An Exploration of Parallelization Techniques for Improving LightGBM Performance

Light Gradient Boosting Machine (LightGBM) is a highly efficient gradient boosting framework developed for speed and scalability. It has gained widespread adoption in the machine learning community due to its ability to handle large-scale data and deliver high predictive accuracy with minimal computational overhead. LightGBM is particularly suitable for real-time applications and big data environments, enabling its use across diverse domains such as finance, energy, healthcare, and high-performance computing. This report will cover the findings of our experiment in further improving the performance of LightGBM through various techniques used for parallelization.

Reference Paper: “Challenges and Opportunities of Building Fast GBDT Systems" by Zeyi Wen et al. (2021)

We closely examined the paper by Wen et al. due to its focus on LightGBM and used it as a reference for our experiment. Below is a summary of key insights from their work.

Gradient Boosting Decision Trees (GBDTs) are known for their accuracy and robustness in structured data tasks. Among GBDT implementations, LightGBM, developed by Microsoft, stands out for its speed and scalability. Wen et al.’s 2021 paper, “Challenges and Opportunities of Building Fast GBDT Systems,” outlines the innovations that drive LightGBM’s performance.

A key feature of LightGBM is its histogram-based algorithm, which bins continuous features to reduce computation and memory usage. This approach accelerates split finding and improves efficiency, especially on large, high-dimensional datasets. The authors also highlight GOSS, which keeps all instances with large gradients and randomly samples those with small gradients. This reduces data processed per iteration while maintaining accuracy. Another technique, EFB, combines mutually exclusive features to reduce dimensionality in sparse datasets. This is particularly useful for text and categorical data. Unlike XGBoost’s level-wise growth, LightGBM uses a leaf-wise strategy, splitting the leaf with the greatest loss reduction. This often improves accuracy but can cause overfitting on small datasets. Wen et al. also detail LightGBM’s optimized support for multithreading and GPU acceleration, particularly its efficient histogram construction on GPUs.

Despite its strengths, the authors note areas for improvement, such as dynamic hyperparameter tuning, better handling of imbalanced data, and enhanced support for multi-GPU and distributed systems. They advocate for more adaptive, hardware-aware algorithms to further advance GBDT systems like LightGBM.

Methodology Overview

Our code implements a full machine learning pipeline using LightGBM on structured data. We load the dataset files of train_set.csv and test_set.csv from disk. We keep track of execution time through time.time().

For parallel preprocessing, we use ThreadPoolExecutor, with the parallelization happening in following part in particular:

```
with ThreadPoolExecutor() as executor:

    future_train = executor.submit(preprocess, train_set, True)

    future_test = executor.submit(preprocess, test_set, False)
```

Also, two functions (preprocess) are run in parallel threads, with one handling training data (splitting X and Y, dropping RecordId, X71, X76) and the other preparing test data (drops same columns). This reduces I/O-bound preprocessing time but offers limited gains for CPU-bound tasks.

Note that this is not a LightGBM feature, but a Python-level threading optimization. Wen et al. also mention that LightGBM optimizations are C++/GPU-level, not Python wrapper-level.

The imputing and scaling stage is done sequentially, but fast, as follows:

```
imputer = SimpleImputer(strategy='mean')

scaler = MinMaxScaler()
```

SimpleImputer fills the missing values using column-wise mean, and MinMaxScaler scales all features to [0,1]. This stage is not parallelized, but would benefit less from it compared to model training.

The data is then split using the standard scikit-learn method and is non-parallel:

```
train_test_split(..., test_size=0.3)
```

The model training is where most of our experiments with parallelization techniques were done. We made use of GPU and multithreading to improve performance as follows:

```
model = lgb.LGBMClassifier(
    boosting_type='gbdt',
    device_type='gpu',
    ...
    n_jobs=-1,
    num_threads=-1
)
```

device_type='gpu' indicates that GPU acceleration is enabled for training. This results in LightGBM using GPU memory to store histograms. It also performs faster split finding by offloading histogram construction to the GPU. Moreover, it also enabled by default. n_jobs=-1 and num_threads=-1 uses all available CPU cores during training, meaning multithreaded histogram construction and training.

Wen et al. describe LightGBM's GPU training as carefully optimized to avoid memory contention and reduce transfer overhead, which is what we are leveraging.

For model evaluation, we made use of the standard ROC-AUC for binary classification: roc_auc_score(...). For inference on the test set, we used model.predict_proba(...) which is parallelized internally in LightGBM using the n_jobs and GPU settings. Finally, we save the output in a simple CSV file.

The following table provides a straightforward comparison of our approach with the Wen et al. paper:

Technique	**Used in Code?**	**Described in Paper**	**Comments**
Histogram-based learning	Yes (default in LGBM)	Yes	Improves speed and memory efficiency

GOSS (Gradient One-Side Sampling)	No	Yes	Accelerates convergence, helps with large data
EFB (Exclusive Feature Bundling)	No	Yes	Especially helpful in sparse, high-dimensional data
Leaf-wise growth	Yes (default)	Yes	Allows deeper trees, may overfit without tuning
GPU acceleration	Yes (device_type='gpu')	Yes	LightGBM uses optimized GPU kernel for histogram
Multithreading	Yes (n_jobs=-1)	Yes	Parallel split finding and training
Python-level parallelization	Yes (ThreadPoolExecutor)	No	Our addition, simplifies multithreading

The following is a brief overview of all the parallelization techniques in this project:

Stage	**Parallelization Type**	**Description**	**Benefit**
Preprocessing	Thread-based (ThreadPoolExecutor)	Runs two preprocessing tasks in parallel	Slight gain in speed

Imputation/Scaling	None	Sequential scikit-learn calls	–
Model Training	Multithreaded + GPU	LightGBM with n_jobs=-1, device_type='gpu'	Major boost because of GPU histogram and fast leaf-wise growth
Prediction	Multithreaded/GPU	Parallelized inside LightGBM	Fast inference

Final Observations on Comparison with Wen et al.

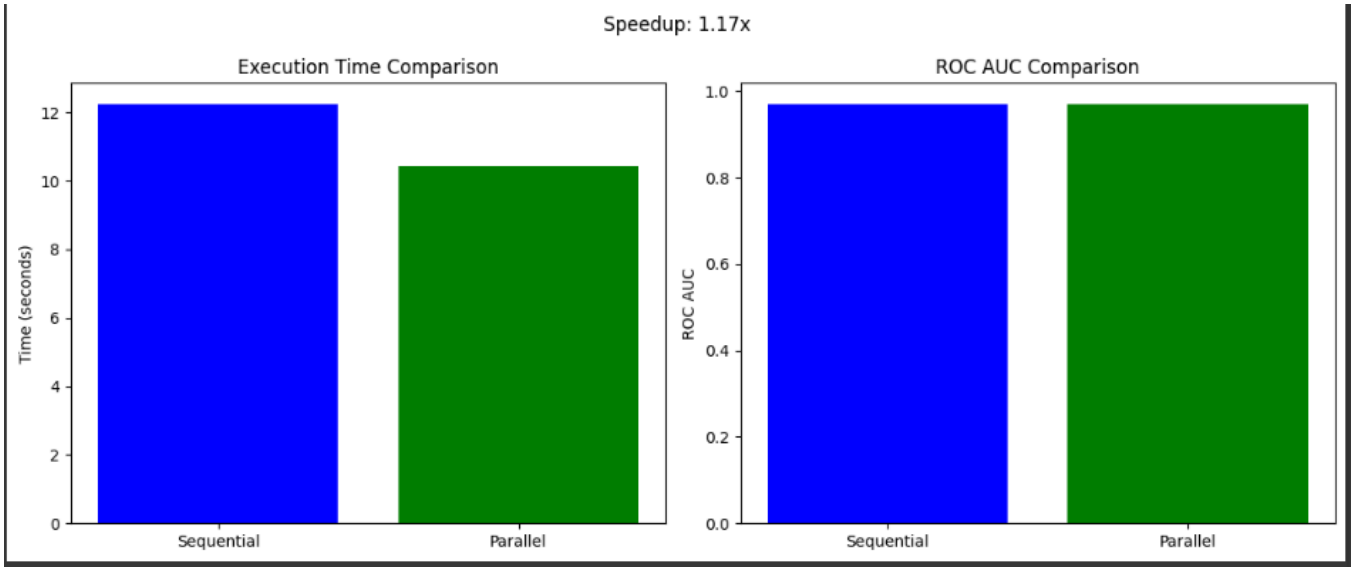
Wen et al. imply best practices for fast GBDT and we have made meticulously ensured our code closely aligns with what Wen et al. by ensuring the following in our code: histogram-based training, GPU acceleration, multithreaded training, and use of default LightGBM strategies like leaf-wise growth. We have made an additional use of ThreadPoolExecutor, which is not discussed in the paper, as it's a Python-level addition, but it helps to reduce total pipeline time beyond the model training phase.

During this endeavor, we also encountered several challenges. Initially, we used the pipeline skeleton, assuming that the sequential nature of our code would benefit from stage-level parallelism like in the pipeline skeleton. However, instead of improving performance, it introduced significant overhead. According to our research, this is due to the cost of synchronizing between pipeline stages and the difficulty in balancing workloads across stages. Since many of the LightGBM operations are not uniform in execution time, some stages became bottlenecks, which limits the effectiveness of pipelining.

We also tested map skeleton and reduce skeleton, assuming that we could treat LightGBM as a reduction over mapped preprocessing or training jobs (meaning processing batches or folds in parallel). This idea was promising, especially in ensemble scenarios or distributed settings. However, it led to huge overhead with LightGBM, possibly because LightGBM already uses efficient parallelization (via feature- or data-parallel GBDT training). Duplicating the training step across mappers adds redundant computation, especially when using the GPU trainer which is already highly optimized for large datasets. Also, synchronization and merging in the reduce

step added non-trivial communication cost, which was unnecessary in our case since we only needed one final model rather than aggregating multiple.

The resulting performance is as follows, with a 1.17 times improvement in execution time as follows in the table below. This is calculated by dividing the sequential time by the parallel time. Note that ROC AUC remains about the same as improving it was out of the scope of this project.



Code	Roc auc	Exec time	CPU usage	Memory usage
sequential	96.2%	20-21 seconds	159.3%	1328.79 mb
parallel	96.9%	15-16 seconds	209.8%	1605.11 mb

The slight increase in ROC AUC for parallel may be due to minor randomness or better exploration during training (meaning due to faster iteration allowing more fine-tuning in the same wall time).The execution time reflects the expected performance boost from parallelizing, especially since LightGBM is known to scale well with threads. As for CPU usage, it being greater than 100% means multi-threading is being used. The higher usage in parallel mode aligns with increased core utilization. As for memory usage, slightly higher memory usage is expected in parallel mode due to thread overhead and potential replication of data buffers.

Conclusion

Our exploration of parallelization in LightGBM showed a 1.17 times speedup in execution time without compromising model quality. By combining Python-level thread parallelism in

preprocessing with LightGBM's built-in GPU and multithreading features, we further optimized performance. These results support Wen et al.'s (2021) findings on LightGBM's efficiency through histogram-based learning, leaf-wise growth, and GPU acceleration. Our use of ThreadPoolExecutor complemented these features, though some parallelization attempts (such as the pipeline and map-reduce skeletons) introduced excessive overhead and were ineffective. The increased CPU and memory use were acceptable trade-offs for the gains achieved, confirming effective use of available resources. Future work could explore dynamic hyperparameter tuning, distributed training, and deeper integration of GOSS and EFB to further enhance performance on larger or more complex datasets. Overall, our results show that parallelization, when aligned with algorithm-specific optimizations, can significantly improve even well-optimized systems like LightGBM.