Parallel Processing for Artificial Intelligence 3
J. Geller, H. Kitano and C.B. Suttner (Editors)
© 1997 Elsevier Science B.V. All rights reserved.

Parallel Operations on Class Hierarchies with Double Strand Representation

Eunice (Yugyung) Lee* and James Geller†
Department of Computer and Information Sciences
New Jersey Institute of Technology
Newark, NJ 07102

This paper continues a series of papers dealing with the problems of (1) fast verification of the existence of a transitive relation in an IS-A hierarchy, and (2) dynamic update of such a hierarchy. As in our previous work, a directed acyclic graph (DAG) of IS-A relationships is replaced by a set of nodes, annotated by number pairs, and stored on a massively parallel computer. In this paper a new mapping of this set of nodes onto the processors is described, called the Double Strand Representation (DSR). The DSR improves the processor usage compared to our previously used Grid Representation (GR). This paper shows IS-A verification and number pair propagation algorithms for the Double Strand Representation. Test runs on a CM-5 Connection Machine[3] are reported.

1. INTRODUCTION

Most Knowledge Representation systems, as well as all object-oriented languages and databases, use an IS-A hierarchy as their backbone. An efficient hierarchy encoding technique would aid all these areas, especially for large hierarchies. The IS-A hierarchy has been especially important in the KL-ONE family of Knowledge Representation (KR) systems [1–9].

Object-oriented systems, based on SIMULA [13] and Smalltalk [14], always incorporate generalization hierarchies with inheritance behavior. Object-oriented methods have been applied to the design of programming languages, e.g., C++ [15], type systems [16], object-oriented extensions of existing languages, e.g., CLOS [17], and object-oriented database systems such as ORION [10], O_2 [11], and ONTOS [12].

Given the importance of the IS-A hierarchy, one would like to achieve the fastest possible processing for query and update operations in this hierarchy. If we assume that it is known that a Mammal is an Animal, a Dog is a Mammal, and a Collie is a Dog, then we want

*This research was (partially) done under a cooperative agreement between the National Institute of Standards and Technology Advanced Technology Program (under the HIIT contract, number 70NANB5H1011) and the Healthcare Open Systems and Trials, Inc consortium.
†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.
[3]The Connection Machine is a trademark of Thinking Machines Corp.

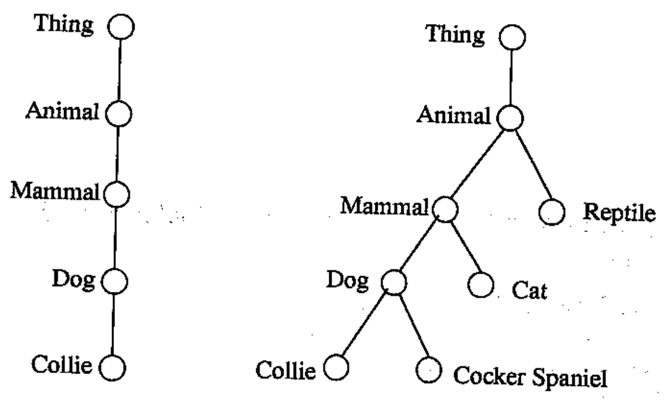


Figure 1. IS-A Hierarchy.

to quickly answer "yes" to the query whether a Collie is an Animal (Figure 1). We also want to be able to quickly update the hierarchy, e.g., when adding the facts that Cocker Spaniels are Dogs, Cats are Mammals, and Reptiles are Animals.

In the past we have been especially interested in techniques where the response time for an IS-A query does not depend on the length of the chain of IS-A links that must be traversed to answer the query. Besides these techniques, our tool of choice for achieving fast query and update operations is fine-grained parallelism. This raises the question of how to map the IS-A hierarchy onto the available space of processors. The most obvious intuitive choice is to assign every class of the hierarchy to a single processor. However, this intuitive choice does not carry over to the links between classes. If the whole hierarchy were known at the beginning of system design, one could opt for a strong form of isomorphism, where every IS-A link is implemented as a hardware link. However, our basic assumption is that Artificial Intelligence is not intelligence at all, if knowledge structures cannot be updated dynamically. Therefore, the isomorphism solution would require dynamic hardware changes as part of any update of the IS-A hierarchy, a solution that is currently still not practical. The idea of custom-made hardware is also not appealing to us.

The solution that we have been using in a series of papers [23,25–30] has been to eliminate the need for the IS-A links as much as possible, while still maintaining all the knowledge that is contained in the IS-A hierarchy. In this way, we do not have to worry about the mapping of the IS-A links onto the actual hardware.

Like other researchers, we are representing the classes of an IS-A hierarchy by nodes in a graph. The IS-A links are represented by the directed edges of this graph. In our first paper on the subject [26] it was shown that for the special case of a tree-shaped IS-A hierarchy of nodes, the hierarchy could be replaced by a linear order of the same nodes, together with *one* number pair assigned to each node. The assignment of number pairs was based on an encoding due to Schubert [31].

Later on we extended the representation of IS-A hierarchies "without explicit IS-A

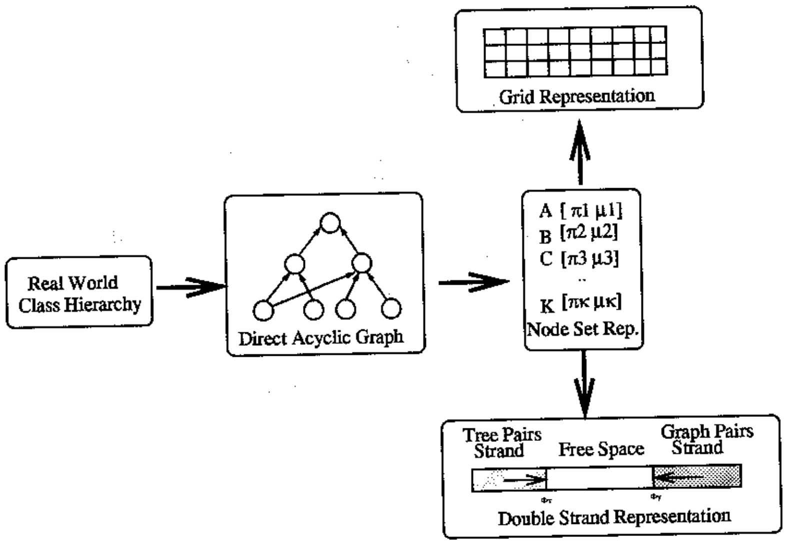


Figure 2. Three Step Mapping

links" to directed acyclic graphs (DAG). In this representation, *several* number pairs became necessary at some nodes. The assignment of these number pairs was based on an extension of [31] by Agrawal *et al.* [18]. While doing this, we were able to prove that the linear order used in [26] is not necessary at all. Rather, a set of nodes with an associated number pair(s) at each node could perfectly represent a DAG-shaped IS-A hierarchy without explicitly maintaining the IS-A links [23].

Unfortunately, the original fast algorithms [26] were possible due to the fact that one *number pair* was assigned to one processor, and *not* due to the fact that one *node* was assigned to one processor. So, in order to maintain the speed of processing, at least for queries, it became necessary to change the mapping of nodes onto processors. In [23,28] we mapped each node onto one column in a two-dimensional grid of processors. Every number pair of each node was assigned to a different processor (row) in its column.

A pleasant side effect of eliminating explicit links is that the time necessary to traverse them is also eliminated, giving, within certain limitations, constant time responses for transitive closure queries [26]. In other words, by using the Schubert/Agrawal representation, it takes as much time to verify that a Collie is an Animal as it takes to verify that a Collie is a Dog. By adding parallelism, updates can be performed in "almost" constant time. Experimental verification of this claim was provided in [26] for the case of trees.

All necessary details of Agrawal *et al.*'s encoding, our node set representation, and the GR will be described later on in this paper. However, we summarize now that the main feature of our previous work is a three step mapping (Figure 2). In the first step, an IS-A

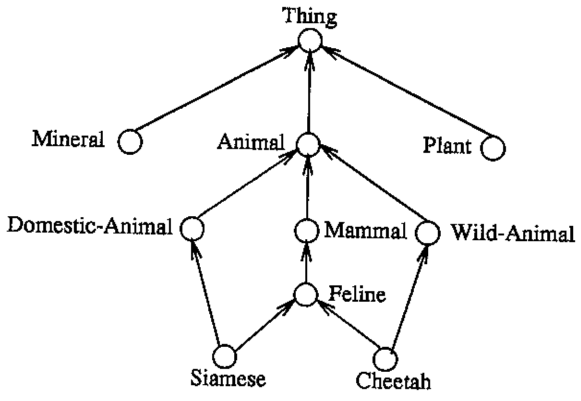


Figure 3. A Class Hierarchy

hierarchy of classes of the real world is mapped onto an isomorphic DAG of nodes, with one class per node. In the second step, the *hierarchy* of nodes is mapped into a *set* of those nodes, so that every node is annotated with one or more number pairs. In the third step, this node set and the associated number pairs are mapped onto the processor space of a fine-grained parallel computer. In brief, *class hierarchy* $\rightarrow$ *directed acyclic graph* $\rightarrow$ *node set + number pairs* $\rightarrow$ *processor space.*

In our previous papers, the third step was performed by organizing processors as a two-dimensional grid, with one node per column, and one number pair per row. Unfortunately, the GR causes a number of difficulties. We will describe the major problem now, while mentioning some other problems later in the paper. Because some nodes have only one number pair, while others have many number pairs, some columns might be virtually empty, while other columns might run out of processors, disrupting the functioning of our algorithms. Therefore, in this paper, we are showing a different representational approach.

The general idea of a three step mapping *class hierarchy* $\rightarrow$ *DAG* $\rightarrow$ *node set + number pairs* $\rightarrow$ *processor space* is still maintained. Indeed, the first two steps of the mapping are not changed at all. However, the assignment of number pairs to processors is changed in a way that eliminates the main problem of the GR described above. The new representation is called *Double Strand Representation* (DSR) and forms the main subject of this paper.

In addition, we will show parallel algorithms for fast IS-A queries in the DSR. We will also show parallel algorithms for an important operation of Agrawal *et al.*'s encoding [18], called *propagation*, in the DSR. Propagation is a necessary part of every update operation on a DAG with number pair annotation.

Work related to ours in the symbolic paradigm has been published, e.g., [32,33,39]. The PARKA system, a symbolic approach to combining KR with massive parallelism, has been described there. It is a frame system for handling large amounts of knowledge. It is implemented on the Connection Machine, and its temporal behavior has been extensively

Class Name	*Number Pair*
Thing	[1 10]
Animal	[3 9]
Plant	[2 2]
Mineral	[10 10]
Mammal	[5 8]
Domestic-Animal	[9 9](8 8)
Wild-Animal	[4 4](7 7)
Feline	[6 8]
Siamese	[8 8]
Cheetah	[7 7]

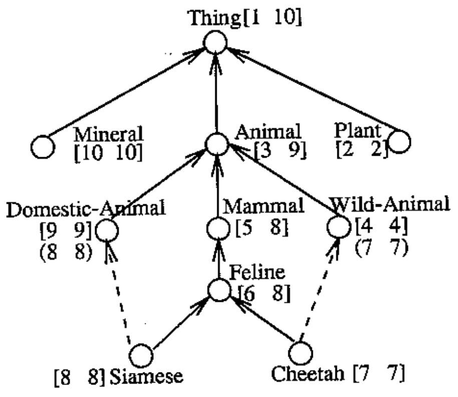


Figure 4. Node Set Representation for Class Hierarchy

tested. The newer version runs on an IBM SP2 [34].

Neural network approaches close in spirit to ours are, e.g., [40,42–46]. Shastri's work [40,41] combines massive parallelism implemented on a neural network simulator with a well defined, limited inference approach. According to Shastri [42], the distinction between the processes of a special-purpose reasoner and a general-purpose reasoner is akin to two human modes of reasoning, namely, reflexive reasoning and reflective reasoning. Sun [44], on the other hand, presents an intensional neural network approach of reasoning based on the semantic closeness of concepts. His work implements inheritance employing massive parallelism.

In Section 2 we discuss the numeric encoding of class hierarchies and the node set representation. In Section 3 the Double Strand Representation is presented. In Section 4 constant time subclass verification in the Double Strand Representation is discussed. Section 5 presents massively parallel propagation algorithms with the Double Strand Representation. We compare the performances of both representations, giving experimental results on the Connection Machine in Section 6. Finally, we conclude this paper in Section 7.

2. NODE SET REPRESENTATION AND AGRAWAL/SCHUBERT ENCODING

Our work has been based on Agrawal *et al.*'s [18] extension of Schubert's class hierarchy reasoner [31] towards directed acyclic graphs. Agrawal *et al.*'s approach [18] makes it possible to verify that A is a subclass of B by comparing a number pair at A to one or more number pairs stored at B. This approach makes no use of the path from A upwards to B at all.

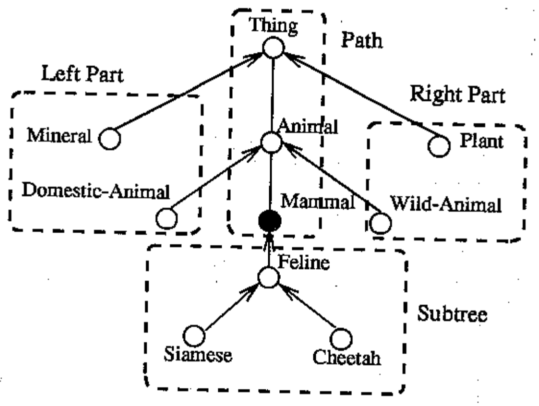


Figure 5. The Four Areas of Spanning Tree

The basic approach of [18] is: (1) Construct an optimal spanning tree of a given DAG such that at every node with multiple parents, we select the link to the parent with the maximum number of predecessors. Predecessors are nodes that are reachable from a node by an "up search." (2) Assign a pair of preorder and maximum number to every node. Preorder numbers are generated by a right-to-left preorder traversal of the spanning tree. The maximum number for every node is the maximum preorder number in the subtree rooted at that node. *Tree pairs* result from this step. (3) All the arcs that are not part of the optimal spanning tree are used to propagate number pairs upward. This is done so that every transitive IS-A relation in the DAG can be verified, but no redundant pairs are generated. *Graph pairs* result from this step.

In Figure 4 we use the notation $[\pi \quad \mu]$ for *tree pairs* and the notation $(\pi \quad \mu)$ for *graph pairs*. In this representation a node A is a subclass of a node B iff the tree pair of A is included in or equal to any one of the pairs of B. For instance, a Cheetah is a Feline because [7 7] is a subinterval of [6 8]. A Cheetah is also a Wild Animal, because the tree pair of Cheetah [7 7] is propagated to Wild Animal as a graph pair (7 7). However, a Cheetah is not a Mineral because [6 8] and [10 10] are disjoint.

In [23] our incremental massively parallel encoding of DAGs, called "node set representation," was introduced. We proved that the node set representation together with the number pairs is sufficient to represent a class hierarchy. We can operate with a set of nodes because all important update and retrieval operations require only three items at every node: (1) the key item, e.g., Mammal, (2) the number pairs, and (3) the area of the spanning tree where the node is located [23]. It is easy to see that the tree pair at each node N can be used to determine four areas of the graph (Figure 5). Every node N, except for the root, defines a path of spanning tree arcs that connect N to the root. This path divides the spanning tree into four (possibly empty) areas: (1) the path itself; (2) the left part of the path; (3) the right part of the path; (4) the subtree which is rooted at

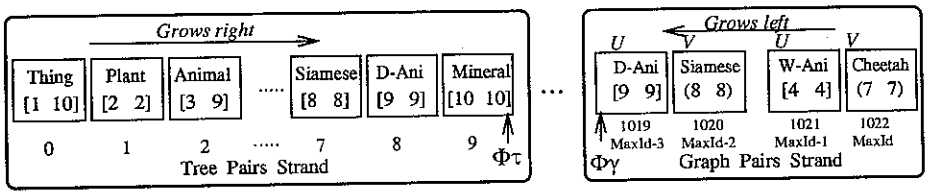


Figure 6. Double Strand Representation for Figure 4

N. For instance, for Mammal in Figure 5, we can easily define area 1: {Thing, Animal, Mammal}, area 2: {Mineral, Domestic-Animal}, area 3: {Plant, Wild-Animal}, and area 4: {Feline, Siamese, Cheetah}. Many important steps of the update operations treat each of these four areas uniformly, with the same operations being applied to all the nodes in one area. Therefore, if we have area information, we do not need the class hierarchy any more. Details can be found in [23].

3. DOUBLE STRAND REPRESENTATION(DSR)

Now we will show how the node set is practically mapped onto the processor space of the CM-5. We are interested in a mapping that will represent tree pairs and graph pairs efficiently, so that it is possible to achieve a high degree of parallelism, memory efficiency, and optimal use of available processors.

We call the result of our new mapping the "Double Strand Representation" (DSR). In this representation the given processors are divided into two areas: *the tree pairs strand* and *the graph pairs strand* (See Figure 6). In the tree pairs strand, every node is represented in a separate processor. The tree pair of a node may be assigned to the tree pairs strand in any order.

In the graph pairs strand, pairs of processors are used to store a sequence of pairs, each consisting of a tree pair and a graph pair. Every processor is assigned an address, called its ID. Let *source of propagation* be a node which propagates its tree pair and let *target of propagation* be a node to which a number pair is propagated from the source of propagation. The tree pair U stored in a processor with an odd ID x is used to represent the target of propagation. The graph pair V in the processor with ID $x+1$ is used to represent the source of propagation. Let Z be a processor pair (U, V) in the graph pairs strand. Let Y be the set of all Z. Every time a pair V is propagated to a node with tree pair U, we will represent (U, V) in the graph pairs strand.

If 1K processors are available, the maximum ID (MaxId) will be 1022 in the DSR, and the first processor pair (U, V) will be stored in the two processors 1021 and 1022. In Figure 6 (representing the hierarchy of Figure 4) the tree pair of *Wild-Animal* [4 4] occurs as U in the graph pairs strand in processor MaxId − 1 (1021) and a propagated pair (7 7) which is the tree pair of *Cheetah* occurs in the even processor MaxId (1022) as a graph pair of *Wild-Animal*. Therefore, we can verify that *Cheetah* is a subclass of *Wild-Animal*. (Details will be shown later.)

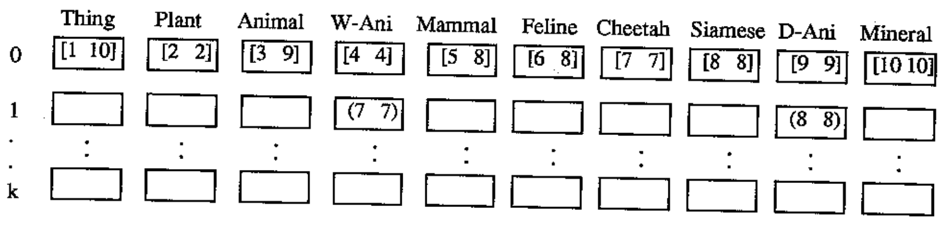


Figure 7. Grid Representation of Figure 4

We will now compare the Double Strand Representation to our Grid Representation used in previous research [27]. The Grid Representation is a distributed representation of a node set (Figure 7). Every node is represented as a column. The first row contains the tree pair of the node, while up to k graph pairs are maintained in the other rows. Nodes may be assigned to columns in the order that the system is informed about their existence. As the Double Strand Representation, this order is irrelevant.

We have been using a grid of 128 columns and 8 rows. On our Connection Machine $128 * 8 = 1024$ is the minimum number processors that may be used. The choice of 128 columns and 8 rows corresponds to a compromise between having a large node set and permitting a reasonably large number of pairs at each node. Note that these 1024 processors are "virtual," meaning that they are simulated on 32 real processors. While we could use larger sets of virtual processors, this would only slow down real-time results. Therefore, we needed to choose the minimum configuration.

While using the grid structure, we encountered some problems. First of all, we have to allocate k processors for graph pairs for every tree pair. This causes a significant number of processors to be left empty. This can lead us to run out of processors when the number of graph pairs in one column exceeds k while in other columns processors are empty.

Secondly, during update of the class hierarchy, we may have an unused processor between two used processors, called a "hole," because of our implementation of Agrawal *et al.*'s subsumption algorithm. Unfortunately, with the current algorithm for the grid structure, we have not found an efficient technique to reclaim such a hole.

Agrawal *et al.*'s subsumption technique, as we have mentioned earlier, is an algorithm which eliminates subsumed pairs during propagation. For example, if a number pair $(\pi_i \ \mu_i)$ is subsumed by another pair $[\pi_j \ \mu_j]$ at the same node (column) due to propagation, i.e., $\pi_j \leq \pi_i$ and $\mu_i \leq \mu_j$, then discard $(\pi_i \ \mu_i)$. It is due to this that some processors are left without pairs and become holes.

All these problems have lead us to abandon the Grid Representation and turn to the Double Strand Representation. One question which arises now is: How do we efficiently organize processors into two strands? Suppose that we organize processors (by SW) in two rows; the first row is used to represent tree pairs and the second row for graph pairs. Since these two strands are growing at different rates of speed, we may encounter a case where all the processors in the tree pairs strand are used up while a lot of unused processors remain in the graph pairs strand. This is clearly not a good representation. It is necessary

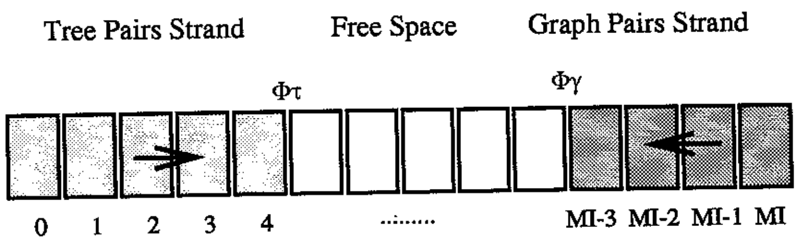


Figure 8. Dynamic Storage Management of Double Strand Representation

to design a processor-efficient technique to avoid this problem.

The main idea of our storage management is borrowed from the dynamic paradigm of languages such as Pascal that maintain a stack and a heap. In our representation, processors of the tree pairs strand are allocated starting at processor 0 and grow towards higher processor IDs. Processors of the graph pairs strand are allocated starting at the processor with the largest ID and grow towards lower processor IDs. There are two pointers, Φ_τ and Φ_γ, to show the borders of both areas. We define $\mathcal{F}$ to be the size of "free space."

$$\mathcal{F} = \Phi_\gamma - \Phi_\tau \tag{1}$$

$\mathcal{F}$ should be bigger than a certain threshold, say 10% of processor space. If this is not the case, we can take some corrective measures, to be described in [24].

4. SUBCLASS VERIFICATION

Now we will describe how to verify subclass relationships. Suppose that we want to verify whether B IS-A A. There are two cases: (1) A is a tree predecessor of B. (2) A is a graph predecessor of B. The first case can be easily verified by a subsumption test: the tree pair of A subsumes the tree pair of B. For the second case, we have to check whether A has a graph pair propagated from B or from a tree predecessor of B.

We now introduce some CM-5 terminology. A parallel variable (*pvar*) is an array (perhaps multi-dimensional) where every component is maintained by its own processor and all values are usually changed in the same way and in parallel [47].

In the algorithm, variables marked with !! are parallel variables, and operations marked with !! or involving parallel variables are parallel operations. The parallel variable pre!! contains for every node a preorder number, and the expression max!! stands for a parallel variable that contains for every node a maximum number. The functions prenum(A), maxnum(A), and tree-pair(A) retrieve the preorder number, maximum number, and the tree pair, respectively, for the given node A.

Additionally, the variable *g-lb* (Graph Strand Lower Bound) represents Φ_γ and *t-ub* (Tree Strand Upper Bound) represents Φ_τ. The parallel function *self-address!!* returns IDs of all active processors and *oddp!!* contains TRUE on a processor if the processor's

ID is an odd number. The algorithm ACTIVATE-PROCESSORS-WITH consists of two parts. The first part describes a set of processors to be activated. The second part, starting with DO, describes what operations should be performed on all active processors.

We now show a function IS-A-VERIFY that performs subclass verification. As we mentioned above, if A is a tree predecessor of B (by IS-A-VERIFY-1) or A is a graph predecessor of B (by IS-A-VERIFY-2), then B IS-A A.

```
; B is-a A iff IS-A-VERIFY returns T.
IS-A-VERIFY (B, A: Node): BOOLEAN
    return(IS-A-VERIFY-1(B, A) OR IS-A-VERIFY-2(B, A))


IS-A-VERIFY-1 (B, A: Node): BOOLEAN
    ; If A is a tree predecessor of B, then the tree pair of A subsumes the tree
    ; pair of B.
    ACTIVATE-PROCESSORS-WITH
        prenum(B) ≥!! prenum(A) AND!!
        maxnum(B)≤!! maxnum(A) AND!!
        self-address!!() ≤!! t-ub
    DO BEGIN
        IF any processor is still active THEN return T
    END
```

Now we will show how to verify that B IS-A A when A is a graph predecessor of B. Remember that a pair of processors (U, V) in the graph pairs strand is used to represent a graph pair. The tree pair in the odd processor (U) is used to represent a node S and the graph pair in the even processor (V) is used to represent a node which propagates its tree pair to S. Therefore, we are looking for a pair of processors (U, V) such that the tree pair of A is contained in processor U and the graph pair of B or its tree predecessor is contained in processor V. In the following functions the expression *mark*!![x] := y means that the pvar *mark*!! on the processor with the ID x is assigned the value y.

```
IS-A-VERIFY-2 (B, A: Node): BOOLEAN
    ; Activate every occurrence of the tree pair of A in the graph pairs strand.
    ; Set the parallel flag mark!! on the right neighbor processors of the active
    ; processors.
    ACTIVATE-PROCESSORS-WITH
        pre!! ≤!! prenum(tree-pair(A)) AND!!
        max!! ≥!! maxnum(tree-pair(A)) AND!!
        self-address!!() ≥!! g-lb AND!!
        oddp!! (self-address!!())
    DO BEGIN
        mark!![self-address!!() +!! 1]:= 1
    END
```

```
; Test whether any marked processor has the tree pair from B or
; from a tree predecessor of B, as a graph pair. If this is the case, return T.
ACTIVATE-PROCESSORS-WITH
    pre!! ≤!! prenum(tree-pair(B)) AND!!
    max!! ≥!! maxnum(tree-pair(B)) AND!!
    mark!![self-address!!()] =!! 1
DO BEGIN
    IF any processor is still active THEN return T
END
```

In our example (Figure 6), Feline is a subclass of Animal because [6 8] is a subinterval of [3 9] (by IS-A-VERIFY-1). IS-A-VERIFY-2 will verify that Siamese is a subclass of Domestic-Animal because the tree pair [8 8] of Siamese will occur as (8 8) together with the tree pair [9 9] in the graph pairs strand. Feline is not a Plant because [6 8] is neither a subinterval of [2 2] nor is there a processor pair ([2 2], (6 8)) in the graph pairs strand. In summary, with the Double Strand Representation, it can be rapidly decided whether a subclass relation exists between two classes.

5. INCREMENTAL UPDATE OF CLASS HIERARCHY WITH DSR

In a directed acyclic relationship graph, there are two "obvious" incremental update operations: (a) inserting a graph component into another graph component, when both of them are initially disconnected components; (b) adding a new link between two nodes of the same graph component. We call (a) graph insertion and (b) link insertion, while *insertion* and *update* may refer to either one of them. We previously presented algorithms for graph insertion in [28] and link insertion in [23].

We need to show how the insertions can be performed with the DSR. The steps that have to be taken for an insertion consist of (1) the global number pair update [23] and (2) the propagation of number pairs.

As shown before, in the Double Strand Representation, a class is represented by storing its tree pair in the tree pairs strand. Graph pairs are stored in pairs of processors in the graph pairs strand. Note that this makes the graph pairs strand a list of pairs of number pairs. In Section 2 we pointed out that all important update and retrieval operations require only three items of information at every node. The first two, the key item of the node and the number pairs at the node, need to be stored explicitly. The third item, the tree area, can be determined easily, as will be argued now.

The global update operations (1) treat every number pair (whether it is a tree pair or a graph pair) in the two strands uniformly, with the same operations being applied to all the processors. The reason for that is as follows. The change of a graph pair has to mirror the change of the tree pair from which it was created by propagation. But how does the algorithm know which transformation to apply to a tree pair? It makes this decision based completely on the tree pair itself! Therefore, the same criteria can be applied to the graph pairs that are identical to a specific tree pair. It should be noted that we are not dealing with every possible transformation resulting from a global number pair update as this is the subject of a separate paper [23].

Now, we will discuss a parallel number pair propagation algorithm for (2) in detail. The propagation of number pairs is performed as follows. Suppose that a graph link is inserted from a node C to a node N. We have to propagate the tree pair of C to every predecessor N_i of N (including N itself). For every processor N_i to which a pair V is propagated we need to generate a new entry for the graph pairs strand. This new entry consists of the tree pair of N_i and V: (Tree-Pair(N_1), V), (Tree-Pair(N_2), V), ..., (Tree-Pair(N_k), V), where k is the number of predecessors. These newly generated pairs have to be assigned to 2 * k currently unused contiguous processors to the left of Φ_γ (Figure 9(c, d)). If several pairs V_i need to be propagated, this process needs to be done serially.

As an example (Figure 9(a, b)), due to inserting the arc from H to E, the tree pair $V = [5 \;\; 5]$ of H should be propagated to every predecessor of E, and E itself (namely, E, C, B, A). As A has the pair $[1 \;\; 8]$, we do not need to propagate $[5 \;\; 5]$ to A because $[1 \;\; 8]$ subsumes $[5 \;\; 5]$. In our terminology, only E, C, and B are targets of propagation. For propagating $[5 \;\; 5]$, we need to find the appropriate IDs of processors to assign $[5 \;\; 5]$ to (in parallel).

For this, we develop a parallel function to find proper processor IDs for each propagated pair (Figure 9(c)). First, we activate processors in the tree and graph pairs strands that correspond to predecessors N_i to which we want to propagate a specific graph pair V. Second, there is a parallel operation, *enumerate!!*, on the CM-5 that will assign numbers 0, 1, 2 ... to active processors. Third, we define a parallel function $\mathcal{T}$ to compute the processor ID where the processor with the number x (assigned by the enumerate function) should deposit its number pair.

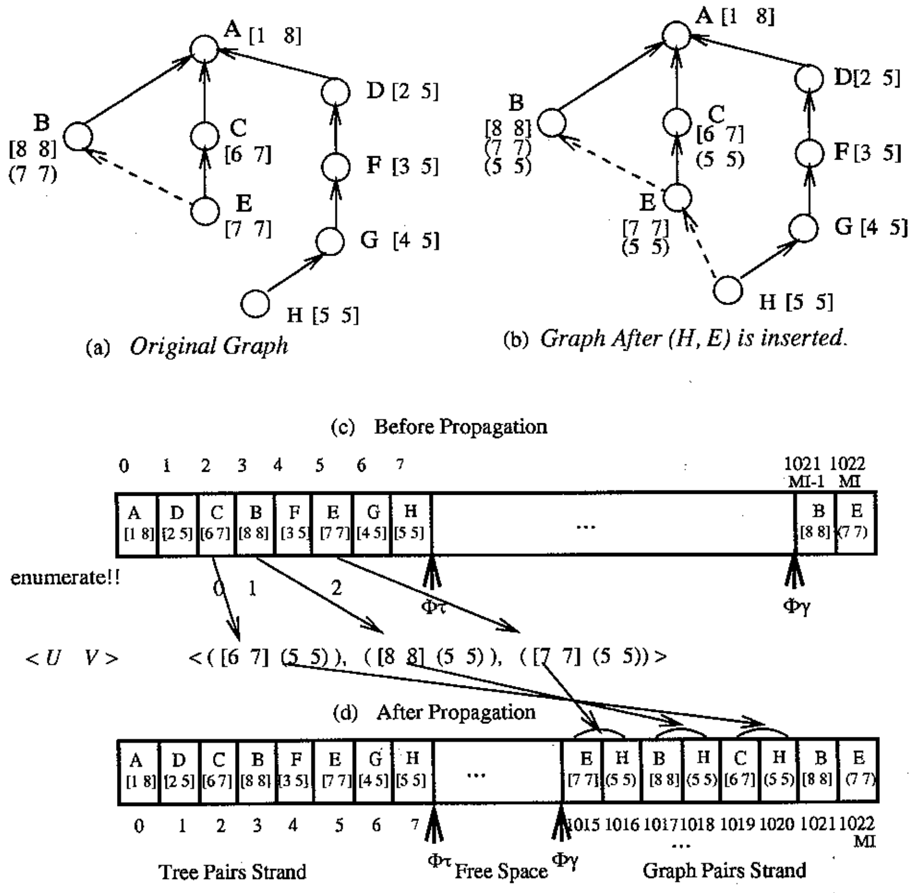
$$\mathcal{T}(x) = \Phi_\gamma - 2(x+1) \qquad (2)$$

where $x \geq 0$ *and* $x \leq \Phi_\tau$. $\mathcal{T}$ computes the odd position, and we generate the pair $(\mathcal{T}(x), \mathcal{T}(x)+1)$ for (Tree-Pair(N_i), V). With these three steps, mapping each predecessor to its corresponding processor ID in the graph pairs strand can be completed.

For instance, in Figure 9(a, b), when inserting the arc from H to E, we first activate every tree predecessor of E (C and E itself), but not A. Similarly we activate every graph predecessor of E (just B). Then, we call *enumerate!!* and assign numbers, 0, 1, and 2, respectively. The tree pairs of C and H are assigned to 1019 and 1020 which are $\mathrm{T}(0) = \Phi_\gamma - 2 * 1$ and $\mathrm{T}(0) + 1 = \Phi_\gamma - 2 * 1 + 1$. Similarly the tree pairs of E and H are stored at 1017 and 1018, and the tree pairs of B and H, at 1015 and 1016.

We will now present our parallel propagation algorithm. During the propagation, we may have to consider two problem cases caused by redundant pairs. Let a pair $(\pi_i \;\; \mu_i)$ be the newly propagated pair and let another pair $(\pi_j \;\; \mu_j)$ be a pair at a target node of propagation. In the first problem case, a pair $(\pi_j \;\; \mu_j)$ at the target is enclosed by the propagated pair $(\pi_i \;\; \mu_i)$, i.e., $\pi_i < \pi_j$ and $\mu_j < \mu_i$, then the pair $(\pi_j \;\; \mu_j)$ must be replaced by $(\pi_i \;\; \mu_i)$. In the second problem case, the pair $(\pi_j \;\; \mu_j)$ encloses the newly propagated pair $(\pi_i \;\; \mu_i)$, i.e., $\pi_j \leq \pi_i$ and $\mu_i \leq \mu_j$, and we do not need to propagate the pair $(\pi_i \;\; \mu_i)$ to this target.

In the propagation, we replace the redundant pairs just described. The results of this algorithm correspond to the results of Agrawal *et al.*'s algorithm. The boolean function *evenp!!* returns TRUE on a processor if the processor's ID is an even number. In the



A [1 8]
B [8 8] (7 7)
C [6 7]
D [2 5]
E [7 7]
F [3 5]
G [4 5]
H [5 5]
(a) Original Graph
A [1 8]
B [8 8] (7 7) (5 5)
C [6 7] (5 5)
D[2 5]
E [7 7] (5 5)
F [3 5]
G [4 5]
H [5 5]
(b) Graph After (H, E) is inserted.
(c) Before Propagation
0 1 2 3 4 5 6 7 1021 MI-1 1022 MI
A [1 8] D [2 5] C [6 7] B [8 8] F [3 5] E [7 7] G [4 5] H [5 5] ... B [8 8] E (7 7)
enumerate!!
0 1 2
Φτ
Φγ
< U V > <([6 7] (5 5)), ([8 8] (5 5)), ([7 7] (5 5))>
(d) After Propagation
A [1 8] D [2 5] C [6 7] B [8 8] F [3 5] E [7 7] G [4 5] H [5 5] ... E [7 7] H (5 5) B [8 8] H (5 5) C [6 7] H (5 5) B [8 8] E (7 7)
0 1 2 3 4 5 6 7 1015 1016 1017 1018 1019 1020 1021 1022 MI
Φτ Free Space Φγ
Tree Pairs Strand
Graph Pairs Strand

Figure 9. Propagation in Double Strand Representation

algorithm, the expression *redundant!!* stands for a boolean parallel variable that represents any redundant pairs in the predecessors. As before, in the following functions the expression *mark*!![x] := y means that the pvar *mark*!! on the processor with ID x is assigned the value y.

Finding tree predecessors will be different from finding graph predecessors because the tree pairs and the graph pairs are stored in a different form in the tree pairs strand and in the graph pairs strand, respectively. The function *target-address!!* returns addresses of the target processors of the propagated pairs for tree predecessors and graph predecessors uniformly.

```
Mark-Predecessor($N$-Pair, $M$-Pair : Pair)
    ; Activate every graph predecessor of a node $N$ which is not predecessor
    ; of the node $M$, where $N$ is a new parent node of $C$ and $M$ is the tree
    ; parent of the child node $C$. The nodes $N$ and $M$ have the tree pairs $N$-Pair
    ; and $M$-Pair, respectively. Then set the flag mark!! on the graph predecessors.
    ACTIVATE-PROCESSORS-WITH
        pre!! $\leq$!! prenum($N$-Pair) AND!!
        max!! $\geq$!! maxnum($N$-Pair) AND!!
        NOT!!(pre!! $\leq$!! prenum($M$-Pair) AND!!
              max!! $\geq$!! maxnum($M$-Pair))
    DO BEGIN
            mark!![target-address!!()]:= 1      ; set predecessors
    END
```

Note that, due to propagation, redundant pairs could appear in the marked predecessors. As mentioned before, there are two problem cases caused by redundant pairs. In the first case, the problem could occur only in graph pairs because in this step we are dealing with replacing enclosed pairs with enclosing pairs while in the second case it could occur either in tree pairs or in graph pairs.

In the following algorithm, we will present the solution for these problems. For the first case, in the IF!! clause, we examine whether any graph pair in the predecessors is subsumed by the newly propagated pairs but only check the even processors in the graph pairs strand using *evenp!!* because every graph pair is stored at the even processors in the graph pairs strand. In contrast, for the second case, we examine whether any graph pair and any tree pair in the predecessors is subsuming the newly propagated pair because if that is true, we do not have to propagate the new pair any further. In both cases, the boolean pvar *redundant!!* is set and additionally, in the first case, the enclosed pair is replaced with the number pair to be propagated.

```
Redundant-Pair-Elimination($PM$-Pair-$V$ : Pair)
    ; Replace the pair at the target processor with the newly propagated
    ; pair $PM$-pair-$V$ in the first case, set the flag redundant!! on
    ; the target processor in both cases.
    ACTIVATE-PROCESSORS-WITH
        mark!![target-address!!()] =!! 1
```

```
DO BEGIN
    ; check whether it is the first case of redundant pairs
    IF!! (pre!! >!! prenum($PM$-Pair-$V$) AND!!
        max!! ≤!! maxnum($PM$-Pair-$V$) AND!!
        evenp!!(self-address!!()) AND!!
        self-address!!() ≥!! g-lb) THEN
          pre!![self-address!!()]:= prenum($PM$-Pair-$V$)    ; replace the prenum
          max!![self-address!!()]:= maxnum($PM$-Pair-$V$) ; replace the maxnum
          redundant!![target-address!!()]:= 1                 ; set the flag
    ; check whether it is the second case of redundant pairs
    ELSE IF!! (pre!! ≤!! prenum($PM$-Pair-$V$) AND!!
              max!! ≥!! maxnum($PM$-Pair-$V$)) THEN
              redundant!![target-address!!()]:= 1             ; set the flag
    END IF!!
END
```

At this stage, the boolean pvar *mark!!* is set for every predecessor of the given node and the boolean pvar *redundant!!* is set for the processors at which redundant pairs could appear due to the number pair propagation. The next step is to enumerate processors which are predecessors without redundant pairs.

```
Order-Strand( )
    ; Enumerate the marked predecessors. No parameter is needed,
    ; because the global variable mark!! is already set on the predecessors.
    ACTIVATE-PROCESSORS-WITH
        mark!![self-address!!()] =!! 1 AND!!
        NOT!!(redundant!![self-address!!()] =!! 1) AND!!
        self-address!!() ≤!! t-ub
    DO BEGIN
        Pos!![self-address!!()]:= enumerate!!(self-address!!())
    END
```

Now every preliminary step for mapping each predecessor to its corresponding processor ID in the graph pairs strand is finished. Finally, using the functions $\mathcal{T}(x)$ and $\mathcal{T}(x+1)$, the propagation is performed in the following two steps. First, the copies of the tree pairs of the target nodes are copied to their destinations on odd processors. Then the unique pair to be propagated, V, is propagated to the corresponding even processors. Pos!! stands for a parallel variable that contains the numbers 0, 1, 2 ... assigned by *enumerate*!! in Order-Strand.

```
Propagate-Pair($PM$-Pair-$V$ : Pair)
    ; Propagate $U_i$ pairs to the targets of propagation. The processor IDs
    ; for the targets are calculated by $\mathcal{T}(x)$.
    ; Propagate the same pair PM-Pair-V to the targets of propagation.
; The processor IDs for the targets are calculated by $\mathcal{T}(x) + 1$.
ACTIVATE-PROCESSORS-WITH
    pos!! ≥!! 0
DO BEGIN
    pre!![g-lb – (pos!! + 1) * 2]:= pre!!                            ; $\mathcal{T}(x)$
    max!![g-lb – (pos!! + 1) * 2]:= max!!
    pre!![g-lb – (pos!! + 1) * 2 + 1]:= prenum($PM$-Pair-$V$)   ; $\mathcal{T}(x+1)$
    max!![g-lb – (pos!! + 1) * 2 + 1]:= maxnum($PM$-Pair-$V$)
END
```

Now comes the top level propagation algorithm which combines the above algorithms. It propagates every number pair of a node C to the targets of propagation which were defined by the predecessors of N. The node M is the tree parent of the child node C.

```
Propagation($N$, $M$, $C$ : Node)
; $N$-Pair and $M$-Pair are the tree pairs of a node $N$ and a node $M$, respectively.
; $PM$-Pair-$V$ is a pair at $C$ to be propagated.
    Initialize-Pvars()
    Mark-Predecessor ($N$-Pair, $M$-Pair)          ; mark tree predecessors of $N$
    FOR Each number pair $PM$-Pair-$V$ at $C$ DO
        Redundant-Pair-Elimination ($PM$-Pair-$V$); eliminate any redundant pairs
        Order-Strand()                             ; enumerate the marked processors
        Propagate-Pair($PM$-Pair-$V$)              ; propagate $U$ and $V$ pairs
        Unset-Pvars                                ; do some house keeping
    END FOR
END
```

6. PERFORMANCE IN THE GR AND THE DSR

6.1. Space and run-time complexities

We now analyze how many processors are required for implementing the GR and DSR. Agrawal *et al.* proved that $\left\lfloor\frac{(N+1)^2}{4}\right\rfloor$ number pairs are required to represent the worst case of a bipartite graph G with N nodes [18]. Let N be the number of nodes (the number of tree pairs) and P be the number of graph pairs in a DAG; then in the worst case

$$P = \left\lfloor\frac{(N+1)^2}{4}\right\rfloor - N. \tag{3}$$

Let k be a predefined maximum number of graph pairs for the GR. For the GR the total space requirement is O(k * N). In the worst case k can be O(N) and the space complexity for the GR is O(N^2). Note that we are currently fixing k to be 8 as a good compromise between processor use and efficiency of the algorithm. In the DSR the space complexity is

$$O(N + 2 * G) = O\left(N + 2 * \left(\left\lfloor\frac{(N+1)^2}{4}\right\rfloor - N\right)\right) = O(N^2) \tag{4}$$

i.e., the same space complexity in the worst case. However, the DSR does not have the problem of unused processors.

Now we analyze the run-time complexity for the GR and DSR. Our parallel algorithms for subclass verification and propagation were presented in Sections 4 and 5. In order to analyze the time complexity of these algorithms, we need to define the following parameters:

$T_t(N,C)$	: Parallel time to determine whether the tree pair of N encloses the tree pair of C.
$T_g(N,C)$	: Parallel time to determine whether the predecessors of N have a graph pair from C.
$T_d(N)$	: Parallel time to determine every predecessor of a node N.
$T_m(N)$	: Parallel time to determine every tree predecessor of a node N.
$T_n(N)$	: Parallel time to determine every graph predecessor of a node N.
$T_r(N,C)$	: Parallel time to replace pairs at the predecessors of N with pairs from C or mark the processors where redundant pairs may have appeared.
$T_p(N)$	: Parallel time to propagate a number pair to the marked predecessors.
$P(C)$	: Average number of number pairs in C.

In the subclass verification algorithm, there are two possible cases with IS-A-VERIFY (N, C). If N is a tree predecessor of C, the run-time for this operation is T_t. If N is a graph predecessor of C, the run-time is T_g. Assuming a unit communication time [21], a commonly made assumption, T_t and T_g are O(1). Therefore, overall run-time complexity for subclass verification is constant.

One question which arises now is whether there are any differences in run-time complexity between the GR and DSR. The difference between the two representations is not in the verification processing, but in the graph pair distribution. The run-time complexity of the subclass verification for the DSR is the same as that in the GR. Consequently, we have a constant subclass verification algorithm in both cases.

In the propagation algorithm for the Double Strand Representation, there are two processing steps required as mentioned in Section 5: one for tree predecessors and another for graph predecessors. There are three phases: (1) identify the tree predecessors (T_m) and the graph predecessors (T_n), (2) replace any redundant pairs (T_r), and (3) enumerate the predecessors and propagate number pairs (T_p). We can formulate the average run-time for the propagation algorithm as follows:

$$T = T_m(N) + T_n(N) + P(C) * (T_r(N,C) + T_p(N)). \tag{5}$$

As before T_m, T_n, T_r, and T_p can be regarded as constants because within constant processor set size, these do not grow with increasing knowledge base size. Then, we can simplify the run-time complexity to $O(P)$. Similarly, the run-time of the propagation algorithm in the Grid Representation is

$$T' = T_d(N) + P(C) * (T_r(N,C) + T_p(N)). \tag{6}$$

By the same reasoning, it can be simplified to $O(P)$.

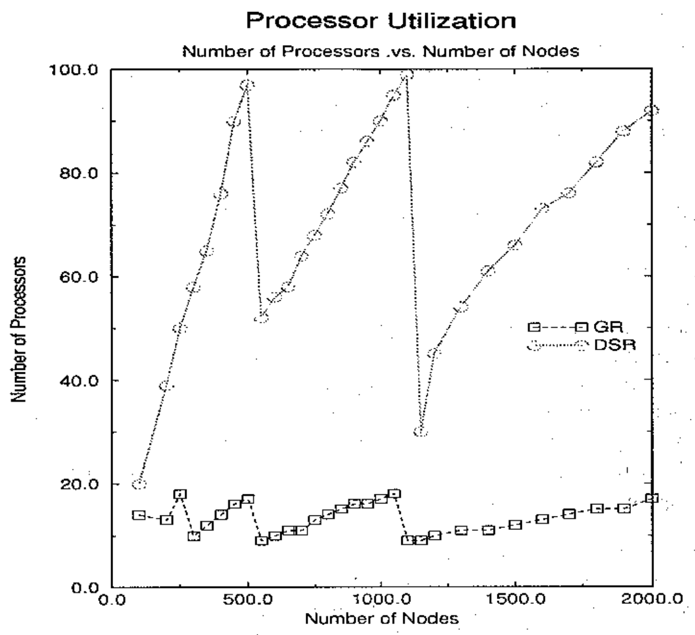


Figure 10. Processors Utilization

6.2. Experimental results

In this section we present experimental results of the parallel subclass verification and number pair propagation algorithms for the GR and DSR. The experiments were done on a Connection Machine CM-5, which makes use of groups of virtual processors executing serially on real processors. There are 32 real processors on the NPAC CM-5. Every processor emulates the activities of at least 32 virtual processors. The CM-5 [47] is programmed in *LISP, a dialect of Common LISP, by mapping parallel variables (*pvar*) onto distinct processors.

6.2.1. Experiments with random data

For our experiments, we are using a random generator for DAGs. The following parameters are supplied as input to this generator: the number of nodes (N), the branching factor of each node (B), and the depth (D). Preliminary experiments with several values of B and D showed that the computation time seems to be unaffected by B and D. This should be expected as we have eliminated the explicit representation of the IS-A links from the outset. Therefore, we limited $D = 9, \ldots, 12$ and set $B = 5$. The effect of graph size on run-time was determined for both representations. The number of nodes was varied from 25 to 2000. Graphs have approximately 20% of graph arcs, e.g., a graph with 2000 nodes has about 400 graph arcs.

For the GR, assume that k is 8. Then 1K processors are required for 1 to 128 nodes, 2K for 1 to 256, ..., 16K for up to 2K nodes. Processor utilization is very low, only up

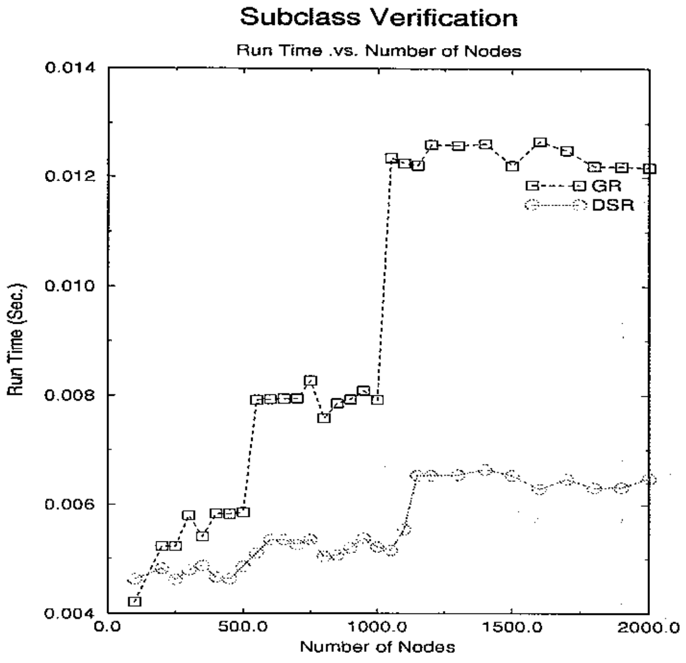


Figure 11. Run Time for Subclass Verification

to 18%. We also determined that the maximum number of actually used rows in the GR was 5.

Our experiments with random graphs showed that the number of graph pairs increased at approximately the same rate as the number of nodes. For instance, 48 graph pairs are generated in a 100-node graph, ..., 900 graph pairs in a 2000-node graph. In our experiments, typically, the number of graph pairs is limited to less than half the number of tree pairs. According to that, for the DSR, approximately 1K processors are required for graphs with up to 0.5K nodes, 2K processors for graphs with up to 1K nodes, and 4K for up to 2K nodes with very high processor utilization (up to 99%). In Figures 11–13, the run-times jump at two critical points, namely at the node numbers 500 and 1000. These jumps are due to doubling of the number of allocated virtual processors, i.e., from 1K to 2K and 2K to 4K. As the number of real processors stays the same, every real processor has to double the number of operations it performs. The DSR shows better performance than the GR in terms of both the amount and utilization of processors with increasing knowledge base size (Figure 10).

For the comparative run-time evaluation of DSR and GR with various sizes of the knowledge base, we implemented the graph insertion, link insertion, and subclass verification algorithms. The test data for link insertion makes a number of simplifying assumptions which are based on problems described in [23]. Figures 11–13 show the results of experiments with various sizes of the knowledge base. The figures show the run times in seconds

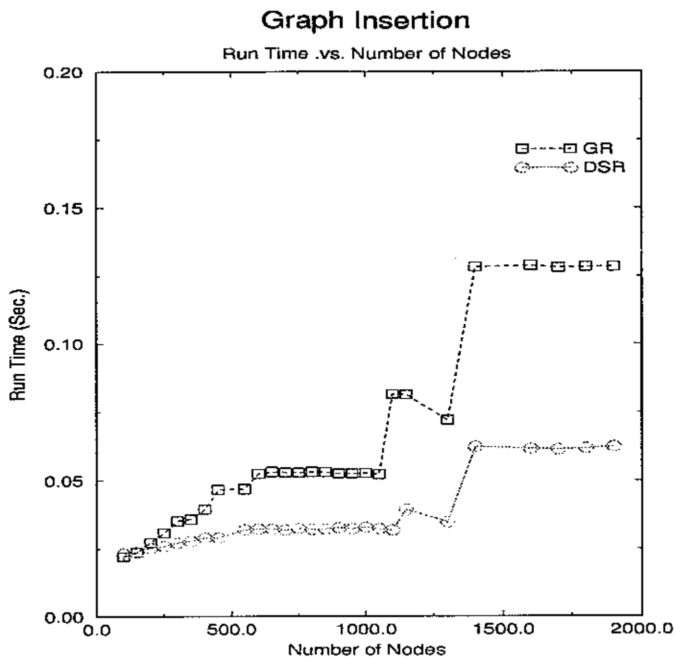


Figure 12. Run Time for Graph Insertion

over the total numbers of nodes in a graph for the subclass verification (Figure 11), graph insertion (Figure 12), and link insertion (Figure 13) in both representations. As can be seen, the computation times of subclass verification and dynamic update algorithms in the DSR grow much more slowly than in the GR. It is interesting that the execution time for increasing numbers of nodes in a graph are almost the same for constant processor set size.

In summary, when we are increasing the size of the graphs, the cost of implementing the subclass verification and number pair propagation algorithms in the DSR in terms of processor utilization is much lower than that in the GR. For run-times, the DSR becomes better for over 500 nodes.

6.2.2. Experiments with INTERMED

Now we want to show experimental results using an existing large medical vocabulary. We have tested our verification and update operations in the Grid Representation and the Double Strand Representation using the INTERMED (INTERnet version of the Medical Entities Dictionary) system of CPMC (Columbia Presbyterian Medical Center) [35–38].

The INTERMED system currently has about 2,500 medical terms. These terms are related by general relations such as IS-A, but also by domain specific relations such as PHARMACEUTIC-COMPONENT-OF.

The following results show the necessity of the Double Strand Representation as well as

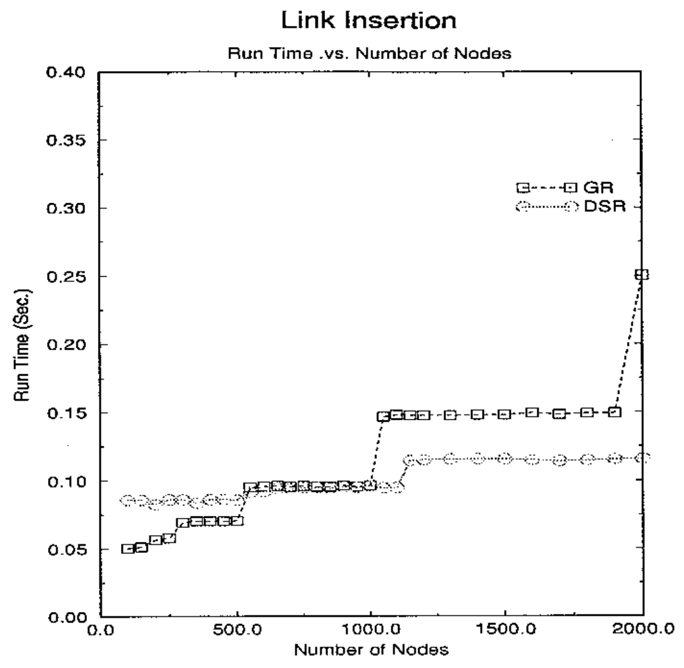


Figure 13. Run Time for Link Insertion without Propagation

the efficiency of the Double Strand Representation. First, we extracted information from the INTERMED and then translated it into a format fit for our system. Each term in the INTERMED is treated as a node and we maintain only the IS-A relation. We used 8K processors with the Double Strand Representation while 32k would be necessary with the Grid Representation. The average run-times for a subclass verification, a graph insertion, and a link insertion in the Double Strand Representation are 0.0004 sec, 0.023 sec, 0.139 sec, respectively. Figure 14 shows the run-time results for the link insertion algorithm over the number of pairs propagated. As the number of number pairs to be propagated increases, the run-time for the link insertion algorithm increases. This confirms our claim in Section 6.1 that the run-time for the number pair propagation algorithm is proportional to the number of number pairs to be propagated.

In the Double Strand Representation, 2494 tree pairs and 1442 graph pairs are generated. However, some nodes have up to 456 graph pairs. This means 449 graph pairs cannot be represented in the Grid Representation because the Grid Representation restricts the number of rows to 8.

As it would be quite unacceptable to extend the Grid Representation to 512 rows, this result shows that with real data the Grid Representation is not practical at all, while the Double Strand Representation performs well.

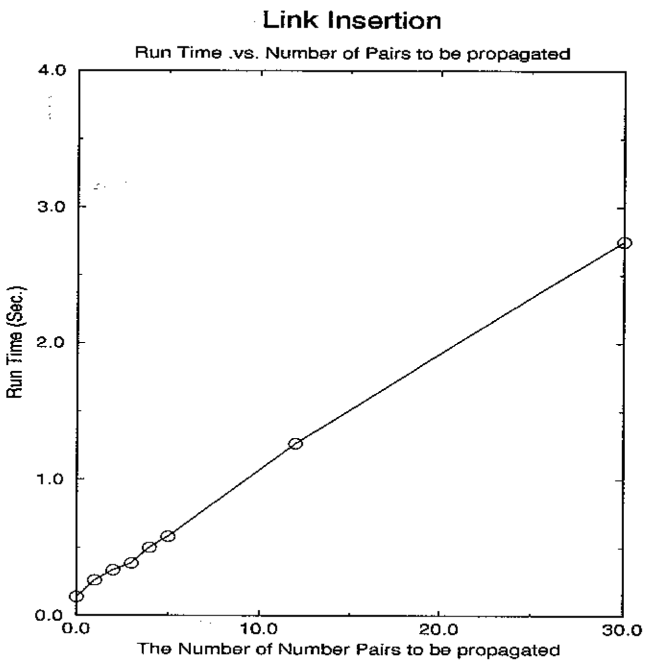


Figure 14. Run Time for Link Insertion with Number Pair Propagation

7. CONCLUSION

In this paper, we have introduced a new massively parallel representation for class hierarchies that are constrained to be representable by directed acyclic graphs. We call it the Double Strand Representation. This representation maps the node set representation onto a linear space of processors. Processor space is divided into two strands, the tree pairs strand and the graph pairs strand.

We showed how the DSR can be used to quickly verify the existence of an IS-A relationship that is possibly the result of transitive closure, and therefore not explicitly represented. We also showed how to implement the propagation of a number pair in parallel. Propagation is fundamentally necessary for every update operation based on Agrawal *et al.*'s encoding.

Experimental results show that the processing times of subclass verification and number pair propagation are mainly affected by the number of allocated processors. The DSR achieves high performance compared with the GR in terms of processor utilization and run-time.

Acknowledgments
We thank James Cimino for making the MED available to us and for explaining it to us. Scott C. Pearson has written the random graph generator that was used for providing us

with random test data. We also thank Mike Halper who has improved the presentation of this book chapter.

REFERENCES

1. R. J. Brachman, On the epistemological status of semantic networks, *Associative Networks* (N. Findler, ed.), pp. 3–50, New York, NY: Academic Press, 1979.
2. R. J. Brachman and J. Schmolze, An overview of the KL-ONE knowledge representation system, *Cognitive Science*, vol. 9, no. 2, pp. 171–216, 1985.
3. W. A. Woods, What's in a link. foundations for semantic networks, in *Representation and Understanding* (D. G. Bobrow and A. M. Collins, eds.), pp. 35–82, New York, NY: Academic Press, 1975.
4. W. A. Woods, Knowledge representation: What's important about it?, in *The Knowledge Frontier* (N. Cercone and G. McCalla, eds.), pp. 44–79, New York, NY: Springer Verlag, 1987.
5. A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick, CLASSIC: A structural data model for objects, in *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, appeared as SIGMOD*, vol. 18, no. 2, pp. 58–67, 1989.
6. R. J. Brachman, R. E. Fikes, and H. J. Levesque, KRYPTON: A functional approach to knowledge representation, *IEEE Computer*, vol. 16, no. 10, pp. 67–73, 1983.
7. R. M. MacGregor, A deductive pattern matcher, *Seventh National Conference on Artificial Intelligence*, pp. 403–408, San Mateo, CA: Morgan Kaufmann, 1988.
8. B. Nebel and K. von Luck, Issues of integration and balancing in hybrid knowledge, *GWAI-87* (K. Morik, ed.), pp. 114–123, Berlin, Germany: Springer Verlag, 1987.
9. S. Bayer and M. Vilain, The relation-based knowledge representation of King Kong, *SIGART Bulletin*, vol. 2, no. 3, pp. 15–21, 1991.
10. W. Kim, N. Ballou, H.-T. Chou, and J. F. Garza, Features of the orion object-oriented database system, in *Object-Oriented Concepts, Databases, and Applications* (W. Kim and F. H. Lochovsky, eds.), Reading, MA: Addison Wesley, 1989.
11. C. Lecluse, P. Richard, and F. Velez, O_2 an object-oriented data model, *Readings in Object-Oriented Database Systems* (S. B. Zdonik and D. Maier, eds.), pp. 227–236, San Mateo, CA: Morgan Kaufmann, 1990.
12. V. Soloviev, An overview of three commercial object-oriented database management systems: ONTOS, ObjectStore and O_2, *SIGMOD Record*, vol. 21, no. 1, pp. 93–104, 1992.
13. O. J. Dahl and K. Nygaard, SIMULA–an ALGOL–based simulation language, *Communications of the ACM*, vol. 9, no. 9, pp. 671–678, 1966.
14. A. Goldberg and D. Robson, *Smalltalk-80: The language and its implementation.* Reading, MA: Addison Wesley, 1983.
15. S. C. Dewhurst and K. T. Stark, *Programming in C++.* Englewood Cliffs, NJ: Prentice Hall, 1989.
16. L. Cardelli and P. Wegner, On understanding types data abstraction, and polymorphism, *ACM Computing Surveys*, vol. 17, pp. 471–522, 1985.
17. S. E. Keene, *Object-Oriented Programming in COMMON LISP.* Reading, MA:

Addison-Wesley, 1989.

18. R. Agrawal, A. Borgida, and H. V. Jagadish, Efficient management of transitive relationships in large data and knowledge bases, in *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, (Portland, OR), pp. 253–262, 1989.
19. H. V. Jagadish, A compressed transitive closure technique for efficient fixed-point query processing, (San Francisco, CA), pp. 209–223, 1988.
20. K. Guh and C. Yu, Efficient management of materialized generalized transitive closure in centralized and parallel environments, *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 4, pp. 371–381, 1992.
21. W. Wang, S. Iyengar, and L. M. Patnaik, Memory-based reasoning approach for pattern recognition of binary images, *Pattern Recognition*, vol. 22, no 5, pp. 505–518, 1989.
22. J. Han and W. Lu, Asynchronous chain recursions, *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, pp. 185–195, 1989.
23. E. Y. Lee and J. Geller, Representing transitive relationships with parallel node sets, in *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems* (B. Bhargava, ed.), pp. 140–145, Los Alamitos, CA: IEEE Computer Society Press, 1993.
24. E. Y. Lee, Massively Parallel Reasoning in Transitive Relationship Hierarchies *PhD dissertation*, NJIT, 1996.
25. J. Geller, Upward-inductive inheritance and constant time downward inheritance in massively parallel knowledge representation, in *Proceedings of the Workshop on Parallel Processing for AI at IJCAI 1991*, (Sydney, Australia), pp. 63–68, 1991.
26. J. Geller and C. Y. Du, Parallel implementation of a class reasoner, *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 3, pp. 109–127, 1991.
27. J. Geller, Innovative applications of massive parallelism, *AAAI 1993 Spring Symposium Series Reports, AI Magazine*, vol. 14, no. 3, p. 36, 1993.
28. J. Geller, Massively parallel knowledge representation, in *AAAI Spring Symposium Series Working Notes: Innovative Applications of Massive Parallelism*, pp. 90–97, 1993.
29. L. Kanal, V. Kumar, H. Kitano, and C. Suttner, Inheritance operations in massively parallel knowledge representation, *Parallel Processing for Artificial Intelligence*, pp. 95–113, New York: North Holland Publishing, 1994.
30. H. Kitano and J. Hendler, Advanced update operations in massively parallel knowledge representation, in *Massively Parallel Artificial Intelligence*, pp. 74–101, AAAI/MIT Press, 1994.
31. L. K. Schubert, M. A. Papalaskaris, and J. Taugher, Accelerating deductive inference: special methods for taxonomies colors and times, in *The Knowledge Frontier* (N. Cercone and G. McCalla, eds.), pp. 187–220, New York, NY: Springer Verlag, 1987.
32. M. Evett, L. Spector, and J. Hendler, Knowledge representation on the connection machine, in *Supercomputing '89*, (Reno, Nevada), pp. 283–293, 1989.
33. M. P. Evett, J. A. Hendler, and W. A. Andersen, Massively parallel support for computationally effective recognition queries, in *Proceedings of the Eleventh National*

Conference on Artificial Intelligence, pp. 297–302, Cambridge, MA: MIT Press, 1993.

34. K. Stoffel and J. Hendler, PARKA on MIND-supercomputers, in *IJCAI-95 Workshop Program Working Notes*, (Montreal, Quebec), pp. 132-142, 1995.
35. J. J. Cimino, G. Hripcsak, S. B. Johnson, and P. D. Clayton, "Designing an introspective, multipurpose, controlled medical vocabulary," in *Proceedings of the Thirteenth Annual Symposium on Computer Applications in Medical Care*, pp. 513–518, Los Alamitos, CA: IEEE Computer Society Press, 1989.
36. J. J. Cimino, P. L. Elkin, and G. O. Barnett, "As we may think: The concept space and medical hypertext," *Computers and Biomedical Research*, vol. 25, pp. 238–263, 1992.
37. J. J. Cimino, A. A. Aguirre, S. B. Johnson, and P. Peng, "Generic queries for meeting clinical information needs," *Bulletin of the Medical Library Association*, vol. 81, no. 2, pp. 195–206, 1993.
38. J. J. Cimino, P. D. Clayton, G. Hripcsak, and S. B. Johnson, "Knowledge-based approaches to the maintenance of a large controlled medical terminology," *Journal of the American Medical Informatics Association*, vol. 1, no. 1, pp. 35–50, 1994.
39. M. P. Evett, W. A. Andersen, and J. A. Hendler, Massively parallel support for efficient knowledge representation, in *Proc. of the 13th Int. Joint Conference on Artificial Intelligence*, pp. 1325–1330, San Mateo, CA: Morgan Kaufmann, 1993.
40. L. Shastri, Default reasoning in semantic networks: a formalization of recognition and inheritance, *Artificial Intelligence*, vol. 39, no. 3, pp. 283–356, 1989.
41. L. Shastri, Semantic Networks: an Evidential Formalization and its Connectionist Realization, *Morgan Kaufmann Publishers*, (San Mateo, CA),1988.
42. L. Shastri and V. Ajjanagadde, An optimally efficient limited inference system, in *Proceedings of IJCAI-90*, (Boston, MA), pp. 563–570, 1990.
43. L. Shastri, A computational model of tractable reasoning – taking inspiration from cognition, in *Proc. of the 13th Int. Joint Conference on Artificial Intelligence*, pp. 202–207, San Mateo, CA: Morgan Kaufmann, 1993.
44. R. Sun, An efficient feature-based connectionist inheritance scheme, *IEEE Transactions on SMC*, vol. 23, no. 2, 1993.
45. R. Sun, Integrating Neural and Symbolic Processes, *Connection Science*, 1994.
46. R. Sun, Robust Reasoning: Integrating Rule-Based and Similarity-Based Reasoning, *Artificial Intelligence*, no. 1, 1995.
47. Thinking Machines Corporation, **LISP Reference Manual Version 5.0 edition.* Cambridge, MA: Thinking Machines Corporation, 1988.

Eunice (Yugyung) Lee

Eunice (Yugyung) Lee received a BS degree in Computer Science from the University of Washington at Seattle, in 1990. She is currently a Ph.D. candidate, expecting her Ph.D. in the Summer of 1996, at the New Jersey Institute of Technology. Her research has been published in workshops on parallel AI and on parallel and distributed systems. Her current research interests include massively parallel knowledge representation and reasoning, object-oriented modeling, and high performance distributed databases.

James Geller

James Geller received an Electrical Engineering Diploma from the Technical University Vienna, Austria, in 1979. His M.S. degree (1984) and his Ph.D. degree (1988) in Computer Science were received from the State University of New York at Buffalo. He spent the year before his doctoral defense at the Information Sciences Institute (ISI) of USC in Los Angeles, working with their Intelligent Interfaces group. James Geller received tenure in 1993 and is currently associate professor in the Computer and Information Science Department of the New Jersey Institute of Technology, where he is also Director of the AI & OODB Laboratory. Dr. Geller has published numerous journal and conference papers in a number of areas, including knowledge representation, parallel artificial intelligence, and object-oriented databases. His current research interests concentrate on object-oriented modeling of medical vocabularies, and on massively parallel knowledge representation and reasoning. James Geller was elected SIGART Treasurer in 1995. His Data Structures and Algorithms class is broadcast on New Jersey cable TV.

Home Page: http://hertz.njit.edu/~geller