Parallel Induction Systems based on Branch and Bound

Ken Jung and Evangelos Simoudis,
Lockheed Artificial Intelligence Center,
3251 Hanover Street, Palo Alto CA 94304.
kkj.simoudis@aic.lockheed.com

Ramesh Subramonian*
Division of Computer Science
University of California, Berkeley
Berkeley, CA 94720
subramon@cs.berkeley.edu

Abstract

Symbolic inductive learning systems explore a space of hypotheses to derive the definition of concepts. Due to the size and complexity of this space, sequential algorithms use various heuristics that limit the classification accuracy of the generated definitions. Parallel search algorithm can rectify such limitations. In this paper we present the PARIS parallel symbolic inductive system. PARIS uses the branch and bound search algorithm. It has been implemented on the Thinking Machines CM-5 and the MasPar MP-1 computers, and has been tested on the domain of financial analysis.

1 Introduction

The goal of an inductive learning system, either symbolic or neural, is to derive the definition of a concept (called the target concept) after being presented with a set of examples (called the training set) of this concept. In symbolic inductive learning, the derivation process can be formulated as a search through a space of hypotheses, where each hypothesis is a particular definition of the target concept. The size of the hypothesis space is related to the concept's complexity. Obviously, the more exhaustive the search through this space, the more accurate the definition of the target concept. Unfortunately, with sequential computers this is not always possible because of either space or time constraints. As a result, inductive learning systems that have been implemented on sequential computers (e.g. ID3 [Qui83], COBWEB [Fis84]) use heuristic search algorithms (e.g., hill climbing) to address such constraints. In this paper we present PARIS, a parallel version of the OTIS supervised symbolic inductive learning system [Ker88, Ker91], that uses the branch and bound search algorithm. PARIS has been implemented on two massively parallel computers, a Thinking Machines CM-5 and a MasPar MP-1, and has been tested on the domain of financial analysis.

The paper is organized as follows. In Section 2, we present a brief overview of symbolic induction and describe OTIS. In Section 3, we review the branch and bound algorithm and describe how it is used in OTIS and PARIS. In Section 4, we describe two approaches we have developed for parallelizing the branch and bound algorithm. The implementations of these approaches on the massively parallel hardware are presented in Section 5. In Section 6 we present our results from applying PARIS in a financial domain. Our conclusions are presented in Section 7.

2 Inductive learning systems

An inductive learning system is described in terms of:

1. A language for describing examples. Each example is a set of attribute/value pairs. Attributes are divided into predictor attributes and goal attributes. In the example shown in Figure 1, the predictor attributes are: company-name, 13-week-%-change, quarterly-earnings, and industry-type, whereas the goal attribute is: company-strength.

2. A language to describe the induced concepts. This is also called the bias. As part of an inductive learning system's specification is the representation in which the concepts are expressed given a particular bias, e.g., decision trees, rules, or class hierarchies. For example, a concept can be described as a conjunction of a set of features. A
(rule1 (> 13-week-%-change 0) (industry-type auto-industry) → (company-strength A))

Figure 2: An induced rule

(rule1 (> 13-week-%-change 0) (industry-type auto-industry) (quarterly-earnings 245M) → (company-strength A))

Figure 3: A specialization of the rule1

3. An algorithm for searching the hypothesis space and thus implementing the bias. For example, hill climbing, branch and bound, and gradient descent are commonly used search algorithms.

4. A matching predicate to evaluate how well each hypothesized definition of the concept being learned can predict the values of the goal attributes of previously unseen examples.

The inductive learning task proceeds as follows. The inductive learning algorithm receives a set of examples, each of which is labeled as being an instance of a particular concept. Given a particular bias and the algorithm that implements this bias, the inductive learning algorithm tries to automatically create definitions for these concepts so that previously unseen examples can be correctly assigned to the corresponding concept.

2.1 OTIS

The inductive learning system OTIS accepts examples that are described as sets of attribute/value pairs, defines concepts as conjunctions of features, and represents a concept as rules. OTIS uses a variant of the branch and bound algorithm to search the hypothesis space. During the search process, OTIS either proposes a new hypothesis (rule) or modifies an existing one. There are three basic ways in which hypotheses are modified: specialization, generalization and constructive induction. Specialization involves making a hypothesis more specific. For example, the rule of Figure 2 can be specialized to the one shown in Figure 3.

Generalization is the opposite. Constructive induction is the process of extending the original concept description language by creating new descriptors. For example, the term "electronics-industry" could be created from the values "computer-industry," and "semiconductor-industry" of the attribute "industry-type."

Mitchell’s formulation of the inductive task requires that solutions be consistent. This requirement implies that (1) the training instances contain no errors; and (2) it is possible to formulate a correct description of the target generalization within the given generalization language. OTIS deviates from this requirement, allowing instances to be misclassified and allowing for the fact that there may be no consistent solution. Experts are not always consistent in classifying instances, especially border line calls, giving rise to the need for second opinions! An induction system that does not allow for such errors might be overly optimistic in its requirements on the training set.

As was mentioned above, inductive algorithms need to be parallelized in order to address time and space issues associated with the induction task. In particular, inductive algorithms must be able to search larger hypothesis spaces, deal with data sets of ever increasing size, and perform faster, at least, the induction operations. While the operations performed by neural networks have been parallelized, symbolic induction algorithms have not yet been parallelized. Symbolic induction algorithms, however, have three advantages over their neural network equivalents. First, the concept definitions produced by symbolic induction systems are understandable by humans and can be used by a problem-solving component to explain how it reached its classification decisions. Second, symbolic induction systems can accept examples that are described in terms of both nominal and numeric values. Finally, symbolic induction systems can utilize explicit knowledge to facilitate the induction process by expediting it and/or increasing the accuracy of each concept's definition.

2.2 PARIS

We have developed the PARIS symbolic inductive learning system that is based on the OTIS system in that it uses the branch and bound algorithm to search the hypothesis space. PARIS accepts examples that are described as vectors of attribute/value pairs, defines concepts as conjunctions of features. An instance is represented as a vector $C_i$ is a vector $C_i = \{ v^i_1, v^i_2, ..., v^i_j \}$ where $v^i_j \in V_j$, with $V_j$ being the set of values that attribute $j$ takes in the members of the training set. A definition of a concept is similar to an object except that (i) $v^i_j \in V_j \cup \{?\}$ where ? is the "don't care" symbol and (ii) the goal attribute is set to a given value, which represents the concept being learned. The heart of PARIS is a parallel implementation of the branch and bound algorithm we describe in the next section. PARIS can currently define with the definition of one concept at a time.
Similar to OTIS, PARIS does not attempt to find a consistent definition of the target concept. This requirement presents a problem since the quality of the created definitions cannot be established in absolute terms. For this reason we associate a cost function with each concept definition. The cost of a concept definition is the fraction of false positives and false negatives produced after using this definition on the members of the training set. False positives and false negatives are weighed differently since usually the relative cost of these two types of errors, called $\alpha_{pos}$ and $\alpha_{neg}$, are different. The values for $\alpha_{pos}$ and $\alpha_{neg}$ are determined by a domain expert. Given a training set and a target concept (goal attribute), PARIS tries to induce the minimum cost definition of the target concept.

PARIS' bias prefers the most general concept descriptions. For this reason, whenever necessary, the cost function can include the term $\alpha_L \times L(C)$, where $L(C)$ is the length of the concept definition, defined as the number of precisely specified values, i.e., no "don't care" symbols. The precise value of $\alpha_L$ is problem-dependent.

As an example, a training set might include the instances (Red, Fruit, Light) and (Red, Vegetable, Heavy). A target concept may be of the form (Red) and a definition of that concept might be (Red, Fruit, ?). What this means is that when presented with a new instance, if the second attribute is Fruit, then it is classified as Red; else, it is classified as non-Red.

3 The Branch and Bound Algorithm

Since Branch and Bound is a well-known heuristic search algorithm, we describe it briefly here (see [Bal85] for details). The algorithm has two prerequisites. First, it must be possible to specify the cost of both the partial and the complete solutions to the problem being solved. Second, the cost function must be monotonic, i.e., the cost of all extensions of a problem's partial solution should always be greater than or equal to the partial solution being extended. Therefore, the cost function provides a lower bound on the cost of all complete solutions that can be generated from each partial solution.

The branch and bound algorithm evaluates the cost of the set of available partial solutions. It selects the solution with the lowest cost (Bounding step), and derives a new set of partial solutions (branching step). If the solution to the problem being addressed is not included in the set of derived solutions, the branching and bounding steps are repeated. The algorithm maintains the minimum cost of a complete solution (the bound), that is initially set to infinity. Every partial solution whose cost is greater than the maintained bound is removed from the set of partial solutions. An efficient implementation of the branch and bound algorithm requires the use of a heap.

3.1 Branch and Bound in OTIS

OTIS searches the hypothesis space using the "best-first" variation of the basic branch and bound algorithm. OTIS maintains in an agenda the list of tasks associated with modifying a concept's definition. Each task is assigned a score by the agenda evaluation function. This function takes into account the classification accuracy of each definition (rule) on the training set, and the conciseness of each definition (recall that the general definitions are preferred over the more specific ones). During each iteration, the task with the highest score is removed from the agenda and executed resulting on a new definition. Every new definition is first evaluated using a set of heuristics, and is either added to the agenda or discarded based on the evaluation's results. This process terminates when the agenda is empty or the user decides to terminate the search.

3.2 Branch and Bound in PARIS

PARIS uses two cost functions. The first, provides the cost of a complete definition $C_l$ (complete solution) and is defined as:

$$\text{Cost}(C_l) = \sum_{j=1}^{n} f(G_i, I_j)$$

where $G_i$ is the goal attribute, $I_j$ is the name of $j$th example in the training set, $f(C_l, I_j) = \alpha_{neg}$ if $G_i \neq G_j \land \exists i, s_i \neq \emptyset : s_i \neq a_j$

$$f(C_l, I_j) = \alpha_{pos}$$

The second cost function establishes the cost of a partial solution so that it meets the monotonicity requirement. For this reason we introduce the symbol * indicating that a value has not been assigned yet to a particular attribute. Therefore, a partial solution is one with at least one of its attribute values being *. The partial solution cost evaluation function charges for (1) attributes whose values have been set, i.e., whose values are neither ? nor *, and (2) for only the false negatives the solution (definition) predicts.

The space of hypotheses is organized as a ternary tree. The root of this tree is the partial solution (*, *, ..., *). The leaves of this tree, i.e., complete concept definitions, have values (including ?) for all the attributes. The interior nodes of the tree are the partial definitions of the concept being induced.
In order to avoid examining every member of the training set during the evaluation of each partial definition, PARIS maintains with each partial solution the examples of the training set that matched the definition, and a count of the false negatives and true negatives in this subset. This information requires \( O(|TS|) \) bits, where \( TS \) is the training set.

4 Parallelization of Branch and Bound

We have developed two ways to parallelize the branch and bound algorithm. The first is based on a Lookahead approach, while the second, that is theoretically optimal, is based on a Randomized approach [KZ88]. In each case we assume that the host element (host) of the parallel computer communicates with computer's Processing Elements (PEs). The host maintains all complete concept definitions in a heap. The host also maintains the parameter bound whose value is the cost of the cheapest complete concept definition identified during the computation. Each PE maintains partial definitions in a local heap.

4.1 Lookahead Approach

The ability to look ahead assumes that the search tree has regular structure. As a result, this approach can only be used when the structure of the search tree is predictably regular.

Given a particular partial solution (node in the search tree) that is selected during the bounding step, \( p \) new partial solutions can be generated at a depth of \( \lfloor \log_2 p \rfloor \), or \( \lfloor \log_2 p \rfloor + 1 \), from that node. The value of \( p \) is defined by the user. Therefore, the PEs are used to look ahead from the most promising partial solution.

For example, if the most promising partial solution is \((T,*,*,*)\), and \( p \) is set to nine, then the following solutions (partial or complete) are generated during the lookahead operation: \((T,T,*,*)\), \((T,T,F,*)\), \((T,T,?,*)\), \((T,F,T,*)\), \((T,F,F,*)\), \((T,F,?,*)\), \((T,?,T,*)\), \((T,?,F,*)\), \((T,?,?,*)\).

The lookahead version of the branch and bound algorithm consists of the following steps. First, the host broadcasts to the PEs the best partial definition (solution), initially \((*,*,*,*)\). Each PE expands differently the definition it receives, obtaining a new definition, and calculates the definition's cost. If the cost is greater than the value of the bound parameter, the definition is discarded.

If the obtained solution is a complete definition of the target concept, the PE sends it to the host. The host inserts the received definition in the heap and updates the value of the bound parameter, if the cost of the received definition is lower than the parameter's value. If the obtained definition is a partial, then it is added to the PE's local heap.

The least expensive of all available partial solutions is then identified. The cost of this solution is compared to the value of the bound parameter. If the solution's cost is greater than the value of the bound then the computation terminates. Otherwise, the partial solution is deleted from the local heap where it is stored and is communicated to the host and the branching operation is repeated. Computation terminates when either all the local heaps are empty, or the user signals termination.

The Lookahead approach has two advantages. First, it does not require arbitrary inter-processor communication. Second, the search is localized around the most promising partial solution. A disadvantage of the approach is that for certain traversal patterns its efficiency becomes \( \frac{p \log d}{d \log p} \), where \( p \) is the number of PEs and \( d \) is the depth of the sub-tree being explored during the lookahead operation. However, we have not been able to establish through our experimentation that patterns that can cause this problem actually occur in practice.

4.2 Randomized Approach

The natural parallel extension of sequential "best-first" search is "global best-first." Global best-first search requires the simultaneous creation of all \( p \) partial solutions of minimum cost. This requirement guarantees that the least expensive complete solution will be found with the minimum number of partial solution expansions. However, global best-first search also requires a shared heap that can simultaneously supply each PE with a different partial solution. This is a difficult requirement to meet.

In [KZ88] the last requirement of the global best-first algorithm is addressed using randomization. In particular, each PE uses the "sequential best-first" algorithm to expand all partial solutions that are stored in its local heap. Each time a PE creates a new partial solution, it sends it to a random PE. Therefore, during each branching step, a PE expands the root of its local heap, evaluates the cost of the generated solutions, and sends each solution whose cost is lower than the value of the bound parameter to a random PE. After the solutions are transmitted, the PE inserts all partial solutions it has received from other PEs into its local heap. This process is repeated until all local heaps are empty.
5 Implementation

We have implemented the lookahead approach on both the Thinking Machines CM-5 and the MasPar MP-1. We have implemented the Randomized approach on the MP-1. In this section we provide details of our implementations discussing issues relating to these implementations.

Issue 1: Memory Requirements. The memory requirements are high because of: (i) The “best-first” variation of the branch and bound algorithm has, in general, large space requirements; (ii) the size of the training set. In PARIS' present implementation the training set is maintained in each PE on the CM-5 and on the host on the MP-1; (iii) If one exercises the option to maintain with each partial solution a bit vector which tells us which instances it matches and which it does not, this can further increase PARIS' memory requirements. However, the memory capacity of the two computers we have chosen to implement PARIS satisfy these requirements.

Issue 2: Bandwidth Requirements. The Lookahead approach, while not requiring inter-processor bandwidth, it requires that the host and the PEs communicate quickly. Both the MasPar MP-1 and the TMC CM-5 use dedicated hardware to support the communication operations needed eg., broadcast, reduction (finding the maximum over a set of processors), etc.

In the Randomized approach, one needs high inter-processor bandwidth. The CM-5 provides 4 Mbytes/sec/processor bandwidth for random permutations. The MasPar MP-1 provides high inter-processor bandwidth but only for communication between a PE and its four neighbors. Inter-processor communication is required to balance the heaps of the processors in terms of number and cost of partial solutions. We will need to determine the cost of a communication step as opposed to a node expansion step (dependent on the hardware) and the frequency of communication steps relative to node expansion steps (dependent on the problem) before we can determine whether or not bandwidth will be a limitation.

For the randomized approach, the bandwidth requirements are higher than that needed to support random permutations due to possible contention among the PEs. In particular, if every PE sends a partial solution to some random PE, then, most probably, some PE will receive $\Theta(\frac{\log p}{\log \log p})$ partial solutions, where $p$ is the number of PEs. As a result, the network of PEs will be congested, thus reducing the effective bandwidth.

The MP-1 does not efficiently support random permutations. However, it provides high communication bandwidth between neighboring PEs. We capitalized on this capability and perform the following random mixing strategy. Each communication step is broken up into 4 sub-steps, since there are 4 neighbors/processor. In the first sub-step, all PEs send the root of the local heap to the processor to the left and delete the root of the local heap. The next sub-step is a shift to the right and so on. The mixing strategy is to keep the heaps of the PEs balanced in terms of size and cost of the partial solutions. We use a randomized mixing strategy by randomly permuting the order in which the shifts are made. For example, one communication step might shift in the order (Left, Right, Top, Bottom) whereas another may shift in the order (Right, Top, Left, Bottom). We have not been able to establish through our initial experimentation with PARIS whether the balance will be achieved fast enough for optimal efficiency to hold.

In both the above approaches, if one chooses to maintain side information on th instances which each partial solution matches, then the bandwidth requirements increase substantially.

Issue 3: Randomization versus send-and-receive. The requirement that partial solutions must be sent to random PEs has some interesting ramifications from a programming standpoint. First, it rules out the use of the “synchronous send-and-receive” approach that is available in the CM-5. Instead, we have used the CMAM function library, developed at the University of California, Berkeley [vECGS92]. This library includes functions that implement Active Messages, that utilize the network of PEs more efficiently. An Active Message incorporates with each message a handler that is executed by the PE receiving the message. In this way, the data included in the message is incorporated into the computation performed by the receiving PE with a minimum overhead. The need for message buffering, an expensive requirement, is eliminated. However, the message handler can only (1) include a small piece of data, i.e., the message handler is not a general remote procedure call mechanism, and (2) generate a restricted set of messages in order to avoid deadlock.

The restricted capability of message handlers creates certain problems. In particular, consider the following situation. Processor $p_1$ wants to communicate a partial solution to processor $p_2$. However, due to its size a partial solution must be partitioned among several messages, each with its own handler. Now, how should each such message handler (executed on $p_2$) behave? In particular, the handler can neither allocate space for the contents of the message, nor can it search for space for
the message because its execution is required to terminate quickly.

In order to avoid this problem, \texttt{p1} should send a partial only if it knows the exact location in \texttt{p2}'s local memory where the solution must be copied. This requirement implies that a PE must always keep free space (call \texttt{free spot}) in its local memory that is accessible by all other PEs with which it communicates. Before a PE sends a partial solution, it performs a "destructive read" on a copy of the receiving PE's \texttt{free spot}. A destructive read of \( X \) is an atomic operation that returns \( X \)'s value and sets it to some special value, \#. Since messages are polled for and do not interrupt the operation of the destination PE, atomicity is enforced. Only if the value returned by the destructive read operation is not \#, the transmitting PE can (1) send a partial solution to the target PE, and (2) instruct the message handler where to place the solution.

**Issue 4: Interrupt-driven versus polled messages.** In PARIS' current implementation the PEs must poll the network to determine whether messages are waiting to be processed. It is important for a PE to be able to "pull" outstanding messages off the network. Otherwise, a message congestion will ensue that will affect communication among other PEs as well.

The performance impact of such congestion can be quite serious. It is the programmer's responsibility to insert explicit polling statements in PARIS. As a result, the system may be polling for messages for long periods without sending any messages in the meantime. It has been suggested that the development of interrupt-driven Active Messages would obviate this problem.

**Issue 5: Drawbacks of the cost function.** The quality of a cost function is determined by its ability to prune branches with inappropriate partial solutions. Because of our requirement for a monotonically increasing cost function, a partial solution is not charged for false positives. Since PARIS uses a simple conjunction bias, a positive example of the target concept that is not correctly classified by a partial solution cannot be later re-classified by a more complete definition of the solution. Consequently, depending on the training set, partial solutions being pursued may be rejected only after the value of the solution's last attribute is established.

## 6 Results

### 6.1 Randomized Approach

We present some of our results, which are still at a preliminary stage. These results have been obtained on a 8192-processor MasPar MP-1. We have deliberately refrained from including timing measurements since the parallel code has not been sufficiently tuned to make such a comparison meaningful.

The results show the massive computational, communication and storage requirements of these problems which mandate the use of massively parallel processors to obtain solutions.

In using many processors, the partial solutions are distributed over the heaps of the different processors. If these heaps are reasonably balanced in size, their sizes will be much smaller than if all the partial solutions were maintained in a single heap. This makes accesses to the individual heaps faster.

Let \( c \) be the number of node-expansion steps between communication steps. Ideally, the parallel algorithm should expand no more nodes than the sequential algorithm. We found an interesting tradeoff between between communication and computation over here. As \( c \) increases, communication decreases but computation increases because of poorer load balancing.

Note that even if the number of nodes expanded by the parallel algorithm is the same as the sequential algorithm, this does not give us a complete picture of the speedup since processors might be idling. So, we measured the number of node-expansion steps. Ideally, the parallel algorithm should take \( 1/p \) of the number of expansion steps of the sequential algorithm. Once again, we found that as \( c \) increased, the number of node expansion steps increased.

The previous two results show the computation/communication tradeoff. Another interesting tradeoff is the space/communication tradeoff. As \( c \) decreases, the imbalance of the heaps on the different processors increases, necessitating higher storage costs. Ideally, the size of the heap on one of the parallel processors should be \( 1/p \) of the total number of nodes at any instant of time.

## 7 Conclusions and Future Work

Given a training set of pre-classified examples of a particular concept, an inductive learning system must produce a definition of this concept. Since the space of possible generalizations is large, effective strategies for searching this space are necessary. Our experiences with
the OTIS sequential induction system showed us that the search process is computationally expensive and a good candidate for parallelization. We have developed the PARIS induction system that uses a parallel branch and bound algorithm. We have designed the parallel branch and bound using two different approaches, and have implemented them on a Thinking Machines CM-5 and a MasPar MP-1 computers.

We will continue our work by exploring incorporating biases into PARIS. In particular, we will search for a definition that can explain some fixed fraction of the data. Once such a definition is found, all examples it classifies correctly will be removed from the training set. The order in which the search for such definitions is performed will heavily influence the quality of the overall set of definitions. In particular, a promising partial solution will stop being considered if it happens to match a large subset of the examples that are also matched by an already accepted solution, since these examples will be removed from the training set.

In the current implementation of PARIS, the training set is duplicated on every processor on the CM-5 or maintained on the host on the MP-1. These represent two extremes solutions where one trades off space for speed. An intermediate approach would be to group processors into sets such that each set shares a copy of the training set. The processors in a set will then collaborate in way that will make each set appear as a single processor.

Acknowledgements We would like to thank Tracy Smull and the folks at MasPar for the use of their machine. National Science Foundation Infrastructure Grant number CDA-8722788 is gratefully acknowledged for making available the use of the CM-5. We would also like to thank Randy Kerber for many helpful discussions on the OTIS algorithm.

References


