File No. UIUCDCS-F-86-966

Parallel AQ and The Connection Machine

by

Gordon Skorstad
Janice C. Skorstad*

AI Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign

ISG Report 86-18

June 1986

* The author is currently in the Qualitative Reasoning Group/Cognition and Language Lab, Department of Computer Science, University of Illinois.

Abstract

We examine the possibility of applying massive parallelism to the powerful machine learning algorithm *AQ* developed at the University of Illinois. We show how one new tool, the Connection Machine, may be used to significantly speed up AQ. We present complexity estimates of parallel source code written in *Connection Machine Lisp*. By exploiting parallelism on the Connection Machine, the complexity of AQ's major routines are reduced from polynomial to logarithmic time.

Acknowledgements

This work benefited from many helpful suggestions provided by R.S. Michalski and I. Mozetic. Thanks also go to I. Mozetic, B. Katz, and C. Uhrik for reviewing an earlier version of this paper.

This research was supported in part by the National Science Foundation under grant DCR 84-06801, Office of Naval Research under grant N00014-82-K-0186 and Defense Advanced Research Project Agency under grant N00014-K-85-0878.

1. Introduction

A number of researchers are currently investigating the use of parallelism in AI. Some of the driving forces behind this research are:

- the emergence of low cost VLSI technology,
- the realization that we are approaching the limit of single processor speed,
- the need to perform costly AI computations in real time (eg., in battlefield management).

Several researchers have claimed dramatic performance increases through the use of parallelism. Shaw [6] for example, claims two orders of magnitude speedup in the execution of production systems on the Non-Von parallel computer using the OPS5 language. Flynn and Harris [2] claim a three to four order magnitude speedup for an object recognition algorithm on the Connection Machine. These speedups are relative to sequential versions of the same algorithm.

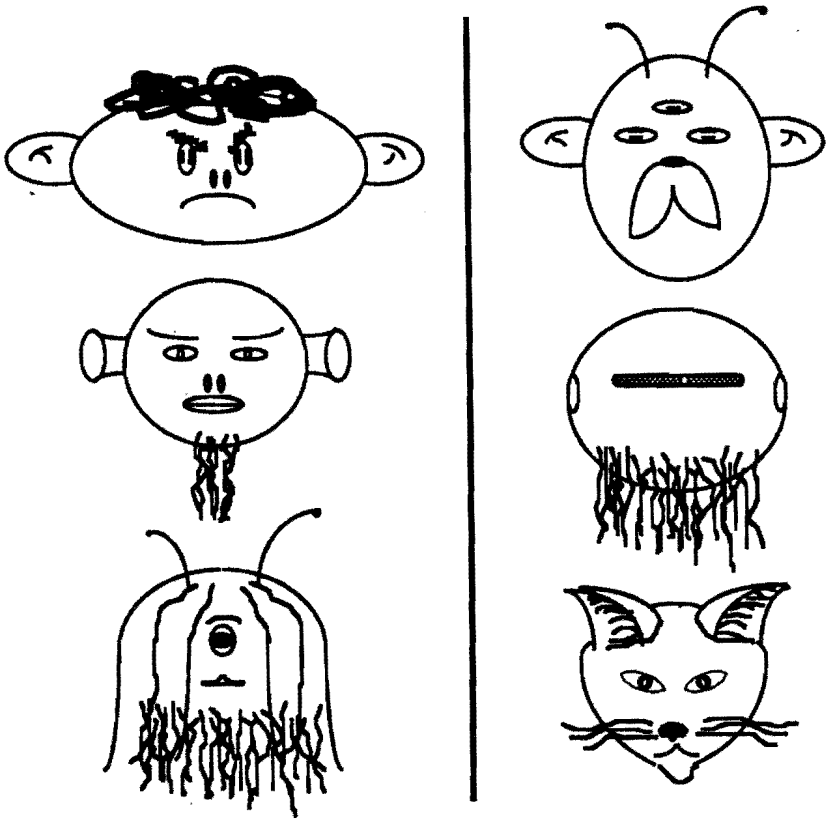
In this paper we show how parallelism on the Connection Machine (Hillis [3]) can be used to significantly speed up Michalski's AQ algorithm [4]. AQ is a *quasi-optimal* algorithm for solving the *General Covering problem*. The General Covering problem is encountered often in machine learning, pattern recognition, switching theory and other fields. AQ is quasi-optimal because in those cases where its result is not optimal, it provides an estimate of how close it is to the optimum. AQ's optimality criteria are adjustable by the user and are typically based on syntactic qualities such as brevity and completeness of the result.

2. Sequential AQ

Briefly, the General Covering problem can be defined as the problem of determining the optimal decision rules for classifying groups of objects or events. An example of the General Covering problem in the field of machine learning is *learning from examples*. Learning from examples has been a subject of intensive research over the last decade. In learning from examples, the goal is to induce general descriptions of concepts from preclassified instances of these concepts.

Figure 1 shows a simple learning from examples problem. A series of aliens have been classified into friendly and deadly groups. The goal is to learn the common rule that distinguishes the friendly aliens from the deadly ones. The internal form of AQ's concept recognition rule for the *friendly* aliens and its English translation are as follows:

Internal rule:	[eye-brow = yes]
English paraphrase:	if the alien has an eye-brow, then it's friendly.



friendly aliens deadly aliens

Figure 1. Learning Concepts From Examples

AQ generates a minimal or quasi-minimal description of classes of events called a *cover*. Events and class descriptions (covers) are represented in the enhanced propositional calculus VL_1. For a detailed description of VL_1 and the AQ algorithm, see Michalski [4].

VL_1 expresses multi-valued formulas with multi-valued variables by using *selectors* which are two-valued functions. An example of a selector is $[x_3 = 2,3,5]$. This is interpreted as meaning "x_3 has the value 2, 3, or 5". Conjunctions of selectors are called *complexes*. An example of a complex is $[x_3 = 2,3,5][x_4 = 2,4]$, which is the conjunction of two selectors. Finally, disjunctions of complexes are called *covers*. An example of a cover is $\{[x_2 = 0,2] \vee [x_3 = 2,3,5][x_4 = 2,4]\}$.

The sequential AQ algorithm consists of two main functions, *AQ* and *Star*, listed in figures 2 and 3 (Falkenhainer [1]). Function AQ produces the maximally general disjunction of complexes that includes (or *covers*) all of the positive events and none of the negative events. *AQ's* most computationally expensive step is the *Star* generation step. The concept of a *star* is central to much of the machine learning research done in the AI Lab. In VL_1, the star of an event *e* against a set of negative events *F* is defined as the set of all maximally general complexes that cover event *e* and that do not cover any negative events in *F*. It can be shown (Skorstad [7]), that generating a star in the sequential AQ algorithm requires at most $O[\#negs \cdot \#vars^2 \cdot \max(domsize \cdot \#vars, \#pos)]$ steps, where:

domsize = the maximum number of values any variable can assume,
#negs = the number of events outside the positive event class,
#pos = the number of positive events (events being covered),
#vars = the number of variables or attributes in the problem space.

```
Function AQ (pos_events, neg_events : events) : cover;
    Var
        seed    : event;                    {Positive seed event}
        star    : complex_list;
        best    : complex;
    Begin
        While (pos_events <> nil) do begin
            seed    := head(pos_events);        {Randomly choose the seed event}
            star    := Star(seed, neg_events);  { Most expensive step }
            best    := Bestcomplex(star);       {Select best complex in "star"}
            cover   := append(best, cover);     {Add best complex to "cover"}
            pos_events := Knockout(pos_events, best)
        end
    end;
```

Figure 2. Sequential **AQ** algorithm

```
Function Star (seed, neg_events : events) : complex_list;
    Var
        ElemStar : complex_list;  {Elementary Star}
    Begin
        star := universe;                                  {Initially, Partial star = domain space}
        For neg in neg_events do
            ElemStar := ExtendAgainst(seed, neg);   {Generate Elementary Star}
            star     := Multiply(star, ElemStar, neg); {Intersect Elementary & Partial stars}
            If length(star) > maxstar
                then begin
                    star := Absorb(star);              {Remove redundant complexes}
                    star := SelectBest(star, maxstar, LEF);  { Keep "best" complexes}
                    end
            end
    end;
```

Figure 3. Sequential **Star** Algorithm

Before describing our parallel AQ code, we describe the major Lisp data structures and operators used to control parallelism on the Connection Machine.

3. Connection Machine Lisp

Connection Machine Lisp, or CmLisp, is an extension of Common Lisp designed to support the parallelism of the Connection Machine. For a description of Common Lisp, see Steele [8]. For a more detailed description of CmLisp than given here, see Hillis [3]. CmLisp achieves its parallelism by executing commands on active data structures called *xectors.* A xector is roughly a set of processors, each of which contains a value. It is similar to a vector stored across many processors. Unlike a vector, xector elements have three parts: a *domain,* a *range,* and a *mapping* between them. Each object in the domain is called an *index* of the xector. Each object in the range is called a *value.* An index/value pair is called an *element.* Each xector is a set of elements with unique indices. An example of a xector is:

{STUDENT→HUNGRY PROF→MAD MD→GREEDY}.

This xector has three elements. It maps the three indices STUDENT, PROF, and MD to the three values HUNGRY, MAD, and GREEDY, respectively. In the Connection Machine, each index corresponds to a processor/memory cell.

A special type of xector where each index maps onto itself, is used to represent a *set.* Here the index and the value are the same, and we omit the arrow. For example,

{PROF→PROF 2→2 A→A} ≡ {PROF 2 A}.

Another special case occurs when the indices are a sequence of integers starting from zero. Square brackets are used to represent these xectors. These xectors resemble vectors:

{0→STUDENT 1→PROF 2→C} ≡ [STUDENT PROF C].

Xectors are treated like any other normal Lisp object. They can be stored in arrays, passed as parameters, bound to variables, etc. Xectors are analogous to *sequences* in Common Lisp, and we have many generic sequence functions at our disposal. These functions work on xectors, using the canonical order of the indices as the order of the elements.

There are two operators in CmLisp which allow it to exploit parallelism. These are the α (alpha) and β (beta) operators. The α operator can be used to convert a value into a constant xector. (This is essentially loading a value into every processor.) When α precedes an expression, the expression is interpreted as a xector with the constant value of the expression. For example, the following expressions load a "1" and a "6", respectively, into every processor in the machine:

α 1 ⇒ {→ 1}.
α (- 8 2) ⇒ {→ 6}

α can also be used to generate a xector of functions. In the following example, the α operator generates a xector of PLUS functions which is applied to each element in two xectors. Note that the addition is applied to the values of the elements with corresponding indices. The result of this operation is a xector.

(α+ '{A→1 B→2} '{A→10 B→20 C→30}) ⇒ {A→11 B→22}.

(α*f* $xector_j$ $xector_k$) will perform function *f* as many times as there are elements in the smaller xector. In general, α takes a single value or function and makes many copies of it.

In a sense, β is the reverse of α. Beta combines a xector into a single value. Beta applies a two-argument function to a xector's values in parallel. The reduction is done in logarithmic time. For example,

(β+ '{A→1 PROF→5 C→2 }) ⇒ 8.

Alpha and Beta can be combined to form useful functions. For example, NORM calculates the euclidean norm of a vector:

(Defun Norm (x) (Sqrt (β+ (α* x x)))).

Beta can also be used to combine two xectors. The *indices* of the new xector are taken from the *values* of the second xector. The *values* of the new xector are taken from the values of the *first* xector. For example,

(β '{A→1 B→2} '{A→X B→Y}) ⇒ {X→1 Y→2}

These two operators, α and β, shield the programmer from the low level details of the Connection Machine while giving him control over its parallelism.

4. Parallel AQ

One of the first and most important decisions in parallelizing AQ is how to represent events, selectors and complexes. Our goal was to spread the structures over as many processors as possible, thus maximizing potential parallelism.

For events, the simplest possible parallel representation was chosen. Events are represented as simple xectors of values. The xector indices 0, 1, ... etc., correspond to the independent variables x_0, x_1, ... x_n. For example, the positive event $(x_0x_1x_2x_3)$ = (0031) is represented by the xector:

{0→0 1→0 2→3 3→1} = [0 0 3 1].

Selectors and complexes are represented as xectors of xectors. For example, the selector $[x_2=0,1,2]$ is represented as **{2→[0 1 2]}**. The complex $[x_2=0,1,2][x_3=1]$, which is a conjunction of two selectors, is represented as: **{2→[0 1 2] 3→[1]}.**

Finally, a star, which is a disjunction of complexes, is represented by three levels of xectors. Each element of the outermost xector is a conjunction of selectors. The innermost xectors contain the selector references. For example, the star $([x_0=1,2]\vee[x_2=0,1,2][x_3=1])$, which contains two complexes, is represented as:

[{0→[1 2]} {2→[0 1 2] 3→[1]}].

Our parallel star generation algorithm is diagrammed in figure 4. The thread of control flows from top to bottom. The input to the algorithm is one positive event, called the *seed* event, and *#negs* negative events. The branching into *#negs* paths at the top of the figure represents the spawning of *#negs* parallel *One-Estar* processes. This spawning is accomplished using the α operator of CMLisp. Each of the *One-Estar* processes generates an *elementary star*, which is the maximally general cover of a single positive event relative to a single negative event. Thus, for a single positive event, we generate *#negs* elementary stars, one for each negative event.

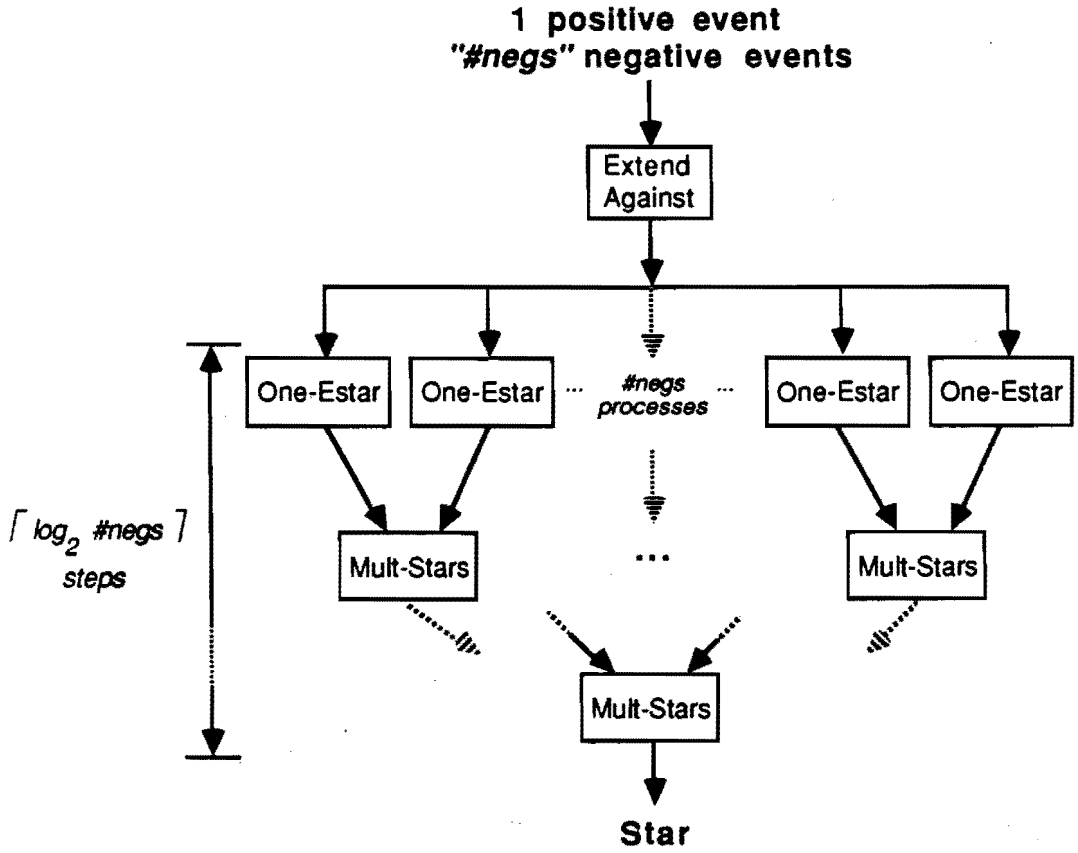


Figure 4. Parallel Star Flow Diagram

In order to form the star of the seed event, all of its elementary stars must be intersected. The logical intersections yield the star of the seed. This star covers the seed but none of the negative events. In figure 4, the logical intersection of two stars is performed by *Mult-Stars*. The tree-like reduction of all the stars is accomplished using the β operator of CMLisp.

The upper level parallel code corresponding to the flow diagram is shown in figure 5. The function *Mult-Stars* is executed in parallel on all pairs of elementary stars produced by *ExtendAgainst*. The returned values from each *Mult-Stars* function is then passed up to the next level in the β tree. The total number of levels in the tree is $\lceil \log_2(\#negs) \rceil$, where *#negs* is the number of negative events.

Note: the Lisp program segments we list in this paper have not yet been run on the Connection Machine. Therefore, they may need minor refinements when implemented.

```
;;;
;;; Generate the Star for positive event "a".
;;;
(defun MakeStar (a B)
    (βMult-Stars (ExtendAgainst a B)))
```

Figure 5. Parallel Star Code

4.1 Elementary Star Generation

The parallel code for generating elementary stars is shown in figure 6. As illustrated in figure 3, this function is usually performed sequentially, inside a loop. The top level function, *ExtendAgainst*, concurrently calls *One-Estar* for each negative event. The code *αOne-Estar* in *ExtendAgainst* spawns the parallel paths shown in figure 4. The code *αa* sends a copy of the positive event xector *a* for each call to One-Estar. Thus the parameters to One-Estar are always positive event *a* and one negative event b_i. Both parameters are xectors.

The function One-Estar returns a xector representing the elementary star covering *a* but not b_i. This function uses α to simultaneously process each pair of attributes in its two parameters. For example, a function call *(One-Estar '[1 0 0 1] '[0 0 3 1])* generates four calls to *Negate*, one call for each attribute (or variable) in [1 0 0 1] and [0 0 3 1]. The four calls to *Negate* return the xector: **{0→[1 2] 1→* 2→[0 1 2] 3→*}**.

The *delete* function in One-Estar then changes this to **{0→[1 2] 2→[0 1 2]}**. After calling *Nest*, the final result of One-Estar is an elementary star, a disjunction of single-selector complexes:

[{0→[1 2]} {2→[0 1 2]}].

It can be shown that on a hypothetical, *perfect* Connection Machine, where α parallelization is achieved in constant time and no message contention occurs, the generation of all elementary stars is performed in $O[\log(\#pos) \cdot \log^2(\#vars)]$ steps. This can be shown by tracing through the code, starting with the lowest level routines.

```
;;;
;;; Nest a xector element to a deeper level.
;;;
(defun Nest (var value)
    '( ,var -> ,value ) )
;;;
;;; Negate one attribute in event "b".
;;;
(Defun Negate (ai bi DOMi)
    (cond
        ((eq bi ai) '*)
        (T (remove bi DOMi)) ) )
;;;
;;; Generate one elementary star for "a" against "b". This is
;;; done simultaneously for all attributes of "b" and "a".
;;;
(Defun One-Estar (a b)
    (let ((Estar (delete '* (αNegate b a *DOM*))))
        (setf Estar (αNest Estar (Domain Estar)))
        (setf Estar (sort Estar #'LEF))
        (delete-if 'T Estar :start (+ 1 *maxstar))))
;;;
;;; Generate all elementary stars for positive
;;; event "a" against all negative events bi in B.
;;;
(Defun ExtendAgainst (a B)
    (αOne-Estar αa B) )
```

Figure 6. Parallel Generation of Elementary Stars

The routine *Nest* executes in constant time. This is assuming that a xector can be constructed from an index/value pair in constant time.

The routine *Negate* has at most two steps:

1) (eq ax bx) '*)
2) (T (remove bx Domx)).

ax and *bx* are single integers, *Domx* is a xector. Step 1 is performed in constant time. Step 2 is also performed in constant time since all elements in *Domx* can be compared to *bx* in parallel. Thus, *Nest* executes in constant time.

The next higher level routine is *One-Estar*. This has five main steps:

1) (αNegate b a *Dom*)
2) (delete '* (result of 1))
3) (αNest Estar (Domain Estar))
4) (sort (result of 3) #'LEF)
5) (delete-if 'T Estar :start (+ 1 *maxstar*))))

Since *Negate* and *Nest* execute in constant time, steps 1 and 3 do also. Notice that because of the α operator, execution time is independent of the number of attributes in events *a* and *b*. Step 2 is also performed in constant time since deletion of a single value from a xector can be done by examining each element in parallel. Steps 4 and 5 trim the size of the elementary star down to *maxstar* complexes. Variable *maxstar* is a user adjustable quantity which defines how many complexes are kept during star generation. In the worst case, *#vars* complexes will be generated in an elementary star. According to Hillis [3], sorting this number of objects would require $O[\log^2(\#vars)]$ steps on the Connection Machine if the LEF function executed in constant time. The LEF function is actually $O[\log(\#pos)]$, so sorting is $O[\log(\#pos)\cdot\log^2(\#vars)]$. Step 4 is thus the dominating step in the *One-Estar* routine. If the sort procedure assigns ascending integers to the xector elements, step 5 can be performed in constant time as follows:

(i) The value (*maxstar* + 1) is broadcast to all elements.
(ii) Each element whose value is ≥ (*maxstar* + 1) removes itself from the xector.

Finally, since *ExtendAgainst* is just a set of parallel calls to the function *One-Estar*, it executes in the same time as *One-Estar*, that is, $O[\log(\#pos) \cdot \log^2(\#vars)]$.

4.2 Star Multiplication

The parallel *Mult-Stars* code is shown in figure 7. Its purpose is to logically intersect, or multiply, two stars. As illustrated in figure 3, multiplication is ordinarily performed once for each negative event. Our code also performs *absorption* to remove all redundant complexes from the final star, and *trimming*, which trims the least desirable complexes during star generation.

Mult-Stars requires ten steps. Lower level functions such as *Absorb-Star* are listed in the appendix along with their complexity estimates. Steps *(1)* through *(6)* collect a priori product complexes from the two stars, *star1* and *star2*, being multiplied. A priori complexes are complexes which would remain unchanged after multiplication. It is more efficient to identify and remove these product complexes as soon as possible. A priori complexes are identifiable as complexes in star1 which subsume complexes in star2 and vice versa.

```
;;;
;;; Multiply two stars.
;;;
(defun Mult-Stars (star1 star2)
   ;; remove a priori product complexes from star1 and star2.
   (let ((product nil) (prod (Absorb-Star star1 star2)))          ;(1)
      (nset-difference star2 prod)                                ;(2)
      (nconc product prod)                                        ;(3)
      (setf prod (Absorb-Star star2 star1))                       ;(4)
      (nset-difference star1 prod)                                ;(5)
      (nconc product prod)                                        ;(6)
      ;; multiply remaining complexes in star1 and star2.
      (nconc product (αMult-Comp-Star star2 astar1))              ;(7)
      ;; Absorb all product complexes that subsume other complexes.
      (nset-difference product                                    ;(8)
         (bnconc (αAbsorb-Complex
                  product aproduct (domain product))))
      ;; Trim product, leaving only *maxstar* best complexes.
      (sort product #'LEF)                                        ;(9)
      (delete-if 'T product :start (1+ *maxstar*)))               ;(10)
```

Figure 7. Parallel Star Multiplication

Steps *(1)* through *(3)* remove a priori product complexes from *star2* and collect them in the variable *product.* Step *(1)* is $O[\log(\text{maxstar}) \cdot \log(\#vars)]$, which can be shown by tracing the lower level routine *Absorb-Star.* The logarithm function follows from the β reduction performed in the lower level routines *Absorb-Star* and *Mult-Comps* (see appendix). Step *(2)*, which uses the destructive *nset-difference* set operation, is $O(\log \text{maxstar})$. This follows from the fact that set difference can be performed in parallel as (βintersect (αdelete prod αstar2)). Step *(3)* executes in constant time.

Steps *(4)* through *(6)* remove a prior product complexes from *star1* and add them to the variable *product.* These steps have the same complexity as the corresponding steps *(1)* through *(3).*

Step *(7)* is where the actual multiplication of star1 and star2 occurs. Like steps *(1)* and *(4)*, step *(7)* is $O[\log(maxstar) \cdot \log(\#vars)]$. In step *(8)*, we remove all redundant complexes from the product. That is, we remove all product complexes which subsume other complexes. Once again, complexity is $O[\log(maxstar) \cdot \log(\#vars)]$.

Finally, to control the growth of complexes, the least desirable complexes are removed in steps *(9)* and *(10).* This is accomplished by first sorting (step *(9)*) the product complexes, and then discarding (step *(10)*) all but the best *maxstar* ones. In the worst case, $maxstar^2$ complexes will be formed by the multiplication step *(7).* Sorting these complexes on the Connection Machine requires $O[\log^2(maxstar) \cdot \log(\#pos)]$ operations. Assuming that sorting assigns a unique integer to each xector element, step *(10)* is performed in constant time.

Since *Mult-Stars* is executed $\log_2(\#negs)$ times (see figure 4), the worst case number of steps required for the entire star generation process is the maximum of:

(i)	$O[\log(\#pos) \cdot \log^2(\#vars)]$	*ExtendAgainst complexity*
(ii)	$O[\log(maxstar) \cdot \log(\#negs) \cdot \log(\#vars)]$	*Mult-Stars multiplication*
(iii)	$O[\log^2(maxstar) \cdot \log(\#negs) \cdot \log(\#pos)]$	*Mult-Stars sorting complexity*

In practice, *maxstar* is typically held constant. If we treat *maxstar* as a constant, the following upper bound is derived for parallel star complexity:

$$\text{Max}\left\{\begin{array}{l} O\,[\log(\#pos)\cdot\log^2(\#vars)], \\ O\,[\log(\#negs)\cdot\log(\#vars)], \\ O\,[\log(\#negs)\cdot\log(\#pos)] \end{array}\right\}$$

Parallel Star Complexity

This is dramatically less than the complexity of the sequential Star algorithm, which can be shown (Skorstad [7]) to be:

$$\text{Max}\left\{\begin{array}{l} O\,[\#negs\cdot\#pos\cdot\#vars^{\,2}], \\ O\,[domsize\cdot\#negs\cdot\#vars^{\,3}] \end{array}\right\}$$

Sequential Star Complexity

5. Summary and Further Research

We have shown that the Connection Machine can, in theory, be used to reduce the complexity of major portions of the AQ algorithm from polynomial to logarithmic. Examples of how much slower the parallel Star complexity function grows can be seen in figure 8. For each row in the table, the sizes of the variables, *domsize*, *#negs*, *#pos*, and *#vars* are equal. Base 2 logarithms are used.

Independent Variable Value *(domsize=#negs=#pos=#vars)*	**Complexity Function**	
	Sequential	Parallel
4	1024	8
16	1,048,576	64
64	$1.07 * 10^{9}$	216
1024	$1.13 * 10^{15}$	1000

Figure 8. Example Complexity Function Values.

There are several avenues of research which could be explored further. An obvious goal would be to implement and test our code on a Connection Machine. How closely a real machine can approach theoretical performance is an interesting question.

In our work, we have reduced time complexity from polynomial to logarithmic. We achieved this by trading space for time. How *much* space we consumed in the process remains an important question that relates to the practicality of our code.

There are several machine learning algorithms which may be suitable for parallelization. For example, the INDUCE structural learning program [5], developed at the University of Illinois at Urbana, consumes most of its time performing graph matching on its internal representation of events. Such graph operations are ideally suited to the Connection Machine architecture. Achieving speedups in computationally expensive AI operations such as this will extend the range of problems our programs can successfully attack.

References

[1] Falkenhainer, B.C., Quantitative Empirical Learning: An Analysis and Methodology, MS Thesis, University of Illinois at Urbana-Champaign, (1985) 44-47.

[2] Flynn, A.M. and Harris, J.G., Recognition Algorithms for the Connection Machine, *IJCAI,* Los Angeles, CA (1985) 57-60.

[3] Hillis, W. D., The Connection Machine, PhD Thesis, MIT (1985).

[4] Michalski, R.S., Synthesis of Optimal and Quasi-Optimal Variable-Valued Logic Formulas, *Proceedings of the 1975 International Symposium on Multiple-Valued Logic,* Indiana University, Bloomington, IN (1975) 76-87.

[5] Michalski, R.S. and Stepp, R., "INDUCE 2: A Program for Learning Structural Descriptions from Examples", ISG 83-4, UIUCDCS-F-83-904, Department of Computer Science, University of Illinois, Urbana, IL, 1983.

[6] Shaw, D.E., NON-VON's Applicability to Three AI Task Areas, *IJCAI,* Los Angeles, CA (1985) 61-72.

[7] Skorstad, G., "AQ Complexity", to appear in *Reports of the Intelligent Systems Group,* Dept. of Computer Science, University of Illinois at Urbana, 1986.

[8] Steele Jr., G.L., *COMMON LISP: The Language,* (Digital Press, 1984).

APPENDIX

Lisp Star Generation Code
Including Time Complexities

The Lisp program segments in this paper have not yet been run on the Connection Machine. Therefore, they may require minor changes when implemented.

```
;;;;
;;;; EXTEND-AGAINST FUNCTIONS
;;;;
;;; Nest a xector element to a deeper level.
;;;
(defun Nest (var value)                                  ; O(1) Complexity
    '{ ,var -> ,value } )
;;;
;;; Negate one attribute in event "b".
;;;
(Defun Negate (ai bi DOMi)                               ; O(1) Complexity
    (cond
        ((eq bi ai)  '*)
        (T (remove bi DOMi)) ) )
;;;
;;; Generate one elementary star: a -/ b.
;;;
(Defun One-Estar (a b)                                   ; O[log #pos)*(log2#vars)] Complexity
    (let ((Estar (delete '*  ( αNegate b a  *DOM*))))
        (setf Estar (αNest Estar (Domain Estar)))
        (setf Estar (sort Estar #'LEF))
        (delete-if 'T Estar :start  (+ 1 *maxstar))))
;;;
;;; Generate elementary stars for positive
;;; event "a" against all negative events b_i in B.
;;;
(Defun ExtendAgainst (a B)                               ; O[log #pos)*(log2#vars)] Complexity
    (αOne-Estar αa B) )
```

```
;;;;
;;;; MULTIPLICATION FUNCTIONS
;;;;
;;; Multiply two complexes.
;;;     O(log #vars)
;;;
(defun Mult-Comps (comp1 comp2)
    (βintersect (nconc comp1 comp2)
        (nconc (domain comp1) (domain comp2))))
;;;
;;; Multiply a complex by a star.
;;;     O(log #vars)
;;;
(defun Mult-Comp-Star (comp star)
    (αMult-Comps αcomp star))
;;;;
;;;; ABSORPTION FUNCTIONS
;;;;
;;; Check if complex2 subsumes complex1.
;;; If so, then return complex2.
;;;     O(log #vars)
;;;
(defun Subsumes (complex1 complex2)
    (let ((product (Mult-Comps complex1 complex2)))
        (if (equal product complex2)
            complex2 nil)))
;;;
;;; Return all complexes in "star" which subsume "complex".
;;; CompIndex", when non-nil, is the xector index of "complex".
;;;     O(log #vars)
;;;
(defun Absorb-Complex (complex star CompIndex)
    (αSubsumes αcomplex
        (if CompIndex
            (setf (aref star CompIndex) nil)
            star)))
```

```
;;;
;;; Return all complexes in "star2" which
;;; subsume complexes in "star1".
;;;
(defun Absorb-Star (star1 star2)                    ; O(log maxstar)(log #vars)
    (βnconc (αAbsorb-Complex star1 αstar2 αnil)))


;;;;
;;;; LEF FUNCTIONS
;;;;
;;; Is "value" within the "reference" range?
;;; If so, return T, otherwise nil.
;;;
(defun Value-Covered (value reference)              ; O(1) Complexity
    (if (find value reference)
        T nil))
;;;
;;; Is event "a" covered by "complex"?
;;; If so, return 1, otherwise 0.
;;;
(defun Event-Covered (a complex)                    ; O(1)
    (if (find nil (αValue-Covered a complex))
        0 1))
;;;
;;; Count the number of positive events covered
;;; by "complex". "A" is the xector of all positive events.
;;;
(defun Pos#Covered (complex)                        ; O(log #vars)
    (β+ (αEvent-Covered *A* αcomplex)))
```

```
;;; Lexico Graphical Function (LEF).
;;; Returns T if complex1 is less desirable than complex2.
;;;       O(log #pos)
;;;
(defun LEF (complex1  complex2)
    (let ((len1 (length complex1)) (len2 (length complex2)))
        (cond                                          ; test length of complexes
            ((< len1  len2) nil)
            ((> len1  len2)  T)
            (T (cond                                   ; test coverage of pos events
                ((< (Pos#Covered complex1)
                    (Pos#Covered complex2)  T)
                (T nil) ))))))
;;;;
;;;; MAIN FUNCTIONS
;;;;
;;; Multiply two stars.  Resulting product has no redundant complexes.
;;;       O[ (log^2 maxstar)*(log #pos)]   -or-   O[(log maxstar)*(log #vars)]
;;;
(defun Mult-Stars (star1 star2)
    ;; collect a priori product complexes from star1 and star2.
    (let ((product nil) (prod (Absorb-Star star1 star2)))
        (nset-difference star2 prod)                   ; remove star2 comps
        (nconc product prod)
        (setf prod (Absorb-Star star2 star1))
        (nset-difference star1 prod)                   ; remove star1 comps
        (nconc product prod)
        ;; multiply remaining complexes in star1 and star2.
        (nconc product (αMult-Comp-Star star2 αstar1))
        ;; Absorb all product complexes that subsume other complexes.
        (nset-difference product
            (βnconc (αAbsorb-Complex
                        product αproduct (domain product))))
        ;; Trim product, leaving only *maxstar* best complexes.
        (sort product #'LEF)
        (delete-if 'T product  :start (1+ *maxstar*)))
```

```
;;;
;;; Generate the Star of positive event "a" against all negative events "B".
;;; Complexity is the maximum of:
;;;
;;;      O[(log #pos) * (log^2 #vars)]
;;;      O[ (log^2 maxstar) * (log #negs) * (log#pos)]
;;;      O[(log maxstar) * (log #negs) * (log #vars)]
;;;
(defun MakeStar (a B)
    (βMult-Stars (ExtendAgainst a B)))
```

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-F-86-966	2.	3. Recipient's Accession No.

4. Title and Subtitle

Parallel Concept Learning on the Connection Machine

5. Report Date: June 1986

6.

7. Author(s)

Gordon Skorstad and Janice C. Skorstad

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Artificial Intelligence Laboratory
Department of Computer Science
University of Illinois
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.
NSF DCR 84-06801
N00014-82-K-0186
N00014-85-K-0878

12. Sponsoring Organization Name and Address

National Science Foundation, Washington, D.C.
Office of Naval Research, Arlington, VA

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

We examine the possibility of applying massive parallelism to the powerful machine learning algorithm AQ developed at the University of Illinois. We show how one new tool, the Connection Machine, may be used to significantly speed up AQ. We present complexity estimates and parallelized source code written in Connection Machine Lisp. By exploiting parallelism on the Connection Machine, the complexity of AQ's major routines are reduced from polynomial to logarithmic time.

17. Key Words and Document Analysis. 17a. Descriptors

Knowledge Acquisition and Learning
Parallel Processing

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 23
	20. Security Class (This Page) UNCLASSIFIED	22. Price