Parallelization of Machine Learning Techniques

Navneet Shankar
ECE
Virginia Tech
VA, USA
navneets099@vt.edu

Anagha Manjunatha
ECE
Virginia Tech
VA, USA
anaghahm@vt.edu

***Abstract*—This paper details our work on parallelizing various machine learning algorithms, specifically Principal Component Analysis (PCA), Random Forest, Neural Networks, and K-Nearest Neighbors (KNN). We implemented these algorithms using Python and evaluated their performance by experimenting with different thread counts and dataset sizes. The primary objective was to enhance computational efficiency through parallelization techniques while maintaining the accuracy of the models. Our results indicate substantial speedups in most cases, though some algorithm-specific characteristics introduced overheads that limited performance gains. The work aims to contribute to the efficient execution of machine learning models on multi-core systems, facilitating faster analysis on larger datasets.**

I. Introduction

Machine learning models are becoming increasingly complex, and datasets continue to grow in size. This trend necessitates faster training and inference methods. Parallelization offers a promising approach to improve the performance of machine learning techniques. In this project, we focused on parallelizing five key algorithms: PCA, Random Forest, Neural Networks, Decision Tree, and KNN.

II. Overview of State-of-the-Art

Parallelization in machine learning has been an active area of research. Below is an overview of past efforts:

A. K-Nearest Neighbors (KNN)

KNN classifies data points based on the majority class of their k nearest neighbors. It's simple and effective for various classification tasks. Parallelization is needed because KNN's distance calculations become computationally intensive for large datasets, with time complexity of O(D) where D is the number of records. Recent approaches to parallelize KNN include:

- GPU-based implementations for distance calculations. [1]
- Distributed KNN algorithms using frameworks like Apache Spark

.

B. Principal Component Analysis (PCA)

PCA is a dimensionality reduction technique that identifies the most important features in a dataset. It's useful for data compression and visualization. Parallelization of PCA is necessary for handling large matrices efficiently, especially when dealing with high-dimensional datasets. PCA parallelization efforts include distributed implementations using libraries like ScaLAPACK for handling large matrices efficiently and GPU-accelerated PCA for high-dimensional datasets

C. Decision Trees

Decision Trees make predictions by splitting data based on feature values, forming a tree-like structure. They're interpretable and handle both classification and regression tasks. Parallelization can speed up node splitting evaluations, especially for large datasets with many features. Decision Tree parallelization has been explored through various approaches:

- The Streaming Parallel Decision Tree (SPDT) algorithm, designed for distributed environments and streaming data.[2]
- Parallel approaches for decision tree-based explainability algorithms. [3]
- MR-P-Tree, a parallelization of the P-Tree algorithm using Map-Reduce for large-scale datasets. [4]

D. Random Forest

Random Forest is an ensemble learning method that constructs multiple decision trees and combines their outputs for predictions. It's robust against overfitting and handles missing data well. Parallelization is beneficial as each tree can be built independently, allowing for faster processing of large-scale projects. Random Forest parallelization has seen significant advancements:

- Horizontal parallelism, where different processors handle different examples.
- Independent tree construction allows for efficient parallelization

E. Neural Networks

Neural Networks consist of interconnected nodes (neurons) organized in layers, learning complex patterns through backpropagation. They're versatile and powerful for various machine learning tasks. Parallelization is crucial for neural networks to distribute computations across multiple processors, reducing training time for large models. Neural network parallelization has been a major focus in recent years:

- DeepPCR, a novel algorithm that parallelizes sequential operations in neural networks, achieving up to 3x faster training. [5]
- Data-parallel and model-parallel approaches for distributing computations across multiple processors. [1]

III. Methodology

A. Language and System Choices

We chose Python due to its extensive ML libraries. Key libraries include NumPy, Scikit-learn, SciPy, PyTorch, and concurrent.futures.

B. Concurrency and Synchronization Techniques

- **KNN**: n the K-Nearest Neighbors (KNN) implementation, data parallelism is employed as the primary strategy. The test set is divided into manageable chunks, which are then processed concurrently using ThreadPoolExecutor. This approach leverages the inherent parallelism in the KNN algorithm, where predictions for different test samples can be computed independently. Synchronization in the KNN implementation is achieved implicitly through ThreadPoolExecutor's map function, with results from all threads being collected and concatenated upon completion of parallel processing.
- **PCA**: For the Principal Component Analysis (PCA) implementation, a hybrid approach to concurrency is adopted. While the computation of the covariance matrix remains single-threaded, the computationally intensive eigen decomposition step utilizes ThreadPoolExecutor. This strategic use of concurrency targets the most time-consuming part of the PCA algorithm. Synchronization in the PCA implementation is managed through ThreadPoolExecutor's submit method, with the eigen decomposition result retrieved asynchronously using the Future object's result method.
- **Neural Networks**: The Neural Network implementation takes advantage of PyTorch's DataParallel functionality to achieve model parallelism. This technique replicates the neural network model across multiple GPUs when available, allowing for distributed computation of forward and backward passes. PyTorch's internal mechanisms handle the synchronization aspects of DataParallel, ensuring consistent model updates across all devices. The main execution thread maintains overall control, coordinating the progression of epochs and batches during training.
- **Random Forest**: The Random Forest algorithm uses a divide-and-conquer approach, distributing estimators among threads for independent training of sub-forests, which are later combined into a single classifier. All implementations employ ThreadPoolExecutor for efficient thread management, with the number of threads controlled by a parameter for flexible scaling. This parallel processing approach significantly reduces computation time while maintaining the statistical properties and predictive power of the algorithms, showcasing the effective use of concurrent programming techniques in machine learning applications.

C. Experimental Setup

Synthetic datasets of varying sizes were used. Thread counts tested ranged from 1 to 64. Metrics included execution time, speedup, and efficiency, scalability and parallel overhead.

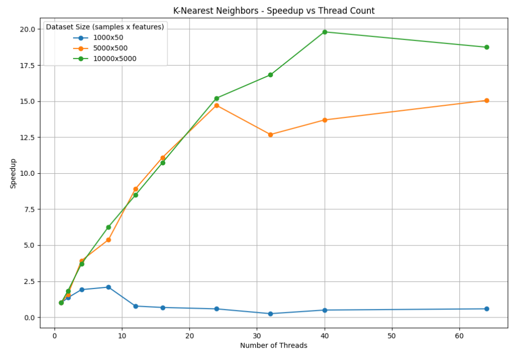

Fig. 1. KNN: Speedup vs number of threads

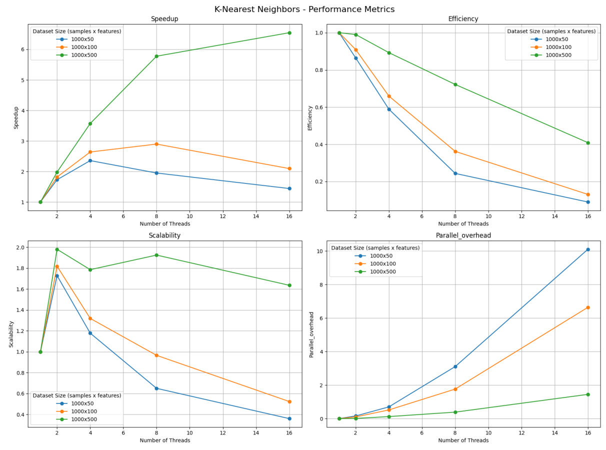

Fig. 2. KNN metrics

IV. Conclusions

Our experimental results revealed distinct parallelization patterns across different machine learning algorithms:

A. Algorithm-Specific Performance

K-Nearest Neighbors demonstrated exceptional parallelization efficiency as seen in Fig 1 and Fig 2:

- Achieved up to 20x speedup with 40 threads for large datasets (10000x5000)
- Showed near-linear scaling up to 20 threads
- Small datasets (1000x50) exhibited poor scaling beyond 8 threads
- Maintains high efficiency (0.4) even at 16 threads
- Best scalability metrics among all algorithms
- Lower parallel overhead for larger datasets

Principal Component Analysis showed limited parallelization benefits that can be observed in Fig 3 and Fig 4 :

- Larger datasets showed minimal improvement (1.1-1.2x)
- Performance remained relatively constant regardless of thread count
- Maximum speedup of 1.23x for smaller datasets (1000x50)

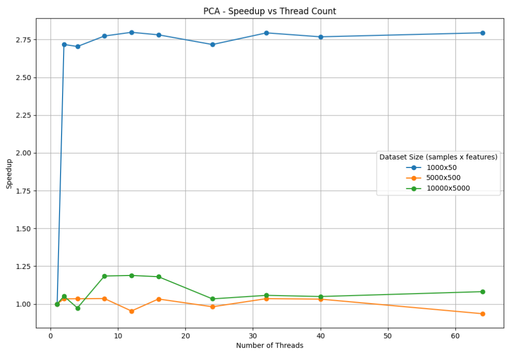

Fig. 3. PCA: Speedup vs number of threads

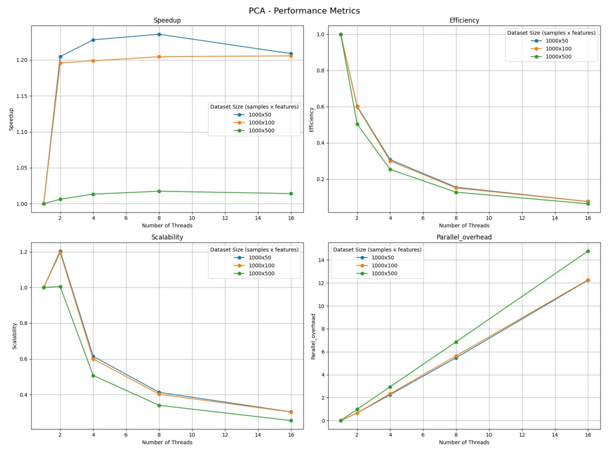

Fig. 4. PCA metrics

- Performance plateaus after 8 threads
- Efficiency drops rapidly from 1.0 to 0.1 as thread count increases
- Parallel overhead grows linearly, reaching 15x at 16 threads for 1000x500 dataset

Random Forest displayed moderate but consistent improvements as seen in Fig 5 and Fig 6:

- Peak speedup of 1.9x for medium-sized datasets (5000x500)
- Consistent performance across different dataset sizes
- Plateaued after 20-30 threads
- Best performance with 1000x500 dataset (1.8x speedup)
- Near-linear speedup improvement up to 8 threads
- Uniform efficiency decline across all dataset sizes
- Parallel overhead increases linearly but remains manageable

Neural Network performances can be seen in Fig 7 and Fig 8:

- Minimal speedup across all dataset sizes (maximum 1.075x)
- Smaller dataset (128x32) shows slightly better performance

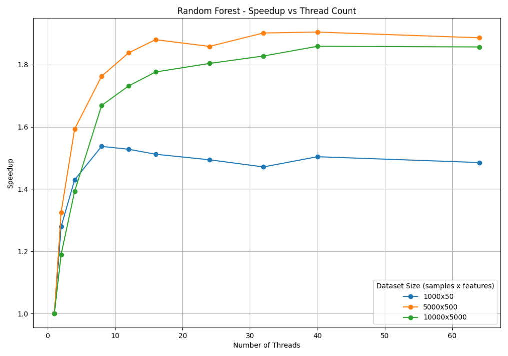

Fig. 5. Random Forest: Speedup vs number of threads

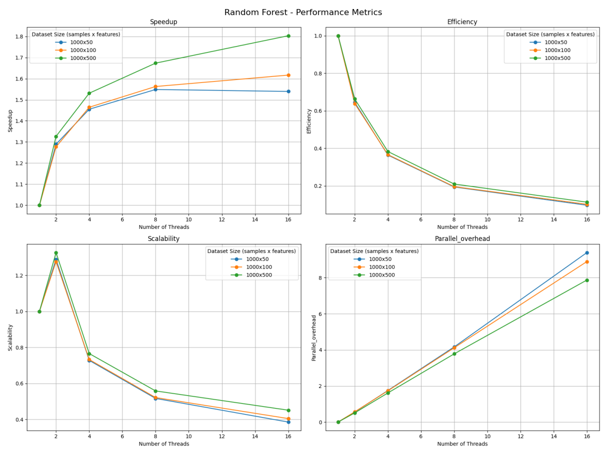

Fig. 6. Random Forest metrics

- Performance degradation for larger dataset (1024x64)
- Reasoning: Neural networks are inherently sequential in training, with strong dependencies between layers. The overhead of thread management often outweighs potential parallel processing benefits.
- Small dataset (128x32) achieves 1.6x speedup
- Larger dataset (512x64) shows minimal improvement
- Efficiency declines sharply after 2 threads

B. Cross-Algorithm Comparison

Metric	PCA	NN	RF	KNN
Max Speedup	1.23x	1.6x	1.8x	6.5x
Optimal Threads	8	2	16	16
Scalability	Poor	Poor	Good	Excellent
Overhead Growth	Linear	Steep	Linear	Moderate

TABLE I
ALGORITHM PERFORMANCE COMPARISON

These results demonstrate that parallelization strategies must be carefully tailored to both algorithm characteristics and dataset properties for optimal performance.

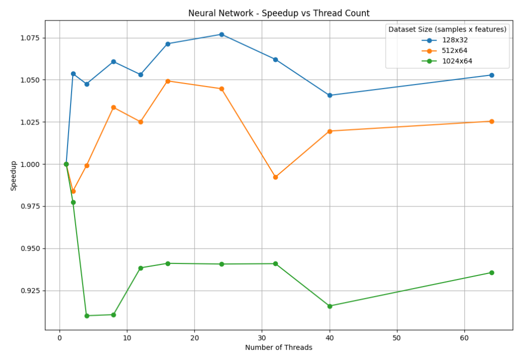

Fig. 7. Neural Network: Speedup vs number of threads

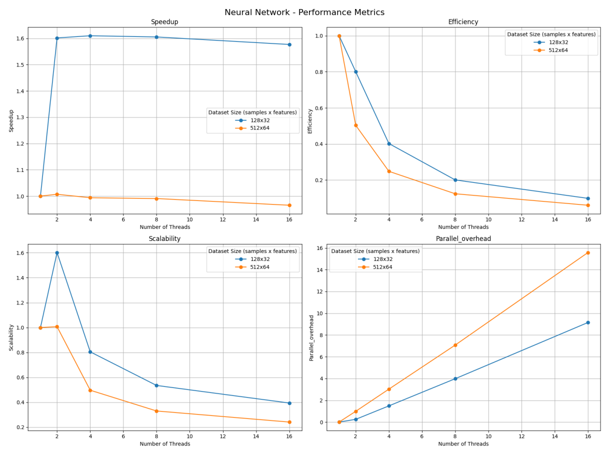

Fig. 8. Neural Network metrics

C. Key Findings

- Dataset size significantly impacts parallelization effectiveness
- Thread scaling varies by algorithm and problem size
- Optimal thread count differs for each algorithm
- Diminishing returns observed beyond 40 threads across all algorithms

D. Limitations

- Memory bandwidth constraints in high-dimensional datasets
- Thread management overhead at higher thread counts
- Algorithm-specific bottlenecks (e.g., PCA's matrix operations)
- Limited scalability for small datasets

These findings highlight the importance of algorithm-specific parallelization strategies and careful consideration of dataset characteristics when implementing parallel machine learning solutions.

V. Future Work

Future work in parallelizing machine learning algorithms offers promising avenues for enhancing computational efficiency and scalability. Building on current implementations of KNN, Neural Networks, PCA, and Random Forests, several directions can be explored:

1) Hybrid Parallelization: Combining data and model parallelism for improved performance.
2) Adaptive Threading: Implementing dynamic thread allocation based on real-time metrics.
3) Distributed Computing: Extending parallelization to distributed systems for processing massive datasets.
4) Algorithm-Specific Optimizations: Tailoring strategies to each algorithm's unique characteristics.
5) Hardware-Aware Implementations: Developing techniques adapted to specific hardware architectures.
6) Asynchronous Learning: Investigating methods to reduce synchronization overhead in distributed settings.
7) Automated Parallelization Frameworks: Creating systems that automatically parallelize algorithms based on data and hardware resources.

These advancements could significantly enhance the processing of larger datasets, more complex models, and faster iteration cycles in AI research and applications, pushing the boundaries of parallel processing in machine learning.

References

[1] Seyedalireza Yektamaram, Optimization algorithms for machine learning designed for parallel and distributed environments, PhD Dissertation, Lehigh University, 2018.

[2] Yael Ben-Haim and Elad Tom-Tov, Ä Streaming Parallel Decision Tree Algorithm,Journal of Machine Learning Research, vol. 11, pp. 849-872, 2010.

[3] Daniela Loreti and Giorgio Visani, P̈arallel approaches for a decision tree-based explainability algorithm,F̈uture Generation Computer Systems, vol. 158, pp. 308-322, 2024.

[4] Hasna Chamlal et al., Ä preordonance-based decision tree method and its parallel implementation in the framework of Map-Reduce,Äpplied Soft Computing, vol. 167, 2024.

[5] Federico Danieli et al., D̈eepPCR: Parallelizing Sequential Operations in Neural Networks,Ädvances in Neural Information Processing Systems, vol. 36, 2024.