Scalable Learning with Thread-Level Parallelism

Karsten Steinhaeuser, Nitesh V. Chawla
University of Notre Dame
{ksteinha, nchawla}@cse.nd.edu

Abstract

A significant increase in the ability to collect and store diverse information over the past decade has led to an outright data explosion, providing larger and richer datasets than ever before. This proliferation in dataset size is accompanied by the dilemma of successfully analyzing this data to discover patterns of interest. Extreme dataset sizes place unprecedented demands on high-performance computing infrastructures, and a gap has developed between the available real-world datasets and our ability to process them; data volumes are quickly approaching Tera and Petabytes. This rate of increase also defies the subsampling paradigm, as even a subsample of data runs well into Gigabytes. To counter this challenge, we exploit advances in multi-threaded processor technology. We explore massive thread-level parallelism – provided by the Cray MTA-2 – as a platform for scalable data mining. We conjecture that such an architecture is well suited for the application of machine learning to large datasets. To this end, we present a thorough complexity analysis and experimental evaluation of a popular decision tree algorithm implemented using fine-grain parallelism, including a comparison to two more conventional architectures. We use diverse datasets with sizes varying in both dimensions (number of records and attributes). Our results lead us to the conclusion that a massively parallel architecture is an appropriate platform for the implementation of highly scalable learning algorithms.

Introduction

The past decade has witnessed an exceptional increase in the amount of data available to mine. Driving this phenomenon are a sundry of applications including medical informatics, bioinformatics, telecommunications, finance, marketing, etc. We quote from the UC Berkeley Executive Summary on *How Much Information?*:

> "Print, film, magnetic, and optical storage media produced about 5 exabytes of new information in 2002. Ninety-two percent of the new information was stored on magnetic media, mostly hard disks (Lyman & Varian 2003)."

This is a compelling example of the deluge of electronic data and information we are confronted with, requiring not only unprecedented advances in high-performance computing, but also an ability of data mining algorithms to scale.

Today, high-dimensional datasets containing many millions of records and/or thousands of attributes are quite common. Despite steady progress in processor and memory technologies, dataset sizes and algorithmic complexities of advanced methods continue to outpace processing power, making learning from such massive datasets computationally prohibitive. One approach to processing larger datasets is the use of distributed computing, wherein the workload is spread across a number of compute nodes (Chawla *et al.* 2004; Jin & Agrawal 2003). The most common platforms for this kind of computation are symmetric multiprocessor (SMP) clusters. However, many learning algorithms have strong data dependencies, perform random data accesses, or require frequent exchange of information between nodes, so that the cache-miss penalties of hierarchical memory systems and inter-processor communication become performance bottlenecks as the size of the system increases. Therefore not only improvements, but rather radical changes in hardware technology and/or computational model may be required to achieve the desired increase in the scalability of learning algorithms.

One possible solution is the use of a massively parallel architecture with a corresponding programming model for fine-grain parallelism. In addition to spreading the computation across multiple processors, it provides hardware support for multiple threads at each processor, enabling the low-level parallelization of learning algorithms. While many algorithms are subject to the aforementioned data dependencies, at an algorithmic level they also require a large number of independent and therefore parallelizable operations.

Contributions With this work we explore one such architecture, the Cray MTA-2 (Multi-Threaded Architecture), as platform for data-intensive learning algorithms. We present a detailed analysis and implementation of a popular decision trees classifier and show that we are able to leverage multi-threading to increase performance. In addition, we provide a direct comparison to other architectures and demonstrate unprecedented scalability.

Organization The remainder of this paper is organized as follows. The next section gives a detailed introduction to the MTA-2. We then provide a description and analysis of the decision tree algorithm. The other architectures and datasets

used for evaluation are presented, followed by experimental results. Finally, we discuss related work before concluding with a summary of the most important observations.

Multi-Threaded Architecture

In this section we describe the Cray MTA-2, and outline its system specifications and architectural features; see also (Alverson *et al.* 1990). This introduction provides the necessary basis for the ensuing analysis and discussion of the parallel learning algorithms.

The MTA is a shared-memory multiprocessor architecture; the machine used for this work consists of 40 processors and 160 Gigabytes of memory. In contrast to traditional SMPs there is no concept of local memory and there are no data caches. Instead of a memory hierarchy, the MTA relies on fine-grain parallelism to hide memory and synchronization latencies.

The machine is equipped with C, C++, and Fortran compilers – we used C++ for our implementation – and an additional set of compiler directives called *pragmas*, which give the programmer control over code parallelization. Unlike other parallel programming environments the MTA hides many of the synchronization mechanisms from the user, reducing the programming effort. In addition, the MTA also has a tool for code analysis, which provides useful information such as potentially limiting data dependencies and the expected parallelization capacity of each loop.

The MTA Processor

The processor speed of the MTA-2 processors is 220Mhz. Each processor provides hardware support for 128 streams; a stream consists of 32 registers, a status word, and space in the instruction cache. The processor performs a context-switch between streams every clock cycle and executes an instruction from a non-blocking stream. Each stream can have 8 concurrent outstanding memory operations, enabling it to continue executing instructions without blocking.

A schematic diagram of an MTA processor is shown in Figure 1. The top level shows two programs running in parallel. The first level down illustrates their logical flow: program 1 consists of a parallelized loop broken into its iterations and program 2 is strictly a serial task. At the next level, we see that each iteration of the loop is allocated its own stream in hardware, while the serial thread receives a single stream (unused streams are to the right). Finally, the bottom level shows how instructions from the different streams are interleaved for execution. This means that as long as one stream has an instruction ready at every clock cycle, the processor will remain fully utilized.

The Memory System

The MTA-2 has a total memory capacity of 160 Gigabytes. Since there are no caches, all data references must be serviced by main memory directly. Logical addresses are hashed across physical memory to avoid hot spots due to sequential memory references. As Bader et al. pointed out

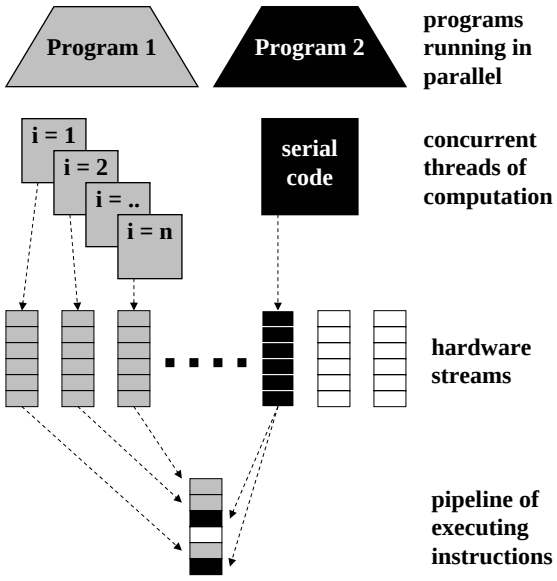


Figure 1: Schematic diagram of an MTA processor during execution.

(Bader, Cong, & Feo 2005), this setup is essentially analogous to an SMP system in which all memory references must access a remote location. The memory interconnection is a partially connected 3-D torus capable of delivering one word per cycle per processor. The latency of a memory reference in the system is approximately 150 clock cycles.

The Programming Model

In order to overcome the memory latency, the MTA relies on fine-grain parallelism in the code. The idea is the following: there must always be a sufficient number of instructions ready to keep the processor busy while memory references are being serviced. Recall that every processor supports 128 streams, and each stream can have as many as 8 concurrent memory references pending without blocking. Therefore, as long as the algorithmic structure of the code allows some streams to execute instructions while others wait for memory references, the processor remains fully utilized and the memory latency is masked by "real work". The MTA mainly accomplishes this task by employing loop-parallelism, that is, performing the independent iterations of a loop simultaneously. In this paper, we demonstrate an example of a learning algorithm that allows for precisely such a parallel structure.

Parallel I/O

Thus far we have ignored one important aspect related to the memory system of the MTA. Throughout our discussion of the unique memory layout and latencies we presumed that when memory is referenced the data is already there, waiting to be delivered to the processor. Like most other systems, however, the MTA must first read the data from disk

and store it in memory using appropriate data structures. Assuming that the data is stored in a comma-separated file, one may naïvely implement this task as follows: read one line of data into a buffer, parse the line, allocate space in the data structure, and store the data elements; repeat these steps until the entire file is read and stored. Although this procedure will work on the MTA, it is extremely inefficient. Given the high memory latency, allocating and storing a single line per access is impractical.

Instead we implemented the procedure as follows: first, determine the total number of bytes in the file (e.g. using the *stat* system call) and allocate a buffer of that size; second, determine the number of instances by counting the number of lines in the file and the number of attributes by parsing the first line, then allocate memory for data structures accordingly; third, read the entire file into the buffer at once; finally, use a loop similar to the one above to parse and store the data line by line, reading from the buffer instead of the actual file. On a sequential architecture the difference between these two procedures is marginal, but on the MTA this final loop can be performed in parallel using a separate stream of execution for each line, because all of the data is now accessible by indexing the buffer.

We performed a comparison of the I/O procedures on a dataset of 1 million records. Using the naïve implementation, reading the data takes 4,346 seconds, or *over 1 hour*. An analysis of the code revealed that most of the time is in fact spent allocating space in the data structure for every line that is read. In contrast, the modified procedure reduces the execution time to 57.3 seconds – a 75x speedup. The implications of this result are two-fold: first, we observed that a simple modification of a procedure can have a profound effect on the MTA performance and second, the speedup due to parallelization is absolutely necessary to attain the desired algorithm scalability.

Decision Trees

In this section we introduce the decision tree classifier, analyze the computational complexity of the algorithm, and explain how the thread-level parallelism provided by the MTA-2 can greatly improve scalability. The corresponding experimental evaluation is provided in the following section.

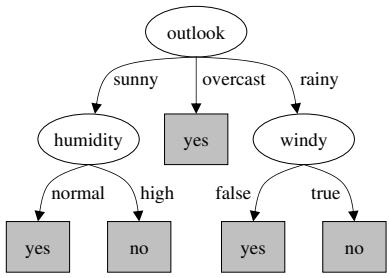
Introduction to Decision Trees

We selected decision trees for our study because they are among the most commonly used classification methods in applied machine learning today. Despite having a relatively low algorithmic complexity compared to other powerful classifiers, the execution time for building a decision tree is still quite high (and sometimes intractable) as datasets become extremely large. We have already shown that the tree construction time can be significantly reduced by exploiting thread-level parallelism (Steinhaeuser & Chawla 2006). The work presented here extends this proof-of-concept by generalizing the procedure to continuous attributes, and more effectively leveraging the architectural features of the MTA-2 to implement parallel structures in the algorithm.

Decision trees are generally constructed from flat-file data, or the equivalent of a single table in a database. Each

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) Sample data describing weather conditions and decisions whether or not to play tennis that day.



(b) Decision tree constructed from the weather data (above).

Figure 2: Example of flat-file data and the corresponding decision tree.

row in the table is called an *instance* and corresponds to a single data item or observation; each column contains one *attribute* describing the data. A prototypical example of such a dataset and the resulting decision tree are shown in Figures 2(a) and 2(b), respectively.

The procedure for constructing a decision tree is an intuitive recursive process. Starting with all training instances at the root node, the data is evaluated to determine the best attribute to split on – the notion of "best" in this context will be discussed shortly. Once the optimal attribute has been identified, a branch is created for each possible value. The dataset is then divided into disjoint subsets of instances according to the values of the chosen attribute, thereby forming new nodes in a tree structure. These steps are repeated for every node at subsequent levels of the tree until a termination condition is met. At this point a leaf node is created and it is given a class value. The resulting tree can then be used to classify unlabeled test examples by traversing the tree according to the attribute values and assigning it the class at the corresponding leaf.

Algorithm Details

For this work we implemented the ID3 decision tree algorithm as described in (Mitchell 1997), including the extension to handle continuous attributes by evaluating all possi-

ble binary splits. The procedure is identical to that implemented by the popular C4.5 decision tree algorithm, but we excluded the pruning features as our primary interest is in the tree growth phase.

We want to construct a tree that is compact while accurately representing the training data. The major task underlying decision tree construction is the selection of a splitting attribute at a given node. Our ultimate goal is to produce nodes containing only instances of a single class, also called *pure nodes*, so we can create a corresponding leaf. Intuitively we should therefore select the attribute that best separates the training instances according to their class, since this will more quickly lead to pure nodes. This notion is quantified by the information gain (Gain) metric. Gain measures the reduction in class entropy of all training instances at the node with respect to the entropy conditioned on an attribute.

Formally, the Gain of attribute A on dataset S is defined as

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

where S_v is the set of examples with value v for attribute A and the entropy of dataset S over the c possible class values is computed as

$$Entropy(S) \equiv \sum_{i=1}^{c} -p_i \, log_2 \, p_i$$

where p_i is the probability of belonging to class i. The attribute with the highest Gain is selected to split on as it best separates the instances by class.

It is straightforward to compute Gain for nominal attributes. All we need is a count matrix of the class distribution by attribute, i.e. a class frequency count for each possible attribute value.

For continuous attributes, the process is more complicated. We need to find the point which best separates the data using a less-than/greater-than relationship. To evaluate a continuous attribute, we first sort the instances at the node according to that attribute. We then consider every possible binary split, i.e. every point where the attribute value changes, and compute the Gain for each. The point at which the maximum Gain occurs is the best possible split, and it is reported as the information gain for that attribute.

The splitting stops when all instances at a node belong to the same class, but in practice this approach is sensitive to noise and results in overfitting. Therefore, we use an additional stopping criterion called *minimum support*, which requires a specified minimum number of instances to be at any node to consider a split. If less instances are available, the data is not split any further and a leaf with the most frequent class among the instances present is created instead.

Analysis Pseudocode for the procedure described above is shown in Algorithm 1. Parallelized loops are indicated in **bold**. We analyze the computational complexity of this algorithm: let the number of training instances be denoted by n, the number of attributes by m, and the number of possible split points by s. For nominal attributes the procedure is only dependent on m and n: $O(mn) + O(m) = O(mn)$. For continuous attributes, the procedure is dependent on m, n, and s: $O(m * (n \; log \; n + sn)) + O(m)$. The number of split points s is bounded by the number of instances n, so we can re-write the complexity in terms of m and n: $O(m*(n \, log \, n+n^2))+O(m) = O(mn^2)$. These complexities are the costs *per level* because the procedure as shown is called recursively on each node. Since the depth of the tree is limited by the number of attributes at $log \, m$, the total costs are $O(nm \, log \, m)$ and $O(n^2 m \, log \, m)$ for nominal and continuous attributes, respectively. The evaluation of continuous attributes dominates the overall complexity.

Algorithm 1 Evaluation of attributes in decision tree construction

Input: Dataset S

```
1:  for each attribute A_i in S, i = 1...m do
2:    if A_i is nominal then
3:      for each training instance x_j, j = 1...n do
4:        compute class distribution (count matrix)
5:      end for
6:      compute entropy E(S)
7:    else if A_i is continuous then
8:      sort n training instances
9:      for s possible split points do
10:       for each training instance x_j, j = 1...n do
11:         compute class distribution (count matrix)
12:       end for
13:       compute entropy E(S)
14:     end for
15:   end if
16: end for
```

The crucial observation for parallelizing decision tree construction is the independence between attributes. The evaluation of one attribute (nominal or continuous) does not depend on any other, so they can all be evaluated simultaneously. We examine the necessary steps more closely. For each nominal attribute, we construct the count matrix and compute the conditional class distribution. This can be performed very efficiently using a parallel loop over all training instances, which performs atomic increments of counters for each class. Based on this distribution, a parallel loop computes the conditional entropy for each attribute. For a more detailed treatment of nominal attribute evaluation see (Steinhaeuser & Chawla 2006).

Evaluating continuous attributes is more difficult, but also amenable to parallelization. For each continuous attribute, we first sort the instances according to that attribute using a parallel sorting routine. We determine all candidate split points with a single pass over the instances. Finally, we evaluate all points simultaneously by computing the conditional entropy for each split in parallel and report the one with the highest Gain. Note that all loops run over a fixed number of iterations and there are no data dependencies in the algorithm, so we could potentially parallelize every loop. However, the loop on line 10 inside the split point evaluation for continuous features is not parallelized for practical reasons, as shown in the pseudocode. To understand the reason be-

hind this design decision, we must explain one of the nuances of multi-threaded programming.

Nested Parallelism The problem with the loop on line 10 is *nested parallelism*, a construct that can be both a blessing and a curse for the programmer. Note that the loops on lines 1 and 9 are parallelized, resulting in a two-level nesting of loops. Such parallelism is acceptable and even encouraged as long as sufficient streams are available to perform all iterations of the nested construct in parallel. This is useful when performing simple operations like constructing a count matrix, but the opportunity for great speedup turns into an even greater problem when insufficient streams are available to perform the operations in the innermost loop. In this case, some initial number of iterations receive all the streams needed, but when resources become scarce a new request is only allocated a single stream. These later iterations of the outer loop then proceed using this one stream because the resources are not re-allocated once the earlier iterations finish executing, thereby becoming a severe bottleneck.

For decision tree construction with continuous attributes, we observe that the two loops over m and s create enough parallelism for any reasonable dataset size to keep the processor saturated; hence the innermost loop over n should be executed in serial to avoid the problem of excessive nested parallelism. An experimental evaluation of this phenomenon confirmed our assessment, as the execution time using three-level nested parallelism was up five times longer than using only two levels. Understanding the expected program behavior is critical to parallelizing the code for optimal performance.

Evaluation

To evaluate the parallelization of decision tree construction, we compare the serial performance on the MTA-2 using a single thread of execution to the parallel performance. We also include the performance on two compute servers for reference. The algorithm is evaluated on two diverse datasets.

Architectural Comparison

We briefly describe the specifications of the other two architectures included in the comparison. Both of the systems have more traditional configurations, consisting of multiple processors with a memory hierarchy of local caches and a large shared main memory. There are two primary differences between these systems and the MTA. First, the clock speed of the MTA-2 processor is approximately 15 times slower than that of the conventional architectures, and due to the round-robin scheduling of its 128 threads the number of instructions issued per unit time can be much lower. Second, the presence of a cache hierarchy can drastically affect data access times, decreasing them for sequential or repetitive operations but increasing them for random access.

Hewlett-Packard HP DL380 G4 – The first system is an HP compute and storage server, model DL380 G4, containing two Dual-Core 3.6Ghz Intel Xeon processors. The memory hierarchy consists of a 1MB local cache at each processor, and an 8GB main memory shared among both processors. The machine is running RedHat Linux kernel 2.6.9 as an operating system. For the remainder of this paper, we will refer to this system as *HP*.

Sun Microsystems SunFire V440 – The second system is a Sun compute server, model SunFire V440, with four 1.3Ghz UltraSparc-III processors. The memory hierarchy consists of a 1MB local cache at each processor, and a 16GB main memory shared among all processors. The machine is running Solaris SunOS 5.10 as an operating system. For the remainder of this paper, we will refer to this system as *Sun*.

Datasets

We evaluate our implementation on one synthetic dataset, which gives us control over the characteristics of the data, and a real-world dataset from the scientific domain. The datasets are summarized in Table 1.

Table 1: Summary of datasets.

Dataset	Instances	Attributes
Synthetic	1,000 - 50,000,000	9
Thrombin	800	10,000

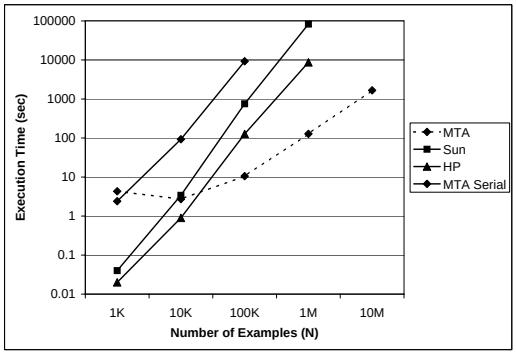
The synthetic dataset was created using the QUEST data generator (Agrawal, Imielinski, & Swami 1993), which has been used extensively for the evaluation of parallel decision tree implementations (Gehrke, Ramakrishnan, & Ganti 2000; Jin & Agrawal 2003). There are different functions (numbered 1-10) for assigning the class values. In accordance with previous work, we use a perturbation factor of 0.05 and function 1 for all experiments.

The second dataset, Thrombin, was specifically chosen for its contrasting characteristics. It is a chemical interaction dataset included in the 2003 NIPS Feature Selection Challenge and consists of 800 instances with 10,000 binary attributes and two classes describing the ability of different compounds to bind to the blood-clotting protein thrombin. With these datasets we emphasize the need to explore scalability characteristics in both data dimensions, i.e. number of instances and number of attributes.

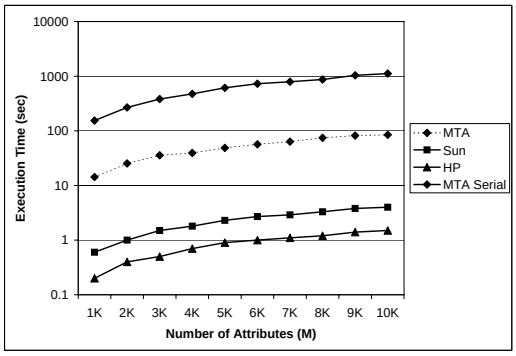
Experimental Results

For all results reported here, we used a minimum support of 1% of the total dataset size. Figure 3(a) shows the execution time for decision tree construction as a function of training set size. First and foremost, note that there is a significant improvement of the parallel over the serial execution on the MTA. With the smallest set the MTA is slower due to insufficient parallelism – of course we would not use the MTA to build a decision tree from only 1,000 instances. But at 10,000 instances there is a cross-over point where all three architectures provide approximately equal performance, and for all larger datasets the MTA is clearly superior, outperforming the HP by two and the Sun by three orders of magnitude for 1 million instances. The MTA also shows a slower rate of growth as compared to the three serial execution models as the margin widens with increasing dataset size. Note that we do not have data points for the

HP and Sun systems at 10 million instances because the execution time became prohibitive. This limitation lends additional support to the massively parallel architecture as a suitable platform for scalable decision tree construction.



(a) Synthetic Dataset



(b) Thrombin Dataset

Figure 3: Experimental results for decision tree construction. Figures (a) and (b) show the scalability with respect to the number of instances and attributes, respectively.

Figure 3(b) shows the execution time for decision tree construction as a function of the number of attributes. The results here are two-fold. Once again we find that parallel execution is at least an order of magnitude faster than the serial version, indicating that parallelization is indeed effective. However, we also observe that the MTA performs worse than the other two architectures. We attribute this disparity to the fact that the Thrombin dataset only has 800 examples, rendering the two-level nested loops much less effective than they are for the datasets with many thousand instances (especially after several splits). Moreover, the Thrombin consists exclusively of binary attributes, which further trivializes the classifier construction and diminishes the advantages of parallel execution.

Ensembles and Data Partitioning

An *ensemble* is a collection of classifiers built on subsets of the training data. Members of the ensemble classify test instances independently, then combine their decisions using a voting scheme. The schematic of a generic ensemble is shown in Figure 4. While it is often desirable to construct a single model from all of the training data, it has been shown that ensemble approaches can improve classification accuracy over the use of a single classifier (Chawla, Eschrich, & Hall 2001) (although the interpretability of the classifier is usually lost).

A different and often more compelling reason to use an ensemble of trees (also called *forest*) is to reduce the demands on the system and circumvent resource constraints. For example, in some applications the datasets are too large to fit into memory, prohibiting the learning of a single classifier on all of the data at once. In this case, several classifiers may be learned on subsets of the data and combined using an ensemble method (Chawla *et al.* 2004). Training classifiers on partitions of data has the additional benefit that the execution time to produce n members on $1/n$-th of the data is generally lower than building a single large model. We examine the effects of parallel execution on ensemble methods and data partitioning, and compare our work on the MTA-2 to other high-performance architectures.

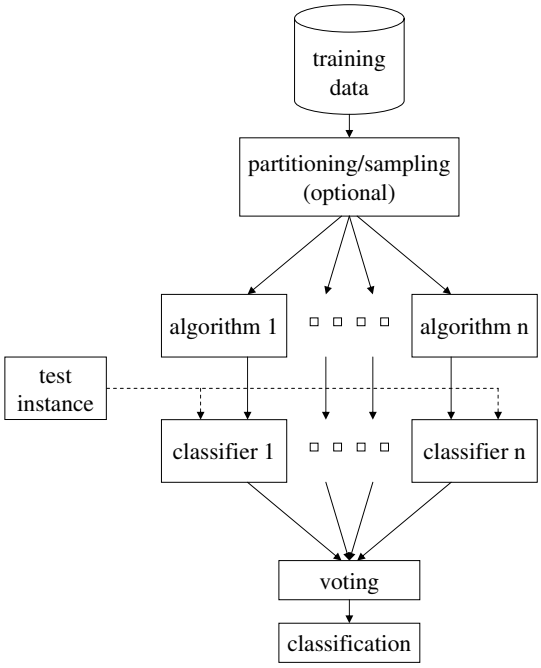


Figure 4: Example of a generic ensemble framework for classification.

Chawla et al. provide an evaluation of ensembles of decision trees on different high-performance architectures (Chawla *et al.* 2003). We obtained the Protein dataset with 209,529 instances and 315 continuous attributes to compare our implementation to the results reported on a 24-node (900Mhz) Beowulf cluster. Table 2 shows the execution times on the two architectures for the full dataset and four disjoint partitions.

Table 2: Comparison of execution times per partition on high-performance architectures.

partitions	Beowulf (sec)	MTA-2 (sec)
1	9,094	7,798
4	964	2,034

The MTA-2 offers slightly better performance for building a single tree from the full dataset (one partition). However, the Beowulf cluster can construct a classifier from $1/4$-th of the data in approximately half of the time required by the MTA-2. This result is not surprising since the margin of improvement for the MTA over the other architectures was lower for smaller datasets. This implies that the benefit from parallelization will be reduced when learning on partitions of data.

Chawla et al. also report results on the "ASCI Red" parallel supercomputer, and on the SGI IRIX64 with 32 Gigabytes of main memory, both housed at Sandia National Labs. We were unable to obtain the large PDB dataset (3,619,461 instances) used by the authors to replicate experiments on the MTA-2, but we can extrapolate reasonable performance estimates for comparison as follows.

Table 3: Execution times on the Protein dataset with partitions of increasing size.

partition size	execution time (sec)
1/4	2,034
1/2	3,967
3/4	5,719
1	7,798

The authors constructed decision trees on $1/8$-th partitions of the PDB data (about 450K instances) using the ASCI Red, with an execution time of approximately eight hours per partition. To extrapolate as accurately as possible, we first obtain scalability data on several smaller partitions of the Protein data. The results are shown in Table 3. We observe a growth in execution time that is slightly less than linear with respect to the partition size. Given that $1/8$-th partition of the PDF is about 2.3 times larger than the full Protein dataset, a reasonable estimate for the MTA would be 18,000 seconds, or 5 hours. Hence the MTA would provide a 40% reduction in execution time over the ASCI Red.

Learning a single tree from the entire PDB dataset on one node of the SGI took approximately thirty days. This dataset is about 18 times larger than the Protein dataset. Based on our results, a conservative estimate for the MTA would be no more than four days – once again a significant reduction in execution time. The authors also had to modify code and make the data structures more compact to fit the data into memory. In contrast, the MTA is capable of handling the PDB and far larger datasets without any adjustments to the code.

Based on these comparisons to other high-performance architectures, we are confident that the fine-grain parallel model of the MTA-2 offers competitive performance in a variety of real-world applications. We emphasize again that a major motivation for the use of the multi-threaded model is to eliminate the need for partitioning large datasets on most other architectures. Our implementation is aimed specifically at processing extremely large datasets *without* partitioning, resulting in a single classifier from all of the data and preserving the interpretability of the model.

Unprecedented Scalability

To our knowledge, the largest dataset used with a decision tree algorithm in literature consisted of "nearly 40 million instances" (Jin & Agrawal 2003). In an effort to challenge our parallel implementation, we increased the size by 25% to 50 million instances. We were able to construct a single decision tree from the entire dataset in 14,850 seconds (just over 4 hours). This is an unprecedented accomplishment and constitutes an important contribution to the area of scalable data mining algorithm implementations.

Related Work

Mining Massive Datasets Learning from extreme datasets – containing a large number of examples, high dimensionality, or *both* – is a curse for most data mining algorithms. Even simple methods are troubled by the massive quantities of data we are faced with. To enable the processing of such datasets, two different computational paradigms have been considered: distributed computing and parallel computing (Kargupta & Chan 2000). Of the two, the distributed model has been studied and applied much more extensively, which is in large part due to availability of the necessary hardware. Such platforms consist of conventional processors tightly connected via a high-bandwidth network and a large shared memory. The problem with these systems is that memory capacity and communication latencies become bottlenecks, limiting the scalability of algorithms (Jin & Agrawal 2003). We have shown that leveraging multi-threaded processor technology can help overcome these restrictions.

Experience with the MTA The first scientific use of the MTA found in literature was by Bokhari et. al (Bokhari & Sauer 2004). The authors present a parallel implementation of their molecular dynamics code SIMU-MD on the MTA-2, which is used to simulate DNA translocation through a nanopore. The experimental results show that the MTA offers competitive performance when multiple processors are used, with a 2x speedup over a previous implementation on the Compaq Alpha. The authors praise the MTA for requir-

ing a relatively small porting and programming effort compared to other architectures.

More recently, Bader et al. have used the MTA-2 to attack a number of graph-theoretical problems (Bader, Cong, & Feo 2005), many of which involve independent computations on individual nodes or subgraphs and therefore are easily parallelizable. Their experimental results demonstrate excellent scalability of simple graph algorithms (e.g. breadth-first search), capable of processing a graph with over 200 million nodes and one billion edges. To our knowledge, with the exception of our proof-of-concept (Steinhaeuser & Chawla 2006), nobody has applied this technology to machine learning and data mining algorithms.

Conclusion

Machine learning algorithms continue to be challenged by ever-increasing dataset sizes. With this work, we have taken a new step towards meeting the emerging computational needs. Specifically, we have demonstrated improved scalability for decision tree construction on massive datasets using thread-level parallelism. Our implementation on the Cray MTA-2 exhibits good speedup characteristics as it scales along both the number of instances and the number of attributes. We performed additional experiments to compare its performance to other computing platforms and evaluate its use with ensemble approaches. Finally, we constructed a decision tree from a dataset 25% larger than any other used in literature. Based on our experimental results, we believe that massively multi-threaded architectures are suitable for an array of machine learning and data mining applications, especially in situations where the dataset size exceeds the memory capacity of other platforms.

Acknowledgements This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. BNCH3039003. We would hereby like to thank Peter Kogge and Jay Brockman for their comments and feedback. Thanks also to Cray Inc. for permission to use the MTA-2, and John Feo for his assistance.

References

Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Database Mining: A Performance Perspective. *IEEE Trans. Knowl. Data Eng.* 5(6):914–925.

Alverson, R.; Callahan, D.; Cummings, D.; Koblenz, B.; Porterfield, A.; and Smith, B. 1990. The Tera Computer System. In *ACM Int'l Conference on Supercomputing*, 1–6.

Bader, D. A.; Cong, G.; and Feo, J. 2005. On the Architectural Requirements for Efficient Execution of Graph Algorithms. In *33rd Int'l Conference on Parallel Processing*, 547–556.

Bokhari, S. H., and Sauer, J. R. 2004. Sequence alignment on the Cray MTA-2. *Concurrency and Computation: Practice and Experience, Special Issue on High Performance Computational Biology* 16(9):823–839.

Chawla, N. V.; Thomas E. Moore, J.; Hall, L. O.; Bowyer, K. W.; Kegelmeyer, P.; and Springer, C. 2003. Distributed Learning with Bagging-Like Performance. *Pattern Recognition Letters* 24(1-3):455–471.

Chawla, N. V.; Hall, L. O.; Bowyer, K. W.; and Kegelmeyer, W. P. 2004. Learning Ensembles from Bites: A Scalable and Accurate Approach. *Journal of Machine Learning* 5:421–451.

Chawla, N. V.; Eschrich, S.; and Hall, L. O. 2001. Creating Ensembles of Classifiers. In *ICDM*, 580–581.

Gehrke, J.; Ramakrishnan, R.; and Ganti, V. 2000. RainForest – A Framework for Fast Decision Tree Construction of Large Datasets. *Data Mining and Knowledge Discovery* 4(2-3):127–162.

Jin, R., and Agrawal, G. 2003. Communication and memory efficient parallel decision tree construction. In *Proceedings of Third SIAM Conference on Data Mining*.

Kargupta, H., and Chan, P. 2000. *Advances in Distributed and Parallel Knowledge Discovery*. AAAI/MIT Press.

Lyman, P., and Varian, H. R. 2003. How much information.

Mitchell, T. M. 1997. *Machine Learning*. McGraw-Hill Education.

Steinhaeuser, K., and Chawla, N. V. 2006. Exploiting Thread-Level Parallelism to Build Decision Trees. In *ECML PKDD Workshop on Parallel Data Mining*, 13–24.