by the following parameters:

- d_i: Assigned data volume (in units of d)
- r_i: Processing rate (in d/t — data units per time unit)
- m_i: Maximum data volume the node can handle at once (in units of d)
- ℓ_i: Network latency (in time units t)
- b_i: Network bandwidth (in d/t — data units per time unit)
- u_i: Usage billing rate (in c/t — cost units per time unit)

The units above are illustrative and can be adapted depending on context. For instance, processing speed r may be measured in records/ms, memory m in megabytes (MB), and cost u in USD/hour. In this work, we use normalized or synthetic units to model relative performance and cost differences between nodes, without binding the system to a specific infrastructure or currency.

3.2. Derived Parameters

We derive additional parameters from the node characteristics to facilitate the MIP formulation:

3.2.1. Processing Time

$$x_i = \frac{d_i}{r_i} \tag{1}$$

where x_i is the time required by node i to process its assigned data volume d_i.

3.2.2. Transfer Time

$$y_i = \ell_i + \frac{d_i}{b_i} \tag{2}$$

where y_i is the communication overhead for node i, computed from latency ℓ_i and bandwidth b_i.

3.2.3. Total Cost

$$w = \sum_{i=1}^{N} u_i \cdot x_i \tag{3}$$

where w denotes the cumulative cost of utilizing all selected nodes.

3.3. Optimization Objectives

The primary objective of our MIP model is to minimize the total execution time (makespan) of the parallel sorting process. This ensures that the overall completion time is minimized, taking into account both computation and communication overheads. Formally, the primary objective is defined as:

$$\min z = \max_{i=1}^{N}(x_i + y_i) \tag{4}$$

where z represents the maximum time taken by any node i to complete its assigned data processing and communication tasks. This formulation captures the makespan of the entire parallel sorting operation, ensuring that the slowest node determines the overall completion time. In addition to minimizing makespan, we aim to choose the most cost-effective solution among equivalent combinations. To achieve this, we introduce a secondary objective that minimizes the total financial expenditure. This is incorporated into the model using a lexicographic optimization approach:

$$\min \ z + \varepsilon \cdot w \tag{5}$$

In this formulation, w represents the cumulative cost of utilizing the selected nodes, and ε is set to a negligible value (e.g., 10^{-6}) to prioritize makespan minimization while breaking ties in favor of cost efficiency. This approach ensures that among solutions with equivalent makespan, the one with the lowest cost is selected.

3.4. Constraint Definitions

The proposed LP model is subject to several constraints that reflect system limitations and ensure a feasible allocation of data across nodes. These constraints incorporate resource boundaries (e.g., memory and budget), performance considerations (e.g., makespan), and completeness of the partitioning scheme.

3.4.1. Makespan Constraint

$$z \geq \frac{d_i}{r_i} + \ell_i + \frac{d_i}{b_i} \quad \forall i \in \{1, \ldots, N\} \tag{6}$$

This constraint ensures that the total execution time, represented by the variable z, is at least as large as the time taken by any individual node i to both process its assigned data and transfer it to the next stage. It combines computation time $\frac{d_i}{r_i}$ with communication latency ℓ_i and data transfer time $\frac{d_i}{b_i}$.

3.4.2. Memory Constraint

$$d_i \leq m_i \quad \forall i \in \{1, \ldots, N\} \tag{7}$$

Each node has a limited memory capacity m_i, and this constraint ensures that the volume of data d_i assigned to node i does not exceed its available memory.

3.4.3. Coverage Constraint

$$\sum_{i=1}^{N} d_i = D \tag{8}$$

This constraint enforces full data allocation: the entire dataset of size D must be partitioned and distributed among the N available nodes without omission or duplication.

3.4.4. Non-negativity Constraint

$$d_i \geq 0 \quad \forall i \in \{1, \ldots, N\} \tag{9}$$

This standard constraint ensures that each node receives a non-negative volume of data, reflecting the physical impossibility of assigning negative quantities.

3.4.5. Budget Constraint (Optional)

$$0 \leq \sum_{i=1}^{N} u_i \cdot d_i \leq B \tag{10}$$

In addition to node-specific parameters, we optionally define a user-defined input B, representing the maximum allowable expenditure. This parameter must be a non-negative number, with a default value of 0. Including this constraint models scenarios where financial limitations must be respected.

4. Heterogeneous PSRS (Baseline Algorithm)

This is the model of Monga and Lodhi [53] which we will use as a baseline. The algorithm is designed to sort a list of items using multiple workers, each with different speeds.

4.1. Pseudocode

```
Algorithm 1: Heterogeneous PSRS (Baseline Algorithm)
Inputs:
    • $data$: Input data to be sorted
    • $n$: Total number of data items
    • $p$: Number of processors
    • $perf[0 \dots p-1]$: Relative performance of each processor
Output:
    • Sorted data
Phase 1: Local Sorting and Sampling
for each processor $i$ do
    $size_i = \frac{n \cdot perf[i]}{\sum perf}$
    Assign $size_i$ items to processor $i$
    Processor $i$ performs sequential quicksort on its local data
    Select $L = (p-1) \cdot perf[i]$ samples from local sorted data
    Send samples to the designated coordinator
Phase 2: Pivot Selection
Designated coordinator gathers and sorts all samples
Selects $p-1$ global pivots and broadcasts them to all processors
Phase 3: Data Redistribution
for each processor do
    Partition local data using global pivots
    Send partitioned data to corresponding processors
    Retain the portion corresponding to its final range
Phase 4: Final Sorting and Merge
Each processor sorts its final local data
Coordinator concatenates all sorted segments for final output
```

4.2. Limitations

The Heterogeneous PSRS algorithm by Monga and Lodhi[53] effectively distributes sorting tasks across processors with varying speeds but has notable inefficiencies. Its local sorting phase precedes global partitioning, often leading to imbalanced data redistribution, where some nodes receive disproportionately larger portions. This imbalance increases memory usage and processing delays. Additionally, the algorithm overlooks communication costs and network latency, critical factors in distributed and cloud-based systems. These limitations highlight the need for a more adaptive approach that integrates performance and cost considerations when assigning data ranges.

5. LP-Guided Sorting (Proposed Algorithm)

We propose a modified algorithm that replaces the sampling and redistribution stages with a LP formulation that statically partitions the global key range across heterogeneous distributed nodes.

5.1. Bucket Assignment

Given the total dataset size D and a heterogeneous set of N nodes with parameters $(r_i, m_i, \ell_i, b_i, u_i)$, we solve the MIP formulation (Section **??**) to compute optimal partition sizes d_i such that:

- Processing time is balanced according to each node's capability
- Communication and processing costs are minimized
- Constraints on memory, load, and budget are respected

Once d_i values are determined, each node is assigned a fixed key range. For example (illustrative only):

Worker	**Percentage of Total Data**
A (4x speed)	50%
B (3x speed)	30%
C (2x speed)	20%

5.1.1. Distribution Strategies

Aspect	**Option A: Centralized Bucketing**	**Option B: Asynchronous Routing**
Memory Usage	Requires 2× dataset size in memory	Lower memory footprint, suited for streaming or distributed sources
Latency	Low latency and simple coordination	May incur higher latency due to decentralized routing
Scalability	Limited scalability due to centralized coordination	Better scalability with distributed routing
Coordination Complexity	Simple, centralized coordination	Decentralized routing increases complexity
Billing Costs	Lower due to fewer node interactions	Higher due to more frequent node usage

Table 1. Comparison of Centralized Bucketing and Asynchronous Routing

5.1.2. Merge Strategies

Aspect	**Option A: Block Merge (Centralized Bucketing)**	**Option B: Coordinated Min-Max Pop (Asynchronous Routing)**
Data Handling	Each node returns a fully sorted block corresponding to a unique key range	Coordinator continuously requests the current min and max from each node's local sorted data
Merge Strategy	Coordinator performs a two-ended merge by popping global min and max from the node outputs	Dynamically merges results by selecting the global min and max across all nodes
Efficiency	Efficient for well-partitioned data; avoids re-querying nodes	Suitable when node outputs are fragmented or partially overlapping
Memory Usage	Requires all blocks to be received and held in memory before merging	Lower memory usage as data is processed dynamically
Coordination Overhead	Minimal; relies on pre-sorted blocks	Higher coordination overhead; increases latency and total merge time

Table 2. Comparison of Merge Strategies

5.2. Proposed Algorithm

```
Algorithm 2: LP-Guided Bucket-Quicksort Hybrid (Proposed Algorithm)
Inputs:
    • $D$: Total dataset
    • $N$: Number of nodes
    • $r_i$: Processing rate of node $i$
    • $m_i$: Memory capacity of node $i$
    • $b_i$: Network bandwidth of node $i$
    • $\ell_i$: Network latency of node $i$
    • $u_i$: Usage cost rate of node $i$
Output:
    • Sorted data
Phase 1: Static Partitioning via LP
Solve LP to determine proportion $p_i$ of data that node $i$ should process, where $\sum p_i = 1$
Compute data volume per node: $d_i = p_i \cdot |D|$
Phase 2: Data Distribution
Option A: Centralized Bucketing (Low Latency, High Memory)
Initialize $N$ local buckets $B_1, B_2, \ldots, B_N$
for each item $x$ in $D$ do
    Assign $x$ to a bucket $B_i$ such that the current size of $B_i$ is less than $d_i$
foreach bucket $B_i$ do
    Send $B_i$ to node $i$ for sorting
Option B: Asynchronous Routing (Low Memory, High Latency)
Initialize counters $c_i \leftarrow 0$ for all nodes
for each item $x$ in $D$ do
    Route $x$ to node $i$ such that $c_i < d_i$ and increment $c_i$
Phase 3: Local Sorting
for each node $i$ do
    Sort received data using sequential quicksort
Phase 4: Final Merge
Option A: Block Merge (Centralized Bucketing)
Let $S_1, S_2, \ldots, S_N$ be the sorted blocks received from each node
Initialize a double-ended queue: merged_sorted $\leftarrow$ empty deque
while any $S_i$ is non-empty do
    Identify the global minimum and maximum from heads and tails of non-empty $S_i$
    Pop the global min and append to the left, max to the right of merged_sorted
return merged_sorted
Option B: Coordinated Min-Max Pop (Asynchronous Routing)
Initialize a double-ended queue: merged_sorted $\leftarrow$ empty deque
Let each node expose head and tail of its local sorted buffer on request
while total number of popped elements $< |D|$ do
    foreach node $i$ do
        Send $h_i$ and $t_i$ (head and tail) to coordinator
    Coordinator selects global min and max from all $h_i$, $t_i$
    Source nodes pop their respective elements
    Append global min to left, global max to right of merged_sorted
return merged_sorted
```

6. Theoretical Complexity Analysis

A rigorous complexity analysis highlights the theoretical advantages of our LP-guided approach over the baseline PSRS algorithm, particularly in heterogeneous distributed nodes.

6.1. Baseline Algorithm

6.1.1. Time Complexity

The original Heterogeneous PSRS algorithm involves four distinct phases, each contributing to the overall time complexity:

6.1.1.1. Phase 1: Local Sorting and Sampling. Each of the p processors sorts its local data of size $O(n/p)$ on average, taking $O\left(\frac{n}{p}\log\frac{n}{p}\right)$ time using quicksort. Sampling takes $O(p)$ time per processor. The communication of samples to the coordinator takes $O(p^2)$ in the worst case if all processors send their $O(p)$ samples sequentially. Thus, this phase has an average time complexity of $O\left(\frac{n}{p}\log\frac{n}{p}+p^2\right)$.

6.1.1.2. Phase 2: Pivot Selection. The coordinator sorts $O(p^2)$ samples, which takes $O(p^2\log p^2)=O(p^2\log p)$ time. Broadcasting the $p-1$ pivots to all p processors takes $O(p)$ time in parallel or $O(p^2)$ sequentially.

6.1.1.3. Phase 3: Data Redistribution. Each processor partitions its $O(n/p)$ local data using $p-1$ pivots, which takes $O(n/p\cdot p)=O(n)$ in the worst case if the pivots are poorly chosen. The redistribution of data among processors can take up to $O(n)$ in the worst-case scenario where one processor receives almost all the data.

6.1.1.4. Phase 4: Final Sorting and Merge. Each processor sorts its received data, which can be up to $O(n)$ in the worst case, leading to $O(n\log n)$ for a single processor if load balancing is poor. The coordinator then merges the p sorted segments, which takes $O(n)$ time.

6.1.1.5. Overall Complexity. Combining these phases, the average-case time complexity of the original PSRS algorithm is dominated by the local sorting and pivot selection, yielding $O\left(\frac{n}{p}\log\frac{n}{p}+p^2\log p+n\right)$. Under ideal load balancing, this simplifies to $O\left(\frac{n\log n}{p}+p^2\log p\right)$. The worst-case time complexity, particularly due to potential imbalances in data redistribution, can reach $O(n\log n)$ if one processor ends up with most of the data, or even $O(n^2/p)$ if local sorting on an imbalanced partition degrades to quadratic time.

6.1.2. Space Complexity

For space complexity, the PSRS algorithm requires local storage for each processor to store its assigned data, which is $O(n/p)$ on average. Sampling generates $O(p)$ samples per processor, and the coordinator stores $O(p^2)$ samples. During data redistribution, processors might need temporary buffers to hold data being sent or received, potentially up to $O(n/p)$ in size per processor. Each processor holds a sorted segment of approximately $O(n/p)$. The total space complexity of the PSRS algorithm is $O(n)$ as the total data is partitioned and stored across the processors. The coordinator requires additional space for the samples, $O(p^2)$.

6.1.3. Complexity Analysis

The PSRS algorithm executes in four distinct phases: local sorting and sampling, pivot selection, data redistribution, and final sorting or merging. The overall time complexity is characterized as follows:

$$\mathcal{O}\left(\underbrace{\frac{n}{p}\log\frac{n}{p}}_{\text{local sorting and sampling}} + \underbrace{p^2\log p}_{\text{pivot selection}} + \underbrace{n}_{\text{redistribution}} \right) \tag{11}$$

Under ideal load balancing and efficient communication, this simplifies to:

$$\mathcal{O}\left(\underbrace{\frac{n\log n}{p}}_{\text{local sorting and sampling}} + \underbrace{p^2\log p}_{\text{pivot selection}} \right) \tag{12}$$

However, due to dynamic pivot selection and possible load imbalance, the worst-case time complexity may degrade to:

$$\mathcal{O}\left(\underbrace{n\log n}_{\text{worst-case due to load imbalance}} \right) \tag{13}$$

This degradation arises when a significant portion of data is allocated to a single processor due to poor pivot selection.

6.2. Proposed Algorithm

For the LP-guided algorithm, the process is significantly streamlined.

6.2.1. Time Complexity

6.2.1.1. Phase 1: Static Partitioning via LP. This is solved once as a precomputation step. The complexity of solving a Linear Programming problem is polynomial, typically handled by efficient algorithms like the interior-point method or simplex method. For practical LP solvers, the complexity is often approximated as $O(N^3)$, where N is the number of nodes, based on the number of variables (proportional to N) and constraints (also proportional to N).

Although the constraint matrix of a linear program has $O(N^2)$ entries, solving the LP involves more than just accessing these entries. Matrix operations such as factorization or inversion generally take $O(N^3)$ time. While sparsity can reduce this in practice, we conservatively estimate the complexity as $O(N^3)$. This is a one-time overhead, and for a fixed number of nodes N, it becomes negligible over repeated runs.

6.2.1.2. Phase 2: Data Distribution. For Option A (Centralized Bucketing), data items are first collected and assigned to N local buckets at the coordinator before being sent to the nodes. This incurs a time complexity of $O(n+N)$. Option B (Asynchronous Routing), on the other hand, routes each data item directly to its target node using the LP-generated partition map, avoiding central bucketing. This results in a slightly simpler time complexity of $O(n)$. In both cases, the time is linear in the dataset size.

6.2.1.3. Phase 3: Local Sorting. Each of the N nodes sorts its assigned partition of size d_i. Since the LP aims to balance the workload, we expect $d_i \approx O(n/N)$. The sorting time per node is $O\left(\frac{n}{N}\log\frac{n}{N}\right)$. With parallel execution, the overall sorting time remains $O\left(\frac{n}{N}\log\frac{n}{N}\right)$.

6.2.1.4. Phase 4: Final Merge. Both options use a coordinated merge phase where the coordinator retrieves the global minimum and maximum elements in each round. This requires scanning N candidates per iteration, with $n/2$ iterations in total (since two items are merged per round), yielding a worst-case time complexity of $O(nN)$ for the merge. This step is optional if a distributed sorted output is acceptable.

6.2.1.5. Overall Complexity. The total time complexity for both options is thus:

- **Option A (Centralized Bucketing)**: $O(N^3+n+N+\frac{n}{N}\log\frac{n}{N}+nN)$
- **Option B (Asynchronous Routing)**: $O(N^3+n+\frac{n}{N}\log\frac{n}{N}+nN)$

When $N \ll n$, both simplify to $O(nN+\frac{n}{N}\log\frac{n}{N})$ (ignoring constants and lower-order terms). The $O(N^3)$ LP precomputation cost becomes negligible when amortized over repeated sorting tasks with fixed N.

6.2.2. Space Complexity

Each node stores only its local partition of size d_i, where $\sum d_i = n$, giving a total data storage of $O(n)$. The differences arise in the temporary memory usage during distribution and merging.

6.2.2.1. Option A (Centralized Bucketing). The coordinator temporarily holds all n items in memory for bucketing and routing, resulting in a peak memory usage of $O(2n)$ at the coordinator. In addition, the LP solver uses $O(N^2)$ space for the constraint matrix and working buffers. Final merging may also require temporarily storing all N sorted segments at the coordinator, although this can be streamed in practice.

6.2.2.2. Option B (Asynchronous Routing). This avoids central bucketing. Each item is routed immediately to its target node, minimizing the coordinator's memory usage. The total memory remains $O(n)$ but is fully distributed. The LP solver still uses $O(N^2)$ memory once. The coordinator only needs to retain $O(N)$ pointers (e.g., current head/tail) for the merge phase, making it more space-efficient than Option A.

6.2.3. Complexity Analysis

The LP-guided algorithm cleanly separates the static partitioning step (a one-time cost) from the dynamic sorting pipeline. After solving the LP, sorting is guided by a cost-aware, precomputed partitioning strategy. The two variants differ primarily in memory footprint and routing complexity:

6.2.3.1. Option A: Centralized Bucketing. This option routes through the coordinator and temporarily buffers all n items before dispatching. The final merge involves scanning N node outputs per round, leading to a total time complexity of:

$$O\left(\underbrace{N^3}_{\text{LP solve}} + \underbrace{n}_{\text{distribution}} + \underbrace{N}_{\text{routing overhead}} + \underbrace{\frac{n}{N}\log\frac{n}{N}}_{\text{local sort}} + \underbrace{nN}_{\text{merge}}\right) \tag{14}$$

Peak space usage is $O(2n)$ at the coordinator due to centralized routing.

6.2.3.2. Option B: Asynchronous Routing. This option avoids centralized buffers by routing items directly to the nodes. It requires minimal space at the coordinator and achieves the same computational complexity (except the $O(N)$ routing overhead is avoided):

$$O\left(\underbrace{N^3}_{\text{LP solve}} + \underbrace{n}_{\text{distribution}} + \underbrace{\frac{n}{N}\log\frac{n}{N}}_{\text{local sort}} + \underbrace{nN}_{\text{merge}}\right) \tag{15}$$

Peak space usage is $O(n)$, distributed evenly across the nodes, with only $O(N)$ overhead at the coordinator for merge coordination.

6.3. Complexity Comparison

In this section, we formally analyze and compare the time complexity of the baseline PSRS algorithm and the proposed LP-guided algorithm, incorporating both Option A and Option B for data distribution. The analysis emphasizes both average-case behavior and worst-case scenarios under heterogeneous processing environments.

Algorithm	**Average Case**	**Worst Case**
PSRS	$O\left(\frac{n\log n}{p} + p^2\log p\right)$	$O(n\log n)$
LP-guided (Option A)	$O(n + N + \frac{n}{N}\log\frac{n}{N} + nN)$	$O(N^3 + n + N + \frac{n}{N}\log\frac{n}{N} + nN)$
LP-guided (Option B)	$O(n + \frac{n}{N}\log\frac{n}{N} + nN)$	$O(N^3 + n + \frac{n}{N}\log\frac{n}{N} + nN)$

Table 3. Time Complexity Comparison (Average vs Worst Case)

The LP-guided algorithm offers theoretical advantages in both average and worst-case runtime. By removing the need for sampling and pivot-based partitioning at runtime, it eliminates a primary source of imbalance and communication overhead. In contrast, the PSRS algorithm is more susceptible to skewed load distributions and requires additional

Algorithm	**Average Case**	**Worst Case**
PSRS (Baseline)	$O(n)$	$O(n)$
LP-guided (Option A)	$O(n)$	$O(2n + N^2)$
LP-guided (Option B)	$O(n)$	$O(n + N^2)$

Table 4. Space Complexity Comparison (Average vs Worst Case)

Aspect	**PSRS**	**LP-Guided**
Load Balancing	Data-dependent and dynamic, prone to imbalance	Statically balanced using node profiles
Synchronization	Multiple barriers: sampling, pivoting, merging	Minimal, only at merge (if needed)
Communication Overhead	$O(p^2)$ (samples), $O(n)$ (data)	$O(n)$ (initial only)
Preprocessing	None	One-time $O(N^3)$ for LP
Worst-case	$O(n \log n)$	Consistently balanced, better in heterogeneity

Table 5. Performance Factors Comparison

synchronization steps, which may limit scalability on heterogeneous hardware. Although solving the LP model incurs an initial computational cost of $O(N^3)$, this cost becomes negligible in scenarios involving repeated sorting tasks or large datasets. As such, the LP-guided approach is especially advantageous in environments where preprocessing is feasible and performance consistency is critical. The LP-guided algorithm demonstrates superior asymptotic performance under realistic workload and platform assumptions, particularly when amortizing the LP overhead across repeated executions or in systems with significant node heterogeneity.