Accelerated Learning on the Connection Machine

Diane J. Cook **Lawrence B. Holder**

University of Illinois
Beckman Institute
405 North Mathews, Urbana, IL 61801

Abstract

The complexity of most machine learning techniques can be improved by transforming iterative components into their parallel equivalent. Although this parallelization has been considered in theory, few implementations have been performed on existing parallel machines. The parallel architecture of the Connection Machine provides a platform for the implementation and evaluation of parallel learning techniques. The architecture of the Connection Machine is described along with limitations of the language interface that constrain the implementation of learning programs. Connection Machine implementations of two learning programs, Perceptron and AQ, are described, and their computational complexity is compared to that of the corresponding sequential versions using actual runs on the Connection Machine. Techniques for parallelizing ID3 are also analyzed, and the advantages and disadvantages of parallel implementation on the Connection Machine are discussed in the context of machine learning.

Introduction

The explosive complexity of most machine learning algorithms is dominated by an iterative search through a large concept space. This complexity can be reduced by transforming the iterative components into parallel procedures. The advent of parallel machines such as the Connection Machine provides the means to implement parallel versions of machine learning algorithms. Despite the availability of parallel hardware and theoretical methods for parallelization, virtually no actual implementations of parallel learning algorithms have been attempted.

This paper describes the Connection Machine implementation of two learning algorithms: Perceptron and AQ. Samples of the parallel Lisp code are given to illustrate the Lisp interface language. Actual runs of the parallel versions are compared to the sequential versions to demonstrate the significant increase in speed. This paper also describes constraints imposed by the Connection Machine environment which affect the design of these parallel algorithms and future parallel machine learning algorithms.

In this paper, we first provide a general description of the Connection Machine's architecture and *Lisp programming environment. Next, we describe the parallel implementation of the perceptron learning algorithm and compare its performance to that of the sequential version. A similar treatment of the AQ learning algorithm is then presented. Finally, we illustrate the difficulties met while attempting to parallelize the ID3 learning algorithm and possible alternative approaches.

The Connection Machine

Computers use their powerful hardware inefficiently – most of the transistors in a von Neumann computer are idle at any given instant. It is speculated that humans make much better use of their brains by computing in parallel. Only recently have engineers simulated this approach by building concurrent machines – machines that are able to split a task among various processors which solve the sub-tasks in parallel.

A few applications of parallelism that are venturing into the realm of Artificial Intelligence are mentioned in [6, 7, 8], including pattern perception, computer vision, image processing, knowledge representation, and natural language understanding. Virtually no work has been done to parallelize inductive learning techniques.

The Connection Machine is a "data parallel" computing system. Most high-performance computers, including popular parallel machines such as the Cray, include a small number of fast, relatively complex processors and a single large working-memory space. In contrast, a data-parallel machine has a large number of individual processors, each with a relatively small memory space of its own, and an interconnection scheme allowing rapid communication with and among processors. The goal is to speed up computation by "parceling out" individual data elements (or small groups of elements) to individual processors, and allowing the processors to operate simultaneously on their own small pieces of the overall problem.

The algorithms described in this paper are implemented on the Connection Machine Model CM-2, built by Thinking Machines Corporation (See [2] for a more complete description of the CM-2). The CM-2 has 32,768 processors structured in a two-dimensional grid. The programming environment includes the *Lisp language [1], which we have used to implement Parallel-Perceptron and Parallel-AQ.

The main data structure found in *Lisp is the *pvar*. A pvar represents a collection of values. Each element of the pvar is stored in a unique processor on the Connection Machine. The components of a pvar may be (theoretically) any Lisp value. *Lisp also contains a set of predefined functions that operator in parallel on pvars. These functions usually end with !! (if they return a pvar) or begin with *. For example, the command (*defvar a (!! 5)) creates a pvar that stores the integer 5 in each processor. The function (*setf a (+!! b c)) sets the contents of pvar *a* to the sum of the contents of pvar *b* and pvar *c*.

The current version of *Lisp is very limited in its capabilities. This version has a very small set of parallel lisp functions, and the data structures do not allow symbols or lists to be represented, but only numbers and arrays. This means that each symbol must be converted to a number, and each list must be converted to an array. Thus, there is a great deal of overhead in transforming a sequential representation into its *Lisp representation.

Each processor in the Connection Machine is very small, and by itself could not hold much data. Instead, the data is distributed across the processors and the host computer signals a single operation that each processor performs in parallel. The *Single-Instruction-Multiple-Data (SIMD)* architecture of the Connection Machine limits the type of parallelization that can be performed. A procedure such as examining the contents of each node in a graph is easy to perform in parallel on the Connection Machine, while a procedure which applies a different function to each node in the graph is not possible to parallelize on the Connection Machine.

The advantages and disadvantages of the Connection Machine architecture and language interface will be illustrated as we describe the parallel implementations of Parallel-Perceptron and Parallel-AQ in the next few sections. Constraints on the design of these algorithms resulting from the Connection Machine environment will be shown to affect the design of parallel learning programs in general.

Perceptron

This section presents the perceptron learning model and describes a parallel implementation of the algorithm. The complexity of the sequential and parallel procedures are compared.

The Algorithm

The perceptron learning algorithm invented by Rosenblatt [5] is a very natural one to parallelize. The perceptron is a neural network model. In a simple perceptron, each node x_i in the first layer represents an element of the vector of feature values, and is assigned a weight w_i. The output y of the network is a linear combination of the feature vector x and the weight vector w:

$$y = wx = \sum_i x_i w_i.$$

The sequential Perceptron procedure uses this perceptron model to learn a concept. Each node represents a particular attribute-value pair v_i. Each example e fed to the network is described by a set of attribute-value pairs. If the example contains the attribute-value pair v_i, the corresponding node x_i is assigned the value 1, otherwise x_i has the value 0. To classify the input as a positive or negative example of the concept the network represents, the procedure uses the formula above to determine the output y of the network. The output is compared to a threshold t:

```
If $y > t$
Then $e$ is a positive example of the concept
Else $e$ is a negative example of the concept
```

The perceptron is usually given a set of examples on which it is *trained*. If it incorrectly classifies a training example, the threshold and the weights corresponding to the misclassified example are adjusted. The network is said to have learned the concept when it correctly classifies the training set of examples.

Parallel Implementation

Two operations dominate the computation performed by Perceptron: the calculation of the output, and the adjustment of the weights. These operations are also consistent across the nodes in the network; that is, the same set of operations are performed for each node. The parallel version of Perceptron, called Parallel-Perceptron, vectorizes these operations.

To parallelize Perceptron, we must first decide what *data* to parallelize. The operations we wish to parallelize are performed across the nodes in the network, so the nodes can be spread out across the processors. Each processor contains the following structure:

```
(*defstruct NODE
                ;; Weight assigned to this node
  (weight 0.0)
                ;; Instance values for this node
  (examples (make-array *numexamples*)))
```

The structure contains the current weight for the node, and an array of examples. Each element of the example array has the value 0 or 1. If the example includes that particular attribute value, it is assigned a 1, otherwise it is assigned a 0. When a set of training examples is input to the system, Parallel-Perceptron performs the following functions until all of the training examples are correctly classified:

```
                ;; Look at all of the examples
(dotimes (i *numexamples*)
  (*if
    (compute-perceptron-output!!
          nodes i threshold)
    (*cond
                ;; Misclassified a negative example
      ((false-positive!!
          (node-examples!! nodes) i)
        (*setf threshold (+!! threshold delta))
                ;; Adjust the node weights
        (*setf (node-weights!! nodes)
          (-!! (node-weights!! nodes) delta)))
                ;; Correctly classified the example
      (t nil!!))
    (*cond
                ;; Misclassified a positive example
      ((false-negative!!
          (node-examples!! nodes) i)
        (*setf threshold
          (-!! threshold delta))
                ;; Adjust the node weights
        (*setf (node-weights!! nodes)
          (+!! (node-weights!! nodes) delta)))
                ;; Correctly classified the example
      (t nil!!))))
```

The above code first calls **compute-perceptron-output!!**, shown below, which returns TRUE if the example is classified by the network as a positive example, and NIL (false) otherwise. If the example is incorrectly classified as positive (as determined by the call to **false-positive!!**), then the threshold is increased and the weights reduced. If the example is incorrectly classified as a negative instance (as determined by the call to **false-negative!!**), then the threshold is reduced and the weights increased.

Parallel-Perceptron computes the output of the network for $example_i$ by summing the values of the corresponding weights. The function that computes the output value is defined as:

```
(*defun compute-perceptron-output!!
             (nodes i threshold)
    ;; Returns network's classification of example i
  (let ((sum 0))
    ;; Select attribute-values referenced by example
    (*when
      (eq!!
        (aref!! (node-examples!! nodes) (!! i))
        (!! 1))
    ;; Sum node weights
      (setf sum (*sum (node-weights!! nodes))))
    ;; Compare sum to the threshold
    (>!! (!! sum) threshold)))
```

When a parallel conditional statement such as *cond or *when is executed, *Lisp activates only the processors for which the condition returns TRUE. Only those processors can be accessed throughout the corresponding block of code. Because each processor represents a distinct attribute-value pair, **compute-perceptron-output!!** selects those processors whose corresponding attribute-value pair is referenced by the current example. The weights for only those selected processors is summed and compared to the threshold.

The output is used to determine if the input is a positive or negative example of the concept. If the network misclassifies the example, Parallel-Perceptron updates the weights for each node using a single parallel operation.

It should be noted here that the operation used to sum the weights is not a purely parallel operation, since a single variable is being updated by the value in each processor. However, *Lisp provides the *sum operator which utilizes the organization of the processors to minimize the number of operations performed. While the time spent performing the summation is thus greater than a single operation, it is much less than would be required to iterate over every node, as is done in the sequential implementation

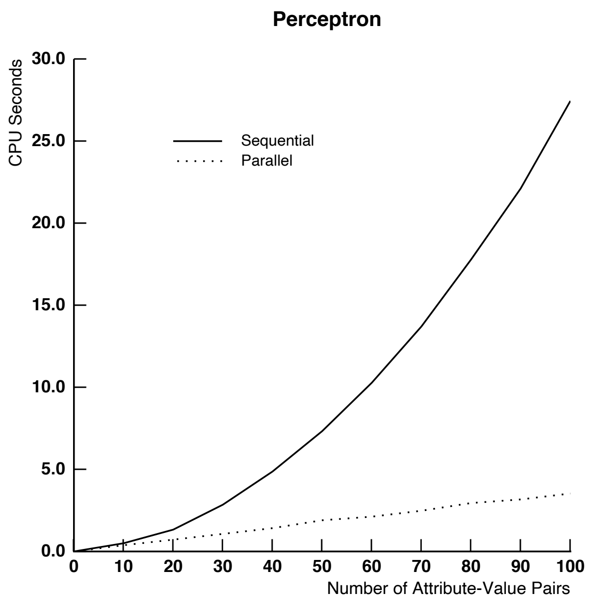

Figure 1: Sequential and Parallel Perceptron Results

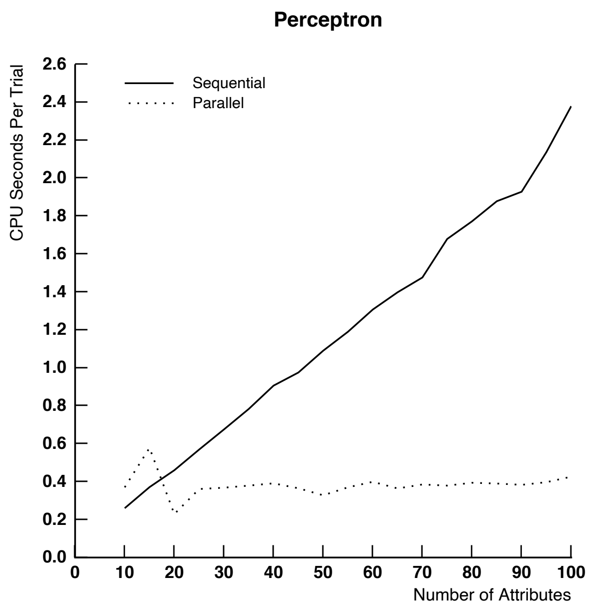

Figure 2: Perceptron Time per Trial

of Perceptron.

Comparison of Complexity Results

Because the operations involving the nodes of a perceptron are now parallelized, it is expected that the time complexity of Parallel-Perceptron would remain constant as the number of attribute-value pairs increases, while the complexity of sequential Perceptron would increase linearly. To test this hypothesis, the sequential and parallel versions of Perceptron were run on a series of examples. The number of attribute-value pairs were increased by 10 for each test, and the number of examples was always half the number of attribute-value pairs (half of these were positive and half were negative). The sequential version was run on the Connection Machine, but did not utilize multiple processors.

Figure 1 shows the results of this experiment. The x axis represents the number of attribute-value pairs, and the y axis represents the time in CPU seconds taken to learn the concept from the training set. As is expected, the line representing the sequential implementation run on this data increases more rapidly than the parallel version. The parallel version also increases in complexity over the number of attribute-value pairs, but this is primarily due to the corresponding increase in examples.

The previous experiment increases the number of examples along with the number of attributes. A second experiment was run in order to compare the performance of the sequential and parallel perceptron programs as only the number of attributes increases. Starting with ten binary-valued attributes, the experimental method selects three sets of fifty examples (chosen at random). Each set is run through both sequential and parallel versions, and the execution time per trial is averaged over the three runs. Further results are gathered for increments of five attributes, up to 100 attributes. A trial occurs when the perceptron classifies every example in the example set, adjusting node weights for misclassifications. The perceptron continues trials until every example is classified correctly.

Figure 2 shows the results of this second experiment. The x axis displays the number of attributes, and the y axis displays the average time per trial (in seconds) over the three sets of fifty examples. The results show that the sequential perceptron's time per trial increases with the number of attributes, while the parallel perceptron's time per trial remains essentially constant.

As is demonstrated in this section, learning algorithms exist whose parallel implementation is straightforward and resulting decrease in complexity is impressive. Neural network models in general lend themselves to parallelization. The following sections describe algorithms whose operations are not as easily segregated. As a result, the parallelization of these procedures is much more difficult.

AQ

This section describes the sequential AQ algorithm and the parallel implementation. Complexity results for both implementations are compared.

The Algorithm

The AQ algorithm was developed by Michalski [3]. Given a set of positive and negative examples expressed as a conjunction of attribute-value pairs, AQ finds a disjunctive normal form description (called a *complex*) of a concept covering all the positive examples and none of the negative examples. The algorithm proceeds by first selecting a seed from the set of positive examples. Next, a *star* is generated for the seed. A star contains a set of complexes that cover the seed, but do not cover any of the negative examples. The search for a star is a beam search with **maxstar** as the beam width.

Once a star has been generated for a seed, the complexes in the star are sorted using a lexicographic evaluation function (LEF). The LEF used in both the sequential and parallel implementations sorts first by number of positive examples covered and then by the generality of the complex. The best complex according to the LEF is retained as a disjunct in the final concept, and the positive examples covered by the disjunct are removed from the set of positive examples. Until all positive examples are covered, AQ selects a new seed, generates a star for the seed and retains the best complex from the star according to the LEF. When all positive examples are covered, AQ returns the set of retained disjuncts as the final concept.

Most versions of AQ allow internal disjunction within the final concept description. Internal disjunction provides the ability for features within a single disjunct to have more than one value. Neither the sequential or parallel versions of AQ used here allow for internal disjunction.

Parallel Implementation

The main operation that dominates the computation of AQ is star generation. The star for a seed begins with a single complex covering every example. For each negative example, the complexes in the star are specialized so that they no longer cover the negative example. Because there are multiple specializations possible for each negative example, the star can grow to be exponential in size. Specifically, the star can grow to be 2^f in size, where f is the number of features. This exponential growth is the reason for the beam search constrained by the beam width **maxstar**.

AQ can be parallelized by storing each element of the star in a separate processor. The corresponding *Lisp structure is defined as:

```
(*defstruct STAR-ELEMENT
                    ;; Pointer to a positive event
  (seed-index -1)
                    ;; Pointer to a negative event
  (neg-index -1)
                    ;; Lef value for this element
  (lef -1)
                    ;; Complex for this element
  (complex (make-array *numfeatures*)))
```

Whereas sequential AQ must consider the elements of the star one at a time, Parallel-AQ specializes each element simultaneously. With 2^{15} processors on the Connection Machine, problems with 15 features or less can be handled. If the number of features is less than 15, another avenue for parallelizing AQ is revealed. Because each seed requires 2^f processors to hold its star, $(15 - f)$ seeds may be considered simultaneously instead of just one. The best complex can then be selected from amongst several stars generated for multiple seeds, possibly improving the accuracy or reducing the number of disjuncts in the final concept.

The final parallelization possible for AQ is the calculation of the LEF. Because each complex is now contained in a unique processor, the LEF for each complex in each star can be calculated simultaneously. The sequential AQ must calculate the LEF iteratively for each complex in the star.

Parallel-AQ employs all of the parallelizations described above. There is no need for **maxstar**, because all possible complexes in the star can be operated on simultaneously. Multiple seeds are used when the problem does not exceed the capacity of the CM-2.

Comparison of Complexity Results

In order to compare the complexity of sequential and parallel versions of AQ, the sequential AQ was modified to allow multiple seeds, and **maxstar** was set to infinity. Thus, both the sequential and parallel

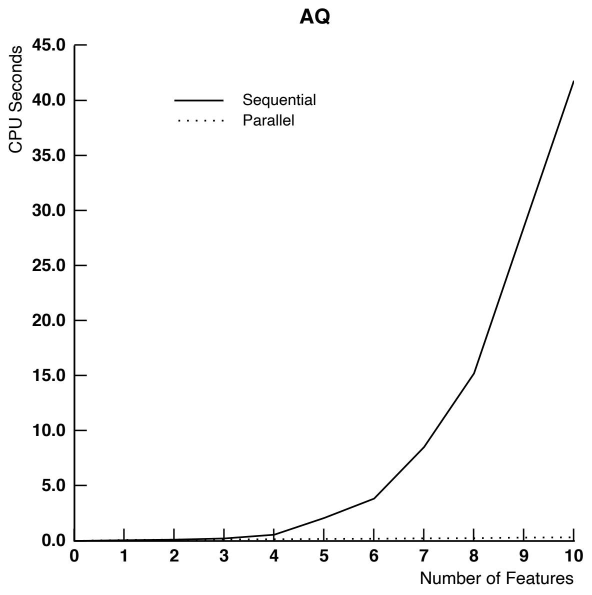

Figure 3: Sequential and Parallel AQ Results

versions are considering | *seeds* | $*2^f$ complexes for each disjunct in the final concept. Because Parallel-AQ operates on each complex simultaneously, we expect an exponential speedup over sequential AQ.

Figure 3 compares the results of 10 runs of both sequential and parallel versions of AQ. For each run, the number of positive and negative examples was set to the number of features. The sequential version was run on the Connection Machine, but was not vectorized. As expected, the time complexity of the sequential AQ increases exponentially with the number of features, while the CPU time for Parallel-AQ remains virtually constant. The time complexity of Parallel-AQ actually increases linearly with the number of negative examples.

In addition to the improvement in time complexity, the parallelization of AQ offers the possibility of producing a better concept (i.e., simpler or more accurate) over traditional sequential implementations of AQ. This is due to the capability of considering multiple seeds. Although the sequential version used here could consider multiple seeds, this ability greatly degrades the time complexity of sequential AQ. The disadvantage for Parallel-AQ is that each 2^f complexes for a seed is stored in a processor. Thus, the CM-2 may be unable to handle problems with a large number of features. Furthermore, the addition of internal disjunction functionality increases the number of complexes per seed to 2^{vf}, where v is the number of possible values for each feature. However, for problems that fit within the number of processors of the Connection Machine, Parallel-AQ offers considerable improvements over the sequential version.

ID3

This section describes the sequential ID3 algorithm used to build a decision tree for a given concept. Ways of parallelizing this algorithm are presented, and limitations are described that make such an implementation difficult.

The Algorithm

ID3 is a program developed by Quinlan to induce decision trees [4]. A decision tree represents a decision procedure for determining whether an instance is a positive or negative example of the concept. A decision tree is formally defined to be either:

1. A leaf node (or answer node) that contains a class name, or
2. A non-leaf node (or decision node) that contains an attribute test with a branch to another decision tree for each possible value of the attribute.

The sequential ID3 procedure builds a decision tree from a set of training examples by choosing one of the given features as the root of the tree and recursively making trees for each of the resulting categories. The feature is chosen based on its entropy value at that point in the tree – the feature with the highest entropy value is selected. Computing the entropy value for a feature involves counting the number of positive and negative examples for each of the values.

Possible Parallel Implementations

There are a number of ways that ID3 could be parallelized. The features could be distributed across the processors, and for a given level in the tree each feature computes its entropy value in parallel. Alternatively, the feature values could be distributed across the processors, each calculating in parallel the number of positive and negative examples referencing that value (this sum is then used to compute the corresponding feature's entropy value). Finally, the examples themselves could be distributed across the processors and a *sum could be used to compute the number of positive or negative examples with a certain value.

These ideas for parallelization are appealing in theory but are actually very difficult to implement. Although it is desirable to have each feature compute its entropy value in parallel, it is not a feasible task. This is due to the fact that each node at the current level of the decision tree examines each feature. Let n represent the number of nodes at the current level i. Each feature is being considered not just once at a level i, but n times, using n different sets of data. Unfortunately, this requires performing a separate operation for each set of data, yet using only one processor's feature information, a task which is not possible on the Connection Machine.

The only way this implementation would be possible is if the features are duplicated. A duplicate must be created for each node at level i. As the number of nodes increases, the number of duplicates and thus processors required becomes very large. The time expense is traded here for space expense, and the algorithm used to duplicate the features and calculate which processor to hold them is very complex. The same problem occurs when values are distributed among the processors – the number of corresponding positive and negative values for each node at level i must be counted, thus the values must also be duplicated.

Neither of these alternatives provides much benefit over the sequential ID3 procedure. The last alternative, distributing the examples over the processors, provides some benefit, but as the *sum operation must be used to count the number of possible positive or negative examples for a given value, this is not a completely parallel implementation.

The discussion in this section should bring to light the nature of algorithms that can be parallelized on the Connection Machine. While the machine can parallelize operations, these operations must be the same for each piece of data. It is the *data* that is truly being parallelized. There may be additional methods of parallelizing ID3, but this remains an avenue of future research.

Conclusion

The parallel implementations described in this paper push the intersection of parallel methodologies and parallel hardware into the realm of machine learning. Experimentation with the perceptron and AQ learning algorithms illustrate the computational benefits possible by implementing these algorithms on the Connection Machine. Descriptions of the implementations offer a valuable analysis of the types of parallelizations and performance benefits possible on the Connection Machine using *Lisp as the parallel implementation language.

The implementations also revealed constraints imposed by the Connection Machine architecture and environment. The "data parallel" architecture limits the types of parallelizations that can be performed. Constraints on the availability of data structures provided by the *Lisp programming environment complicates the transformation of sequential programs. These constraints will affect the design of future parallel implementations of learning algorithms on the Connection Machine. Despite the complications in merging theoretical parallelizations with existing parallel hardware, the parallel implementations of perceptron and AQ demonstrate the feasibility of accelerated learning on the Connection Machine.

Acknowledgements

We would like to thank Robert Stepp for his helpful suggestions. We would also like to thank the National Center for Supercomputing Applications for the use of the Connection Machine. The parallel versions of the programs were adapted from sequential versions written by Ray Mooney.

References

[1] Thinking Machines Corporation. *Connection Machine: Programming in Lisp.* Thinking Machines Corporation, Cambridge, MA, 1988.

[2] W. C. Mayse and R. Kufrin. A fast look at the CM-2 connection machine. In *Data Link.* National Center for Supercomputing Applications, September–October 1989.

[3] R. S. Michalski and J. B. Larson. Selection of most representative training examples and incremental generation of VL_1 hypotheses: the underlying methodology and the description of programs ESEL and AQ11. Technical Report 867, University of Illinois, Urbana-Champaign, IL, 1978.

[4] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[5] D. E. Rumelhart and D. Zipser. Feature discovery by competitive learning. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing, Volume 1*, chapter 5, pages 151–193. MIT Press, Cambridge, MA, 1986.

[6] L. W. Tucker and G. G. Robertson. Architecture and applications of the connection machine. *Computer*, 21(8):26–38, August 1988.

[7] L. Uhr. *Multi-computer Architectures for Artificial Intelligence.* John Wiley and Sons, Inc., New York, NY, 1987.

[8] D. L. Waltz. Applications of the connection machine. *Computer*, 20(1):85–97, January 1987.