J. Expt. Theor. Artif. Intell. 3(1991) 109–127

Parallel implementation of a class reasoner

JAMES GELLER and CHARLES (YAOGUI) DU

Computer and Information Sciences Department, New Jersey Institute of Technology, Newark, NJ 07102, USA.

Abstract. One method to overcome the notorious efficiency problems of logical reasoning algorithms in AI has been to combine a general-purpose reasoner with several special-purpose reasoners for commonly used subtasks. In this paper we are using Schubert's (Schubert *et al.* 1983, 1987) method of implementing a special-purpose class reasoner. We show that it is possible to replace Schubert's preorder number class tree by a preorder number list without loss of functionality. This form of the algorithm lends itself perfectly towards a parallel implementation,[1] and we describe design, coding and testing of such an implementation. Our algorithm is practically independent of the size of the class list, and even with several thousand nodes learning times are under a second and retrieval times are under 500 ms.

Keywords: Massively Parallel Knowledge Representation (M&PKR), semantic networks, special purpose reasoner, class hierarchies.

Received 30 May 1990; revision received 11 October 1990

1. Introduction

In this paper we share the beliefs expressed by Shastri (1988, 1989) that 'To be deemed intelligent, a system must be capable of action within a specified time frame', (1988, p. 3) because 'in spite of operating with a large knowledge base, human agents take but a few hundred milliseconds to perform a broad range of cognitive tasks' (1988, p. 2). As Shastri we conclude that 'A possible solution of the computational effectiveness problem lies in a synthesis of the *limited inference* approach and *massive parallelism*' (1988, p. 4; author's italics).

One of the most investigated constructs of artificial intelligence and knowledge representation is the class hierarchy. It has been treated, with slightly different orientations, by many researchers. A class hierarchy is the central element of KL-ONE (Brachman 1979, Brachman & Schmolze 1985, Woods 1975) and of other members of the KL-ONE family, e.g. KRYPTON (Brachman *et al.* 1983), LOOM (MacGregor 1988), KL-TWO (Vilain 1985), and BACK (Nebel & von Luck 1987).

Class hierarchies also form the basis of work in the 'inheritance community'. Based on Fahlman's (1979) landmark dissertation, Touretzky (1986) has described the mathematics of inheritance systems. Different approaches to inheritance following this paradigm have been reported recently, e.g. by Horty & Thomason (1988), Padgham (1988), and Stein (1989).

Ideas about class hierarchies go back to Quillian's (1968) original work on semantic networks which already contained a class subclass link. His memory

model was the basis for subsequent spreading activation theories (Collins & Loftus 1975, Anderson 1983) and as such became a major source for the development of cognitive science as a field. For more details on the development of class hierarchies the reader is referred to Brachman & Levesque (1985) and Brachman *et al.* (1989), which cover the knowledge representation view; Collins & Smith (1988) which covers the cognitive science aspects, and finally Shastri's (1989) review.

In his papers Shastri describes a fast method to deal with attribute inheritance *and recognition of instances with given attributes* in an integrated framework that lends itself to a connectionist implementation. The task that we are attacking in this paper is more limited than Shastri's. We are interested only in maintaining a class hierarchy for the purpose of verifying whether two given classes stand in a subclass–superclass relation or not. However, we show the complete implementation of our ideas on an existing parallel hardware facility. We agree with Wilson (1989) that the SIMD architecture (= single instruction multiple data) is a good candidate for building parallel intelligent systems, and with Evett *et al.* (1990) that the Connection Machine is a good candidate for building parallel knowledge representation systems.

Among the large number of papers that deal with class hierarchies we have found Schubert *et al.*'s work (1983, 1987) especially valuable, because it shares our interests in limited inference class reasoners. Schubert's papers describe a system that combines a general-purpose logic based reasoner with four special-purpose reasoners for commonly performed mental operations. These special-purpose reasoners deal with classes, parts, time, and colors.

The basic idea of Schubert's class reasoner is to define one (or a small number of) class tree(s), i.e. strict hierarchies. In one such tree, a node represents a subclass of the class represented by its parent node. Every node has a pair of numbers associated with it. Figure 1 shows an example of such a tree. The first number of every pair is the right to left preorder traversal number of this node in the tree. That means that we perform a standard preorder traversal, but we always start with the rightmost child instead of the leftmost child. On the way we count the nodes that we are traversing and store with each node its own count. This is the preorder traversal number.

The second number of every pair is the highest preorder traversal number among all the descendents of this node. If a node is a leaf node, then the second number is by definition identical to the first number. (One can replace this definition by saying that a node is a descendent of itself.) We will refer to the first number of every number pair as the PRENUM and to the second number of every number pair as the MAXNUM of the given node.

These number pairs make it possible to decide whether a class B is a subclass of a class A by comparing the number pairs of A and B. If the interval defined by the number pair of B is a subinterval of the interval defined by the number pair of A, then B is a subclass of A. If the two intervals defined by the two number pairs are disjoint, then no such subclass relation exists. Note that Schubert's definition makes cases of partial overlap of two intervals impossible.

Compare in Figure 1 the node Bird with reference to the node Animal; (5, 7) is a subinterval of (3, 7), and therefore a Bird is an Animal. On the other hand, (5, 7) is not a subinterval of (8, 8), therefore a Bird is not a Mineral. If number pairs are associated with node symbols through a hashing mechanism very efficient

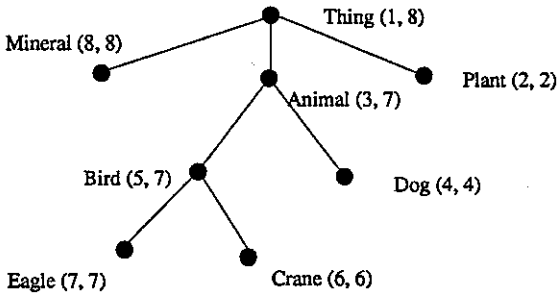


Figure 1. A tree with preorder numbering.

verification of subclass relations becomes possible, without requiring an actual traversal of the tree.

Unfortunately, Schubert does not make any mention of an update operation in the above papers. The extreme retrieval efficiency of his algorithm stands in stark contrast to naive update algorithms which require the complete recomputation of all number pairs. In previous work (Simha *et al.*, 1989, Kerven & Geller 1989) we have investigated methods to improve the update behavior of this class tree. During this work both we and Simha independently realized that the numbering scheme effectively makes the tree unnecessary.

In this paper we will limit ourselves to one single update operation, (*is-a B C*), which asserts that *B* is a subclass of *C*. For now we assume that *B* is a previously unknown class, a limitation that will be relaxed in Section 3. A class tree is built by repeated calls to the *is-a* function. For instance, the tree in Figure 1 could have been built by a sequence such as *(is-a plant thing), (is-a animal thing), (is-a dog animal), (is-a mineral thing), (is-a bird animal), (is-a crane bird), (is-a eagle bird)*. Contrary to the impression created by this example, it is permissible to maintain several disconnected class trees belonging to the same hierarchy. This happens if the node *C* is *not* previously known. We will permit only one retrieval operation (*isa-p D E*), which returns true if *D* is a subclass of *E*, and nil otherwise.

In Section 2 and Section 3 of this paper we will show that a linear representation of a Schubert tree is sufficient for all necessary update and retrieval operations. In Section 4 we will argue that the list representation of a Schubert tree is ideal for the fine-grained parallelism as it is incorporated in the Connection Machine.[2] In Section 5 we will discuss an implementation of our linear class trees on the Connection Machine and show that it performs well in the temporal ranges required by Shastri's notion of computational effectiveness (Shastri 1988, p. 3). Section 6 describes limitations and future work, and Section 7 contains our conclusions.

2. The linear tree representation

2.1 *General description*

Given that all the information necessary for answering questions about subclass relations is contained in the number pairs, we may eliminate the explicit parenthood relations and represent the tree as a list of nodes. Of course it would be nice if we can choose a structure such that update operations become easy. It

turns out that this is the case if we generate a list of nodes from the tree by collecting nodes during a *left to right* preorder traversal.

In this section we will show that this representation conveniently discriminates between the nodes that need to be updated and the nodes for which the number ranges do not change at all. We will also show that the list can be updated consistently with the tree representation by adding a new child node in the list right after its parent node. Finally we will show how the pairs have to be updated such that the list stays the functional equivalent of a Schubert tree.

2.2 *Update rule and update range for the tree*

An important property of a Schubert tree is that the order of a set of siblings does not contain any relevant information. We make use of this fact by requiring that a new sibling is always added at the left-most position.

Theorem 1. Let a tree T be transformed into a list L by a left to right preorder traversal. If a node X is added to T as a child of a node P (as a left-most sibling), resulting in a tree T', then the following holds true. The list L' created from T' by a left to right preorder traversal will be identical to the list L'' created from L by inserting X to the immediate right of P.

Proof: Assume that a node X is added under a node P as its first descendent or as the left-most sibling of all its descendents. Assume farther that before X is added the node in L that comes immediately after P is N (if it exists). That is, L'' contains the consecutive subsequence (P,X,N). Clearly N must be the left-most child of the nearest ancestor of P (including P itself) that is to the right of the path from the root R to P.

It is well known that in a left to right preorder traversal all the nodes of a subtree come immediately after its root, and before any other nodes. Therefore, if X is added to T as described, then L' will contain the sequence (P,X,N). No other area of the tree will be influenced. Therefore L' is identical to L''. □

Figure (2b) shows two examples of possible positions of N. Figure (2a) shows the situation where P is interpreted as its own ancestor.

Let $\mathbf{P}(X)$ be the path in the tree T from the parent P of X to the root R. The path $\mathbf{P}(X)$ partitions the tree into three areas (see Figure 3):

1. The left part $L_T(X)$ of all nodes to the left of $\mathbf{P}(X)$.
2. The right part $R_T(X)$ of all nodes to the right of $\mathbf{P}(X)$.
3. The path $\mathbf{P}(X)$ itself, containing all the ancestors of X.

Theorem 2. If a new child node is added to a Schubert tree, then the following update rule holds in the tree:

1. There is no change to the pairs of $R_T(X)$.
2. For all the nodes of $L_T(X)$ the PRENUM and the MAXNUM have to be incremented by 1.
3. For all the nodes of $\mathbf{P}(X)$ the MAXNUM has to be incremented by 1.

Proof: Theorem 1 implies that the PRENUM of each node in $L_T(X)$ is incremented by 1, as its position in the right to left traversal order is delayed

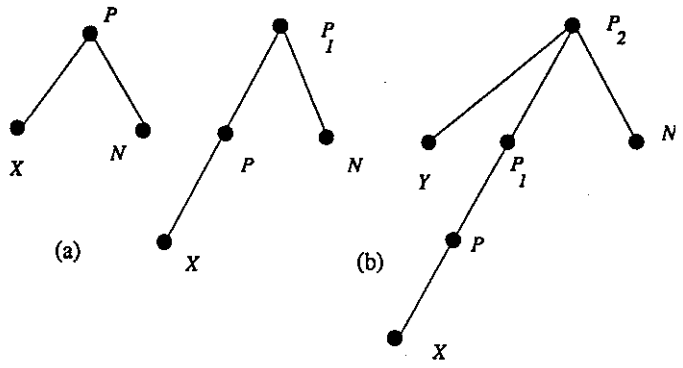


Figure 2. Possible positions of succeeding nodes.

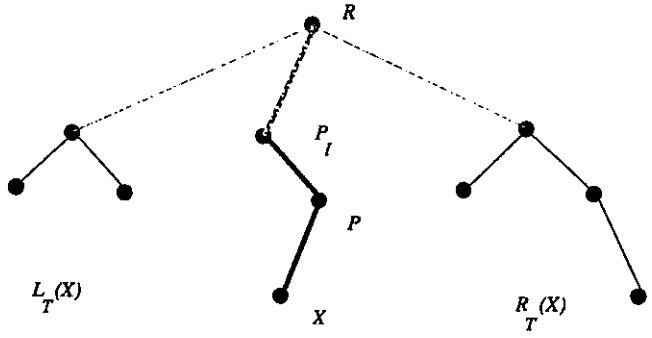


Figure 3. Left and right parts of a tree.

by one, due to the addition of X. Similarly, one can see that the PRENUM of each node in $R_T(X) \cup \mathbf{P}(X)$ is not changed, because these nodes come before X in the right to left traversal.

Since the PRENUM of each node in $L_T(X)$ is incremented by 1, the PRENUM of all descendents of each node in $L_T(X)$ is also incremented by 1, because they are all in $L_T(X)$. Hence, the MAXNUM of each node in $L_T(X)$, which is defined as the largest PRENUM of any of its descendent nodes, is also incremented by 1.

All descendents of each node in $R_T(X)$ are also in $R_T(X)$, and thus their MAXNUM is not changed. The descendent of highest PRENUM for each node in $\mathbf{P}(X)$ is either the new node X or a node in $L_T(X)$. Thus the MAXNUM of every node in $\mathbf{P}(X)$ is incremented by 1. □

2.3 *Correct update in the list representation*

So far we have shown how to update the tree representation correctly. Now we have to show how to do the same update for the list representation. This is not a trivial problem, because in the list representation the elements of $L_T(X)$ and $\mathbf{P}(X)$ are intermingled. We define $L_L(X)$ to be the part of the list representation to the left of X, and $R_L(X)$ to be the list representation to the right of X.

Theorem 3. The update rule for the list representation of a Schubert tree under the operation of adding a new leaf X as a child of a node P is as follows:

1. The PRENUM is maintained correctly if we add 1 to the PRENUM of every node N in $L_L(X)$ for which PRENUM(N) > PRENUM(P) and leave it unchanged otherwise.
2. The MAXNUM is maintained correctly if we add 1 to the MAXNUM of every node in $L_L(X)$ and leave it unchanged otherwise.

Proof: Using Theorem 1 it is obvious that the nodes in $R_T(X)$ are the same nodes as in $R_L(X)$. Using Theorem 2 we know that the PRENUM and MAXNUM of these nodes do not change.

$L_L(X)$ consists of the union of $L_T(X)$ and $\mathbf{P}(X)$. According to Theorem 2 the MAXNUM for all nodes in $L_T(X)$ and $\mathbf{P}(X)$ have to be incremented by 1. These are exactly the nodes in $L_L(X)$.

In order to update the PRENUM in $L_L(X)$ correctly, we need to distinguish which nodes in $L_L(X)$ belong to $\mathbf{P}(X)$ and which nodes belong to $L_T(X)$. Clearly the right to left preorder numbering will visit all the nodes in $\mathbf{P}(X)$ before it visits any nodes in $L_T(X)$. Therefore, the PRENUM itself tells us which nodes N belong to $L_T(X)$, namely the ones that have PRENUM(N) > PRENUM(P). □

The final question we have to address is how to construct the pair for the new node X itself. This is the subject of Theorem 4.

Theorem 4. The overall numbering scheme of the linear tree representation is maintained correctly if we assign X the number pair (MAXNUM(P), MAXNUM(P)). This has to be done after P has been updated according to Theorem 3.

Proof: This follows directly from our assumption that new nodes are added to the tree in the left-most sibling position. Every parent node has as its MAXNUM the largest PRENUM of any of its descendents, and by updating P we have precisely made space for X in P's number range. (Remember that updating P meant incrementing its MAXNUM by 1.) □

2.4 *Description of the algorithm*

The theorems proven in the previous sections provide justifications for the following serial algorithm that describes how to update the linear representation of a Schubert tree while adding one single new node which is always a leaf.

Later, we will show that this algorithm is a special case of an algorithm that permits adding a whole new subtree of independently developed class relations into an already existing tree. The root of the second tree may be attached under any node of the first tree as a left-most sibling. With this generalized algorithm it also becomes possible to add a new root on top of an already existing tree. The only cases that are not permitted by our generalized algorithm deal with structural changes to the tree. For instance, it is not permitted to say that 'A cat is a thing' and then later on to interpolate that 'A cat is an animal' and 'An animal is a thing'.

Suppose that the tree T has n nodes and its nodes are stored in a list L in the way described above, $L = (N_1, N_2, N_3, \ldots, N_p, N_{p+1}, \ldots, N_n)$. X is a new

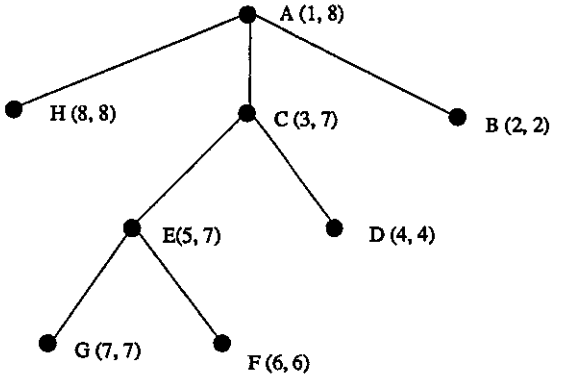
node and is going to be attached under node $N_p = P$ of T. PRENUM(N_k) and MAXNUM(N_k) represent the right–left preorder number of node N_k and the maximal preorder number among its descendents in the tree.

```
ATTACHING_NODE(X, Np, L)
BEGIN
        FOR k := 1 to p DO
        BEGIN
                MAXNUM(Nk) := 1 + MAXNUM(Nk);

                IF PRENUM(Nk) > PRENUM(Np)
                THEN PRENUM (Nk) := 1 + PRENUM(Nk);
        END;

        PRENUM(X) := MAXNUM(Np);
        MAXNUM(X) := MAXNUM(Np);
        INSERT X into L right after Np;
END.
```

We make the assumption that when the tree is empty, just assign PRENUM(X) := 1 and MAXNUM(X) := 1 and put X into the empty list. Figure 4 shows a tree before adding a new node X, and Figure 5 shows the same tree after adding it.

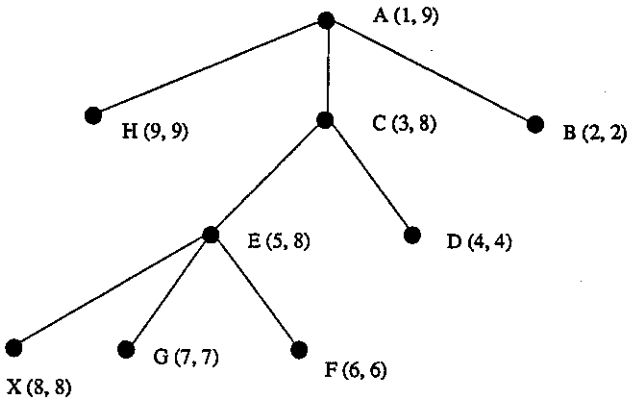
3. Extended algorithm for attaching a tree under a node

The case of connecting two trees T_1, T_2, such that the root of T_1 stands in a subclass relation to a node of T_2 is a simple generalization of the algorithm for adding a single node. The operation of inserting one node into a list is replaced by splicing the list of T_1 into the list of T_2 to the immediate right of the parent node P.



(A (1, 8) H (8, 8) C (3, 7) E (5, 7) G (7, 7) F (6, 6) D (4, 4) B (2, 2))

Figure 4. A tree with PRENUM and MAXNUM.



(A (1, 9) H (9, 9) C (3, 8) E (5, 8) X (8, 8) G (7, 7) F (6, 6) D (4, 4) B (2, 2))

Figure 5. The tree from Figure 4, with one node added.

The necessary operations for updating the number ranges for T_2 change in so far that we are now adding a 'supernode' that contributes more than an increment of 1 to the nodes in L_2 that need updating. The increment to be used is identical to the number of nodes of T_1 which is itself identical to MAXNUM(R_1). (L_2 is the list representation of the nodes in T_2, and R_1 is the root of T_1). The only other question is how the number ranges of T_1 itself have to be updated. This is the subject of the following theorem, for which we will first prove a lemma.

Definition 1: *The operation* of shifting a tree T by C *is defined as adding a constant C to the PRENUM and the MAXNUM of every node in T.*

Lemma 1: Shifting the linear representation of a tree T does not change the retrieval properties and the update properties of T.

Proof: In order to verify that a class B is a subclass of a class A we need to compare the number pairs of those two classes. B is a subclass of A iff the interval (PRENUM(B), MAXNUM(B)) of B is a subinterval of the interval (PRENUM(A), MAXNUM(A)) of A. If we add C to all number pairs of T, then we will get (PRENUM(B)+C, MAXNUM(B)+C) for B and (PRENUM(A)+C, MAXNUM(A)+C) for A, and the subinterval relation will not change. This shows the insensitivity of retrieval operations with respect to adding a constant C to all number pairs.

Update operations rely only on the number pairs that the tree has at the time of updating and are therefore naturally independent of an added constant. □

Theorem 5: In order to attach a tree T_1 at its root R_1 under a node P of a tree T_2 using the linear representations L_1 and L_2 of those two trees the following has to be done. The value that MAXNUM(P) had at the time *before* L_2 was updated has to be added to the PRENUM and MAXNUM of every node of L_1, i.e. L_1 has to be shifted by MAXNUM(P). L_2 has to be updated as in Theorem 3, except that every increment by 1 has to be replaced by an increment by M, the number of nodes of L_1.

Proof: Following our Lemma it is obvious that the segment representing L_1 after the splice operation will be internally numbered in a consistent way if

we shift L_1 by MAXNUM(P). What is left to show is that the new number pairs of L_1 are consistent with the updated pairs of L_2.

Clearly, R_1 will be numbered before the splicing operation with (1, M). All the nodes that were originally to the right of P will not be changed. All the nodes that were originally to the left of P, including P, will now be delayed in their preorder numbering by M positions. The rest of the proof is analog to the proofs in Section 2.2 and Section 2.3. □

We will now present the extended algorithm. Assume that T_1 is stored as $L_1 = (N_1, N_2, N_3, \ldots, N_p, N_{p+1}, \ldots, N_n)$ and T_2 as $L_2 = (M_1, M_2, \ldots, M_m)$. $M_1 = R_2$ is the root of T_2 and M_1's number pair is (1, MAXNUM(M_1)); the node $N_p = P$ of T_1 has (PRENUM(N_p), MAXNUM(N_p)). According to the above assumptions, the result of attaching T_2 under N_p into T_1 will be:

$$L_1' = (N_1', N_2', N_3', \ldots, N_p', M_1', M_2', \ldots, M_m', N_{p+1}, \ldots, N_n).$$

In the above equation Φ' denotes a node with the same class as Φ, but with its number pair changed according to Theorem 5.

```
ATTACHING_TREE(L2, Np, L1)
BEGIN
        MAX_Np_NUM := MAXNUM(Np);
        FOR k :=1 to p DO
        BEGIN
                MAXNUM(Nk) := MAXNUM(M1) + MAXNUM(Nk);

                IF PRENUM(Nk) > PRENUM(Np)
                THEN PRENUM(Nk) := MAXNUM(M1) + PRENUM(Nk);
        END;

        FOR k :=1 to m DO
        BEGIN
                PRENUM(Mk) := PRENUM(Mk) + MAX_Np_NUM;
                MAXNUM(Mk) := MAXNUM(Mk) + MAX_Np_NUM;
        END
        INSERT L2 into L1 right after Np;
END.
```

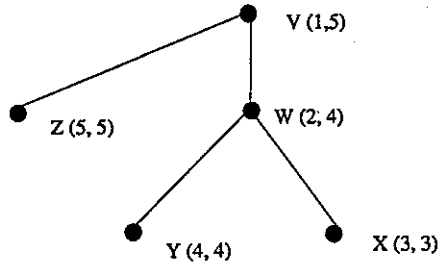
An example of a complete update operation is shown in Figure 7. We attach the tree from Figure 6 under the tree from Figure 4 such that the node V becomes a child of the node G.

4. Parallelizing the algorithm

In this section we will first discuss a few basic concepts of the Connection Machine. This discussion alone should make it clear that the update algorithm can be parallelized with moderate effort, while the retrieval algorithm is almost a paradigm example for the use of the Connection Machine and can be completely parallelized with a few lines of code.

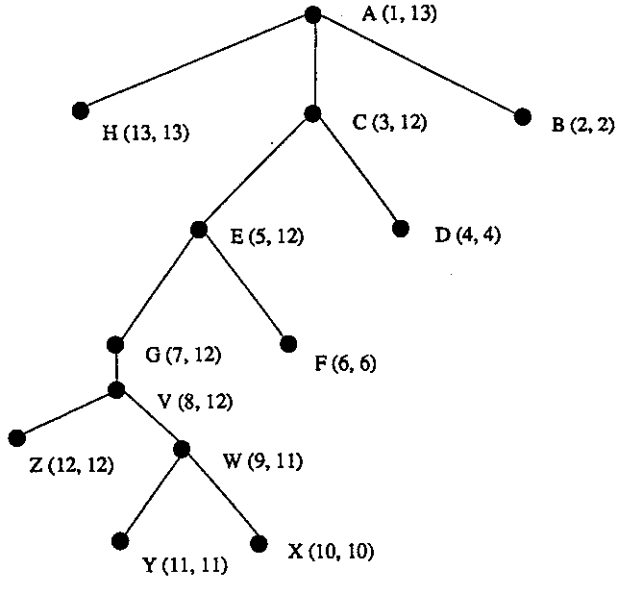
4.1 *The Connection Machine*

The Connection Machine is a single-instruction multiple-data (SIMD) high-power fine-grained parallel processing system. The basic architecture consists of a front end and a set of small interconnected processors. The front end is usually a VAX



(V (1, 5) Z (5, 5) W (2, 4) Y (4, 4) X (3, 3))

Figure 6. The tree to be attached.



(A (1, 13) H (13, 13) C (3, 12) E (5, 12) G (7, 12) V (8, 12) Z (12, 12) W (9, 11) Y (11, 11), X (10, 10) F (6, 6) D (4, 4) B (2, 2))

Figure 7. The result of attaching a tree under G.

or a Symbolics. The largest Connection Machine configuration consists of 64K processors. Due to the astronomic price of this full configuration, smaller versions with fewer processors are also available.

Every one of the small processors has a limited amount of its own memory, depending on the model, between 8K and 64K. Sixteen processors are implemented on one chip, and the chips are interconnected as a hypercube. Nevertheless, the user has a wide degree of flexibility in how he wants the processors to be organized logically. To do this he can define so-called virtual processor sets using software commands.

The two most useful configurations (according to our experience) organize all available processors as one long chain, or as a two-dimensional grid.

The Connection Machine can be programmed with Paris, a low-level language, with a dialect of C, or with a dialect of LISP. The dialect of LISP is called '*LISP' (pronounced Star LISP; Thinking Machines Corporation 1988) and is based on the Common LISP standard, incorporating necessary extensions and limitations. While the small processors are mostly using these extensions, the front end processor has access to a full implementation of Common LISP.

The most important entity of *LISP (and Connection Machine software in general) is a construct called a Pvar (parallel variable). A Pvar is a variable with one name but many memory locations. More precisely, there is one memory location on every small processor that corresponds to this variable name.

LISP has two kinds of special constructs which are marked by a preceding '' or a succeeding '!!' respectively. For instance, the *LISP command (+!! a b) will add the values of the two Pvars a and b *in every single processor*. The *LISP command (*sum c) will return the sum of the values of the Pvar c *taken from all processors*.

One more important concept will finish this minimal introduction. It is possible to select at any time a set of active processors. Only these processors will be considered for any operation executed. This construct permits selective operation on processor groups in parallel and is therefore of fundamental importance.

In this research we are using the CM2 of the NPAC computing center in Syracuse, and we are running it as a linear configuration of 8K processors.

4.2 *Parallelizing the update algorithm*

This section presents a parallel version of the update algorithm used when adding a new node. We will informally explain the overall approach and then give an algorithmic description of the part of the program that updates the pairs. Details that distract from the general principles are omitted.

Our basic approach is that one class node is represented by one processor. We define three Pvars that store the name of the class of this processor, the value of the PRENUM for that class, and the value of the MAXNUM for that class. A list (standing for a tree) is represented by a segment of consecutive processors. Because it is possible that several trees exist at the same time (as before any attach operation) we distinguish between different lists by assigning all processors of one list the same segment number, which is stored in another Pvar. Global variables on the front end keep track of the number of processors used and of the number of segments.

Entering a new node consists of the following steps:

1. Assume that a contiguous block of processors starting with the left-most processor (processor 0) is in use.
2. Load the new child node into the first unused processor.
3. Identify the parent node. This is basically done by asking all processors in parallel whether they are the parent node, and only one of them will respond.
4. Copy the whole block of all processors starting right after the parent node and up to the last processor by one position to the right. More precisely, we are of course not copying processors, but we are copying their contents to the right. The important point here is that all processors are copied from the point of view of *LISP by one single operation per Pvar used. There is no looping or recursion, and the operation does not change whether three nodes

are moved, or 3000 nodes are moved.
5. Copy the right-most processor in use, which contains the new child node, to the position right after its parent node.
6. Update the PRENUM and MAXNUM of the child processor in the same way as described for the serial case.
7. Update the PRENUM and MAXNUM of all processors to the left of the processor that contains the parent node (including the parent node). This is clearly also done in parallel and independent of the number of nodes in the tree.

The last item in above explanation corresponds to the FOR loop in our serial algorithm.

```
FOR k := 1 to p DO
BEGIN
      MAXNUM(N_k) := 1 + MAXNUM(N_k);

      IF PRENUM(N_k) > PRENUM(N_p)
      THEN PRENUM(N_k) := 1 + PRENUM(N_k);
END;
```

A short look at this loop shows why it is an ideal candidate for parallelization. Every single iteration in this loop is completely independent from every other iteration. It is therefore possible to execute all iterations in parallel.

We would like to express this fact as a parallel algorithm. As is well known, most algorithm descriptions are given in a form of pseudo-PASCAL. However, the advent of parallel algorithms makes it impossible to be limited to such a language, disproving once and for all claims that the algorithmic level is completely abstract and separate from the language level of computing.[3] Our own algorithmic description is therefore a mixture of *LISP and PASCAL without care about parentheses, prefix notation and exact function names.

```
ACTIVATE-PROCESSORS-WITH
          (SELF-ADDRESS!! >!! 0!!) AND!!
          (SELF-ADDRESS!! <=!! (!! p))
DO BEGIN
          MAXNUM!! :=!! 1!! +!! MAXNUM!!

          IF!! PRENUM!! >!! (!! PRENUM(N_p)) THEN!!
                PRENUM!! :=!! 1!! +!! PRENUM!!
END
```

SELF-ADDRESS!! is a Pvar that contains in every processor the running number of this processor, starting with 0 and up to 8K. PRENUM!! and MAXNUM!! are Pvars containing the corresponding PRENUM and MAXNUM values for every processor. 0!! and 1!! are 'constant Pvars' containing the values 0 and 1 in every processor. The operations >!!, <=!!, IF!!, +!!, and :=!! do exactly what one would expect them to do, except that they do it in parallel for every single processor. Finally, the (!!) prefix operator that is used e.g. in (!! p) transforms a single value into a parallel value.

The first four lines select a set of active processors, and therefore these lines correspond functionally to defining the limits of the FOR loop. The fifth line

adds 1 to the MAXNUM value in every single activated processor. The next two lines add 1 to the PRENUM in every single processor, under the condition that the previous value of the PRENUM is larger than the PRENUM of the parent node. Note that all these operations are done in parallel.

4.3 *Parallelizing the retrieval algorithm*

For the retrieval algorithm two classes are given, and we want to verify whether they stand in a superclass–subclass relation, by comparing their number pairs. Thus, the front end processor broadcasts a signal to all used processors (in parallel), asking them whether they contain the superclass or the subclass. Under normal circumstances only two of the processors will respond (at most) and we can then compare their number ranges.

5. Implementation

5.1 *The program*

The program is completely implemented following the descriptions given above. Initial test clauses for a new *is-a* assertion have to decide whether the given information does not contradict previously entered information, is not already known, and does not contradict the required tree format of the class hierarchy.

If all these conditions are passed, a number of parameters have to be computed, e.g. where the segment starts that has to be moved, etc. We have somewhat simplified our previous explanation of the algorithmic steps by ignoring that an attach operation might have to be done to the left, or to the right, which slightly changes these parameters. We have also ignored intermediate segments not involved in the actual operation, which complicates the movements slightly.

After those steps, the parallel version of the attach algorithm, i.e. of the second algorithm, is executed. Finally, processors that were set and contain now invalid information are erased.

There are two parts of this solution which could be improved. The first is the unpleasant fact that *LISP cannot process symbolic atoms in parallel. We have claimed before that there is a Pvar with the name of every class. This is not true, because we could not test for these atoms in parallel. Rather, at the first mention of every atom a hash table on the front end processor is used to translate the class name into an integer, and all further operations are performed with these integers.

One (unattractive) alternative would be to store the class names as arrays of characters. Other than that, we would hope that future versions of *LISP permit storage and comparison of atoms on the individual processors.

A second problem is that the copying of groups of processors is done in a wasteful way. Even if 8K of processors are available, it would currently be impossible to connect two trees of size 4K each, because the algorithm would involve copying a 4K segment to a group of unused processors, which would limit the second segment in the extreme to the size 0. A safe version of the program should limit the use of nodes to groups of under 8K/3 size. Alternatively the reuse of one processor for several nodes has to be considered.

5.2 *The test data*

A program that tries to emulate human performance will need to deal with several 10,000s of nodes, so it is necessary to do testing with realistic problem sizes. To

avoid the typing of several thousand assertions, test data were generated mechanically. To make sure that these test data did not result in any erroneous cases, which are uninteresting for our efficiency testing, the following strategy was used.

A function built on (gensym) was written that creates new and unique LISP atoms. This function takes three parameters, i, j, and k. Three pools of LISP atoms are generated. The first pool is of size i. The second pool is of size $i*j$, and the third pool is of size $i*i*k$.

After these steps, every atom of the first pool is paired with j atoms of the second pool. Every atom of the second pool is paired with k atoms of the third pool. By doing this we create a set of $i*j + i*j*k$ pairs of atoms. Every one of these pairs is then interpreted as one is-a assertion and stored in a list. These $i*j + i*j*k$ assertions are then randomized in their order. The result list might look as follows. (*(is-a cl0012 cl0031) (is-a cl0021 cl0056) . . .*).

The described setup corresponds to having i independent two-level class trees with j intermediate nodes in every tree and $j*k$ leaf nodes in every tree.

Testing means that all is-a assertions of the randomized list will be executed. The time of executing the whole list is recorded, and a number of retrieval operations (is an X a Y?) are executed and timed after that.

5.3 *The test results*

We will now present the results of our timing tests, which show that learning of a new is-a relation can be done even for several thousand nodes in under 1 second, and that retrieval can be done even for the same large trees in under 0.5 s. All experiments were done with compiled *LISP. However, the *LISP compiler is more of a macro expander and does not compile every structure, even if careful type declarations are given (which we did).

An issue that has to be addressed before showing actual timings are the garbage collections of *LISP. Garbage collections are a fact of life in LISP, and for large problems it becomes difficult as well as unrealistic to try to ignore them, or to try to eliminate the time component due to garbage collections. Therefore, when we report the times of building a system of is-a trees with several thousand nodes, the garbage collection times are distributed throughout those times.

On the other hand, if a garbage collection occurs during a retrieval operation which takes at most 5 seconds (namely when we repeat the same operation 10 times to average the timing!) then 10 seconds of garbage collection do serious harm to our measurement, and we will ignore such values.

In Table 1 we show the times it took to build class hierarchies. Some experiments were repeated, and for those we will show all results. For every test run we show the problem size, (that is i, j, k), the total number of is-a relations, the user run time as reported by *LISP, and the user run time per is-a assertion. This last number is, of course, the most interesting to us.

Visual inspection of the above data shows a small increase of the time for one assertion, when changing the problem size from 12 assertions to 4352 assertions. More than doubling the problem size from 1872 assertions to 4352 assertions results in a change of about 5%. Even for 4352 assertions, which is quite a decent size for a realistic system, update times stay well under 1 s.

For retrieval tests we used three groups, the first of which has three subgroups of data items.

Table 1. Run-times for update operations

Problem size	*Number of assertions*	*Total run-time*	*Run-time/assertion*
(2 2 2)	12	4.8	0.4
(3 3 3)	36	27.26	0.75
(3 3 3)	36	15.89	0.44
(4 4 4)	80	49.18	0.613
(5 5 5)	150	107.43	0.716
(6 6 6)	252	176.08	0.698
(6 6 6)	252	193.57	0.768
(7 7 7)	392	296.43	0.755
(8 8 8)	576	462.22	0.802
(9 9 9)	810	650.96	0.802
(9 9 9)	810	639.87	0.789
(10 10 10)	1100	896.83	0.814
(11 11 11)	1452	1213.8	0.835
(12 12 12)	1872	1610.66	0.860
(12 12 12)	1872	1583.57	0.845
(13 13 13)	2366	2065.41	0.872
(14 14 14)	2940	2633.15	0.8955
(15 15 15)	3600	3376.99	0.9377
(16 16 16)	4352	3924.77	0.901

1. Both atoms are known and are also known to stand in a class–subclass relation. (This case will be subdivided below.) In fact we retrieve the atoms directly from the randomized list of test data.
2. Both atoms are known, but they are selected from different sets, such that the is-a relation does not hold.
3. One of the atoms is known, one is a newly generated atom.

As might be expected, the first group is the slowest, and the most interesting. It is the slowest because the PRENUM!! and MAXNUM!! are never retrieved for an atom that is not known; therefore only in the first case all operations are executed, and the time is therefore relatively the longest.

The first group needs to be subdivided as follows:

tm: One atom is from the top pool, and one item is from the middle pool.
mb: One atom is from the middle pool, and one item is from the bottom pool.
tb: One item is from the top pool, and one item is from the bottom pool.

Because no tree traversal is performed, the top–bottom cases that span two levels should not be slower than the other two cases.

Table 2 shows run-times for retrieval operations. The first row defines again the problem size and the total number of is-a assertions processed. The first column defines the type of operation according to the above subdivision, i.e. 1.tm 1.mb 1.tb 2. or 3. The numbers in the fields show the run-times.

Many retrieval operations were repeated 10 times, to permit averaging of the run-times. Those cases are marked by prefixing the subdivision number by '10×.' In that case the time is *not* divided by 10, but given as required for 10 runs. As mentioned above, we omit runs that were interrupted by garbage collections. A '–' in Table 2 indicates a garbage collection. All times are given in seconds.

Table 2. Retrieval times

Problem size/ total assertions	*(3 3 3) 36*	*(6 6 6) 252*	*(9 9 9) 810*	*(12 12 12) 1872*	*(16 16 16) 4352*
10 × 1.tm	–	3.88	–	3.7	4.06
1.tm	0.38	0.39	0.4	0.37	0.42
10 × 1.mb	3.7	3.86	3.73	3.67	–
1.mb	0.4	0.38	0.37	0.36	0.41
10 × 1.tb	3.7	3.87	3.73	3.67	4.0
1.tb	0.4	0.38	0.37	–	0.39
10 × 2	3.73	1.33	1.27	1.26	1.36
2	0.39	0.13	0.12	0.12	0.14
10 × 2	–	1.3	1.27	1.27	1.35
2	0.26	0.12	0.14	0.13	0.14
10 × 3	1.11	–	1.19	1.09	1.27
3	0.11	0.12	0.12	0.11	0.14
10 × 3	1.22	1.23	1.19	1.11	1.26
3	0.12	0.12	–	0.11	0.12

Visual inspection of the data confirms that retrieval times barely change with the number of nodes in the system, and retrieval times do not grow for two-level (tb) access as compared to one-level (tm, mb) access. As expected, times in group 1 are considerably longer than times in group 2 and group 3. The longest single access time measure for all operations is 420 ms. This is well in the requested range of 'a couple of 100 ms' for human-like information retrieval.

6. Current limitations and future work

Our current theory is limited in that it permits only one kind of update operation. We intend to investigate more sophisticated update operations that permit the restructuring of an existing tree.

The current version of the program permits only one class hierarchy, although Schubert *et al.*'s theory and common sense require the use of several independent hierarchies. There is no difficulty in achieving this goal. The only change necessary is to add several new Pvars. The same processor can be used as a node of several independent class hierarchies.

Schubert *et al.* use a special-purpose reasoner for part hierarchies. Given our own previous research on part hierarchies (Geller 1988, 1991, Geller & Shapiro 1987), this is an obvious choice for continuing our work on parallelizing special-purpose reasoners.

7. Conclusions

Following work by Shastri (1988, 1989) and Schubert *et al.* (1983, 1987) on special-purpose reasoners we have shown a parallel algorithm that maintains and updates a large system of subclass assertions.

The algorithm is based on a linear representation of a tree with nodes that are numbered according to a preorder traversal scheme. We have proven in detail that this linear representation has the full power of the original tree representation with respect to update and traversal.

It was argued that the parallel algorithm is ideal for the fine-grained parallelism exhibited by the Connection Machine. Such a parallel implementation was presented and discussed in detail.

Timing experiments have shown that retrieval times are practically constant with respect to tree size and are well under 0.5 s even for systems with several thousand nodes. Update times grow very moderately with the size of the class tree, but even for an example with over 4000 assertions, learning times for new subclass assertions are well under 1 s.

We see this successful implementation as the proof that it is possible to solve an important class of limited inference problems by using massive parallelism. However, this work differs from Shastri's in two major ways. The problem that we are solving is much more limited. We are interested only in updating and retrieving relations in a class hierarchy. No attributes are assigned to the classes, and no inheritance or recognition is intended. On the other hand, we are using existing parallel hardware, as opposed to a connectionist simulation on a serial computer.

Acknowledgements

We would like to thank NPAC and RADC full-heartedly for giving us unbureaucratic and *free* access to the Connection Machine. We would like to

thank Thinking Machines Corporation for building such a wonderful machine. We also thank Dr Yehoshua Perl for commenting on an earlier draft of this paper.

Notes

1. This work was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by and operates under contract to DARPA and the Air Force Systems Command, Rome Air Development Center (RADC), Griffiss Air Force Base, NY, under contract F306002-88-C-0031.
2. Connection Machine is a Trademark of Thinking Machines Corporation.
3. It is hard to resist reporting a student quip on the topic. 'An algorithm is when the professor cannot remember the semicolon rules of PASCAL, or when he is too lazy to declare his variables.'

References

Anderson, J. R. (1983) A spreading activation theory of memory. *Journal of Verbal Learning and Verbal Behavior*, **22** 261–295.

Brachman, R. J. (1979) On the epistemological status of semantic networks. In Findler, N. (ed.), *Associative Networks*, (New York: Academic Press), pp. 3–50.

Brachman, R. J., Fikes, R. E. and Levesque, H. J. (1983) Krypton: A functional approach to knowledge representation. *IEEE Computer*, **16**(10): 67–73.

Brachman, R. J. and Levesque, H. J. (1985) *Readings in Knowledge Representation* (Los Altos, CA: Morgan Kaufmann).

Brachman, R. J., Levesque, H. J. and Reiter, R., (eds) (1989) *Principles of Knowledge Representation and Reasoning* (San Mateo, CA: Morgan Kaufmann).

Brachman, R. J. and Schmolze, J. (1985) An overview of the KL-ONE knowledge representation system. *Cognitive Science*, **9**(2): 171–216.

Collins, A. M. and Loftus, E. F. (1975) A spreading activation theory of semantic processing. *Psychological Review*, **82**(6): 407–428.

Collins, A. and Smith, E. E. (eds) (1988) *Readings in Cognitive Science* San Mateo, (CA: Morgan Kaufmann).

Evett, M., Hendler, J. and Spector, L. (1990) Parka: Parallel knowledge representation on the connection machine. Technical Report UMIACS-TR-90-22, Department of Computer Science, University of Maryland.

Fahlman, S. E. (1979) *NETL: A System for Representing and Using Real-World Knowledge* (Cambridge, MA: MIT Press).

Geller, J. (1988) A Knowledge Representation Theory for Natural Language Graphics. PhD thesis, SUNY at Buffalo, CS Department. Report TR 88-15.

Geller, J. (1991) Propositional representation for graphical knowledge. *International Journal of Man-Machine Studies*, **34**: 97–131.

Geller, J. and Shapiro, S. C. (1987) Graphical deep knowledge for intelligent machine drafting. In *Tenth International Joint Conference on Artificial Intelligence* (Los Altos, CA: Morgan Kaufmann), pp. 545–551.

Horty, J. F. and Thomason, R. H. (1988) Mixing strict and defeasible inheritance. In *Seventh National Conference on Artificial Intelligence* (San Mateo, CA: Morgan Kaufmann), pp. 427–432.

Kerven, D. S. and Geller, J. (1989) Knowledge acquisition for a special purpose type-reasoner. Unpublished manuscript, CIS Department, New Jersey Institute of Technology.

MacGregor, R. M. (1988) A deductive pattern matcher. In *Seventh National Conference on Artificial Intelligence* (San Mateo, CA: Morgan Kaufmann), pp. 403–408.

Nebel, B. and von Luck, K. (1987) Issues of integration and balancing in hybrid knowledge representation systems. In Morik, K., (ed.) *GWAI-87* (Berlin: Springer Verlag), pp. 114–123.

Padgham, L. (1988) A model and representation for type information and its use in reasoning with defaults. In *Seventh National Conference on Artificial Intelligence* (San Mateo, CA: Morgan Kaufmann), pp. 409–414.

Quillian, M. R. (1968) Semantic memory. In Minsky, M. L. (ed.), *Semantic Information Processing* (Cambridge, MA: MIT Press), pp. 227–270.

Schubert, L. K. Papalaskaris, M. A. and Taugher, J. (1983) Determining type, part, color, and time relationships. *Computer*, **16**(10): 53–60.

Schubert, L. K., Papalaskaris, M. A. and Taugher, J. (1987) Accelerating deductive inference: special methods for taxonomies, colors and times. In Cercone, N. and McCalla, G. (eds). *The Knowledge Frontier* (New York: Springer Verlag), pp. 187–220.

Shastri, L. (1988) *Semantic Networks: An Evidential formalization and its Connectionist Realization* (San Mateo, CA: Morgan Kaufmann).

Shastri, L. (1989) Default reasoning in semantic networks: a formalization of recognition and inheritance. *Artificial Intelligence*, **39**(3): 283–356.

Simha, A. R., Sahoo, A. and Geller, J. (1989) An acquisition oriented knowledge organization to accelerate deductive inference. Unpublished manuscript, CIS Department, New Jersey Institute of Technology.

Stein, L. A. (1989) Skeptical inheritance: computing the intersection of credulous extensions. In *Eleventh International Joint Conference on Artificial Intelligence* (San Mateo, CA: Morgan Kaufmann), pp. 1153–1158.

Thinking Machines Corporation (1988) **Lisp Reference Manual* (Cambridge, MA: Thinking Machines Corporation) version 5.0 edition.

Touretzky, D. S. (1986) *The Mathematics of Inheritance Systems* (San Mateo, CA: Morgan Kaufmann).

Vilain, M. (1985) The restricted language architecture of a hybrid representation system. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (San Mateo, CA: Morgan Kaufmann), pp. 547–551.

Wilson, S. S. (1989) Neural computing on a one dimensional SIMD array. In *Eleventh International Joint Conference on Artificial Intelligence* (San Mateo, CA: Morgan Kaufmann), pp. 206–211.

Woods, W. A. (1975) What's in a link? foundations for semantic networks. In Bobrow, D. G. and Collings, A. M. (eds), *Representation and Understanding* (New York: Academic Press), pp. 35–82.