Distributed Path-Based Inference in Semantic Networks

Chain-Wu Lee
Dept. of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, NY 14260
lee-d@cse.buffalo.edu

Chun-Hsi Huang, Sanguthevar Rajasekaran
Dept. of Computer Science and Engineering
University of Connecticut
Storrs, CT 06269
{huang, rajasek}@cse.uconn.edu

Laurence Tianruo Yang
Dept. of Computer Science
St. Francis Xavier University
Antigonish, NS, B2G 2W5, Canada.
lyang@stfx.ca

D. Frank Hsu
Dept. of Computer and Information Science
Fordham University
Bronx, NY 10458
hsu@cis.fordham.edu

Abstract

This paper introduces the task model, instruction set, (path-based) reasoning scheme, software infrastructure, as well as the experimental results on a SUN multiprocessing machine, of a new distributed semantic network system. The experiments demonstrate significant speedups.

1. Preface

The backbone knowledge representation systems in such applications as natural language processing and expert systems have long been implemented as *semantic networks* [2, 6, 1, 13, 15], in which the knowledge entities are represented by nodes (or vertices), while the edges (or arcs) are the relations between entities. Most of these systems support the fundamental *path-based* inference scheme, in which, by tracing the arcs between nodes, new knowledge can be derived. Previous and current parallel semantic network systems such as the SNAP [3, 4, 18, 12] and PARKA [5, 17] use the "marker passing algorithm" [9] as the inference mechanism. The marker passing algorithm is a synchronous algorithm and cannot be efficiently implemented on asynchronous (distributed) systems. Besides, in the marker passing algorithm, the nodes are statically assigned to processors. Without applying additional load balancing algorithm, which causes significant overhead, workloads among processors can become extremely unbalanced.

Several factors attribute to the difficulties implementing a semantic network on distributed systems. First, the semantic network is a naturally fine-grained system, since only a few instructions are needed for each basic operation. Second, the computation structure is so highly irregular and dynamic that efficient runtime load balancing is necessary. Third, the heavy communication and random communication patterns usually make it difficult to achieve communication efficiency.

The proposed distributed semantic network system, the TROJAN, uses a combination of task and data parallelism, a task-based message-driven model, while handling given queries, as opposed to the pure data parallel schemes used in other parallel semantic network systems. In the task-based message-driven model, queries are decomposed into tasks and then scheduled for execution. Other system support activities are also broken down into system tasks. The TROJAN consists of two collaborating components: the *host* module and the *slave* module. The host module interacts with the user and processes information for the slave modules, while the slave modules perform task execution. Communication between modules is accomplished by an object-oriented message packing system. Current implementation of the TROJAN focuses on the *path-based* knowledge inferences, using the MPICH-G2, ANSI C, as well as the *flex* lexical analyzer and the *yacc* parser generator. As indicated in the experimental result section, the performance tests demonstrate promising speedups.

2. Instruction Set and Reasoning of TROJAN

2.1. Instruction Set

Commands in TROJAN are generally categorized into three groups: (1) network building (e.g. `build` and `assert`, etc.), (2) inferencing (e.g. `find`, `findassert`, etc.) and (3) others (e.g. `nodeset operation commands`, etc.) that are answered directly inside the host module. Commands in groups (1) and (2) usually need to communicate with slave PEs. The TROJAN provides three commands, `build`, `assert` and `add`, to construct the semantic network. The syntax of these commands are listed below:

- `build`: (`build` {*relation nodeset*}*)
- `assert`: (`assert` {*relation nodeset*}* *context-specifier*)
- `add`: (`add` {*relation nodeset*}* *context-specifier*)

These commands put a node into the network with an arc labeled *relation* to each node in the following *nodeset*, and returns the newly built node. An attempt to build a currently existing node will immediately return such an existing node. `build` creates an unasserted node unless an asserted node exists in the network with a superset of the relations of the new node, in which case the new node is also asserted. `assert` is just like `build`, but creates the node with assertion. `add` acts like `assert`, but in addition triggers forward inference. *relation* has to be a unit-path and non converse. Converse relations *relation-*, which connects each node of the nodeset to the built node, are constructed implicitly by the system. (An example is given in 2.2.)

Several (path-based) inference commands are provided by the TROJAN system, including the `find` family (`find`, `findassert`, `findbase`, `findconstant`, `findpattern` and `findvariable`). Details are elided due to the page limit.

2.2. Path-Based Reasoning

Path-based inference is the fundamental inference mechanism of all semantic networks. By tracing the arcs between nodes, new knowledge can be derived. For example, the command

```
(assert member Socrates class human)
```

defines the concept "Socrates is human", shown in Fig. 1. In the system, two base nodes `Socrates` and `human` are generated by the command. The molecular node `M1` (index depending on the current knowledge base state) is generated by the system, where "!" stands for the "assertion" concept. Two forward links `member` and `class` are defined by the user, two reverse links `member-` and `class-`, indicated by dash lines, are generated by TROJAN automatically. Hierarchical concepts "human is an animal" and "animal is a thing" can be constructed similarly. And by following the links of "subclass-" and "supclass", derivation can be made that Socrates is human, an animal and a thing.

In TROJAN, the relation between two nodes can be either explicit (direct arc between two nodes), or implicit (an arc across several intermediate nodes). The implicit relation is defined by the TROJAN command `define-path`. For example, in Fig. 1, the `class` inference rule can be defined by the following command:

```
(define-path class (compose class
 (kstar (compose subclass- supclass))))
```

, which indicates that the `class` relation can be defined by a direct arc `class` followed by zero or more occurrences of combinations of the direct arcs `subclass-` and `supclass`. Several path definition primitives are defined in the path-based inference rules. These will be used in Section 3 for defining the parallel computation task units in the system.

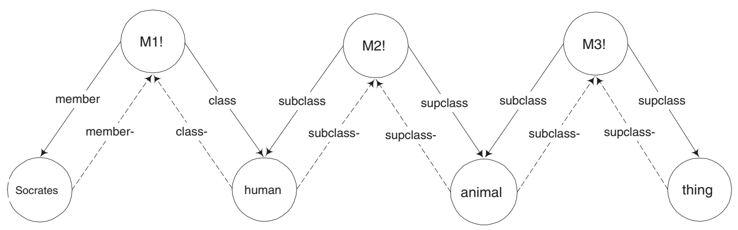

Figure 1. TROJAN Hierarchy Example

The TROJAN command `find`, designed for path-based inference queries, has the following syntax:

(`find` {*path nodeset*}*).

This command returns a set of nodes such that each node in the set has every specified *path* going from it to at least one node in the accompanying *nodeset*. For example, in Fig. 1, when the command

```
(find subclass human supclass animal)
```

is issued, the TROJAN answers `M2` since `M2` is the only node with incident edge `subclass` leading to node `human` and edge `supclass` to `animal`. When the command

```
(find subclass (human animal))
```

is issued, the TROJAN answers (`M2 M3`) since `M2` and `M3` each has an edge `subclass` to *either* node `human` *or* `animal`.

3. Software Architecture

The host system is composed of the following major components. The (*language front-end*) interacts with user and decomposes the commands into either knowledge or tasks. All the preprocessing and distributing are carried out in the *command processing module*. The *object-oriented packing module* is the communication channel between PEs. When the slave module finishes a query, the answer messages are then sent back to the *host answer processing*

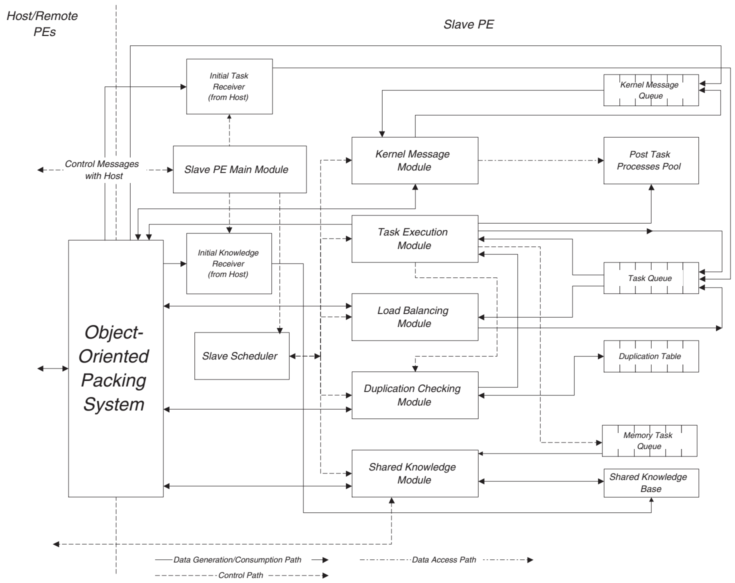

Figure 2. Slave System Architecture

module of the host system to be merged into a final inference conclusion. Some knowledge is kept in the *host knowledge base* for simple queries.

The major components comprising the slave system are as follows. The *shared knowledge management module* stores and exchanges knowledge in the *shared knowledge base*. The *task execution module* is the kernel of task execution. Several sub-modules are embedded in the task execution module, including the *kernel message module*, the *task execution engine*, and the *load balancing module*, etc. The *duplicate checking module* records the answers that have been reached to save repeated executions. The *slave scheduler* schedules task execution and swapping. The *object-oriented packing system* is similar to that of the host.

Fig. 2 illustrates the software architecture of the slave system (host system architecture elided).

3.1. Task Model

In the path-based inference, every task carries a starting node N, a path P, and other bookkeeping information. Yet the task alone cannot solve the TROJAN queries. Unlike many other parallel systems where tasks are so called *"fire-and-forget"* tasks, there are three major components in the TROJAN task execution model: the *tasks*, the *post task processes* (ptask) and the *kernel messages* (kmesg). When a query is issued, the TROJAN decomposes it into two components: the task, carrying the information for realizing the query; and the ptask, indicating current status and what needs to be done once the tasks are solved. During the execution of the task, new tasks/ptask might be generated and solved. In case the task is solved, a kmesg is sent back to its corresponding ptask. The ptask, triggered by the kmesg, might generate new tasks or send another kmesg back to its parent ptask depending on the status.

This *task-ptask-kmesg* model can be viewed as generalized function executions, where the functions are executed in parallel whenever possible. While a function is called, the *task* part is performed. Upon completion, a *kmesg* containing the answer is returned, and the *ptask* will decide whether any further processing of the answer is necessary. The traditional programming model, which uses a stack as the supporting mechanism (which is inherently sequential in terms of stack growth and shrink), is limited to the sequential computers. This motivates our design of a model expressing the function behavior while maintaining the capability of being parallelized. The formal definitions of the task, ptask and kmesg are described as below.

Definition 1 (Task)
A *task* T_k is defined by a tuple of $\langle P_{id}, P, N\rangle$ where P_{id} is an identification tag which represents where the task comes from (or the parent ptask id), P is a path and N is a node.

Definition 2 (Post Task Process (ptask))
A *post task process* (ptask) is defined by the tuple of $\langle Id, P_{id}, Act, Ans\rangle$ where Id is the identification tag for the ptask itself, P_{id} is the identification tag for the parent ptask, Act indicates what kind of job the corresponding task is performing and what kind of action should be performed when accepting incoming kmesgs, and Ans is the answer nodeset we got so far. Some of the entries are path symbol-specific and therefore are not listed here.

Definition 3 (Kernel Message (kmesg))
A *Kernel Message* (kmesg) is defined as a tuple of $\langle Id, NS, Aid\rangle$ where Id is an identification tag (for ptask) and NS is the body of the message, which is a nodeset in the TROJAN implementation. Aid is an auxiliary data structure which is used only for the inference of some particular operations. (`RELATIVE-COMPLEMENT` and `RANGE-RESTRICT`, specifically).

3.2. Task Execution Engine

The task execution module consists of several smaller components: the *kernel message queue*, the *kernel message processor*, the *kernel message receiver*, the *post task process pool*, the *task execution engine*, and some *task queues*. Furthermore, the *load balancing module* is hooked up with the ptask pool for load balancing among the PEs.

When a query is made by the user, the TROJAN decomposes it into tasks. In the mean time, ptasks are spawned to represent a specific action to take after the answers of the tasks are generated. At the beginning the tasks and the ptasks are located at the Host PE. The task distribution module sends the tasks to the slave PEs to be executed by the slave task execution engine. Executing a task might generate new tasks and ptasks when the solution needs to be investigated further or it might generate kernel messages when a task is solved.

The *kernel message receiver* acts as a daemon for remote kernel messages, which are placed into the *kernel message queue*. The *kernel message processor* then dequeues the messages from the kernel message queue, looks for the matching ptask from the *post task process pool* and executes them. The execution of a ptask might generate new kmesgs, new tasks, or new ptasks, all being put into proper places to be picked up and executed again. Meanwhile, the *load balancing module* is invoked periodically to balance the load difference between different PEs. It transmits the ptasks to the remote PE and places them back in the proper location when the incoming ptasks are received.

3.2.1. Phase Transition A task in the TROJAN system is a non-preemptive, indivisible execution unit carrying the information needed for realizing the queries. A task performs only one function at a time, but depending on the *status* and the *action* of the task, a task can generate new tasks performing different functions. It also may produce system inquiries to other modules such as the knowledge sharing module and the duplicate checking module.

The life cycle of a task starts from the *enqueuing phase* and goes through the following phases until the task is terminated: the *knowledge access phase*, the *execution phase* and the optional *duplicate checking phase*.

During the enqueuing phase, the task is spawned and allocated into the task queue. Before any execution can be realized, the system must ensure node N is local. If the node has been in the local PE already, the task will then change the status to either the task execution phase or the duplicate checking phase based on certain criteria. Otherwise the task enters the knowledge access phase until node N is transferred from the remote PE and is accessible to the task. The task in the execution phase will be executed. Instead of being executed directly, if a task enters the duplicate checking phase, the system will check whether the task has been solved by other PEs.

3.3. Load Balancing

The migration of ptasks imposes overheads. When moving a ptask to a remote PE, the dependencies of task-ptask-kmesgs in a local PE are destroyed. To maintain the dependencies among different objects, some bookkeeping procedure needs to be performed. Also the P_{id} in each data structure has to be consistently updated to ensure the coherence. Another immediate overhead accompanying load balancing is the introduction of new knowledge faults. The TROJAN implements a load balancing strategy , the *affinity based scheduling* [10], in which a task stays within a specific PE till completion. This strategy saves a significant amount of time on network traffic and eliminates possible knowledge faults.

3.4. Other Slave Modules

Shared Knowledge Management Module
The basic unit in the TROJAN shared-knowledge module is a “block” of semantic nodes. Each block contains a certain amount of semantic node information. Since the path-based inference does not generate new nodes during the inference, there is no node writing actions and the write coherence checking can be saved. This property greatly simplifies the current design of the shared knowledge module in TROJAN. The knowledge base can be separated into the *private knowledge system*, where knowledge has permanent residence, and the *cache knowledge system*, where knowledge can be swapped out due to capacity limits. Each knowledge system has two major embedded components: the knowledge base, storing the TROJAN nodes and arcs, and the control data structure, storing control information for knowledge base accesses.

Control Modules
Two control modules are related to knowledge access, both monitored by the scheduler: the *knowledge task processor* (KTP) and the *knowledge task receiver* (KTR). The knowledge task processor fetches knowledge tasks from the knowledge task queue (KTQ) and executes them according to the defined actions. The knowledge task receiver receives the dispatched knowledge tasks from the system buffer and arranges them into the knowledge task queue.

Duplicate Checking Module
Several components comprise the duplicate checking module, including three data receivers: the *duplicate query receiver*, the *duplicate response receiver* and the *duplicate answer register message receiver*; four storage data structures: the *duplicate answer register message queue*, the *duplicate query task queue*, the *duplicate answer repository*, and a *duplicate task hashing table*, and two execution modules: the *duplicate checking engine* and the *duplicate checking answer registrant* . All the receivers and the duplicate checking engine are monitored by the scheduler and are activated periodically.

Object-Oriented Message Packing Module
Conventional message-passing systems, such as the PVM or MPI, provide only essential functions to transfer basic

types of information such as characters, integers, and floating point numbers. Creating a type template in such systems for dynamic data structures (e.g. linked lists), or composite data structures, imposes significant message communications. In addition, the data format conversions for different representation systems incur constant overheads. The object-oriented message packing module (OOMP) needs the system designer to distinguish the pack/unpack function calls from the send/receive function calls. A sending buffer and a receiving buffer are maintained. When the user packs data into the system, the OOMP module translates it into byte codes and stores them in the sending buffer. The buffer is flushed only upon the request of the programmer. Generic communication functions are then invoked to transmit the buffer. After receiving the message at the destination PE, the OOMP module stores the data in the receiving buffer. The destination PE then unpacks and restores the data from the receiving buffer to its original format.

3.5. Host System

The TROJAN user interface implementation follows a context-free grammar specified in SNePSUL [16]. The command processing module receives commands from the parser, shapes them into proper data structures, dispatches them to the designated slave PEs and updates the associated data structures in the host knowledge base. Due to the page limitation, the readers are referred to a previous article [11] on TROJAN for host system details.

4. Experimental Results

The current implementation of TROJAN focuses on *path-based* knowledge inferences, using ANSI C, MPICH-G2, along with the *flex* lexical analyzer and the *yacc* parser generator. MPICH-G2 is a portable implementation of the Message Passing Interfaces (MPI) supported by the Globus middleware for the Grid environments [7, 8, 14]. Tests of individual TROJAN components and the overall speedups have been performed on a SUN Enterprise 4000 server, containing 1GB of memory and 8 168-MHz UltraSPARC CPUs.

In the experiments using multiple inheritance classification trees (every non-leaf node has 4 children and every non-root node has 2 parents), The query is in the format of

`(find (subclass- supclass ...subclass- supclass)` $(B_{i_1} \dots B_{i_7}\ B_{i_8})$)

The path length is 16 in the test query. The same query was asked twice to examine the effect of cache knowledge. In total 50 randomly generated test cases are sent to the system to measure the average performance. Fig. 3 and Fig. 4 show the expected total number of faults in all three sites and the speedup for each query, respectively.

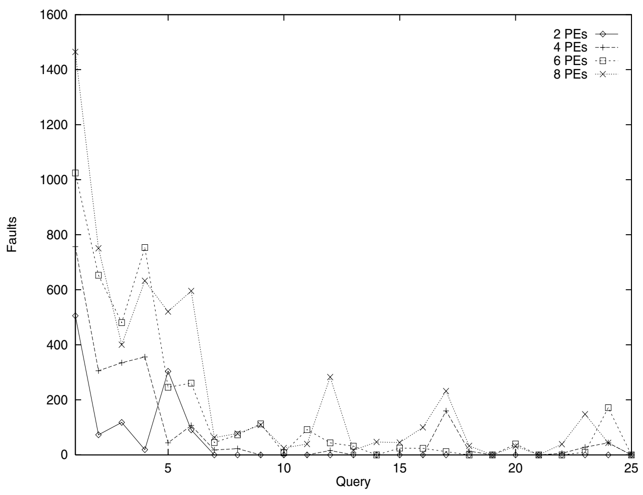

Figure 3. Multiple Inheritance Classification Tree: Faults

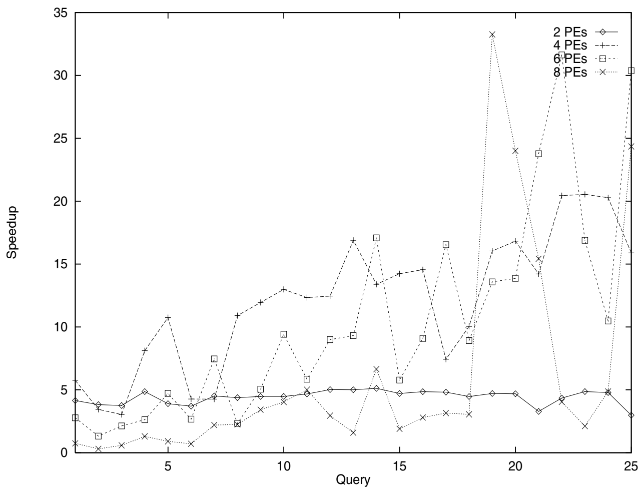

Figure 4. Multiple Inheritance Classification Tree: Speedups

Figure 5 shows the picture of the average cache hit ratio v.s. speedup, indicating that the more PEs in the system, the more sensitive the performance is to the hit ratio. The reason is that the penalties accessing a remote node are relatively low in a system with a smaller number of PEs. When the number of PEs increases, the penalties become significant. A good node allocation strategy is crucial to achieve a higher hit ratio. The results also indicate that the impact by knowledge faults are relatively mild when the number of PEs is low. With more PEs, the hit ratio plays a more crucial factor determining the performance.

Faults	Speedup	2-PE	4-PE	6-PE	8-PE
Yes	Ave.	3.27	8.09	6.66	7.94
	Std. Dev.	1.10	5.56	4.22	4.40
No	Ave.	3.52	10.58	14.82	15.25
	Std. Dev.	1.08	3.80	7.06	6.53

Table 1. Speedup Distribution

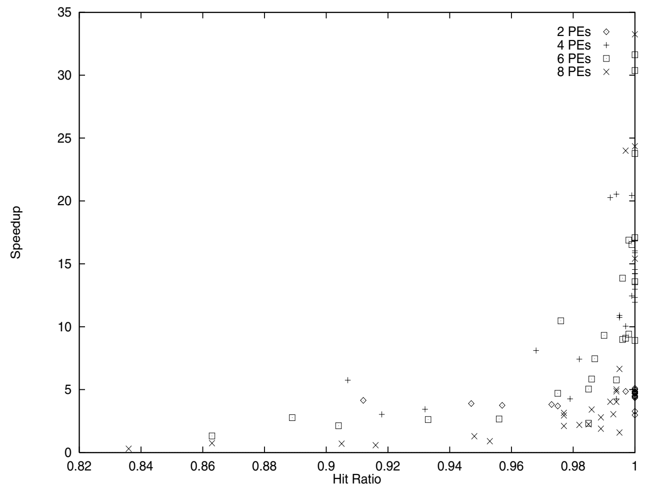

Figure 5. Hit Ratio v.s. Speedup

Table 1 also indicates that as the number of PEs increases, the miss rate also increases. Since the hit ratio is crucial to the performance, reducing possible data faults becomes essential while improving the system performance. From time to time, the speedups in the experiments are super linear. While examining the program by the profiler `gprof`, such speedups are caused by the dramatic time decrease in the task execution module. Further examination indicates that these speedups are caused by the diminishing number of node location lookups in the shared knowledge module, in both private knowledge and cache knowledge.

References

[1] R. J. Brachman and J. G. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Sci.*, 9:171–216, 1985.

[2] N. Cercone. The ECO Family. *Computers Math. Applic.*, 23(2-5):95–131, 1992.

[3] S. Chung and D. I. Moldovan. Modeling Semantic Networks on the Connection Machine. *Journal of Parallel and Distributed Computing*, 17:152–163, 1993.

[4] R. F. DeMara and D. I. Moldovan. The SNAP-1 Parallel AI Prototype. IEEE Trans. on Parallel and Distributed Systems,4(8):841–854, Aug. 1993.

[5] M. P. Evett, J. A. Hendler, and L. Spector. Parallel Knowledge Representation on the Connection Machine. *Journal of Parallel and Distributed Computing*, 22:168–184, 1991.

[6] S. E. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge*. The MIT Press, 1982.

[7] I. Foster. The Grid: A New Infrastructure for 21st Century. *Physics Today*, 55(2):42–47, 2002.

[8] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[9] J. A. Hendler. Massively-Parallel Marker-Passing in Semantic Networks. *Computers Math. Applic.*, 23(2-5):277–291, 1992.

[10] E. Lazowska and M. Squillante. Using Processor-Cache Affinity in Shared-Memory Multiprocessor Scheduling. *IEEE Trans. on Parallel and Distributed Systems*, 4(2):131–143, Feb. 1993.

[11] C.-W. Lee, C.-H. Huang, and S. Rajasekaran. Trojan: A scalable parallel semantic network system. In *Proceedings of the 15th IEEE International Conference on Tools eith Artificial Intelligence*, pages 219–223, Sacramento, CA, 2003.

[12] D. Moldovan, W. Lee, C. Lin, and M. Chung. SNAP, Parallel Processing Applied to AI. *IEEE Computer*, pages 39–49, May 1992.

[13] D. I. Moldovan, M. Pasca, S. Harabagiu, and M. Surdeanu. Performance issues and error analysis in an open-domain question answering system. *ACM Tran. on Information Systems*, 21(2):133–154, 2003.

[14] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[15] S. C. Shapiro. Knowledge Representation. *Lynn Nadel, Ed. Encyclopedia of Cognitive Science, Macmillan Publishers Ltd.*, 2:671–680, 2003.

[16] S. C. Shapiro and The SNePS Implementation Group. *SNePS-2.4 User's Manual*. Department of Computer Science at SUNY Buffalo, 1998.

[17] K. Stoffel, J. Hendler, J. Saltz, and B. Anderson. Parka on MIMD-Supercomputers. Technical Report CS-TR-3672, Computer Science Dept., UM Institute for Advanced Computer Studies, University of Maryland, College Park, 1996.

[18] M. Surdeanu, D. I. Moldovan, and S. M. Harabagiu. Performance Analysis of a Distributed Question/Answering System. *IEEE Trans. on Parallel and Distributed Systems*, 13(6):579–596, 2002.