INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Massively Parallel Real-Time Reasoning with Very Large Knowledge Bases:

An Interim Report

D. R. Mani* **Lokendra Shastri†**

TR-94-031

August 1994

Abstract

We map structured connectionist models of knowledge representation and reasoning onto existing general purpose massively parallel architectures with the objective of developing and implementing practical, real-time reasoning systems. SHRUTI, a connectionist knowledge representation and reasoning system which attempts to model reflexive reasoning, serves as our representative connectionist model. Realizations of SHRUTI are developed on the Connection Machine CM-2—an SIMD architecture—and on the Connection Machine CM-5—an MIMD architecture.

Though SIMD implementations on the CM-2 are reasonably fast—requiring a few seconds to tens of seconds for answering queries—experiments indicate that SPMD message passing systems are vastly superior to SIMD systems and offer hundred-fold speedups. The CM-5 implementation can encode large knowledge bases with several hundred thousand (randomly generated) rules and facts, and respond in under 500 milliseconds to a range of queries requiring inference depths of up to eight.

This work provides some new insights into the simulation of structured connectionist networks on massively parallel machines and is a step toward developing large yet efficient knowledge representation and reasoning systems.

*E-Mail: mani@icsi.berkeley.edu

†E-Mail: shastri@icsi.berkeley.edu

1 Introduction

Connectionist models are fast developing into widely explored architectures for cognition and intelligence. These models use a large number of simple nodes which are profusely interconnected by direct hard wired links, carrying simple, scalar messages. Massive parallelism is an important feature of any connectionist model. Since any system that purports to model human cognition must use some form of massive parallelism if it has to react in real-time (Feldman and Ballard, 1982; Shastri, 1991; Newell, 1992), *structured* connectionist models—with their inherent parallelism and their ability to represent structured knowledge—seem to be promising architectures for high-level—or symbolic—processing. Several structured connectionist models have been proposed for rule-based reasoning, language processing, planning and other high-level cognitive processes (Barnden and Pollack, 1991). From a practical standpoint, if such systems have to be fast, efficient and usable, we will need to be able to simulate or emulate them on massively parallel platforms. From a cognitive standpoint, where our concern is to design, test and prototype connectionist models of cognition, we would require suitable platforms for implementing and experimenting with these highly parallel models. Hence mapping connectionist systems onto currently existing massively parallel architectures appears to be an avenue worth exploring.

In this report we investigate the mapping of structured connectionist models of knowledge representation and reasoning onto existing general purpose massively parallel architectures with the objective of developing and implementing practical, real-time reasoning systems. We define *rapid* or *real-time reasoning* to be reasoning that is fast enough to support real-time language understanding. We can understand written language at the rate of about 150–400 words per minute—i.e., we can understand a typical sentence in a second or two (Carpenter and Just, 1977).

SHRUTI, a connectionist knowledge representation and reasoning system which attempts to model reflexive reasoning (Shastri and Ajjanagadde, 1993), will serve as our representative connectionist model. Efficient realizations of SHRUTI are developed on the Connection Machine CM-2—an SIMD architecture—and on the Connection Machine CM-5—an MIMD architecture.[1] We shall use the term *parallel rapid reasoning system* to designate these SHRUTI-based, massively parallel systems that can handle very large knowledge bases and perform a large yet limited class of reasoning in real-time.

Though SIMD implementations on the CM-2 are reasonably fast—requiring a few seconds to tens of seconds for answering simple queries—experiments indicate that SPMD message passing systems are vastly superior to SIMD systems and offer hundred-fold speedups. The CM-5 implementation can encode large knowledge bases with several hundred thousand (randomly generated) rules and facts, and respond in under 500 milliseconds to a range of queries requiring inference depths of up to eight.

In addition to developing viable technology for supporting large-scale knowledge base systems, this work provides some new insights into the simulation of structured connectionist networks on massively parallel machines and is a step toward developing large yet efficient knowledge representation and reasoning systems.

Section 2 is an overview of the system described in the rest of this report. Section 3 provides a brief description of SHRUTI, our representative structured connectionist knowledge representation and reasoning system. Section 4 is a general discussion of the issues involved in mapping SHRUTI onto massively parallel machines. Section 5 deals with the design, implementation and characteristics of the SPMD parallel rapid reasoning system on the CM-5. Similar issues for the SIMD CM-2 architecture are considered in Appendix A.

2 Overview of the System

The parallel rapid reasoning system supports the encoding of very large knowledge bases and their use for real-time inference and retrieval. Toward this end, the system includes the following suite of programs and

[1] Though the CM-5 is an MIMD architecture, it can only be used in SPMD (Single Program Multiple Data) mode with current software. See Section 5.

tools:

- A parser for accepting knowledge-base items expressed in a human readable input language. The language's syntax is similar to that of first-order logic (see Appendix D).
- A preprocessor for mapping a knowledge base onto the underlying parallel machine. This involves mapping the knowledge base to an inferential dependency network whose structure is analogous to that of SHRUTI, and partitioning this network among the processors of the parallel machine.
- A reasoning algorithm for answering queries. This runs on the parallel machine and efficiently mimics the reasoning process of our connectionist models.
- Procedures for collecting a number of statistics about the knowledge base and the reasoning process. These include the distribution of knowledge base items among processors, the processor load and message traffic during query answering, and a count of knowledge base items of each type (rules, facts, concepts, etc.) activated during processing.
- A utility for generating large psuedo-random knowledge bases given a specification of broad structural constraints. Examples of such constraints are: the number of knowledge base items of each type, any subdivision of the knowledge base into domains, the ratio of inter- and intra-domain rules, and the depth of the type hierarchy.
- Several tools for analyzing and visualizing the knowledge base and the statistics gathered during query answering.

This collection of programs and tools facilitates automatic loading of large knowledge bases, incremental addition of items to an existing knowledge base, posing of queries and recording of answers, and off-line visualization and analysis of system behavior. It also allows a user to construct large artificial knowledge bases for experimentation.

The system is interactive and allows the user to load and browse knowledge bases, and process queries by issuing commands at a prompt. At the same time it is also possible to process command files and use the system in an unattended batch processing mode.

3 SHRUTI—A Connectionist Reasoning System

SHRUTI, a connectionist reasoning system that can represent systematic knowledge involving n-ary predicates and variables, has been proposed by Shastri and Ajjanagadde (Shastri and Ajjanagadde, 1993; Ajjanagadde and Shastri, 1991). SHRUTI can perform a broad class of reasoning with extreme efficiency. In principle, the time taken by the reasoning system to draw an inference is only proportional to the *length* of the chain of inference and is independent of the number of rules and facts encoded by the system. The reasoning system maintains and propagates variable bindings using temporally synchronous—i.e., in-phase—firing of appropriate nodes. This allows the system to maintain and propagate a large number of variable bindings *simultaneously* as long as the number of *distinct* entities participating in the bindings during any given episode of reasoning remains bounded. Reasoning in the system is the transient but systematic flow of *rhythmic* patterns of activation, where each *phase* in the rhythmic pattern corresponds to a distinct *entity* involved in the reasoning process and where variable bindings are represented as the synchronous firing of appropriate role and filler nodes. A fact behaves as a temporal pattern matcher that becomes 'active' when it detects that the bindings corresponding to the fact are present in the system's pattern of activity. Finally, rules are interconnection patterns that propagate and transform rhythmic patterns of activity.

SHRUTI attempts to model reflexive reasoning over a large body of knowledge. SHRUTI has been extended in (Mani and Shastri, 1993) to effectively reason with a less restricted set of rules and facts and enhance the system's ability to model common-sense reflexive reasoning.

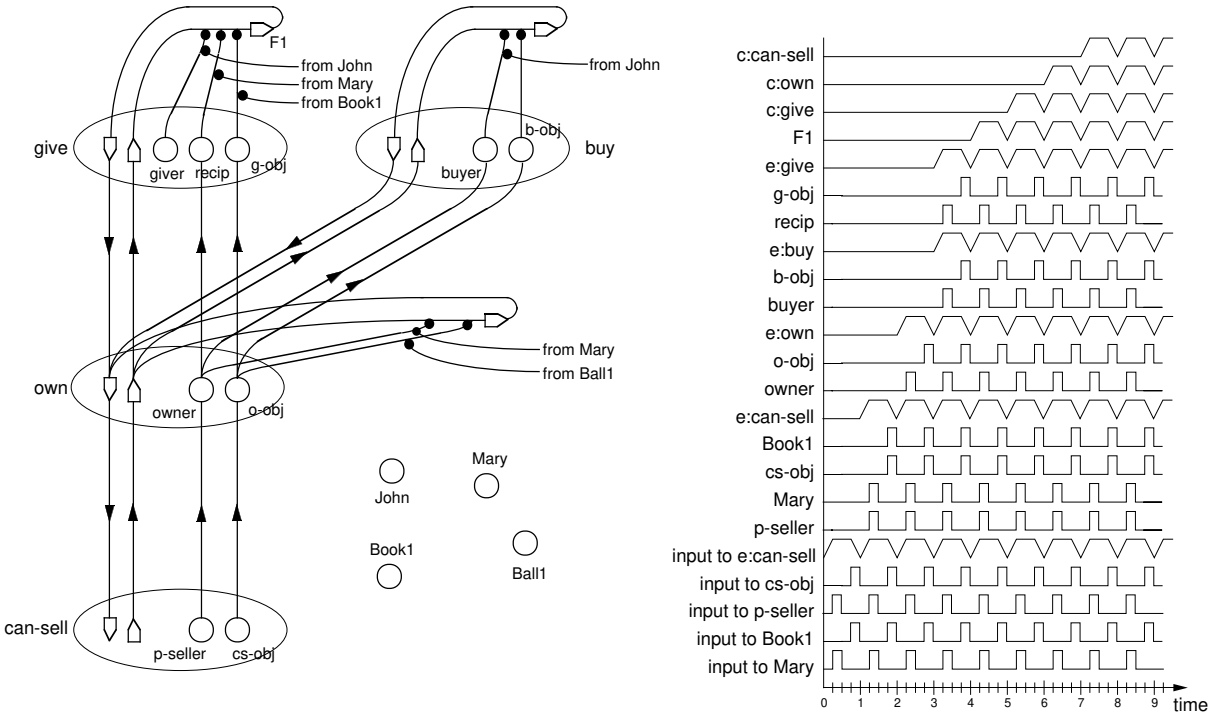

Figure 1: (**a**) An example encoding of rules and facts. (**b**) Activation trace for the query *can-sell(Mary,Book1)?*.

3.1 Terminology

We clarify some terminology before proceeding with a description of knowledge representation and reasoning in SHRUTI.

A *predicate* is a relation. For example, *give(x,y,z)* is a predicate which represents the relation: x gives y to z. Here x, y and z constitute the *arguments* or *roles* of the *give* predicate. A *fact* is a partially or completely instantiated predicate—like *give(John,Mary,Book1)*. Entities which are bound to predicate arguments are *fillers*. A *rule* specifies the systematic correspondence between predicate arguments. The rule $\forall x,y,z\ [\ give(x,y,z) \Rightarrow own(y,z)\ ]$ states that "if x gives y to z, then y owns z".

The term *entity* or *concept* is used to collectively refer to types (or categories) and instances (or individuals). An *is-a relation* or *is-a fact* captures the superconcept-subconcept relation between types, and the instance-of relation between types and instances.

Predicates, along with associated rules and facts, constitute the *rule-base* while concepts and their associated *is-a* relations constitute the *type-hierarchy*. Predicates, concepts, facts, rules and *is-a* relations together constitute *knowledge-base elements*.

3.2 Encoding Knowledge

We briefly describe the reasoning system using an example. Figure 1a illustrates how long-term knowledge is encoded in the rule-based reasoning system. The network encodes the following *rules* and *facts*:

$\forall x,y,z\ [\ give(x,y,z) \Rightarrow own(y,z)\ ]$,
$\forall x,y\ [\ buy(x,y) \Rightarrow own(x,y)\ ]$,
$\forall x,y\ [\ own(x,y) \Rightarrow can\text{-}sell(x,y)\ ]$,

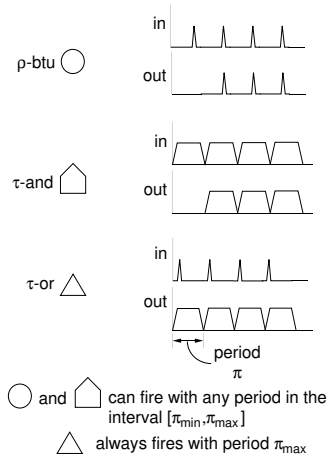

Figure 2: Behavior of ρ-btu, τ-and and τ-or nodes in the reasoning system.

give(John,Mary,Book1),
buy(John,x), and
own(Mary,Ball1).

Rule and fact encoding makes use of several types of nodes (see Figure 2): **ρ-btu** nodes (depicted as circles), **τ-and** nodes (depicted as pentagons) and **τ-or** nodes (depicted as triangles). These nodes have the following idealized behavior: On receiving a spike train, a ρ-btu node produces a spike train that is *synchronous* (i.e., *in-phase*) with the driving input. We assume that ρ-btu nodes can respond in this manner as long as the inter-spike distance, π, lies in the interval $[\pi_{min}, \pi_{max}]$. Here π_{min} and π_{max} are the minimum and maximum inter-spike gaps for which the system can sustain synchronous activity (Shastri and Ajjanagadde, 1993). A τ-and node behaves like a *temporal* AND node, and becomes active on receiving an uninterrupted pulse train. On becoming active, a τ-and node produces a pulse train comparable to the input pulse train. A τ-or node on the other hand becomes active on receiving *any* activation; its output is a pulse whose width and period equal π_{max}. Figure 2 summarizes node behavior. The encoding also makes use of *inhibitory modifiers*—links that impinge upon and inhibit other links. A pulse propagating along an inhibitory modifier will block a pulse propagating along the link it impinges upon. In Figure 1a, inhibitory modifiers are shown as links ending in dark blobs.

Each entity in the domain is encoded by a ρ-btu node. An n-ary predicate P is encoded by a pair of τ-and nodes and n ρ-btu nodes, one for each of the n arguments. One of the τ-and nodes is referred to as the *enabler*, *e:P*, and the other as the *collector*, *c:P*. In Figure 1a, *enablers* point upward while *collectors* point downward. The *enabler e:P* becomes active whenever the system is being queried about P. On the other hand, the system activates the *collector c:P* of a predicate P whenever the system wants to assert that the current dynamic bindings of the arguments of P follow from the knowledge encoded in the system. A rule is encoded by connecting the *collector* of the antecedent predicate to the *collector* of the consequent predicate, the *enabler* of the consequent predicate to the *enabler* of the antecedent predicate, and the arguments of the consequent predicate to the arguments of the antecedent predicate in accordance with the correspondence between these arguments specified in the rule. A fact is encoded using a τ-and node that receives an input from the enabler of the associated predicate. This input is modified by inhibitory modifiers from the argument nodes of the associated predicate. If an argument is bound to an entity in the fact then the modifier from such an argument node is in turn modified by an inhibitory modifier from the appropriate entity node. The output of the τ-and node is connected to the *collector* of the associated predicate. Figure 1a shows the encoding of the facts *give(John,Mary,Book1)* and *buy(John,x)*. The fact *give(John,Mary,Book1)* states that 'John gave Mary Book1' while *buy(John,x)* implies that 'John bought *something*'.

3.3 The Inference Process

Posing a query to the system involves specifying the query predicate and its argument bindings. The query predicate is specified by activating its *enabler* with a pulse train of width and periodicity π. Argument bindings are specified by activating each entity, and the argument nodes bound to that entity, in a distinct *phase*, phases being non-overlapping time intervals within a period of oscillation.

We illustrate the reasoning process with the help of an example. Consider the query *can-sell(Mary, Book1)?* (i.e., Can Mary sell Book1?) The query is posed by (i) Activating the *enabler e:can-sell*; (ii) Activating *Mary* and *p-seller* in the same phase (say, ρ_1), and (iii) Activating *Book1* and *cs-obj* in some other phase (say, ρ_2). As a result of these inputs, *Mary* and *p-seller* fire synchronously in phase ρ_1 of every period of oscillation, while *Book1* and *cs-obj* fire synchronously in phase ρ_2. See Figure 1b. The activation from the *can-sell* predicate propagates to the *own*, *give* and *buy* predicates via the links encoding the rules. Eventually, as shown in Figure 1b, *Mary*, *p-seller*, *owner*, *buyer* and *recip* will all be active in phase ρ_1, while *Book1*, *cs-obj*, *o-obj*, *g-obj* and *b-obj* would be active in phase ρ_2. The activation of *e:can-sell* causes the enablers of all other predicates to go active. In effect, the system is asking itself three more queries—*own(Mary,Book1)?*, *give(x,Mary,Book1)?* (i.e., Did *someone* give Mary Book1?), and *buy(Mary,Book1)?*. The τ-and node F1, associated with the fact *give(John,Mary,Book1)* becomes active as a result of the uninterrupted activation it receives from *e:give*, thereby answering *give(x,Mary,Book1)?* affirmatively. The activation from F1 spreads downward to *c:give*, *c:own* and *c:can-sell*. Activation of *c:can-sell* signals an affirmative answer to the original query *can-sell(Mary,Book1)?*.

3.4 The Type Hierarchy

Integrating a type hierarchy with the reasoning system (Mani and Shastri, 1993) allows the use of types (categories) as well as instances in rules, facts, and queries. This has the following consequences:

- The reasoning system can combine rule-based reasoning with *inheritance* and *classification*. For example, such a system can infer that 'Tweety is scared of Sylvester', based on the generic fact 'cats prey on birds', the rule 'if x preys on y then y is scared of x' and the *is-a* relations 'Sylvester is a cat' and 'Tweety is a bird'.
- The integrated system can use category information to *qualify* rules by specifying restrictions on the type of argument fillers. An example of such a rule is:

  $$\forall x{:}animate,\ y{:}solid\text{-}obj\ [\ walk\text{-}into(x,y) \Rightarrow hurt(x)\ ]$$

  which specifies that the rule is applicable only if the two arguments of 'walk-into' are of the type 'animate' and 'solid-object', respectively.

Each entity is now represented as a cluster of nodes and is associated with two *type-hierarchy switches*—a *top-down* T-switch and a *bottom-up* T-switch. Any entity can now accommodate up to k_1 dynamic instantiations, k_1 being the multiple instantiation constant for concepts. The T-switches regulate the flow of activation to bring about efficient and automatic dynamic allocation of concept banks to ensure that:

- Any concept represents at most k_1 instantiations.
- A given instantiation is represented at most once; in other words, no two banks represent the same instantiation.

3.5 Multiple Dynamic Instantiation of Predicates

Extending the reasoning system to incorporate multiple instantiation of predicates (Mani and Shastri, 1993) provides SHRUTI with the ability to *simultaneously* represent multiple dynamic facts involving a predicate.

For example, the dynamic facts *loves(John,Mary)* and *loves(Mary,Tom)* can now be represented *at the same time*. As a result, we can represent and reason using a set of rules which cause a predicate to be instantiated more than once. We can now encode rules like:

$\forall x, y$ [*sibling(x,y)* $\Rightarrow$ *sibling(y,x)*] and
$\forall x, y, z$ [*greater-than(x,y)* $\wedge$ *greater-than(y,z)* $\Rightarrow$ *greater-than(x,z)*]

thereby introducing the capability to handle bounded symmetry, transitivity and recursion.

Introduction of multiple dynamic instantiation of predicates relies on the assumption that, during an episode of reflexive reasoning, any given predicate need only be instantiated a *bounded* number of times. In (Shastri and Ajjanagadde, 1993), it is argued that a reasonable value for this bound is around three. We shall refer to this bound as the multiple instantiation constant for predicates, k_2.[2]

Predicate representations are augmented so that each predicate can represent up to k_2 dynamic instantiations. Each predicate also has an associated *multiple instantiation switch* (or M-switch) through which all inputs to the predicate nodes are routed. The switch arbitrates inputs and brings about efficient and automatic dynamic allocation of predicate banks to ensure that predicates represent at most k_2 instantiations, no two of which are identical.

4 Mapping SHRUTI onto Massively Parallel Machines

When mapping SHRUTI onto any massively parallel machine, several issues need to be considered in order to obtain effective performance and to strike a compromise between resource usage and response time. Several of these issues are discussed here. The discussion here is applicable when mapping SHRUTI onto any massively parallel machine. Later sections bring out how these issues are resolved in actual implementations on the CM-2 and CM-5. The CM-2 is an SIMD machine while the CM-5 is an MIMD machine. We have chosen the CM-2 and CM-5 as our target machines since they are representatives of their class and offer similar user interfaces and program development environments.

4.1 Exploiting Constraints Imposed by SHRUTI

As brought out in the previous sections, SHRUTI is a *limited* inference system, and imposes several psychologically and/or biologically motivated constraints in order to make reasoning tractable:

- The form of rules and facts that can be encoded is constrained. SHRUTI attains its tractability from this fundamental constraint (Shastri, 1993; Dietz et al., 1993), which implicitly influences the resulting network encoding the knowledge base.
- The number of distinct entities that can participate in an episode of reasoning is bounded. This restricts the number of active entities and hence the amount of information contained in an instantiation.
- Entities and predicates can only represent a limited number of dynamic instantiations. Entities and predicates therefore have a bounded number of banks which constrains both the space and time requirements.
- The depth of inference is bounded. This constrains the spread of activation in the network and therefore directly affects time and resource usage.

The motivation for these constraints and their impact are discussed in (Shastri and Ajjanagadde, 1993). In terms of mapping SHRUTI onto parallel machines, it would be to our advantage to exploit these constraints

[2]This is the factor that limits symmetry, transitivity and recursion, since each predicate can accommodate at most k_2 dynamic instantiations.

to the fullest extent to achieve efficient resource usage and rapid response with large knowledge bases. Of course, if any of these constraints can be relaxed without paying a severe performance penalty, we would like to obtain a more powerful system by relaxing these constraints.

4.2 Granularity

For effective mapping, the SHRUTI network encoding a knowledge base must be partitioned among the processors in the machine. The network partitioning can be specified at different levels of granularity. At the fine-grained *network-level*, the partitioning would be at the level of the basic nodes and links constituting the network. A more coarse-grained *knowledge-level* mapping would partition the network at the level of knowledge elements like predicates, concepts, facts, rules and *is-a* relations.

The appropriate level of granularity for a given situation depends on several factors including the characteristics of the network, the processing power of individual processors on the machine and interprocessor communication mechanisms.

4.3 Network-Level Mapping

At this level of granularity, the network is viewed as a collection of *nodes* and *links*. Factors that need to be considered when using network-level partitioning include:

Processor Allocation Nodes and links in the network should be assigned to processors on the target machine so as to minimize response time. Several options are possible: Each node and link could be assigned to a separate processor; groups of nodes and/or links could be assigned to a single processor; processors could be partitioned so that some handle only nodes and some handle only links; and so on.

Nodes The network which SHRUTI uses to encode a knowledge base consists of several different types of nodes. A given processor could handle only one type of node or could simulate an assorted combination of node types. The complexity of the node function should also be taken into consideration.

Links Like nodes, the links can also be of several types—including weighted, unweighted and inhibitory links. Placement of the links (on processors) relative to the placement of the nodes they connect is important since this is a major factor determining the volume of interprocessor communication.

Communication and Computation The partitioning scheme used to assign network components to processing elements should take into account the balance of computation and communication in the resulting system. Communication between network nodes, and hence interprocessor communication, is an essential aspect of connectionist network simulation. Trying to eliminate or unduly minimize interprocessor communication could lead to severe load imbalances whereby a few of the processing elements are overburdened with computation. Trying to evenly spread the computational load among the processing elements could result in increased communication and poor performance. A well designed system should strike a compromise between communication and computation so as to achieve effective performance.

4.4 Knowledge-Level Mapping

Knowledge-level mapping views the network at a relatively abstract level. At this granularity, knowledge base elements like predicates, concepts, facts, rules and *is-a* relations form the primitives. As is evident from Section 3, each primitive is constituted by a group of nodes and/or links. The behavior of these primitives could be directly simulated without recourse to the underlying nodes and links constituting the primitive. Issues at this level include:

Predicates Each predicate could be assigned to a separate processor, or a group of predicates could be assigned to a single processor. In the latter case, predicates constituting a rule could all be placed on the same processor or could be scattered on different processors. Grouping predicates on any given processor could reduce the number of messages required to spread activation, but would make load balancing more difficult. If the number of predicates is larger than the number of processors, grouping predicates is unavoidable.

Facts Facts could be stored on the same processors to which the corresponding predicates have been assigned. An alternative approach would be to have dedicated processors for encoding facts. Such processors receive inputs from both the predicate and the type hierarchy, and signal fact matches globally or by communicating with the processor containing the predicate under consideration. In any case, we may need some mechanism to circumvent the situation where processors run out of memory since predicates could have a large number of associated facts.

Concepts Concept clusters are used in the type hierarchy to represent types and instances. Apart from being linked up to form the type hierarchy, these clusters must also communicate with the rule base. Careful choice of the mechanisms used to communicate concept activations to the rule base could make the system more effective and reduce the number of messages exchanged in the system.

Rules When encoding rules, effective placement of predicates constituting the rule can minimize communication costs. The arbitration mechanism for accommodating multiple instantiations of a predicate also needs to be taken into account.

- When encoding rules, there are several choices available for the placement of predicates constituting the rule:
  - Depending on the processor allocation scheme used, we could allocate predicates occurring in a rule to the same processor. This would reduce interprocessor communication since fewer messages are required when the rule fires. This may not be easy to accomplish if predicates present in the rule being encoded have already been assigned to different processors.
  - A weaker form of the above scheme is to allocate predicates in a rule on *nearby* processors. This scheme is easier to execute but will require relatively more messages in order to fire a rule.
  - The other extreme is to scatter the predicates randomly. Though this would require more messages, and messages would travel a longer average distance than for the previous two schemes, there are indications that random allocation may distribute messages uniformly over the entire machine instead of localizing it to "hot spots" where all the action happens, and would therefore reduce the incidence of message collisions. Further, this scheme would provide better load balancing when answering a query.
  - Making copies of a given predicate on more than one processor is also an option, especially when the predicate has a large number of rules and/or facts. In such a case, the rules and/or facts would be partitioned among the copies of the predicate. Though this requires extra resources and complicates book-keeping, it might be worthwhile since it could provide increased parallelism and improved load balancing.
- Identifying suitable performance measures and attempting to optimize these will aid in the objective placement of predicates when encoding rules. The performance measure could take into account factors like load balancing, cost of computation and communication, etc. It should be easy to compute the measure—or at least approximate it—using only local information.
- Predicate instance arbitration mechanisms ("switches") may need to be redesigned. When one or more predicates are assigned to each processor, switches may be unnecessary. Space ("banks") can be allocated for k_2 instances of each predicate. Incoming activation can be received in a buffer and then allocated to an empty bank under program control.

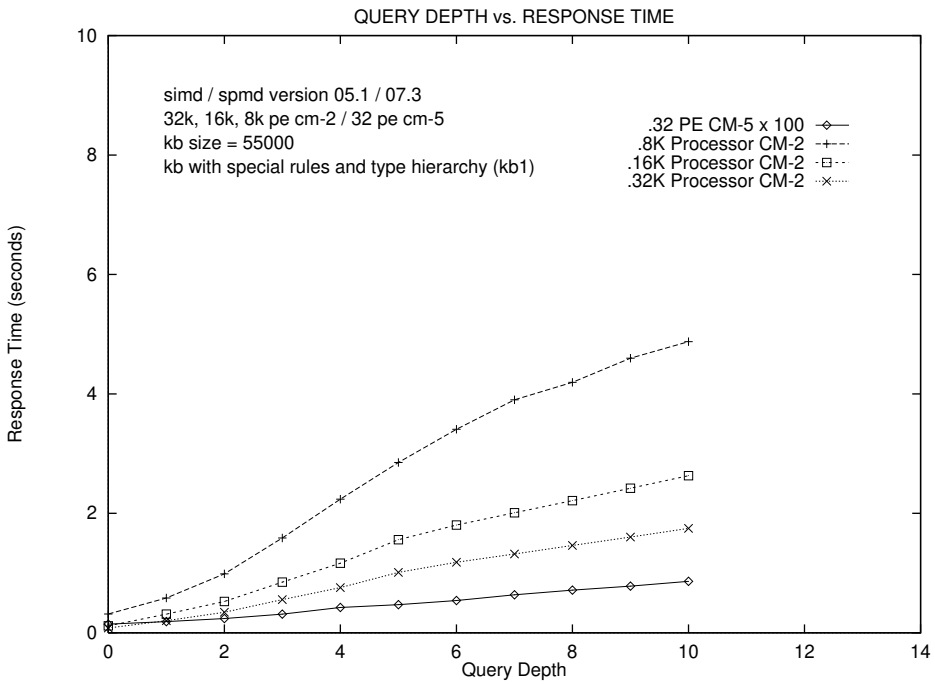

Figure 3: A comparison of SHRUTI-CM2 running on 32K, 16K and 8K processor CM-2 machines and SHRUTI-CM5 running on a 32 PE CM-5. The same full-fledged, structured, random knowledge base with special rules and a type hierarchy was used on all the machines. Note that the timing curve for the CM-5 has been multiplied by 100. Queries used were not randomly generated.

Type Hierarchy Most of the issues raised above will also need to be reconsidered with respect to the location and interaction of concepts in the type hierarchy. We would also need to streamline the interaction between the type hierarchy and the rule base for enhanced efficiency and effectiveness. Extending the scheme mentioned above for dealing with multiple instantiation, we might be able to do away with the type hierarchy T-switch.

Most of the concerns addressed above are intertwined in that choosing one aspect will affect the choice of other aspects of the mapping. On a global scale, our aim is to develop an efficient and effective mapping by ensuring load balancing, minimizing interprocessor communication and by efficiently using resources including processors and memory.

We believe that knowledge-level partitioning is the appropriate granularity for both the CM-2 and CM-5. The processing elements on the CM-2 are reasonably powerful (Appendix A) while the processing elements on the CM-5 (Section 5) are full-fledged SPARC processors. Thus, subnetworks corresponding to knowledge-level primitives can be implemented using appropriate data structures and associated procedures without necessarily mimicking the detailed behavior of individual nodes and links in the subnetwork.

5 SHRUTI on the CM-5

In this section we describe the design and implementation of the SPMD asynchronous message passing parallel rapid reasoning system—SHRUTI-CM5—that has been developed for the CM-5.

5.1 SHRUTI on the CM-2

Initially we developed SHRUTI-CM2, a data parallel implementation of SHRUTI on the Connection Machine CM-2 (TMC, 1991a). A detailed description of SHRUTI-CM2, including design, knowledge encoding, spreading activation and performance characteristics can be found in Appendix A. However, due to the overwhelmingly superior performance of SHRUTI-CM5—the SPMD implementation on the CM-5—the SHRUTI-CM2 project was abandoned. Figure 3 compares the performance of SHRUTI-CM2 and SHRUTI-CM5. SHRUTI-CM2 can also be run on the CM-5 in data-parallel mode. Results of these experiments are described in Appendix B.

5.2 The Connection Machine CM-5

The Connection Machine model CM-5 (TMC, 1991b) is an MIMD machine consisting of anywhere from 32 to 1024 powerful processors.[3] Each processing node is a general-purpose computer which can execute instructions autonomously and perform interprocessor communication. Each processor can have up to 32 megabytes of local memory[4] and optional vector processing hardware. The processors constitute the leaves of a *fat tree* interconnection network, where the bandwidth increases as one approaches the root of the tree. Every CM-5 system has one or more control processors which are similar to the processing nodes but are specialized to perform managerial and diagnostic functions. A low-latency control network provides tightly coupled communications including synchronization, broadcasting, global reduction and scan operations. A high bandwidth data network provides loosely coupled interprocessor communication. A standard network interface connects nodes and I/O units to the control and data networks. The virtual machine emerging from a combination of the hardware and operating system consists of a control processor acting as a partition manager, a set of processing nodes, facilities for interprocessor communication and a UNIX-like programming interface. A typical user task consists of a process running on the partition manager and a process running on each of the processing nodes.

Though the basic architecture of the CM-5 supports MIMD style programming, operating system and other software constraints restrict users to SPMD (Single Program Multiple Data) style programs (TMC, 1994). In SPMD operation, a single program runs on all the processors, each acting on its share of data items. Both data parallel (SIMD) and message-passing programming on the CM-5 use the SPMD model. If the user program takes a primarily global view of the system—with a global address space and a single thread of control—and processors run in synchrony, the operation is data parallel; if the program enforces a local, node-level view of the system and processors function asynchronously, the machine is used in a more MIMD fashion. We shall consistently use "SPMD" to be synonymous with the latter mode of operation. In this mode, all communication, synchronization and data layout are under the programs' explicit control.

5.3 Design Considerations

Granularity of Mapping

The individual processing elements on the CM-5 are powerful processors and therefore a subnetwork in the connectionist model can be implemented on a processor using appropriate data structures and associated

[3] In principle, the CM-5 architecture can support up to 16K processors.

[4] The amount of local memory is based on 4-Mbit DRAM technology and will increase as DRAM densities increase.

procedures without necessarily mimicking the detailed behavior of individual nodes and links in the subnetwork. This suggests that knowledge-level partitioning (Section 4) is the appropriate granularity for mapping SHRUTI onto the CM-5.

Representing Synchrony

SHRUTI-CM5 represents temporal synchrony by using "markers"—integers with values ranging from 1 to the maximum number of phases. Though temporal synchrony can be simulated on the CM-5 by using repeated processor synchronization, we have opted against this approach since unnecessary processor synchronization can slow down the system. Moreover, the use of markers makes SHRUTI-CM5 flexible so that it can be adapted to support other related marker-passing systems.

Temporal synchrony is one of the most distinguishing features of SHRUTI. In spite of not explicitly using temporal synchrony, SHRUTI-CM5 retains its SHRUTI-like flavor by exploiting the characteristics and constraints derived from SHRUTI's temporal synchrony approach to reasoning.

Active Messages and Communication

SHRUTI-CM5 uses CMMD library functions (TMC, 1993) for broadcasting and synchronization, while almost all interprocessor communication is achieved using CMAML (CM Active Message Library) routines.

CMAML provides efficient, low-latency interprocessor communication for short messages (TMC, 1993; von Eicken et al., 1992). Active messages are asynchronous (non-blocking) and have very low communication overhead. A processor can send off an active message and continue processing without having to wait for the message to be delivered to its destination. When the message arrives at the destination, a handler procedure is automatically invoked to process the message. The use of active messages improves communication performance by about an order of magnitude compared with the usual send/receive protocol. The main restriction on such messages is their size—they can only carry 16 bytes of information. However, given the constraints on the number of entities involved in dynamic bindings (≈ 10), there is an excellent match between the size of an active message and the amount of variable binding information that needs to be communicated between predicate instances during reasoning as specified by SHRUTI. SHRUTI-CM5 exploits this match to the fullest extent.

5.4 Encoding the Knowledge Base

In the SHRUTI-CM5 system, the knowledge base is encoded by presenting rules and facts expressed in a human readable, first-order logic-like syntax specified in Appendix D. The commands recognized by SHRUTI-CM5 are described in Appendix E.

Input Processing

Knowledge encoding in SHRUTI-CM5 is a two-part process:

1. **Serial preprocessing.** A serial preprocessor running on a workstation processes the input knowledge base and partitions it into as many chunks as there are processors on the CM-5 partition. The preprocessor outputs a set of files—one file for each processor—which are subsequently read by the respective CM-5 processors.

2. **Parallel knowledge base encoding.** Each processor on the CM-5 independently and asynchronously reads and encodes the fragment of the knowledge structure assigned to it by the preprocessor. Depending on the processor assignment scheme used, each processor on a n processor CM-5 would typically need to process only $\frac{1}{n}$-th of the entire input knowledge base.

```
typedef struct cm_predbank          /* predicate bank on the CM */
{
  /* no fields used to encode KB */

  byte   collector;
  byte   enabler;
  byte   args[MAX_ARGS];            /* arg activation phase */
  char   qDepth;                    /* depth of reasoning chain
                                       which makes c: active */
} CM_PredBank;

typedef struct cm_pred              /* predicate on the CM */
{
  byte               noOfArgs;
  struct cm_list     *rules;        /* list of rules with pred as conseq */
  struct cm_list     *facts;        /* list of facts for pred */

  byte               nextFree;      /* index of next free bank (minst) */
  struct cm_predbank bank[K2];      /* predicate banks */
  struct cm_list     *ruleBPtr[K2]; /* rule back-pointers (for c: activation) */
} CM_Pred;
```

Figure 4: C structures used to represent predicates in SHRUTI-CM5. `MAX-ARGS` is the maximum number of arguments a predicate can have. `K2` is the multiple instantiation constant for predicates. The top part of the `typedef`s contain fields used to encode the knowledge base while the bottom part has fields used in a given episode of reasoning.

This two-part, asynchronous parallel input processing is well suited for large-scale knowledge bases. In addition, SHRUTI-CM5 also provides a direct input mode. In this mode, all processors synchronously read the same input and cooperatively decide—based on the processor allocation scheme—on who encodes the current knowledge base item. This mode can be used to by-pass serial preprocessing and is useful when small knowledge base fragments need to be added to an existing (large) knowledge base. SHRUTI-CM5 also supports convenient and consistent parallel updating of large knowledge bases via incremental preprocessing.

In either of the input modes, the knowledge base is scanned by a lexical analyzer and parser, resulting in the construction of internal data structures. Once a rule, fact or *is-a* relation has been recognized and processed, these internal data structures will be used to encode the knowledge base element on the Connection Machine processors. In the case of a query, the data structures will be used to pose the query to the system.

A specially designated *server* processor builds hash tables which keep track of processor assignments. Whenever the system needs to know which processor houses some predicate P, the server broadcasts the required information. The system is designed in such a manner that the server does not become a bottleneck during the reasoning process. Information from the server is needed only when posing a query.[5] Once a query has been posed, the system data structures are so configured that spreading activation will proceed without the need for any information from the server. Maintaining a server processor therefore does not affect inference timing in any way.

[5] The server is also accessed when encoding knowledge in synchronous direct input mode.

```
typedef struct cm_rule                  /* rule slot on the CM */
{
  /* knowledge base encoding */
  struct cm_antlist  *antecedent;       /* list of antecedent predicates */
  struct cm_list     *consequent;       /* consequent predicate */
  byte               noOfAnts;          /* number of ant predicates for rule */
  int                weight;            /* weight; currently unused */
  byte               splCond[MAX_ARGS]; /* list of special conditions */
  int                splIndex[MAX_ARGS];/* procs containing spl cond constants */
  index              splPtr[MAX_ARGS];  /* ptr to spl cond constants */

  /* reasoning episode */
  byte   conseqCollector[K2];     /* c: values for the conseq pred are
                                     accumulated here; reqd for supporting
                                     multiple antecedent rules */
  char   qDepth[K2];              /* reasoning chain depth; reqd for multiple
                                     antecedent rules */
} CM_Rule;

typedef struct cm_fact                      /* fact on the CM */
{
  struct cm_pred   *factPred;               /* fact predicate */
  index            constant[MAX_ARGS];      /* fact argument pointers */
  index            constLocation[MAX_ARGS]; /* proc containing const */

  bool             active;                  /* fact active if set */
} CM_Fact;
```

Figure 5: C structures used to encode rules and facts in SHRUTI-CM5. `MAX-ARGS` is the maximum number of arguments a predicate can have. `K2` is the multiple instantiation constant for predicates. Processor indices have type `index` and flags have type `bool`. Pointers are also of type `index` and index into local translation tables on the respective processors. The top part of the `typedef`s contain fields used to encode the knowledge base while the bottom part has fields used in a given episode of reasoning.

```
typedef struct cm_entitybank        /* entity bank on the CM */
{
  /* no fields used to encode KB */

  bool   buRelay;                   /* bottom-up relay */
  bool   tdRelay;                   /* top-down relay */
  byte   activation;                /* entity activation phase */
} CM_EntityBank;


typedef struct cm_entity                /* entity on the CM */
{
  struct cm_list       *superConcepts;  /* bottom-up links */
  struct cm_list       *subConcepts;    /* top-down links */

  byte                 nextFree;        /* index of next free bank */
  struct cm_entitybank bank[K1];        /* entity banks */
} CM_Entity;
```

Figure 6: C structures used to represent entities in the type hierarchy (in SHRUTI-CM5). `K1` is the multiple instantiation constant for concepts in the type hierarchy. Flags have type `bool`. The top part of the `typedef`s contain fields used to encode the knowledge base while the bottom part has fields used in a given episode of reasoning.

```
typedef struct cm_isalink       /* is-a links on the CM */
{
  index            destination;  /* index of destination proc */
  index            concept;      /* destination concept */

  /* no fields used during reasoning episode */
} CM_isALink;
```

Figure 7: C structure used to encode *is-a* relationships in SHRUTI-CM5. Processor indices have type `index`. Pointers are also of type `index` and index into local translation tables on the respective processors. The top part of the `typedef` contains fields used to encode the knowledge base while the bottom part has fields used in a given episode of reasoning.

Representing Knowledge Base Elements

Each knowledge base element (Section 3.1) is assigned to a single processor and represented using suitable structures, defined in Figures 4–7. All processors in the partition except the server can encode knowledge base elements. The SHRUTI network is internally encoded by a series of pointers which serve to link predicate and concept representations. Unlike a serial machine, a "pointer" on the CM-5 would need both a memory address and the index of the processor to which the required fragment of memory belongs. In order to support parallel knowledge base encoding, the "memory addresses" are indirect and index into translation tables on the respective processors.

Encoding Rules and Facts

Depending on the processor allocation scheme (Section 4), every predicate and concept appearing in the knowledge base will be assigned to a processing node on the CM-5. Further, a rule, fact or *is-a* relation that is being encoded will also be assigned to a processor. The actual details of the processor allocation are dictated by the processor assignment scheme being used. The SHRUTI-CM5 design offers several options for processor assignment schemes. SHRUTI-CM5 implementations use random processor assignment for predicates and concepts. Facts and *is-a* links are encoded on the processors containing the relevant predicate or concept[6] and rules were encoded on the processor containing the consequent predicate. Any processor in the machine (except the server) can have both predicates and concepts assigned to it.

Once the predicates, concepts and other knowledge base elements under consideration are assigned to processing elements on the CM-5, the knowledge base structures are built and/or updated. Rules, facts, and *is-a* links are encoded by a series of pointers which link predicate and concept representations to form the entire network.

5.5 Spreading Activation and Inference

Queries can be posed after the knowledge base has been encoded. Queries result in the activation of the relevant predicate and concepts as described in (Shastri and Ajjanagadde, 1993) and (Mani and Shastri, 1993). The activation propagation loop is shown in Figure 8. As noted in Section 5.3, markers are used to represent SHRUTI phases.

The system runs asynchronously in that each processor continues with its processing irrespective of the progress made by other processors. If an answer to the query is found, the reasoning episode terminates immediately. If no answer is found after a certain number of asynchronous iterations, all processors synchronize and iterate synchronously. This synchronization ensures that activation has had a chance to traverse the depth of the network and is a safeguard against unlikely, but possible, cases of pathological imbalances in computation and interprocessor communication load. If no answer is found even after a fixed number of synchronous propagation steps, the reasoning episode terminates without an answer. This termination criteria is in keeping with the constraint that reflexive reasoning can only occur up to a bounded depth. The user can experiment with the terminating criteria by setting the number of asynchronous and synchronous iterations at compile time.

Each processing node maintains several activation "frontiers" for both the rule base and the type hierarchy. Each frontier is essentially a list of predicates or concepts that are active and which need to be considered in the current activation propagation step. The following frontiers are maintained: A rule-frontier consisting of consequent predicates of rules under consideration in the current step; A fact-frontier consisting of predicates for which fact matches need to be checked; A reverse-propagation-frontier for handling

[6] Assigning facts (*is-a* links) to the processor housing the associated predicate (concept) could result in deteriorating performance if the distribution of facts (*is-a* relations) is skewed—i.e., a few predicates (concepts) have a disproportionately large number of facts (*is-a* relations). Under such situations, other schemes such as splitting facts (*is-a* links) across processors may have to be considered.

```
initialize global statistics collection variables;

while (termination condition not met) {
  /* propagate activation in the type hierarchy */
  spread bottom-up activation;
  spread top-down activation;

  /* propagate activation in the rule base */
  reverse-propagate collector activation;
  check fact matches;
  propagate enabler activation by rule-firing;

  update statistics collection variables;
}
```

Figure 8: The main propagation loop used in spreading activation during an episode of reasoning. The termination condition is met when the query is answered or the system determines that the query has no answer. Note that the order of the operations is crucial while propagating rule base activation. Activation of predicates whose collectors became active in the *previous* step must be reverse-propagated before facts are matched, since fact matching could activate other predicate collectors whose activation should be spread in the *next* propagation step. Further, fact matching for predicates that became active in the previous step must occur before new rules are fired, since firing rules could activate more predicates and fact matches for these predicates should be checked in the next iteration.

reverse-propagation of collector activation; and a type-hierarchy-frontier for activation propagation in the type hierarchy. During each propagation step, all frontiers are consistently updated in preparation for the next step in the iteration. Frontier elements are deleted after performing the required operation. A frontier element will reappear in the frontier for the *next* propagation step only if the operation attempted in the current step was unsuccessful. This ensures that the same operation—like firing a specific rule, matching a fact or firing an *is-a* fact—is not unnecessarily repeated. All frontiers are created and deleted asynchronously on *each* processor.

During an episode of reasoning, all interprocessor communication—including firing rules, spreading activation in the type hierarchy and reverse-propagating collector activation—is effected using active messages supported by the CMAML routines. The system has been tailored so that any information that needs to be exchanged between two processors will always fit in a single active message.

In each activation propagation step, every processor scans its frontiers and takes appropriate action: firing rules for predicates on the rule-frontier; propagating activation in the type hierarchy for concepts in the type-hierarchy-frontier; propagating collector activation for predicates in the reverse-propagation-frontier; and matching facts for predicates on the fact-frontier. Processors send out active messages if the predicates or concepts that need to receive activation are located on another processor. When these active messages arrive at their destinations, they invoke handler functions which receive and process the incoming activation, and update the relevant frontiers. In the asynchronous phase, each processing node operates independently of the others.

Type Hierarchy and Multiple Instantiation

The type hierarchy is handled in a manner that is essentially similar to the rule base. Spreading bottom-up and top-down activation is separate and sequential. As entities go active, they broadcast their activations to all the processors in the partition. The processors cache this information for fast, local access during fact

matching and special condition checking. In order to handle multiple instantiation (also see Appendix C), whenever a predicate or concept receives activation, it is compared with existing activation in the banks. If the incoming activation is not already represented, it is then deposited into the next available bank. The predicate representing the instantiation keeps track of the source of the instantiation in order to reverse-propagate collector activation. An instantiation will need to be identified using (i) the processor housing the predicate or concept; (ii) the predicate or concept that originated the instantiation and (iii) the bank under consideration. Enough information is maintained when an instantiation is received so that collector activation can be propagated back to the predicate bank which originated the activation. Note that multiple instantiation is handled without the use of switches (Mani and Shastri, 1993); the above protocol is functionally equivalent to these switches and ensures that (i) any predicate or concept represents at most a bounded number of instantiations (the number being decided by the multiple instantiation constants `K1` and `K2`) and (ii) a given instantiation is represented at most once so that no two banks of a predicate or concept represent the same instantiation.

Statistics Collection

SHRUTI-CM5 can be configured to collect statistics about various aspects of the system like knowledge base parameters, processor communication and computation, and the reasoning process. These include the distribution of knowledge base items among processors, the processor load and message traffic during query answering, and a count of knowledge base items of each type (rules, facts, concepts, etc.) activated during processing. Full-fledged data collection can slow down the system due to the extra time needed to accumulate required data.

5.6 Characteristics of SHRUTI-CM5

SHRUTI-CM5 has been tested using artificial knowledge bases containing up to several hundred thousand rules and facts. Most of the experimentation has been carried out on a 32 node machine.

Figures 9—14 illustrate the performance, timing and resource usage of SHRUTI-CM5. Figure 9 plots response time for varying query depths and knowledge base sizes. Figure 10 shows the number of rules fired when answering the respective queries. In both these figures, the queries used were generated randomly, and the values shown are averages for a given knowledge base size and query depth. About 100 queries with depths ranging from 0 to 8 were used; some of the queries were answered while several were not. The graphs depict the average for queries that were answered. The number of queries contributing to each data point ranges from about 15 (for depth 0) to 1 (for maximum depth). As the number of queries averaged over increases, we expect the curves to get smoother and statistically more reliable.

Figure 11 shows the average time needed to fire a rule as a function of knowledge base size and query depth. When a reasonably large number of rules fire in a given reasoning episode, the time needed per rule firing settles to a small, relatively constant value. Due to random queries being posed to a random knowledge base, there is lot of variation in the response time and other performance statistics for a given knowledge base size and query depth. Among all this variation, the behavior of the "time-per-rule" metric seems to be consistent over a variety of knowledge bases. We however do not know whether the "time-per-rule" metric will remain constant if the knowledge bases are significantly larger than the ones we have experimented with.

Figure 12 shows the distribution of a knowledge base with approximately 300,000 elements among the CM-5 processors. It is easily seen that the distribution is very even as a result of random processor allocation. Finally, Figures 13 and 14 show the computation and communication load on each processor for a 300,000 element knowledge base and a query of depth 8. Computation load is measured as the number of active predicates, entities and facts on each processor, while communication load is the number of active messages sent out by each processor. In spite of the unpredictable nature of the activation trail in the knowledge base, communication and computation load are relatively well balanced. Processor load is reasonably balanced irrespective of the query.

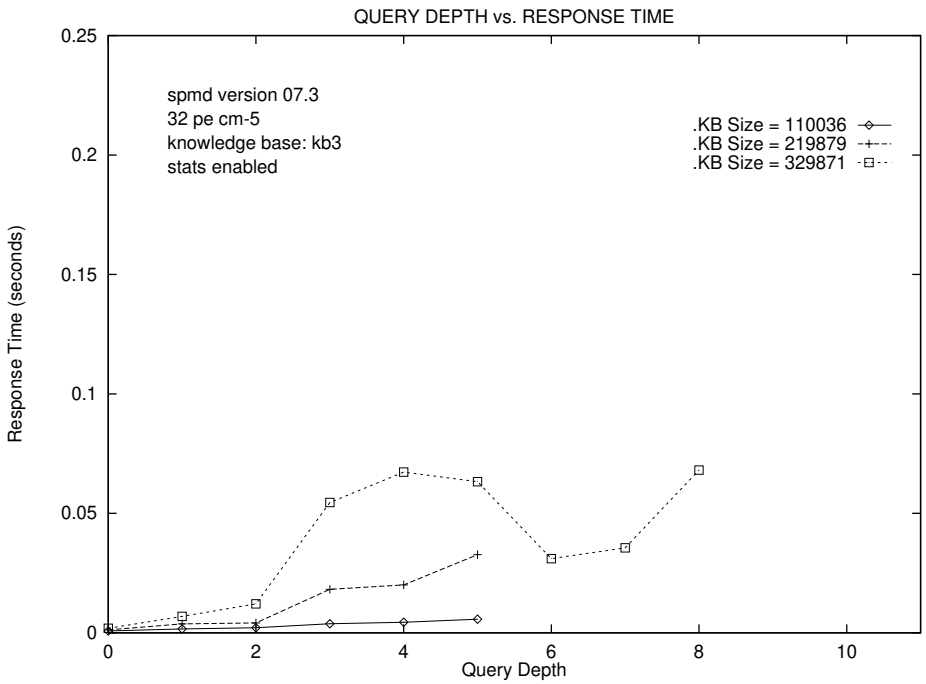

Figure 9: SHRUTI-CM5 running on a CM-5 with 32 processors. The graph shows the effect of the size of the knowledge base on response time for queries with varying inference depths. Due to the random nature of the knowledge base and the queries used, response times for a given depth are statistically reliable only when a large number of data points are averaged. For the larger depths, very few data points were available and this accounts for the seemingly better performance at larger depths. We expect the "dip" in the curve to "straighten out" as more data points are averaged.

The timing reported in the graphs is the elapsed time needed to process the queries. Random, structured knowledge bases were used in these tests (see Section 5.7). These knowledge bases exploited the full functionality of the reasoning system and had a mix of regular rules and facts, rules with special conditions, quantified facts and *is-a* relations. Rules with special conditions included rules with repeated variables, typed variables, existential variables and entities; rules with multiple predicates in the antecedent and rules which lead to multiple instantiation of predicates. In spite of the large scale of these experiments, it is evident that SHRUTI-CM5 provides relatively good performance. Figure 3 compares the performance of SHRUTI-CM5 and SHRUTI-CM2.

5.7 Generating Knowledge Bases

Almost all experimentation with SHRUTI-CM5 has been carried out using randomly generated structured knowledge bases. Though the individual knowledge base elements are generated at random, these elements are organized into *domains* thereby imposing structure on the knowledge base. Each domain is a cluster of predicates along with their associated rules and facts. Domains could be of two types: *target* domains, which correspond to "expert" knowledge about various real-world domains; and *special* domains, which represent basic cognitive and perceptual knowledge about the world. A typical structured knowledge base would consist of several target domains and a small number of special domains. The predicates within each target or special domain, and predicates across target and special domains, are richly connected by rules; predicates across different target domains are sparsely connected. The structure imposed on the knowledge base is a gross attempt to mimic a plausible structuring of real-world knowledge bases. This is motivated

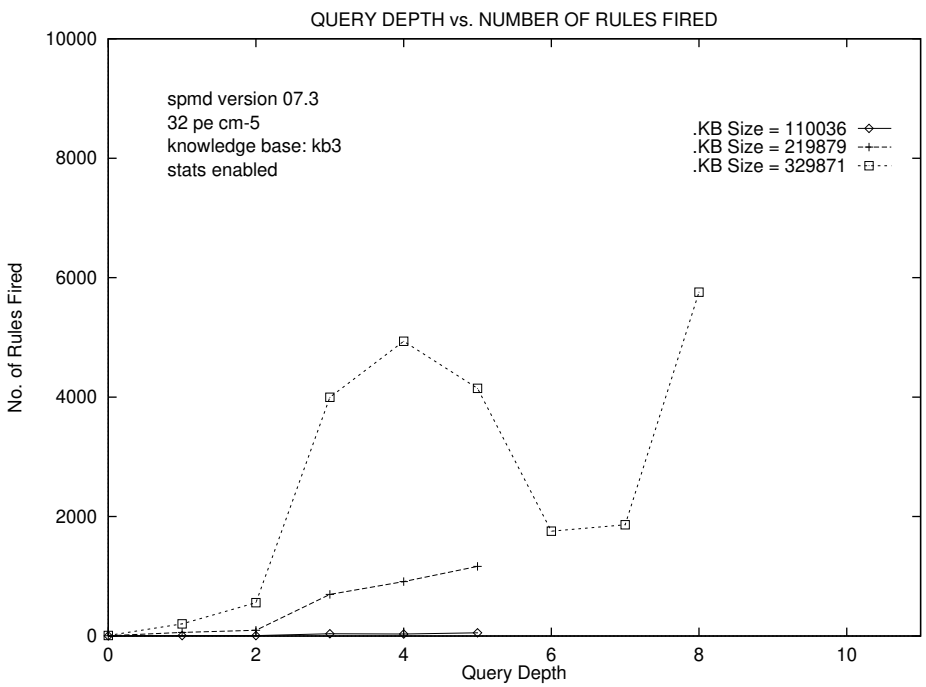

Figure 10: SHRUTI-CM5 running on a CM-5 with 32 processors. The graph shows the number of rules fired in answering queries with varying inference depths. See caption for previous figure for an explanation of the unexpected "dip" in the curve. Also note that the shape of the curves are very similar to those in the previous figure.

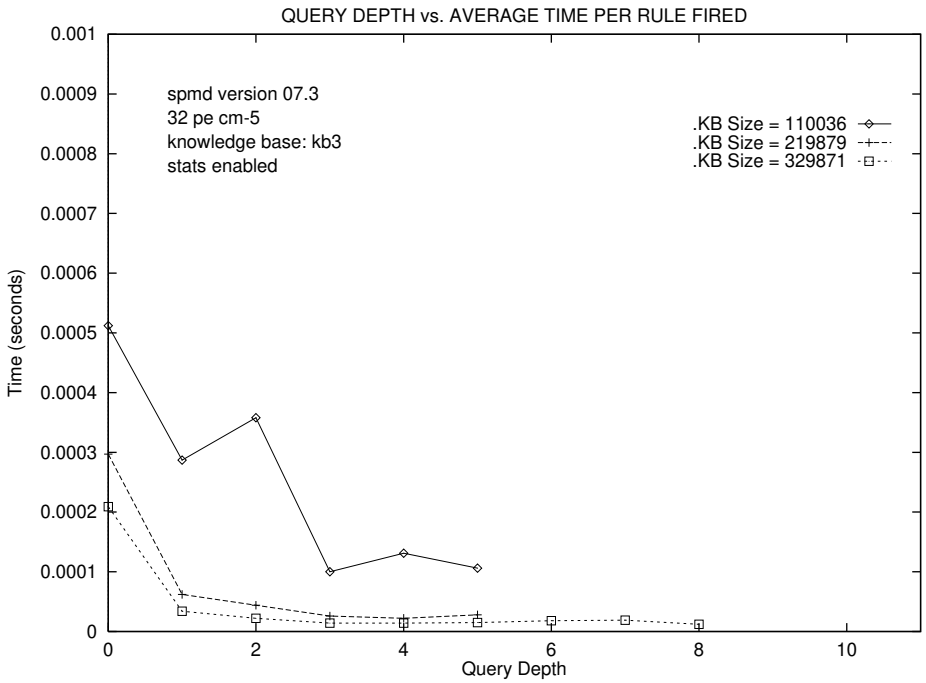

Figure 11: SHRUTI-CM5 running on a CM-5 with 32 processors. The graph shows the average time needed to fire a rule, shown as a function of knowledge base size and query depth.

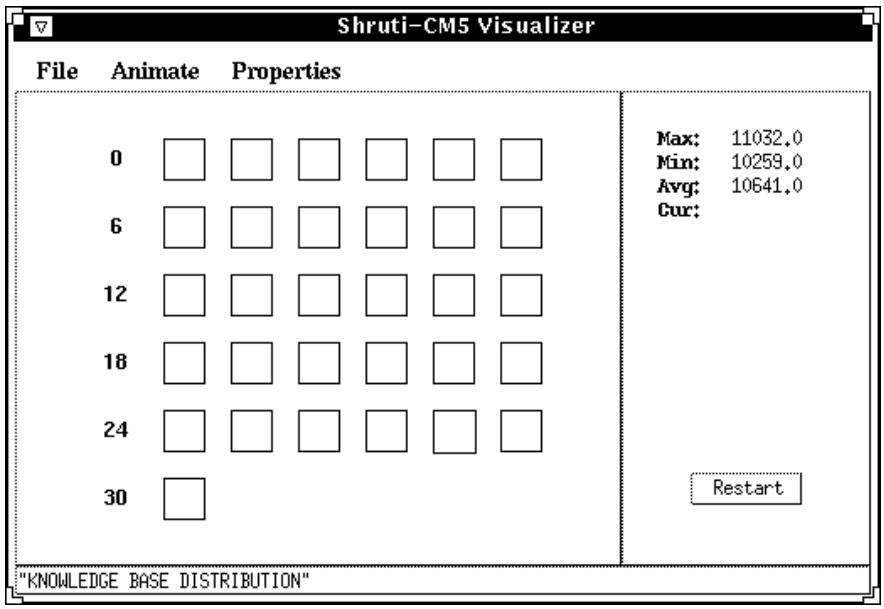

Figure 12: SHRUTI-CM5 running on a CM-5 with 32 processors. Distribution of knowledge base elements (rules, facts and *is-a* relations) on the CM-5 processors for a knowledge base with approximately 300,000 elements. Note that the server (processor number 31) is not shown.

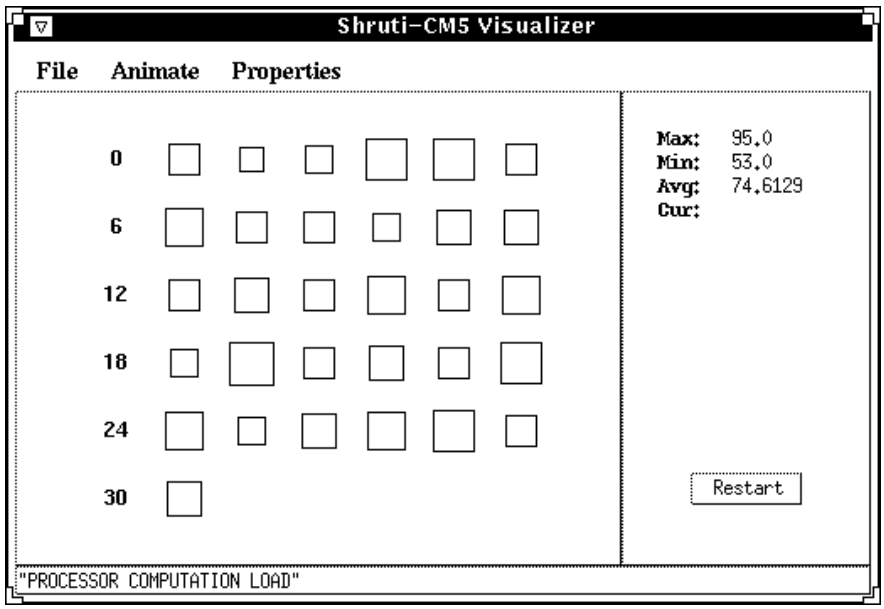

Figure 13: SHRUTI-CM5 running on a CM-5 with 32 processors. Computational load distribution on the CM-5 processors. The number of active predicates, entities and facts on each processor is shown. This load distribution was obtained when answering a query of depth 8 with a knowledge base of size approximately 300,000. Note that the server (processor number 31) is not shown.

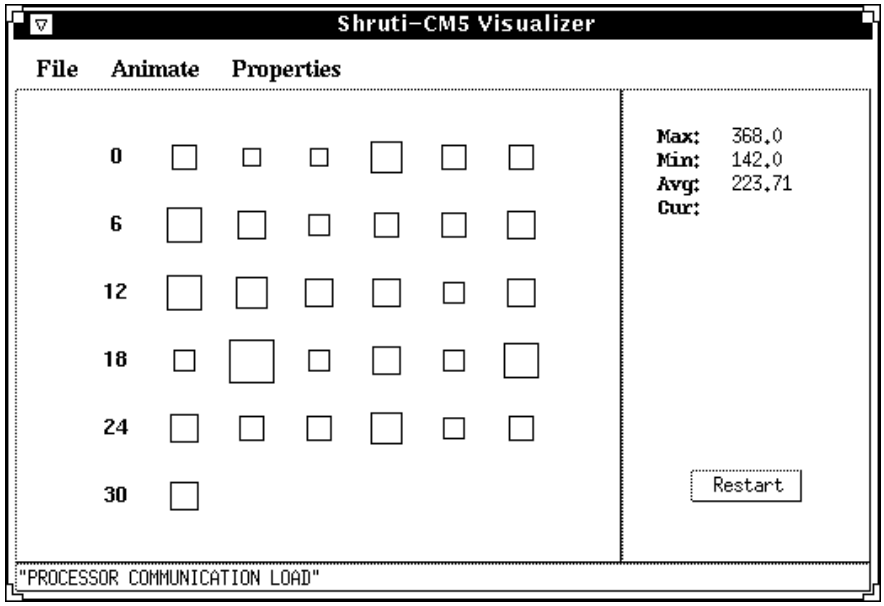

Figure 14: SHRUTI-CM5 running on a CM-5 with 32 processors. Communication load distribution on the CM-5 processors. The number of active messages sent by each processor is shown. This load distribution was obtained when answering a query of depth 8 with a knowledge base of size approximately 300,000. Note that the server (processor number 31) is not shown.

by the notion that knowledge about complex domains are learned and grounded in metaphorical mappings from certain basic, perceptually grounded domains (Lakoff and Johnson, 1980). However, the "knowledge" in each domain is currently being generated at random.

The knowledge base generator takes several parameters as input. These parameters decide the number of predicates, entities, rules and facts that will be generated, the fractions of various special rules, facts and *is-a* relations, the number of domains, the distribution of the knowledge base among the domains and the fraction of inter- and intra-domain rules. The number and maximum depth of the type hierarchies generated can also be controlled.

The parameters supplied to generate the knowledge base used for the CM-5 experiments (identified in the graphs as `kb3`) are shown below:

```
------------  Knowledge Base Parameters  ------------
 Number of rules:                          150000
 Number of facts:                          150000
 Number of predicates:                     50000
 Number of concepts:                       50000
 Multiple antecedent rule fraction:        0.10
 Multiple instantiation rule fraction:     0.10
 Special rule fraction:                    0.40
 Fraction of is-a facts:                   0.25
 Fraction of facts with E vars:            0.10
------------  Domain Parameters  --------------------
 Number of special domains:                3
 Number of target domains:                 150
 Spl-Tgt knowledge base split:             0.02
 Fraction of intra-special-domain rules:   1.00
```

```
Fraction of inter-special-domain rules:   0.00
Fraction of intra-target-domain rules:    0.96
Fraction of inter-target-domain rules:    0.01
Number of type hierarchies:               10
Maximum depth of type hierarchies:        5
Fraction of shared leaves in type hiers.: 0.05
```

5.8 Proposed Experiments with Real-World Knowledge Bases

Recently we have obtained WordNet (Miller et al., 1990) and plan to map it to our system. Although WordNet does not exercise the full expressive and inferential power of our system, it is a sufficiently large knowledge structure with numerous applications and can be used to test the effectiveness of certain aspects of our system design, especially those having to do with message passing. We have also obtained a large knowledge base consisting of over 14,000 frames and 170,000 attribute-value pairs about plant anatomy and physiology from Bruce Porter of the University of Texas at Austin (Porter et al., 1988). The mapping of this knowledge base to our system is very similar to that of WordNet. We are also trying to acquire a subset of the CYC knowledge base (Lenat et al., 1990).

A planned application of our knowledge base system is to couple it to the Berkeley Restaurant Project (BeRP) speech understanding system being developed at the International Computer Science Institute (Jurafsky et al., 1994a; Jurafsky et al., 1994b). BeRP functions as a knowledge consultant whose domain is restaurants in the city of Berkeley, California. Users ask spoken-language questions of BeRP which then queries a database of restaurants and gives advice based on cost, type of food, and location. The current BeRP system cannot perform inferences and any possible inferences are either hard wired into the grammar or added to the restaurant database. Our knowledge base system will allow BeRP to make inheritance-like inferences (a Chinese restaurant is an Asian restaurant) as well as more complex inference (if the user has a car they can get to more distant restaurants). The rapid response of our knowledge base system will be particularly useful for an on-line speech understanding system like BeRP.

5.9 The SHRUTI-CM5 User Interface

The following example illustrates the existing user interface to SHRUTI-CM5 and supporting utilities.

1. **Knowledge base generation.** The user must begin with a knowledge base in a syntax recognized by SHRUTI-CM5. Knowledge bases in other formats should be translated into a form accepted by the system. The following is an example knowledge base in SHRUTI-CM5 syntax.

```
/* Rules */
Forall x,y,z  [ give(x,y,z) => own(y,z) ];
Forall x,y    [ own(x,y) => can_sell(x,y) ];
Forall x:Animal, y:Animal
              [ preys_on(x,y) => scared_of(y,x) ];
Forall x,y,z Exists t
              [ move(x,y,z) => present(x,z,t) ];
Forall x,y,z  [ move(x,y,z) => present(x,y,t) ];
Forall x,y    [ sibling(x,y) & born_together(x,y) => twins(x,y) ];
Forall x,y    [ sibling(x,y) => sibling(y,x) ];

/* Facts */
give (John,Mary,Book1);
move (John,Nyc,Boston);
```

```
sibling (John,x);
Forall x:Cat, y:Bird [ preys_on(x,y) ];
Exists x:Robin [ own(Mary,x) ];

/* Type hierarchy */
is-a (Bird,Animal);
is-a (Cat,Animal);
is-a (Canary,Bird);
is-a (Tweety,Canary);
is-a (Sylvester,Cat).
```

It is also possible to create a (pseudo-random) knowledge base using the knowledge base generator (Section 5.7). The output of the generator is in the above syntax.

2. **Preprocessing and loading.** The preprocessor reads the input knowledge base, assigns knowledge base items to CM-5 processors (using one of several available processor assignment schemes) and writes out a set of files. These files are read and encoded on the CM-5.

3. **Parallel knowledge processing.** Once the KB has been loaded on the CM-5 one can pose queries, obtain answers, and gather performance and timing data. The following dialog illustrates how the user interacts with the system. The system prompt is `>>`. User input is in `typewriter` font while system output is shown in *slanted* font.

```
>> i input-kb.pp
     Processing file input-kb.pp .... done
>> m -g
>> i
     Enter Rules/Facts or Query:
     can_sell(Mary,Book1)?
>> r
     Simulating ... done
     Query answered affirmatively in 0.001638 seconds
>> z
     Resetting network ... done.
>> i query
     Processing file query .... done
>> r
     Simulating ... done
     Query not answered
>>
```

The input command `i` is used to input the knowledge base and to pose queries.[7] The run command `r` runs a reasoning episode. It reports elapsed time if the query is answered (as in the case of `can_sell(Mary,Book1)?`). If the query is not answered, no timing is displayed (as in the case of the query contained in the file `query`). Further commands can be used to view knowledge base distribution on the processors, processor load, individual processor timing, number of rules fired, active predicates and concepts, number of messages sent, and so on (see Appendix E).

The system also provides the capability to process command files in order to facilitate unattended batch processing.

[7] The `m -g` command puts the system in direct input mode. The system always starts up in parallel input mode and hence the first `i` commands reads input in parallel. In order to pose the query directly using the second `i` command, the input mode is changed using the `m` command. See Appendix E.

4. **Analysis and visualization.** The data obtained from reasoning episodes can be analyzed and plotted as graphs (Figures 9–11); dynamic processor load, timing, etc. can be visualized (Figures 13 and 14); knowledge base distribution can be analyzed and visualized (Figure 12); and the actual connectivity of the knowledge base can be graphically displayed. All analysis and visualization are done off-line.

Integrated User Environment

In the existing SHRUTI-CM5 system, all tools and utilities are separate programs. The user must manually invoke the required program or script in order to execute any kind of processing, analysis or visualization. Future versions of SHRUTI-CM5 will provide an easy-to-use graphical user interface which integrates the entire suite of programs and tools (Section 2). The parallel rapid reasoning system would form the core of the SHRUTI-CM5 system around which all the other programs and tools would be organized. Data processing, analysis and visualization tools would be a combination of scripts, already existing tools and custom generated programs. Except for the parallel part, all the other tools would be off-line and usable on a workstation.

The SHRUTI-CM5 system would also provide for automated remote access to the CM-5 so that all off-line tools and processing can be confined to the local workstation. The parallel reasoning episodes will be run on the remote CM-5 and the results and output transferred back to the local workstation for further processing.

6 Related Work

There has been considerable work in the conceptual design of massively parallel systems based on spreading activation, marker passing, and connectionism (Lange and Dyer, 1989; Sun, 1992; Barnden and Srinivas, 1991; Waltz and Pollack, 1985; Charniak, 1983; Fahlman, 1979). However, only very few researchers have tried to implement knowledge base systems on existing parallel platforms. A salient example of such work is the PARKA system (Evett et al., 1993) implemented on the CM-2. PARKA encodes frame-based knowledge (analogous to a semantic network) and supports efficient computation of inheritance, recognition, and structure retrieval which is a generalization of recognition. The performance of PARKA has been tested using pseudo-random networks (with up to 130,000 nodes) as well as subsets of CYC (Evett et al., 1993; Lenat et al., 1990). The CYC subsets used had about 26,000 units. PARKA's run time for inheritance queries is $O(d)$ and for recognition queries is $O(d+p)$ where d is the depth of the *is-a* hierarchy and p is the number of property constraints. Actual run-times range from a fraction of a second (for inheritance queries) to a little more than a second (for recognition queries with 15–20 conjuncts). PARKA does not support rule-based reasoning; it can only handle frame-based knowledge with some extensions to deal with memory-based reasoning.

Semantic Networks on Special Purpose Hardware

Fahlman (1979) proposed the design of NETL, a massively parallel machine that could execute marker passing algorithms for computing inheritance and recognition in parallel. Although this machine was never built, it influenced the design of the CM-2 (Hillis, 1985). Researchers such as Moldavan (1993) have also proposed and built special purpose hardware for realizing semantic networks and production systems.

The Semantic Network Array Processor (SNAP) developed at the University of Southern California is described in (Moldovan et al., 1992). The conceptual design of the SNAP is based on associative memory and marker passing, and is optimized for representing and reasoning with semantic networks. The SNAP provides a special instruction set for network creation and maintenance, marker creation and propagation, logic operations and search/retrieval. A SNAP prototype has been built with off-the-shelf components and used to implement a parallel, memory-based parser (Moldovan et al., 1992). The parser is capable of processing sentences in 1–10 seconds depending on the sentence length and the size of the knowledge base used. The largest knowledge base used consisted of about 2,000 nodes.

Unlike SHRUTI and PARKA, SNAP-based knowledge representation systems use special purpose hardware. Further, SNAP-based systems can only deal with semantic networks and do not currently support the full range of inferences supported by SHRUTI.

The partitioning and mapping of production systems (or rule-based systems) onto multiprocessors is considered in (Moldovan, 1989). A performance index is obtained by analyzing rule interdependencies. This performance index is optimized so as to maximize inherent parallelism and minimize interprocessor communication. Optimizing the performance index is intractable and approximations and simplifications are necessary in order to make the problem tractable. A message-passing multiprocessor architecture (RUBIC, for Rule-Based Inference Computer) for parallel execution of production systems is also described.

7 Conclusion

We have described an SPMD mapping of SHRUTI on to the Connection Machine CM-5. We have discussed issues involved in the design and implementation of this system—both from machine independent and machine dependent points of view. From the test results summarized in the previous sections, it is evident that SPMD implementations are vastly superior in comparison with the SIMD system and offer several hundred-fold speedups. In view of its greatly improved performance, we plan to expend our effort in improving and extending the asynchronous (SPMD) message passing system on the CM-5. The SPMD rapid reasoning system on the CM-5 is also being mathematically analyzed (Mani, 1994) with the objective of obtaining quantitative measures which can be used to further improve performance.

SHRUTI-CM5[8] currently supports only backward reasoning. Future work on the CM-5 will involve developing a forward reasoning system and an integration of the forward and backward reasoners.

All experiments reported here have used randomly generated knowledge bases. As noted in Section 5.8, we plan to encode large real-world knowledge bases on the system and interface it with applications. This will not only help us evaluate the parallel rapid reasoning systems more thoroughly, but will also result in practical and usable systems. Depending on the kind of knowledge bases used, we also expect this endeavor to provide insights into aspects of reflexive reasoning.

8 Acknowledgements

This work has been supported by ARO grants DAA29-84-9-0027 and DAAL03-89-C-0031 to the Army Research Center at the University of Pennsylvania, ONR grant N00014-93-1-1149 and NSF resource grant CCR930001N to Lokendra Shastri, and National Science Foundation Infrastructure Grant CDA-8722788 to the University of California at Berkeley. All CM-5 and 32K, 16K and 8K CM-2 experiments were run on the Connection Machines at the National Center for Supercomputing Applications (NCSA), University of Illinois at Urbana-Champaign. All experiments on the 4K CM-2 were run at the University of Pennsylvania. The CM-5 workshop at NCSA—conceived by the MetaCenter Computational Science Institutes in Parallel Computing, sponsored by the NSF, and organized by the National Center for Supercomputing Applications and the San Diego Supercomputing Center—provided D. R. Mani with some computing resources and a great learning opportunity.

Thanks to David Bailey, Susan Davidson, Jerry Feldman, Ben Gomes, Dan Jurafsky, Marilynn Livingston, Srini Narayanan, Chris Overton, Sanguthevar Rajasekaran, and David Waltz for several comments and suggestions which have contributed to this work. Thanks to Tom Fontaine and the CM-2 support staff at the University of Pennsylvania for their help in getting started with the CM-2 and `C*`, the consulting staff at NCSA and the University at California at Berkeley for their help with the CM-5.

[8] And SHRUTI-CM2, see Appendix A.

A SHRUTI on the CM-2

The CM-2 (TMC, 1991a) is an SIMD data parallel computing machine which can be configured with up to 64K processing elements. Each processor has several kilobits of local memory and can execute arithmetic and logical instructions, calculate memory addresses, read and store information in memory and perform interprocessor communication. The processors are organized as an n-dimensional hypercube. The CM-2 is controlled by a standard serial front end processor (usually a VAX or SUN machine). A sequencer decodes commands from the front end and broadcasts them to the data processors, all of which then execute the same instruction simultaneously and synchronously. A NEWS grid provides fast communication between adjacent processors and a router network provides general interprocessor communication between any two processors.

The design and implementation of the SIMD parallel rapid reasoning system on the CM-2—SHRUTI-CM2—is based on knowledge-level partitioning (Section 4) of the underlying network generated by a knowledge base. We describe techniques used to encode the knowledge base and implement spreading activation when answering queries. We then explore the characteristics of the system by running a battery of tests. All discussion pertains only to backward reasoning.

A.1 Encoding the Knowledge Base

The knowledge base is encoded by presenting rules, facts and *is-a* relations to the SHRUTI-CM2 system. The input syntax for rules, facts, *is-a* relations and queries is specified in Appendix D. Appendix E gives a listing of commands recognized by SHRUTI-CM2.

Input Processing

A lexical analyzer and parser read the input, parse it and build internal data structures which represent the rules and/or facts presented to the system. All input processing is performed sequentially on the front-end.

As predicates and entities (or concepts) are recognized in the input, the parser builds hash tables which keep track of processor assignments. The hash tables can be used to efficiently access these predicates and entities while encoding rules and facts, posing queries and inspecting their state.

Once a rule, fact or *is-a* relation has been recognized and processed, the resulting internal data structures can be used to encode the knowledge base element on the Connection Machine processors. In the case of a query, the data structures will be used to pose the query to the system.

Representing Knowledge Base Elements

Knowledge base elements are represented on the processors using *parallel* structures. A parallel structure allocates space for the specified structure on *every* processor. Figures 15 and 16 indicate the structures used to encode predicates, rules and facts in the rule-base. The structures used to encode concepts and *is-a* relationships in the type hierarchy are similar (though simpler). Note that a parallel structure will be allocated for each knowledge base element: predicate, fact, rule, concept and *is-a* link. When the knowledge base grows and more space is needed, the size of the parallel structure is doubled. The virtual processor capability of the CM-2 ensures that each (physical) processor now houses two structures. This is transparent to the programmer and one can still assume that each processor houses one structure, with double the number of (virtual) processors in the machine. Using this scheme, the representation automatically scales with the size of the knowledge base. As the number of virtual processors increases, the system will run proportionately slower. The virtual processor mechanism therefore provides a simple, scalable and transparent way of trading off time for space.

```
typedef struct cm_pred           /* predicate on the CM */
{
  bool               used;       /* flag */
  byte               noOfArgs;

  byte               nextFree;   /* index of next free bank (minst) */
  struct cm_predbank bank[K2];   /* predicate banks */
} CM_Pred;


typedef struct cm_predbank       /* predicate bank on the CM */
{
  /* no fields used to encode KB */

  bool   cChange;                /* collector value changed */
  bool   eChange;                /* enabler value changed */
  byte   collector;
  byte   enabler;
  byte   args[MAX_ARGS];         /* arg activation phase */
} CM_PredBank;
```

Figure 15: Structures used to represent predicates in SHRUTI-CM2. `MAX-ARGS` is the maximum number of arguments a predicate can have. `K2` is the multiple instantiation constant for predicates. Flags have type `bool`. The top part of the `typedefs` contain fields used to encode the knowledge base while the bottom part has fields used in a given episode of reasoning.

Encoding Rules and Facts

Depending on the processor allocation scheme used (Section 4), every predicate and entity appearing in the knowledge base will be assigned to a (virtual) processing element on the CM-2. Further, a rule, fact or *is-a* relation that is being encoded will also be assigned to a (virtual) processor. These two processor allocations—one for the relevant predicates/entities and the other for the rule/fact under consideration—may or may not be independent. The actual details of the processor allocation are dictated by the processor assignment scheme being used.

The current and more recent versions of SHRUTI-CM2 use random processor assignment schemes for all knowledge base elements. Earlier versions used random allocation for predicates and concepts; however, facts and *is-a* links were encoded on the processors containing the relevant predicate or concept and rules were encoded on the processor containing the consequent predicate.

Once the predicates, concepts and other knowledge base elements under consideration are assigned to processing elements on the CM-2, all that remains to be done in order to encode the rule/fact is to correctly fill out the various fields in the relevant structures. Encoding a fact involves the corresponding predicate and the entities filling the arguments of the predicate. Encoding a rule (*is-a* relation) involves two predicates (concepts) and a rule-slot (*is-a* link). If a rule has multiple predicates in the antecedent, the encoding is slightly more complex, as pictured in Figure 17.

A.2 Spreading Activation and Inference

Queries can be posed after the knowledge base has been encoded. Again, queries have a specific syntax (as described in Appendix D) and result in activating the relevant predicate and concepts in keeping with the

```
typedef struct cm_rule           /* rule slot on the CM */
{
  /* knowledge base encoding */
  bool   used;                   /* flag */
  bool   dummy;                  /* rule slot is dummy if flag set */
  index  antecedent;             /* invalid for head rule slots */
  index  consequent;             /* points to head slot in a dummy */
  byte   noOfAnts;               /* > 1 in a head rule slot */
  int    weight;
  byte   antNoOfArgs;            /* invalid for head rule slots */
  byte   argMap[MAX_ARGS];       /* arg mapping; invalid on head slot */
  byte   splCond[MAX_ARGS];      /* not used in dummy slots */
  int    splIndex[MAX_ARGS];     /* not used in dummy slots */

  /* reasoning episode */
  byte   dummyCollector[K2];     /* used only in dummy slots */
  bool   fire;                   /* rule can fire if set */
  bool   selected;               /* instantiation selected if set */
  byte   nextBank;               /* next conseq pred bank to consider */
  byte   bankSelected[K2];       /* rule back pointer */
  /* NOTE: bankSelected[i] == j if bank i in the ant pred has
     instantiation from bank j in the conseq pred;  valid only on
     non-head rule slots; in a head rule slot bankSelected[i] == i */
} CM_Rule;


typedef struct cm_fact           /* fact on the CM */
{
  bool   used;                   /* flag */
  index  factPred;               /* fact predicate index */
  byte   noOfArgs;
  index  constant[MAX_ARGS];     /* fact arguments */

  bool   active;                 /* fact active if set */
} CM_Fact;
```

Figure 16: Structures used to encode rules and facts in SHRUTI-CM2. `MAX-ARGS` is the maximum number of arguments a predicate can have. `K2` is the multiple instantiation constant for predicates. Flags have type `bool` while processor indices have type `index`. The top part of the `typedef`s contain fields used to encode the knowledge base while the bottom part has fields used in a given episode of reasoning.

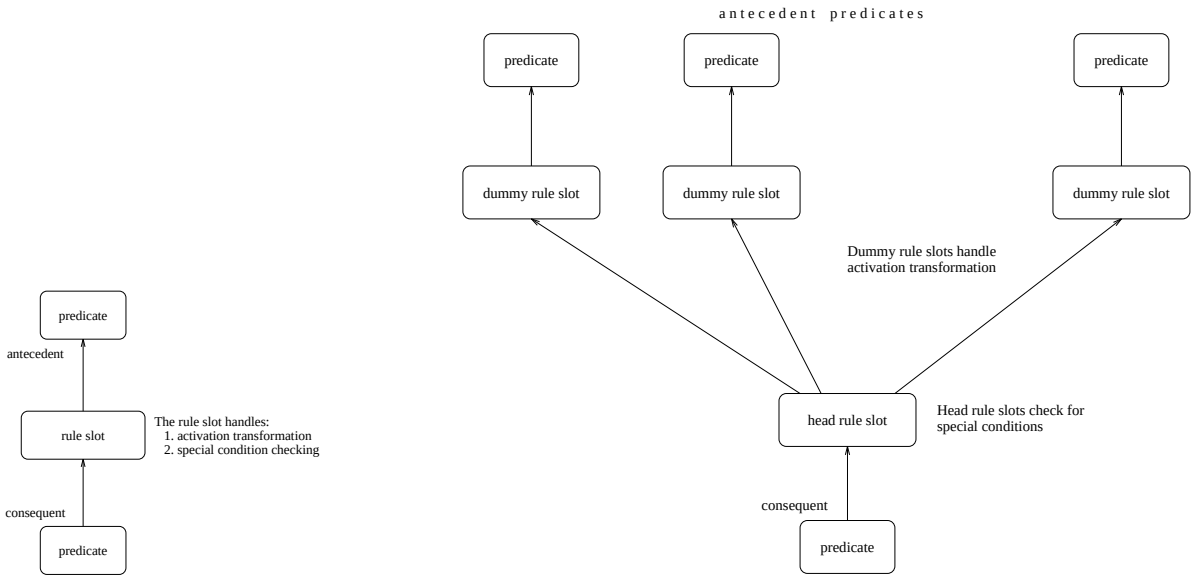

Figure 17: Encoding single- and multiple-antecedent rules. The figure on the left indicates the encoding of single-antecedent rules while the figure on the right depicts the encoding of multiple antecedent rules. Every predicate and rule-slot is housed on a processor. Arrows indicate links which are implemented using interprocessor communication.

description in (Shastri and Ajjanagadde, 1993) and (Mani and Shastri, 1993). The reasoning episode can then be run, either step-wise or to completion. We now describe the mechanics of spreading activation and matching facts in the system. The gross structure of the activation propagation loop is indicated in Figure 8. Phases in SHRUTI are represented as "markers"—integers with values ranging from 1 to the maximum number of phases.

The Rule Base

As shown in Figure 8 spreading activation in the rule base consists of three steps:

- *Propagating rule activation.* Spreading activation in the rule base by rule firing is achieved by executing the following:

  1. Every non-dummy rule-slot **get**s the instantiation in the consequent predicate bank under consideration.
  2. All non-dummy rule-slots check if all special conditions in the rule are satisfied.
  3. If all special conditions are satisfied, the dummy rule-slots **get** the respective instantiations from the corresponding head rule-slot.
  4. All non-head rule-slots transform the activation and **send** it to the respective antecedent predicates.

  In the process of firing a rule, the system maintains sufficient book-keeping information to reverse-propagate collector activation to the consequent of a rule.

  Once a rule fires, it will not fire again unless a new bank of the consequent predicate becomes active. This ensures that the same rule does not repeatedly fire thereby minimizing unnecessary interprocessor communication. Note also that the processor housing the rule-slot will need to communicate with other

processors in order to `get` predicate bank instantiations, `get` information from the head rule-slot, `send` information to dummy rule-slots and `send` the transformed activation to the antecedent predicate.

- *Checking fact matches for active predicates.* All facts for predicates which have active collectors are matched simultaneously. Processors encoding the facts communicate with the processors housing the relevant predicates and concepts in order to check if the firing "phases" match. If a fact "fires", the collector of the corresponding predicate is activated.

- *Reverse-propagating collector activation.* Sending collector activation to predicate banks which originated the activation involves the following:

  1. Non-head rule-slots `get` the state of the predicate collector.
  2. Dummy rule-slots send the collector value to the head rule slot which accumulates all the incoming values.
  3. Non-dummy rule-slots send the activation to the respective consequent predicates provided the collector activation exceeds a threshold. The threshold could depend on the number of antecedent predicates for the rule, the level of activation of antecedent predicate(s), and/or other factors.

  Rule-slots that have already propagated collector activation to the corresponding predicate bank will not participate in this step. Again, this is done in order to minimize unnecessary interprocessor communication.

The Type Hierarchy

Propagating activation in the type hierarchy is similar to spreading activation in the rule-base, except that it is much simpler. Spreading bottom-up activation and top-down activation are handled separately (and sequentially) in the type hierarchy. When spreading bottom-up (top-down) activation, all *is-a* links which have an active bank in the subconcept (superconcept) "fire" and spread activation to the respective superconcept (subconcept). The *is-a* link `get`s activation from the subconcept (superconcept) and `send`s it to the superconcept (subconcept). Again, in order to minimize communication, we ensure that any new activation traverses corresponding *is-a* links exactly once.

Multiple Instantiation

Multiple instantiation in SHRUTI-CM2 is handled without the use of switches (Mani and Shastri, 1993). Predicates and concepts can accommodate `K2` and `K1` instances respectively. When spreading activation in the network, predicate and concept banks are considered one at a time. In other words, in a given clock cycle (i.e., in one iteration of the propagation loop; see Figure 8) only one active bank of a predicate or concept will be considered. As described in Appendix C, care is taken to avoid potential problems that could result from this technique.

Whenever a predicate or concept receives activation, it is compared with existing activation in the banks. If the incoming activation is not already represented, it is then deposited into the next available bank. The rule- or link-slot that sent in the activation is notified that the instantiation it sent has been selected. In the rule base, the rule-slot receives the bank number accommodating the new instantiation. This information is needed when reverse-propagating collector activation. If the incoming activation is already represented in the predicate or concept, or if all banks are already in use, the incoming activation is discarded. Even in this case, rule-slots are notified so that they can proceed to the next bank of the consequent predicate. A rule-slot retries sending the same instantiation if it does not receive notification that the activation was either selected or discarded. This protocol simulates the function of the multiple instantiation switches, and brings about efficient dynamic allocation of predicate and concept banks.

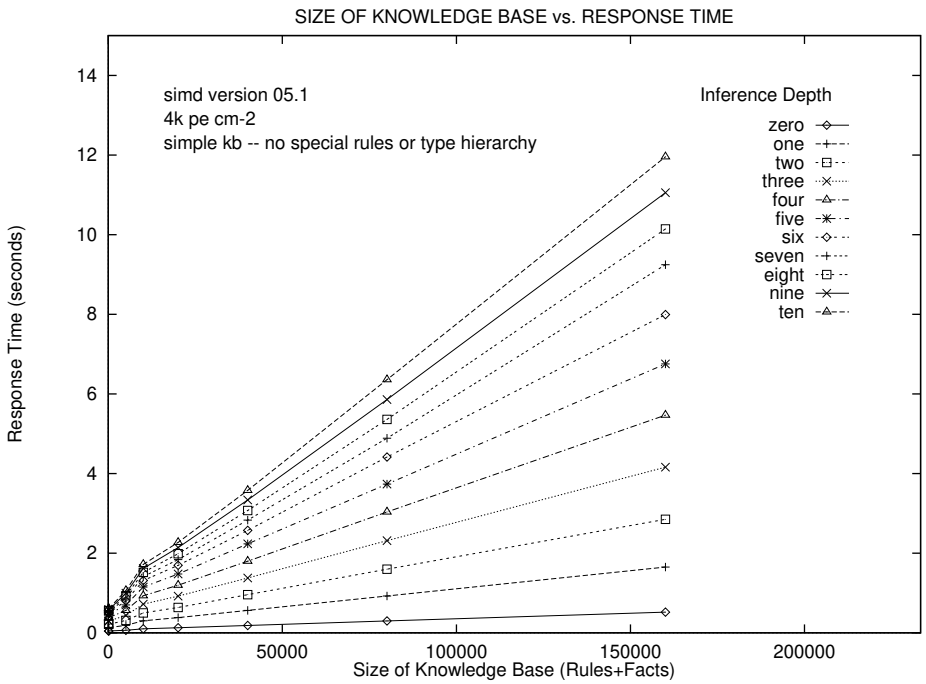

Figure 18: SHRUTI-CM2 running on a CM-2 with 4K processors. The graph shows the effect of the size of the knowledge base on response time for queries which require inference depths ranging from 0 to 10. Queries used were not randomly generated. The knowledge base used was not structured.

Statistics Collection

Apart from timing the reasoning episodes, SHRUTI-CM2 can also be configured to gather data about several other aspects including knowledge base parameters (number of rules, facts, *is-a* relationships, and concepts) and communication data (number of messages, `send`s and `get`s). Enabling full-fledged data collection can slow down the system due to the extra time needed to accumulate the required data.

A.3 Characteristics of SHRUTI-CM2

SHRUTI-CM2 has been run on a 4K CM-2 and on a 32K CM-2. Both machines had 256 kilobits of memory on each processor. Figures 18 and 19 summarize the results of experiments run on these machines. In these figures, the response time shown is the actual CM time used. The timing routines available on the CM-2 also report elapsed time for the reasoning episode. Elapsed time is affected by other processes running on the front end and is therefore unreliable. The knowledge bases used in these experiments were generated at random, and did not contain *is-a* relationships or rules with special conditions. The inference path for a given query was tailored to ensure a reasonable branching factor—at least one of the predicates in the activation frontier had five or more outgoing links originating from it.

Based on these and other experiments, and on the design of SHRUTI-CM2, we can summarize the characteristics of the system:

- The response time is approximately linear with respect to the size of the knowledge base, for knowledge bases with up to 160,000 elements. Thus, as the size of the knowledge base increased, query answering time increased proportionately. This is to be expected since more predicates would be active on the average and would entail proportionately more processing and interprocessor communication as the

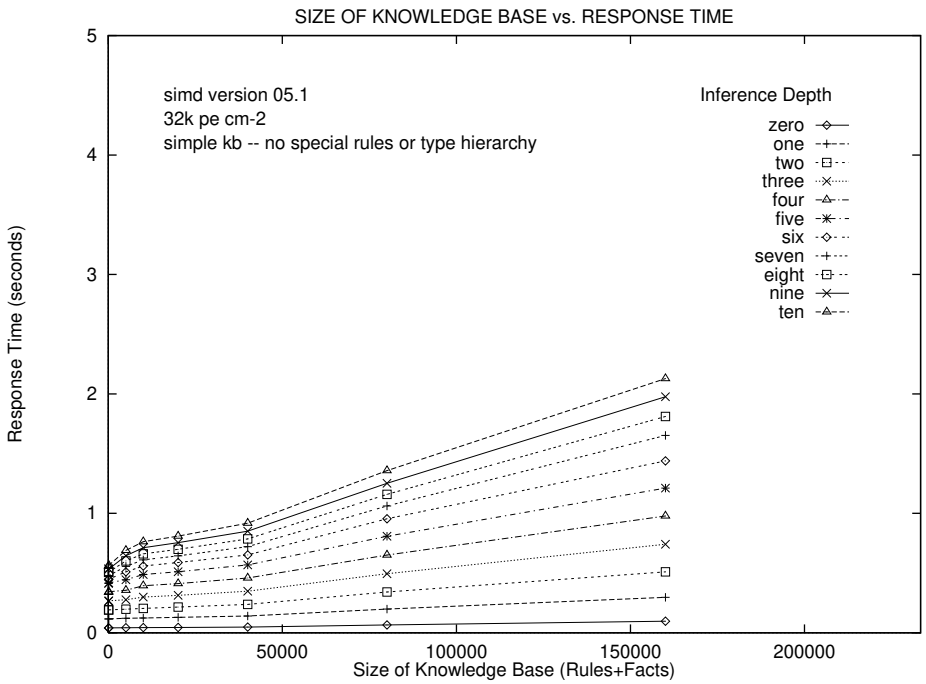

Figure 19: SHRUTI-CM2 running on a CM-2 with 32K processors. The graph shows the effect of the size of the knowledge base on response time for queries which require inference depths ranging from 0 to 10. Queries used were not randomly generated. The knowledge base used was not structured.

size of the knowledge base increases.

Beyond a certain limit, we expect response time to increase steeply with the size of the knowledge base. However, effort was not expended in locating this limit or studying the characteristics of the system near this threshold since our focus shifted to the CM-5. As a result, all timing results stated here apply only to knowledge bases with up to 160,000 rules and facts.

- Time taken to answer a query increases as the average branching factor of the knowledge base increases. This again is caused by increased processing and interprocessor communication.
- Increasing inference depth needed to answer a query proportionately increases response time. Every extra inference step requires an extra activation propagation step (i.e., an extra iteration of the loop in Figure 8).
- Response time is approximately inversely proportional to the number of (physical) processing elements on the machine. This can be attributed to the increased computing power and the lower "density" (with fewer knowledge base elements per processor) which results in enhanced parallelism.
- The time taken to answer a query ranges from a fraction of a second to a few tens of seconds.
- An inherent problem with the use of parallel variables on the CM-2 is inefficient memory usage. Since the number of virtual processors must always be a power of two, this could potentially lead to significant waste of memory. There appears to be no simple solution to this problem without breaking out of SIMD operation. SPMD implementations on the CM-5 avoid this problem entirely.
- The maximum size of the knowledge base that can be encoded on a machine depends on the total amount of memory available on the machine. In addition, with increasingly large knowledge bases, the communication bottleneck would also significantly slow down the system.

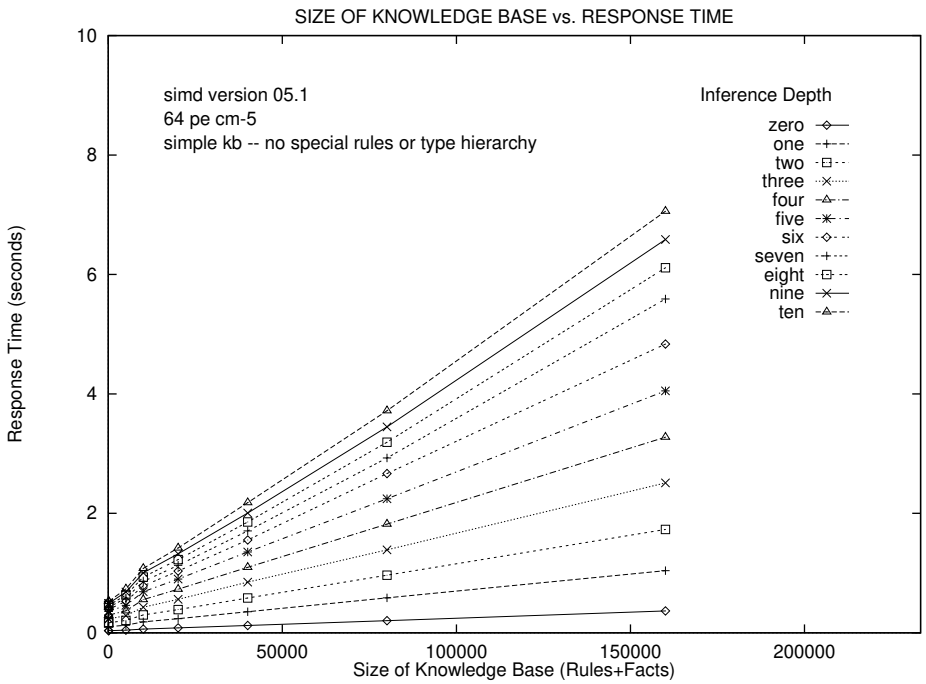

Figure 20: SHRUTI-CM2 running on a CM-5 with 64 processors. The processing nodes on the CM-5 are used in SIMD mode. The graph shows the effect of the size of the knowledge base on response time for queries which require inference depths ranging from 0 to 10. Queries were not randomly generated. The knowledge base used was not structured.

B SHRUTI-CM2 on the CM-5

In this section, we briefly evaluate SHRUTI-CM2 running on the CM-5. Since SHRUTI-CM2 is written in `C*`, and a `C*` compiler is available for the CM-5, SHRUTI-CM2 was recompiled and run on the CM-5. SHRUTI-CM2 running on the CM-5 uses the CM-5 in data-parallel (SIMD) mode. Figure 20 summarizes the results. Comparing with Figures 18 and 19, we observe that the performance of SHRUTI-CM2 on the CM-5 is comparable to that on the CM-2[9], though message passing on the CM-5 appears to be more robust.

C Multiple Instantiation—Some Technical Details

Multiple instantiation in both SHRUTI-CM2 and SHRUTI-CM5 is handled *without* the use of switches (Mani and Shastri, 1993). When spreading activation in the network, predicate and concept banks are considered one at a time. In other words, in a given iteration of the activation propagation loop (see Figure 8) only one active bank of a predicate or concept will be considered. This technique could cause indefinite waits in the rule base. To illustrate the problem, suppose we are currently considering bank i of predicate P. Let P be the consequent of rules r_1 and r_2. Let R_1 and R_2 be rule-structures that represent r_1 and r_2. At propagation step t, suppose r_1 fires and r_2 does not. The fact that r_1 fired for bank i of P will be noted in R_1, and R_1 can shift its focus to the next active bank $i+1$ in the next propagation step. Since r_2 did not fire, R_2 is stuck at bank i. R_2 cannot skip bank i and go on to bank $i+1$ since r_2 could fire later due to activation propagating in the type hierarchy. We circumvent this problem by defining special protocols.

[9]The rule of thumb seems to be that a 32 node CM-5 is approximately equivalent to a CM-2 with 8K processing elements.

Note that this problem does not arise in the type hierarchy since all *is-a* links originating at a concept always fire—unlike a rule, no preconditions need to be satisfied for an *is-a* link to fire.

SHRUTI-CM2

Let D_{th} be the depth of the type hierarchy. Then,

- If a rule r for bank i of some predicate fires at time step t, then update R, the structure representing r, to consider bank $i+1$ of the corresponding predicate in step $t+1$ (subject to the conditions mentioned below).
- If a rule r for bank i of some predicate does not fire at time step t, then two cases are possible:
  1. If $t \leq D_{th}$, then do not update R. Thus, bank i will be reconsidered in step $i+1$.
  2. If $t > D_{th}$, update R to consider bank $i+1$ in the next time step.[10]

Since activation spread in the type hierarchy will not activate any new concepts after D_{th} time steps, this scheme ensures that all banks of a predicate will eventually be considered.

SHRUTI-CM5

In SHRUTI-CM5, the multiple instantiation indefinite wait problem is handled by placing special elements on the rule-frontier. Normally, a rule-frontier element is a (consequent) predicate, along with the bank that was instantiated. All rules for that predicate bank are considered in a given propagation step. If any rule does not fire for this bank, then a special pair of elements is added to the rule-frontier. This pair specifies the predicate bank *and* the associated rule that need to be reconsidered in the next propagation step. Whenever such a pair is encountered on the rule-frontier, only the specified rule is processed. If subsequent banks of the predicate become active, these predicate banks will be placed on the frontier as usual, irrespective of the fact that previous banks could have rules which have not yet fired.

[10] Whenever any rule-slot R is updated to consider an inactive predicate bank, R waits till an instance has been assigned to that bank.

D Input Syntax for Rules, Facts and Queries

To illustrate the input syntax for rules, facts and *is-a* relations, we begin with an extension of the example in Section 5.9.

```
/* RULES */
forall x,y,z [give(x,y,z) => own(y,z)];
forall x,y   [buy(x,y) => own(x,y)];
forall x,y   [own(x,y) => can_sell(x,y)];
forall x,y   [sibling(x,y) & born_together(x,y) => twins(x,y)];
forall x,y   [preys_on(x,y) => scared_of(y,x)];
forall x,y,z [move(x,y,z) => present(x,z,t)];
forall x,y,z [move(x,y,z) => present(x,y,t)];
forall x,y exists t
             [born(x,y)   => present(x,y,t)];
forall x:Animate, y:Solid_obj
             [walk_into(x,y) => hurt(x)];

/* FACTS */
give (John, Mary, Book1);
give (x, Susan, Ball2);
forall x:Cat, y:Bird preys_on (x,y);
exists x:Robin [own(Mary,x)];

/* IS-A FACTS */
is-a (Bird,Animal);
is-a (Cat,Animal);
is-a (Robin,Bird);
is-a (Canary,Bird);
is-a (Tweety,Canary);
is-a (Sylvester,Cat).
```

NOTE: Any text included between /'s are comments. The comments given above are enclosed between /* ... */ so that they look identical to comments in C code.

The above example illustrates the input syntax accepted by the parallel rapid reasoning systems. Most of the features are self-evident. Some points to be noted regarding the input syntax follow. Items prefixed by a dagger (†) are supported only by SHRUTI-CM5.

- A rule meant for the backward reasoner is said to be *balanced* if the following conditions are satisfied:
  - Repeated variables in the antecedent are also present in the consequent.
  - Typed variables, existential variables and entities present in the antecedent are also present in the consequent.

  Only balanced rules will be accepted by the system. Rules which do not satisfy the above conditions will be rejected. A warning message to this effect will be printed.
- Any variable (used in a rule) which is not listed in either the list of universally quantified variables or in the list of existentially quantified variables is assumed to be existentially quantified.
- Any name beginning with an uppercase alphabetic character is assumed to be an entity. All names beginning with lowercase are variable names. Names of predicates can begin with either uppercase

or lowercase letters. Capitalization of names should be consistently used — for example, `name1` and `Name1` would represent two *different* predicates; similarly, `Const_a` and `Const_A` are different entities.

- A semicolon (`;`) indicates that a rule, fact or *is-a* fact has been entered; it also indicates that more input is to follow. The occurrence of a period (`.`) in the input indicates the end of a rule, fact or *is-a* fact and also terminates the input. A (quantified or unquantified) predicate terminated by a `?` is interpreted as a *query*.

- The lexical analyzer removes all whitespace; the input is therefore unaffected by the addition of extra blanks, tabs or newlines. Further, spaces can be omitted wherever it is not essential[11].

- The lexical analyzer also removes all *comments*. Any text enclosed between `/`'s (`/ ... /`) is a comment. The text of a comment can contain any character or symbol except `/`. A comment can start and end at any point in the input. In particular, a comment may span several lines or may be limited to part of a single input line.

- †**Tags.** Predicates and entities can be tagged (with a non-zero, positive integer) by using the `< >` construct: `<give(x,y,z),3>` or `<Mary,6>`. Tags can be used to group "similar" predicates and entities together.

- **Error Handling.** When syntax errors are detected in the input, the action taken depends on the mode of input:

  - If input is being read from the terminal (`stdin`), an error message is issued, and the *last* rule or fact should be re-entered after typing one or more semi-colons (;).
  - If input is being read from a file, the parser prints the line number containing the syntax error and continues reading the file, so that all syntax errors in the file are listed. Rules or facts in the input that were correctly recognized (i.e., had no syntax error) will be encoded; the others will be ignored.

Below is the formal grammar for the input language (for rules, facts, *is-a* relations and queries) which specifies the exact form of each input structure. The grammar is accurate for SHRUTI-CM5. Though most of the constructs are identical in SHRUTI-CM2, there are some minor differences. Further, SHRUTI-CM2 does not support tags.

```
input → .                                   /* stop – no more input */
        | ; input                           /* continue – more input */
        | input-item input
input-item → query                          /* query */
        | fact                              /* fact */
        | rule                              /* rule */
        | tag-def                           /* tag definition */
rule → q-prefix [ pred-list => predicate ]
        | pred-list => predicate
fact → predicate
        | q-pred
query → predicate ?
        | q-pred ?
tag-def → < predicate , NUM >
        | < constant , NUM >
q-pred → q-prefix [ predicate ]
```

[11] To distinguish between the variable '`forallx`' and '`forall x`', a space is *essential*. But a space is not required after the '`,`' in '`own(x,y)`'. In general, spaces are not essential before and after punctuation symbols.

```
q-prefix → FORALL type-list
         | EXISTS type-list
         | FORALL type-list EXISTS type-list
         | EXISTS type-list FORALL type-list
type-list → variable
         | variable : constant
         | variable , type-list
         | variable : constant , type-list
pred-list → predicate & pred-list
         | predicate
predicate → arg-or-pred ( arg-list )
         | arg-or-pred ( )
arg-list → arg-or-pred , arg-list
         | arg-or-pred
arg-or-pred → constant | variable
constant → CONST
variable → VAR
```

Here, CONST represents entities (any token starting with an uppercase letter), VAR are variables (quantified or unquantified) in the rules, facts or queries and are tokens beginning with lowercase letters. The variable and entity tokens are represented by a sequence of alphanumeric characters along with _ and *. Any integer is recognized as a NUM. The tokens FORALL and EXISTS are recognized when the input contains these words, spelled with any combination of uppercase and lowercase letters (i.e., arbitrarily capitalized).

E SHRUTI-CM Commands

Commands recognized by SHRUTI-CM2 and SHRUTI-CM5 are listed below. Some of the commands and descriptions are applicable only to SHRUTI-CM5 and are prefixed by a dagger (†). The SHRUTI-CM5 preprocessor only supports the commands `i`, `w` and `q`. Each command is invoked by using a single character. The first non-blank character typed at the input prompt is taken to be the command. Any non-blank text following the first character forms the argument(s) for the command. The list below indicates the purpose of the command, the command syntax and a brief description of the command.

Quit Syntax: `q`

Terminates the SHRUTI-CM program.

Help Syntax: `?`

Prints out a list of available commands and the command-line options and/or arguments which the commands accept.

Read Input Syntax: `i [ -f | -b ] [input-file]`

Reads input from the terminal (when `input-file` is not specified) or a file (when `input-file` is specified). The `-b` option is used to build a backward reasoning system (default), while the `-f` option builds a forward reasoning system (currently unsupported).

†In SHRUTI-CM5 the behavior of this command is dictated by the current input mode. The system always starts up in parallel asynchronous mode; the mode can be changed using the `m` command. In parallel asynchronous mode, each processor in the partition processes a different input file `input-file.pid` where `pid` is a three digit processor index (prefixed by zeros if necessary). In global synchronous mode, all processors cooperatively process the same input file `input-file`.

†Syntax: `i [-h hash-table-file] [ -f | -b ] [input-file]`

The `-h` option for read input is supported by the SHRUTI-CM5 preprocessor and can be used to update the internal server hash tables which store processor assignment and other details for predicates and concepts. This feature is useful for incremental preprocessing of large knowledge bases.

†**Change Input Mode** Syntax: `m [ -p | -g ]`

Changes input mode to parallel asynchronous (with the `-p` option) or to serial, global synchronous (with the `-g` option). Without any option, this command prints out the current input mode. The current input mode dictates the behavior of the `i` command.

†**Write Out Hash Table** Syntax: `w [-o output-file-prefix]`

Writes out the current server hash tables to the specified file (with a `.hashtables` extension). If no output file prefix is given, `kb.pp` is used as default. The hash tables written out can be read by the preprocessor (using the `i` command with the `-h` option) and supports incremental preprocessing of large knowledge bases.

†Syntax: `w [ -g ] [-o output-file-prefix]`

This command, when used on the SHRUTI-CM5 preprocessor, writes out the preprocessed knowledge base. The output file names are suffixed with the processor number. If the output file prefix is not specified, `kb.pp` is used as the default. If the `-g` option is absent, the inference dependency graph for the knowledge base is also written out (with file extension `.idg`)

Run Reasoning Episode Syntax: `r [[-f] #steps]`

Runs the reasoning episode after a query has been posed. It is an error to invoke this command when a query has not been posed. Without any options or arguments, `r` runs the reasoning episode to completion—till the query is answered or the reasoning episode has proceeded long enough to conclude that there will be no answer. When `#steps` is specified with the `-f` option, the reasoning episode is forced to run for `#steps` propagation steps (irrespective of whether the query has been answered or

not). If the `-f` option is not specified, the reasoning episode terminates either after `#steps` cycles or after the query has been answered, whichever happens first.

†Since SHRUTI-CM5 runs reasoning episodes asynchronously, this command does not support the `-f` and/or `#steps` arguments.

Reset Network Syntax: `z [ -q | -v ]`
Resets the network and removes all activation including the query. With the `-v` option, a message is printed out indicating that the network has been reset (default). The message can be suppressed by using the `-q` option.

Set Phases Syntax: `p [#phases]`
Sets the number of phases per clock cycle to `#phases`. The current number of phases is printed out if the command is invoked without an argument.

Display Syntax: `d { -p | -c } name`
Displays the current instantiations of the predicate (with the `-p` option) or concept (with the `-c` option) specified by `name`. An error message is printed if the named predicate or concept is not present in the system.

†Syntax: `d { -p name | -c name }*`
SHRUTI-CM5 supports multiple `-p` and/or `-c` options.

Statistics Syntax: `s [ -a | -k | -q | -c | -s ]`
Prints out knowledge base and reasoning episode statistics. When the system is configured for detailed statistics collection, this command will print out more information. The `-a` option prints out all the accumulated data (default). The `-k` option prints out information about the knowledge base. All details about the current reasoning episode are printed out by the `-q` option. The `-c` and `-s` options print out cumulative data and data from the last propagation step respectively, for the current query.

†Due to the asynchronous nature of the SHRUTI-CM5 system, a global propagation step is not well defined. Hence, SHRUTI-CM5 does not support the `-c` and `-s` options.

†**Display Tagged Activation** Syntax: `a -f first-tag [-l last-tag]`
Displays the number of active predicates and entities with tag values in the specified range. If the `-l` option is not specified, active predicates and entities with tag value equal to `first-tag` are printed.

†**Display Processor Load** Syntax: `l [ -a | -k | -q | -t ] [-n processor]`
Prints out the processor load for the current reasoning episode. When the system is configured for detailed statistics collection, this command will print out more information. The `-a` option prints out all information (default). The `-k` option prints out the distribution of the knowledge base on the processing elements. The distribution of active elements for the current reasoning episode are printed out by the `-q` option. The timing for individual processors (for the current reasoning episode) is displayed by the `-t` option. If the `-n` option is given, required information is displayed for the specified processor. If the `-n` option is not used, data is displayed for all processors in the partition.

References

Ajjanagadde, V. and Shastri, L. (1991). Rules and variables in neural nets. *Neural Computation*, 3:121–134.

Barnden, J. A. and Pollack, J. B., editor (1991). *Advances in Connectionist and Neural Computation Theory, Volume 1*. Ablex Publishing Corporation, Norwood, NJ.

Barnden, J. A. and Srinivas, K. (1991). Encoding techniques for complex information structures in connectionist systems. *Connection Science*, 3(3):269–315.

Carpenter, P. A. and Just, M. A. (1977). Reading comprehension as eyes see it. In Just, M. A. and Carpenter, P. A., editor, *Cognitive Processes in Comprehension*. Erlbaum.

Charniak, E. (1983). Passing markers: A theory of contextual inference in language comprehension. *Cognitive Science*, 7(3):171–190.

Dietz, P., Krizanc, D., Rajasekaran, S., and Shastri, L. (1993). A lower bound result for the common element problem and its implication for reflexive reasoning. Technical Report MS-CIS-93-73, Department of Computer and Information Science, University of Pennsylvania.

Evett, M. P., Andersen, W. A., and Hendler, J. A. (1993). Massively parallel support for efficient knowledge representation. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1325–1330.

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

Feldman, J. A. and Ballard, D. H. (1982). Connectionist models and their properties. *Cognitive Science*, 6(3):205–254.

Hillis, W. D. (1985). *The Connection Machine*. MIT Press, Cambridge, MA.

Jurafsky, D., Wooters, C., Tajchman, G., Segal, J., Stolcke, A., Fosler, E., and Morgan, N. (1994a). The Berkeley restaurant project. In *Proceedings of the International Conference on Speech and Language Processing*, Yokohama, Japan. To appear.

Jurafsky, D., Wooters, C., Tajchman, G., Segal, J., Stolcke, A., and Morgan, N. (1994b). Integrating advanced models of syntax, phonology, and accent/dialect with a speech recognizer. In *AAAI Workshop on Integrating Speech and Natural Language Processing*, Seattle. To appear.

Lakoff, G. and Johnson, M. (1980). *Metaphors We Live By*. University of Chicago Press, Chicago.

Lange, T. E. and Dyer, M. G. (1989). High-level inferencing in a connectionist network. *Connection Science*, 1(2):181–217.

Lenat, D. B., Guha, R. V., et al. (1990). CYC: Towards programs with common sense. *Communications of the ACM*, 33(8):30–49.

Mani, D. R. (1994). A mathematical analysis of message passing (MIMD) SHRUTI simulation systems. Technical report, University of Pennsylvania. Forthcoming.

Mani, D. R. and Shastri, L. (1993). Reflexive reasoning with multiple instantiation in a connectionist reasoning system with a type hierarchy. *Connection Science*, 5(3 & 4):205–242.

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D., Miller, K., and Tengi, R. (1990). Five papers on WordNet. Technical Report CSL-43, Princeton University. Revised March 1993.

Moldovan, D. I. (1989). RUBIC: A multiprocessor for rule-based systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(4):699–706.

Moldovan, D. I. (1993). *Parallel Processing: From Applications to Systems*. Morgan Kaufmann, San Mateo, CA.

Moldovan, D. I., Lee, W., Lin, C., and Chung, M. (1992). SNAP: Parallel processing applied to AI. *Computer*, 25(5):39–50.

Newell, A. (1992). Unified theories of cognition and the role of Soar. In Michon, J. A. and Akyürek, A., editor, *Soar: A Cognitive Architecture in Perspective*, pages 25–79. Kluwer Academic, Netherlands.

Porter, B., Lester, J., Murray, K., Pittman, K., Souther, A., Acker, L., and Jones, T. (1988). AI research in the context of a multifunctional knowledge base: The botany knowledge base project. Technical Report AI88-88, University of Texas.

Shastri, L. (1991). Why semantic networks? In Sowa, J. F., editors, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, San Mateo, CA.

Shastri, L. (1993). A computational model of tractable reasoning—taking inspiration from cognition. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*.

Shastri, L. and Ajjanagadde, V. (1993). From simple associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony. *Behavioral and Brain Sciences*, 16(3):417–494.

Sun, R. (1992). On variable binding in connectionist networks. *Connection Science*, 4(2):93–124.

TMC (1991a). Connection machine CM-200 technical summary. Technical Report CMD-TS200, Thinking Machines Corporation, Cambridge, MA.

TMC (1991b). Connection machine CM-5 technical summary. Technical Report CMD-TS5, Thinking Machines Corporation, Cambridge, MA.

TMC (1993). *CMMD Reference Manual. Version 3.0*. Thinking Machines Corporation, Cambridge, MA.

TMC (1994). *CM-5 User's Guide. CMost Version 7.3*. Thinking Machines Corporation, Cambridge, MA.

von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. (1992). Active messages: A mechanism for integrated communication and computation. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*. ACM Press.

Waltz, D. L. and Pollack, J. B. (1985). Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science*, 9(1):51–74.