Parka on MIMD-supercomputers *

Kilian Stoffel , James Hendler, Joel Saltz and Bill Andersen
Computer Science Dept./UM Institute for Advanced Computer Studies
University of Maryland, College Park

We have ported the SIMD Parka knowledge representation system to generic MIMD machines. The system has been recoded in C and supported using runtime optimization packages developed in the High Performance Systems Software Laboratory at the University of Maryland. New "scanning" algorithms have been developed for inheritance and recognition inferences. These algorithms have been tested with both random networks and on a recoding of the ontology of the CYC knowledge base as well as on large planning case-bases. Tests show that the new version is significantly faster than the SIMD system, and that it promises to scale well to knowledge bases orders of magnitude larger than CYC.

1. Introduction

In the past, research in the field of (very) large knowledge bases has primarily concentrated on defining KR languages and analyzing their complexity. This research has resulted in a wide range of well-studied languages, with much known about their theoretical performance. Less work, however, has been spent examining details of efficiently implementing such languages, or on the scaling properties of these KR systems using large, real-world, applications. Up until recently, this negligence has largely been justified with the excuse that very large KBs don't yet exist, and that the community could postpone dealing with these issues. However, this excuse is no longer valid. Current research has resulted in the development of extremely large KBs including not only the "common sense" KB of Lenat's CYC [15], but also (much larger) KBs including machine-readable dictionaries [14], large ontologies [9], and very large case-bases for AI planning systems [13]. Despite this, most of the current KR systems are not able to accommodate these KBs, which may contain millions of assertions about many thousands of objects.

The ParalleL Understanding Systems (PLUS) Laboratory at the University of Maryland has been working for a number of years on parallel support for very large knowledge bases. The main system to come out of this laboratory was the Parka system, which was the SIMD implementation of a frame-based AI KR language on the CM-2. The system was shown to be extremely fast, and to perform extremely well on very large knowledge bases

*This research was supported in part by grants from NSF(IRI-9306580), ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039, DABT-63-95-C-0037), the ARPA I3 Initiative (N00014-94-10907). Dr. Hendler is also affiliated with the UM Institute for Systems Research (NSF Grant NSF EEC 94-02384).

(having on the order of 10^5 frames). Details of the Parka system and its implementation, as well as a comparison of Parka to other efforts in VLKBs, can be found in [8,6].

There were two main problems with Parka in its SIMD form. First, there was the obvious problem that SIMD computers are currently less popular than MIMD platforms. Second, the ability to scale to much larger knowledge bases than what could be maintained in the memory of the CM2 (i.e. we needed at least one (virtual) processor per node) was constrained by the hardware. This led us to reimplement Parka on a more conventional MIMD platform, also moving from *Lisp[2] to C.

In this paper, we describe this reimplementation effort. In particular, the goal of this project is to reimplement Parka for generic MIMD computers. To reach this goal, we use the CHAOS-Library developed by Dr. Saltz and his students at the High Performance Computing Software Systems Laboratory at the University of Maryland [17]. The language is being implemented in as portable a manner as possible so as to work on a number of different MIMD computers.

We will first overview the UM Parka KR project and present some of the applications for which we are developing very large KBs. We then describe in the next two sections Parka's basic data structures and their current implementation. In Section 5 we will analyze the potential parallelism in the system. The central element of the Parka system is the inference engine which is described in Section 6. We present some performance results in Section 7 and we conclude with a summary of the contribution of this work and future research directions.

2. Parka Overview

The Parka system is a frame-based AI language (sometimes called a "property/class" system in today's literature) which was designed to be supported efficiently using high-performance computing techniques. The goal of the project is, and has always been, to develop a fairly traditional AI language/tool that can scale to the extremely large size applications mandated by the needs of today's information technology revolution.

More specifically, Parka allows the user to define a frame-based knowledge base with class, subclass, and property links used to encode the ontology. Property values can themselves be frames, or alternatively can be string, numeric values, or specialized data structures (used primarily in the implementation). The language allows exceptions, in the form of multiple-inheritance, and provides extremely efficient (and efficiently parallelizable) algorithms for performing inheritance using a true inferential-distance-ordering calculation [11].[3] Parka has also been shown to effectively compute recognition, and also to handle extremely complex "structure matching" queries – a class of conjunctive queries relating a set of variables and constraints and unifying these against the larger KB. While it is difficult to exactly compare KR languages, a very loose categorization would put Parka as more expressive than Classic [3], due to the presence of exception handling,

[2] *Lisp is the parallel version of Lisp which runs on the Thinking Machine Corporation CM-2 and CM-5 multiprocessors.

[3] Although it has been shown that IDO with fully general exceptions (i.e. cancellation links) is exponentially hard, we've demonstrated that IDO with multiple inheritance exceptions can be computed in polynomial time and is efficiently parallelizable [18].

although slightly less expressive than Loom [16] due to the lack of extensive numerical capabilities. A full description of the language, and more details on past results can be found in http://www.cs.umd.edu/projects/plus/Parka.

Parka is being used in a number of research initiatives at the University of Maryland. Some of the projects using Parka include

- A Knowledge Discovery in Databases application for a large medical knowledge base. In particular, it has been noted that data mining systems can benefit from the use of KBs to provide semantic information for datamining [4]. We are exploiting this sort of information by using medical knowledge in the datamining of an OB-Gyn patient database containing records on over 20,000 patients. The current Parka KB has over 1.2 million assertions [20].[4].
- Several Case-based planning applications including the memory-intensive case-based planning system, Caper, developed by Brian Kettler in his recent doctoral thesis [13]. The Caper KBs, the largest of which contains over 2,000,000 assertions, are described later in this paper (section 4). Parka is also being used to support a logistics planning project, jointly being developed with MITRE Corp. [10], with a KB currently containing over 650,000 assertions.
- A recent project involves the use of Parka as a hybrid knowledge and database for storing and retrieving designs and process plans for mechanical products, as part of ongoing effort to develop a new hybrid variant/generative approach to process planning in manufacturing. Parka's content-based retrieval of designs and plans is important in order to achieve the desired functionality of the process planning system, and will represent a significant advance over current "variant" approaches to process planning, which (among other things) involves retrieving process plans from databases based on fixed-length alphanumeric keys. The KB for this project is currently under construction, but is expected to dwarf those described previously, since current part databases are extremely large, containing information about the machining of thousands of parts.

3. Parka's basic data structures

The need for portability has influenced the choice of the environments to implement Parka :

Compiler: The compiler used was a standard ANSI C one. This compiler exists on all of the computers we currently access, particularly including the IBM SP-2 and the CRAY T3D.

Communication Library: Our implementation is based on functionality provided by the CHAOS run-time support package which provides scheduling libraries that have been ported to a wide range of MIMD supercomputers. (Work on the Parka project has actually led to a new scheduling approach which is being integrated into the CHAOS libraries[19]).

[4]We report KB sizes in "assertions," basically the number of links in the semantic network corresponding to the frames, as this accurately reflects the total number of relations between items in the KB and corresponds directly to the "concepts" in a description logic representation.

We have focused on the design and implementation of a very general system that can easily be ported to the different supercomputers. This means that currently, no special optimizations are done using the specific characteristics of any one machine. (Future work will include optimizing on several important architectures.)

3.1. The data structures of the KB

To use CHAOS and other generic tools, it was necessary to reimplement the Parka system to be based on arrays, rather than on the linked lists of the earlier *Lisp implementation. There are three major structures used:

Frames: The primary data structure used in the new implementation is a *frame*. A frame is an array of a variable size. Each element of the array points to an element of the Network Array described below. This pointer defines the properties of the frame. If an element points to a "normal" frame then the element represents an ISA-link. If the pointer goes to a frame that is characterized to be a property, then the link defines an explicitly valued property of the frame. This semantic difference allows us to separate the links into two sub-arrays, one for ISAs and one for properties.

A *property frame* (property) has a different structure than a normal frame. The basic structure of a property frame is again an array. This array contains two sub-arrays of the same size. For each non-ISA property in every normal frame, there is a pointer to a property frame. In the property frame, there are two links which are created, each in one of the sub-arrays. The link created in the first sub-array points back to the frame for which the property was defined. The second link points to the value for the property of that frame. In addition, a property frame also contains the arrays of a normal frame (see Figure 1). (This allows properties themselves to be frames with hierarchies and other properties, which was not allowed in the earlier SIMD system.)

Network-Array: The network array is an array which holds all of the normal and property frames in the system as sub-arrays. (This array can be quite large for very large KBs.)

Frame-Table: The frame table is an array of strings. In each element of the frame table the name of a frame is stored. The name corresponds to the name of the frame located at the same position in the Network Array. (e.g. the fifth element of the frame table contains the name of the fifth element of the network). This allows a mapping from "human" to machine descriptors, for use in querying, etc.

This representation of the KB has several considerable advantages.

- The representation of the KB as an array (of arrays) allows us to use standard communication libraries like CHAOS. In addition, computation over array-based data structures is a well-studied area for MIMD Systems.
- The separation of the properties from the ISA links allows a fast parallel implementation for inheritance algorithms.

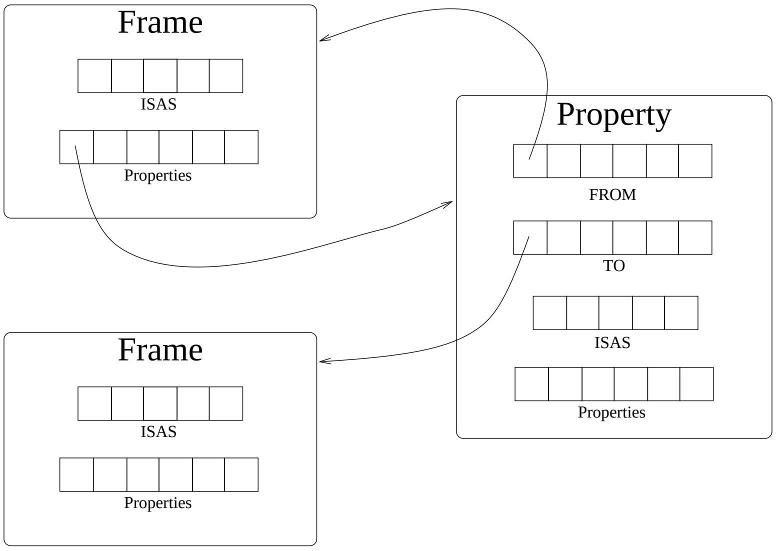

Figure 1. Frame Arrays in MIMD Parka

- The replacement of the single property link by a frame permits efficient implementations of different parallel algorithms such as recognition and structure matching.
- A separate name space (Frame Table) can be used to calculate starting points for scan operations without the presence of the whole KB.

4. Current Status

We have currently reimplemented all of the major algorithms of the Parka system – the creation of frame-based semantic networks, the performance of inheritance operations thereon, the execution of recognition queries and structure matching queries, all of which we explain describe in the following sections.

4.1. Construction of the Frames, the Network and the Frame Table

There are three methods for creating and/or loading the networks on a parallel machine.

1. The KB is defined directly by a program using commands such as *define_frame, define_property, define_isa* and *assert.*
2. A KB can be read in from an ASCII file containing these same commands.
3. If a KB is once constructed then the memory can be dumped on the disk. The dump files can be read in again later. It is possible to generate either a dump of the whole network or a distributed dump for multiprocessors. In the latter case, a separate dump for each node is created.

All three methods can be used to generate one network on a single processor or a network being distributed over the nodes of the parallel computer. There are two predefined methods that distribute the network in blocks or stripes. (It is also possible to use other methods to distribute the network which are predefined or user-defined.)

In the first two methods (where a network is created) a consistency check for the whole network is performed. For the third method it's assumed that the network is correct.

4.2. Inheritance

For the efficient performance of inheritance, we require a fast algorithm for scanning the ISA hierarchy. The efficiency of this algorithm is crucial to supporting very large knowledge bases because, as we argue in [8], most interesting inferences in large knowledge bases are dependent on the efficiency (and correctness) of the inheritance algorithm.

4.2.1. The Scanning Algorithm

The scanning algorithm is dependent on the distribution of the Network Array among the processors. The current implementation supports the two predefined types (block and cyclic) of distributions. The user is free to define other distributions, but all of these distributions must be static.

In principle, there are two approaches as to how the scan algorithm could work:

1. To expand the ISA hierarchy of a frame, a processor has to find out on which node the frame is located. If the frame is on the local node the ISA hierarchy can directly be expanded. If the frame is installed on a remote processor, then the local processor could ask for a copy of the frame from the remote processor and could then expand the hierarchy. This is essentially a "load balancing" solution, as frame information is propagated amongst processors.

2. The local processor has again to locate the frame in question and expand it directly if its local. Otherwise it will inform the remote processor to expand the corresponding frame. This approach (which we use) has more of a scheduling burden (coordinating communication) but needs less explicit load-balancing.

In the second approach, the work load is dependent on the distribution of the Network Array. Each frame is expanded on the processor which hosts that frame. The first approach does not contain such an implicit distribution of the work load. Instead, it would require that an explicit distribution of the work must be defined.

However, the amount of work to realize during the expansion of a frame is very small (only a few instructions) compared with the communication overhead. It would be very hard to implement an efficient load balancer because of the supplementary network load produced by the balancer. For this reason we chose the second approach for our implementation. Each frame is expanded by the processor on which it is stored. This leads to the following basic scan algorithm as used in this implementation:

```
1. locate the starting frame

2. look up the ISA array of the frame

3. send msgs to all processors which host a frame defined in the ISA array

4. receive all msgs from remote processors
```

```
5. for each received frame goto step 2
```

This program is executed in SPMD mode on each processor of the parallel computer. Based on this scan algorithm it's now possible to define an inheritance algorithm.

4.2.2. Implementation of the Scanning Algorithm

Our programming model attempts to have as much of a data-parallel flavor as possible. This places restrictions on the types of task-parallelism we allow. We list these restrictions below.

1. The task-graphs (network array) are static and acyclic.
2. Each data item (frame) is updated by a unique task and all data-items updated by a given task are on the same processor. This allows an owner-computes strategy.
3. The same task-graph is repetitively used for different data-sets. This allows runtime preprocessing optimizations.

These restrictions are not severe and the previously described implementation of the Parka system satisfies these restrictions.

The nodes for the task-graph are distributed across processors based on a user-specified distribution and the computation is distributed using the owner-computes rule. Since the task-graph computation is iterative, certain optimizations can be performed once in a preprocessing step and reused. One such optimization is to perform a topological sort of the DAG, thus dividing it into levels. As shown in Figure 2a, the synchronization requirements can now be confined to the levels thus increasing the computation granularity. This method has been suggested in [17], and was used to parallelize sparse matrix codes with loop carried dependencies. However global synchronization does not work very well because the synchronization constraints affect the load balancing. All processors have to wait for the slowest processor at each level. The total computation time is thus $\sum_{i=1}^{\#levels}\left(\max_{j=1}^{\#processors} Comp(proc_j, level_i)\right)$[5].

The alternative approach, considered in [5] is to use better distribution mechanisms to increase locality and then use low-latency active messages to communicate the data that does need to go off-processor. The arrival of data automatically triggers computation, thus the synchronization is implicit. The synchronization requirements of this method are very relaxed and thus the load-balance is better. The total computation time is $\max_{j=1}^{\#processors}\left(\sum_{i=1}^{\#levels} Comp(proc_j, level_i)\right)$. While the computation time is better than the level-synchronized scheme, the communication/synchronization time may be worse since more messages are sent, even though the amount of data sent is the same. [5] found that high efficiencies could be achieved on the CM-5 using its very low overhead communication support. A drawback is that a data flow programming model must be used.

Our approach is to use the level-based data-parallel model but relax the synchronization constraints by using split-phase synchronization mechanisms. This allows processors to continue processing incoming data from the next level while waiting for the slowest

[5]Where $Comp(proc_j, level_i)$ is the time used by processor j to compute level i.

```
DO K = 1, num_levels
  MPI_scatter( level(k-1))
  MPI_gather( level(k))
  COMPUTE( level(k))
END DO
```

a) Collective synchronization

```
DO K = 1, num_levels
 id = FZY_scatter( level(k-1))
 WHILE ( FZY_recv(id, data)
      == LEVEL_NOT_DONE) )
   COMPUTE( level(k))
END DO
```

b) Fuzzy synchronization

Figure 2. Collective/Fuzzy synchronization

processor from the previous level. Since each processor does not progresses to the next level until it has processed all nodes at the current level, the skew is limited to a maximum of one level. This is most useful when a processor can have a high load at one level and a low load at the next level. In such a case, that processor can catch up without slowing the others down.

Figure 2b shows how a DAG could be parallelized using such a *fuzzy barrier*. Each processor sends out all the outgoing data at the end of each level, but processes each incoming data-chunk as soon it arrives. The condition NOT_DONE remains true until all the incoming messages from a level have arrived, or when a termination condition has been detected. Thus even if processor A is on level X, processor B can begin to process data on level X+1 (until it eventually waits to receive A's message). Though, we do not discuss it here, the termination detection check can be incorporated to implement branch-and-bound algorithms. The computation time is thus better than that of the level-based scheme with no extra communication overhead.

Figures 3 and 4 shows the results of using fuzzy synchronization and the level-based runtime techniques on the inheritance network. These experiments used an inheritance network with $500,000$ classes and instances (i.e the DAG has 500K nodes) and several links between each (for a total of about 1.5M links). As can be seen from 3, fuzzy synchronization provides significant benefits over using collective synchronization. Figure 4 shows that high efficiencies can be obtained over a wide range of platforms using these techniques. The efficiencies are around 70%.

4.2.3. The Inheritance Algorithm

The inheritance algorithm we use is based on the Touretzky IDO algorithm as modified in Parka . The theoretical worst case complexity of this algorithm is $O(d * n/k)$ where d is the depth of the ISA-hierarchy, n is the number of frames in the network and k is the number of nodes of the parallel computer.

The idea of our algorithm is that in one pass through the ISA-hierarchy, we can simultaneously create a list of all properties (direct and inherited) for each node and also a list of all properties "overwritten" by more specific properties (therefore being eliminated from the first list) The difference of these two lists contains the properties inherited according to IDO. We discuss the efficiency of this algorithm in section 7.2.2 (Table 1).

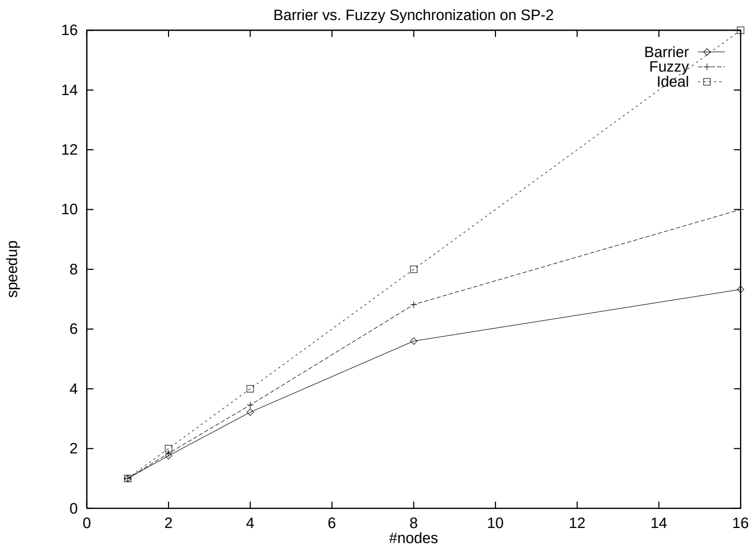

Figure 3. Barrier vs. Fuzzy Synchronization

5. Potential Parallelism in KB

In this section, we discuss the kind and degree of MIMD-parallelism we exploit. The parallelism of a KB is hidden in the ontology. In our case the ontology is represented as a directed acyclic graph. In such a graph the parallelism usable by the scanning algorithm is defined by the branch out of each frame in the network (i.e. the number of ISA links per frame).

A network in which each frame has only one ISA link defined has no parallelism exploitable by the scanning algorithm. Each frame has to be completed before the descendent frame can be attacked. Such a graph imposes serial execution of the scanning algorithm. On the other hand, a graph with depth two and a large amount of ISAs defined for the root frame has the highest possible degree of parallelism for the scanning algorithm.

This algorithm therefore seems to be well adapted for AI KBs. Generally, these are not very deep but often very bushy. The fact that the parallelism is low for narrow ISA hierarchies is not very harmful. In addition, in several types of queries there are multiple scans to perform. These scans can be combined and executed as one single larger scan. Such a method is used to implement recognition queries (see Section 6.1) and in more complex inference methods like the structure matcher (see section 6.2).

6. Complex Inferencing in MIMD Parka

As mentioned previously, we have built several other inference classes on top of the inheritance mechanism. In this section, we discuss the two most important: recognition queries and structure matching inferences.

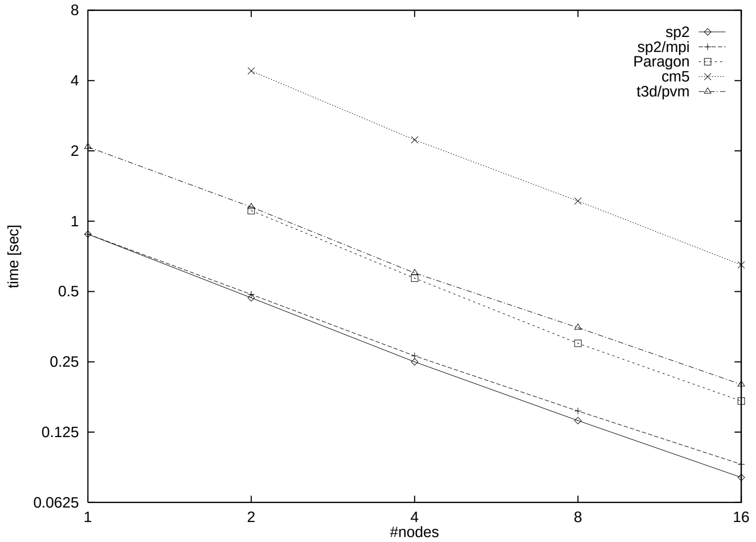

Figure 4. Performance of parallelized Inheritance Network code: Timings shown include the native communication systems for the SP2, the Paragon and the CM5 use the native communication system as well as MPI and PVM times for the SP2 and T3D machines respectively.

6.1. Recognition

We have implemented a new version of Parka's *recognition* algorithm – its means for handling conjunctive property queries.[6] The algorithm is based on the special structure of the property frames and again uses the scanning algorithm. Essentially, we handle these queries as a set of inheritance queries and some join operations, thus we can maintain the complexity of $O(d * n/k)$.

To process the conjunctive property query, we split the conjunction into the conjunctive elements. This produces a list of simple expressions of the form *(prop1, x, y)* (where x and y can both be variables, or one of the two can be a constant). So we have to find all the frames which have an explicit or an inherited definition for the given property and their value in respect to the property.

It is easy to find all explicitly labeled frames for a given property. The sub-array of the property frame with the back pointers contains all the pointers to the frames which are directly labeled for this property. The other sub-array of the property frame holds the pointers to the value for the property. On one hand, it's possible to create a list with all the frames for which the property is directly defined and which have the searched value. On the other hand, a list with all the frames which do not have the value being searched for, but which are able to be "overwritten" with the correct values during the inheritance can also be created.

The basic idea of the recognition algorithm is that all the frames which have the required property value are activated and start to send information downward along the ISA links.

[6]The SIMD algorithm for performing this was one of the main contributions of the original Parka system[7].

During this scan, all frames in the ISA hierarchy will inherit the values of the property. If the inherited information arrives on a node which is directly labeled for a property, but with a different value, then all the descendants of this node are commanded to remove the value they inherited from the same node as this node. Thus, it's possible to correctly handle all inherited properties according to IDO in one single scan.

We can find the solution to each sub-query defined by the elements of the original conjunctive property query in one single IDO run (scan run). The results of this run is a number of sets containing the frames and their value for the analyzed property. Frames which are elements of all the sets and their corresponding values are the solutions of the original conjunctive query.[7]

The potential degree of parallelism of this algorithm is high, because there are multiple starting points (all the frames with direct correct property values). At the same time it's possible to send multiple properties across the ISA hierarchy during one scan step which results in a large scan graph with a high degree of parallelism.

6.2. Structure Matcher

The second even more complex inference method is the structure matching algorithm. It allows general relation-based structures to be retrieved from a knowledge base. The algorithm takes a conjunctive query similar in form to the recognition query, but with binary constraints (i.e. with two variables). A set of these conjuncts defines a graph structure, where variables or constants are the nodes and predicates are the arcs. The algorithm compares the probe structure with the knowledge base and returns a set of all satisfying matches.

Our description of the problem of structure matching follows that given in [21]. A knowledge base defines a set, P, of unary and binary predicates. Unary predicates have the form $P_i(x)$ and binary predicates have the form $P_j(x_1, x_2)$, where each x_i is a variable on the set of frames in the KB. An existential conjunctive expression on these predicates is a formula of the form $\exists x_1, \ldots, x_m : P_1 \wedge P_2 \wedge, \ldots, \wedge P_n$, where $n \geq 1$. Our task is to retrieve all structures from memory which match a given conjunctive expression. Therefore, we would like to find all such satisfying assignments for the x_i.

We can view the problem of matching knowledge structures in two ways. The first is as a subgraph isomorphism problem[8]. We view variables as nodes and binary predicates as edges in a graph. We want to find structures in memory which "line up" with the graph structure of the query expression. The other way to view the matching problem is as a problem of unification or constraint satisfaction. If we can find a structure in memory which provides a consistent assignment to the variables x_i (i.e., unification), then that structure matches the conjunctive expression.

6.3. Overview of the algorithm

The structure matching algorithm operates by comparing a retrieval probe, P, against a knowledge base (KB) to find all structures in the KB which are consistent with P. This match process occurs in parallel across the entire knowledge base. A Parka KB

[7] This can be formalized as a set of joins in the relational calculus.

[8] More specifically, this is a problem of subgraph isomorphism with typed edges, the edges being the relations in the KB between frames.

consists of a set of frames and a set of relations (defined by predicates) on those frames. Most relations are only implicitly specified and so must be made explicit by expanding the relation with the appropriate inference method. By computing inherited values for a relation, all pairs defining the relation are made explicit. We currently allow only unary and binary relations[9].

A retrieval probe is specified as a graph consisting of a set of variables $V(P)$ and a set of predicates (or constraints) $C(P)$ that must simultaneously hold on frames bound to those variables. The result of the algorithm is a set of k-tuples, where each k-tuple encodes a unique $1-1$ mapping of frames to variables in $V(P)$, that unifies with the description of the structure in memory with $C(P)$. Figure 5 shows a simple example of the structure matching algorithm; given a knowledge base represented as a semantic network and a probe graph, the algorithm finds all matching subgraphs in the network which are isomorphic to the probe.

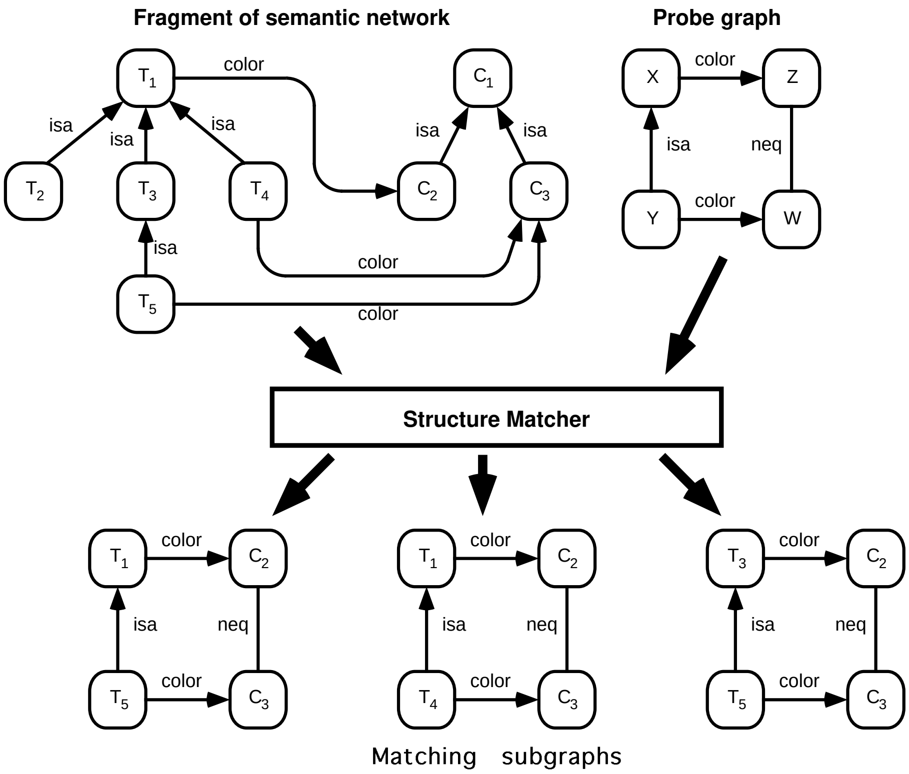

Figure 5. A simple example of structure matching.

The set of frames that can bind to each variable is initially restricted by a set of constraints indicated by unary predicates. Each unary constraint may only constrain the

[9]Future versions of Parka may support higher-order predicates.

values of one variable. Examples of these constraints are "X is a dog" or "the color of X is yellow". We allow set theoretic combinations of the unary constraints, for example "X is a dog and the color of X is yellow", or "X is a dog but X is not yellow." The domains for each variable are maintained throughout the match process and are further restricted as more constraints are processed.

Constraints between frames bound to variables are specified by a set of binary constraints. For example, we can say "the color of X must be Y", or "X is a part of Y", for some X and Y in $V(P)$. Binary constraints are processed by "expanding" the binary relation given in the constraint. By expansion we mean that all pairs participating in a relation R in the KB are made explicit by invoking the inference method for the associated predicate. The pairs allowed to participate in the expanded relation are restricted to those in the domains of the variables related by R. For example, a binary constraint may be expressed as (Color X Y). In this case the values for each concept in the domain of X are computed for the color predicate and pairs that have values outside the domain of Y are excluded.

Two additional binary predicates, "eq" and "neq" are provided to provide co-designation and non-co-designation of variables. These constraints act as a filter, eliminating any tuples from the result for which the constrained variables are(not) bound to the same frame. (Other binary predicates could be supported as well, such as inequality and other numeric constraints. However, in the current implementation, we do not provide support for these constraints.)

The result of a structure match is a set of k-tuples, where each tuple corresponds to a satisfying assignment of the k variables. Alternatively, the result can be viewed as a relation. Initially, the matcher begins with an empty set of relations. During the match, several intermediate relations may be constructed. Simple binary relations result from the expansion of a binary constraint. These are later fused (via a relational join operation) or filtered (via co-designation or non-co-designation) until a single relation remains. The algorithm selects binary constraints to process using a greedy algorithm based on a simple cost model stored in the meta-data.

7. Testing

To realize the tests it was necessary first to program a network generator. The networks generated by this program were afterwards used to test the implemented KB.

7.1. Network generator

The network generator is able to generate two kinds of KB formats:

1. ASCII file
2. memory dump files

There are three characteristics of the network that can be defined:

1. the branching factor in the network
2. the number of levels

3. the number of levels with the maximal size

Further it is possible to randomly generate properties for different frames. There is a parameter to define the density of these properties. This allows us to duplicate some of the tests made on the original Parka system on the SIMD CM-2 computer (cf. [8]).

7.2. Results on IBM SP2 (16 processors)

7.2.1. Degree of Parallelism

As mentioned above, the performance of the scan algorithm depends on the structure of the ISA hierarchy. To test this, we created two networks, both with $2,500$ frames. The first network is $2,500$ levels deep and each level consists of one frame. The second network has two levels, a first where one frame is inserted and and a second with $2,499$ frames. For the sequential algorithm the amount of work is the same for both networks. This is also true for the parallel scanning algorithm but the big difference is that for the first network there is little opportunity for parallelism – one level has to be executed after another. In the second network it's different. All the frames on the second level can be treated in parallel if they are distributed among the processors.

To execute this first test, we used 4 processors on an IBM SP2. A cyclic network distribution was used. Thus, processor 1 owned the frames $\{0, 4, 8, ...\}$, processor 2 owned the frames $\{1, 5, 9, ...\}$, etc.

The execution of the scan algorithm on the first network took $0.4sec$. The scan through the second network only $0.008sec$. Note that although the theoretical speedup should be no more than 4, we have a speedup of 50! This is related to the communication system of the SP2 (and most other MIMD machines). For the first network we have to communicate 2499 short messages, in the second one long message. The throughput of the communication system is much higher on a small number of long messages then on a large number of short messages.

This result seems to be very promising for real KB, because most of the existing KBs have a very flat ontology. For example, in a recoding of the ontology of CYC for Parka (see section 7.2.3), the deepest subgraph we found was only 27 levels deep, whereas maximum branch out was over $1,000$.

7.2.2. Scaling and Speedup

As a second test, we want to analyze how the scanning algorithm performs if the configuration of the computer changes. The results presented here are again based on generated networks. The branching factor was fixed to 3. The depth of the ISA-hierarchy was varied from 9 to 12, generating networks of about 29,500, 88,500, 265,700, and 797,000 frames respectively. (As a point of comparison, our encoding of the ontology of CYC (Section 7.2.3) has about $32,500$ frames.) A property to find was placed on the root node and the inheritance was done by one of the leaf nodes (this corresponds to the worst case for the previous Parka implementation). The absolute values represented in Figure 6 and Table 1 are extremely fast, however we note that they are not so important because the programs were not optimized for the SP2 where the tests were done. What's interesting is the comportment of the algorithms. There are two points that seem to be interesting:

1. the scalability in terms of processor number

nbr. nodes/netsize	1	2	4	8	16
29524	0.115	0.060	0.033	0.021	0.017
88573	0.345	0.175	0.091	0.050	0.032
265720	1.037	0.523	0.265	0.137	0.088
797161		1.689	0.735	0.388	0.201

Table 1
Inheritance

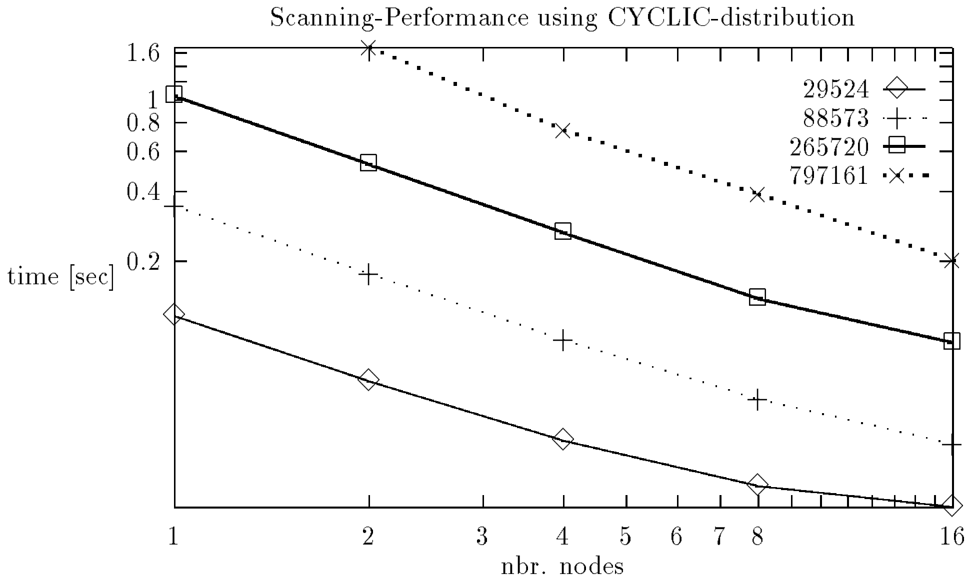

Figure 6. Scanning Performance

2. the scalability in terms of network size

On both measures, we see that the algorithms perform quite well. In addition, we believe further speedup would be available if we optimized the algorithms for this particular platform.

7.2.3. Results for the CYC Ontology

We have performed a number of tests on KBs that were not randomly created. The largest one we had used previously in testing the SIMD version of the system was the CYC ontology. As previously reported in [7], we encoded the ontology of version 8 of MCC's CYC system into Parka . In our version, this ontology has about $32,000$ frames and about $150,000$ assertions relating them (property and ISA links). The biggest difference between the generated networks and CYC is that the test network had only a single property on the root, thus making all "subgraphs" equal to the entire KB. In CYC the ISA hierarchies under particular properties are only subnetworks of the full $32,000$ frames.

We hoped to directly compare the SIMD and MIMD versions using queries on the CYC system, but most of the tests we'd run previously were, basically, too easy for the new version. In particular, the subnetworks in CYC are all much smaller than the smallest test network described in the previous section. For this reason simple queries like "What's the color of frame X" could not be used to show parallel effects – their time was on the order of 50-100μseconds on a single processor! Thus, instead of timing single inheritance queries, we generated a new class of recognition-like queries that were designed to stress the new system.

To start with, we tried the timing of recognition queries in some of CYC's largest subtrees. The results were quite promising, but again too fast for showing parallel speedups. For example, using the scan algorithms, we executed some recognitions with more than twenty conjuncts and had single processor response times of under 1/100 of a second (compared with about 1 second for the SIMD system).

To further stress the scanning algorithms and to explore speedups, we designed a new set of inferences specifically to that purpose. In particular, we used queries that would include great amounts of both inheritance and recognition. These queries were of the following form: "Give me all the frames which have one or more properties in common with frame X". To be sure to get "slow" times, we ran these queries on frames in big ISA sub-hierarchies of CYC. Two of the biggest, plant and animal, were chosen for testing, since they had very large numbers (in the tens of thousands) of other frames which shared at least one property.

The execution time for the queries "Give me all the frames with the same properties as Animal" and " Give me all the frames with the same property as Plant" are presented in Figure 7, which compares these times to the optimal speedup curve.

As one can see the recognition algorithm behaves very well. The efficiency is about 75%. That's very high with respect to the small amount of CPU time needed for the inheritance algorithm. The queries represented in the Figure 7 are much more complex than standard queries in CYC.

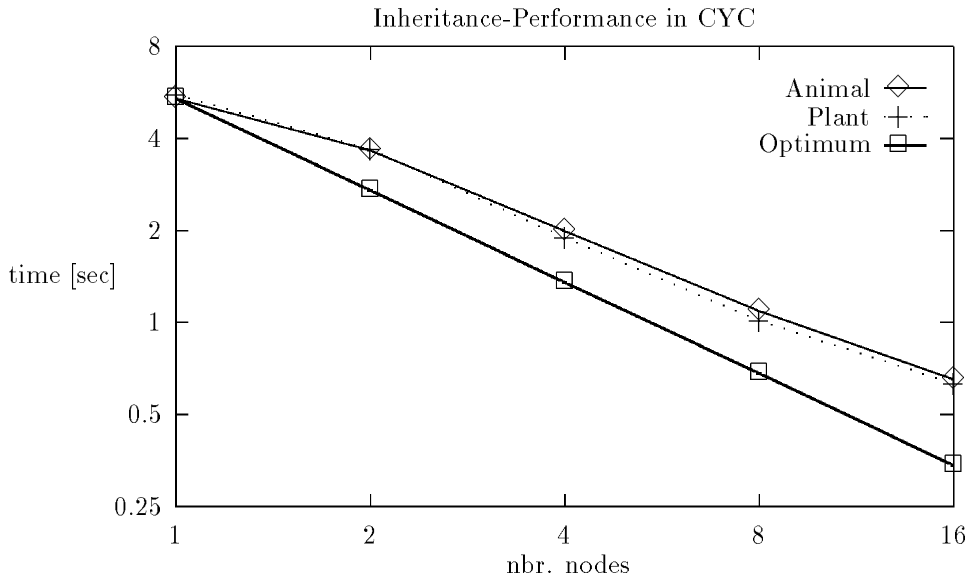

Figure 7. Recognition-Performance in CYC

7.3. Results for UM-Translog

A second domain used for testing the MIMD implementation was on knowledge bases that were created as part of doctoral thesis work on the development of a system for memory intensive case-based planning [12]. Part of this project involved the automatic seeding of large case memories by a generative planner. One domain used in this work was the "UM Translog" domain, a logistics planning domain developed for the evaluation and comparison of AI planning systems. Case-bases of various size were created, with each containing a number of plans, derivation information, and planning related ontologies.[10] To measure the performance of the structure matcher we used the UM-Translog KB in different sizes (10 cases, 100 cases and 200 cases).

The results presented in this section are especially interesting for two reasons. On one hand we are presenting the timings for the structure matcher, the most complex retrieval algorithm implemented in Parka to date. On the other hand we are also able to show, that the new Parka system has the capability to handle very large KBs on a single Sparc workstation.

The queries we used to test the system are presented in detail in Appendix A. These are all queries used by the case-based planning system during its problem solving. They were chosen at random from a large number of stored queries. The results are summarized in Table 2. Table 2a presents the timings for the six queries on a SPARC 20 using the three different KBs, and Table 2b presents the timings on 1, 8 and 16 nodes of an SP2 using the 200 case KB (the largest of the three). The actual sizes of these KBs are shown

[10]A full description of the domain, the complete specification of the planning operators used, and the case-bases themselves are available on the World-Wide Web at *http://www.cs.umd.edu/projects/plus/UMT/*.

Query	20 CB	100 CB	200 CB
1	1020	4740	6990
2	195	1305	1635
3	225	1470	1725
4	630	3570	4590
5	675	3600	4605
6	405	585	645

a: Timings on a SPARC 20 for a 20, 100 and 200 cases KB

Query	1	8	16	1:8	1:16
1	3041	546	313	69.6	60.7
2	713	129	75	69.1	59.4
3	753	135	79	70.0	56.6
4	1997	361	205	69.1	60.9
5	2003	360	206	69.5	60.8
6	284	52	29	68.2	61.2

b: Timings on a 1, 8 and 16 nodes of an SP2 on 200 case KB

Table 2
UM-Translog Timings[msec] (serial and parallel).

cases	frames	structural links	property links
20	11612	26412	114359
100	59915	130558	800481
200	123173	266176	1620456

Table 3
Sizes of the UM-Translog KBs.

in Table 3, where frames is the number of nodes in the DAG, structural links are those in the ISA ontology, and property links are all others (i.e. the number of the edges in the DAG is equal to the structural links plus the property links). (We believe the 200 case case-base to be the largest meaningful semantic net created to date, as can be seen it contains about 1.8M assertions concerning over 123,000 entities.)

As can be seen, the sequential timings range from under a second for the simplest query to about 7 seconds for the most complex. On the parallel system all queries were executed in under one second, with the simplest query taking only 29 milliseconds on 16 processors, and the most complex taking only 303 millisecond. Table 2b also shows the efficiencies of the parallel algorithm. Even in the current non-optimized form, the efficiency averages about 69.3% for eight processors and 59.9% for 16 processors.

8. Future work

We are currently working on pushing the current implementation and developing new algorithms and applications. Algorithmically, current work is focused on doing empirical research on how these algorithms scale to much larger networks and greater numbers of processors. In particular, the current implementation is done on a small IBM SP2 machine (16 "wide" nodes). The next step will be to test these programs on bigger SP2s to see how the scaling continues in larger networks and on larger machines. We also will do work on optimizing communication patterns for the SP2 implementation (which should significantly improve absolute performance) as well as porting this implementation to the Cray T3D machine, which has significantly faster communication times. We expect to be able to run extremely large networks (potentially billions of frames) extremely efficiently on this machine. First tests of the basic scan algorithm seem to support this assumption.

In addition, we are exploring the use of I/O to allow the use of secondary storage for processing extremely large knowledge bases where even the online memory of the large parallel machines is not adequate. The CHAOS runtime support package is being extended to support efficient I/O, and we are also beginning an examination of the special I/O requirements for the storage and caching needs of much larger knowledge bases (containing 100s of millions of assertions). The SP2 at Maryland is being configured to contain about 200Gigabytes of disk storage, making it uniquely suited for exploring massive knowledge and database applications.

In regard to applications, we are currently exploring the use of knowledge based techniques in the guidance of data-mining applications. In data mining applications, a knowledge repository is searched for information with the aim of obtaining as much information as possible. One weakness with current data-mining systems, however, is their lack of access to semantic knowledge about the domain being searched. Storing such semantics, however, requires much larger knowledge bases that function significantly faster than most of today's systems. Such systems will be needed for doing significant knowledge discovery against large, heterogeneous data corpora such as medical data systems or the World Wide Web. Parka promises to provide support for such knowledge bases. We are currently exploring the development of such hybrid knowledge and databases in the areas of medical informatics, military transportation logistics, and health care financing.

9. Conclusion

In summary, we have ported the SIMD Parka system to more generic MIMD machines. The system has been recoded in C and supported using runtime optimization packages developed in the high performance computing laboratory at Maryland. New "scanning" algorithms have been developed for inheritance, recognition, and structure matching inferences. These algorithms have been tested with both random networks and on two very large knowledge bases (CYC and the UM-translog case bases). Tests show that the new version is significantly faster than the SIMD system, and that it promises to scale well to knowledge bases of the order of magnitude needed for complex information technology applications.

REFERENCES

1. *Proceedings of the 12th National Conference of the American Association for Artificial Intelligence.* AAAI Press, MIT Press, 1994.
2. W. Andersen, , M. Evett, J. Hendler, and B. Kettler. *Massively Parallel Artificial Intelligence*, chapter Massively Parallel Matching of Knowledge Structures. AAAI/MIT Press, 1994.
3. A. Borgida and P.F. Patel-Schneider. A semantics and complete algorithm for subsumption in the classic description logic. *Journal of Artificial Intelligence Research*, 1, 1994.

4. Paolo Bresciani. The challenge of integrating knowledge representation and databases. In *KARP-95 Second International Symposium on Knowledge Acquisition, Representation and Processing*, pages 16–18. Auburn University, AL, USA, 1995. Extended version in IRST TR 9412-06, IRST, Povo TN (Italy).
5. Fredric T. Chong, Shamik D. Sharma, Eric Brewer, and Joel Saltz. Multiprocessor runtime support for fine-grained, irregular DAGs. Technical Report CS-TR-3266, University of Maryland, Mar 1994. To appear in Parallel Processing Letters.
6. M.P. Evett. *PARKA: A System for Massively Parallel Knowledge Representation.* PhD thesis, Dept. of Computer Science, University of Maryland, College Park, 1994.
7. M.P. Evett, J.A. Hendler, and W.A. Andersen. Massively parallel support for computationally effective recognition queries. In *Proc. Eleventh National Conference on Artificial Intelligence*. 1993.
8. M.P. Evett, J.A. Hendler, and L. Spector. Parallel knowledge representation on the connection machine. *Journal of Parallel and Distributed Computing*, 1994.
9. T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
10. J. Hendler, K. Stoffel, and A. Mulvehill. *High Performance Support for Case-Based Planning Applications*. Advanced Planning Technology. MIT/AAAI Press, Menlo Park, CA., USA, May 1996.
11. J.F. Horty, R.H. Thomason, and D.S. Touretzky. A skeptical theroy of inheritance in nonmonotonic semantic networks. Technical Report CMU-CS-87-175, Carnegie Mellon, Department of Computer Science, Pittsburgh, PA, USA, Oct. 1987.
12. B.P Kettler. *Case-based Planning with a High-Performance Parallel Memory.* PhD thesis, Dept. of Computer Science, University of Maryland, College Park, 1995.
13. Brian B. Kettler. *Case-based Planning with High-Performance Parallel Memory.* PhD thesis, University of Maryland, College Park, November 1995.
14. K. Knight and S. Luk. Building a large knowledge base for machine translation. In AAAI-94 [1].
15. Douglas B. Lenat and R.V. Guha. *Building Large Knowledge-Based Systems.* Addison-Wesley, 1991.
16. Robert M. MacGregor. A description classifier for the predicate calculus. In AAAI-94 [1].
17. Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, pages 603–611, 1991.
18. K. Stoffel and J. Hendler. An efficient inferential-distance-ordering algorithm for parallel and distributed systems. submitted for publication, 1996.
19. K. Stoffel, J.A. Hendler, and J. Saltz. High performance support for very large knowledge bases. In *Proc. Frontiers of Massively Parallel Computing.* Feb. 1995.
20. Merwyn Taylor. Hybrid kb-db. In *Proceedings of the 13th National Conference of the American Association for Artificial Intelligence.* AAAI Press, MIT Press, 1996. to appear.
21. L. Watanabe and L. Rendell. Effective generalization of relational descriptions. In *AAAI Eighth National Conference on Artificial Intelligence.* 1990.

A. UM-Translog Queries

```
;;; QUERY 1: "find all plans for getting a package from a LOC to a LOC in DIFF
              city connected by a direct route"
(query! '(:and
              (#!instanceOf      ?plan  #!E-Plan)
              (#!instanceOf      ?goal  #!CE-Located-At)
              (#!everyInstanceOf ?pkg   #!Package)
              (#!everyInstanceOf ?oloc  #!Place)
              (#!everyInstanceOf ?dloc  #!Place)
              (#!instanceOf      ?ocity #!City)
              (#!instanceOf      ?dcity #!City)
              (#!instanceOf      ?sit   #!Initial-Situation)
              (#!instanceOf      ?sval  #!Situated-Value)
              (#!instanceOf      ?pkg-pvar #!Plan-Variable)
              (#!instanceOf      ?dest-pvar #!Plan-Variable)
              (#!everyInstanceOf ?route     #!Route)
              (#!plan-Goal=      ?plan ?goal)
              (#!thing=          ?goal ?pkg-pvar)
              (#!var-Value=      ?pkg-pvar ?pkg)
              (#!location=       ?goal ?dest-pvar)
              (#!var-Value=      ?dest-pvar ?dloc)
              (#!situation-Before= ?plan ?sit)
              (#!location-sit=   ?pkg   ?sval)
              (#!situation=      ?sval  ?sit)
              (#!sslot-Val=      ?sval  ?oloc)
              (#!in-City         ?oloc  ?ocity)
              (#!in-City         ?dloc  ?dcity)
              (#!neq             ?ocity ?dcity)
              (#!route-Origin=   ?route ?ocity)
              (#!route-Destin=   ?route ?dcity)
              ))

;;; QUERY 2: "find all plans using a particular train station, Hub-Ts"
(query! '(:and
          (#!instanceOf      ?act    #!P-Move-Vehicle)
          (#!instanceOf      ?plan   #!E-Plan)
          (#!everyInstanceOf ?train  #!Train)
          (#!instanceOf      ?p-varp #!Plan-Variable)
          (#!instanceOf      ?pvara  #!Plan-Variable)
          (#!vehicle=        ?act    ?p-varp)
          (#!var-Value=      ?p-varp ?train)
          (#!destin=         ?act    ?pvara)
          (#!var-Value=      ?pvara  #!Hub-Ts)
          (#!action-Of-Plan  ?act    ?plan)
          ))

;;; QUERY 3: "find all top-level plans using a Regular truck
              (uses IDO and transitive inheritance)"
(query! '(:and
          (#!everyInstanceOf ?truck1 #!Regular-Truck)
          (#!instanceOf      ?act    #!P-Move-Vehicle)
          (#!instanceOf      ?plan   #!E-Plan)
          (#!instanceOf      ?plant  #!E-Plan)
```

```
          (#!instanceOf      ?case   #!Case)
          (#!instanceOf      ?pvart   #!Plan-Variable)
          (#!var-Value=      ?pvart  ?truck1)
          (#!vehicle=        ?act    ?pvart)
          (#!action-Of-Plan  ?act    ?plan)
          (#!tmember-Of      ?plan   ?plant)
          (#!plan-of-Case    ?plant  ?case)
          ))

;;; QUERY 4: "find all plans for getting an object from a LOC to a
              LOC in SAME city"
(query! '(:and
          (#!instanceOf      ?plan  #!E-Plan)
          (#!instanceOf      ?goal  #!CE-Located-At)
          (#!everyInstanceOf ?pkg   #!Package)
          (#!everyInstanceOf ?oloc  #!Place)
          (#!everyInstanceOf ?dloc  #!Place)
          (#!instanceOf      ?city  #!City)
          (#!instanceOf      ?sit   #!Initial-Situation)
          (#!instanceOf      ?sval  #!Situated-Value)
          (#!instanceOf      ?pkg-pvar  #!Plan-Variable)
          (#!instanceOf      ?dest-pvar #!Plan-Variable)
          (#!plan-Goal=      ?plan ?goal)
          (#!thing=          ?goal ?pkg-pvar)
          (#!var-Value=      ?pkg-pvar ?pkg)
          (#!location=       ?goal ?dest-pvar)
          (#!var-Value=      ?dest-pvar ?dloc)
          (#!situation-Before= ?plan ?sit)
          (#!location-sit=   ?pkg   ?sval)
          (#!situation=      ?sval  ?sit)
          (#!sslot-Val=      ?sval  ?oloc)
          (#!in-City         ?oloc  ?city)
          (#!in-City         ?dloc  ?city)
          ))

;;; QUERY 5: "find all plans for getting an object from a LOC to a LOC in
              DIFF city"
(QUERY! '(:and
          (#!instanceOf      ?plan  #!E-Plan)
          (#!instanceOf      ?goal  #!CE-Located-At)
          (#!everyInstanceOf ?pkg   #!Package)
          (#!everyInstanceOf ?oloc  #!Place)
          (#!everyInstanceOf ?dloc  #!Place)
          (#!instanceOf      ?ocity #!City)
          (#!instanceOf      ?dcity #!City)
          (#!instanceOf      ?sit   #!Initial-Situation)
          (#!instanceOf      ?sval  #!Situated-Value)
          (#!instanceOf      ?pkg-pvar #!Plan-Variable)
          (#!instanceOf      ?dest-pvar #!Plan-Variable)
          (#!plan-Goal=      ?plan ?goal)
          (#!thing=          ?goal ?pkg-pvar)
          (#!var-Value=      ?pkg-pvar ?pkg)
          (#!location=       ?goal ?dest-pvar)
          (#!var-Value=      ?dest-pvar ?dloc)
```

```
        (#!situation-Before= ?plan ?sit)
        (#!location-sit=   ?pkg   ?sval)
        (#!situation=      ?sval  ?sit)
        (#!sslot-Val=      ?sval  ?oloc)
        (#!in-City         ?oloc  ?ocity)
        (#!in-City         ?dloc  ?dcity)
        (#!neq             ?ocity ?dcity)
        ))


;;; QUERY 6: "find compatible packages and vehicles"
(query! '(:and
          (#!everyInstanceOf ?pkg   #!Package)
          (#!ancestor        ?ptype #!Package)
          (#!ancestor        ?vtype #!Vehicle)
          (#!everyInstanceOf ?veh   #!Vehicle)
          (#!instanceOf      ?pkg   ?ptype)
          (#!instanceOf      ?veh   ?vtype)
          (#!pv-Compatible=  ?ptype ?vtype)
          ))
```