Exploiting Thread-Level Parallelism to Build Decision Trees

Karsten Steinhaeuser, Nitesh V. Chawla, Peter M. Kogge

Department of Computer Science & Engineering
University of Notre Dame, Notre Dame, IN 46556, USA
{ksteinha, nchawla, kogge}@cse.nd.edu

Abstract. Classification is an important data mining task, and *decision trees* have emerged as a popular classifier due to their simplicity and relatively low computational complexity. However, as datasets get extremely large, the time required to build a decision tree still becomes intractable. Hence, there is an increasing need for more efficient tree-building algorithms. One approach to this problem involves using a parallel mode of computation. Prior work has successfully used processor-level parallelism to partition the data and computation. We propose to use Cray's Multi-Threaded Architecture (MTA) and extend the idea by employing thread-level parallelism to reduce the execution time of the tree building process. Decision tree building is well-suited for such low-level parallelism as it requires a large number of independent computations. In this paper, we present the analysis and parallel implementation of the ID3 algorithm, along with experimental results.

1 Introduction

Classification is one of the most important data mining tasks [1] as it arises in a large number of real-world problems. It has been studied extensively for many years [6], and a variety of classification algorithms have been presented in literature. They have evolved from simple extensions of statistical models to complex algorithms rooted in both statistics and computer science. Yet over the last two decades, *decision trees* [2][7] – one of the first and most fundamental classification techniques – have emerged as one of the most commonly used methods, and still continue to be a subject of research today.

There are several reasons why decision trees enjoy such widespread popularity. First, they are simple in concept, meaning that it is easy to understand how the clssifier is constructed from the data. Second, decision tree construction is computationally feasible (we present an analysis of the complexity in a later section). Third, a decision tree is easily interpretable by the user and can often be used directly as part of an application. Finally, decision trees have proven to perform well in a diverse range of real-world problems. Therefore, decision trees remain a very important classifier.

Despite having a relatively low computational complexity as compared to other classification algorithms, the time required to build a decision tree is still

quite high (and sometimes intractable) as datasets become extremely large. High-dimensional data with millions of examples or thousands of features is no longer uncommon, particularly for applications in the scientific domain. However, the tree building process lends itself to parallelization as there are (computationally expensive) feature-based calculations that can be performed simultaneously. Hence we want to exploit fine-grain parallelism for scaling tree learning in two dimensions (number of examples and number of features) to reduce the running time of the tree growing algorithm. Prior work exists in the parallelization of decision trees at the processor level (i.e. cluster computing), but modern architectures enable us to also take advantage of thread-level parallelism to capitalize on the inherently parallel nature of the computation.

In this paper, we present a decision tree implementation on Cray's Multi-Threaded Architecture (MTA). We analyzed the ID3 growing algorithm [7] for potential (theoretical) gains from parallelization, implemented it in C++, and performed several experiments on the MTA running in serial and parallel modes.

The remainder of the paper is organized as follows. In Section 2 we present relevant prior work in this area. In Section 3 we discuss the tree building process and analyze its complexity. Section 4 explains the unique architecture of the MTA and its technical specifications. Section 5 contains an overview of our implementation and a detailed example of parallelization. In Section 6 we present our experimental results. Section 7 outlines our future work, and in Section 8 we close with some concluding observations.

2 Related Work

SLIQ was one of the first decision tree algorithms designed specifically for high scalability [5]. It handles both numerical and nominal features, though the authors placed an emphasis on handling the former. Because the sorting of numeric attributes is one of the most expensive operations associated with classifying numeric data, SLIQ integrates a special pre-sorting procedure into the breadth-first tree growth phase. It also uses only a limited number of disk-resident data structures for metadata, while assuming that the bulk of the training data can be stored on disk. However, as datasets grow larger even these seemingly small quantities of metadata turn out to challenge memory capacities, prompting a need for memory-independent data structures and algorithms.

In [8] Shafer et al. present SPRINT, a scalable tree growing algorithm. As mentioned above, at the time memory restrictions were one of the major concerns as datasets grew larger than physical memory, which necessitated the development of efficient algorithms for disk-resident data. Therefore, SPRINT is designed to use data structures with minimal memory restrictions and as such provides fast sequential execution and exhibits good scalability (by the standards of its time). In addition, SPRINT is easily parallelizable at the processor level to further reduce execution time. However, the inter-processor communication (between the 66Mhz workstations with only 16MB of memory!) still becomes a performance bottleneck.

Murthy conducted a very thorough survey of decision tree algorithms [6]. It covers the basic method of tree construction, shortcomings, problems, solutions, variations, extensions, real-world applications and a vision for the future of decision trees.

Srivastava et al. provide a thorough theoretical treatment of the parallelization of decision tree algorithms[9]. As part of this work, the authors design and implement a hybrid approach to tree building parallelization, wherein it uses a combination of the synchronous (all processors work on the same node) and partitioned (processor subsets work on different nodes) tree construction methods. This approach is meant to balance the cost of communication overhead with the cost of load balancing at each step. Experimental results show that the algorithm scales gracefully and exhibits good speedup characteristics as the number of processors is increased.

About the same time, Zaki et al. present a parallel classification algorithm for SMP systems based on SPRINT [10][11]. The algorithm divides the computation for evaluating the best possible split between up to four processors. The algorithm shows acceptable speedup characteristics, but only for relatively small datasets. In addition, one of the great disadvantages of SMP systems is that only a single processor can access memory at any time. Because the tree building process is a very memory-intensive operation, this restriction becomes prohibitive to scalability as the processor-memory gap widens.

Later Jin and Agrawal present SPIES [4], a tree growing algorithm designed for both scalability and parallelizability. It is built on RainForest [3], a generalized framework for scaling decision tree construction. The algorithm has no memory restrictions at all and efficiently partitions the computation to minimize the amount of disk traffic required if the data is too large to fit into memory. However, this algorithm is also parallelized only at the processor level (both in a shared and distributed memory environment).

Although these parallel algorithms – SPIES in particular – exhibit reasonable performance and scalability characteristics, we argue that it is possible to further improve decision tree growing algorithms by using thread-level parallelization. The following sections will discuss the design process of such an algorithm in more detail.

3 Decision Trees

In this section, we discuss the construction of decision trees and present an analysis of the computational complexity of the tree building process.

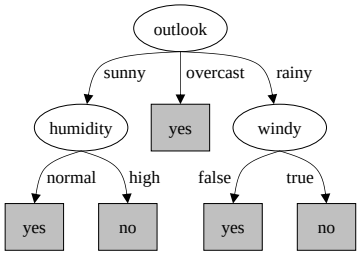
3.1 Tree Construction

Decision trees are generally constructed from *flat-file data*, or a single table in a database where each record (row) represents one example (or instance) and each column corresponds to one attribute (or feature). By convention, we let the number of examples be N and the number of features be M.

A classic example of such a dataset is shown in Figure 1(a), along with the corresponding decision tree in Figure 1(b). Here, the weather on different days is described by four attributes (outlook, temperature, humidity, windy) and the class indicates whether or not a person plays tennis on that particular day.

outlook	temperature	humidity	windy	play tennis?
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

(a) Tennis Training Data



(b) Tennis Decision Tree

Fig. 1. Example of Training Data and Decision Tree

3.2 Complexity Analysis

We shall now discuss the complexity of the ID3 tree building algorithm we chose for our implementation. First note that for this stage, we are only considering data with *nominal features* (like the tennis dataset); that is, no *continuous* or real-valued features are allowed. Including such features would have significant impact on the complexity of the algorithm, and in fact we expect even more of an improvement for this extended version. However, we chose to start with the simple case, and building a decision tree on a nominal dataset has complexity $O(MN)$ [for comparison, it is $O(M^2N)$ with continuous features].

It is also worth noting that in general the "extremely large" datasets fall into one of two categories: they either have a very large number of features and a relatively small number of examples or a very large number of examples but a manageable number of features. Our algorithm is designed to perform well in general, i.e. it should give acceptable performance for both of these extreme cases (and everything inbetween).

In order to take full advantage of the architecture, we want to parallelize in both dimensions. First, let us consider how we can parallelize along M, the number of features. Without going into too much detail of the algorithm, it should suffice to explain that at each node of the tree, all features must be evaluated and the "best" one chosen to split (by some metric, we use *information gain* with ID3). This per-feature evaluation generally involves computation on disjoint sets of data (namely only the column containing that feature) and can therefore be performed in parallel.

In addition, we can parallelize along N, the number of examples. For instance, in several places throughout the code we are required to count the occurences of some value in a column. Using atomic increments to a counter variable, we can parallelize this operation and significantly reduce the execution time by a factor roughly equal to the number of threads available. It is precisely this low-level parallelization, which can be exploited by using multiple threads on the MTA, that is lacking from other approaches in literature.

Depending on the hardware resources available (more on this shortly) we should be able to reduce the complexity with respect to both M and N by a constant factor. If M is sufficiently small, it could even be reduced to 1, resulting a complexity of $O(N)$. Note that these reductions in no way guarantee a reduction in analytical complexity; however, the bottom line for us is execution time, and hence even a reduction by a constant factor will most definitely be of value.

4 The MTA

The Cray MTA[1] is a high-performance computer with a very unique architecture. The particular machine we are using consists of 40 processors clocked at 220Mhz. Four of these processors are mounted together on a board along with 4GB of memory per processor, resulting in a total of 160GB of main memory. This memory is *shared* among all processors, i.e. there is no notion of a per-processor local memory. In addition there is *no data caching*, which means that every load or store must actually go to memory. The processors are connected using Seastar chips, which are on-board high-performance interconnects with built-in processing and routing capability. Still, servicing a memory request takes on the order of 130 clock cycles.

One may now question how this memory latency can be overcome to achieve good performance on such an architecture. The answer lies in the design of the individual processors. More precisely, the MTA has hardware resources for *128 active threads per processor* (also called streams) and is capable of context-switching between threads in a single clock cycle. In addition, each thread can issue up to 8 concurrent memory accesses (for additional information and specs of the MTA, see http://www.cmf.nrl.navy.mil/CCS/help/mta/). Therefore, we can mask the memory overhead by using enough threads to keep a processor

[1] http://www.cray.com/products/programs/mta_2/

occupied while memory references are being serviced. Our goal then is to have not only a few or even a few dozen, but ideally on the order of hundreds or thousands of threads executing at any time to ensure that the processor remains saturated, keeping it busy with "real work" while memory accesses are serviced. And because the memory accesses required by the decision tree construction process have very poor locality, this model of multiple simultaneous accesses enjoys a great advantage over a traditional cache-based system, which incurs frequent cache miss penalties.

5 Implementation

In this section we briefly discuss the ID3 algorithm, our implementation, and a detailed example of a parallelized operation.

5.1 The ID3 Decision Tree Algorithm

We chose to use ID3 because it is very fundamental in the sense that it only performs the most basic operations of decision tree construction: creating a node, testing the termination conditions, selecting the best split, dividing the data, and calling itself recursively. Figure 2 shows the pseudocode of the ID3 decision tree algorithm.

```
ID3(Examples, Attributes, TargetAttribute)

Create RootNode for the tree
if all members of Examples are in the same class C
        then RootNode = single-node tree with label = C
else if Attributes is empty
        then RootNode = single-node tree with label = most common value of Target_attribute in
                                                      Examples;
else
        A = element in Attributes that maximizes InformationGain(Examples, A)
A is decision attribute for RootNode
for each possible value v of A
        add a Branch below RootNode, testing for A = v
        Examples_v = subset of Examples with A = v
        if Examples_v is empty
                then below Branch add Leaf with label = most common value
                                                        of Target_attribute in Examples;
        else
                below Branch add Subtree ID3(Examples_v, Attributes - {A}, TargetAttribute);

return RootNode;
```

Fig. 2. Pseudocode of ID3 Decision Tree Algorithm

5.2 An Example of Parallelism

Looking at the pseudocode (Figure 2), one should notice a call to an external procedure *InformationGain*. The purpose of this function is to determine the best feature to split on. While such a simple function call looks rather harmless, it is in fact the most computationally expensive step of the algorithm. Therefore, we present an in-depth look at this operation and explain how parallelism helps us reduce its execution time.

Computing the information gain for a feature involves computing the entropy of the target attribute for the entire dataset and subtracting the conditional entropies for each possible value of that feature. Therefore, the entropy of a subset is a fundamental calculation to compute information gain, making it the focus of our discussion. The entropy calculation requires a frequency count of the target attribute by feature value, so the straightforward way of computing the entropy for all possible values of a feature works as follows: select all examples with some feature value v, then count the number of occurences of each class within those examples, and compute the entropy for v. This step is repeated for each possible value v of the feature, which can easily be done using a for-loop.

The entropy of a subset can actually be computed more easily by constructing a count matrix, which tallies the class membership of the training examples by feature value. An example of such a count matrix for the feature *outlook* is shown in Figure 3.

	yes	no
sunny	2	3
overcast	4	0
rainy	3	2

Fig. 3. Example of Count Matrix for Outlook

Note that it only takes a single pass over the data to construct this count matrix for any feature (and in fact the count matrices for *all* features!). Once the matrices are constructed, we can easily use them as look-up tables to compute all the entropies and the information gain of each feature.

Now let us bring parallelism back into the picture and show how the information gain calculation benefits from it. Recall that we want to parallelize the computation along both the number of features and the number of attributes. To this end, we created a three-dimensional array to store the count matrices for all features in a single data structure. Instead of looking at each field in the data array sequentially, we can now use a separate thread to inspect each element and increment the corresponding variable in the appropriate count matrix.

We can actually perform one additional optimization step on this process. Notice that with many threads attempting to increment a limited number of counters, atomic add operations must be used. The necessary synchronization can be quite expensive. Fortunately, the MTA provides an instrution called

INT_FETCH_ADD, which not only ensures that the increment is atomic but even performs the increment with a single instruction. Thus, we have taken the information gain calculation from a complex operation involving the copying of data and additional function calls – which is often implemented as described earlier – to a relatively simple tallying process that can be highly optimized.

```
for each attribute A
        Gain[A] = Entropy(Examples)
        for each possible value v of A
                Examples_v = subset of Examples with A = v
                for each example in Examples_v
                        CountMatrix[TargetAttribute]++
                Gain[A] = Gain[A] – Entropy(CountMatrix)
select A such that Gain[A] is maximized
```

(a) Serial

```
for each example E
        for each attribute A
                INT_FETCH_ADD(CountMatrix[A][A.value][TargetAttribute])
for each attribute A
        Gain[A] = Entropy(Examples)
        for each possible value v of A
                Gain[A] = Gain – Entropy(CountMatrix[A][v]);
select A such that Gain[A] is maximized
```

(b) Parallel

Fig. 4. Pseudocode for Information Gain Calculation

To further illustrate the parallelization and optimization process, we have included the pseudocode for the information gain calculations. Figure 4(a) shows the operations for the straightforward serial version. Notice that at each iteration of the first nested loop, the entire training data must be scanned to construct the subset of data corresponding to the current feature value. In some implementations this subset is even copied to a separate data structure for processing, which is an expensive operation.

In contrast, Figure 4(b) shows the operations for the parallelizable version. It uses two separate loops, each of which can be performed in parallel. In the first loop, a separate thread can be used for each combination of attribute A and value v, effectively making the construction of the count matrix a constant-time operation. Of course the number of threads is limited and there is some memory latency, but with hundreds of simultaneous lookups those penalties are easily overcome. Similarly in the second loop, one thread of computation can be used per attribute, each spawning additional threads for each possible value used by

the gain calculations in the nested loop. Given that the number of attributes can be on the order of thousands and the number of examples in the hundreds of thousands of more, using this thread model will achieve the desired goal of issuing enough memory references to make up for the latency and outperform the serial version.

6 Experiments and Results

This section introduces the datasets, explains the experimental setup, presents the results, and discusses them in the context of the analysis provided earlier.

6.1 The Data

We selected two datasets for our experiments; the Forest CoverType dataset from the UCI KDD Archive[2], which contains 581,000 examples of 44 binary features and 7 classes (after removing the continuous features); and a gene dataset called Dorothea[3], which contains only 800 examples, but has 10,000 features and 2 classes. These datasets represent the two extreme situations mentioned earlier, wherein there are either many examples and a relatively small number of features, or few examples with a large number of features.

To test the scalability characteristics along the number of examples N, we constructed 20 datasets of increasing size from the CoverType data. Similarly, to test the scalability characteristics along the number of features M, we used the Dorothea dataset to construct 10 datasets with an increasing number of features (using the first 1,000 features, then 2,000, etc).

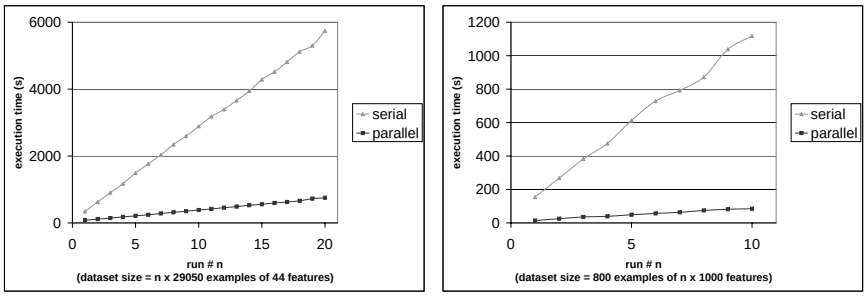
6.2 Experimental Setup

We ran our decision tree code the MTA in serial mode (i.e. forcing it to use a single thread of execution, making it very similar to a regular processor) and in parallel mode (allowing up to 128 active threads of execution) for both the CoverType and the Dorothea dataset. The results are shown in Figures 5(a) and 5(b), respectively. Notice that the size of the dataset (plotted on the x-axis) increases with each run for both datasets, but in a different dimension.

6.3 Discussion of Results

In this section, we discuss two important observations in the result data. Let us begin with the more obvious of the two.

[2] http://kdd.ics.uci.edu/
[3] http://clopinet.com/isabelle/Projects/NIPS2003/



(a) CoverType (b) Dorothea

Fig. 5. Execution Time vs. Training Set Size

Execution Time For both of the CoverType and the Dorothea data, the execution time is significantly lower in parallel execution than in serial execution. This confirms our main hypothesis, namely that we can exploit thread-level parallelism to reduce the execution time of the decision tree building process.

Note that for the CoverType data, the MTA reported using only an average of 11.8 threads for the entire tree building process and 20.8 threads for the Dorothea data. These values are a large part of the explanation why the execution times are only reduced by a factor of about 5 in the case of CoverType, and a factor of 10 for Dorothea. One would think that the execution time is cut by a factor roughly equal to the total number of threads available. However, this is actually not the case because (i) not all of the code can be parallelized, so there will be sections that run entirely sequentially and (ii) even in parallel sections there are not necessarily enough instructions to saturate all threads, especially as computation proceeds further down the tree and the dataset at each node becomes smaller.

Nonetheless, we get a significant performance improvement when using the parallel mode of execution, as illustrated by the raw time savings: we can process the full CoverType dataset in under 13 minutes in parallel mode versus 96 minutes in serial mode; for the Dorothea dataset the contrast is equally noticable with 85 seconds versus 19 minutes.

Scalability The more subtle observation in the results is that the execution time scales more gracefully in parallel mode of execution as compared to the serial mode. It is difficult to see in the graphs, so we have reproduced a condensed version of the data in Figure 6, which shows the ratio of the execution times between the serial and parallel modes for the same dataset sizes. Here we see that as the size of the dataset increases, the execution time for the parallel mode is not only absolutely faster, but it also scales more gracefully as it does for the serial mode.

Forest CoverType	
Fraction of Data	***Ratio of Execution Time Serial / Parallel***
5%	4.30
50%	7.48
100%	7.63

Dorothea	
Fraction of Data	***Ratio of Execution Time Serial / Parallel***
10%	10.78
50%	12.55
100%	13.16

Fig. 6. Serial vs. Parallel Scalability Characteristics

7 Future Work

This work merely lays the foundation for a more extensive study of using thread-level parallelism in conjunction with decision trees. There are still several areas which we would like to explore.

First, it should be possible to further optimize the code by modifying it to allow for more fine-grain parallelsim. In addition, we could introduce an entirely new level of parallelism by making the recursive call of the main function a parallel operation, thereby enabling the MTA to evaluate the data and continue building the tree at multiple nodes simultaneously.

Second, we would like to work with larger real-world datasets. The largest dataset among those used in this study only produced a program footprint of 1.5 GB, which can still fit into main memory of a well-equipped workstation. However, we are garnering datasets that reach well into the gigabytes – such as the protein database data – and at this point the MTA's 160GB of shared memory should have a tremendous advantage. Our goal is to run a large suite of experiments on datasets that push both dimensions to the extreme (examples and features).

Finally, we are looking to extend the functionality of the code to include continuous values, which poses new challenges but also enables the use of parallelism to reduce the complexity. This would make the code compatible with other popular decision tree implementations and allow direct performance and scalability comparisons.

8 Conclusion

In this paper, we have theoretically established and empirically confirmed the hypothesis that exploiting thread-level parallelism can significantly reduce the time of the decision tree building process. By using example datasets from the extreme ends of the spectrum, we showed that our parallel decision tree implementation is consistently faster than the sequential version and scales more gracefully as the size of the input increases (both in number of examples and number of features).

Having successfully produced these important initial results, we are continuing this line of research to produce highly efficient and extremely scalable decision tree building code by taking full advantage of the inherent parallelism in the tree building process and the unique architecture and features of the Cray MTA.

Acknowledgements This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. BNCH3039003. We would sincerely like to thank Cray Inc. for their permission to use the MTA at the SDSC, especially John Feo for his tireless assistance and and expertise in parallel programming on the MTA.

References

1. Agrawal, R., Imielinski, T., Swami, A.: Database Mining: A Performance Perspective. IEEE Transactions on Knowledge and Data Engineering **5** (1993).
2. Breiman, L., Friedman, J. H., Olshen, R. A., Stone, C. J.: Classification and Regression Trees. Wadsworth Adv. Book Prog., Belmont, CA (1984).
3. Gehrke, J., Ramakrishnan, V., Ganti, V.: RainForest – A Framework for Fast Decision Tree Construction of Large Datasets. Journal of Data Mining and Knowledge Discovery **4** (2000) 127–162.
4. Jin, R., Agrawal, G.: Communication and Memory Efficient Parallel Decision Tree Construction. 3^{rd} SIAM Int'l Conference on Data Mining, San Francisco, CA (2003).
5. Mehta, M., Agrawal, R., Rissanen, J.: SLIQ: A Fast Scalable Classifier for Data Mining. 5^{th} Int'l Conference on Extending Database Technology, Avignon, France (1996).
6. Murthy, S. K.: Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey. Journal of Data Mining and Knowledge Discovery **2** (1998) 345–389.
7. Quinlan, J. R.: Induction of Decision Trees. Journal of Machine Learning **1** (1986) 81–106.
8. Shafer, J., Agrawal, R., Mehta, M.: SPRINT: A Scalable Parallel Classifier for Data Mining. 22^{nd} Int'l Conference of Very Large DAtabases, Bombay, India (1996).
9. Srivastava, A., Han, E.-H., Kumar, V., Singh, V.: Parallel Formulations of Decision-Tree Classification Algorithms. Int'l Conference on Parallel Processing, Minneapolis, MN (1998).
10. Zaki, M., Ho, C.-T., Agrawal, R.: Parallel Classification for Data Mining on a SMP Systems IBM Technical Report, Almaden Research Center, San Jose, CA (1998).
11. Zaki, M., Ho, C.-T., Agrawal, R.: Parallel Classification for Data Mining on a Shared-Memory Multiprocessors. 15^{th} Int'l Conference on Data Engineering, Sydney, Australia (1999).