Probability Convergence in a Multithreaded Counting Application

Chad Scherrer[1], Nathaniel Beagley[1], Jarek Nieplocha[1], Andrés Márquez[1],
John Feo[2], and Daniel Chavarría-Miranda[1]

[1]Pacific Northwest National Laboratory
Richland, WA 99352
{chad.scherrer, nathaniel.beagley, jarek.nieplocha,
andres.marquez, daniel.chavarria}@pnl.gov

[2]Cray, Incorporated
Seattle, WA 98104-2860
feo@cray.com

Abstract

The problem of counting specified combinations of a given set of variables arises in many statistical and data mining applications. To solve this problem, we introduce the PDtree *data structure, which avoids exponential time and space complexity associated with prior work by allowing user specification of the tree structure. A straightforward parallelization approach using a Cray MTA-2 provides a speedup that is linear in the number of processors, but introduces nondeterminism into probability estimates. We prove a general convergence result that bounds the nondeterministic deviation of probability estimates relative to a sequential implementation. Beyond PDtrees, this convergence result applies to any counting application that takes a multithreaded streaming approach.*

1. Introduction

A common problem in the analysis of multivariate discrete data is to count the number of times a given set of variables takes on a particular configuration. From a statistical perspective, this requires a set of *contingency tables* [1]; from a database perspective it is related to the idea of an *OLAP cube* [4]. Applications include mining data for association rules [2], and learning Bayesian networks [5] or other graphical models [3] or conditional independence structures [7].

A	B	C
1	0	1
0	1	1
0	0	2
0	2	1
1	0	3

Figure 1. A simple counting example.

For example, suppose that for variables A, B, and C, we have five data points, as shown in Figure 1. Many applications require calculations such as the estimation of $P(A = 0 \wedge C = 1)$. Note that only rows 2 and 4 contain data that simultaneously satisfy the constraints $A = 0$ and $C = 1$. Thus we estimate $P(A = 0 \wedge C = 1) = \frac{2}{5}$.

Though this brute-force approach is straightforward and works well for a simple example like this, each additional query requires the reprocessing of the entire data set. Better is to store a set of *contingency tables*. We might, for example, store $P(A = a \wedge C = c)$ for all values of a and c in a matrix, $P(A = a)$ for all values of a in a vector, *etc.* In total, then, we would have $2^3 - 1 = 7$ tables for this 3-variable example. But this data structure still contains a lot of redundancy, since for example $P(A = 0) = \sum_c P(A = 0 \wedge C = c)$.

Our alternative approach is based on a variant of Moore and Lee's ADtree data structure [5], which allows for very efficient count queries. In an ADtree, the root of the tree contains the total count. At each step down into the tree, another variable is instantiated with a particular value, with counts tracked at each level. ADtrees take advantage of the sparsity that typically results when many variables are simultaneously instantiated, and they allow fast queries: any count query can be answered in a number of steps proportional to the number of variables instantiated in the query. However, they can be quite expensive to populate (though no more so than a set of contingency tables); memory usage and computation time are both at best exponential in the

This work was funded by the U.S. Department of Energy's Pacific Northwest National Laboratory (PNNL) under the Data Intensive Computing Initiative. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

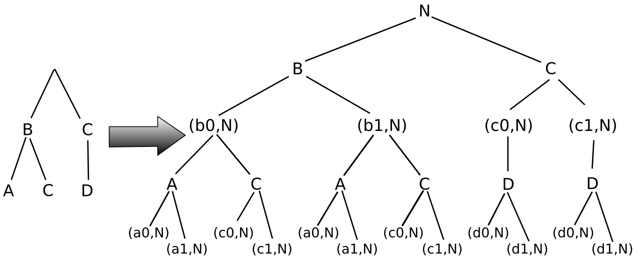

Figure 2. A guide tree and corresponding PDtree.

number of variables, since each is at best linear in the total number of combinations of variables.

Fortunately, in many cases, we need not store counts for every such combination. This may be due to the availability of a statistical model for the data, or to requirements involving computation time or memory. In such cases, limiting the data structure to reflect this can result in a corresponding increase in performance. Because the resulting data structure differs from an ADtree primarily in that we no longer store "all dimensions", but only "partial dimensions", we refer to it as a "PDtree". By specifying *a priori* which combinations of variables are to be stored, we reduce the number of steps required to traverse the PDtree each time a new record is inserted. The nested structure of the variables is specified in an auxiliary data structure called a "guide tree". Figure 2, to be described further in Section 2, shows an example of a guide tree and a PDtree built by using it.

Computation and memory requirements for such applications increase as we increase the number of variables, the number of levels of a given variable, and the total number of observations. For many real-world problems, at least one (and often all three) of these are large; thus parallelization quickly becomes attractive. We have chosen to parallelize over the insertion of records. Due to the irregular access patterns associated with PDtree updates, a shared-memory multithreaded approach is natural. Section 3 discusses further details about the multithreaded implementation.

2. PDtree Data Structure

Suppose we are given data consisting of observations on the variables $\{A, B, C, D\}$, and consider the problem of fitting the data to the Bayesian network model. For this example we make use of notation from *conditional probability* where, for example, $P(A|B) = \frac{P(AB)}{P(B)}$. For the directed graph $A \to B \to C \to D$, the probability factors as

$$\begin{aligned} P(ABCD) &= P(A)\,P(B|A)\,P(C|B)\,P(D|C) \\ &= \frac{P(AB)\,P(BC)\,P(CD)}{P(B)\,P(C)}. \end{aligned}$$

In particular, note that the only combinations of variables for which we are required to store counts are $\{AB, BC, CD, B, C\}$. The guide tree shown in Figure 2 (left) is based on the observation that this is equivalent to storing counts for $\{B, C, (A|B), (C|B), (D|C)\}$. Variables appearing by themselves correspond to the first level of the tree, while those conditional on one other variable appear one level down in the tree. This nested structure allows the relevant information to be stored with very little redundancy.

Even without a particular model to be fit, this flexibility of specifying the nesting structure of the combinations to be stored can be of great benefit. For example, the depth of a guide tree might be specified as a constant number of variables, or perhaps in terms of an entropy threshold. If the goal of an analysis is to learn the graphical structure of a Bayesian network, we may want to impose a constraint limiting the number of parents a given node can have. In terms of the guide tree, this just means that we only build the tree to a given pre-specified depth.

Note that once a PDtree is built, the guide tree is not needed for queries; all required information about the variables is contained within the PDtree itself. Thus the guide tree can be represented in any form that allows efficient traversal, such as a list-of-lists or similar structure, but we needn't be concerned with random lookup efficiency of the guide tree.

Another advantage that comes along with this approach is the potential to dynamically modify the set of combinations of variables being stored. Pruning the guide tree at any point in time prevents the corresponding variable combinations from being stored in the PDtree. This pruning can be based either on external stimuli or on characteristics calculated from the data being ingested into the tree. Consider, for example, an anomaly detection context. If it is determined that a particular variable combination is nearly uniformly distributed and thus unlikely to provide any anomaly information, the corresponding branch of the guide tree can be pruned, ensuring that resources are conserved for more influential calculations.

A number of reductions are made in an ADtree which we do not duplicate for PDtrees. Firstly, at a given depth the "most common value" (MCV) is specified in an ADtree to avoid redundant counting. We chose not to implement this reduction at this stage because this additional complication could adversely affect scalability, and the corresponding storage freed would not be worth the cost. In addition, the MCV reduction prevents the structure from being easily updateable, which is important in a streaming data application. Still, we plan to investigate MCV in a static context in future work.

Secondly, after recursing sufficiently deeply into an ADtree, the nested structure is replaced with a pointer to

```
while true {
  ptr = readfe(node.next)
  if ptr is null
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
  end if
} end while
```

```
while true {
  ptr = node.next
  if ptr is null
    ptr = readfe(node.next)
    if ptr is not null then continue
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
    node = ptr
  end if
} end while
```

Figure 3. Initial (left) and final (right) implementations of static PDtree population on MTA-2. Changes are marked in bold.

raw data. We did not implement this because the guide tree already bounds the recursion depth, and this change would require testing at each stage in the recursion whether to switch to raw data. This also would be awkward to implement for streaming data.

3. Multithreaded Framework

For a multithreaded implementation of a PDtree population algorithm, we chose to use the Cray MTA-2, which offers a simplified programming model and a way of hiding memory latencies, making it convenient for large linked data structures. The keys to a highly scalable, efficient PDtree algorithm on the MTA are: 1) a scalable data structure that supports increasing numbers of insertions, and 2) an insertion operation that is safe, concurrent, and minimizes synchronization costs. The MTA's shared memory and insensitivity to memory access patterns gives the programmer great latitude in choice of data structure. One can choose the data structure that best fits the problem rather than try to force the problem into a data structure that best fits the architecture.

On the MTA, the PDtree is a multiple type, recursive tree structure. A root node (one root node per column) is an array of ValueNodes. Interior and leaf nodes are linked lists of ValueNodes. Since a root is just a histogram of column values, its size and contents are easy to set. Our original implementation used a linked list at the top level; but, as we explain below, it suffered from high synchronization costs and did not scale past eight processors. Inserting a record at the top level requires only that we increment the counter of the right ValueNode.

Inserting the record at other levels of the tree requires us to traverse a linked list to find the right ValueNode. If we find the node, we increment its counter. If we do not find the node, we add the node to the end of the list and set its counter to 1. We use the MTA's `int_fetch_add` operation to increment counters. This operation is safe, performed in memory, and costs only one instruction cycle. Inserting the record at other levels of the tree when no node is found is trickier. To insure safety, a thread must lock the list's end pointer before inserting a new node. Implementing a critical section is easy on the MTA using the synchronized read and write operations, `readfe` and `writeef`, respectively. Our first implementation of this operation is shown in Figure 3 (left). This code is safe, but overly serial. It has a critical section per link rather than a critical section only at the end. For example, a better implementation is to first test the pointer before locking it. Note that in the time between the test and when the `readfe` instruction returns, another thread may grab the lock and insert a new node; thus, we must retest the pointer after the `readfe` instruction. If it is still null, then we can insert a new node; otherwise, we must continue to traverse the list. Our final version is both safe and efficient, see Figure 3 (right).

In this implementation, a critical section exists only at the end of a list. An insertion operation waits only if it tries to insert a node at the same time at the end of the same list as another insertion operation. Since the PDtree grows quickly into a massive, sparse structure, the probability of two threads colliding quickly approaches zero. Moreover, even if an insertion waits, the MTA processor on which it executes does not wait; the processor merely switches to another thread and continues executing instruction. Note that reason that we do not want to use a linked list at the top level is that all records are inserted in the same top level structure; thus, the probability of two insertions colliding is significant.

On a test data set consisting of one million records over nine variables, we have found this approach to scale linearly on up to 32 processors (all that were available at the time of testing). The modest size of this test data set is due to preprocessing that is currently done on a conventional 1-processor machine. For this test, we used a guide tree that tracks all possible combinations of variables, and performance scales linearly. Pruning the guide tree would result in smaller absolute runtimes, but could adversely affect scalability. At the other extreme is the degenerate case, corresponding to an empty guide tree. In this the only thing counted is the number of data points, and we would expect very fast absolute runtimes, but poor scalability. It is, in effect, the sparseness of the structure that makes the probability of blocking so low, leading to linear scaling. Further testing will be required to assess scalability for a variety of guide trees. We have found the slope of the best-fit line to be 0.08, indicating that each processor added increases the processing rate by 80000 nodes per second.

n	Sequential	Parallel, 3 threads
0	$-$	$- - -$
1	$+-$	$- - +-$
2	$+ + -$	$+ - + - -$
3	$+ + +-$	$+ - + + --$
4	$+ + + + -$	$+ - + + - + -$
5	$+ + + + +-$	$+ - + + + + --$

Figure 4. Comparing sequential and parallel completion strings

4. Dealing with Nondeterminism

The algorithm and performance results presented to this point are static in the sense that the data are all assumed to be available at once for random access. This is contrasted with a dynamic streaming approach where data are processed sequentially and are never held in memory in their entirety. In a static analysis setting, the size of the data is fixed and known in advance, and the data can be used to fully populate a PDtree before any analysis is performed. In a dynamic analysis setting, the PDtree is never assumed to be "fully populated"; queries can be posed concurrently with population of the data structure, and are answered relative to the state of the PDtree at the time of query.

For static analysis, the parallelization approach used is deterministic, because counts for the data as a whole do not depend on the order of the counting. However, using this same approach for dynamic analysis leads to a race condition; uncertainty of the order in which threads complete leads to nondeterminism of the accumulated counts at a given point in time. Thus, the counts measured are not exact, but only serve as an approximation. Fortunately, this nondeterminism is acceptable, because the approximation satisfies a convergence criterion which we will now describe.

For the schematics that follow, we use a "$+$" to indicate a data point that has been inserted into the PDtree, and a "$-$" to indicate that the insertion of a given data point is still processing. Thus $+ - + + -$ means that the first, third, and fourth values have been inserted, but the second and fifth are still processing (implying that there are two active threads). The variable n will denote the number of values that have been completely inserted. We will refer to the sequence of $+$'s and $-$'s as a *completion string*. Figure 4 compares the sequences of the evolving completion string as the PDtree is populated in the sequential and parallel implementations. Note that at each step, we compare the sequential and parallel completion strings based on the total count n of records inserted into the tree, though the times for each to get to a given n could be very different.

Now, consider a particular count at some point in the tree, corresponding to the number of times some combination of variables has taken on a particular value. This depends on n and, for a parallel implementation, it also depends on the order in which the insertion threads are completed. Let $c_{seq}(n)$ denote this count for the sequential implementation, and let $c_{par}(n)$ denote the count for a parallel implementation, for some arbitrary ordering of thread completions.

How close is $c_{par}(n)$ to $c_{seq}(n)$? The only way a given data point can make a contribution to the absolute difference $|c_{par}(n) - c_{seq}(n)|$ is if that point has been inserted by the sequential implementation, but not by the parallel one, or vice-versa. Thus we can find a bound on the absolute difference by maximizing the number of points completed by one implementation but not the other.

Suppose the parallel implementation uses k threads. At a given time, there is always one thread working on the most recent data point, but the remaining $k-1$ could, in principle, all be processing data points that the sequential implementation has already completed. Each of these $k-1$ data points could contribute to $c_{seq}(n)$, with each such data point potentially altering $c_{seq}(n)$ by a count of one. At the opposite extreme, each of the $k-1$ data points processed in parallel but not sequentially could contribute to $c_{par}(n)$. In either case, we arrive at the following result.

Lemma. Let $c_{seq}(n)$ and $c_{par}(n)$ be the number of times a particular collection of variables takes on a specified configuration, given the number n of observations so far, for a sequential and parallel implementation, respectively. If the parallel implementation uses k threads, then

$$|c_{par}(n) - c_{seq}(n)| < k.$$

□

The most common use of counts as computed using PDtrees is to estimate the probability of a given event. Using maximum likelihood, the estimated probability of a given event as a function of n is given by

$$\hat{p}_{par}(n) = \frac{c_{par}(n)}{n} \qquad \hat{p}_{seq}(n) = \frac{c_{seq}(n)}{n}$$

Theorem. For a counting application, suppose a sequential implementation is compared to a parallel implementation using k threads, and let n be the number of observations. The estimated probabilities are then related by

$$\hat{p}_{par}(n) = \hat{p}_{seq}(n) + O\left(\frac{k}{n}\right).$$

Proof. Using the result from the lemma,

$$\begin{aligned}\left|\hat{p}_{par}\left(n\right)-\hat{p}_{seq}\left(n\right)\right| &= \left|\frac{c_{par}\left(n\right)}{n}-\frac{c_{seq}\left(n\right)}{n}\right| \\ &< \frac{k}{n}.\end{aligned}$$

□

Note that this can be used to determine a number of threads to use to guarantee a desired degree of consistency with a sequential implementation. If $k \leq n\varepsilon$, then

$$\left|\hat{p}_{par}\left(n\right)-\hat{p}_{seq}\left(n\right)\right| < \frac{k}{n} \leq \varepsilon.$$

5. Conclusion

The static implementation of PDtree population scales linearly on a Cray MTA-2, using a full guide tree. The nondeterminism that results when using the same parallelization approach for dynamic analysis might be considered a race condition. However, we have shown that the sequence of probability estimates converge to those given by a sequential implementation, bounding the effects of nondeterminism. This bound applies not just to PDtrees, but to any multithreaded counting application that assigns a thread to each consecutive record.

The linear scalability attained for the static case was dependent upon the ability to have all of the data in memory at once. It remains to be seen whether scalability of dynamic analysis can be pushed to this extreme. Also, as mentioned above, we have not implemented the most-common-value reduction used in ADtrees. Perhaps this could be used at least for static analysis. It remains to be seen whether the MCV reduction leads to any performance hot-spots.

Besides discrete data, the PDtree can be altered to allow for a variety of types at the leaves. In particular, we have begun to investigate storing sufficient statistics for continuous distributions like normal and von Mises. Since these distributions are conditional on a number of discrete values in the PDtree, the marginal distributions in these cases have a very flexible form and allow for a wide variety of modeling approaches.

Finally, our work with PDtrees has opened up a wide variety of research questions. For a given collection of sets of variables to store, can the guide tree be constructed in a way that minimizes expected storage costs, or that maximizes the scalability of population? Are these two ends at odds with each other? Under certain conditions, storage costs can be minimized by using conditional entropy to order each subtree of the guide tree, but it is not clear to what extent this might generalize, or how it might affect scalability.

References

[1] A. Agresti. *Categorical Data Analysis*. John Wiley & Sons, Inc., Hoboken, NJ, second edition, 2002.

[2] B.S. Anderson and A.W. Moore, “AD Trees for Fast Counting and for Fast Learning of Association Rules,” *Proc. Fourth Int’l Conf. Knowledge Discovery and Data Mining*, 1998.

[3] S. Andersson, D. Madigan, and M. Perlman, “An Alternative Markov Property for Chain Graphs,” *Proceedings of the Twelfth conference on Uncertainty in Artificial Intelligence*, pages 40–48. Morgan Kaufmann, 1996.

[4] E.F. Codd, S.B. Codd, and C.T. Salley, “Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate,” *http://dev.hyperion.com/resource_library/white_papers/ providing_olap_to_user_analysts.pdf*, 1994.

[5] A.W. Moore and M.S. Lee, “Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets,” *J. Artificial Intelligence Research*, vol. 8, pp. 67-91, 1998.

[6] D. Pavlov, H. Mannilla, and P. Smyth, “Beyond Independence: Probabilistic Models for Query Approximation on Binary Transaction Data,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 6, pp. 1409-1421, 2003.

[7] C. Scherrer and N. Beagley, “Conditional Independence Modeling for Categorical Anomaly Detection,” *Proc. Join Ann. Meeting of the Interface and Classification Soc. N. America*. 2005