Performance Analysis of Deadlock Detection using MapReduce (PySpark)

Sowreddy Pranavi

November 7, 2025

Abstract

The main objective of this work is to design a scalable algorithm for detecting deadlocks in a distributed system by analyzing resource allocation logs. The algorithm is implemented using the MapReduce programming model in PySpark to build a Resource Allocation Graph (RAG) and detect cycles iteratively.

This report presents the performance analysis conducted by varying the number of CPU cores (up to 8) across multiple dataset sizes. It also discusses the relationship between the number of cores and the corresponding execution time (speedup).

1 Introduction

Apache Spark enables distributed and parallel computation through its Resilient Distributed Datasets (RDDs). However, performance gains depend on the input size and the overhead associated with parallelization. This experiment studies how execution time changes with the number of available cores for datasets of various sizes using Spark's `local[n]` mode, where `n` denotes the number of CPU cores.

2 Experimental Setup

2.1 Environment

- **Platform:** ADA HPC Cluster (executed in local mode)
- **Language:** Python (PySpark)
- **Input Datasets:** 200 lines, 1k, 5k, 10k, 20k, and 40k samples
- **Execution Command:** `spark-submit Assignment4.py sample_X.log n`
- **Cores Tested:** 1, 2, 4, 6, and 8

2.2 Objective

To analyze how execution time varies with the number of cores across different dataset sizes, and to determine the point where parallelism provides optimal benefits.

3 Results and Analysis

Table 1 presents the execution times observed for different datasets and core counts.

Dataset	1 Core	2 Cores	4 Cores	6 Cores	8 Cores	Observation
200 lines	4.20	4.69	4.76	4.54	4.38	No improvement
1k	15.23	16.38	17.35	17.65	18.23	Slightly slower
5k	19.53	20.29	19.52	20.96	19.95	Similar
10k	25.03	26.26	25.90	25.25	25.36	Similar
20k	29.38	27.68	27.33	27.03	26.67	∼9% faster
40k	41.83	34.39	34.16	34.99	34.40	∼18% faster

Table 1: Execution times (in seconds) for different datasets and core counts.

3.1 Trend Observations

- For smaller datasets, execution time remains nearly constant or increases slightly as more cores are used. This happens because Spark's job setup and scheduling overhead outweighs parallel benefits.
- Medium-sized datasets (10k–20k) show modest improvements (about 9–10%) when using 4–8 cores.
- The largest dataset (40k) achieves significant improvement, with about 18% faster execution due to effective parallelization.
- Beyond 4–6 cores, execution time levels off, showing diminishing returns from additional cores.

4 Graphical Analysis

Figure 1 illustrates how execution time varies with the number of cores for different dataset sizes. Execution time decreases initially but stabilizes as core count increases, showing limited scalability within a single node.

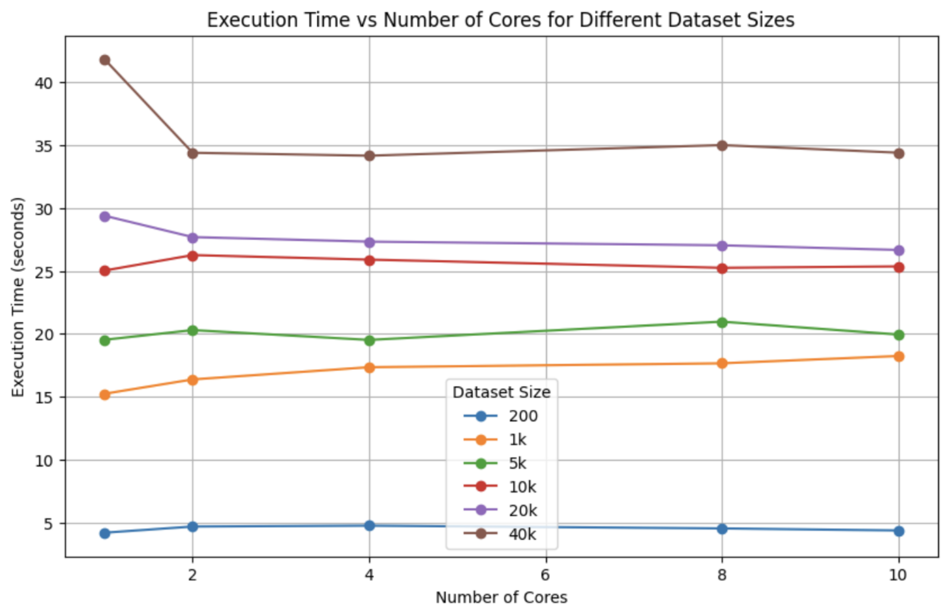

Figure 1: Execution time vs. number of cores for various dataset sizes.

5 Combiner Step Usage

The **combiner** is applied in the following code segment:

```
adj_list = edges_pairs.combineByKey(
    lambda v: [v],
    lambda acc, v: acc + [v],
    lambda acc1, acc2: acc1 + acc2
).collectAsMap()
```

This code constructs the adjacency list (`adj_list`) for the graph using the `combineByKey()` transformation. This operation works like a **map-side reduce**—similar to a **combiner** in the traditional MapReduce framework.

- `combineByKey()` performs partial aggregation on the mapper side before the shuffle phase.
- It reduces the amount of data transferred across the network.
- This helps improve performance by minimizing shuffle overhead.

Table 2: Execution time comparison (4 cores) with and without combiner.

Dataset Size	Without Combiner (s)	With Combiner (s)
200	4.76	5.43
1K	17.13	19.30
5K	22.63	23.20
10K	22.54	25.45
20K	27.13	27.76
40K	38.73	36.61

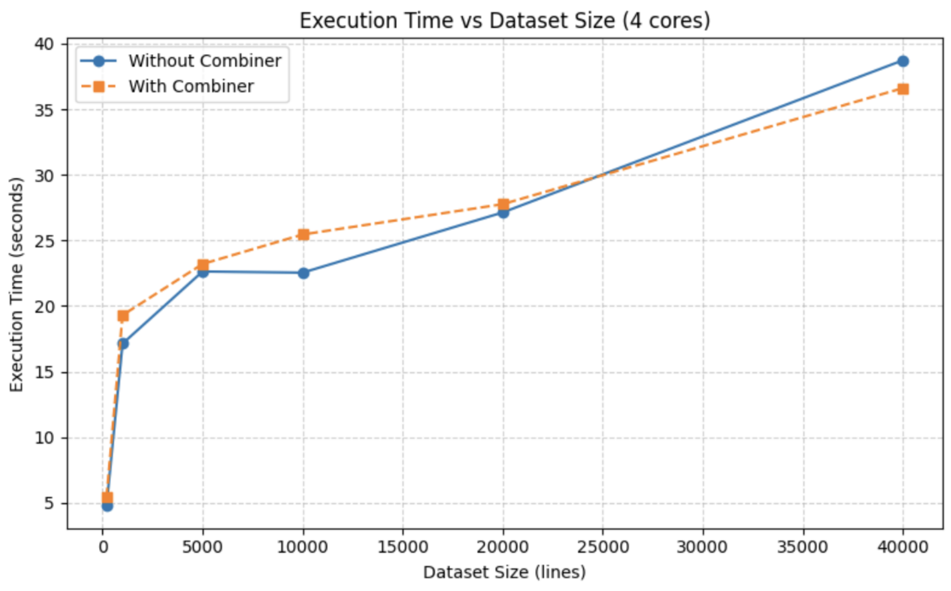

Figure 2: Execution time comparison with and without combiner.

Interpretation: The results show that for smaller datasets (200–20K), execution time with the combiner is similar or slightly higher. This occurs because the overhead of creating and merging combiners outweighs the benefits when the dataset is small. However, for larger datasets (e.g., 40K), the combiner improves performance by reducing shuffle costs through local (map-side) aggregation before the reduce stage. Overall, combiners are beneficial for large-scale data processing, but their impact is negligible—or even slightly negative—for small inputs.

6 Discussion

6.1 Impact of Input Size

For small datasets, Spark's inherent overheads (such as task scheduling and serialization) dominate total runtime. As a result, using multiple cores does not provide performance gains and can even increase latency.

6.2 Diminishing Returns

As the dataset size increases, adding more cores initially reduces execution time. However, after 4–6 cores, improvements flatten due to:

- A limited number of parallel tasks
- Data locality constraints
- Overheads in Spark's stage synchronization

6.3 Amdahl's Law

According to Amdahl's Law, the potential speedup from parallelization is limited by the portion of the program that cannot be parallelized. In this experiment, Spark's job setup, shuffle, and I/O operations form the sequential portion, limiting overall scalability beyond a certain number of cores.

6.4 Effect of Combiner

The combiner reduces shuffle data and improves performance for large datasets, but its benefits are noticeable only when data transfer becomes a significant part of total execution time.

7 Conclusions

- Spark's parallelism benefits become significant only for large datasets ($\geq$ 20K lines).
- Increasing cores beyond 4–6 yields minimal improvement due to system overheads.
- For small datasets, job setup costs dominate total execution time.
- Parallel execution provides up to $\sim$20% performance gain for large inputs, while smaller datasets show negligible or negative gains.