A PARALLEL CONCEPT LEARNING ALGORITHM
BASED UPON VERSION SPACE STRATEGY

T.P. Hong
Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.

S.S. Tseng
Department of Computer and Information Science
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.

ABSTRACT

In this paper, we attempt to apply the technique of parallel processing to concept learning. We propose a parallel model suited for parallel learning, and a parallel learning algorithm based upon this model by applying the strategy of version space. A comparison is made between the time complexity of our algorithm and sequential learning algorithm, and a quite satisfactory result is obtained - with \( N/2 \) processors, most learning problems can be processed in \( O(\log N) \). Besides, we also study the influence of the arrangement of the training instances on the performance of our parallel algorithm. The final result shows that when the number of the training instances is large, it is worth learning in parallel.

1. Introduction

Learning general concept from a set of training instances has become increasingly important for artificial intelligence researchers on constructing knowledge based systems[1][2][4][5][19]. This problem has been studied by many researchers over the last two decades, and many approaches have been proposed to solve it[11][12]. The generalization strategy adopted can be divided into two classes: data-driven strategy and model-driven strategy[3][15]. The data-driven strategy processes the input examples one at a time, and gradually generalizes the current set of description until a final conjunctive generalization is computed. Therefore, it processes in a bottom-up way. On the other hand, the model-driven strategy processes a set of possible generalizations in an attempt to find a few best hypotheses that satisfy certain requirements by considering the entire set of training instances as a whole. So, it processes in a top-down manner.

No matter which strategy is adopted, its efficiency is limited by its learning speed. Due to the dramatic increase in computing power and the concomitant decrease in computing cost over the last decade, learning from examples by parallel processing becomes a feasible way for conquering the speed problem in learning within a single processor.

In this paper, we discuss the possibility and performance of parallel learning by a data-driven strategy called "version space"[14] and propose a parallel model for solving it. We also analyze the time complexity of parallel learning and get a pleasing result - with \( N/2 \) processors, the parallel learning method proposed in our model can be accomplished in \( O(\log N) \) where \( N \) is the number of training instances. Note that the time complexity with a single processor is \( O(N) \). Therefore, it is worth learning in parallel when the number of training instances is large.

This paper is organized as follows. Some data-driven generalization strategies are introduced in Section 2. A parallel learning model and a parallel learning algorithm are proposed in Section 3. This is followed by the analysis of time complexity in Section 4. In Section 5, we study the influence of the arrangement of the training instances on the performance of our parallel algorithm. Future work is given in Section 6 and the result of this paper is summarized in Section 7.

2. Overview of data-driven generalization strategies

Data-driven generalization strategies can be grouped into three classes: depth-first strategy, specific-to-general breadth-first strategy and version space strategy. A detailed discussion can be found in [13][15]. Here, we only give a brief review about these approaches.

In depth-first search strategy, a single generalization is chosen as the current best hypothesis for describing the training instances which have been observed. This current best hypothesis is then tested against each newly presented training instance, and is altered as needed so that the resulting generalization is consistent with each new instance and all previous instances. Though it keeps only one generalization at any time, it has two main disadvantages.

1. It needs additional cost of maintaining consistency with past instance.
2. It needs backtracking.

In contrast to depth-first search programs, programs which employ a specific-to-general breadth-first search strategy maintain a set of several alternative hypotheses. The set of alternative plausible hypotheses computed is the set of maximally specific generalizations consistent with the observed training instances. We call this set as \( S \) and it is defined as...
S = \{s\} s is a generalization that is consistent with the observed instances, and there is no generalization which is both more specific than s and consistent with the observed instances.

The advantage of the specific-to-general breadth-first strategy over the depth-first strategy is that it need not check past positive instances when the generalization process is done. But it must still check past negative instances to assure that the revised generalization is not overly general.

For overcoming the awkwardness of checking past negative instances in specific-to-general breadth-first search, Tom M. Mitchell[14] proposed a new strategy called "version space". The term "version space" is used to represent all legal hypotheses that are describable within the given generalization language and consistent with the observed training instances. The version space can be represented by two sets of generalizations: the set \( S \) as defined in the specific-to-general breadth-first search strategy, and the dual set \( G \), which is defined as follows:

\[ G = \{g\} g \text{ is consistent with the observed instances, and there is no generalization which is both more general than } g \text{ and consistent with the observed instances}. \]

Together, the sets \( S \) and \( G \) precisely delimit the version space. The advantage of the version space strategy lies in the fact that the set \( G \) summarizes the information implicit in the negative instances that bounds the acceptable level of generality of hypotheses, while the set \( S \) summarizes the information from the positive instances that limits the acceptable level of specialization of hypotheses. Therefore, testing whether a given generalization is consistent with all the observed instances is logically equivalent to testing whether it lies between the sets \( S \) and \( G \). So, it is not necessary to check past training instances.

After comparison among the above three strategies, it is obvious that the strategy of version space is suitable for processing in parallel. This is because in parallel processing, the dependency among different processors should be as little as possible. Owing to the characteristic of not checking past training instances, the version space strategy satisfies the requirement of parallel processing. That is why we choose it as the learning strategy for parallel processing. This point will be illuminated in later discussion.

3. A computational model for parallel learning

In this section, we will propose a model of parallel computation which is suitable for concept learning by the version space strategy. Based on this model, a parallel learning algorithm is presented. Besides, some important steps in parallel learning are also illuminated in detail.

3.1 The parallel model

To simplify our discussion, the computational model proposed is basically a shared memory model[18][21]. This model can be realized in a tree machine as DADO[6][20] does.

In our shared memory model, it is assumed that there are a very large set of processing elements which share a common memory. The shared memory may be either a single large memory structure, or multiple memory modules. We further assume that all the processing elements will execute the same action, of course with different data. Therefore, there must exist a control unit supervising them. Figure 1 presents a simple view of this model.

![Diagram of a shared memory model](image)

For the purpose of learning from examples in parallel, the action performed by the processing elements can be conceptually thought of as being processed bottom-up in a binary tree. This thought is demonstrated in Figure 2.

![Processed bottom-up in a binary tree](image)

Note that in the shared memory model, processing elements are not linked to one another. Therefore, the data can not be communicated directly from one processing element to another. Instead, this communication is done through the shared memory. But logically, we can think the processing elements as being
linked as a binary tree.

In Figure 2, each circle represents a training instance and each rectangle represents a processing element. All processing elements on the same level of the tree can work concurrently and in parallel. The learning process starts from the bottom rectangular level of the tree. In the beginning, each processing element inputs two separate training instances from the shared memory, finds the sets G and S which are defined in the version space strategy, and forms a version space as the output. After doing that, each processing element lying on one level higher than the previous ones inputs two version spaces and processes them in order to form an equivalently single version space as the output. This process is done repeatedly along with the tree bottom-up until a final version space is formed. Now, the question is how two version spaces are processed to form an equivalently single version space? This question will be discussed in next section.

3.2. Merging of two version spaces

The version space strategy allows a convenient, consistent method for merging several sets of generalizations generated from distinct training data sets. The intersection of two version spaces formed from two different sets of training instances yields the version space consistent with the union of the two training sets[14]. Before we show how to do the work of intersection in our parallel model, let's look at an example.

Example 1: LEX[16] is a problem-solving system, which operates in the domain of symbolic integration. It contains a subsystem called Generalizer, which can learn problem-solving heuristics from practice problems. Part of its generalization language is described in Figure 3.

Now, assume the following positive instances are given:

\[ \int 3x \sin(x) \, dx \rightarrow \text{Apply op2} \]
with \( u=3x \) and \( dv=\sin(x) \, dx \).

\[ \int 3x \cos(x) \, dx \rightarrow \text{Apply op2} \]

\[ \int 3x \ln(x) \, dx \rightarrow \text{Apply op2} \]
with \( u=3x \) and \( dv=\ln(x) \, dx \).

\[ \int 3x \exp(x) \, dx \rightarrow \text{Apply op2} \]
with \( u=3x \) and \( dv=\exp(x) \, dx \).

The first instance means that when the integration formula is \( 3x \sin(x) \, dx \), LEX should solve it by executing the second operation, which is "integration by parts". Performed in our parallel model, the learning process is shown in Figure 4.

Figure 4 Parallel processing in Example 1.

It is easily seen from the above example that the process of merging two version spaces into an equivalent version space is the same as the one of learning from two examples, except that the generalization in each set \( S \) lies on a higher level in the generalization language than the training instance does. Note that in the above case, only one generalization exists in each set \( S \). The same relation also exists between the negative training instance and the set \( G \) except in the inverse direction.

Next, we will discuss the case in which the number of the generalizations in the set \( S \) or \( G \) is larger than one. Again, we give an example extracted from [13] to illustrate our thought.

Example 2: Given unordered pairs of simple objects characterized by three attributes. Each object is described by its shape (e.g., circle, triangle), its color (e.g., red, blue), and its size (e.g., large, small). Each instance is described as an unordered pair of attribute vectors, each of which specifies the size, color, and shape of an object. Assume there are two positive training instances as follows:

\text{instance 1: } \{(\text{Large Red Triangle}) \ (\text{Small Blue Circle})\}
\text{instance 2: } \{(\text{Large Blue Circle}) \ (\text{Small Red Triangle})\}

By the version space strategy, the sets \( S \) and \( G \) for the above two training examples are formed as follows:

\[ S: \ [(\text{Red Triangle}) \ (\text{Blue Circle})] \]
\[ G: \ [(\text{Large} \ ?) \ (\text{Small} \ ?)] \]
where each component in the bracket is a
generalization.

There are two generalizations in the set S and one
generalization in the set G. Note that each
generalization in S, but not disjunction of both
generalizations, is consistent with the two positive
training instances. Apparently, if there exists two
version spaces, each of which has two generalizations in
the set S, forming the new set S' from the two
separate sets S needs 2x2 generalization processes. Here
we take the generalization process of two training
examples as a processing unit. Note that in the above
consideration, we do not include the checking time yet.
The checking time is used for the examination of
redundancy, subsumption, and contraction among the
generalizations in sets S and G. This factor will be
discussed in the next section. For the purpose of having
a criteria about processing time, we define an unit
operation as follows:

Definition. unit operation:
An unit operation in learning process is the
formation of set S (G) from two positive
(negative) training instances without considering
the checking time.

The unit operation defined above will be
considered as a basic time unit of processing and
adopted in later sections. In general, assume there exist
two version spaces V1 and V2. S1 contains s1
generalizations and S2 contain s2 generalizations, that is

\[ S1: \text{[generalization}_1, \text{generalization}_2, \ldots, \text{generalization}_{s1}] \]

\[ S2: \text{[generalization}_1, \text{generalization}_2, \ldots, \text{generalization}_{s2}] \]

The process of merging S1 and S2 into an
equivalent S is the Cartesian product of S1 and S2; that
is, each generalization in S1 combines with each
generalization in S2. We denote this operation as S1 X
S2. In the meantime, the newly formed generalizations
must be checked against other generalizations in sets S
and G about the redundancy, subsumption and
contradiction. Therefore, the formation of the new set S
from S1 and S2 spends about \(s1s2\) unit operations (not
including checking time). The merging of G sets is
processed in the same way.

3.3 Checking time analysis

In this section, we will probe the consequence of
checking time on learning process. Checking is mainly for
two purpose:

1. eliminating the redundancy and subsumption in
   separate set S or G,
2. detecting the contradiction between sets S and G.

Assume there exists a set S in some version space,
with s generalizations. The checking time against the
redundancy and subsumption among the s
generalizations needs less than \(s(s-1)/2\) unit operations
because the checking process between two
generalizations in set S is simpler than an unit
operation defined above. It only checks whether one
generalization is more specific or general than the other,
but need not find the generalizations of the two as an
unit operation need. Therefore, we can think \(s(s-1)/2\)
unit operations as the upper bound of time for checking
against the redundancy and subsumption in set S.
Similarly, \(g(g-1)/2\) unit operations can be thought as
the upper bound of time for checking against the
redundancy and subsumption in set G. In addition to
the above two kinds of checking, we also need to check
against the contradiction between sets S and G. That
is, there may exist a generalization in S which is out of the
domain specified by the set G and vice versa. This
action can be performed within \(s+g\) unit operations.

Next, we will consider the situation where two
version spaces are merged into an equivalently single
version space. Again, two version spaces V1 with S1, G1
and V2 with S2, G2 are assumed. We will discuss the
checking time when V1 and V2 are merged. There can be
two alternatives to check against redundancy and
subsumption described below:

Alternative 1. Perform the S1 X S2 and G1 X G2
merging processes described in Section 3.2 first. And
then check the newly formed sets S and G against
redundancy and subsumption.

Alternative 2. As soon as a new generalization is
formed in S1 X S2 (G1 X G2) merging processes,
check the one with the generalizations already checked in the
partially newly formed set S (G) against redundancy
and subsumption.

We will select one from the two alternatives for
checking against the redundancy and subsumption as
our approach. Apparently, the second alternative will
operate faster and need less storage space than the first
one. This is because the former always keeps as few
generalizations as possible in set S (G) along with the
processes of S1 X S2 (G1 X G2), but the latter produces
all generalizations without pruning first. Therefore, the
second alternative will be adopted in our parallel
algorithm for concept learning.

After checking against redundancy and
subsumption, checking for contradiction between sets S
and G is done as described in the beginning of this
section. After doing that, the checking for the merging
process of two version spaces is finished.

3.4 Parallel learning algorithm

From the above discussion, the parallel algorithm
for concept learning can be outlined below. Basically,
this is based upon the divide-and-conquer strategy[9].

Parallel learning algorithm:

INPUT: A set I of n training instances.
OUTPUT: A version space V with sets S and G
consistent with the set I.

STEP 1: Divide I into I1 and I2. The sizes of I1
and I2 are equal.

STEP 2: Recursively and parallelly apply the
algorithm to find the version space V1 for
I1 and V2 for I2.

STEP 3: Merge the two version spaces V1 and V2
into an equivalent version space V.
The final output V is what we want. Here, we can think each positive training instance as a version space with set S containing the instance itself and each negative training instance as a version space with set G excluding the instance itself. Moreover, the merging algorithm is described as follows:

**Version space merging algorithm:**

**INPUT:** Two version spaces V1 with S1, G1 and V2 with S2, G2.

**OUTPUT:** An equivalent version space V with S and G.

**STEP 1:** Initialize both the sets S and G to be ∅.

**STEP 2:** Take a generalization in S1 and a generalization in S2 to perform a generalization process. Set the newly formed generalizations to be S'. Check S' with S against redundancy and subsumption and there may exist three cases:

1. If a generalization s' in S' is more specific than some generalization s in S, discard s and add s' to set S.
2. If a generalization s' in S' is more general than some generalization s in S, discard s' in S' and do nothing.
3. Otherwise, add s' to set S.

**STEP 3:** Repeat **STEP 2** until each generalization in S1 is processed with each generalization in S2.

**STEP 4:** Take a generalization in G1 and a generalization in G2 to perform a generalization process. Set the newly formed generalizations to be G'. Check G' with G against redundancy and subsumption and there may exist three cases:

1. If a generalization g' in G' is more general than some generalization g in G, discard g and add g' to set G.
2. If a generalization g' in G' is more specific than some generalization g in G, discard g' in G' and do nothing.
3. Otherwise, add g' to set G.

**STEP 5:** Repeat **STEP 4** until each generalization in G1 is processed with each generalization in G2.

**STEP 6:** Take a generalization s in S and a generalization g in G. Check s with g against contradiction and there may exist two cases:

1. If g is more specific than s, discard both g in G and s in S.
2. Otherwise, do nothing.

**STEP 7:** Repeat **STEP 6** until each generalization in S is processed with each generalization in G.

After the execution of **STEP 7**, the desired version space is obtained.

4. **Time complexity analysis**

In this section, we will discuss the time complexity of our parallel learning algorithm and compare it with the sequential learning algorithm proposed by Tom M. Mitchell[13][15]. Let TP(n) denotes the time complexity of our algorithm with n training instances. Let M(n) denotes the time complexity of merging two version space V1 and V2, each of which is formed from n/2 training instances, into an equivalent version space. In the parallel learning environment, the following formula holds:

\[ T_P(n) = T_P(n/2) + M(n) \]

Now, we must consider the time complexity of M(n). Let \( s_{\text{max}} \) and \( g_{\text{max}} \) denote respectively the maximum number of generalizations in sets S and G appearing in the learning process for n training instances. Based upon the discussion in previous section, M(n) will satisfy the following inequality:

\[ M(n) \leq O(s_{\text{max}}^2 + g_{\text{max}}^2 + s_{\text{max}} + g_{\text{max}}^3) \]

The first term on the right hand side of the above inequality represents the number of the unit operations in S1 X S2 merging process. The second term represents the checking against redundancy and subsumption in S according to the second alternative in Section 3.3. The third and fourth terms represent the same meanings for processing set G. The fifth term represents the checking against contradiction between sets S and G. For clarity, the above formula can be simplified as follows:

\[ M(n) \leq O(s_{\text{max}}^3 + g_{\text{max}}^3) \]

Substitute M(n) into \( T_P(n) \), we can get the following formula:

\[ T_P(n) = T_P(n/2) + M(n) \leq T_P(n/2) + O(s_{\text{max}}^3 + g_{\text{max}}^3) = O((s_{\text{max}}^3 + g_{\text{max}}^3) \log n) \]

In comparison with the parallel learning algorithm, the time complexity of the sequential algorithm proposed by Tom M. Mitchell[13][15] is given as follows:

\[ T_S(n) = O(s_{\text{max}} \times g_{\text{max}} \times n + s_{\text{max}}^2 \times p + g_{\text{max}}^2 \times p) \]

Here, p represents the number of positive instances and \( \bar{p} \) represents the number of negative instances, \( p + \bar{p} = n \). From the time complexity of parallel learning algorithm and sequential learning algorithm, we can easily make a comparison between them. For the parallel learning algorithm, the time complexity is \( O(s_{\text{max}}^3 + g_{\text{max}}^3 \log n) \) with \( n/2 \) processors. For the sequential learning algorithm, the time complexity is \( O(s_{\text{max}} \times g_{\text{max}} \times n + s_{\text{max}}^2 \times p + g_{\text{max}}^2 \times \bar{p}) \) with uniprocessor. If \( s_{\text{max}} \) and \( g_{\text{max}} \) are constants and small when compared with the number n of the training instances, the time complexity of the parallel learning algorithm will be reduced to \( O(\log n) \) and the time complexity of the sequential algorithm be reduced to \( O(n) \). It is a quite satisfiable result for our parallel learning algorithm to reduce \( O(n) \) with uniprocessor to \( O(\log n) \) with \( n/2 \) processors. Nevertheless, there still exists another problem we must consider. That is the relationship between the sizes of the sets S, G and the number n of the training instances. This relationship is...
discussed in detail in [8] and it is shown that for most learning problems, $s_{max}$ and $g_{max}$ are some constant related to $m^2$, where $c$ is a small constant and $m$ is the number of attributes. Therefore, for most learning problems, we can process them in parallel in $O(\log n)$ with $n/2$ processors.

5. The arrangement of the training instances

It is obvious that on which processor each training instance is located is an important factor influencing the performance of the parallel learning algorithm. If the training instances are arranged so that the positive training instances aggregate on one side and the negative ones on the other side, apparently, it is not necessary for the side of processing the positive training instances to care about the management of set G. The same argument is for the side processing the negative training instances. Besides, the checking for contradiction is not required anymore except in the last stage of the merging of the two sides. Therefore, it can save much processing time. But it has a disadvantage; that is, if noise or inconsistency exists, it can not find the occurrence of the inconsistency until the merging of the two version spaces formed by the positive training instances only and negative ones only is processed. Conversely, if the training instances are arranged in an interleaving way of positive training ones and negative training ones, processing in each level should manage sets $S$ and $G$ respectively, and the checking against contradiction is necessary. But if there is inconsistency found in any stage, the whole learning process stop. Therefore, it can detect noise very effectively. Besides, it may lower the numbers of the generalizations in sets $S$ and $G$ owing to the contradiction checking.

There exists a trade-off between the two data arrangements. How to choose the arrangement of training instances appropriately depends on the application domain and is still a research problem we will probe in the future.

6. Future work

There are still many aspects we will study in the future.

(1). Bounded processors:

In the model we propose, the number of the processors is not limited. As mentioned in the proceeding sections, our parallel learning algorithm needs $N/2$ processors where $N$ is the number of the training instances and may be very large. But in real applications, the number of the available processors is limited within some constant. Therefore, it is necessary for us to develop parallel learning algorithm suitable for the environment of bounded processors.

(2). Noise:

It has been pointed out that data-driven approaches are sensitive to noisy data[9] and there have been some papers proposing the solutions to the learning problems containing noises[7][10][11]. In the real world, noisy environment is very common. From the viewpoint, parallel learning in noisy environment should be studied in order to combine with the real world.

(3). The arrangement of the training instances:

As mentioned in Section 5, the arrangement of the training instances on the processors affects the performance of the parallel algorithm seriously. How to find a appropriate arrangement for a particular domain is still an interesting topic of research.

7. Conclusion

We have combined the parallel processing with concept learning to conquer the problem of slow speed in the uniprocessor. We have proposed in this paper a parallel learning model which is based upon the shared memory model, and a parallel learning algorithm based upon this model by applying the strategy of the version space.

From our discussion, we find that, with $N/2$ processors, the learning process can be accomplished within $O((s_{max}^3 + g_{max}^3) \log N)$, where $n$ is the number of the training instances and $s_{max}$ ($g_{max}$) is the maximum number of generalizations in set $S$ ($G$). Besides, because in most learning problems, $s_{max}$ and $g_{max}$ are related to $m^2$, where $m$ is the number of the attributes and $c$ is a small constant[8], the time complexity can be reduced to $O(\log N)$ and it is a quite satisiable result.

If very unfortunately, $s_{max}$ or $g_{max}$ increases along with the number $N$ of the training instances when $N$ is large, which scarcely happens, $s_{max}$ or $g_{max}$ can be constrained within some constant by the usage of the domain knowledge[14]. Therefore, it is concluded that it is worth learning in parallel when $N$ is large.

REFERENCES


University, Republic of China, 1988.


