Design, Implementation, and Analysis of a Parallel Description Classifier*

Eric R. Melz
Robert M. Macgregor
USC Information Sciences Institute
4676 Admiralty Way, Marina Del Rey, CA 90292
Email: melz@isi.edu, macgregor@isi.edu
Phone: (310) 822-1511
Fax: (310) 823-6714

Abstract

A classifier is a central reasoning component of modern knowledge representation systems. Classifiers provide such fundamental intelligent services as concept categorization, instance recognition, and query processing. Unfortunately, as the size of the knowledge base grows, classifiers become less useful because the classifier must process a significant fraction of the knowledge base to perform any given inference. This paper investigates the extent to which parallel processing may be applied to the classification problem. We describe a MIMD implementation of a parallel classifier which uses a message-passing paradigm to effect interprocessor communications. Simulations and analysis of a local-area network implementation of the parallel classifier indicate that very large speedups may be obtained, and that speedups are limited only by the depth of the knowledge base. Preliminary results indicate that graph partitioning algorithms that cluster interdependent portions of the knowledge base may help to improve the efficiency of the parallel classifier.

* Kevin Knight and Milind Tambe provided comments on an earlier draft which greatly improved this paper. Jeff Koller gave us useful advice on parallel machines and introduced us to PVM. Support for this work was provided by the Advanced Research Projects Agency under contract no. DABT63-91-C-0025. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or the U.S. Government.

Introduction

A description classifier [Schmolze & Lipkis, 1983] is a central component of many modern knowledge representation (KR) languages. Classifiers provide many useful services within a unified framework, including categorization of new concepts, recognition of instances, and answering of queries about the state of a knowledge base. An undesirable property of current classifiers is that the speed of any given inference is inversely proportional to the size of the knowledge base that the classifier maintains. This is because any instance, concept or query that the classifier processes must be compared with a significant fraction of the knowledge base in order to compute an inference. As knowledge bases increase in size to hundreds of thousands and even millions of concepts, the speed, and hence the utility, of current classifiers becomes unacceptably poor.

A natural way to overcome the performance limitations of current classifiers is to employ parallel processing technology. We agree with numerous researchers (e.g., [Kitano, 1993, Shastri, 1986, Waltz 1990]) who argue that parallel processing is essential if AI applications are to exhibit human-like performance in real-world domains. Recent developments in hardware technology have made the prospect of parallel AI, and in particular, a parallel classifier, particularly viable. There is a trend towards using off-the-shelf RISC microprocessors in highly scaleable configurations. For example, Cray's T3D computer achieves parallelism by employing up to 2048 DEC Alpha processors. This new generation of computers tend to be far more cost-effective than their vector-processing and SIMD counterparts, which use expensive custom microprocessors.

In this paper, we propose a MIMD implementation of a parallel classifier. We take as a starting point a state-of-the-art serial classifier capable of classifying descriptions expressed in first-order predicate calculus [Macgregor, 1994]. We envision the entire process of parallel classification occurring in three phases. The first phase is a rapid "loading" phase in which a set of conceptual definitions are read in and distributed among processors in a parallel machine. Processors receiving definitions create classes that represent the meaning of the definitions, and classify the classes to build a tentative generalization hierarchy. After the loading phase is a background classification phase, in which the generalization hierarchy is refined over a protracted period of time while the user or application is busy performing unrelated tasks. In this phase, classes migrate between processors and find their most specific position in the genralization hierarchy. The final phase is rapid query processing phase, where queries issued by the user or application simultaneously match many parts of the distributed knowledge base.

In this paper, we describe a MIMD implementation of the first phase, namely, the rapid loading of concepts. This phase involves parsing a file of definitions into basic data structures, performing inheritance and other transformations on the data structures to convert the conceptual representations into a canonical form, and classifying the canonicalized data structures to construct a tentative generalization hierarchy. Other researchers have discussed or implemented parallel inheritance systems within the context of SIMD computation (e.g. [Fahlman, 1979, Evett, 1994, Aronis, 1993]). However, the data structures and computations required by our classifier are highly nonuniform, and hence are not amenable to the SIMD approach. To our knowledge, no other parallel system has been implemented that performs all of the component computations (i.e., inheritance, canonicalization, and classification) of the loading phase.

The organization of the paper is as follows. First, we provide an overview of a serial version of the predicate-calculus classifier. Next, we discuss the design and implementation of the parallel classifier. After that, we present a series of performance results based on an implementation of the parallel classifier running on a local-area network of workstations, and, with the use of a simulator, predict how the classifier will perform on multiprocessor configurations with many processors. Finally, we discuss the possibility of improving the efficiency of the parallel classifier with the use of partitioning algorithms that cluster related classes within processors.

The Serial Predicate Calculus Classifier

The basic function of a classifier is to place descriptions of concepts or individuals into a generalization hierarchy. In this section, we provide a brief overview of the serial predicate-calculus classifier that was used as a basis for developing the parallel classifier. For a more detailed description of the classifier, and examples of the classifier's use, see [Macgregor, 1994].

The classifier makes use of three important data structures:

- Descriptions- A description represents a one-place predicates defining a set of individuals, or an n-place predicate defining a set of tuples. A description consists of a *name* that uniquely identifies the description, a list of *variables*, a predicate-calculus *definition*, and a flag indicating whether or not the description is *partial*, that is whether the description's definition specifies necessary and sufficient conditions or only necessary conditions.
- Classes- Classes are the basic unit of classification: the generalization hierarchy consists of a directed, acyclic graph of classes. Associated with each class are a list of *descriptions* that define the class (a class may have more than one description if those descriptions are equivalent), *a canonical set,* consisting of a normalized representation of the "meaning" of the class, an *elaborated set* consisting of an embellished version of the canonical set, and a flag indicating whether or not the class is *primitive*, that is, whether or not the descriptions associated with the class are partial.
- Sets- Sets are graphs of arbitrary complexity that represent an interpretation of a description's definition. Nodes in a set's graph may represent either sets or individuals. Nodes representing individuals may represent either variables or constants. Edges between the nodes represent relations and are labeled with either user

defined sets (e.g., child), or built-in relations (e.g., greater-than). As mentioned above, each class points to two distinct sets, a *canonical* set and an *elaborated* set.

The classification process proceeds in six distinct steps:

- Reading- A concept definition is read in from a file, and a description is created for the definition. In addition, a class and a blank canonical and elaborated set are created and associated with the description.
- Construction- A rudimentary canonical set is created by parsing the class's description into a conceptual graph.
- Expansion- References to user-defined sets are expanded by inheriting the structure associated with the referenced set into the current canonical set.
- Canonicalization- The canonical set is converted into a canonical format by performing a series of normalization operations on the set. For example "orphan" nodes (i.e. those with no incident edges) are deleted, and nodes representing sets with only one member are converted into individual nodes.
- Elaboration- The canonical set is copied to the elaborated set, and new structure (i.e. edges and nodes) is added to the elaborated set via the application a series elaboration rules. An example of an elaboration rule is "If a node denotes a set, then the node must have a cardinality that is an integer greater than or equal to 0".
- Classification- The class is placed in the generalization hierarchy by performing a depth first traversal of the generalization hierarchy, and performing a *subsumption* test at each class in the hierarchy. The subsumption test determines if the class being classified is more specific than or equivalent to the class in the hierarchy. A class *A* subsumes a class *B* if the classifier can prove that *A*'s canonical set is isomorphic to a subgraph of *B*'s elaborated set[1]. The classifying class is placed under the most specific class that subsumes it. If it is equivalent to another class (i.e., *A* subsumes *B* and *B* subsumes *A*), it merges with that class, and its description is added to the list of descriptions of the class it merges with.

The Parallel Classifier

Architecture

The parallel classifier employs a hierarchical division of labor: a single "supervisor" process oversees the work of many "worker" processes. To avoid bottlenecks, the workload on the supervisor must be minimized. In our implementation, the supervisor is

[1] The subsumption test is incomplete; however, in practice it tends to produce high-quality results.

responsible for reading definitions from a file, creating initial descriptions, classes and (empty) sets, sending the initial data structures to the workers, and collecting results from the workers. Workers are responsible for performing the core classifier tasks, including construction, expansion, canonicalization, elaboration, and classification.

Execution of the parallel classifier can be broken down into 3 roughly distinct phases:

- Preparation- The supervisor reads a file of concept definitions, and creates initial data structures for the definitions. The supervisor also creates a dependency graph to keep track of expansion-related interdependencies between descriptions. The supervisor partitions the descriptions, along with dependency information, into a number of queues equal to the number of workers, and then sends each worker one of the queues. Currently, this partitioning process randomly distributes descriptions to queues, but special partitioning algorithms may be applied to cluster interrelated descriptions within queues (see below).
- Work- Each worker receives a set of descriptions from the supervisor. Associated with each description is a list of other workers that depend on the results of computations which involve the description. For example, if another worker*W1* needs the canonical set *cs* produced by the current description for the expansion of one of *W1*'s canonical sets, the current worker should send *cs* to *W1* once *cs* has been computed. Each worker proceeds to construct, expand, canonicalize, elaborate, and classify the data structures assigned to it. If at any point a piece of data (e.g. a canonical or elaborated set) that a computation needs is missing, the computation is aborted and put on a task queue. The computation will be resumed when the requisite data arrives on the worker.
- Result Collection- The supervisor broadcasts requests to all workers for the results of their classification computations. When the workers have finished their work, they send back a queue of classification results, which the supervisor uses to update a global generalization hierarchy.

Implementation

The parallel classifier is implemented in Common Lisp. It is directly adapted from the serial classifier, and most of the core routines (e.g. canonicalization, elaboration, etc.) from the original classifier remain intact on the parallel classifier. Message-passing is implemented via the PVM (Parallel Virtual Machine) message-passing library [Geist *et al.*, 1994]. The classifier communicates with PVM, which is C-based, via a foreign function interface. Since PVM is capable of only transmitting byte vectors, routines were added to "vectorize" and "devectorize" complex data structures such as sets and descriptions. For example, vectorization of a graph involves assigning a series of numbers to nodes in the graph, and encoding edges in the graph as a set of triples, where each triple consists of an integer associated with the edge's label, and the integers associated with each node on either end of the edge.

Distribution Issues: Dependencies and Efficiency

If a description *d1* references a description *d2*, the canonical set associated with *d1* may not be expanded until the canonical set associated with *d2* has finished its expansion. This is because *d1*'s canonical set may need to inherit structure from *d2*'s canonical set in the expansion phase. The fact that constraints exist on the order of description processing means that the parallel classifier can only produce a speedup of n, where n is the number of nodes in a knowledge base, if no description in the knowledge base references any other description. In fact, the parallel classifier can only execute in time $O(d)$, where d is length of the longest path in the graph of description dependencies. This fact is verified by our simulation results (see below).

Another consequence of ordering constraints is that the amount of time a worker spends in an idle state is dependent on the particular assignments of descriptions to workers. To see this, consider 2 linked lists, each of length d. If one list is assigned to each worker in a 2-worker parallel classifier, the classifier finishes its computation in $O(d)$ time. However, if the first half of each list is assigned to one worker and the second half of each list is assigned to the other worker, the parallel classifier takes at least $O(3/2d)$ time to execute. This is because the latter worker must wait until the first half of one of the lists is processed by the former worker before it may begin working.

The fact that an arbitrary assignment of descriptions to workers can adversely affect the classifier's efficiency suggests that an effective load-balancing stratgegy should take into consideration the structure of the dependency network as well as the number of descriptions that are assigned to each worker. We conjecture that workers should be assigned highly interconnected subgraphs of the description dependency network for the parallel classifier to perform at an optimal efficiency. We explore the validity of this conjecture later when we examine the effect of applying graph partitioning algorithms to the description dependency network.

Knowledge Bases

To test performance of the parallel classifier, we used three knowledge bases:

- Artificial- An artificial knowledge base, consisting of concepts of uniform size was generated by an automatic knowledge-base generator. For the experiments described in this paper, the artificial knowledge base was a forest of 4 binary trees, each 11 classes deep. This knowledge base, with the addition of several relations, totals 2,068 classes.
- Penman Upper Model - The Penman Upper Model is a high-level knowledge base used to support natural language applications like English generation [Penman, 1989] and machine translation [Pangloss, 1994]. This knowledge base contains 786 classes.
- Sensus- Sensus [Knight & Luk, 1994] is a large, 70,000-term knowledge base synthesized from resources like the Penman Upper Model, the widely used Wordnet

semantic network [Miller, 1990], and Longman's English dictionary. For the experiments in this paper, we used a 7,270 class subset of the Sensus knowledge base instead of the entire knowledge base for the sake of manageability.

Statistics on the knowledge bases are presented in Table 1. "Max Depth" refers to the length of the longest path in the description-dependency network used by the supervisor.

Knowledge Base	# of Classes	Max Depth
Forest (Artificial)	2068	11
Penman Upper-Model	786	22
Sensus	7270	28

Table 1. Knowledge Base Statistics

Predicting Performance on Message-Passing Multiprocessors

In this section, we describe two ways of predicting performance on a MIMD machine, first using simulation, and second, extrapolating from results obtained on a network of workstations. In this and subsequent sections, we use standard definitions of speedup and efficiency. Speedup is the ratio of the serial and parallel execution times:

$$Speedup = \frac{SerialElapsed}{ParallelElapsed} \qquad (1)$$

Efficiency is the ratio of a given speedup to a linear speedup. A linear speedup occurs when *Speedup* = *n*, where *n* is the number of processors:

$$Efficiency = \frac{Speedup}{n} \qquad (2)$$

Simulation

Performance of the parallel classifier can be studied by simulating the algorithm's execution on a serial machine. We constructed a simple simulator designed to mimic the behavior of a MIMD implementation of the parallel classifier. In order to keep the simulator as simple as possible, several assumptions were made, including:

- Canonicalization and elaboration of a description takes one timestep.
- "Forwarding" a relation from one worker to another takes one timestep.
- There is no overhead associated with the initial phase in which descriptions are distributed to workers.

In all other respects, the simulator closely resembles the parallel classifier. The simulator uses the same dependency graph as the parallel classifier to determine how classes are to be transferred between workers, and workers process descriptions in the same order as the parallel classifier. Execution of the simulator halts when all workers are idle for at least one timestep.

The simulator allows us to predict such performance factors as speedup and processor utilization. It also provides a useful tool for analyzing dependency-related bottlenecks in the parallel classifier. For example, if one processor is idle for a significant fraction of timesteps, the simulator can provide an "execution trace" that can be used to inform the construction of better description-distribution algorithms (i.e., see the section on graph partitioning below).

Predicting MIMD Performance by Adjusting Ethernet Performance

Since the parallel classifier is implemented in Common Lisp, it is not executable on any of today's multiprocessing computers, which currently do not provide strong support for Lisp. However, it is possible to predict speedups on a multiprocessor by adjusting speedups measured on a network of workstations to compensate for computational and communication differences between multiprocessors and local-area networks. The formula for speedup on a multiprocessor MIMD machine is:

$$Speedup_{MIMD} = \frac{SerialElapsed_{MIMD}}{ParallelElapsed_{MIMD}} \qquad (3)$$

To calculuate the MIMD speedup, we must obtain estimates for $ParallelElapsed_{MIMD}$ and $SerialElapsed_{MIMD}$. To do this, we determine the amount of time the ethernet version of the parallel classifier spends computing tasks, waiting in an idle state, and communicating, and adjust those quantities by constants reflecting the ratio of CPU and Communication speeds between an ethernet and a multiprocessor. Hence,

$$SerialElapsed_{MIMD} = SerialElapsed_{ether} \bullet Ratio_{CPU} \qquad (4)$$

and

$$ParallelElapsed_{MIMD} = (CPU_{ether} + Idle_{ether})Ratio_{CPU} + Comm_{ether} \bullet Ratio_{Comm} \qquad (5)$$

Here, we assume that idle time is due to one worker waiting for the results of a computation on another worker. Hence, if computation time is reduced, idle time will also be consequently reduced. Although further steps are need to determine $Idle_{ether}$ and $Comm_{ether}$, a full derivation of these quantities is beyond the scope of this paper.

Results

In this section, we present performance results for the parallel classifier. First, we evaluate the simulator by comparing idle times predicted by the simulator and idle times produced in an actual run. Next we present a set of speedups on various knowledge bases running the parallel classifier on a set of five workstations linked via an ethernet. Finally, we present the results of simulations which predict the performance of the parallel classifier on multiprocessors consisting of thousands of nodes.

Comparison of Simulated and Actual Results

For our first experiment, we sought to determine the validity of the parallel classifier simulator. There are several ways in which the simulator may be compared to the actual implementation. One way is to compare the speedups predicted by the simulator with actual speedups. If there is a reasonable correspondence, we can be reasonably certain that the simulator adequately mimics the real parallel classifier. However, we can perform a finer-grained comparison by comparing the simulator's prediction of the amount of idle time spent on each worker with the actual idle times, $Idle_{ether}$, that an execution of the parallel classifier produces. To effect this comparison, we collect the number of timesteps for each worker on the simulator that are in an "Idle" state. As previously discussed, idle time occurs when a worker is waiting for the results of another worker and has no other work it can perform. If the number of simulated idle timesteps reflects the magnitude of the idle time spent on an actual execution of the classifier, we have evidence that our simulator is working properly.

To test this, we scrambled the descriptions in the artificial forest KB over 4 workers. That is, each worker received fragments from all four trees instead of an intact, self-contained tree. This had the effect of greatly skewing the magnitude of idle times observed on the parallel classifier when run on an ethernet. Execution on the ethernet, excluding the time for the supervisor preparation and result collection phases, produced idle times of 3.79, .39, .28, and .46 seconds, for each respective worker. When the same configuration was executed on the simulator, the number of idle timesteps for the corresponding workers was 382, 50, 39, and 57. While we do not wish to make strong claims about the fit of the simulated data to the actual data, the fact that the ordering of the idle times is consistent, with *worker1* > *worker4* > *worker2* > *worker3*, is suggestive that the simulator provides an accurate reflection of the actual execution of the parallel classifier.

Ethernet and Predicted MIMD Results

To further evaluate the simulator, we can compare the speedups predicted by the simulator and speedups obtained by the actual classifier. We measured speedups in four separate experiments. Our first experiment used an "unscrambled" version of the artificial forest KB, in which a separate tree was placed on each worker. This experiment is interesting because the networks on each worker are completely self-contained: no interprocessor communication is required. The second experiment used a "scrambled" version of the artificial forest KB, where descriptions were distributed in the skewed manner discussed in the previous section. The third experiment randomly distributed descriptions of the Penman Upper Model KB, and the final experiment randomly distributed descriptions of the Sensus KB. As in the previous section, we exclude times for the preparation and result collection phases to facilitate comparison of the ethernet implementation with the simulator.

Table 3 compares real speedups obtained by running the parallel classifier on a network of 5 HP730 workstations, predicted MIMD speedups, obtained using equations 3-5, and speedups obtained by executing the simulator. The predicted MIMD speedups used adjustment constants of $Ratio_{CPU}$ = .5 and $Ratio_{Comm}$ =.004. These values were based on performance figures for HP730s and the CRAY T3D, a popular modern multiprocessor [Cray, 1995, Hewlett-Packard, 1995]. The Integer SPECint92 benchmark for HP's PA RISC chip is 200, and for the T3D's Alpha chip is 136. The bandwidth for an ethernet is 10Mb/s, and the bandwidth for the T3D ranges from 2400Mb/s to 614Gb/s depending on the number of processors in use (there are up to 2048 on the T3D). We used the most conservative figures possible for the computation of $Ratio_{CPU}$ and $Ratio_{Comm}$.

Knowledge Base	Ethernet (Real) Speedup	Predicted MIMD Speedup	Simulated Speedup
Unscrambled Forest	4.06	4.08	3.99
Scrambled Forest	2.62	2.99	2.30
Penman-Upper	1.51	2.70	2.23
Penman-Upper+Wordnet	3.06	3.26	2.31

Table 3. Parallel Classifier Speedups on various Kbs with 4 workers.

Comparing the speedups obtained on the ethernet with the speedups predicted by the simulator, speedups appear to be in the same ballpark. However, 3 of the ethernet speedups are higher than the simulated speedup, and one is lower. This is likely due to a time distortion that occurs when a great deal of interprocessor communication occurs on an ethernet. Since an ethernet consists of a single shared bus, a great deal of contention can occur when many processors are simultaneously trying to communicate to each other. This is not the case on a multiprocessor, and our simulator does not model this effect. Indeed, when we apply equations 3-5 y to obtain the predicted performance on a MIMD machine, we see that the predicted MIMD speedups are uniformly higher than the simulated speedups. Hence, the simulator appears to provide a lower bound on the speedups that would occur on an actual MIMD machine.

One apparent anomaly should be noted: the values 4.06 and 4.08 for the respective ethernet and MIMD speedups on the unscrambled forest KB appear to exceed the theoretical upper-bound of the speedup of 4.0. This is probably due to the small variances that can occur between runs. In this case, either $SerialElapsed_{MIMD}$ may have been slightly higher than normal, or $ParallelElapsed_{MIMD}$ may have been slightly lower than normal. Another interesting possibility is that distributing concepts over processors produces superlinear gains in performance: since each worker only has a small number of concepts to work with relative to the serial version, there is less of a load on the operating system in terms of allocating resources, garbage collection etc.. Consistent with the hypothesis that superlinear speedups can occur, the ethernet execution of the Sensus KB produced significantly larger speedup than the ethernet execution of the Penman Upper Model KB. This may be because the Sensus KB is significantly larger, and thus the effect of superlinear speedups may be amplified.

Simulated Results for Large Knowledge Bases

Since we do not have the resources to execute the parallel classifier on more than a couple dozen workstations, we must rely on the results of the simulator to observe how the classifier behaves as we scale up the number of processors.

Tables 4 and 5 show various statistics produced by simulating the parallel classifier for the Penman-Upper and Penman-Upper + Wordnet KBs, respectively. The total number of idle steps are shown in the second column, and the total number of steps in which a class was copied between processors ("Unpacks") are shown in the third column. The fourth column shows the total number of "Cross-arcs", that is, the number of dependency arcs whose incident nodes reside on different workers.

# of Workers	Idle	Unpacks	CrossArcs	Speedup	Efficiency
2	28	312	584	1.39	.70
4	68	556	766	2.23	.56
8	267	722	846	3.54	.44
16	657	828	890	5.54	.35
32	1031	871	910	9.36	.29
64	1966	896	916	13.79	.22
128	5342	912	921	14.29	.11
256	10065	925	926	17.09	.07
1024	43344	926	926	17.86	.02

Table 4. Simulation results of Penman-Upper KB

# of Workers	Idle	Unpacks	CrossArcs	Speedup	Efficiency
2	289	3865	5302	1.27	.64
4	382	4952	6371	2.31	.58
8	514	5784	6919	4.29	.54
16	1010	6360	7169	7.95	.50
32	1566	6780	7310	14.90	.47
64	5769	7057	7373	23.15	.36
128	4804	7254	7417	48.15	.38
256	12003	7351	7432	69.90	.27

1024	46742	7428	7440	121.17	.12
2048	108173	7437	7442	121.17	.06
4096	214665	7441	7442	129.82	.03
8192	444038	7444	7444	129.82	.02

Table 5. Simulation of Penman-Upper + Wordnet KB

The Penman-Upper KB yields a maximum speedup of 17.86, and the Penman-Upper + Wordnet KB yields a maximum speedup of 129.82. As noted by several other researchers (e.g., [Fahlman, 1979, Evett, 1994, Aronis, 1994]), speedups obtained by parallel inheritance systems are limited by the depth of the knowledge base. Since all descriptions that a description *d* references must have a canonical set before *d* may be parsed into a canonical set (otherwise the expansion phase will produce incorrect results), the parallel classifier can run no faster than the longest dependency path in the knowledge base. Maximum speedups may be obtained by dividing the number of nodes in the knowledge base (corresponding to the elapsed serial time) by the length of the longest path in the knowledge base (corresponding the minimum elapsed parallel time). In the case of the Penman-Upper KB, *Speedup* = (786/22) = 35.72. But recall that the simulator counts each interprocessor copy as taking one timestep. Hence, the parallel time is effectively doubled by the simulator, and the speedup = 35.72 / 2 = 17.86.

The fact that the depth of the knowledge base tends to be the principal limiting factor on the possible speedups means that very large speedups are possible with large knowledge bases. Real-world knowledge bases tend to be fairly shallow relative to their overall size, as can be seen by comparing the Penman Upper Model and Sensus statistics in Table 1: even though the overall size of the knowledge base increased by 6484 nodes, the overall depth only increases by 6. Assuming the depth of the entire Sensus KB (approximately 70,000 nodes) remains roughly the same as the subset of the Sensus KB that we use, we can expect a speedup of at least (70,000 / 35) / 2 = 1000. Of course, this assumes that we have more than 70,000 processors at our disposal, which is not the case with current hardware. To get good speedups when there are fewer processors than nodes in the graph, the processors must be well utilized; i.e., processor efficiency must be high.

Classification Results

As we mentioned in the introduction, this paper is concerned only with the loading phase of classification. We assume that after the loading phase, there will be a protracted phase in which concepts can migrate between processors in a “background” mode to get increasingly better classification results. However, since we do include the classification phase in our current implementation, it is interesting to see how well the classifier does even without the background classification phase. One way to measure the parallel classifier’s performance in this regard is to compare the generalization hierarchy produced the serial classifier with that produced by the parallel classifier. To do this, we collected a list of every possible pair of descriptions that entered into a subclass-superclass relationship after the classifier had finished classifying all descriptions in the Penman Upper Model KB. We collected one such list for the serial classifier and another for the parallel classifier. Comparing the two lists, 3% of the pairs in the serial list were not in the parallel list. Hence, it appears that the current implementation of the parallel classifier produces very good classification results even without the background classification phase.

It should be emphasized that the “missed” classification inferences are not wrong in any sense, they are merely not as good as they might otherwise be. For example, the parallel classifier might classify the concepts “grandchild” and “child” as siblings under the concept “person”. This is not incorrect; however, a better answer would be to classify “child” under “person”, and “grandchild” under “child”.

Effect of Graph Partitioning on Performance

In the runs presented up to this point, descriptions were distributed randomly among processors. We conjecture that the overall efficiency of the parallel classifier can be improved by distributing descriptions so that mutually dependent nodes tend to be clustered within the same processor. If the knowledge base is naturally structured such that clusters of highly interdependent descriptions may be isolated, it may be possible to apply a graph-partitioning algorithm to identify those clusters. In this section, we evaluate the effect of various graph-partioning algorithm on the performance of the parallel classifier.

For our investigation, we examined the effect of three algorithms on the efficiency of the parallel classifier. The first algorithm was proposed by Kernighan & Lin (1970). The basic 2-partition version of the Kernighan & Lin algorithm is an iterative algorithm that initially randomly distributes nodes (i.e., description names) to 2 partitions, and then swaps nodes between the partitions until the number of cross-arcs is minimized. The 2-partition algorithm can be generalized to k-partitions by a technique that we refer to as “chipping”. The chipping algorithm begins by dividing the k partitions into 2 “virtual” partitions consisting of the leftmost partition and all of the remaining partitions. The Kernighan & Lin is applied to the virtual partition, and then a new virtual partition is created, consisting of the partition to the right of the leftmost partition, and the remaining partitions to the right of that partition. Chipping terminates after k-1 iterations, where each iteration consists of the application of the Kernighan & Lin algorithm to a virtual partition.

The second algorithm we evaluated was a simple algorithm which we refer to as “Graph Numbering”. Nodes in the graph were numbered according to a preorder depth-first traversal, and the nodes were assigned to partitions based on their number, with nodes number 1,...,(n/k) placed in the first partition, (n/k)+1,...,2(n/k) in the second partition, etc..

For our final test, we developed a more sophisticated algorithm which we refer to as the “Branch Growing” algorithm. The basic idea of the algorithm is to “seed” each partition with nodes close to roots in the graph, and then iteratively “grow” each seed in each partition by adding connected nodes to each partition. If no connected nodes are available, a partition restarts the process with a new "seed", again, preferably one close to a root.

Table 6 summarizes the simulated results of applying the various partitioning algorithms to a classifier configured with 8 workers, using the Penman Upper Model KB. The first three rows show the effect of limiting the Chipping algorithm to 0 iterations (i.e., random partitions), 1 iteration, and the maximum amount of iterations required by the algorithm. Efficiency of the algorithm improved from .44 to .51, a 15% improvement in performance. The Graph Numbering algorithm yielded an efficiency of .49, or an 11% efficiency improvement. The Branch Growing algorithm outperformed both the Chipping and Graph-Numbering algorithm: efficiency was boosted to .60, for an overall performance improvement of 36%.

Algorithm	Idle	Unpacks	CrossArcs	Speedup	Efficiency
Random	267	722	846	3.54	.44
Chipping, 1 iter.	392	590	741	3.56	.44
Chipping, max iter.	365	392	535	4.07	.51
Graph Numbering	710	103	172	3.93	.49
Branch Growing	354	171	196	4.79	.60

Table 6. Effect of Various Partitioning Algorithms on Penman Upper Model KB (with 8 workers)

These preliminary results are strongly suggestive that significant performance benefits can be obtained when partitioning algorithms are applied to a knowledge base. However, further investigation is needed to determine exactly which types of knowledge bases are amenable to partitioning, and which partitioning algorithms are best-suited for general knowledge bases.

Conclusion

We have demonstrated the parallel approach to classification can yield significant performance benefits. On the largest knowledge base tested, our simulator predicted an increase in speed of 130 times over the serial version, and analysis of the Penman knowledge base indicates that speedups up to 2,000 may be possible with enough processors. We presented preliminary results indicating that graph partitioning algorithms can be employed to significantly increase the efficiency of the parallel classifier.

This paper demonstrates that MIMD parallelism can be effectively applied to problems in knowledge representation. Since MIMD parallelism offers far more programming flexibility than SIMD parallelism, and because MIMD hardware is becoming increasingly accessible, we expect to see more AI applications taking advantage of parallel processing technology in the near future. Hopefully this will enable AI applications to tackle real-world task domains.

References

Aronis, J. M. (1993). *Implementing Inheritance with Roles on the Connection Machine*. Ph.D dissertation, Univ. Pittsburgh, Pittsburgh, PA.

Cray (1995). Cray T3D. World Wide Web Page http://www.cray.com/PUBLIC/production-info/mpp/T3D_overview.html.

Evett, M. P. (1994). *PARKA: A System for Massively Parallel Knowledge Representation*, Ph.D. dissertation, Univ. Maryland, College Park, MD.

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

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994). *PVM*. MIT Press, Cambridge, MA.

Hewlett-Packard (1995). World Wide Web Page http: //billsp.ch.apollo.hp.com/wsg/products/prodhome.html

Kernighan, B. W., and Lin, S. (1970). An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Technical Journal*, February 1970, pp. 291-307.

Kitano, H. (1993). Massively Parallel Artificial Intelligence and Grand Challenge Applications. In *Innovative Applications of Massive Parallelism, Papers from the 1993 Sprint Symposium*. Tech Report SS-93-04, AAAI Press, Menlo Park, CA.

Knight, K., and Chander, I. (1994). Automated Postediting of Documents. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. AAAI Press, Menlo Park, CA.

Macgregor (1994). A Description Classifier for the Predicate Calculus. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. AAAI Press, Menlo Park, CA.

Miller, G. (1990). WordNet: An On-Line Lexical Database. *International Journal of Lexicography*, 3(4).

Penman (1989). Penman Documentation. USC/Information Sciences Institute Tech Report.

Pangloss (1994). The Pangloss Mark III Machine Translation System. CMU Tech Report.

Schmolze, J. G., & Lipkis, T. A. (1983). Classification in the KL-ONE knowledge representation system. In *Proceedings of IJCAI-83*.

Shastri, L. (1986) Massive parallelism in artificial intelligence. Tech. Rep. MS-CIS-86-77 (LINC LAB 43), Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia.

Waltz, D. (1990). Massiveliy Parallel Artificial Intelligence. In *Proceedings of the 8th National Conference on Artificial Intelligence*. AAAI Press, Menlo Park, CA.