On the Parallel Computing Framework for Machine Learning and Data Mining

Liran Li

1 Introduction

As a powerful tool for achieving vast speed-ups, parallel computing has been applied to various fields including machine learning (ML), data mining (DM), databases (DB) and scientific computing. However, it is generally agreed that implementation of a parallel program can be tricky, and writing a non-trivial parallel application from scratch requires substantial expertise.

In machine learning and data mining, multiple systems have been developed to make parallel computing more accessible to practitioners. General purpose systems include MapReduce, which exposes a simple interface but requires a restricted dataflow model, and Dryad, which allows a more flexible dataflow model but may require the user to know the network topology. More dedicated systems include TensorFlow for deep learning and XGBoost for boosted tree learning. In this report, we focus on the design of Dryad and XGBoost, plus one direct application of MapReduce to tree learning. We discuss the systems design decisions made by each work. In this short survey, we will see the tradeoff between system generality, and performance for specific tasks.

2 MapReduce and Tree Learning

2.1 Basics of MapReduce

MapReduce is a programming model developed at Google as early as 2004 to meet their data processing needs. It allows users to parallelize their code on a large cluster of commodity machines without experience of parallel computing. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication [1]. There exist commercial implementations of MapReduce, where Hadoop MapReduce is one of the most popular.

MapReduce exposes an exceptional simple interface. Its minimal usage involves what are called mappers and reducers. Data are input in the format of key/value pairs. Each input pair is processed by a mapper into an intermediate key/value pair. All the intermediate pairs are further processed by reducers into the final output key/value pairs. If drawn out as a directed graph with vertices representing computation and data flowing along the edges, MapReduce dataflow corresponds to a short chain, with data in the format of key/value pairs. Users may implement nothing more than just the mapper and reducer, and details including spawning mappers and reducers are left up to MapReduce.

2.2 Applying MapReduce to Tree Learning

Panda, Herbach, Basu & Bayardo 2009 [2] designed the systme PLANET to apply the MapReduce model to ensemble tree learning. Panda et al. note that the scalability and reliability of MapReduce offer an advantage over the then state of the art tree learning algorithms. These algorithms either require data to reside in memory on one single machine or need special parallel computing architectures that are far from deployment-ready. While there are other machine learning tasks, Panda et al. choose the regression tree learning problem for exploration mainly because of the popularity of tree models and their application to sponsored search advertising. This can be seen as one of the first explorations to scale learning algorithms to large datasets on distributed commodity hardware network using MapReduce. Though not explicitly mentioned in [2], it is conceivable that the successful use of MapReduce in ML would have allowed many more ML algorithms to be routinely parallelized and deployed.

2.2.1 Basics of Tree Model Learning

Leaving aside ensembling and boosting methods, the node splitting algorithm is the core of training a single decision/regression tree. A tree is typically grown from a root node to which all training examples are assigned. New nodes are incrementally added by splitting existing nodes. Most of the popular tree methods adopt binary trees, and all the trees from now on are assumed to be binary unless otherwise noted. Let X be the design matrix input to the learning algorithm, and let X_{ij} denote the jth feature of the ith example. The splitting of a node adds two child nodes to this parent, and results in the parent's examples being assigned to the children: the ith example is assigned to the left child if $X_{ij} \leq t$ and to the right child otherwise, for some j and t. Splitting a node can therefore be regarded as a task of finding the suitable feature j and threshold t for this node. Candidates for the threshold are usually the feature values in X. The general procedure is to look at one feature at a time, and find among the candidates thresholds one that gives the highest quality split. Note that splitting a node involves only those examples assigned to this node.

Various quantities have been in use to measure the quality of a split. In general, splitting a node should make the examples in the child nodes purer. In other words, impurity in child nodes should be in some way lower than in their parent. Let y be the vector of labels corresponding to X so that y_i denotes the label of X_i. Panda et al. use variance of example lables y_i within a node as an impurity measure, and pick the feature and threshold that most reduces this impurity to split a node. Other popular impurity measure include entropy and Gini index.

For a decision tree, each leaf is assigned a class label. For a regression tree, each leaf is assigned a value. A tree predicts on an example $\hat{x}$ by passing it down from the root and outputting the label or value at the leaf where the example ends up. The values at the leafs are often called leaf weights, sometime denoted by w.

2.2.2 Design Decisions

PLANET uses a Controller thread that maintains the current tree model and schedules new node-splitting jobs, each job being carried out by a MapReduce call. At each call, multiple nodes (up to 4 in experiments) are split simultaneously. To reduce program complexity, Panda et al. decide to send the entire X to MapReduce. While many examples in X end up not being used, Panda et al. argue that by splitting multiple nodes at the same time, more examples are likely to be used in node splitting. In addition, whenever the examples of a node fit in the main memory, its splitting is performed using the Controller thread instead of calling MapReduce. Panda et al. empirically find this to greatly speed up their system.

Panda et al. then make a tradeoff between accuracy and speed in finding the split threshold. Since the number of examples is large, searching for a split threshold among all feature values can be time consuming. Instead, PLANET pre-computes an approximate equidepth histogram for each feature, and only considers one split threshold from each histogram bucket. This greatly cuts down on the number of candidate thresholds for each feature. The approximate histogram construction had been well researched by the database community.

Panda et al. choose the variance in y as a measure of impurity. This choice of variance allows a simple implementation for mappers. Roughly speaking, a good node split should ensure that the sum of impurities in the children is lower than the impurity in the parent by a non-trivial margin. In order to find the variance of a node, it suffices to know $\sum_{i \in C} y_i$, $\sum_{i \in C} y_i^2$ and $\sum_{i \in C} 1$ where C denotes the index set of examples assigned to this node. The accumulation of the y_i, y_i^2 and count statistics can therefore be performed in parallel. In particular, each mapper receives a key/value pair $< X_i, y_i >$, determines to which node X_i belongs (say node n), and accumulates the statistics for each pre-computed threshold if n is being split. Given the accumulated statistics, the reducers would simply compute the variance of each split and find the one with the greatest reduction.

For categorical variables, Panda et al. uses an algorithm proposed by L. Breiman et al. in 1984. The treatment of categorical features is not discussed any further in this report for little relevance to parallel computing.

PLANET calls MapReduce as a subroutine for the node splitting task. However, the high start up and tear down costs of MapReduce are found to significantly slow down the system. Panda et al. therefore adopt a forward scheduling trick that hides the start up latency by initializing the next MapReduce call while the previous one is still running. To hide the tear down latency, PLANET doesn't wait for the return of MapReduce, but checks whether its output file is ready.

2.2.3 Validation

Panda et al. performed experiments on sponsored search advertising datasets consisting of 6 categorical features varying in cardinality from 2 to 500, 4 numeric features, and 314 million examples [2]. The size of datasets vary from 10 to 50GB. 50 to 400 machines were used for each MapReduce job with each machine configured to have 768MB of RAM (peak usage <200MB). Panda et al. note that adding more machines to MapReduce initially decreases training time linearly, up to the point where various overheads dominate. The authors also compare PLANET against R using the GBM package, which requires training data to reside in RAM. Their plot in [2] shows that PLANET can train on 30 times larger datasets than R, while taking about 4 times longer in training than R. However, the design process of PLANET shows the deficiency of MapReduce in handling iterative tasks. MapReduce typically maintains no memory about its previoius runs, and has to read and write to disks at start up and tear down. This explains why PLANET fails to achieve higher speedup.

3 Dryad

Dryad is a general-purpose parallel system developed at Microsoft. It takes inspiration from systems like MapReduce, expands the flexibility of dataflow graph, and allows fine-grain control of some lower level details [3]. While MapReduce is accessible to a wide class of developers, Dryad targets experienced programmers who know their distributed network to some extend. This for instance allows users to make a tradeoff between data distribution and parallelism. In contrast to the fixed dataflow graph in MapReduce, Dryad allows its users to program any acyclic directed graph as dataflow, through its graph scripting language. A main advantage of allowing arbitrary dataflow is to make more natural transformations from algorithms to programs. Given a dataflow graph, the Dryad runtime maps the vertices to physical machines. Dryad may also dynamically refine the graph to exploit data locality and network topology. Dryad can serve as a framework on top of which more dedicated systems can be built.

3.1 Design Decisions

Dryad allows the user to fine tune some lower level details. The user can decide the channel of data transportation beyond the default of reading and writing a temporary file. Users can also specify the preference to run a vertex on a particular machine.

Dryad allows the use of legacy executable code at vertices. A "process wrapper" is created in this case to pass arguments unmodified to the executables.

The managing thread in Dryad also performs run-time graph refinement. With properly written heuristics, Dryad is able to identify valid refinements of the dataflow graph dynamically so as to preserve the results of computation while taking advantage of the network topology. Consider two racks of computers. The communication bandwidth between racks can be much lower than that among computers on the same rack. If a vertex needs to aggregate data from all computers, the aggregation manager thread can automatically duplicate this vertex so that aggregation happens on each rack first before communication has to be made between racks.

3.2 Validation

Isard et al. run 2 sets of experiments to demonstrate Dryad's performance. The first set is an SQL query to identify the "gravitational lens" effect, with each table taking 10 to 40GBytes of memory. The computation in the second experiment reads terabytes of query logs gathered by the MSN Search service, extracts the query strings, and builds a histogram of query frequency on a cluster of 1800 machines [3]. This is a typical large-scale map-reduce computation.

In the SQL query experiment, near linear speed up in the number of machines is observed. With just one node in the dataflow graph, Dryad still runs significantly faster than SQLServer 2005.

In the second experiment, the authors combine various features from Dryad to validate their design decisions. For instance, They try out different dynamic refinement schemes to take advantage of data locality. As such, without modifying any vertex code, they are able to achieve significant speed up.

4 XGBoost

XGBoost is a state of the art parallel implementation of the boosted tree learning algorithm. It has been the winning solution in several Kaggle competitions. Chen et al. designed this dedicatedd end-to-end system as a tool for practitioners to solve classification and regression tasks using tree boosting.

4.1 Basics of Tree Boosting

Boosting is a special case of ensembling. In a simple ensemble tree method, multiple trees are learned independently, each based on a random subset of examples (rows) from X. This is sometimes called a random forest. The final prediction of a random forest is the mode of predictions from all constituent trees.

The boosting method trains trees so that every newly added tree corrects for the errors made by previous trees. The final prediction is given by the sum of predictions from constituent trees. When training a new tree in the boosting method, the input to this new tree is X and $\tilde{y}$, where $\tilde{y}$ is derived from y and the current model. For instance, $\tilde{y}$ could be the negative of error made by the existing trees. Given the inputs, the new tree is grown by node splitting as in section 2.2.1, with $\tilde{y}$ replacing y.

4.2 Design Decisions

4.2.1 Loss Function

Chen and Guestrin (2016) [4] adopt a second order method for the loss function to train the boosted trees. In particular, Chen et al. employ the loss function

$$\mathcal{L}(\phi^t) = \sum_i l(y_i, \hat{y}_i) + \sum_{k=1,\ldots,t} \left(\gamma T_k + \frac{1}{2}\lambda ||w_k||^2 \right), \tag{1}$$

where ϕ^t is the learned ensemble consisting of t trees, $\hat{y}_i$ is ϕ^t's prediction for x_i, l is an arbitrary differentiable convex function (e.g. $(y_i - \hat{y}_i)^2$), T_k and w_k are the number of leafs and the leaf weights in the kth tree, respectively. When a new tree is added, existing trees are held fixed. Let $f_k(x)$ denote the prediction of the kth tree on example x. Then the following is minimized to find f_{t+1}:

$$\begin{aligned} \mathcal{L}(\phi^{t+1}) - \mathcal{L}(\phi^t) &= \sum_i \Big(l(y_i, \hat{y}_i + f_{t+1}(x_i)) - l(y_i, \hat{y}_i) \Big) + \gamma T_{t+1} + \frac{1}{2}\lambda ||w_{t+1}||^2 && (2) \\ &\approx \sum_i \Big(g_i f_{t+1}(x_i) + \frac{1}{2} h_i f_{t+1}^2(x_i) \Big) + \gamma T_{t+1} + \frac{1}{2}\lambda ||w_{t+1}||^2 && (3) \end{aligned}$$

where the second line involves the second order Taylor polynomial approximation. This approximation allows a broad class of l to work with XGBoost. Users need only to provide code to compute the statistics g_i and h_i. From now on only consider the $(t+1)$th tree and drop the subscript $t+1$ in (3). Let I_j be the index set of instances assigned to leaf j. Then

$$(3) = \sum_{j=1}^{T} \Big[\Big(\sum_{i \in I_j} g_i \Big) w_j + \frac{1}{2} \Big(\sum_{j \in I_j} h_i + \lambda \Big) w_j^2 \Big] + \gamma T. \tag{4}$$

Equation (4) suggests that the optimal leaf weights can be found by minimizing a single variable quadratic function for each w_j. To find the best node split, some impurity measure has to decrease.

Given the easy solution of optimal leaf weights in (4), a natural measure of impurity is as follows: the decrease in the value of (3) is the decrease in node impurity, where the leafs are always assigned the optimal weights.

At this point, it can be seen that the node splitting task in XGBoost turns into a statistics collecting task, just as in PLANET [2]. Therefore, boosted tree learning could also have been solved using MapReduce. However, XGBoost does a linear scan to accumulate the statistics.

4.2.2 Split Candidates

PLANET constructs a histogram for each feature where all examples are treated equally. Similar to PLANET, XGBoost only considers a subset of all feature values as split threshold candidates. However, XGBoost treats h_i as the weight to example i. The authors argue that

$$(3) = \sum_i \frac{1}{2} h_i \Big(f_{t+1}(x_i) - \frac{g_i}{h_i} \Big)^2 + \gamma T_{t+1} + \frac{1}{2}\lambda ||w_{t+1}||^2. \tag{5}$$

Note that the convexity of l guarantees non-negativity of h_i. Intuitively, a larger h_i requires example i to be predicted more accurately, which in turn requires a split candidate closer to i. To find the approximate split threshold for feature j, the authors argue to find the quantile with repect to the "rank" function

$$r_j(z) = \frac{1}{\sum h_i} \sum_{i' : X_{i'j} < z} h_{i'} \in [0, 1]. \tag{6}$$

The rank function is the sum of weights of data points falling below z, normalized by sum of all weights. The authors want to find samples in X such that their jth feature values divide the range of r_j into roughly uniform intervals. The feature values of these examples will be split candidates. To find them, Chen et al. propose the weighted quantile sketch algorithm. The value of this algorithm lies in its ability to handle distributed computation. Consider a large dataset X whose examples (rows) have to be distributed over several machines. On each machine, the algorithm finds the threshold candidates (along with other information, called quantile summary in [4]). Then by merging the quantile summaries from all machines, followed by pruning to reduce the summary down to size, the candidates for X can be found, up to an error tolerance.

4.2.3 Engineering Issues

In many ML application, X is extremely sparse. XGBoost therefore stores X in compressed column storage (CSC) format, with each column sorted by feature values. This allows convenient accumulation of statistics g_i and h_i by a linear scan over feature values. During node splitting, XGBoost only utilizes non-zeros values in X, which the authors find speeds up the execution by about 50 times.

Due to sorted column values, row indices in CSC storage is out of order. Accessing the statistics during accumulation violates data locality. XGBoost therefore pre-fetches the statistics in batches.

In the out-of-core setting, rows in X are divided into blocks, stored in secondary storage devices. To increase effective bandwidth with secondary storage, the authors employ the sharding trick to distribute the blocks into multiple secondary devices, so that multiple accesses can be made simultaneously.

4.3 Validation

Chen et al. performed experiments on datasets coming from 4 distinct fields including particle physics and click logs, each having 10M to 1 billion examples and 20 to 4000 features. Compared to scikit-learn and R (GBM) implementations, XGBoost either runs much faster (by 10x), or predicts much better.

In the distributed setting, XGBoost outperforms production level systems Spark MLLib and H2O by a factor of 10 and 2, respectively. XGBoost is able to scale to cases where data fail to fit in RAM, while the other systems suffer significant slow down. This validates the out-of-core design decisions.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[2] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB-2009)*, 2009.

[3] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 Eurosys Conference*, Lisbon, Portugal, March 2007. Association for Computing Machinery, Inc.

[4] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016.