Scalable Regression Tree Learning on Hadoop using *OpenPlanet*

Wei Yin
Computer Science Department
University of Southern California
Los Angeles CA 90089 USA
weiyin@usc.edu

Yogesh Simmhan
Electrical Engineering Department
University of Southern California
Los Angeles CA 90089 USA
simmhan@usc.edu

Viktor Prasanna
Electrical Engineering Department
University of Southern California
Los Angeles CA 90089 USA
prasanna@usc.edu

ABSTRACT
As scientific and engineering domains attempt to effectively analyze the deluge of data arriving from sensors and instruments, machine learning is becoming a key data mining tool to build prediction models. Regression tree is a popular learning model that combines decision trees and linear regression to forecast numerical target variables based on a set of input features. Map Reduce is well suited for addressing such data intensive learning applications, and a proprietary regression tree algorithm, PLANET, using MapReduce has been proposed earlier. In this paper, we describe an open source implement of this algorithm, *OpenPlanet*, on the Hadoop framework using a hybrid approach. Further, we evaluate the performance of OpenPlanet using realworld datasets from the Smart Power Grid domain to perform energy use forecasting, and propose tuning strategies of Hadoop parameters to improve the performance of the default configuration by 75% for a training dataset of 17 million tuples on a 64-core Hadoop cluster on FutureGrid.

Categories and Subject Descriptors
D.1.3 **[Concurrent Programming]**: Distributed Programming

General Terms
Algorithms, Performance, Experimentation.

Keywords
Machine learning, regression tree, map reduce, Hadoop, energy use forecasting, smart grids, eEngineering

1. INTRODUCTION
The rapid growth of data generation and collection in scientific and engineering disciplines is motivating the need for automated ways of data analysis and mining. Commonly known as the fourth paradigm of science, or eScience [1], this data driven scientific discovery process leverages data available at fine resolutions, accuracy and rates from instruments to not just test scientific hypothesis but also to mine and discover new models. This pushes the envelope on the traditional theoretical, empirical and computational sciences to allow novel breakthroughs.

Several disciplines are in the vanguard of eScience, including genomics [2], astronomy [3] and environmental sciences [4], and they benefit from shared instruments such as genome sequencers, gigapixel telescopes and pervasive sensor deployments that generate terabytes of data in each run or continuously over time for further analysis. Of late, engineering fields such as Smart Power Grids have also started to confront data intensive challenges in cyber physical systems [5,6]. Infrastructure upgrades are leading to monitoring of the electric grid at fine temporal and spatial granularities. This capability is spreading to the edges of the network where power usage information on individual consumers and even their appliances can be monitored by the utility company using Advanced metering infrastructure (AMI), also known as *smart meters*. Smart meters also allow the utility to communicate back with the consumer to send realtime power pricing and system load status signals to change local consumption pattern and reduce demand.

One key application of Smart Grids is to more accurately forecast future power demand by consumers using the improved monitoring ability [7]. This allows the utility to better manage operation of their generation units and energy market purchases based on the expected load profile. It also helps them determine peak load periods in the future that can then be avoided using customized power pricing and consumer incentives. As part of the DOE-sponsored Los Angeles Smart Grid Demonstration Project, we are working with the largest public utility in the US to examine the use of machine learning algorithms for power consumption forecasting. The University of Southern California campus, with over 170 buildings and 45,000 students and staff, serves as a microgrid testbed for these forecasting models [8] which will have to scale to a city of 1.4 million consumers.

Regression tree models [7] have proved suitable for day ahead or week ahead forecasting of numerical target variables such as power consumption for individual campus buildings at 15min intervals using input features such as day of the week, building area, and outside temperature. The model is trained using smart meter data available for each building on campus at 15 minute intervals for the past three years. While the model itself is effective in terms of prediction accuracy, the training time for the model using out-of-the-box machine learning tools such as MATLAB and Weka [9] proves to be punitive when the data is extrapolated to a city-scale. Parallel machine learning libraries such as Apache Mahout [1] do not support the regression tree model. Given the need to update these forecasting models as new data arrives, and to run them in ensemble for feature selection, a scalable regression tree learning application is required.

In this paper, we describe our implementation of *OpenPlanet*, a scalable regression tree learning application that is built on Hadoop[2]. The parallel algorithm we use is based on Google's PLANET [10] that uses the MapReduce programming model; PLANET, however, is closed-source and not directly usable by the broader community. Further, we analyze the impact of various Hadoop tuning parameters on the performance of OpenPlanet and empirically show an improved speedup from the defaults.

[1] Apache Mahout, http://mahout.apache.org
[2] Apache Hadoop, http://hadoop.apache.org

Specifically, our contributions in this paper are as follows:

1. We implement OpenPlanet, an open-source implementation of the PLANET regression tree algorithm on the Hadoop MapReduce framework using a hybrid approach.
2. We evaluate the performance and scalability of OpenPlanet on a cluster using a realworld gigabyte training dataset with over 17 million tuples, and compare it with baseline implementations of Weka and MATLAB.
3. We tune and analyze the impact of parameters such as HDFS block sizes and threshold for in-memory handoff on the performance of OpenPlanet to improve the default performance.

The rest of the paper is organized as follows. In section 2, we present background and related work on regression trees, and describe the PLANET algorithm. In Section 3, we discuss the OpenPlanet architecture and implementation using Hadoop in a cluster environment. In section 4, we analyze the comparative performance and scalability of OpenPlanet with respect to regression tree libraries in Matlab and Weka. In Section 5, we propose and evaluate optimizations of OpenPlanet and Hadoop that improve the baseline performance. We present our conclusions in Section 6.

2. BACKGROUND AND RELATED WORK

2.1 Regression Tree Model

Regression Tree is a type of decision tree learning model [11] in which each non-leaf node is a classifier based on single feature and the leaves end with a linear regression function. The path from the root to a leaf node is a sequence of decision making steps, and the leaf node function is evaluated to provide a numerical prediction value. Therefore, the regression tree model works as a functional mapping from an input vector composed of 'n' predictor features $\vec{x}: \{x_1, x_2, \ldots, x_n\}$ to a continuous prediction of a target value 'y'.

The construction of a regression tree using training datasets containing tuples of input vectors and the target value is often a greedy top-down approach [11]. It chooses an optimal split value from among all attributes within the training dataset to create two dataset partitions, such that the variance of the target values in each partition is less than their variance in the whole dataset. This introduces one node in the decision tree based on that attribute. This is recursively performed on the dataset partition at each node in a breadth first manner, adding one level of nodes to the binary tree at a time (Figure 1). At a certain threshold of variance in the partition, a linear regression function is fitted for the remaining tuples in the partition. Tree pruning can be used to prevent over-fitting, but this is less of a concern for large datasets.

Regression trees have the advantages of being able to make fast predictions once the tree is constructed since it just walks the binary tree. This makes it well suited for making realtime predictions for continuous valued target variables. The regression tree model is also intuitive to understand by domain users since the features conditions are well specified at each inner node, enabling efforts at energy conservation in smart grids by examining the tree.

2.2 Related Work

Several statistical packages and libraries help train and use machine learning models, and specifically the regression tree model. MATLAB provides the `REPTree` function in its statistics toolbox which can train regression tree models and has features for threshold setting and tree pruning. Weka is a popular open source Java machine learning package which supports several algorithms

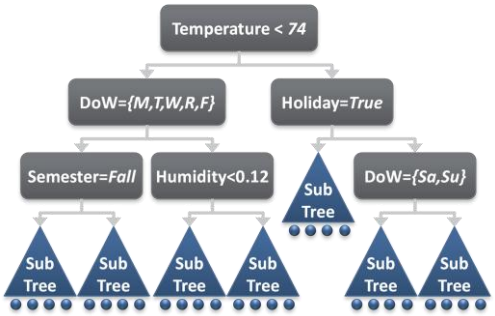


Figure 1. Sample regression tree model. Nodes form decisions. Leaves are a numerical value/regression function.

such as M5P, neural networks and regression tree. Both MATLAB and Weka have limited support for parallel execution of scripts and methods: MATLAB through its Parallel Computing Toolbox[3] and Weka through Weka-Parallel [12]. However, they do not offer a parallel version of the regression tree function, which is itself trained atomically on a single machine and whose training time is constrained by the available physical memory.

Apache Mahout is an ongoing open-source project to implement scalable machine learning libraries using MapReduce. It provides parallel versions of several algorithms for Clustering, Classification, and Regression and Collaborative Filtering, but do not support regression trees. Our work fills this gap. GraphLab [13] proposes a parallel framework based on a graph abstraction to represent some learning algorithms such as belief propagation, Gibbs sampling, and Expectation Maximization. Machine learning algorithms are represented using a data graph where vertices are features and edges are relationships or weights. The graph can then be scheduled on multi-core and distributed platforms. GraphLab provides constructs for data synchronization and consistency. While attempting to be more general than MapReduce, its static graph model does not allow an incremental model such as regression tree to be represented naturally. SECRET [14] is used to efficiently build linear regression trees by reducing the problem into an Expectation Maximization problem for two Gaussian mixtures that is used to decide a split in dataset. However, they focus on single machine training performance that does not scale beyond a certain point.

2.3 PLANET Algorithm

PLANET [10] is a recent regression tree algorithm from Google based on MapReduce. Our work on OpenPlanet is an open source implementation of this algorithm. PLANET is highly relevant to our application, but suffers from several short comings. One, there is no open implementation of PLANET available for the community to use and evaluate. Two, their implementation uses features that go beyond traditional MapReduce and thus does not naively fit into the Hadoop framework. And finally, the Hadoop framework has to be tuned to actually realize the benefits of efficient scaling from PLANET. Here, we briefly describe their algorithm and refer readers to the original paper for further details.

The basic idea of the parallel algorithm is to iteratively build the regression tree in a breadth first manner, one complete level at a time in a distributed environment, until the data partitions are small enough to fit in memory and a "leaf" sub-tree can be constructed locally on a single machine. This limits the scan of the entire dataset (and hence the I/O time complexity) as a linear function of

[3] www.mathworks.com/products/parallel-computing

the tree depth. Building each level starting from the root is accomplished by a MapReduce *ExpandNodes* job for each level, while each leaf sub-tree is built by a separate MapReduce *InMemory* job. As an initialization step before tree induction begins, an equi-depth histogram is constructed for each numerical feature using an *Initialize* MapReduce job and the histogram bucket boundaries serve as a "sample" of candidate point for a potential split. The *ExpandNodes* job evaluates the candidate points for each block of data, the Map generates the sum, sum of squares, and count of the target values for the left and right partitions of each candidate split point, for each node in that level. The Reduce then aggregates these values into the variance calculated for each candidate split point, grouped by the node. The split point that reduces the sum of variance of the two partitions is chosen as the split point for the node. This minimization function is given as:

$$Min\{\,|D_{left}| \times Var(D_{left}) + |D_{right}| \times Var(D_{right})\,\} \quad \ldots \textbf{Eqn. 1}$$

where D_{left} is the left data partition and D_{right} is the right data partition for a node at a certain level. Once the partition size is smaller than a threshold value, the *InMemory* job is used to find further to complete the subtree for that partition. The Mapper task extracts the data partition for the node while the Reducer task finds its local optimal split point.

The PLANET architecture uses a shared "model" file for the tree being built, and a *controller* to iteratively launch MapReduce *ExpandNodes* jobs at each level, use its local results to determine the best split point for each node in a level, and start MapReduce InMemory jobs for the leaf subtrees.

3. *OpenPlanet* ARCHITECTURE

OpenPlanet broadly follows the architectural design of PLANET while using open software, programming framework and hardware infrastructure for the implementation. It has several components working in concert. These components, their logic and implementation are discussed here.

3.1 Controller and Data Structures

The Controller is a Java application that serves as the entry point for OpenPlanet and is responsible for orchestrating the application logic. Given an input training data file, the controller builds a regression tree iteratively, one level at a time, by initiating and coordinating multiple MapReduce jobs in each iteration. OpenPlanet uses Apache Hadoop as the MapReduce programming framework. This means that the training data file passed to the Controller is a file present in the Hadoop Distributed File System (HDFS). For convenience, the Controller runs as a process on the machine hosting the master node of the Hadoop cluster that contains the Job Tracker service used to schedule MapReduce jobs.

The Controller uses several data files and structures for its operations. Besides the input training data file, it maintains a shared model file on HDFS that maintains the current regression tree being built. Three node lists are maintained within the process for each iteration that respectively contain the nodes that need to be expanded, those whose subtrees can be built in memory, and for leaves with tiny data partitions for which an average is recorded.In our implementation, the model file contains a binary serialization of the `TreeModel` Java object that is an in memory representation of the regression tree. Since all read and update operations on the `TreeModel` are by Java applications, sharing it using a binary object serialized file makes it convenient and performant – both for training and, later, prediction. Inner nodes in the `TreeModel` use a

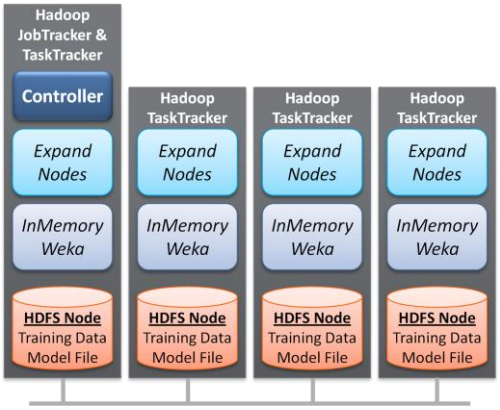


Figure 2. *OpenPlanet* Job and Task layout on Hadoop

simple data structure that represents a binary classification based on a particular feature (e.g. *Temperature* < *74*) that is used to navigate to the left or right sub-tree. Leaf nodes in the `TreeModel` are either a simple leaf with a single average value that represents a prediction, or a Weka `Classifier` object that is a leaf subtree generated from the *InMemory* phase.

3.2 MapReduce Job Types

There are three types of MapReduce Jobs that are launched by the controller: *InitHistogram*, *ExpandNodes* and *InMemoryWeka*.

InitHistogram acts as a pre-processing step to *ExpandNodes*. It operates on the training data present in HDFS, the model file, and the current list of nodes in the *ExpandNodes* phase as its input parameter. Its' Map function scans the training data and extracts only those tuples matching nodes that are currently being expanded and emits them as the value, with the node they belong to as the key. The Reduce function builds an equi-depth histogram from the extracted data for each node, and emits the boundary data points between buckets of the histogram as candidate points for partitioning the data at that node. This heuristic is a form of sampling that reduces the candidates for splitting the node from all data tuples in the partition to just the bucket boundaries of the histogram. We use the open source Colt High-performance Java library[4] for building the histogram in the Reducer. We set the equi-depth histogram to generate 4000 buckets, thus capping the number of candidate split points evaluated at each node. These candidate points are stored as a separate HDFS file. The histogram is used only for numerical features. For categorical features, each possible category value is output as a candidate split point.

ExpandNodes is used to evaluate the local quality of each candidate split point for each block of data, for nodes in the expand phase. It takes the training data file, the candidate split points file from *InitHistogram*, the model file, and the current list of nodes in the *ExpandNodes* phase as its input parameters. The result of *ExpandNodes* provides sufficient information to the controller to pick the global best candidate point. The Map function first identifies the data tuples that belong to each node being expanded by passing them through the current `TreeModel`. For each combination of a node and its candidate split point as key, it emits the count of tuples falling within this bucket ($|D_{left}|$), the sum of their target values (Σy_{D_Left}) and their squares ($\Sigma y_{D\_Left}^2$) (see Eqn. 1). In addition, it also accumulates the count of all tuples falling within this node ($|D_{Total}|$), and the sum of their target (Σy_{D_Total}) and

[4] Colt Project: Open Source Libraries for High Performance Scientific and Technical Computing in Java, acs.lbl.gov/software/colt

Algorithm 1 OpenPlanet Controller

```
function CONTROLLER(trainingFile) -> modelFile
  Initialize InMemoryList[], FinishedList[],
       ExpandList[], treeModel
  Serialize treeModel to modelFile
  ExpandList[0] <- treeModel.Root
  do
    if (ExpandList not empty)
      Launch InitHistogram MapReduce Task
      Launch ExpandNodes MapReduce Task
      Update treeModel from modelFile
    endif
    if (InMemoryList not empty)
      Launch InMemoryWeka MapReduce Task
      Update treeModel from modelFile
    endif
    foreach (node in FinishedList)
      Update node in treeModel with average value
    endfor
    Serialize modelFile <- treeModel
    Clear InMemoryList, FinishedList, ExpandList
    Scan treeModel leaf nodes and place in ExpandList,
         InMemoryList or FinishedList
  while(ExpandList, InMemoryList, FinishedList not empty)
  return modelFile
end function
```

target square ($\Sigma y_{D\_Total}^2$) values and broadcasts this to *all* Reducers with that node as key. The Reduce function uses the "left" and "total" values for each candidate split point for a node to determine the "right" partition values (i.e. *total* minus *left*) and evaluates the minimization function, Equation 1. It then picks the best split point among these for that node and emits it as the output.

InMemoryWeka is a MapReduce job that is used to build a regression subtree in memory for the nodes whose data partitions are smaller than a threshold. The input to this job is the input training data and the current list of nodes in the *InMemoryWeka* phase, and its output is a regression subtree created for each of these nodes. As before, the Map functions scans the entire training data to extract only those data partitions for each input node. The Reduce function passes these tuples to the Weka machine learning Java library and uses it to train a `REPTree` Classifier. The output the Reduce is a Java object serialization of this classifier instance.

3.3 Application Walkthrough

We present a walkthrough of using the OpenPlanet application to build a regression tree to show the interactions between the various components that have been introduced, with pseudocode provided in Algorithm 1.

The Controller is the entry point for OpenPlanet and this Java application is started from commandline. The Controller reads input parameters from a local configuration file that contains the path to the training data file on HDFS, a description of its features, paths to output and log files, the threshold values, and Hadoop runtime information such as the number of Mappers and Reducers. It then initializes the node list data structures, instantiates a new `TreeModel` with an empty root node and serializes it into HDFS as the model file. The Controller then checks the size of the initial data partition, which is the entire training data. This can fall in one of three ranges. If the data is less than a tiny threshold (configurable and defaulted to 20 tuples), it places the root node into the Finished list. Else, if the data is smaller than a memory threshold (configurable, and defaulted to 60,000 tuples), then the root node is placed in the InMemory node list. For all other cases, the root node is placed in the Expand node list. The Controller then performs the following steps iteratively until the Finished, Expand and InMemory node lists are empty.

If Expand node list is not empty, the Controller first schedules an *InitHistogram* MapReduce job by contacting the Hadoop JobManager. It passes a reference to the training data HDFS file and the list of node(s) to expand as input. The Hadoop JobManager then launches this job onto the Hadoop cluster and upon completion, notifies the Controller. The resulting candidate split points is returned in an output file that is passed as input to schedule an *ExpandNodes* MapReduce job through the Hadoop JobManager. Once completed, the Controller is returned the best split point for each node. It then navigates to that node in the `TreeModel` and creates two children based on the feature condition that corresponds to the selected split point (e.g. Left Child: *Temperature < 74* and Right Child: *Temperature >= 74*). The partition size of the child nodes is one of the outputs of the *ExpandNodes* Reducer and this is annotated in the TreeFile by the Controller.

If the InMemory node list is not empty, the Controller schedules an *InMemoryWeka* MapReduce job by contacting the Hadoop JobManager and passes a reference to the training data HDFS file and the list of node(s) to build subtrees in memory. The *InMemoryWeka* task creates a Weka subteee for each node in the list, adds it as a leaf to the `TreeModel` object and serializes it to the model file. Lastly, for each node in the Finished list, the Controller navigates to that node in the `TreeModel` object and sets its child to be a simple leaf value whose prediction value us the average of all target values in that tiny data partition.

The Controller runs one job at a time since they each correspond to one level of the regression tree, and a lower level in the tree depends on its parent level. At the end of each iteration, the updated `TreeModel` object is serialized back into the shared model file on HDFS. The iteration termination condition is reached when all node lists are empty. The `TreeModel` represents the final regression tree and its serialized form in the HDFS model file is returned to the user.

3.4 Distinctions from PLANET

The OpenPlanet design differs from the PLANET in several ways. We make use of existing Java libraries for implementing the equi-depth histogram and building the in memory leaf subtree. In particular, the use of the Weka machine learning library for the latter means that we can build hybrid tree models, as shown in Figure 1, that form a binary regression tree for inner nodes but can be a *REPTree* or any other classifier for the leaf subtree, such as, *M5P*. This gives the flexibility of training the most appropriate classifier for the leaf data partitions that fit in memory while using the scalable regression tree for higher levels in the tree. It also allows us to leverage features such as tree pruning that are available in Weka for the leaf subtrees.

Hadoop does not natively provide the ability to perform a broadcast from the Map stage to the Reduce stage, but this is required for *ExpandNode* MapReduce jobs to send the "total" values for a node ($(|D_{Total}|)$, Σy_{D_Total}, $\Sigma y_{D\_Total}^2$) to all Reducers. We address this by duplicating the "total" values to as many Reducers as present, and using a broadcast key set to the number for each Reducer. Our user defined `MapReduceKey` has an *isBroadcast* flag, and if set to true, overrides the `hashCode` function to return

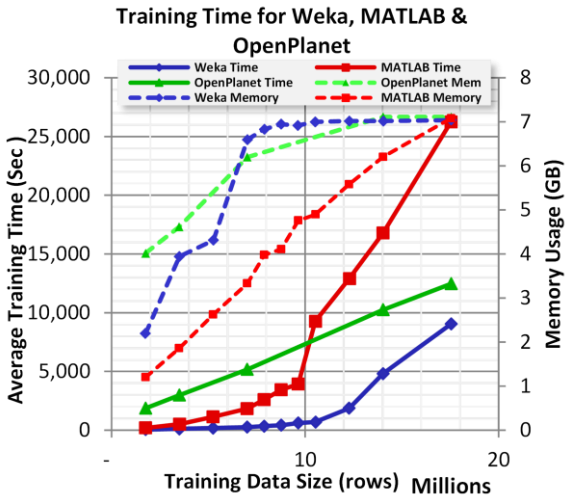


Figure 3. Comparison of Training Time (secs) and Memory usage (GB) for Weka, MATLAB and OpenPlanet running on the 8GB RAM USC workstation as training data size increases.

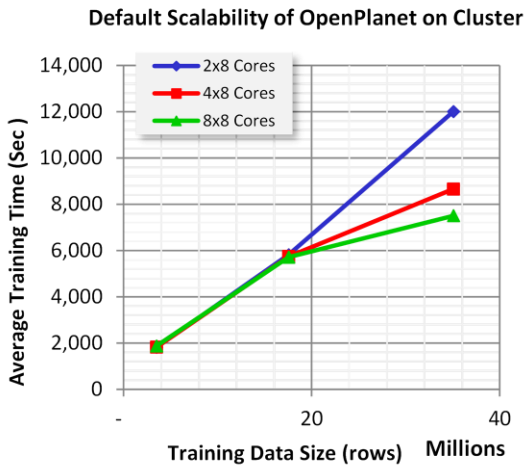


Figure 4. Training time for OpenPlanet when running on different numbers of compute nodes as training data size increases.

the Reducer number encoded in the key which will be used by Hadoop's `HashPartitioner` to send this tuple to that Reducer.

Lastly, we initialize the equi-depth histogram prior to each *ExpandNode* job, instead of just once before tree induction. This strategy re-samples the relevant subset of data at each level and can give more accuracy results.

4. PERFORMANCE EVALUATION

We empirically evaluate OpenPlanet, comparing its performance against popular machine learning libraries that support regression tree as well as measuring its scalability. The baseline comparisons are performed on a standalone workstation at USC which is equipped with a quad-core Intel i5 2.5GHz CPU, with 8GB Memory, and running Windows Server 2008 64bit HPC Edition. The scalability experiments are performed on the Sierra cluster at UCSD that is part of the *FutureGrid* project[5]. Each compute node has an 8-core Intel Xeon 2.5GHz CPU and 32GB memory. The nodes are connected via Gigabit Ethernet.

We use Hadoop v0.20.2 that is installed on the Sierra cluster with a default configuration that uses 64MB HDFS block size and a 2x replication factor. Hadoop jobs are allocated a default of 8 Map and 4 Reduce Slots with 1GB of heap size each. For each run of our experiment, nodes on the cluster are acquired using the Torque batch submission system and Hadoop is deployed on-demand on them using myHadoop [15]. The job submission script copies the input training data file from the local desktop to HDFS, starts the OpenPlanet Java application on the Hadoop JobTracker node with the input parameters, and once completed, copies the trained model file and performance log files back to the local machine.

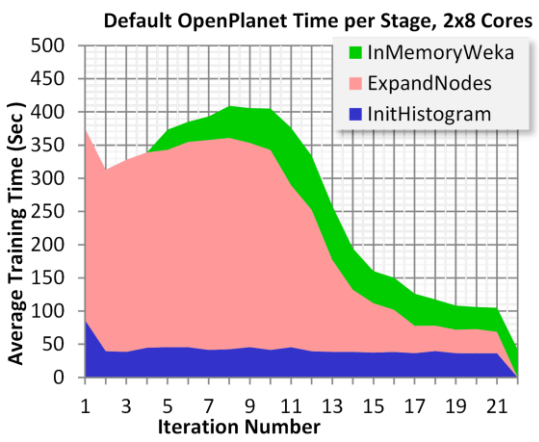
For our experiments, we use two years of data from 24 buildings in the USC campus microgrid as the core training dataset, and use this to extrapolate to a larger synthetic datasets. The core dataset contains 17,544 tuples with nine features each, five numerical and four enumerated, which are used to predict the daily power consumption target feature. The testing is done against a third year of consumption data that is available for these buildings. We extrapolate larger datasets by duplicating the core dataset but introduce a ~1% random perturbation in the numerical features and power consumption value to avoid overfitting or other artifacts. This gives us training datasets of size ranging from 1.7 Million (85MB) to 35 Million (1.7GB) tuples that represent two years of daily data for 2,500 to 50,000 buildings. Unless otherwise noted, all experiments are run and averaged over three runs.

4.1 Comparative Baseline Data Scalability

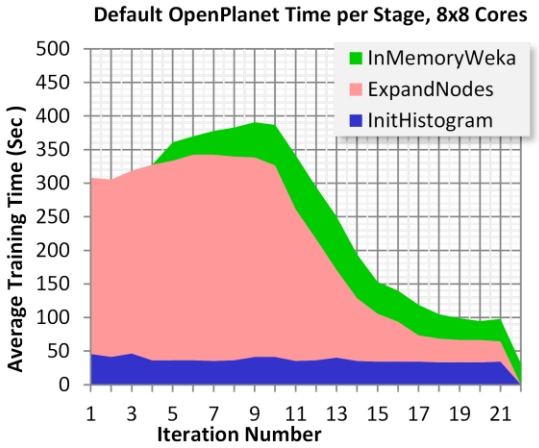
We use the Weka machine learning library and MATLAB statistical application as baselines to compare against OpenPlanet on a single machine. We use the `REPTree` method in Weka v3.6.4 and `classregtree` function in MATLAb v7.11.0.584 (R2010b) to train the model on the quad-core USC workstation. OpenPlanet is run on a pseudo-distributed version of Hadoop on the workstation with an HDFS replication factor. We use a Linux VM, due to Hadoop compatibility issues on Windows, that is allocated all 4 cores and 7.5GB RAM. Besides logging the overall training time for these applications, we also monitor the CPU and memory utilization using *Windows Performance Monitor* and Linux *sysstat*. In these experiments, the generated models were not identical due to the subtle variations in these algorithms. However, the prediction errors of models from all three applications were narrowly clustered within 9.5–10.8% for different data sizes.

Figure 3 shows the average training in seconds (solid lines) for Weka, MATLAB and OpenPlanet on the primary Y Axis as the training data size increases from 1.7M to 17M tuples on the X Axis. It also plots the memory utilization in GB (dotted lines) for one of the runs on the secondary Y Axis. We see that Weka and MATLAB provide better training times than OpenPlanet for smaller data sizes, with Weka proving the better of the two taking 271secs for a 7M tuples (340MB) against 1840secs for MATLAB and 5179secs for OpenPlanet. Until 10M tuples, they show linear increase in time as the data size increases. However, beyond 10M tuples, MATLAB performance starts to degrade and Weka's slope starts to increase, while OpenPlanet continues to exhibit a uniform slope. This causes MATLAB to underperform OpenPlanet beyond 10M tuples, and extrapolating Weka's slope, it would cross beyond OpenPlanet at 20M tuples. This can be explained by observing the memory usage for Weka which hits peak available physical memory at ~10M tuples when the slope changes. This shows that for datasets that fit in memory, Weka and MATLAB are preferred,

[5] Future Grid Portal, http://portal.futuregrid.org

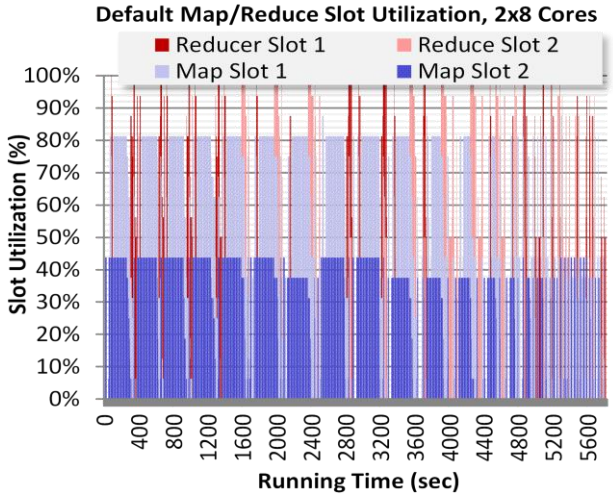


(a) 2x8 Cores

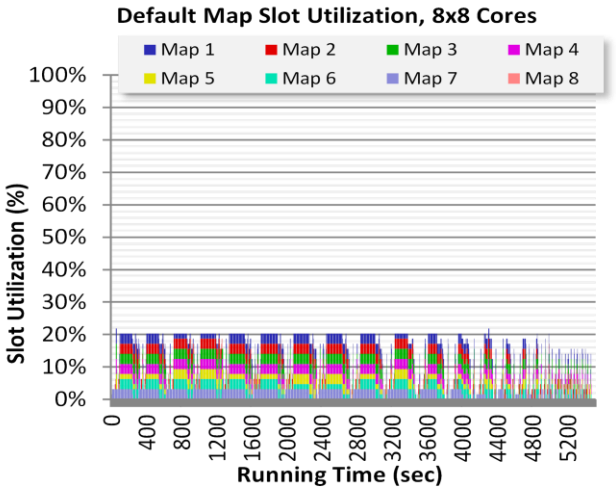


(b) 8x8 cores

Figure 5. Stacked plot of time taken in each MR job type for every iteration of tree building in OpenPlanet on a 17.5M dataset.



(a) Map Reduce Slot Utilization on 2x8 cores



(b) Map Slot Utilization on 8x8 cores

Figure 6. Map Reduce Slot Utilization Percentage on 17.5M dataset

but for larger data, OpenPlanet is more memory efficiently and trades off time against space to scale well even on one machine.

4.2 Scalability with number of Cores

We evaluate the scalability OpenPlanet on a distributed environment, for which it is designed, with increasing training data sizes. It is run on the Sierra cluster for three configurations of 2, 4 and 8 Hadoop nodes with 8 cores each i.e. 2x8=16, 4x8=32 and 8x8=64 cores. Training datasets of 3.5M, 17M and 35M tuples are used. Besides total runtime, we also log the time taken by each MapReduce job type, HDFS I/O metrics and Hadoop slot utilization.

Figure 4 shows the average training time in seconds in the Y Axis for Hadoop running on 2x8, 4x8 and 8x8 cores as the training data size increases along the X Axis. We see a near linear increase in training time as the data sizes increases. For smaller training data (<20M tuples, 850MB), there is no difference in the training times as the number of nodes increases, taking between 5,712 and 5,818 secs. For data sets larger than 20M, having more nodes offers a lower training time but with a poor speedup – we get 1.4x and 1.6x lesser time when using 4x8 and 8x8 nodes as compared to 2x8 nodes for 35M tuples.

It is useful to investigate this under-performance by drilling down into the individual times taken over MapReduce jobs. Figure 5(a) shows a stacked plot showing the time taken by each type of MapReduce job at every iteration of tree building for a single run of OpenPlanet on 2x8 nodes on 17.5M dataset. Figure 5(b) shows the same on 8x8 nodes. The training takes 22 iterations. The total area under each color corresponds to the time spend in that stage.

We see that about 70% of the total time is spent in *ExpandNodes* while *InitHistogram* and *InMemoryWeka* take 15% each for both 2x8 and 8x8 nodes. The time taken by *InitHistogram* remains uniform since it needs to scan the entire dataset to extract data partitions even if there is just one expanding node. *ExpandNodes* takes a bulk of the time initially as the large training data is partitioned into smaller nodes at each iteration, and starts reducing as we reach the lower subtrees where *InMemoryWeka* completes the training. It is only in the 5th iteration that we start seeing *InMemoryWeka* jobs as some data partitions get small enough.

Both 2x8 and 8x8 nodes take similar times in all MapReduce stages and iterations. This can mean that the tasks are I/O bound and the CPU cores are being under-utilized. We verify this by looking at the number of Mapper and Reduce slots that are being

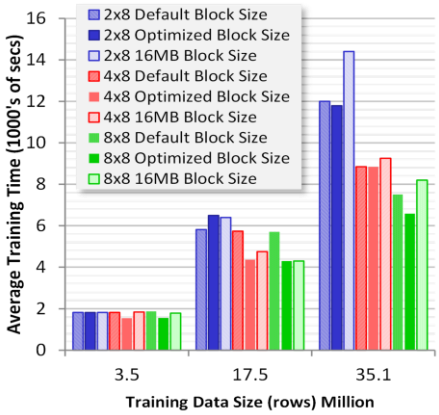


Figure 7. Training time for different HDFS block sizes of 16MB, optimized and 64MB

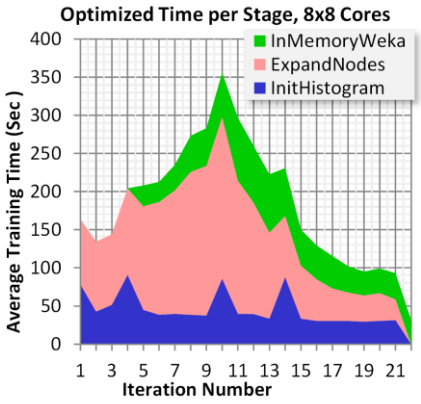


Figure 8. Optimized stacked plot of MR iteration time on 8x8 cores, 17.5M dataset

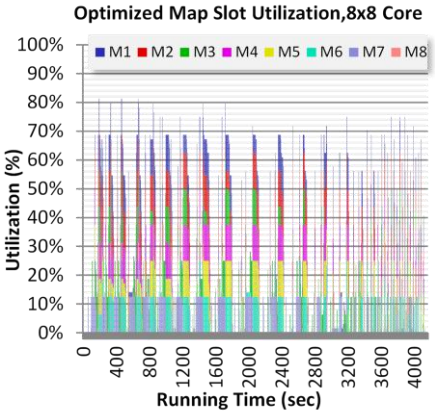


Figure 9. Optimized Map Slot Utilization on 8x8 cores, 17.5M dataset

used in each Hadoop node. Hadoop is configured with 8 slots for the Mappers, corresponding to the number of cores in each node, and 4 for the Reducer. When we plot the % of active Mappers and Reducers across all 2x8 nodes in Figure 6(a) over the duration of the OpenPlanet application, we see that ~80% of the 16 available Mapper slots are mostly active, and a predominant time is spent in Mappers rather than Reducers. The latter is explained by that *ExpandNodes*, where 70% of total time is spent, has more activity in the Mappers while *InMemoryWeka* has more computation in the Reducers, thus getting more utilization in later iterations.

When we compare it against the utilization of the 64 Mapper slots available to the 8x8 node case, which is shown in Figure 6(b), we see that only ~20% of them are used. This shows that a bulk of the Mapper slots do not have sufficient tasks to perform and motivates the need for further tuning of the application to ensure scalability.

5. PERFORMANCE TUNING

Based on the previous analysis, we investigate two optimizations: one, to improve CPU core utilization, and two, to reduce time spent in the *ExpandNodes* stage in favor of time in *InMemoryWeka* stage since the latter is more performant (though less scalable).

5.1 HDFS Data organization for Hadoop

The number of tasks created by Hadoop is a function of the number of available Mapper and Reducer slots and the number of data blocks on HDFS [16]. A Map task operates on one block at a time. So ensuring that there are at least as many blocks of data as the number of Mapper slots will ensure that data parallelism is leveraged. Alternatively, having too many blocks per Map task can lead to increase communication overhead when non-collocated blocks have to be transferred over the network to idle Mappers.

By default, HDFS allocates 64MB block sizes. In our earlier experiments, based on the training data sizes that range from 170MB to 1.7GB, this translates to just 3 – 30 blocks. As a result, when there are more than 30 cores available (4x8 and 8x8), the remaining cores remain idle and we do not get much better performance than the 2x8 node case. Consequently, we tune the HDFS block size in two ways: one, using a static block size of 16MB, and two, by setting the block size based on the training dataset size such that there are as many blocks as the number of Mapper slots. Since we set the number of Mapper slots equal to the number of cores in a node, this means that we have as many blocks of data as the number of cores on which OpenPlanet runs. However, this method has the downside of tuning the HDFS block for a particular run and will not be practical when the HDFS deployment is used persistently and not on-demand as we do.

We repeat the training experiments using both the static (16MB) and optimized (Table 1) block sizes and plot the total runtime for training OpenPlanet on 2x8, 4x8 and 8x8 nodes for three different training data sizes. Figure 7 shows the time in seconds taken to train the model using OpenPlanet on 2x8, 4x8 and 8x8 nodes on the Y axis for different data sizes on the X Axis. The bar graphs show the times for default static 64MB block size (identical to experiment in previous section), optimized static block size of 16MB and optimized dynamic block size as per Table 1. We see that training time when using a dynamic block size is consistently better by up to 25% than the default static block size of 64MB, with the sole exception of training 17M rows on 2x8 nodes where it is marginally higher. We also see that using a smaller static block size of 16MB does not result in similar improvements, especially for larger data sizes. This is explained by the fact that for large datasets, having such a small block size results in a large number of blocks per mapper and the communication overhead starts to data parallelism gains.

The area plot and slot utilization plot for the 8x8 node training 17M tuples using optimized block sizes is shown in Figure 8 and 9. We see that the area under the curve for *ExpandNodes* has reduced and can be attributed to the better slot utilization of ~70% as compared to the previous 20%.

Table 1. Optimized Block Size for different input training data

Data Size / Nodes	**3.5M Tuples/ 170MB**	**17M Tuples/ 840MB**	**35M tuples/ 1.7GB**
2x8	20MB	80MB	128MB
4x8	8MB	32MB	64MB
8x8	4MB	16MB	32MB

5.2 Threshold Condition for *InMemoryWeka*

Our baseline study shows that Weka performs better than OpenPlanet for small data sizes, but does not scale on a single machine as the data size grows. OpenPlanet itself uses Weka for the leaf subtrees by running *InMemoryWeka* for data sizes below a specified threshold. Intuitively, this means that the larger the subtree that can be constructed in Weka using *InMemoryWeka* rather than *ExpandNodes*, the better the likely performance. This is

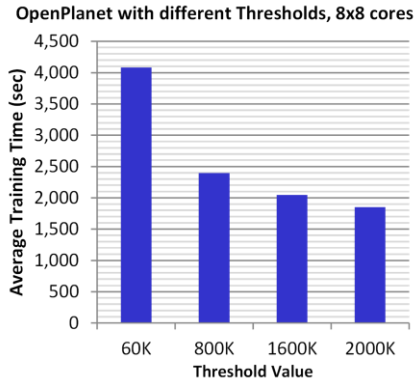


Figure 10. Training Time for different memory thresholds in OpenPlanet using 8x8 cores on 17M dataset

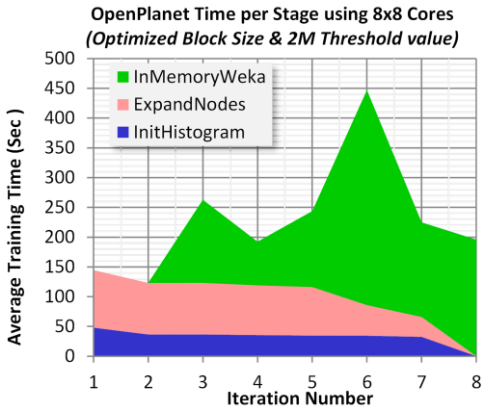


Figure 11. Stacked plot of time in each MR job at every iteration in OpenPlanet using 8x8 cores on 17M dataset

tuned using the memory threshold when a node is added to the InMemory node list.

The original experiments used a default threshold size of 60,000 tuples. Here, we increase these by several orders of magnitude while staying within the available physical memory in the nodes. Figure 10 shows the total training time for 17M tuples on 8x8 nodes when using threshold limits of 60K, 800K, 1.6M and 2M tuples. We use the optimized block sizes in HDFS (Table 1). We see that the training time drops by 40% by increasing the threshold from 60K to 800K tuples. We also see a gradual decrease in the training time as the threshold increases from 800K to 2M. When we look at the area plot for this (Figure 11), we see that the number of iterations reduces from 22 to 8 – bearing out the intuition that we switch from *ExpandNodes* to *InMemoryWeka* at an earlier level in the regression tree construction. We also see that the area taken by *InMemoryWeka* increases from 15% to 57% while that of *ExpandNodes* drops from 70% to 28% of total training time. So this is a more effective tuning parameter than block size. However, we need to use approximations to pick the maximum possible threshold size while staying within available memory. For example, beyond 2M, we run out of memory for *InMemoryWeka* on the machine we run on since each of the 8 Mappers has 1GB heap space allocated to it.

6. CONCLUSION AND FUTURE WORK

We have presented OpenPlanet, an open implementation of a scalable regression tree machine learning algorithm on Hadoop with several performance tuning approaches that reduce the total training time by up to 75%. There are several directions for further optimizing the performance of OpenPlanet as well extending its features. *ExpandNodes* and *InMemoryWeka* jobs are can be run independently to improve utilization of the Mapper and Reducer slots. Currently, we sequence their executions for simplicity. However, their impact of HDFS I/O performance remains to be seen. The iterative nature of OpenPlanet may make it suitable for an iterative MapReduce platform such as Twister [17] and reduce the complexity of the Controller. There may be value in running OpenPlanet on Hadoop deployment on an infrastructure Cloud environment. Besides supporting "OpenPlanet as a Service", this allows on-demand provisioning of Hadoop to fit the block size requirements of a particular OpenPlanet run.

7. ACKNOWLEDGMENT

This material is based upon work supported by the United States Department of Energy under Award Number DEOE0000192 and the National Science Foundation under Award CCF-1048311. The views and opinions of authors expressed herein do not necessarily state or reflect those of the US Government or any agency thereof. We thank FutureGrid for resources provided to run these OpenPlanet experiment, under NSF Award 0910812.

8. REFERENCES

[1] The Fourth Paradigm Data-Intensive Scientific Discovery Tony Hey, Stewart Tansley, and Kristin Tolle, Eds. Microsoft Research, Redmond, WA, 2009.

[2] Schatz MC. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics* 2009;25:1363–9.

[3] Ivezic, Zeljko; LSST Collaboration; et al., LSST: From Science Drivers to Reference Design And Anticipated Data Products, *Bulletin of the AAS*, Vol. 41.

[4] Jie Li; Humphrey, M.; Agarwal, D.; Jackson, K.; van Ingen, C.; Youngryel Ryu, eScience in the cloud: A MODIS satellite data reprojection and reduction pipeline in the Windows Azure platform, In *IPDPS*, 2010.

[5] Ali Ipakchi, Implementing the Smart Grid: Enterprise Information Integration, Technical Report, June 1, 2007

[6] Y. Simmhan, S. Aman, B. Cao, M. Giakkoupis, A. Kumbhare, Q. Zhou, D. Paul, C. Fern, A. Sharma, and V. Prasanna, An informatics approach to demand response optimization in smart grids, Computer Science Department, University of Southern California, Tech. Rep., 2010.

[7] S. Aman, Y. Simmhan, and V. K. Prasanna. Improving Energy Use Forecast for Campus Micro-grids using Indirect Indicators.In *DDDM*, 2011.

[8] Y. Simmhan, V. Prasanna, S. Aman, S. Natarajan, W. Yin, and Q. Zhou. Towards data-driven demand-response optimization in a campus microgrid. In *ACM BuildSys*, 2011.

[9] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. SIGKDD Expl. News. 2009

[10] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. PLANET: massively parallel learning of tree ensembles with MapReduce. In *VLDB*, 2009.

[11] R. O. Duda, P. E. Hart, and D. G. Stork. Pattern Classification. Wiley, New York, second edition, 2001

[12] S. Celis and D. R. Musicant. Weka-parallel: machine learning in parallel. Technical report, Carleton College,CS TR, 2002.

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. In UAI, 2010

[14] Alin Dobra and Johannes Gehrke. 2002. SECRET: a scalable linear regression tree algorithm. In ACM SIGKDD, 2002.

[15] S. Krishnan, M. Tatineni, and C. Baru, myHadoop - Hadoop-on-Demand on Traditional HPC Resources, SDSC Tech. Rep. SDSC-TR-2011-2, 2011.

[16] Tan, Y. S., Tan, J., Chng, E. S., Lee, B.-S., Li, J., Date, S., Chak, H. P., Xiao, X. and Narishige, A., Hadoop framework: impact of data organization on performance. *Software: Practice and Experience*. 2011.

[17] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *HPDC* 2010.