Volume 09 Issue 02 Published, May 19, 2005 ISSN 1535-864X DOI: 10.1535/itj.0902

Intel® Technology Journal

Compute-Intensive, Highly Parallel Applications and Uses

Performance and Scalability Analysis of Tree-Based Models in Large-Scale Data-Mining Problems

More information, including current and past issues of Intel Technology Journal, can be found at:
http://developer.intel.com/technology/itj/index.htm

Performance and Scalability Analysis of Tree-Based Models in Large-Scale Data-Mining Problems

Alexander Borisov, Technology and Manufacturing Group, Intel Corporation
Igor Chikalov, Technology and Manufacturing Group, Intel Corporation
Victor Eruhimov, Corporate Technology Group, Intel Corporation
Eugene Tuv, Technology and Manufacturing Group, Intel Corporation

Index words: machine learning, data mining, decision trees

ABSTRACT

Statistical information processing is needed for many applications to extract patterns and unknown interdependencies between factors. A wide variety of data mining algorithms has been developed over the last decade, but active human intervention is still required to drive an analysis. The intention of expert work is to sequentially clarify models, and to compare models to provide accurate predictions. The productivity of expert work is largely constrained by the amount of time that is needed to compute model updates.

Recent modeling techniques such as classification and regression trees, and ensembles of machine-learning classifiers, incur high computational loads. Building such models in online interactive mode is a challenging task for upcoming platforms.

Tree-based models are applicable to a wide range of problems that include medical expert systems, analysis of manufacturing data, financial analysis, and market prediction. Ensembles of trees are notable for their accuracy. They can handle mixed-type data (consisting of both numerical and categorical data) and missing values. Several commercial packages implement these techniques.

In this paper we consider several data-mining methods based on ensembles of trees. The balance between complexity and accuracy is studied for different parameter sets. We provide an analysis of the computational resources required by the algorithms, and we discuss how they scale for execution on multiprocessor systems with shared memory.

INTRODUCTION

Fast growth and development of digital devices have resulted in a constantly increasing volume of digital data. According to the "How Much Information? 2003" survey [1], the world's total production of information content during 2002 required about 5 million Terabytes to store. More than 90% of this information is stored in electronic form, mostly on hard drives. This enormous amount of information creates a demand for fast and intelligent solutions for data-processing tasks. Many of these tasks can be approached using machine-learning techniques. Machine learning focuses on detecting and recognizing complex patterns in data. Examples are found in biometrics (fingerprint recognition, face recognition, machine vision), network security (intrusion detection), manufacturing (excursion analysis, statistical process control), financial analysis (trend prediction), medical systems (MRI scan analysis [2]), and forensic applications (genetic data analysis). Although these problems belong to different domains, the algorithms solving them have much in common. One of the core concepts commonly used for learning multidimensional patterns from data is a decision tree.

It is difficult to overestimate the influence of decision trees in general and Classification and Regression Trees (CART) [3] in particular on machine and statistical learning. CART has practically all the properties of a universal learner: it is fast, supports both discrete and continuous variables, elegantly handles missing data, and is invariant to monotone transformations of the input variables (and therefore resistant to outliers in input space). Another key advantage of CART is its embedded ability to select important variables during tree construction.

The main limitation of CART is relatively low prediction power. Intensive development of model averaging methods [4-8] over the last decade resulted in a series of very accurate tree-based ensemble techniques. A *tree ensemble* can be understood as a committee, where each member has a vote and the final decision is made based on the majority vote. The two most recent advances in tree ensemble techniques, gradient boosting tree (GBT) [9, 10] and Random Forest (RF) [11], have been proven to be among

the most accurate and versatile state-of-the-art learning machines. GBT serially builds tree ensembles where every new expert construction relies on previously built trees. RF builds trees independent of each other on a randomly selected subset of the training data, and predicts by majority vote (or average in regression).

In the next section we explain the details of the decision tree construction, and various learning algorithms associated with it. We then discuss applications that use decision trees, and we analyze their performance and scalability.

LEARNING WITH DECISION TREES

In supervised machine learning, we are given a dataset with a set of variables or attributes, often called "inputs" or "predictors," and a corresponding target, often called "response" or "output" values. The goal is to build a good model or predictive function that predicts unknown, future target values for given input values. When the response is numeric, the learning problem is called "regression." When the response takes on a discrete set of non-ordered categorical values, the learning problem is called "classification."

Single Tree

Decision trees are one of the most popular universal methods in machine learning/data mining and are commonly used for data exploration and hypothesis generation. CART is a commonly used decision tree algorithm [3]. It uses greedy, top-down recursive partitioning to divide the domain of input variables into sets of rectangular regions. These regions are as homogeneous as possible with respect to the response variable, and they fit a simple model in each region, either by majority vote for classification, or as a constant value for regression. At every step, a decision tree uses exhaustive search, by trying all combinations of variables, and split points to achieve the maximum reduction in impurity. Each split selection requires $O(kn\log n)$ operations, where k is the number of variables and n is the number of training samples.

A single decision tree can be visualized, interpreted, and tuned by an expert. The main limitations of CART are low accuracy and a high contribution to prediction error variance. High variances result from the use of piecewise, constant approximations.

Case Study 1

Figure 1 shows an example of a decision tree constructed to fit a functional dependency depicted in Figure 2a (regression problem with one response and two numeric predictors). The resulting piecewise approximation is shown in Figure 2b.

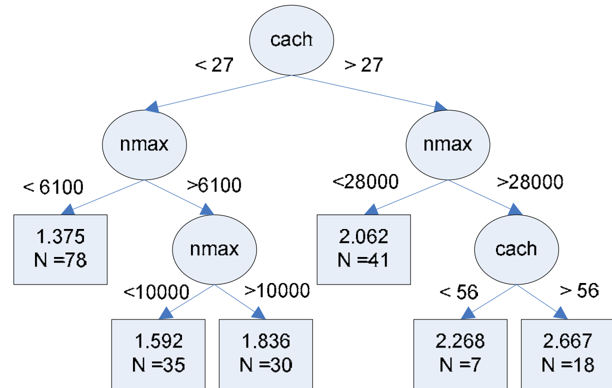

Figure 1: Example of a single regression decision tree

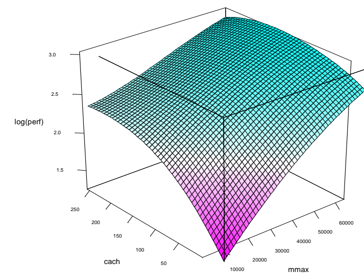

Figure 2(a): Example of a regression tree, original

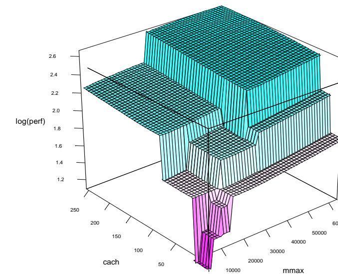

Figure 2(b): Example of a regression tree, CART prediction

Case Study 2

Figure 3 shows a decision tree learned from a car dataset where the response is a country where a car was produced, and predictors are reliability, horse power, mileage, and price. The split (characterized by a variable and its split value) at each node is chosen in order to maximize the number of samples (cars) that correspond to one of the countries. Note that the tree is built so that the presence of a single country in each node (indicated both by the width of the color bar and the text inside the node) grows from

the top of the tree to the bottom. Figure 4 illustrates the properties of the top node of the tree from Figure 3 and the properties of the corresponding split. The top part of the figure shows the table that compares the best split (the first row) to others. For each variable two numbers are reported (left to right): how a split on this variable reduces data impurity in comparison with the best split and how it is similar to the best split in data separation. Similar ("surrogate") splits are used for treating missing values–whenever the value of the variable corresponding to the primary split is missing, the surrogate split is used. The bottom part of Figure 4 shows the box plots of the variable values corresponding to the primary split (Reliability) for each of the response (Country) values. The red horizontal line corresponds to the split value. Note that it cuts off the high values of reliability corresponding to several countries ("Japan" and "Japan/USA").

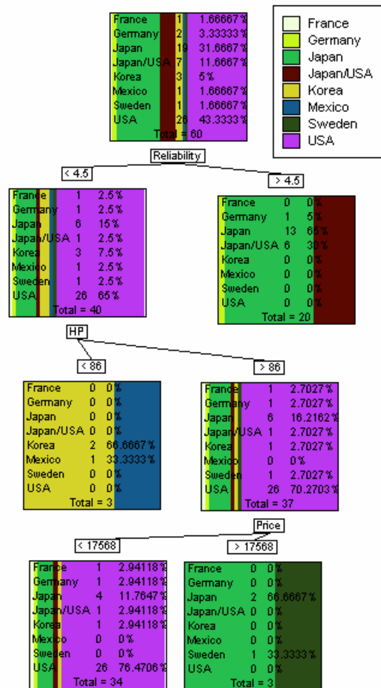

Figure 3: Example of a classification tree

Ensembles of Trees

Ensembles of trees combine outputs from multiple trees and can dramatically improve the performance of the resulting committee. There are two primary approaches to ensemble construction: parallel and serial. A *parallel ensemble* combines independently constructed trees, and therefore targets variance reduction.

In *serial ensembles*, every new constructed tree relies on previously built trees so that the resulting weighted combination of them forms an accurate learning engine. A serial ensemble algorithm is often more complex, but it is targeted to reduce both bias and variance, and usually shows excellent performance.

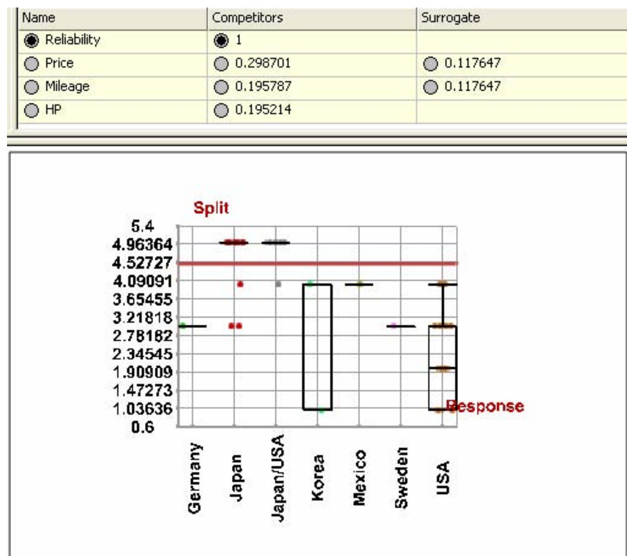

Figure 4: Split competitors and surrogate weights (upper part) and boxplots of split variable (Reliability) vs. response (Country)

APPLICATIONS

Decision tree-based learning is used in a wide range of applications. This list includes experimental data analysis in medicine and physics [3], market and customer analysis [12], manufacturing data exploration [13], and automated spam detection [14]. Single decision trees lack precision in predicting the data but are easier to interpret. For example, Figure 3 shows a tree for a cars dataset. It explains the connection between predictor and response variables using simple and interpretable rules. On the other hand, ensembles of decision trees have much better prediction accuracy, but they lack interpretability. Still, the model can be interpreted with techniques as described in [13].

There are several usage models for tree-based ensembles. For spam detection, the model is learned once and then used to predict the response in real-time with infrequent re-learning. For interactive data analysis, the model is built and re-learned interactively so that an analyst can experiment with parameters, and see the impact of the changes.

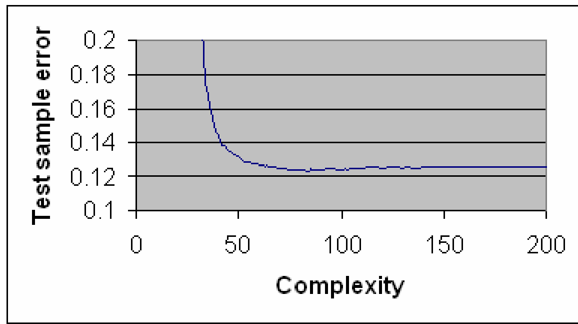

Figure 5: Serial ensemble complexity vs. error

Learning a decision tree ensemble from a large dataset is a computational challenge. For example, GBT ensemble learning on a manufacturing dataset that contains 200,000 samples, 129 predictors (mostly numeric), and binary response takes about eight minutes on a machine with a 3.06 GHz Intel® Xeon™ processor. The resulting ensemble consists of about 70 trees. Figure 5 shows the prediction error of the GBT ensemble depending on its complexity, measured as the number of trees. The optimal size of the ensemble corresponds to the minimum of the prediction error. The time to learn the serial ensemble is roughly proportional to the model complexity. Figure 6 shows the dependence of the learning time on the number of samples in the dataset. Both training data size and model complexity can increase the prediction power of the ensemble but the learning time will also increase.

In the next section, we investigate the computational properties of the parallel ensemble learning algorithm and explain how it can take advantage of the Symmetric Multiprocessor (SMP)-like architecture.

WORKLOAD ANALYSIS

In this section, we describe the requirements for resources used by the algorithm to construct an ensemble of classification trees. We give a description of data structures and methods used to optimize computations. Firstly, *hot operations* are identified that incur the main computational load. Secondly, an interaction with the Arithmetic Logic Unit (ALU), the memory subsystem, and the instruction decoder are described on an example data set. Finally, we present a scheme for parallelizing the algorithm for an SMP system as a function of the number of threads.

® Intel and Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

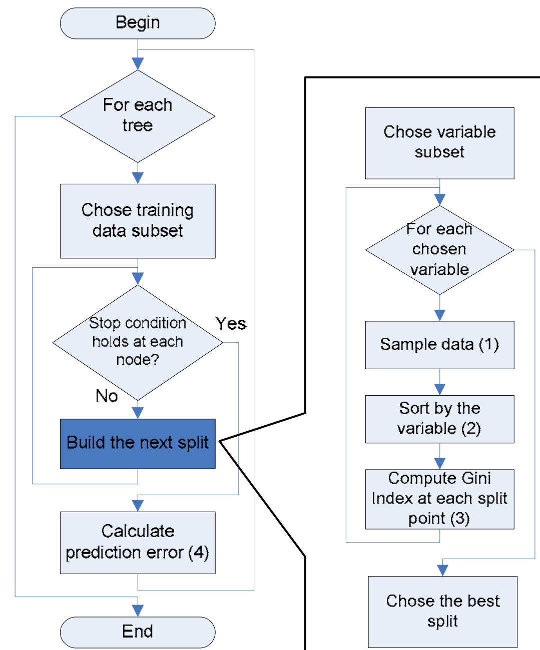

Figure 6: Block scheme of the Random Forest algorithm

Computing Platform Model

A complexity analysis of an algorithm requires a computational model. We consider a simplified model of a modern computer that consists of the following components:

- An ALU that executes logical, integer, and floating-point operations.
- A memory subsystem that consists of the main memory, several caches, and a system bus.
- An instruction decoder that includes a branch predictor.

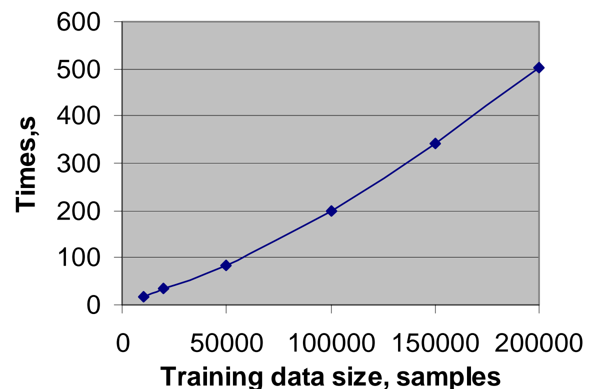

Figure 7: Learning time for a serial ensemble vs. the training data size

In the analysis we investigate the following problems:

- Where is the bottleneck, i.e., which component slows down the overall system performance?
- How many clock ticks are required to execute one instruction?
- What is the traffic to the main memory and each level of cache?
- Does the decoder manage to translate instructions in time, and how do unpredicted branches impair the performance.
- What is the expected performance improvement when the algorithm is executed on an SMP system.

Experimental Results

The training data set consists of 80,000 samples in the feature space that contains 127 numeric and 5 categorical variables, unless stated otherwise. One of the categorical variables is modeled by the Parallel Ensemble Learning (PEL) algorithm.

The following characteristics were measured on a computer with four 1.9 GHz Intel Xeon processors with hyper-threading technology and 3.5 Gb of RAM. The data are presented for a single thread (serial version), unless stated otherwise.

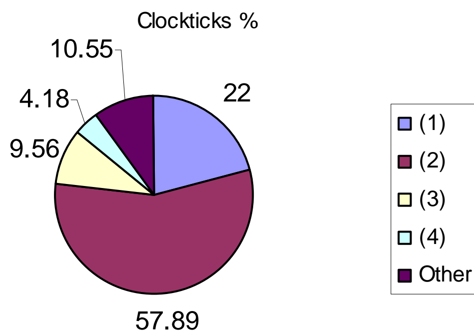

Figure 8: Distribution of execution time by operations

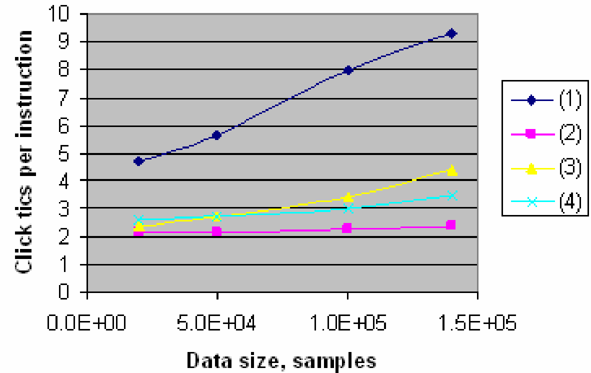

Figure 9: Clock ticks per retired instruction for functions (1)-(4) depending on the number of samples in the training dataset

Hot operations were identified in the optimized code. The main computational complexity is due to the following operations (also see Figure 6):

(1) Sampling a subset of data for calculating the current split. We randomly select both data samples and variables subset (the number of variables to be used in split calculation is a square root of the total number of variables in the dataset).

(2) Sorting training samples on each variable selected for the split. The sorting takes place for each variable independently and thus operates with a relatively small amount of data.

(3) Iterative calculation of data impurity reduction in order to find the best split variable and value. The calculation of the optimal split value is done using exhaustive search that is performed with one pass through the sorted array of values.

(4) Propagate the training samples through the split to the bottom nodes of the tree.

Figures 8 and 9 show the distribution of execution time measured in CPU clock ticks and the mean number of clock ticks per instruction, respectively. One can see that operations (1)-(4) add up to about 90% of the execution time (see the above description of operations (1)-(4)). Note a high CPI value for a sampling operation (1) and the corresponding low clock ticks percentage.

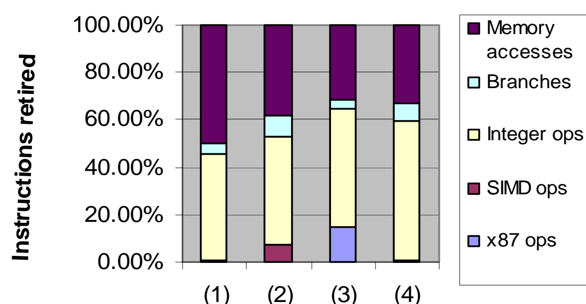

Figure 10: Instruction mix count

One could divide all instructions into four main types: integer and logic operations, floating-point operations, branches, and memory reads/writes. Figure 10 shows the number of retired instructions of each type for (1)-(4). The procedure of sampling is characterized by intensive memory access; sorting requires a large number of comparisons (optimized using SIMD extensions), logical operations and branches, while a search for an optimal split requires intensive FP computations.

Figure 11 illustrates levels of interaction with the memory subsystem. It shows overall memory traffic and its distribution between the caches hierarchy and the main memory (the Xeon processor has 8 KB of L1, 512 KB of L2, and 1 MB of L3 cache). The data explain the high number of clock ticks per retired instructions for (1). The cause is high access rate to the main memory.

Table 1: Characteristics of instructions decoder

	% of de-coded commands	Mispredicted branches per instruc-tions retired	Branch predic-tion rate
(1)	91.697	0.013	78.443
(2)	92.571	0.024	72.964
(3)	95.439	0.002	94.41
(4)	95.322	0.005	94.489

Table 1 characterizes the work of the instruction decoder. It contains the percentage of instructions that did not cause stalls due to the latency of decoding. As the main cause, the number of branches per instructions retired and the BP rate is shown too. As one can see, there is a large number of mispredicted branches in subsampling and sorting procedures that cause stalls of the instruction decoder.

Multithreading

Building a decision tree could be effectively parallelized at the data level. Several algorithms of building tree ensembles (e.g., PEL [11]) assume that each tree is built independently. Then, each sub tree in a decision tree could be built independently, resulting in fine thread granularity. The main thread could make a few first splits, and then assign the building of each sub tree to auxiliary threads. Finally, each of the operations (1)-(3) contains an outer loop on input variables. Iterations are independent, but an aggregation is required at the end (e.g., comparison of split goodness and finding the best one).

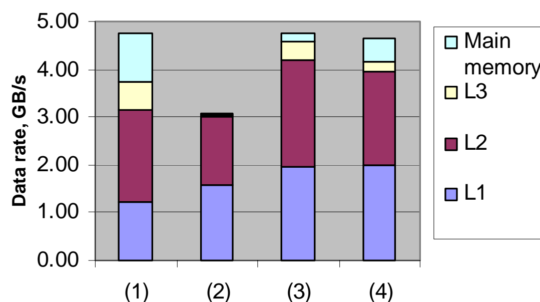

Figure 11: Interaction with memory subsystem

However, access to data structures could be a problem when considering real system architecture. For the cluster-based solution, a subset of training data should be passed to each node, and partitioning is then determined by several first splits. However, the algorithm could be modified to allow effective distributed computations. For SMP systems, the bus becomes a limiting factor with increasing numbers of processors. Figure 12 illustrates this. The vertical axis shows a ratio of execution time per model, related to the execution time of the sequential version. One can see that performance grows near-linearly for up to three threads, and does not grow when the number of threads exceeds six.

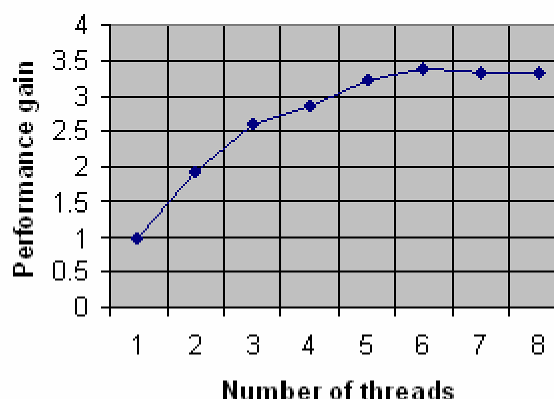

Figure 12: Speedup on the number of threads

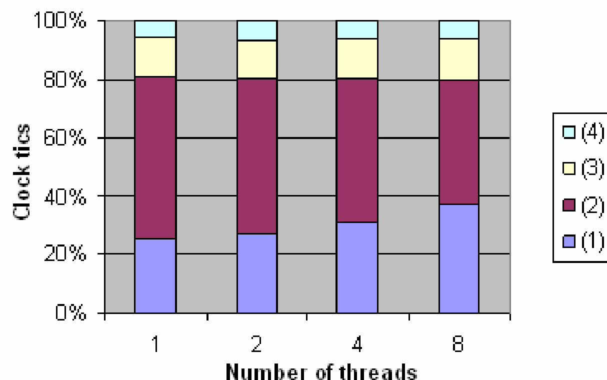

Figure 13: Clock ticks for each function vs. the number of threads

DISCUSSION

The computation of a single tree in a PEL algorithm consists of both intensive data exchange with memory subsystem and CPU load. Function (1) that selects a subset of samples from the original dataset copies a significant amount of data from the source array residing mainly in the main memory to the destination array that is smaller and sits in cache. Figure 11 shows that function (1) causes a significant amount of traffic into main memory. In fact, a significant amount of interaction with cache is caused by a hardware prefetch that reads local data into cache automatically. Still, selecting a subset of samples and variables generates a sparse memory access pattern, and prefetch is not capable of loading all the data we need. The same chart indicates that function (2) works with cache data only. Function (4) divides the training table into right and left according to the currently built split. It generates about a half of the traffic to main memory compared to function (1) but it works more than five times faster. This is because of the difference in memory access patterns: function (4) goes through samples one by one while function (1) jumps between samples and variables in a sparse manner.

Performance gain in Figure 12 shows a sub-linear trend up to three threads. Small gain in Thread 4 is caused by an increased competition between threads for the bus to the main memory. While we do not give a precise interpretation of this effect within this paper, we note that this is in accordance with the data presented in Figure 7. Roughly two-thirds of the execution time in one thread is occupied by the function (2) that is supposed to have excellent scalability. The remaining one-third relates mostly to operations with memory, and this explains the competition that shows up when the number of processors is more than three. The measurements of Figure 11 were taken on a 4-way server with Hyper Threading (HT) enabled. Threads 5-8 share the physical processor with Threads 1-4. The 15-20% gain we get from HT is due to better resource utilization: while one thread waits for the data from memory, executing, for instance, function (1), another thread on the same processor runs calculations of function (2).

Figure 13 supports the hypothesis of poor scalability of function (1): the percentage of time occupied by function (1) grows with the number of threads. If threads have been executed independently from each other, the percentage of clock ticks for each function would have been constant. The data locality issue is crucial here. Figure 8 illustrates the CPI trends for different functions and training samples number. One can see that for a larger dataset it takes more time for function (1) to execute a single instruction, while the CPI rate for function (2) is almost constant.

It is important to note that all data mentioned in the paper have been obtained for the dataset that has sample-wise memory layout. In other words, it is organized in memory so that the data related to one sample occupy a continuous block of memory. (The case of variable-wise representation when the data related to each variable are put into a continuous block of memory lies outside the scope of this paper.) We note, however, that using one representation or another can improve locality and provide an additional speed-up depending on the parameters of the training dataset and PEL.

RESULTS

Learning of tree-based models in the context of large-scale data-mining problems provides many challenges for a computing platform. Often, the more computational power we have the higher prediction power we can get from the model.

The experiments show that an IA-32 system can handle complex ensemble-based learning algorithms very efficiently. The key limiting factor is latency to main memory. A problem-specific data structure can improve data locality and performance gains. For current bus and cache sizes, the algorithm could be effectively parallelized up to four processors. Having several threads per core can provide an additional speed-up.

ACKNOWLEDGMENTS

We thank Roman Belenov and Dmitry Budnikov from Intel Russian Research Center for fruitful discussions.

REFERENCES

[1] Lyman, P. and Hal, R. V., *How Much Information*, 2003 retrieved from http://www.sims.berkeley.edu/how-much-info-2003*.

[2] Spiegelhalter, D. J., Abrams, K., and Myles, J. P., *Bayesian Approaches to Clinical Trials and Health*

*Care Evaluation**, Chichester: John Wiley & Sons, 2004.

[3] Breiman L., Friedman, J.H., Olshen, R.H., and Stone, C.J., *Classification and Regression Trees*, Wadsworth, Belmont, California, 1984.

[4] Breiman, L., Bagging Predictors, *Machine Learning*, vol. 24, no. 2, pp. 123-140, 1996.

[5] Freund, Y. and Schapire, R.E., "Experiments with a New Boosting Algorithm," in *Proceedings of the Thirteenth International Conference on Machine Learning*, pp. 148-156, 1996.

[6] Ho, T. K. Y., "The random subspace method for constructing decision forests," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 832-844, 1998.

[7] Breiman, L., "Using adaptive bagging to debias regressions," *Technical Report 547*, Dept. of Statistics, University of California, Berkeley, 1999.

[8] Dietterich, T.G., "Ensemble methods in machine learning," in *Multiple Classier Systems*, Cagliari, Italy, 2000.

[9] Friedman, J. H., *Stochastic gradient boosting,* 1999. http://www-stat.stanford.edu/~jhf/ftp/stobst.ps*

[10] Friedman, J.H., "Greedy Function Approximation: A Gradient Boosting Machine," *Annals of Statistics*, 29, 5, pp. 1189-1232, 2001.

[11] Breiman, L., "Random forests, random features," *Technical Report*, University of California, Berkeley, 1999.

[12] http://www.salford-systems.com/*

[13] Goodwin, R. et al., "Advancements and Applications of Statistical Learning/Data Mining in Semiconductor Manufacturing," *Intel Technology Journal, Volume* 8, *Issue 4*, 2004.

[14] Hastie T., Tibshrirani R., and Friedman J., *The Elements of Statistical Learning*, Springer, New York, 2001.

AUTHORS' BIOGRAPHIES

Alexander Evgenyevich Borisov was born in Nizhny Novgorod, Russia and received his Masters degree in Mathematics (Lie algebras) at Lobachevsky Nizhny Novgorod State University where he is currently working on a Ph.D. degree in the area of context-free grammars. He currently works at Intel (Nizhny Novgorod) as a software engineer and researcher. His technical interests include artificial intelligence and data mining, especially tree-based classifiers. His e-mail is alexander.borisov at intel.com.

Igor Chikalov is a team leader/research engineer in the Analysis & Control Technology department at Intel. He has been working at the R&D site in Nizhny Novgorod (Russia) since 2000. His research interests include machine learning, test theory, statistical modeling. Igor received his Ph.D. degree from Nizhny Novgorod State University in 2002. His e-mail is igor.chikalov at intel.com.

Victor Eruhimov is a senior research scientist in the Intel Russia Research Center. He is leading a team of researchers investigating the computational properties of algorithms in the area of computer vision and machine learning. He was a senior researcher in the Open Computer Vision project, working on the development of the OpenCV library and research in computer vision. Victor received a Masters degree from the Advanced School of General and Applied Physics in the Institute of Applied Physics, Russian Academy of Sciences, in 1999. His e-mail is victor.eruhimov at intel.com.

Eugene Tuv is a staff research scientist in the Analysis & Control Technology department at Intel. His research interests include supervised and unsupervised non-parametric learning with massive heterogeneous data. He holds postgraduate degrees in Mathematics and Applied Statistics. His e-mail is eugene.tuv at intel.com.

Copyright © Intel Corporation 2005. This publication was downloaded from http://developer.intel.com/.

Legal notices at http://www.intel.com/sites/corporate/tradmarx.htm.

For further information visit:

developer.intel.com/technology/itj/index.htm

Copyright © 2005 Intel Corporation. All rights reserved.
Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.
For a complete listing of trademark information visit: www.intel.com/sites/corporate/tradmarx.htm