*CART: Parallel CART on the CM-2

Chris McConnell

October 1, 1995

Abstract:

*CART is a parallel implementation of the CART decsion tree algorithm in *LISP on the CM-2 from Thinking Machines. It exhibits significant speed up over a serial version of the same algorithm on a SUN-4. The implementation is described together with some performance comparisons with the serial version.

1 Introduction

CART is an algorithm for inducing decision rules from a set of labeled examples each of which has a collection of numeric and catagorical feature attributes possibly with missing values. It is based around the idea of recursively partitioning the feature space with hyperplanes perpendicular to a single feature. It came out of work in the statistical community and is similar to the ID3 algorithm developed in AI. [6]

Several people have proposed parallel implementations of ID3, but they have not actually implemented it. ID3 [6] is a close relative to CART that was developed at the same time only in the AI community as opposed to the statistical one. No one has previously discussed the parallel implementation of CART. One proposal used ID3 as an instance for a class of parallel models for MIMD machines. [5] A second proposal for parallelizing ID3 on the CM-2 suggested some techniques but concluded that the problem was very difficult and unlikely to gain much from parallelization. [2]

*CART is a parallel implementation of CART on the CM-2. The CM-2 is a Single Instruction Multiple Data (SIMD) machine. This means that only

one instruction is executed at a time, but that instruction is done in parallel on the data stored in all of the currently selected processors. *CART supports the full spectrum of capabilities found in the serial CART algorithm including: catagorical and numeric features with missing values, prior class distributions, class misclassification costs, cross validation and feature ranking. As will be seen, the implementation of *CART is complicated, but it also demonstrates significant performance improvements.

2 The CART Algorithm

This section provides a brief overview of the CART methodology. [1] For simplicity, it is assumed throughout that class prior probabilities are computed as sample proportions and that a simple 0-1 loss function is specified. Note, however, that simple modifications to expressions 2 and 3 allows for the accommodation of asymmetric costs and user specified prior probabilities.

The CART algorithm proceeds as follows: A classification tree, T, initially consists only of the root node τ_1, containing all of the cases in $\mathcal{L}$[1]. A search is made through the set of possible binary splits of τ_1 until an optimal split s_{opt} is found. For real-valued attributes A, splits take on the form $A \leq r$, where r is a midpoint of any adjacent pair found in the set of unique values attained by A. For categorical attributes which can attain possible values $D_1, \ldots, D_M$, splits take on the form $A \in \{D\}$ where $\{D\}$ is one of the subsets of $\{D_1, \ldots, D_M\}$. Once s_{opt} is located, the root node acquires two descendants, τ_L and τ_R, each of which now contains a subset of $\mathcal{L}$. Geometrically, s_{opt} describes a partition orthogonal to one of the coordinates of the measurement space. The partitioning process is applied recursively to each leaf, continuing until all leaves are too small, contain cases from a single class only, or contain identical measurement vectors. When partitioning ends, the initial tree has grown to become a terminal tree T_{max}. Figure 1 shows how a two dimensional measurement space can be partitioned.

Since, with a noisy domain, T_{max} almost certainly overfits $\mathcal{L}$, it is pruned backwards, yielding a unique, nested sequence of subtrees $\{T_{max} \preceq, \ldots, \preceq \tau_2 \preceq \tau_1\}$, and an error estimation procedure is then applied to the sequence in order to select an optimally sized subtree. Subsequent observations are

[1] $\mathcal{L}$ consists of N cases each of which consists of a *known* class θ and a vector of attributes.

Figure 1: *The scatter plot on the left illustrates the distribution of two classes (represented by solid and hollow dots) in the coordinate plane. Lines perpendicular to the coordinate axes, and labeled* a *through* d*, illustrate how the space could be partitioned in order to correctly classify the observed cases. The binary tree on the right illustrates how such a partitioning can be represented as a decision rule. Each interior node has a label corresponding to the appropriate partition and asks a question of the form "is* $x < c$ *?". Arcs in the graph represent binary responses to the questions and leaves represent class assignments.*

dropped down the selected tree, and classified according to the class membership of the plurality of cases populating the leaf into which they fall.

2.1 Splitting Criteria

Since the tree building mechanism attempts to generate a tree with increasingly homogeneous (pure) nodes, a natural splitting criterion is one which tends to maximize some index of purity. For C classes, using an impurity function ϕ, and defining $p(j|\tau)$ as the proportion of class j cases at node τ, we define the impurity of node τ as: $\iota(\tau) = \phi(p(1|\tau), \ldots, p(C|\tau))$. Any split s on τ sends a proportion p_L of cases to τ_L and a proportion p_R to τ_R, and the decrease in impurity obtained from splitting τ by s is defined by

$$\Delta\iota(s,\tau) = \iota(\tau) - p_R\iota(\tau_R) - p_L\iota(\tau_L) \tag{1}$$

It turns out that the performance of the classification tree algorithm is relatively insensitive to the choice of ϕ, as long as certain concavity requirements are met [1]. Some interesting candidates for ϕ are the information index (as used in *ID3*) and the Gini[2] index of diversity, defined as:

$$\sum_{i=1}^{C}\sum_{j=1}^{C} p(j|\tau)p(i|\tau), i \neq j. \tag{2}$$

2.2 Pruning the Tree

As described previously, CART avoids the pitfalls of stopping rules, by first growing a tree of maximal size, and then pruning backwards in an appropriate manner. The pruning approach is based upon the notion of assigning a complexity penalty to the growth of large trees. Defining the cost complexity of tree T as $R_\alpha(T) = \hat{R}(T) + \alpha\mathcal{T}$, where α is a real-valued scalar, $\mathcal{T}$ is the number of leaves in T and $\hat{R}(T)$ is the *resubstitution estimate*[3] of $R^*(T)$ (the *true* error rate of T) we can, without exhaustive search, locate that *unique* smallest subtree $T(\alpha)$ of T_{max} which minimizes $R_\alpha(T)$ for fixed α. Although it is not implemented in this manner, minimal cost-complexity pruning can be thought of as a procedure in which α is steadily increased,

[2] Corrado Gini, 1884-1965.

[3] The estimate obtained by classifying the cases in $\mathcal{L}$ by the *same* tree induced from $\mathcal{L}$.

with pruning occurring at threshold values α', thus yielding a nested sequence of minimizing subtrees.

2.3 Error Estimation

Once the sequence of minimizing subtrees has been obtained, a minimum risk subtree, T_{opt} is sought. If $\hat{R}(T)$ is the criterion used to select T_{opt} then T_{max} will always be selected, since $\hat{R}(T)$ decreases with increasing $\mathcal{T}$. However, since $\hat{R}(T)$ is obtained as a result of resubstituting $\mathcal{L}$ into a tree induced from $\mathcal{L}$, $\hat{R}(T)$ is a *downwardly biased* estimate of $R^*(T)$ and therefore inappropriate.

A *cross-validation* (cv) technique for selecting an unbiased estimate of $R^*(T)$ randomly divides $\mathcal{L}$ into V test samples, $\mathcal{L}_1, \ldots \mathcal{L}_V$, each of which has a (nearly) identical number of cases, and V learning samples $\mathcal{L}_{(v)} = \mathcal{L} - \mathcal{L}_v$ are created. For each $\mathcal{L}_{(v)}$, an auxiliary tree, $T_{(v)}$ of maximum size is grown and pruned according the the cost-complexity values established when T_{max} was originally pruned. Since the set of subtrees established from each $\mathcal{L}_{(v)}$ was generated without knowledge of the cases in $\mathcal{L}_v$, the cases in $\mathcal{L}_v$ can serve as an *independent test sample* for those subtrees. Now, for fixed α, $\mathcal{L}_v$ is dropped down the corresponding $T_{(v)}$ each of which yields $N_{ij(v)}$, the number of class j cases in $\mathcal{L}_v$ classified as belonging to class i. Summing across all V, the number N_{ij} of misclassified class j cases is obtained for all i,j. Since the total number N_j of class j cases in *all* test samples is identical to the total number of class j cases in $\mathcal{L}$ then the probability that a class j case is misclassified is estimated by $\frac{N_{ij}}{N_j}$ and a *cross-validation* estimate of misclassification rate is obtained by:

$$\hat{R}^{cv}(T_\alpha) = \sum_{i=1}^{C} \sum_{j=1}^{C} \frac{N_{ij}}{N_j} \tag{3}$$

Finally, 3 is calculated for each α, α_{opt} is selected as that α for which 3 is a minimum, and T_{opt} is selected from the sequence of minimizing subtrees grown from T_{max} as that subtree corresponding to the optimal complexity value α_{opt}.

3 Implementation

There are four places of potential parallelism available in CART, dataset, tree, breadth and node. Parallelism at the dataset level would involve generating trees for different datasets at the same time. This is the level with the least amount of parallelism and was not implemented. The other three levels of parallelism were successfully implemented. Tree level parallelism grows the multiple trees used in the cross-validation procedure at the same time. Breadth parallelism expands all of the nodes at the same depth across all trees. Node parallelism involves the calculations done at each node to decide the best split to make.

*CART takes as input a file of classified examples and produces as output a decision tree that can classify new examples. The decision tree tests one feature at a time and based on that selects further tests. When an example reaches a leaf, it is assigned the class that was most likely to be at that leaf for examples in the training data. To achieve this goal, a primary tree and n cross-validation trees are grown using the training data. Each cross-validation tree is grown on $(n-1)/n$ of the data and the error rate of each leaf is estimated by the remaining $1/n$ of the data.

The *CART algorithm has been implemented on a CM-2 in approximately 4500 lines of *LISP.[7] The machine that we used has 8K processors each of which has 64K bits of memory per processor. It did not have floating point hardware. There are CM-2's with more processors and floating point hardware that would be able to execute the same programs in less time and handle larger datasets. The machine is controlled by a *front end* machine that is a conventional workstation. Software executing on the front end machine will typically mix work done on the front end with that done in the CM-2.

Programs in *LISP work by manipulating parallel variables that are known as *pvars*. A pvar has elements of a single type, but that type can be almost any Common LISP type such as integers, floats, arrays or defstructs. Within a pvar, each processor has one element of the specified type. A subset of processors can be selected by testing the data stored in each processor for specific characteristics. Any instructions that are then executed are only applied to those processors that have been selected. There are two broad classes of *LISP operations–those that manipulate the data stored in an individual processor and those that communicate information between processors.

Most of the time spent doing computation on the CM-2 typically involves communication between processors. This is because operations on individual processors take time proportional to the vp-ratio, but communications operations can take much longer. The algorithm will be described in terms of a few primitive parallel communication operators. Each operation is only applied to the subset of active processors. A *Scan* performs some operation across a sequence of processors. A *Rank* gives each processor its rank in the ordered set of values found in all of the processors. In this paper, all of the scans and ranks are segmented which means that they do not propagate values across boundaries. A *Pref* copies a value from another processor into an active processor. A *Pset* writes a value from an active processor into another processor. If more than one processor tries to write into the same processor, the values can be combined in various ways as well. Throughout the rest of this paper, T is the number of trees being grown, C is the number of class assignments, F is the number of features in each example and S is the number of subsets of the catagorical feature with the most values, n. S is calculated as $2^(n-1)-1$.

3.1 Processor Allocation

One of the most critical issues on the CM is how to allocate processors. The pattern of communication between processors is a crucial determinant for an algorithms performance. Even though CMs are available with up to 64K processors, there will always be some problems that require more. To this end the CM software will simulate "virtual processors" by dividing the memory of one physical processor between multiple virtual processors. The ratio between the number of virtual processors and the number of physical processors is known as the *vp-ratio*. Since different parts of an algorithm may require different vp-ratios, or different communication patterns, multiple layouts are allowed. These layouts are known as *vp-sets*. *CART uses two different vp-sets. The *example vp-set* is used to store the training vectors and classifications from the dataset. The *summary vp-set* stores the information needed to find the best split at each node.

Figure 2 shows a very simple dataset that is used to demonstrate how *CART works. The example vp-set is laid out as a vector or processors. Each example in the dataset is assigned a block of processors with one processor for each of the features in the example. Each processor within a block

Example	*Class*	*N1*	*N2*	*C1*	*C2*
0	A		2.5	D	Y
1	B	1.5	2.0	E	N
2	C	1.0	2.5		Y
3	B	1.0	2.0	E	N
4	A	1.0	2.0	F	Y

Figure 2: Example Dataset

Example	*0*				*1*				*2*			
Feature	*N1*	*N2*	*C1*	*C2*	*N1*	*N2*	*C1*	*C2*	*N1*	*N2*	*C1*	*C2*
Processor	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
CLASS	A	A	A	A	B	B	B	B	C	C	C	C
NUMBER-P	NIL	T	NIL	NIL	T	T	NIL	NIL	T	T	NIL	NIL
NUMBER		2.5			1.5	2.0			1.0	2.5		
SET-P	NIL	NIL	T	T	NIL	NIL	T	T	NIL	NIL	NIL	T
SET			D	Y			E	N				Y

Figure 3: Layout of Example Processors

contains the value associated with that example's feature. The layout for the first three cases in the example is shown in Figure 3.1. *CLASS* is the class assignment for each example. It is the same for every processor in the block of processors assigned to an example. *NUMBER-P* is T for each processor that contains a valid numeric entry. It is NIL if the value is missing or if the feature is a catagorical one. *NUMBER* contains numeric feature values. *SET-P* contains T for each processor that contains a catagorical feature and NIL if a specific value is missing or for numeric feature processors. *SET* contains catagorical feature attributes for processors with *SET-P* being T. The identification of a set attribute is done with a binary number with one bit for each possible feature attribute. This is done to make the correspondence of feature attributes to subsets of feature attributes easy to calculate.

The summary vp-set is also laid out as a vector of processors as shown in Figures 3.1, 3.1, 3.1 and 3.1. The processors are arranged in a hierarchical fashion. At the finest level, each processor is associated with the combination of a class assignment, and a feature value. The C processors with the same

Summary	0											
Feature	*N1*						*N2*					
Value	*1.0*			*1.5*			*2.0*			*2.5*		
Class	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
Processor	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
CLASS-COUNT	1	1	0	0	1	0	1	2	0	0	0	0

Figure 4: Summary Numeric Layout

Summary	0								
Feature	*C1*								
Value	*D*			*DE*			*DF*		
Class	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
Processor	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
CLASS-COUNT	0	0	0	0	2	0	1	0	0

Figure 5: Summary Layout for C1

feature value make up a *split*. There is a split for every unique value found in every feature and for catagorical features for every subset of values unless there are more than 32 subsets in which case each unique value is used as a split. All of the processors that have the same feature make up a *feature* and all of the feature blocks make up a *summary*. *CLASS-COUNT* show the initial values found in each processor for the number of examples in the dataset that have that feature, value and class combination.

In the example vp-set, each processor has a pointer to the node it is in every tree and for catagorical features to each subset split that the example

Summary	0								
Feature	*C1*								
Value	*E*			*EF*			*F*		
Class	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
Processor	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*
CLASS-COUNT	0	2	0	1	2	0	1	0	0

Figure 6: Summary Layout for C1 Continued

Summary	0					
Feature	*C2*					
Value	*Y*			*N*		
Class	*A*	*B*	*C*	*A*	*B*	*C*
Processor	*30*	*31*	*32*	*33*	*34*	*35*
CLASS-COUNT	1	0	0	0	2	0

Figure 7: Summary Layout for C2

feature value is a part of. Looping over these pvars is still a serial part of the *CART algorithm. The loops could be done in parallel, but that would require duplicating the dataset. For small datasets, that would be a good thing to do, but for large datasets it would not be feasible. The current implementation does not duplicate datasets.

There are many more pvars in each vp-set than those shown in the layouts. In order to make the actual process of generating trees run as fast possible, as much as possible is pre-calculated and stored. Examples include markers for the start and end of each processor block, pointers to the processor address of the first processor in each block and places to put various useful intermediate values.

The number of processors in the example vp-set is the first power of two larger than the number of physical processors and greater than the number of features times the number of examples. The number of processors in each summary block is the number of unique feature values times the number of classes. The total number of processors in the summary vp-set is the next power of two greater than the two times the number of tree times size of a summary block. There are as many summary blocks as will fit in this size.

I experimented with two schemes for allocating nodes to summary blocks. In the first scheme, I would dynamically expand the vp-set for summary blocks so that it would be large enough to expand every node available in parallel. This did not work very well because expanding the vp-sets took too much time. What I settled on was to allocate the maximum of two summary blocks per tree or the number of summary blocks that would fit into the number of physical processors. These summary blocks are then dynamically assigned to tree nodes while running.

3.2 The *CART Algorithm

Initially each tree root is assigned a summary block and the subset of examples that are to be used in growing or testing each tree are marked. Figure 8 shows a high level description of the *CART algorithm. Each of the steps is gone through in much more detail.

The first step in building a decision tree is to get the data into the front end. *READ-FILE* reads in a sample dataset and determines the number of samples, features, value types and so on. The data is formatted in a fashion that allows it to be put in the CM-2 relatively quickly.[4] *INIT-ROOTS* then allocates the example and summary vp-sets, downloads the data and initializes all of the auxiliary pvars. After this process, the front-end will have a root node for the primary tree and for each of the cross-validation trees.

Most of the work in *CART is spent splitting nodes in all of the trees. This process continues as long as there is still at least one node that consists of a mixture of example classes. During most of the time spent growing trees, there is much more work to do than there are processors available.

COMPUTE-SUMMARIES is the first step in the process of generating trees. It propagates the information from the example vp-set that the processors in the summary vp-set need to determine the optimal splits. For the S possible subsets and the T possible trees, example processors use a pset to increment the summary block processor that corresponds to its combination of feature, value and class. If an example is not part of a trees training data, it increments an error count instead.

Each split in the summary block represents a possible partition of the input space. For a numeric split, examples with a feature value less than or equal to the split value are considered in the partition. For catagorical splits, examples that have a catagorical value in the subset represented by the split are in the partition. To calculate the number of examples in each partition, C parallel operations are done, one for each possible class. The number of examples in each partition is initialized to the class counts for each split. For catagorical partitions, this is the number of examples in that partition. For numeric feature partitions the number of examples are calculated by a forward additive scan. The total number of examples in each

[4] It takes twelve times as long to get stuff into the machine as out of the machine!

- READ-FILE: Read in the dataset
- INIT-ROOTS: Initialize the example and summary vp-sets
- Repeat while impure nodes:
  - COMPUTE-SUMMARIES:
    - Pset example counts into summaries
    - For each class C:
      - Examples in partition by additive forward scan
      - Total examples by backward scan
      - Examples not in partition by difference
  - COMPUTE-IMPURITIES: $\Delta\iota(s,\tau) = \iota(\tau) - \iota(\tau_R) - \iota(\tau_L)$
  - REMOVE-SPLITS: Remove one class, empty and full partitions
  - UPDATE-NODES: Store summary values in front end and terminate if done
  - PRIMARY-SPLITS:
    - Rank splits by impurity and find primary
    - Find next closest numeric value for each split
  - SPLIT-NODES: Split nodes on front end and allocate new summaries
  - SPLIT-SUMMARIES: Assign summary block to each processor in primary split
  - MARK-EXAMPLES: Mark each example processor with in/out primary partition
  - FIND-SURROGATES: Rank alternative partitions with accuracy on primary partion
  - RECORD-SPLITS: Record partitions and rank in front-end
  - SPLIT-EXAMPLES: Set each example processors new summary
- OPTIMAL-TREE: Generate cross-validated tree

Figure 8: *CART algorithm

- Calculate $p(j|\tau)P(j)C(j)$
- Scans segmented on feature values:
  - $p(\tau)$ forward additive scan
  - Copy by backward copy scan
  - $\sum(p(j|\tau)P(j)C(h)/p(\tau))^2$ by backward additive scan
- $\iota(\tau) = (1 - \sum_{j=1}^{C}(p(j|\tau)P(j)C(j)/p(\tau))^2)p(\tau)$

Figure 9: Computation of Gini ι

class is stored by a backward copy scan of the final processor result in the additive scan. The number of examples not in each partition is calculated as the difference between these two values. The error rate of cross validation trees is calculated by another forward additive scan and a backward copy scan. If there are missing values, an additional max scan and copy scan for each class is done to find the total number of examples for each class present in the summary. Finally an additive scan marks any split that has no examples in it.

Given a set of class counts for each potential partition, the impurity of that partition can be calculated according to the Gini index of diversity. Figure 9 shows this process. To make the index easier to calculate in parallel, and to include prior class probabilities $P(j)$ and class misclassification costs $C(j)$, Equation 2 is rearranged as:

$$(1 - \sum_{j=1}^{C}(p(j|\tau)P(j)C(j)/p(\tau))^2)p(\tau) \tag{4}$$

The product $P(j)C(j)$ is calculated in the initialization phase. If no $P(j)$ is specified, it is just $N(j)/N$. If no $C(j)$ is specified, it defaults to one. Initially $p(j|\tau)P(j)C(j)$ is calculated for each summary block processor by multiplying the appropriate class count times the prior estimation $P(j)C(j)$. Scans are segmented to the processors associated with a single partition. $p(\tau)$ is calculated by a forward additive scan and then propagated by a backward copy scan. A final additive backward scan on $(p(j|\tau)P(j)C(j)/p(\tau))^2$ performs the summation and the impurity ends up in the first processor of each possible split.

COMPUTE-IMPURITIES calculates the change in impurity for each possible partition. This is done by calculating first the impurity of a summary

block with no partition and then subtracting the impurity of the examples in and out of each partition. Because of missing values, partitions can end up decreasing impurity without actually splitting any of the data. To remove these partitions, *REMOVE-SPLITS* does a forward additive scan segmented on possible partitions is done to see if only one class is split and two additional conjunctive scans are done to determine if a partition does not split the examples.

For each active node, *UPDATE-NODES* extracts the class counts, error counts and impurity with prefs from the CM-2 and stores them in the front end. If every node has examples in only one class, the tree growing process stops and the tree is pruned using the information stored in the front end about each node.

PRIMARY-SPLITS finds the partition to actually make for every node. This is done by ranking the impurity of the partitions found in each summary block. The value for numeric partitions needs to be half way in between the selected partition value and the next partition for that feature that still has examples in it. To find the next value, a backward min scan on grid addresses is done followed by a general pref.

SPLIT-NODES splits every active node on the front-end by allocating the same summary block for examples that will be in the partition and allocating a new summary block for the examples that will not be. If there are no free summary blocks available, then an address that cannot be a summary block is used. To propagate the new allocation to the current summary block, the in partition summary block address and out of partition summary block addresses are stored in the first processor of each summary block. All of the addresses are then propagated to the other processors in each summary block by a forward copy scan segmented on summaries.

SPLIT-SUMMARIES assigns to each processor in the split features either the summary block address for examples in the partition or summary block address for examples out of the partition. The address of the processor with the partition is copied to the start of each summary block and then propagated using a summary segmented forward copy scan.

If the primary split is numeric, each processor in the primary split feature, compares its value using a pref to that from the primary split. If is less than or equal, it keeps the address of the new in partition summary block. Otherwise it keeps the address of the out of partition summary block. If the primary split is catagorical, all feature processors that have bits in common with the

primary split as gotten by a pref, keep the old summary block address. All other subsets keep the new out of partition block address.

In order to find surrogate splits that can be used in place of the primary split when the primary split feature is missing, *MARK-EXAMPLES* marks each example with the summary block that the primary split would send that example too. This is done by looping through the possible trees and for each example that is still active in that tree, using pref to select the example processor that points to the primary split. That example processor then does a pref of the summary processor to get whether or not the example will be in the partition. A pset of that bit to the start of each example followed by an example segmented copy scan will mark every processor for whether or not it is in the partition of each tree.

After examples are marked, *FIND-SURROGATES* checks the correspondence of every possible split with how well it matches the primary split. While conceptually simple, this is a very complicated process! For this procedure only the first processor in each split will contain values. To initialize each split, the first step is to calculate the total number of examples present in each summary block by a value segmented backward additive scan. The next step, is to calculate the total number of examples that each partition would contain if it were the primary split. This is done by a value segmented backward additive scan on the class counts calculated when computing the primary split. For each subset and for each tree, the example processors use an additive pset to increment the summary processors by one if they are in the primary split partition and zero otherwise. At this point the first processor in each split has a count of the number of examples with that combination of feature and value that were in the primary partition. A feature segmented additive scan is used for numeric features to get the total number of examples that are in both the primary partition and the surrogate partition. The total number of examples that are in the primary partition are stored in each processor by a summary segmented backward max scan and a forward copy scan. The number of examples each split gets right then is the total number of examples in both the primary partition and the surrogate partition plus the number of examples outside of both partitions. If a surrogate partition gets less than half the cases right, then the mirror partition is used where the direction of the test on a split is reversed.

At this point every surrogate split (or its mirror) has the number of primary split examples it gets right. The best surrogate split within each

feature is selected by using a feature segmented rank. A final summary segmented ranking ranks all of the surrogate splits for each feature. At this point the rank of a features surrogate split and its mirror flag split is copied to the start of each feature and propagated to all of the processors in each feature by a feature segmented copy scan.

Once the primary and surrogate partitions have been selected, all of the partitions are read out of the machine and stored in the front end by *RECORD-SPLITS*. The information stored includes the splits, their ranking, and polarity.

Finally, all of the examples must be connected to their new summary blocks by *SPLIT-EXAMPLES*. If there are no missing values in the dataset, this is done by looping through each tree and example processors that point to primary split summary processors copy the new summary block for that example to the first processor in each example. The new summary block pointer is then copied to the entire example by an example segmented forward copy scan.

If there are missing values, SPLIT-SUMMARIES is called again to include the summary block that surrogate splits would direct examples to. Looping through each tree, every example processor gets the rank of the partition for its feature using pref. Missing values are then set to the last rank. An example segmented rank then determines the split with the best rank. The new summary block for each example is then copied to the start of each example and propagated to the other processors by an example segmented forward copy scan.

At this point all example processors have been assigned their new summary blocks. If the summary block is not on a summary block boundary, the example processors are marked as inactive until a summary block can be allocated to those examples. This happens whenever a node ends up with examples all from one class.

One cycle of processing is now complete. The process will continue until there are no nodes that contain a mixture of example classes. At this point, the primary tree is pruned back using the information stored in the cross-validation trees by *OPTIMAL-TREE*. This process is done entirely in the front end since it is not an expensive process.

Dataset	*Cases*	*Classes*	*Numeric*	*Catagorical*	*Speed up*
Iris	150	3	4		8
Digit	200	10		7	34
Terrain	320	2	34		120

Figure 10: Speed Up

If SURMT14 $\leq$ 0.003 then
  If CCMDR1 $\leq$ 1.0 then ((YES 0.03) (NO 0.97))
  else (NO)
else
  If SURMT9 $\leq$ 0.00006 then (YES)
  else (NO)

Figure 11: Final tree for Terrain

4 The Performance of *CART

To test the performance of *CART, several datasets were run on both a serial version of CART running on a SUN-4 and *CART running on a SUN-4 connected to an 8K CM-2. The datasets that were used are shown in Figure 10. The *Iris* dataset is a classic machine learning example dating back to Fisher.[4] The task is to use plant measurements to induce the type of iris. The *Digit* dataset is based on a probabilistic model of a faulty calculator. [3] Each case is labeled with a digit from 0 to 9 and the appropriate segments of a calculator display is in its proper state some percentage of the time. The *Terrain* dataset contains terrain features from locations that have been labeled with their suitability for running tanks over.

One primary and ten cross-validation trees were grown for each example. Figure 4 shows the final tree from the *Terrain* dataset. As can be seen, it has found a very simple description for the data. The numbers in the leaves are the probability that the classification is correct for that class. The speed ups over the serial version range from 8 for the iris data up to 120 for the terrain data.

Dataset	*Use*	*Allocate*	*VP-Ratio*
Iris	600	8192	1.0
Digit	1400	8192	1.0
Terrain	1280	8192	1.0

Figure 12: Number of Processors Used for Example VP-sets

Dataset	*Summary*	*Number*	*Use*	*Allocate*	*VP-Ratio*
Iris	411	39	16,029	16,384	2
Digit	140	58	8,120	8,192	1
Terrain	10,880	48	522,240	524,288	64

Figure 13: Number of Processors used for Summary VP-Sets

The number of example processors used by each dataset is shown in Figure 4. *Use* is the number of processors that are actually used. *Allocate* is the number of processors that were allocated. *VP-Ratio* is the ratio between the number of processors allocated and the size of the machine. As can be seen, all of the datasets nicely fit in an 8,192 processor machine. With this size of dataset, it would have been good to duplicate the datasets to prevent having to loop through the nodes in each tree.

Figure 4 shows the number of processors used in calculating summaries. *Summary* is the size of each summary block. It is the number of unique feature value combinations times the number of classes. The allocation algorithm requests at least 22 summary blocks, two for every tree being grown. This is used to determine the vp-ratio and then the extra space is filled out with summary blocks. *Number* shows the number of summary blocks actually used.

To get an idea of the number of parallel instructions executed on each cycle of the *CART algorithm, Figures 4 and 4 show the number of each kind of parallel instruction executed. Figure 4 shows the instructions broken down for each example. As can be seen the number of instructions is relatively constant between the problems. The big differences in speed up come because of the number of nodes found in the trees and the work being done to split each node. The biggest speed up is found in the *Terrain* example because it has the most possible splits and also the most complicated trees.

Figure 4 shows the relative amount of time spent in different parts of

Function	*Scan*					*Rank*
	Copy	*+*	*Min*	*Max*	*And*	
COMPUTE-SUMMARIES	$3C$	$2C+1$		C		
COMPUTE-IMPURITIES	3	6				
REMOVE-SPLITS		1			2	
PRIMARY-SPLITS			1			1
SPLIT-SUMMARIES	2					1
MARK-EXAMPLES	T					
FIND-SURROGATES	$T+3$	3		1		2
SPLIT-EXAMPLES	$T+2$					T
TOTALS	$3(T+C)+8$	$2C+11$	1	$C+1$	2	$T+4$

Figure 14: Scans per cycle

Function	*Pref*	**Pset*	
		No Collision	*Add*
COMPUTE-SUMMARIES	$2TS$		$2TS-1$
COMPUTE-IMPURITIES			
REMOVE-SPLITS			
PRIMARY-SPLITS	1		
SPLIT-SUMMARIES	$C+2$	1	
MARK-EXAMPLES	$2T$	S	
FIND-SURROGATES	$2T$	$S+1$	TS
SPLIT-EXAMPLES	$4T$	$S+T$	
TOTALS	$T(8+2S)+C+3$	$T+3S+2$	$3TS-1$

Figure 15: General communication instructions per cycle

Dataset	*Scan*	*Rank*	*Pref*	*Pset*	*Total*
Iris	74	15	116	26	231
Digit	116	15	123	26	280
Terrain	68	15	115	26	224

Figure 16: Parallel Instructions Used Per Cycle

Function	*Percent*	*Calls*	*Seconds*
COMPUTE-SUMMARIES	0.24	237	0.30
SPLIT-EXAMPLES	0.14	113	0.35
FIND-SURROGATES	0.12	113	0.31
MARK-EXAMPLES	0.12	113	0.31
ALLOCATE-SUMMARIES	0.11	1	32.19
RECORD-SPLITS	0.07	113	0.19
COMPUTE-IMPURITIES	0.06	237	0.07
PRIMARY-SPLITS	0.05	113	0.13
READ-FILE	0.03	1	7.68
SETUP-EXAMPLES	0.02	1	6.07
UPDATE-NODES	0.01	237	0.01
ALLOCATE-DATASET	0.01	1	2.42
SPLIT-SUMMARIES	0.01	113	0.02
OPTIMAL-TREE	0.00	1	0.90
INIT-ROOTS	0.00	1	0.15

Figure 17: Terrain *CART monitoring

*CART for the *Terrain* dataset. The actual clock time *CART takes for this dataset is about 3.5 minutes versus over seven hours for the serial version! Figure 18 shows a break down of the front-end instructions to the CM-2. As would be expected various scans and moves are high on the list. More surprising are the instructions having to do with allocating and deallocating stack space.

5 Conclusion

Implementing *CART on the CM-2 was a challenging task. The serial algorithm is complicated and doing it efficiently on the CM-2 was even more complicated. The compiler was difficult to work with and I had to write monitoring tools to get an idea of the relative efficiency of various approaches. The main thing left to help performance on small datasets would be duplicating datasets to prevent looping for trees and subsets. It would also be useful to run some experiments on a larger machine and on a machine with hardware floating point. *CART sucessully implements all of the features

Function	*Percent*	*Calls*	*Seconds*
CM:DEALLOCATE-UPTO-STACK-FIELD	0.13	216739	0.00133
CM:SEND-WITH-S-ADD-1L	0.12	6220	0.04031
CM:ALLOCATE-STACK-FIELD	0.09	391818	0.00048
CM:DEALLOCATE-STACK-THROUGH	0.08	317298	0.00057
CM:SEND-WITH-OVERWRITE-1L	0.04	2486	0.03498
CM:MOVE-ALWAYS	0.04	132508	0.00057
CM:SET-VP-SET	0.03	149818	0.00049
CM:SCAN-WITH-F-ADD-1L	0.03	1422	0.04676
CM:UNSIGNED-RANK	0.03	340	0.16763
CM:MOVE	0.03	92120	0.00059
CM:SCAN-WITH-COPY-1L	0.02	5230	0.00799
CM:UNSIGNED-NEW-SIZE	0.02	71134	0.00058
CM:ENUMERATE-AND-COUNT	0.02	10371	0.00390
CM:MOVE-CONSTANT	0.02	68089	0.00058
CM:UNSIGNED-RESIZE-FIELD-ALWAYS	0.01	5458	0.00511
CM:MOVE-CONSTANT-ALWAYS	0.01	43849	0.00055
CM:GET-1L	0.01	5085	0.00409
CM:UNSIGNED-ADD	0.01	29748	0.00067
CM:MY-CUBE-ADDRESS	0.01	5341	0.00319
CM:DEPOSIT-NEWS-COORDINATE-1L	0.01	6690	0.00246
CM:GEOMETRY-SEND-ADDRESS-LENGTH	0.01	33461	0.00045
CM:GLOBAL-LOGIOR	0.01	27687	0.00055
CM:F-EQ-CONSTANT-1L	0.01	4046	0.00375
CM:ENUMERATE	0.01	3946	0.00334
CM:S-RESIZE-FIELD-ALWAYS	0.01	2679	0.00492

Figure 18: Terrain CM-2 Monitoring

of the CART algorithm including: catagorical and numeric features with missing values, prior class distributions, class misclassification costs, cross validation and feature ranking. In addition it exhibits substantial speed ups over a serial version and should scale nicely to bigger machines.

Thanks to Stuart Crawford for introducing me to CART, ADS for letting me use their CM-2 and Aileen Doherty for all of her support.

References

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

[2] D.J. Cook and L.B. Holder. Accelerated learning on the connection machine. In *Proceedings Second IEEE Symposium on Parallel and Distributed Processing*, pages 448–454, Dallas, TX, 1990.

[3] Stuart L. Crawford. Extensions to the cart algorithm. *International Journal of Man-Machine Studies*, 1989.

[4] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(1):179–188, 1936.

[5] T.P. Hong and S.S. Tseng. Models of parallel learning systems. In *11th International Conference on Distributed computing Systems*, pages 125–132, Arlington, TX, 1991.

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

[7] Thinking Machines Corporation, Cambridge, MA. **Lisp Reference Manual*, 1989.