MNEME

A PARALLEL DATA PREPROCESSING FRAMEWORK FOR LARGE TABULAR DATASETS

Argiris Sofotasios, Dimitris Metaxakis, Panagiotis Hadjidoukas

HPCLab, CEID, UPatras

HLPP 2025, Innsbruck, Austria, July 3, 2025

HLPP 2025 - 18th International Symposium on High-level Parallel Programming and Applications

Outline

1 Why Data Preprocessing (should) come(s) first

- Data sample quality is crucial for the prediction accuracy of ML/DL models
- Low-quality data $\Rightarrow$ poor performance, slow convergence, lack of generalization, unreliable/biased predictions, underfitting

Therefore, data is the **driving force** of ML

However, the diversity of modern data sources leads to low-quality raw datasets → noise, conflicting information, inconsistencies in scales/units, missing values

Data preprocessing: crucial/essential step before training

2 Background & Motivation

- Data preprocessing involves a series of techniques: data integration, cleansing, reduction, **transformation**
- Data transformation includes: feature rescaling, imputation of missing values, encoding of categorical features, etc.
- Most techniques include two steps: **fitting** (feature-wide statistics) & **transformation** (often done on-the-fly during batch fetching)

Objective: To adjust the raw data in such a way that the model learns better (good performance) and faster (fast convergence)

2 Background & Motivation (1 / 2)

The total preprocessing time is mainly affected by **two** factors: **dataset size** and **available RAM capacity**

- In most DL workflows, datasets don't fit in RAM (e.g. Criteo → Terabyte click logs)
- Dataset size > RAM $\Rightarrow$ preprocessing becomes **I/O bound**
- The repeated I/O operations during the fitting phase cause fetch stalls and increased data access latency

2 Background & Motivation (2 / 2)

So, preprocessing becomes a **dominant bottleneck**, slowing down the entire ML workflow

The common sequential chunk-based approach used by popular Python frameworks (e.g. pandas with Scikit-learn) to handle the out-of-core processing remains slow and inefficient and often requires manual optimization

Therefore, it is essential to adopt **parallel I/O schemes** that enable efficient, concurrent fetching and processing of smaller dataset chunks, **accelerating the fitting of preprocessing operators**

3 The MNEME framework

Objective: Accelerate the fitting of various preprocessing operators on large tabular datasets exceeding RAM in **single-node** environments

Tabular data format: Text files, typically in CSV format

How ?

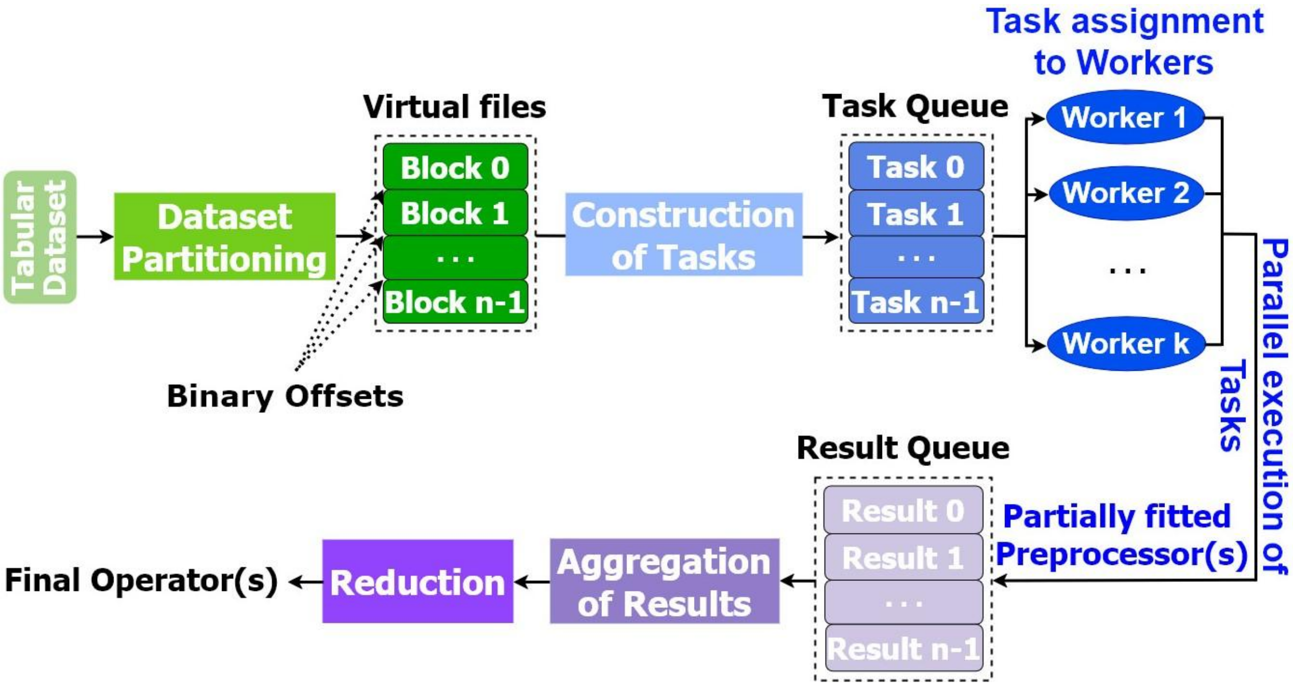
1. Parallelize both data loading and statistics computation part using an **overlapping MapReduce-based task-farm** block scheme
2. Utilize both **multiprocessing** and **multithreading**
3. Introduce a **pipeline scheme** that supports parallel fitting of multiple preprocessing operators with a **single** disk-to-memory data load

Note: The transformation step is performed **on-the-fly** during batch loading in the training process (using the prior fitted operators)

3.1 Key features

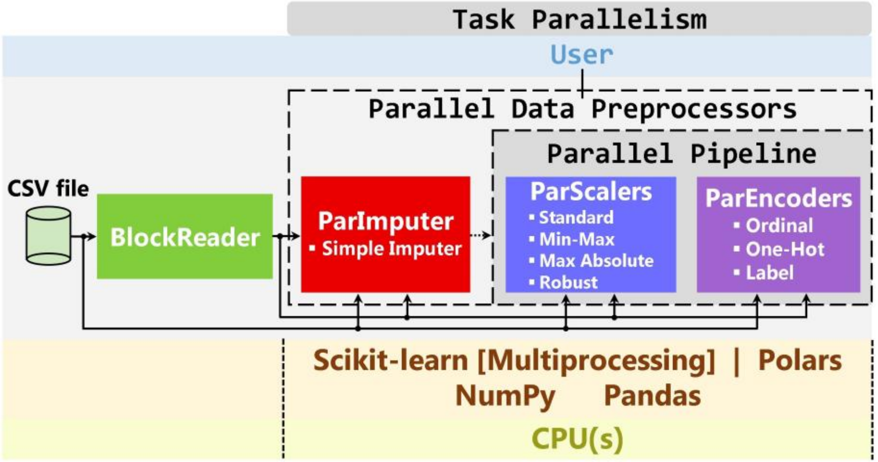
- Specialized for large tabular datasets in raw text CSV files
- Memory-efficient parallel data loading with **minimal memory footprint** without requiring any modification or splitting of the raw file and without generating any intermediate copy of the data in binary format
- Currently supports **4** scalers, **3** encoders and **3** basic imputation techniques
- Includes **two** pipeline structures: one for imputation techniques and one for scalers/encoders, enabling the **parallel fitting of multiple operators** with just a single disk-to-memory data load
- Implemented as a **unified Python framework** with a user-friendly API accessible to users without HPC expertise (‘democratization’ of HPC)

3.2 Methodology



3.3 Software (1 / 2)

- Build on top of **Scikit-learn**
- Task-based parallelism model, employing a **process pool** programming pattern from Python's multiprocessing package
- Rust threads are integrated via the CSV reading capabilities of the **Polars** library
- Adopts the **fit()-transform()** API model popularized by Scikit-learn



3.3 Software (2 / 2)

An illustrative example of Mneme's API

```
from Mneme import BlockReader
from Mneme.preprocessing import ParStandardScaler

datafile = "/path/to/data.csv"
# dataset shape : 10M rows, 701 features (x0, x1, ..., x699, y0)
num_idxs = [f"x{i}" for i in range (700)]
workers = 4; IO_threads = 2; n_blocks = 100

br = BlockReader(datafile, num_blocks = n_blocks)
std_scaler = ParStandardScaler(data_file = datafile, num_idxs = num_idxs)
std_scaler._fit(use_parallel = True, block_reader = br, num_workers = workers, IO_workers = IO_threads)
```

4 Related work (1 / 2)

dask

Flexible Python parallel computing library, primarily designed for out-of-core distributed processing, using a dynamic DAG scheduler for task allocation. Among other features, it also supports scalable preprocessing through Dask-ML preprocessing package

Limitations (compared to Mneme)

- The generated DAG is not always optimized, often causing more I/O operations than necessary — e.g. when fitting multiple preprocessing operators, redundant data passes are introduced
- Dask relies on pandas structures for chunk loading and handling, making it slower than Mneme's Polars-based design

4 Related work (2 / 2)

Polars

Rust-based high-performance multithreaded DataFrame library for large-scale data analysis and processing. Compared to pandas, Polars can achieve more than 30x performance gains (source: Polars official site)

Limitations (compared to Mneme)

Polars does not natively support a dedicated unified preprocessing interface like Dask. As a result, during the fitting of preprocessing operators, chunks loading and statistics computation are performed as separate steps, leading to a suboptimal execution scheme

5 Experiments & Evaluation

Experimental Setup

System I: AMD Ryzen 9 7950X CPU (16C/32T), 32 GB DDR5 RAM, Kingston KC3000 PCIe Gen4 NVMe SSD

System II: NVIDIA DGX A100 with dual AMD EPYC 7742 CPUs (128C/256T), 1 TB DDR4 RAM (restricted to 32 GB during experiments), 15 TB NVMe SSD storage

Python: 3.10.12

Dataset: 126 GB, 701 features (700 numerical, 1 categorical with 4 classes)

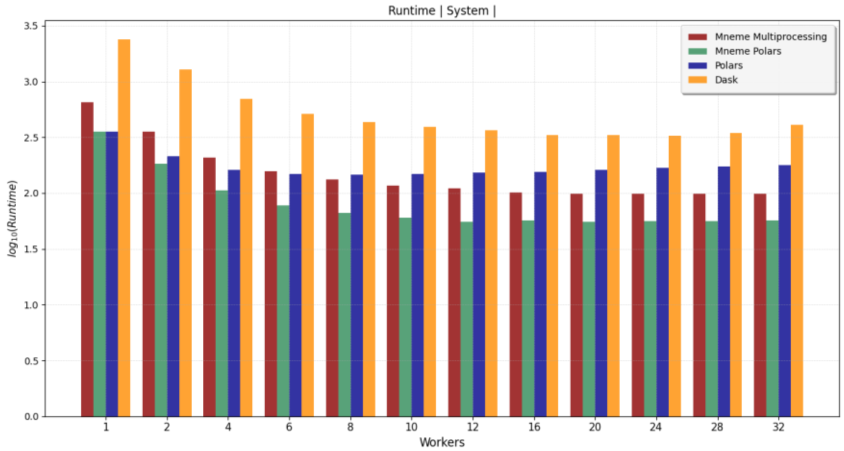
Metrics: The reported execution times correspond to the average of **10** runs for each worker setting (plus a preliminary warmup phase)

Experiment: A preprocessing pipeline applied Z-score normalization to the first 350 features, min-max normalization to the next 350 and label encoding to the target variable

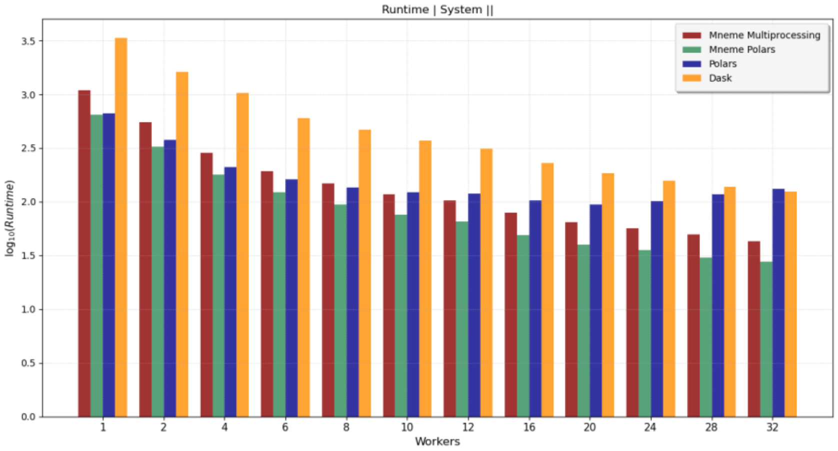
5 Experiments & Evaluation (1 / 4)

Runtime performance - $\log_{10}$ scale

System I



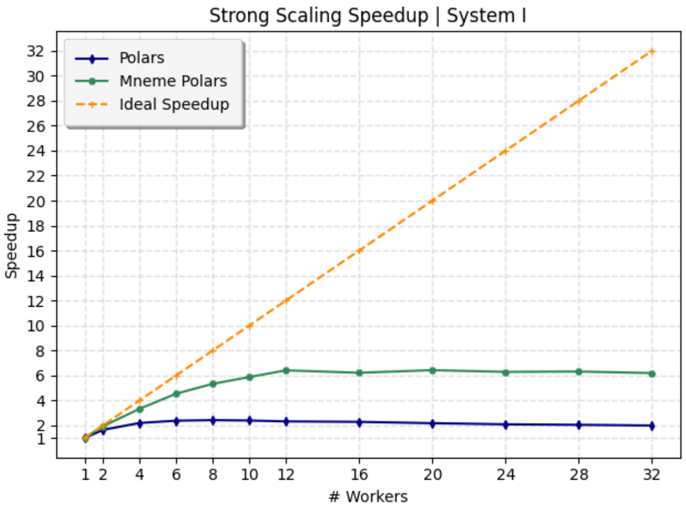
System II



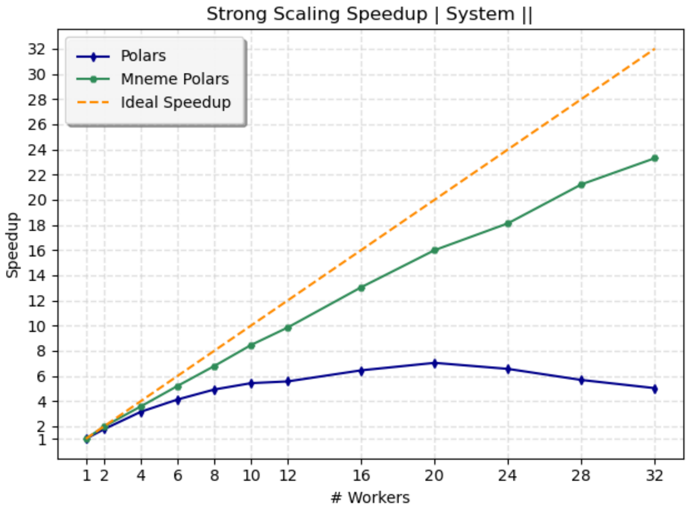
5 Experiments & Evaluation (2 / 4)

Strong scaling speedup

System I



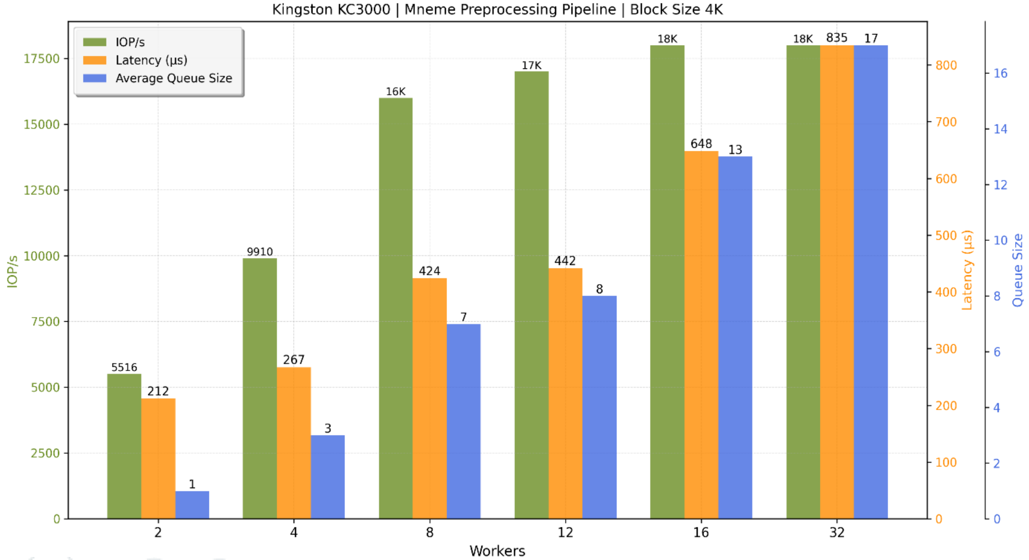
System II



5 Experiments & Evaluation (3 / 4)

System I

I/O performance



Runtime performance (8W/1GPU)

Criteo Dataset	Time (s)
Mneme	143
NVTabular	267
Polars	303

5 Experiments & Evaluation (4 / 4)

- Compared to Dask: up to **5x** better fitting time on System I, **3x** better fitting time on System II
- Compared to Polars: **14–64% speedup** with < **12** workers in both systems
- Polars slows down after **20** workers (System II) and needs ~60% more memory to improve performance, while Mneme scales well

6 Limitations & Future work

- No support for distributed execution (on going work)
- User-defined block size
- Support for raw-text CSV files only
- No GPU acceleration

6 Limitations & **Future work**

- Automatic **block size tuning** based on dataset size and system resources using a heuristics-based or ML-based approach
- Add **Python native multithreading** to the task farm scheme (GIL likely not a bottleneck due to I/O bound nature of tasks; also experimenting with the free-threading mode of 3.13)
- Support **diverse dataset types** and distributed processing via **mpi4py** (Python bindings for the MPI)
- Support **extra** preprocessing operators
- Integrate **GPU acceleration** where beneficial

Thank you!

a.sofotasios@ac.upatras.gr, d.metaxakis@ac.upatras.gr