2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS) | 979-8-3503-6387-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ACSOS61780.2024.00038

# Characterization of Parallelism Techniques for Decision Tree Ensembles

Eduardo Romero Gainza
*The Ohio State University*

Christopher Stewart
*The Ohio State University*

Austin O'Quinn
*The Ohio State University*

***Abstract*—Ensembles of decision trees enhance accuracy of individual decision trees by combining the predictions of multiple trees. Recent research accelerated inference for ensembles running on a single core. However, in most previous work, parallelism is either not sufficiently explored or assumed to be trivial in ensembles. Novel data structures introduced in previous can support well known methods to parallelize inference, i.e., data parallelism and model parallelism. In addition, ensembles also support another method of parallelism: structure parallelism. In structure parallelism, portions of a restructured forest are distributed across cores, enabling parallelism even in the execution of a single sample. We compared the different methods for parallelizing ensembles and showed that structure parallelism can be 42% faster than existing methods alone.**

## I. Introduction

Decision trees [12] are hierarchical structures that split input data sequentially until reaching a classification. Decision trees are fast, simple and explainable Machine Learning (ML) models [4], [6], [14]. However, individual decision trees do not achieve accuracy as high as other ML models [8]. Ensembles of decision trees, such as random forests and deep forests [20], are collections of decision trees in which the predictions of multiple trees are combined to increase accuracy. These models incur performance penalties due to the increase in trees, redundancies across multiple trees and intrinsic inefficiencies of breadth-first execution of individual trees [13].

Recent work has mitigated some of these concerns by producing alternative execution paths on ensemble models. For example, Forest Packing [3] interleaves different trees and reorganizes nodes to increase cache locality. RADE [18] creates a smaller random forest that accurately classifies a high percentage of samples, and only uses the more complex ensemble model in samples where the smaller model has low confidence in its prediction. Bolt [13] restructures ensembles into two separate data structures: a lookup table where paths are matched with corresponding predictions and a dictionary in which subsets of features are matched with entries in the lookup table. While all of these techniques reduce the execution time of ensemble models, they are optimized for single core execution. To be sure, all of these methods can be implemented in a distributed system, but they are not by default optimized for them. While ensemble models are trivially parallelizable, more nuanced techniques could be more effective and are worth exploring. Better techniques for parallelism of ensembles can further increase scalability and allow for more complex models to be deployed.

Parallelism in machine learning is typically done via data parallelism or model parallelism [17]. Data parallelism consists of replicating the model in multiple cores. By having replicas of the model, batches of samples can be distributed across the system so that each core entirely computes classification of a sample in a copy of the model. This approach is only effective if there are enough samples to efficiently use all cores. In an online system, this requires that samples arrive with enough frequency to maximize utilization of all cores. Further, for large ensemble models, a single sample could be slow to compute in a single core. Thus, in some scenarios, model parallelism is more effective. In model parallelism, partitions of the model are created, and separated in different cores. With traditional models, partitioning the ensemble consists of simply assigning subsets of decision trees to each core. During inference, the samples are sent to all cores and then each core reports the answer for a given sample in its tree subset. Then, answers are aggregated, using a central node, establishing communication protocols between multiple cores where some cores combine larger subsets from other cores, or pushing partial answers to edge devices [1], [2]. Regardless of aggregation method, this approach has the advantage of executing a single sample in multiple cores which benefits large models. Thus, depending on the workload demands, either method or a combination can be appropriate.

Bolt [13] allows for both of these kinds of parallelism, plus an additional dimension. Because Bolt transforms ensemble models into two data structures, a lookup table and a dictionary that determine storage and compute requirements respectively, each of these structures can be divided according to system needs. Thus, in Bolt, model parallelism can be done in two ways. We refer to splitting the lookup table, i.e. reducing memory demands, as model parallelism, while we refer to partitioning the dictionary as structure parallelism. Due to this extra dimension for parallelism, in Bolt model and structure parallelism can be adjusted to adapt to memory and compute demands independently. This approach can be further combined with data and model parallelism. For this paper, we studied the tradeoffs associated with these parallelizing inference in ensembles of decision trees. Our experiments showed that structure parallelism can be up to 42% faster than data parallelism on medium-sized and large ensembles, while data parallelism is better for small models. We also showed how model parallelism in Bolt can reduce memory demands by up to 90%.

979-8-3503-6387-6/24/$31.00 ©2024 IEEE
DOI 10.1109/ACSOS61780.2024.00038

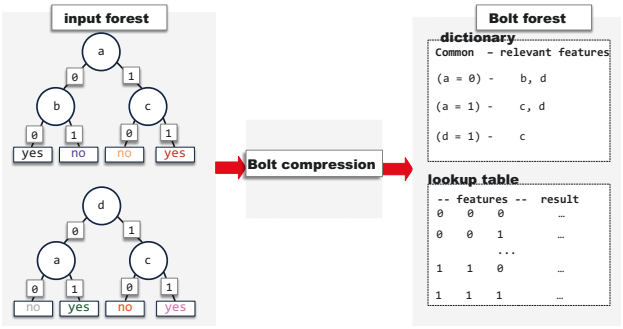


Fig. 1. Bolt forest overview

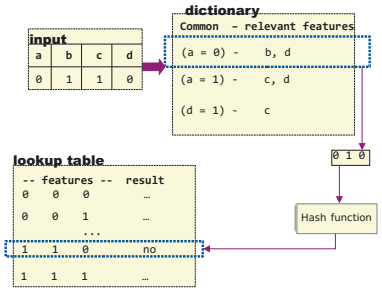


Fig. 2. Sample inference example in Bolt

## II. Background

The key idea behind Bolt [13] is that breadth-first execution of decision trees is inefficient because unpredictable paths to a tree leaf can cause cache misses and errors by the branch predictor. Further, in an ensemble of trees, redundancies are not exploited by breadth-first execution. Bolt is an algorithm that speeds up ensemble models by restructuring ensembles into data structures that fit in cache and follow predictable access patterns, thus minimizing cache misses and branch mispredictions. Figure 1 shows an overview of a Bolt forest. Bolt takes a trained ensemble model as input and outputs two data structures called dictionary and lookup table. Both of these data structures are produced by listing all root-to-leaf paths and then grouping them by similarity. Using a tunable parameter for the group sizes, Bolt creates dictionary entries from each group. As shown in Figure 1, each entry in the dictionary lists all the *relevant features* - i.e. features used in the paths grouped in the entry - and adds information about features that are common in all such paths. The lookup table contains all the responses associated with each path and ordered by the results of a hashing function. Note that the lookup table is not a full memory mapping of all possible paths in the forest. As described in Bolt [13], hashing is used to significantly reduce the space needed by the lookup table.

Figure 2 shows the connection between the two data structures of Bolt by highlighting the inference process. In Figure 2, there is an input sample with four binary features. Following the prediction path in the trees (Figure 1), the classification should be ”no.” During inference the sample is compared to every dictionary entry sequentially. Looking at the first (highlighted) dictionary entry, the input matches the common feature of that dictionary entry, i.e. $a = 0$. So, the first dictionary entry is considered relevant. Afterwards, a binary sequence using all relevant features (a, b and d) is created. This sequence and the location of the entry in the dictionary are used to hash into a position in the lookup table. This table location contains the final prediction (”no”). After computing this result, the other entries in the dictionary still need to be checked. Thus, the input is compared to the entry where $a = 1$ and the entry where $d = 1$. Since the common features do not match the input, these entries are ignored and execution ends.

Bolt showed results that are at least $2X$ faster than competing approaches on a single core [13]. This improvement is due to a few key ideas. First, since there is a fixed order in which the dictionary is accessed, the next entry can always be predicted, and loading an entry in cache likely also loads subsequent entries. This improves cache locality. Additionally, since the compression process also minimizes the size of the lookup table, for small forests both structures fit entirely in cache further reducing cache misses. The inference method also reduces the total number of branches taken during the execution. Branches are limited to identifying if the common features of a dictionary entry match the input. Bolt [13] shows that the total number of branches taken in an execution is much smaller than with breadth-first execution. Thus, the total number of branches missed is also reduced.

However, another advantage of the approach was not explored fully in the initial work on Bolt. The data structures that comprise a Bolt model are easy to parallelize. To be sure, as with any other ensemble model, Bolt models support data parallelism. Normally, a random forest also supports model parallelism by assigning different trees to different cores. Bolt can do the former easily, but the latter has a different meaning in Bolt. Instead of splitting trees, Bolt can partition the lookup table to reduce memory demands. However, this is not the only way of splitting the model in Bolt. Bolt can also support a novel parallelism method, called *structure parallelism*, in which the dictionary is split. This is a step forward with respect to assigning subsets of trees to a core. Since the paths in a dictionary entry do not necessarily correspond to a single subtree, splitting the dictionary can be equivalent to splitting a single decision tree. For example, in Figure 1, the first dictionary entry corresponds to paths in both of the decision trees in the original forest. Further, Bolt [13] showed that its benefits are greater in smaller ensemble models, achieving results that are orders of magnitude faster than competing approaches. This is because with smaller forests the likelihood that both data structures fit entirely in lower levels of cache increases. Thus, the ability to split the dictionary can mimic the effects of having a smaller forest to begin with, further improving performance.

## III. Design of Experiments

In our experiments, we studied the methods of parallelizing Bolt. Figure 3(a) shows the simplest form of parallelism for a Bolt forest, data parallelism. Here there are two cores, each with an identical copy of a full Bolt forest. Input samples are redirected to a free core. Each core handles inference fully on a single sample. This approach is simple and it can increase throughput in the whole system. For that to happen, however, requests should arrive frequently enough to keep core busy, and the ensemble model should be small enough to execute efficiently on a single core. If inference time is bottlenecked by the model's memory footprint, an alternative method could be more beneficial.

Figure 3(b) shows another approach for parallelism on Bolt: structure parallelism. Here, the dictionary of a Bolt forest is split across both cores. Samples are compared to common features of each dictionary entry. Only in those entries in which there is a match, *relevant features* are used to hash into a lookup table. Reducing the dictionary size reduces the compute time of each core.

For larger models, cache size and locality might be a bottleneck. In these models, splitting the lookup table, i.e. model parallelism, can reduce the memory demands of the system. While this does not directly affect the amount of computation during inference, it can speed up the model by reducing cache misses. Figure 3(c) shows how to split Bolt's lookup table. In this example, there are two cores and each of them has only half of the lookup table. For this approach to work, both cores require a full copy of the dictionary. After dictionary comparisons are made, if the entry is relevant, a hashing function of the input features is used to find a location in the lookup table. To avoid errors that can occur if the hash functions goes to a portion of the table not included in a core, such lookups must be ignored. For this not to harm accuracy, replication of the dictionary is necessary. For example, in Figure 3(c), if a sample from the second entry of the dictionary maps to location $101$ in the lookup table, in core 1, this location would not be retrievable. Thus, the lookup is simply ignored and the next dictionary entry is checked. In core 2, however, this entry must be retrieved to maintain correctness. Thus, all dictionary entries are necessary in both cores and results from both cores must be aggregated in the same manner as with the dictionary splits.

## IV. Experimental Setup

We tested the parallelism of Bolt using an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz with 24 cores. The system has 30MB of LLC and 132 GB of memory and runs Red Hat Enterprise Linux version 7.9. Our platform runs Python 3.6.8 and Scikit-Learn [11] 20. We report the results of using 1, 2, 4, 8, 16 and 23 cores for parallelism. Note that we do not use all 24 cores for parallelism because one of them is reserved for the client to send the requests to all other cores.

### A. Datasets

To test the different parallelism methods we used 4 datasets: MNIST [9], CIFAR100 [7], LSTW [10], and the Yelp Restaurant Review Dataset [19]. MNIST and CIFAR100 are common image recognition datasets. MNIST consists of $70000$ black and white images of handwritten digits. $60000$ of those images are used for training and the remaining $10000$ are the test set. Each image is $28 \times 28$ pixels where each pixel is a feature ($784$ in total). There are $10$ output classes corresponding to each digit. In all our experiments using MNIST we report the time it took our model to classify all $10000$ test samples. CIFAR100 has $60000$ color images of common items (such as airplane, cat, dog, truck, etc.) There are $100$ output classes corresponding to each of the items. Each image has $32 \times 32$ pixels and each pixel has $3$ color channels, so, there are $3072$ input features. The training set consists of $50000$ images while the test set uses the remaining $10000$. In all our experiments we report the time to classify $2500$ images in the test set. We also tested our model in a more heterogenous dataset, such as the Large-Scale Traffic and Weather Events (LSTW) dataset [10]. Input data to LSTW includes both numeric and categorical features related to traffic conditions in different locations across the USA. Each sample consists of $11$ input features and the prediction target was the "severity" column in the dataset, which is a categorical assessment of traffic congestion. The dataset is growing, however; at the time of retrieval the dataset had 25M data points. We separated $700000$ datapoints for testing while the rest of the almost 25M samples were used for training. However, after testing, we report the execution time for classification of the first $5000$ samples. Finally, we used the Yelp Restaurant Review Dataset [19]. This is a Natural Language dataset that uses $5200000$ user reviews on $174000$ businesses from $11$ metropolitan areas. The input data is the text written by users. We removed stop words, reduced words to their stems and tokenized them into a vector that indicates the presence and number of occurrences of the most common $1500$ words in the dataset. Thus, we transformed the dataset into $1500$ numerical features. The classification target is the number of stars out of 5 given by the user to the restaurant, expressed as an integer. For this dataset we report the execution time on classifying $300$ reviews.

### B. Measurements

All samples were sent using web sockets. One core was used for the client to send the requests to all the Bolt servers. Each core listened on a different port and was sent all the appropriate samples according to the parallelism design. The predictions given by all cores are then written to different files. The timing of all experiments was measured from the time a core received the samples to the time that a core finished execution.

### C. Ensembles Used

In all experiments we chose 3 different ensembles. All 3 ensembles are random forests that are then restructured using Bolt and deployed in the reported number of cores. The first

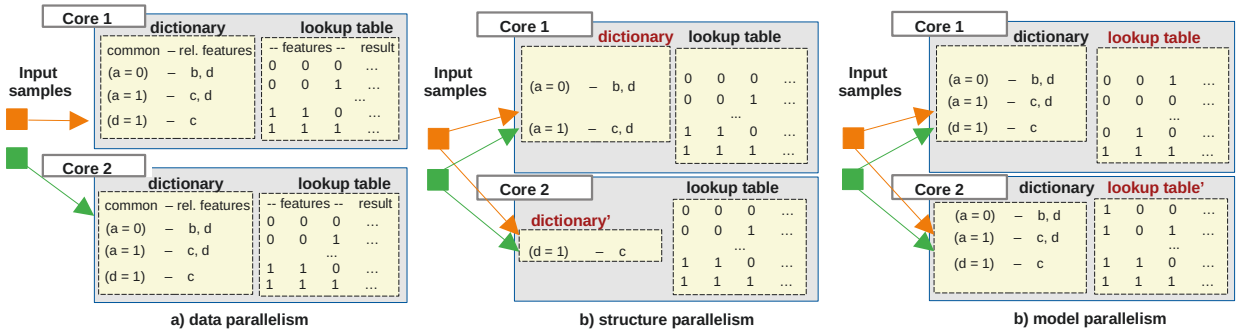


Fig. 3. Parallelism methods in a Bolt forest.

forest reported for each dataset is a small forest comprised by 10 decision trees, each with a maximum height of 5. Next we report the results on a medium forest. For the medium forest we used the same number of trees (10), but increased the maximum height of each tree to 8. This decision is based on the numbers reported in Bolt [13] where the height of the trees had the biggest impact in the execution time. Finally we tested on a large forest. For the large forest we increased the number of trees to 30 and the maximum tree height to 12.

## V. Results

For all results we report the average execution time after 10 runs as well as the error bars indicating the maximum and minimum execution time of any of the cores in any of those runs. Particularly, in structure and model parallelism, since results need to be aggregated, outliers occur often and cause significant performance degradation. However, mitigating tail latency [5], [15], [16], is outside of the scope of this paper.

Figure 4(a) shows the execution time of the test set of MNIST on a small forest. The average execution time is lower for structure parallelism up to 8 cores. However, with 16 and 23 cores it is faster to use data parallelism. Further, the maximum execution time on any of the runs with 4 and 8 cores is larger in structure parallelism even if the average is lower. This is likely caused by the way the Bolt dictionary is split (partitions are made by arbitrarily assigning dictionary entries to different locations). In a very small forest, likely only a few paths are used frequently, thus splitting the dictionary causes some cores to frequently do lookups in Bolt's lookup table, while other cores discard results frequently. For the larger amounts of cores, the advantage of having fewer dictionary entries to explore is overshadowed by the overhead of processing many samples. Thus, on this small ensemble structure parallelism is advantageous up to 2 cores and after that data parallelism becomes better.

Figure 4(b) shows the execution of the medium-sized forest on the same dataset. As shown in Figure 4(b), structure parallelism performs better in all numbers of cores. Structure parallelism achieves results 4% faster on 2 cores and 33% and 30% faster on 16 and 23 cores respectively. In Bolt [13], it is shown that smaller forests benefit more from the approach because of cache locality, even showing that in a small forest Bolt only had two digit cache misses in a whole test set execution. Further, in a small forest the number of dictionary entries are reduced. Since dictionary entries are the only source of branching in Bolt, smaller dictionaries minimize branch mispredictions. Thus the benefits of Bolt on a single core are greater on smaller forests. Structure parallelism is analogous to reducing forest size. Since this forest is large enough to avoid overhead taking over execution time, structure parallelism performs better.

Similarly, structure parallelism also performs better on average on all number of cores with a large forest as shown in Figure 4(c). However, the gain on structure parallelism is small on 8 cores (0.1%) but much larger on 23 cores (16%).

Figure 4(d) and Figure 4(e) show the performance of the small and medium forest on the LSTW dataset. Just like with MNIST, data parallelism is better on the small forest and structure parallelism gives better results on the medium forest. For the medium-sized forest, the smallest gain in time was 19% on 4 cores, and the largest gain was 42% on 16 cores.

Figure 4(f) shows that structure parallelism performs better on average on a large forest on LSTW for many cores. For less cores, the average is better for structure parallelism. However, there are outliers that take longer time in structure parallelism, so the maximum execution time is larger in these experiments. Consequently, the percent improvement on execution time ranges from 2% on 4 cores to 20% on 23 cores. This is likely because in an imbalanced dataset - as are traffic patterns - the same dictionary entries are frequently used (they produce searches in the lookup table), making some of the cores' loads heavier. Additionally, if a few dictionary entries produce lookups more frequently, and these dictionary entries appear in the same core, inference for that core is likely to be much slower. This last situation is more likely to happen when using less cores.

Figure 4(g), Figure 4(h) and Figure 4(i) tell a different story. In all these 3 cases, data parallelism performs better on all number of cores. Further, structure parallelism does not improve performance and is even detrimental when using many cores. Likely, this is due to the fact that even on a single core, the performance of this dataset is very fast (around

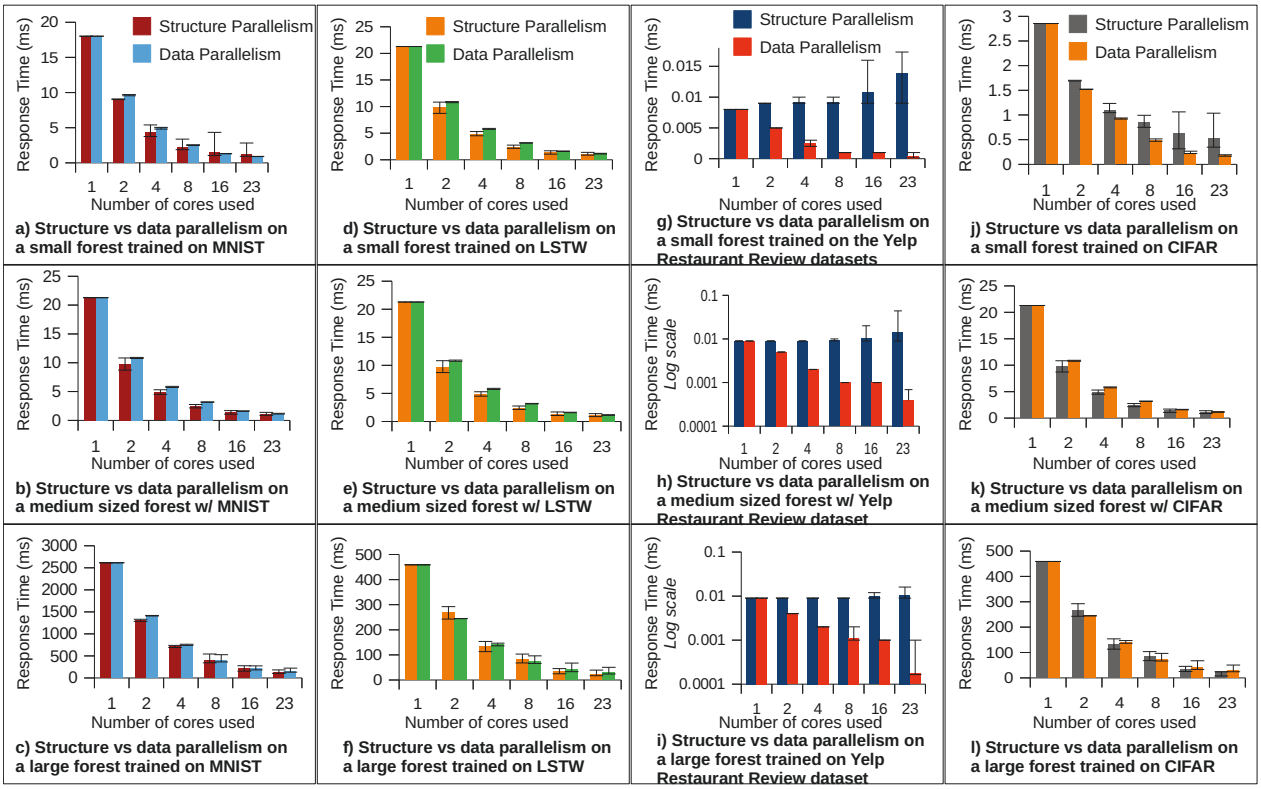


Fig. 4. Comparison of Data and Structure parallelism on a small forest trained on MNIST.

$0.009ms$ on all 300 samples). Likely, the overhead of receiving each sample and storing the sample in cache while accessing the dictionary is larger than the benefit of comparing each sample to fewer entries. This is probably aided by the fact that the ratio of the number of features used by each sample (1500 features) over the number of samples (300) is the greatest in this dataset. This causes storing each sample to be relatively more costly. Similar to the small forest in other datasets, here it seems beneficial to use data parallelism exclusively for all forest sizes.

The CIFAR100 dataset on the other hand is more similar to MNIST than Yelp, and thus the results are closer to the former dataset. Despite this dataset having the largest samples (3072 features), the nature of those features and the dataset size make the results closer to what we had seen previously. Figure 4(j), shows the performance of Bolt on a small forest using CIFAR100. As with the other datasets, on a small forest, data parallelism performs better than structure parallelism. Similarly, Figure 4(k) and Figure 4(l) show the performance of both kinds of parallelism on medium-sized and large forests: as with other datasets, structure parallelism performs better. On the medium forest the improvement of structure parallelism with respect to data parallelism is up to 23%. In the large forest the improvement can be up to 22%.

### A. *Studying Model Parallelism*

Figure 5 shows the effect of using model parallelism on the medium sized forest trained on MNIST (i.e. the same one used in Figure 4(b)). Figure 5 shows how model parallelism on such forest can be ineffective or even counterproductive. This is because, in Bolt, model parallelism does not directly impact execution time. Recall from previous sections, that model parallelism in Bolt splits the lookup table of the model which is only accessed when a relevant dictionary entry is found. The main benefit of model parallelism is reducing memory demands. This could indirectly impact performance if the reduction makes the model fit in cache or even in memory. However, this decision is a trade-off. As the model is reduced to fit in memory, there is additional branching added to the model. Now, once a relevant dictionary entry is found, the model must check if the portion of the table it is being given is relevant to the portion of the table in that core. This introduces overhead that likely offsets the advantages of reducing cache misses. Nonetheless, model parallelism can still be essential for a large forest depending on the underlying system. On the large forest trained on MNIST, (the one used in Figure 4(c)) the lookup table had 17MB in size and the dictionary was about 1MB. Figure 6 shows the effect in total memory usage

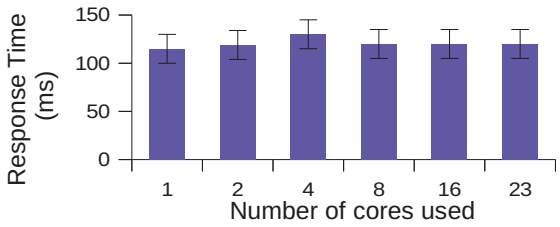


Fig. 5. Model parallelism performance on a medium sized forest trained on MNIST.

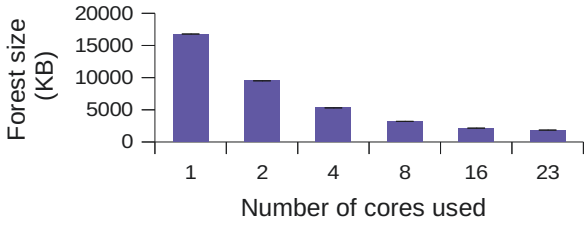


Fig. 6. Total memory needed per core using model parallelism on a large forest trained on MNIST.

of using model parallelism. By total memory usage, we include the space needed to store the dictionary which is unaffected by model parallelism. However, since the lookup table is an order of magnitude larger than the dictionary, the reduction in total memory usage is significant. Up to 8 cores, the reduction is roughly linear (18MB in one core, 9.5MB in 2 cores, 5.3MB in 4 cores and 3.2MB in 8 cores). After that the reduction in memory is limited by the dictionary being a larger percentage of space needed. However, the total memory savings using 23 cores is still an order of magnitude with respect to one core (1.8MB vs the original 18MB). This is indicative of the best use of model parallelism in Bolt. Given a forest that does not fit in available memory, this approach can make execution possible.

## VI. Conclusions

Recent work on ensemble models create novel data structures that speed up such models. These data structures are a good fit for existing parallelism techniques, namely, data parallelism and model parallelism. However, they also add opportunities for parallelism techniques that previously did not exist. Structure parallelism is a new approach afforded by the data structures of Bolt. Our experiments showed that the appropriate parallelism technique depends on many factors such as forest size, dataset, frequency of requests and available hardware. For instance, on ensembles that are sufficiently small or fast on a single core, data parallelism was far more effective than structure parallelism. Model parallelism, on the other hand, is effective only when there are strict memory constraints and the reduction in memory space used is significant. Structure parallelism is an effective technique. In most of our experiments, it is competitive with data parallelism or improves latency (up to a 42%).

While the ideal approach depends on many factors, our results suggest that structure parallelism should be one of many tools available to performance engineers. In addition, continued research on novel data structures for machine learning models could reveal new, subtle, and useful methods for parallelization beyond data and model parallelism.

**Acknowledgements:** This work was funded, in part, by NSF grant OAC-2112606.

## References

[1] J. Boubin, C. Burley, P. Han, B. Li, B. Porter, and C. Stewart. Marble: Multi-agent reinforcement learning at the edge for digital agriculture. In *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, 2022.

[2] J. G. Boubin, N. T. R. Babu, C. Stewart, J. Chumley, and S. Zhang. Managing edge resources for fully autonomous aerial systems. In *ACM/IEEE Symposium on Edge Computing*, 2019.

[3] J. Browne, D. Mhembere, T. M. Tomita, J. T. Vogelstein, and R. Burns. Forest packing: Fast parallel, decision forests. In *Proceedings of the 2019 SIAM International Conference on Data Mining*, 2019.

[4] M. Camilli, R. Mirandola, and P. Scandurra. Xsa: Explainable self-adaptation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5, 2022.

[5] N. Deng, Z. Xu, C. Stewart, and X. Wang. Tell-tale tails: Decomposing response times for live internet services. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, 2015.

[6] F. Feit, A. Metzger, and K. Pohl. Explaining online reinforcement learning decisions of self-adaptive systems. In *2022 IEEE international conference on autonomic computing and self-organizing systems (ACSOS)*, pages 51–60. IEEE, 2022.

[7] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. In *Technical Reports at University of Toronto*, 2009.

[8] P. Langley. *Elements of machine learning*. Morgan Kaufmann, 1996.

[9] Y. LeCun and C. Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010.

[10] S. Moosavi, M. H. Samavatian, A. Nandi, S. Parthasarathy, and R. Ramnath. Short and long-term pattern discovery over large-scale geospatiotemporal data. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12, 2011.

[12] J. R. Quinlan. Learning decision tree classifiers. *ACM Computing Surveys (CSUR)*, 28(1):71–72, 1996.

[13] E. Romero, C. Stewart, A. Li, K. Hale, and N. Morris. Bolt: Fast inference for random forests. In *ACM/IFIP International Middleware Conference*, 2022.

[14] E. Romero-Gainza and C. Stewart. Ai-driven validation of digital agriculture models. *Sensors*, 23(3):1187, 2023.

[15] G. Somashekar, A. Suresh, S. Tyagi, V. Dhyani, K. Donkada, A. Pradhan, and A. Gandhi. Reducing the tail latency of microservices applications via optimal configuration tuning. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems*, 2022.

[16] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *IEEE ICAC*, 2013.

[17] S. Tyagi and P. Sharma. Taming resource heterogeneity in distributed ml training with dynamic batching. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020.

[18] S. Vargaftik, I. Keslassy, A. Orda, and Y. Ben-Itzhak. Rade: resource-efficient supervised anomaly detection using decision tree-based ensemble methods. *Machine Learning*, 110(10):2835–2866, 2021.

[19] Yelp. Yelp dataset version 6. https://www.kaggle.com/, 2014.

[20] Z.-H. Zhou and J. Feng. Deep forest: Towards an alternative to deep neural networks. In *International Joint Conference on Artificial Intelligence*, 2017.