TERRESA: A Task-Based Message-Driven Parallel Semantic Network System

by

Chain-Wu Lee

January 25, 1999

A dissertation submitted to the
Faculty of the Graduate School of State
University of New York at Buffalo
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Abstract

The ultimate goal of research in artificial intelligence is to build a system with human intelligence. Yet, despite considerable effort, artificial intelligence systems are still a long way from that goal. Various reasons attribute to why the goal cannot be achieved at current stage, one of the reasons is the vast amount of information to be processed. When presented with real world domains of millions of concepts and assertions, almost all existing systems are unable to deliver the results within a reasonable amount of time.

This dissertation describes a parallel semantic network system, *TERRESA*, based on SNePS. TERRESA is a task-based, message-driven, multithreaded parallel semantic network system with the support of shared knowledge, load balancing and duplicate checking on MIMD distributed memory/message passing systems.

TERRESA can be split into two collaborating components: the host module and the slave module. The host module interacts with the user and processes the information for the slaves, while the slave module performs task execution. The current version of TERRESA focuses on SNePS path-based inference only.

TERRESA is designed to be portable among different parallel machines, and therefore most of the system dependent functions are either avoided or re-implemented. TERRESA

is written in ANSI C and MPI (Message Passing Interface) library with the help of *flex* lexical analyzer and *yacc* parser generator. Porting TERRESA to a new platform only requires recompiling in most cases. The performance studies of TERRESA have been done on SUN multiprocessor server.

Acknowledgements

First of all, I would like to express my greatest gratitude to my advisor Dr. Min-You Wu, who encouraged and advised me during my study. Without his effort, this dissertation will not exist. And I would also like to thank my committee members Dr. Bharadwaj Jayaraman and Dr. Stuart C. Shapiro for their patience and valuable suggestions regarding this research.

Many thanks to my friends in Buffalo, United States and Taiwan, without their warm friendship, I can not possibly finish my lengthy study in this cold place. Finally I would like to thank my parents for their support and tolerance, their son does not disappoint them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ultimate goal of research in artificial intelligence (AI) is to build a system with human intelligence. Yet, despite considerable effort, the current artificially intelligence systems are still a long way from this goal. Various reasons attribute to why the goal cannot be achieved at current stage, one of the reasons is the vast amount of information to be processed. For example, in CYC, about 10^6 commonsense axioms have been entered into the knowledge base and about a million more have been inferred and cached by the system [39]. To many AI systems, it is not because the theories and formalisms of AI research are not working; actually many of them are very impressive in the domain of small to medium size, but that when the system encounter real world domain of huge number of concepts and assertions, almost all existing systems are unable to deliver the results within a reasonable amount of time.

With the emergence of parallel computers, massive parallel artificial intelligence is bringing us closer to realizing this grand challenge. The notion of parallel computing is quite intuitive: “In a parallel computer, a particular computation is divided among the processors, which complete the job much faster than a single CPU laboring in isolation” [36]. Though many disagree, many researchers believe that parallel computing may be the most powerful weapon in future AI research [37]. One of the most prominent examples is the IBM Deep Blue, which used 32 nodes IBM SP-2 parallel machine, defeated the Chess Grand Master Gary Karparov in a chess tournament in 1997.

However, parallel computers are not silver bullets. The merely existence of parallel computers do not award us a guarantee of a faster vehicle to speed up the traditional AI models [37]. As a matter of fact, like many other domains, many of the AI systems are designed upon the current sequential Von Neumann model - which is by no means suitable for parallel computers. When porting the system into parallel machines, many traditional AI systems need to be re-modeled, re-engineered or most even create new algorithms to make them applicable. In addition to designing and modeling, lots of other issues, such as load balancing, knowledge access and task duplication, surface when the system runs in parallel and need to be taken into consideration when designing a new parallel AI system.

From the perspective of parallel computing, artificial intelligence problems are particularly interesting to the people in the parallel computing community because of the challenging properties of the problems. Comparing to the scientific computing realm which uses the parallel computers most, AI problems are relatively fine grained, the data structures are often irregular and dynamic, the communication demands are higher, and the communication patterns are random. All of these factors present obstacles to planning efficient parallel programs.

Although the exploitation of parallelism in the AI field relies highly on the target system, generally speaking there are several frequently adopted approaches:

1. Parallel extension of traditional AI languages: Numerous approaches have been taken to parallelize the traditional AI languages (lisp, scheme and prolog). Some languages, which have augmentation with parallel programming construct, are designed to be executed in parallel. The objective languages include Lisp variants: **Lisp* [66, 67], *Multilisp* [35], *QLISP* [21], *PScheme* [69] and PROLOG variants: *Strand* [19], *Concurrent PROLOG* [53], *PARLOG* [10].

   Some research teams have designed interfaces for AI languages which can be linked with the existing parallel libraries. For example, *CL-PVM* [43] provides an interface for Lisp-based programs to link with the *PVM*(Parallel Virtual Machine) [23] library. Others provided a automation tool to parallelize existing code automatically, such as *Auto Parallel Lisp* [20].

2. Hardware Solutions: In the history of parallel AI research, various hardwares had been devised for speeding up specific applications. Roughly we can categorize them into two groups: the dedicated hardware/machines and the general purpose machines with AI goals in mind. The examples of the former are *SNAP* [14], which used dedicated hardware to increase the performance of the semantic network inference. *SPHINX* [42], which used the hardware implementation to boost the performance of speech recognition; Deep Thought [33] / Deep Blue [65, 34], which used a custom design chip to play chess.

The examples of general purpose machines are the Thinking Machine Corporation CM-2 [32] and IXM [31]/IXM-2 [30] by ElectroTechnical Laboratory. IXM/IXM-2 used associate memories and transputers as their construction building blocks. The applications range from genetic algorithms to knowledge base searches.

The thinking machine CM-2 is a SIMD massively parallel computer. One of the primary goals of CM-2 is to build a “symbolic crunching” machine. As the Connection Machine chief designer said in [32]:

> “ … retrieving common sense knowledge from a semantic network was one of the primary motivations for the design of the Connection Machine.”

There are advantages as well as disadvantages of adopting hardware instead of software approaches. Their performance are generally better than their software counterparts and hardware can be tailor-made so that it can be applicable to specific requirement for the specific question. An example is the Deep Blue, which the research team claimed that the function performed on one of their ad hoc chips is equivalent to a 10,000 MIPS general purpose computer [34]. Yet the lack of extensiveness and flexibility makes the machine very difficult to upgrade. As for the hardware itself, according to Moore’s law [47], the transistor capacity doubles every 18-24 months, and the constant and rapid changing hardware technologies make the existing hardware superiority obsolete in a relatively short period of time.

3. Parallel AI algorithms with general purpose parallel computers as the implementation platforms: This scheme is the most widely embraced approach. This strategy

is used in the dissertation.

When implementing a system with parallel computers, there are some issues that the designers have to address:

(a) Target Machine: What kind of machine does the program intend to work on? SIMD or MIMD? Massively parallel machines or network workstations? Shared memory machine or distributed memory machines?

(b) Parallelism: Is the existing sequential algorithm ready for parallel execution? If not, how can the algorithm be modified so that it can work concurrently? How can the parallel tasks be partitioned among processors or processing elements (PEs)? data parallelism, task parallelism or mixed strategy?

(c) Efficiency: After finishing the construction of the system, what kind of modification can you make on the system so that it can run faster? Better load balancing strategy? Overlapping computation with communication? Multithreading? Approximation algorithm? And what is your expected speedup?

In this dissertation, a parallel semantic network system, *TERRESA*[1], is proposed. The name *TERRESA* comes from the abbreviation **T**ask-bas**E**d, message-d**R**iven pa**R**all**E**l **S**em**A**ntic network system. It is a multithread system with the support of shared knowledge, dynamic load balancing and duplicate checking on MIMD distributed memory/message passing machines.

[1]TERRESA is former known as *PSNePS* in the previous papers. [40, 41]

TERRESA can be split into two collaborating components: the *host* module and the *slave* module. The host module interacts with the user and processes the information before it can be routed to the slave modules, while the slave modules perform task execution. When executing TERRESA, only one copy of the host module runs on the parallel machine in contrast to the multiple copies of the slave module. There are several sub-modules in both the host and the slave module.

TERRESA is designed to be portable among different parallel machines, and therefore most of the system dependent functions are either avoided, re-implemented or written according to the target platform. Currently, TERRESA is written in ANSI C and the *MPI (Message Passing Interface)* library with the help of the *flex* lexical analyzer and the *yacc* parser generator. Porting TERRESA to new platforms require nothing more than recompiling in most cases. Although SNePS [60, 54, 56, 57] supports two inference schemes, path-based inference and node-based inference, the current version of TERRESA focuses only on the path-based inference.

This dissertation is outlined as follows: Chapter 2 provides background information on the semantic network, and describes parallel computing and previous research on this topic. Chapter 3 covers some issues of designing the TERRESA system, as well as a summary of the architecture and building blocks of the system. Chapter 4 depicts the host module in the system while Chapter 5 through Chapter 8 present various sub-modules in the slave module. Chapter 9 describes our performance experiments, and Chapter 10 concludes the dissertation.

Chapter 2

Background

In this chapter, background information of the research will be presented. First, a brief introduction to *semantic networks* and the target system - *SNePS*, is offered. In Section 2.2, some background of parallel computation is given. Then we will take a look at previous research in this field, specifically, two parallel semantic network systems, the *SNAP* (Semantic Network Array Processor) and *PARKA* (PARallel Knowledge and Association). Since these two parallel semantic network systems use the the *Marker-Passing Algorithm* as their inference mechanism, the algorithm is introduced in Section 2.3.1. Our point of view toward the drawbacks of the marker-passing algorithm, as well as suggestions for a different model for solving the problem from a different perspectives are presented in the last section.

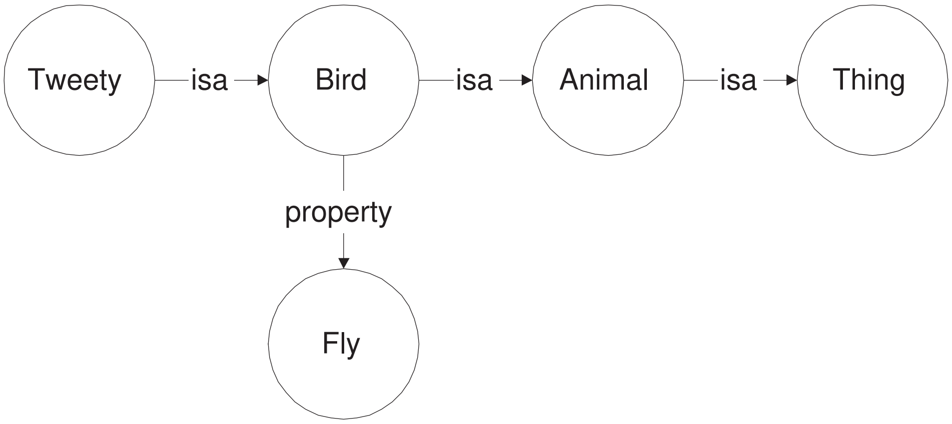

Figure 2.1: A Semantic Network Example

2.1 Semantic Networks and SNePS

The semantic network is a conceptual model for representing knowledge. The notion of the semantic network was proposed by R. Quillian in 1968 [50]. In the semantic network, the knowledge entities are represented by the nodes (or vertices), while the edges (or arcs) are the relations between entities. Clustered network represents the collective concepts related to each other. Semantic networks are used by various AI applications as the backbone knowledge representation system including natural language understanding, machine translation, etc. Fig. 2.1 illustrates a simple example of a common semantic network. In the figure, it indicates that Tweety is a bird, birds can fly, they are animals and an animal is a thing.

Several semantic network systems have been developed by various research groups, including ECO [6], KL-ONE [5], NETL [16], and SNePS [56]. Each of these systems has its distinctive features. SNePS distinguishes itself from other systems by incorporating propositional rules into the network. SNePS, *The Semantic Network Processing Sys-*

tem [57], was designed by Dr. Stuart C. Shapiro in the early 70s and has undergone several revisions. The current version is SNePS 2.4 and in moving toward version 3.0. The system is implemented in common lisp and runs on various machines. There are several auxiliary systems like SNeBR, the SNePS belief revision system which maintains the consistency of the knowledge base [56]; the SNeRE, the SNePS rational engine; the natural language processing system SNaLPS [57] and other various systems as well.

Before any new knowledge is derived, the semantic network must be constructed before inference can be performed. There are two inference schemes in the SNePS system, namely *path-based inference* and *node-based inference*. In the following paragraph, we first introduce how the network is built, and then the two inference schemes are briefly examined.

2.1.1 Building the Network

SNePS provides three commands to construct the semantic network, `build`, `assert` and `add`. The syntax of these commands are listed below:

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

These commands put a node into the network with an arc labelled *relation* to each node in the following *nodeset*, and returns the built node. If this node is trying to build a node exactly like an already existing one, no new node is built and instead the existing node is returned. `build` creates an unasserted node unless an asserted node exists in the network with a superset of the wires of the new node, in which case the new node is also asserted. `assert` is just like `build`, but creates the node with assertion. `add` acts like `assert`, but also it triggers forward inference. Neither a converse relation nor non unit-path [1] is allowed to put into the position *relation*. When building the network, the converse relation *relation-*, which connects each node of the nodeset to the built node, is constructed implicitly by the system.

In the TERESSA implementation, the command `build` and `assert` are supported except that the parameter *context-specifier* is dropped.

2.1.2 Path-Based Inference

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

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

defines the concept of "Socrates is a human", which is shown in Fig. 2.2. In the system, two base nodes `Socrates` and `human` are generated by the command. The molecular node `M1`, which might have different number depending on the current state of the knowledge base, is generated by system to represent the entire concept. The notation "!"

[1]The definition of unit-path can be seen on page 12

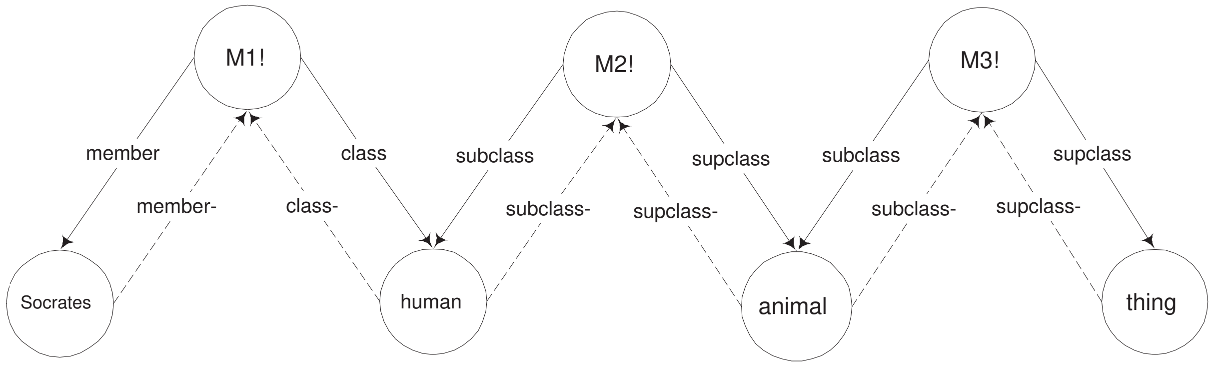

Figure 2.2: SNePS Hierarchy Example

stands for an asserted concept. Two forward links `member` and `class` are defined by the user, two reverse links `member-` and `class-`, indicated by dash lines, are generated by SNePS automatically. Likewise, hierarchical concepts "human is an animal" as well as "animal is a thing" can be constructed in the similar way. And by following the links of "subclass-" and "supclass", derivation can be made that Socrates is human, an animal and a thing.

In SNePS, the relation of two nodes can be either explicit (direct arc between two nodes), or implicit (an arc across several intermediate nodes, or in SNePS terms, subconscious relation [55]). The implicit relation is defined by the SNePS command `define-path`. For example, in Fig. 2.2, the `class` inference rule can be defined by the following command:

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

which indicates the `class` relation can be defined by a direct arc `class` followed by zero or more occurences of combinations of the direct arcs `subclass-` and `supclass`.

Several path definition primitives can be defined in the path-based inference rule. The following rules describe what has been implemented so far in TERESSA. This information will be used in Chapter 5 for defining the parallel computation task units in the system. All the definitions are excerpted from the SNePS manual [57] page 11 to page 12:

- *unitpath ::= relation*.
- *unitpath ::= relation-*: A unit path can be either a relation or a converse of a relation.
- *path ::= unitpath*: A path can be either a unitpath or the composition of various pathes defined by the following definition.
- *path ::= (*`COMPOSE` *{path}*)*

  If $x_1, \ldots x_n$ are nodes and P_i is a path from x_1 to x_{i+1}, then (`COMPOSE` $P_1 \ldots P_{n-1}$) is a path from x_1 to x_n.
- *path ::= (*`KSTAR` *path)*

  If path P is composed with itself zero or more times from node x to node y, then (`KSTAR` P) is a path from x to y.
- *path ::= (*`KPLUS` *path)*

  If path P is a composed with itself one or more times from node x to node y, then (`KPLUS` P) is a path from x to y.
- *path ::= (*`OR` *{path}*)*

  If P_1 is path from node x to node y or P_2 is a path from x to y or $\ldots$ or P_n is a path from x to y then (`OR` $P_1, P_2 \ldots P_n$) is a path from x to y.

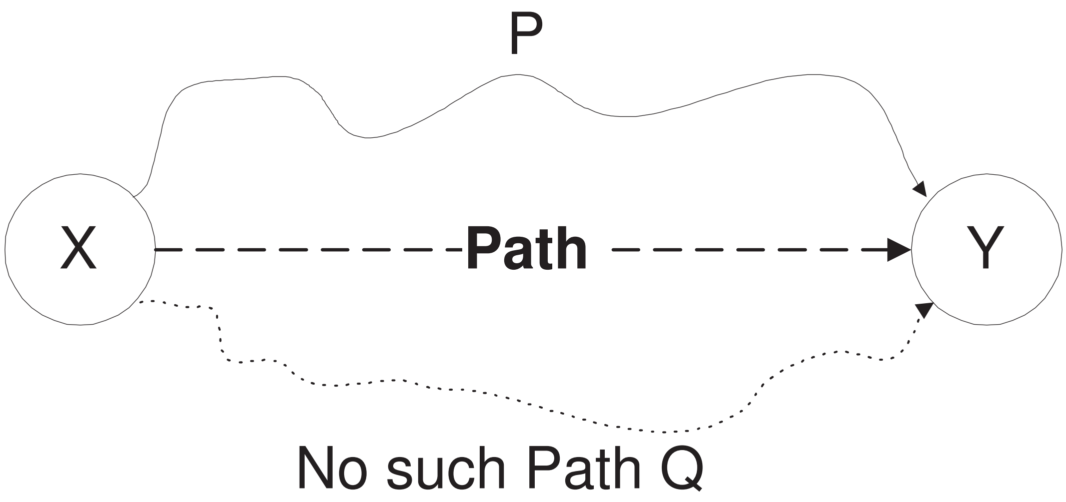

Figure 2.3: (`RELATIVE-COMPLEMENT` P Q)

- *path ::=* (`AND` $\{path\}^*$)

  If P_1 is path from node x to node y and P_2 is a path from x to y and $\ldots$ and P_n is a path from x to y then (`AND` $P_1, P_2 \ldots P_n$) is a path from x to y.

- *path ::=* (`NOT` *path)*

  If there is no path P from node x to node y, then (`NOT` P) is a path from x to y.

- *path ::=* (`RELATIVE-COMPLEMENT` *path path)*

  If P is a path from node x to node y and there is no path Q from x to y, then *(*`RELATIVE-COMPLEMENT` *P Q)* is a path from x to y. The situation can be seen in Fig. 2.3.

- *path ::=* *(*`IRREFLEXIVE-RESTRICT` *path)*

  If P is a path from node x to node y, and $x \neq y$, then *(*`IRREFLEXIVE-RESTRICT` *P)* is a path from x to y. The situation can be seen in Fig. 2.4.

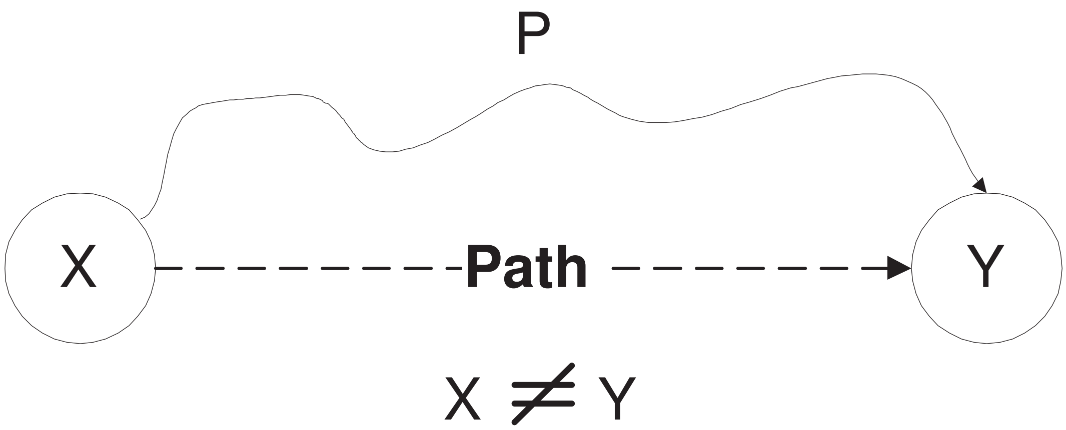

Figure 2.4: (IRREFLEXIVE-RESTRICT P)

- *path ::=* (DOMAIN-RESTRICT *(path node) path)*

  If P is a path from node x to node y and Q is a path from x to node z, then (DOMAIN-RESTRICT *(Q z) P)* is a path from x to y. The situation can be seen in Fig. 2.5.

- *path ::=* (RANGE-RESTRICT *path (node path))*

  If P is a path from node x to node y and Q is a path from y to node z, then (RANGE-RESTRICT *P (Q z))* is a path from x to y. The situation can be seen in Fig. 2.6.

- *path ::= (path*)*

  The definition is the same as (COMPOSE $\{path\}^*$)

The SNePS command `find` is specifically for the path-based inference query. Excerpted from SNePS manual [57] page 15 to page 16, the format of the `find` command is list below:

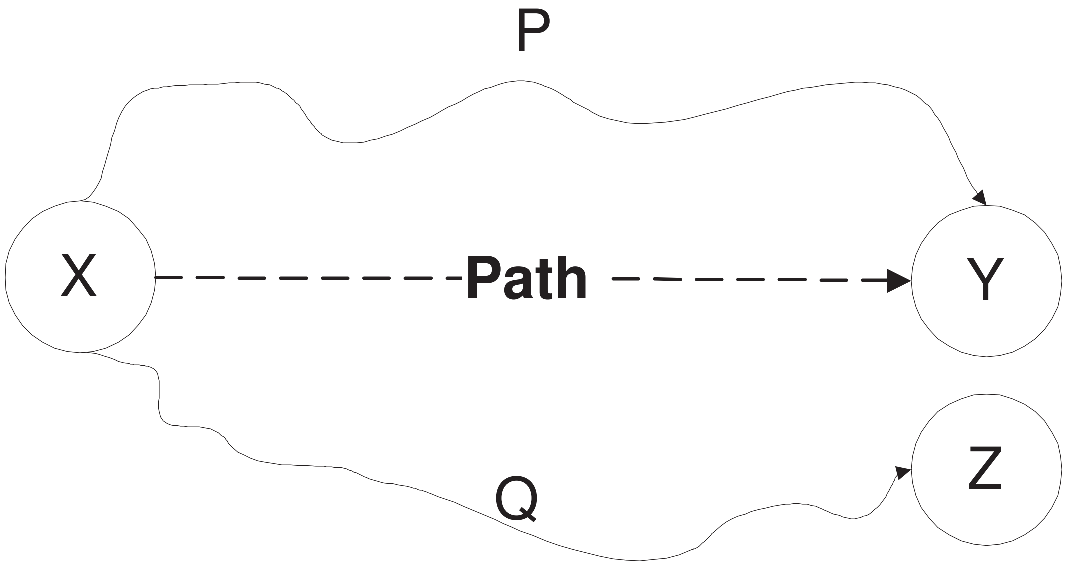

Figure 2.5: (`DOMAIN-RESTRICT` (Q z) P)

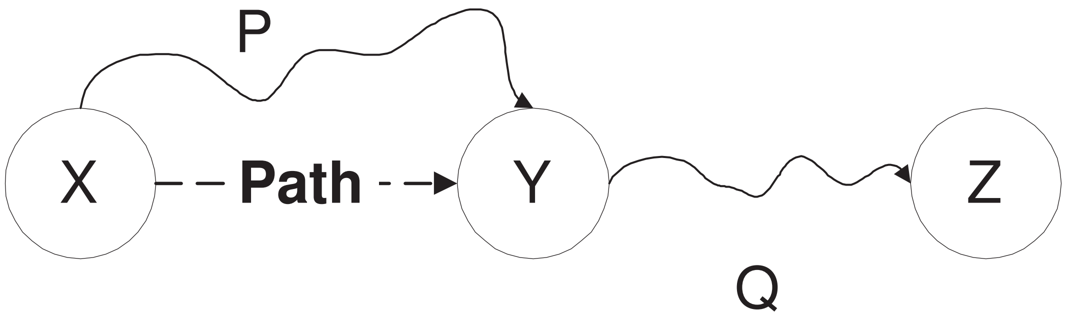

Figure 2.6: (`RANGE-RESTRICT` P (Q z))

$$(\texttt{find}\ \{\textit{path nodeset}\}^*)$$ [2]

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

For example, in Fig. 2.2, When the command

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

is issued, the answer is `M2` because only the node `M2` has edge `subclass` to node `human` *and* edge `supclass` to `animal`.

When the command

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

is issued, the answer is (`M2 M3`) because `M2` and `M3` has an edge `subclass` to *either* node `human` *or* `animal`.

The command `find` does not generate new knowledge (new nodes or new arcs) while running.

2.1.3 Node-Based Inference

The node-based inference of SNePS [17, 7] is similar to rule inference in rule base system. SNePS accepts proposition rules and the system performs automatic deduction on demand. In order to integrate rules into the knowledge base, rules are constructed in the

[2]SNePS `find` command has an optional parameter *context-specifier*. Context is not supported in TERESSA.

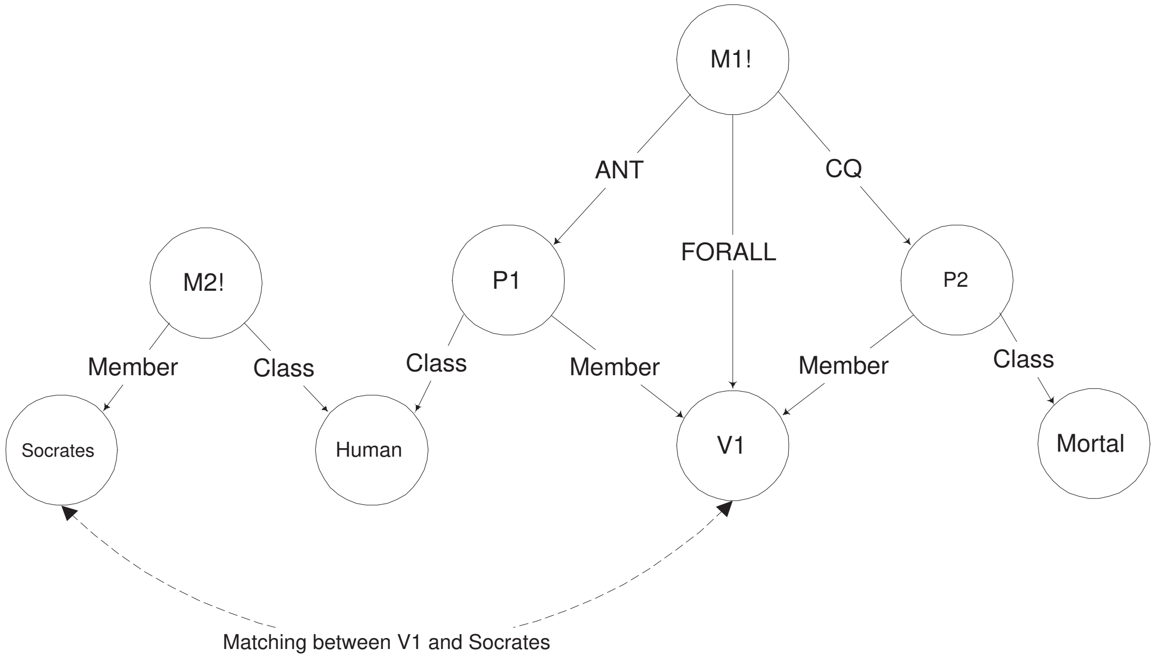

Figure 2.7: SNePS example of Socrates is human and all human are mortal

form of networks. As illustrated in Fig. 2.7, the rule describes the concept of “all humans are mortal” and “Socrates is a human”, which can be defined by the SNePS command

```
(assert forall $x
        &ant (build member *x class human)
        cq   (build member *x class mortal)))
```

and

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

The command `deduce` invokes backward node-based inference in the SNePS. When `deduce` is issued, it first generates a SNePS user process which contains the query question and tries to match it with either an existing knowledge or the consequent of any rule. If the existing knowledge provides the answer, a report is sent back to the user process, which prints out the result and then terminates. Otherwise, the system will trace the antecedents of the rules to see whether the antecedents of the rules are supported by any existing knowledge. The procedure executes repeatedly until the answer is found or all the rules have been consumed. When an answer is found, the report is sent back by the query channel in the same manner but in reverse direction.

SNePS provides a pseudo multi-process environment internally. The inference is implemented with two queues, a high-priority queue and a low-priority queue. When a query is issued, the query process is moved into the high-priority queue first. The computer picks up a process from the high-priority queue and tries to solve it. If additional information is needed, the process is then moved to the low-priority queue for further notification. When the answer is found, the system will generate a report process and then put it into the high priority queue for processing. In SNePS implementation, report processes are always more important than question processes and need to be processed first.

2.2 Parallel Computations

This section provides the information related to parallel computations: The hardware platforms and software libraries.

2.2.1 Parallel Platforms

Michael Flynn [18] proposed a classification of various computer architectures based on the number of instructions and data streams. In his classification, computers can be categorized into four types: single instruction single data (SISD) machines, single instruction multiple data (SIMD) machines, multiple instructions single data (MISD) machines and multiple instructions multiple data (MIMD) machines. From the parallel computing perspective, only SIMD and MIMD computers interest us, hence we will go through the SIMD and MIMD architectures in the following paragraphs.

SIMD Machines

The basic SIMD architecture can be seen in the Fig. 2.8. A typical SIMD machine is composed of several building blocks: the *control unit*, the *PEs*, the *local memories* and the *interconnection network* of different topologies depending on the designers. During execution, the control unit sends commands to each PE and executes the sequential portion of the program. Every PE simultaneously executes the same command sent by the control unit. The interconnection network is responsible for the communication between PEs if information or data exchange is needed. Depending on the data, the PE may be idle (masked) if the condition not being met. The example of the commercially available SIMD machines are the MP-1 from MasPar Computer Corporation [44] and the CM-2 from Thinking Machines Corporation [32, 68].

More often SIMD machines are used for executing *data-parallel* instead of *procedure parallel* problems. Usually the SIMD machine is composed of mass quantities of relatively less powerful PEs (64K 1-bit processors of CM-2 and 16K 4-bit processors of MP-1). By combining massive quantities of PEs, SIMD machines can achieve the high performance of a giga instructions per second.

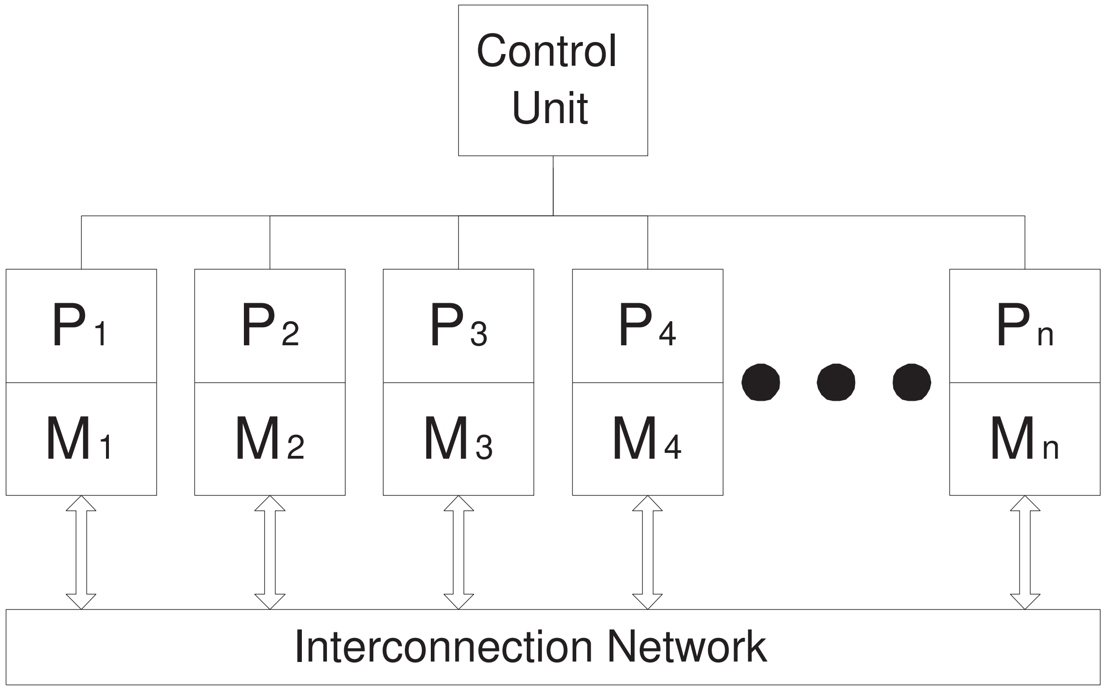

Figure 2.8: Typical SIMD Architecture

MIMD Machines

MIMD machines are the most popular class among the parallel computer clan these days. Unlike SIMD machines where each PE gets its command from the central controller, the PEs of MIMD computers get the command from the program stored locally in the main memory and try to execute them in collaboration. MIMD machines can be further divided into distributed memory machines and shared memory machines according to the memory location and its access privileges.

1. Distributed Memory Machines: In distributed memory machines, as shown in Fig. 2.9, each memory module is attached to a PE. The PE can access the local memory without going through the interconnection network, yet message exchange must be made if remote memory access is necessary. Usually it can be done by calling `send` at one

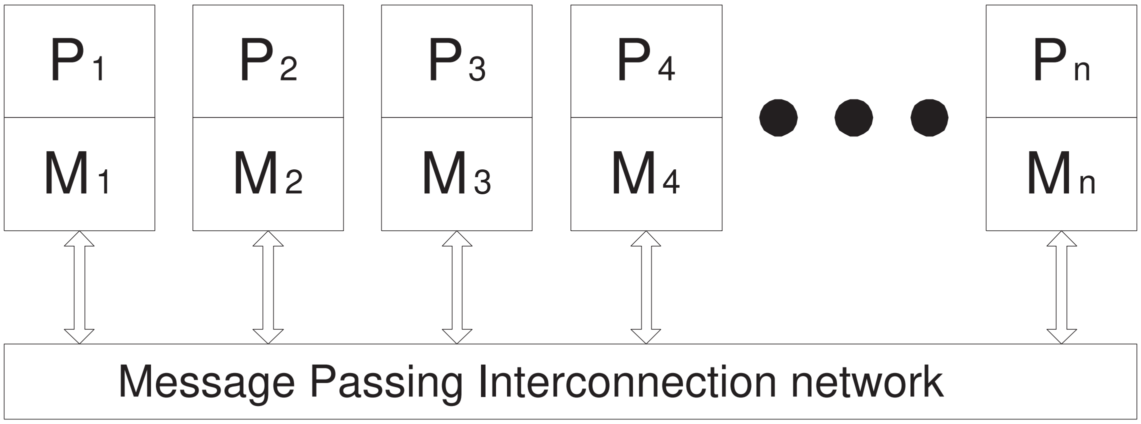

Figure 2.9: MIMD Distribute Memory Computers Architecture

PE and `receive` at the other PE. Because the communication is far more costly than accessing the local memory, the communication bandwidth is one of the crucial indicators of how good the MIMD parallel computers can perform. The raw communication bandwidth of the commercial available parallel computers range from 50 MB/sec up to 200 MB/sec. Examples of these machines include commercially available Intel series (iPSC-2, 860, Delta and Paragon), nCube Machines and Thinking Machine Corp. CM-5[3].

2. Shared Memory Machines: We describe the two most frequently seen shared memory mode, the *uniform memory access* (UMA) architecture and *non-uniform memory access* (NUMA) architecture below.

    (a) UMA Model: The UMA model can be seen in Fig. 2.10. In the UMA model, the physical memories are shared by all the processors via some kind of interconnection network (could be either bus, crossbar switch or multistage net-

[3]CM-5 is backward-compatible with CM-2 by providing SIMD mode.

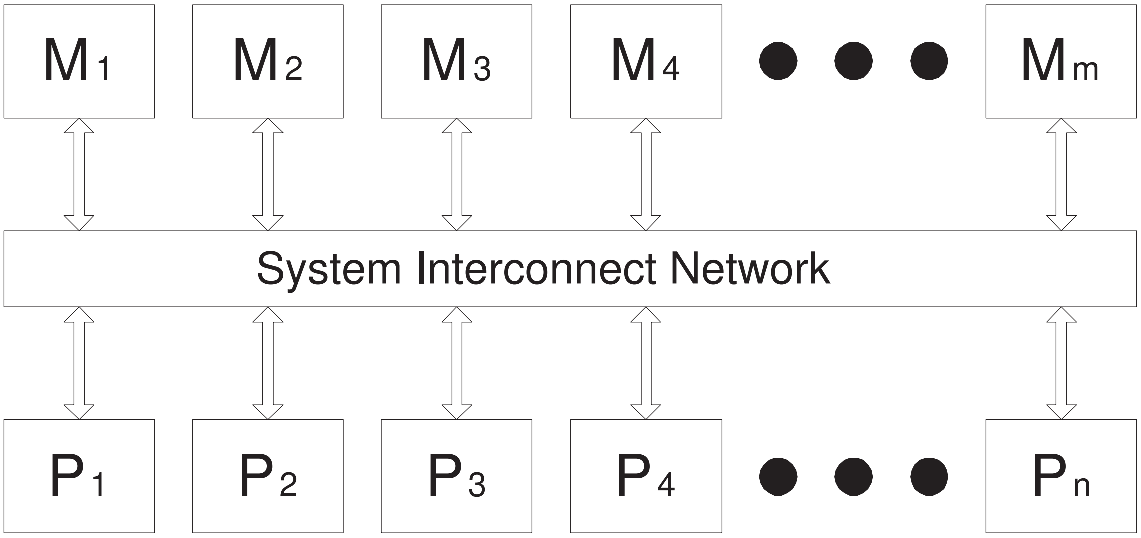

Figure 2.10: MIMD Shared Memory Computers, UMA (Uniform Memory Access) Architecture

work). It is the case that all processors have equal access time, so we call this model the *uniform model access* model. Examples of the machine are the Sequent Symmetry S-81 and the Symmetry MultiProcessors (SMP) PC. Because the interconnection network is the most widely demanded resource of the computer, the number of PEs are usually relatively small compared to other models.

(b) NUMA Model: The NUMA model is depicted in Fig 2.11. Although the memory modules are distributed among PEs just as for the MIMD distributed memory machines, the memory modules all use a shared memory address. In this case, it is faster to access local memory than remote memory, and therefore it is called the *non uniform memory access* model. BBN TC-200 [3], Cray T3-D/E [11] and some of the shared-memory workstations belong to this model.

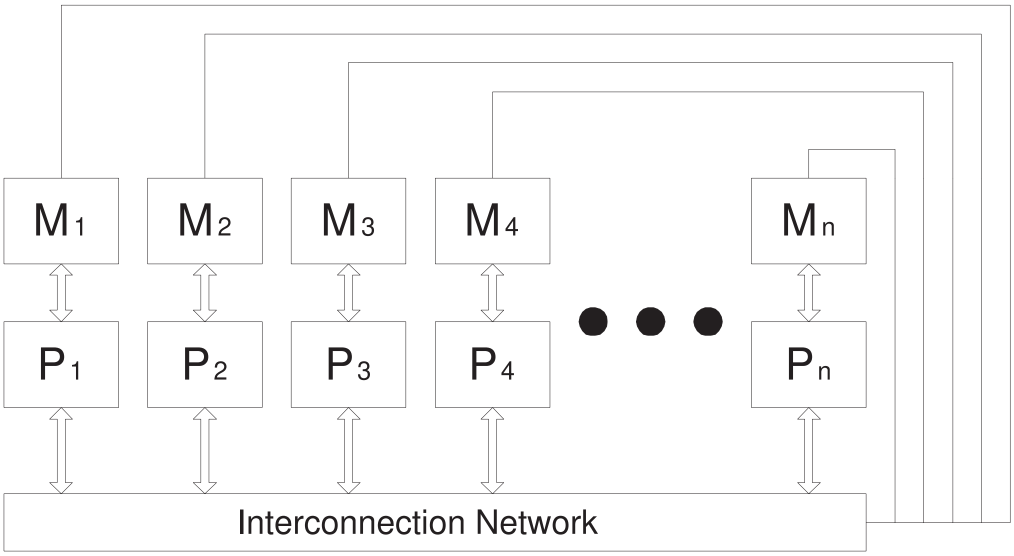

Figure 2.11: MIMD Shared Memory Computers, NUMA(Non Uniform Memory Access) Architecture

2.2.2 MPI - Message Passing Interfaces

There are various approaches to writing a parallel program. In our implementation, the message passing model is used. Among many message passing libraries, the MPI (Message Passing Interface) is used as the underneath communication primitive.

Ever since the dawn of parallel computing, message passing has been one of the most used schemes in writing parallel programs specially for distributed memory machines. Yet the different libraries provided by different vendors make the programs hard to port among different parallel machines. In 1992, a forum was formed to standardize the message passing interfaces, which resulted in MPI [45, 27].

MPI provides a wealth of function calls for the different needs of applications. Since the MPI forum only defined the standard instead of making the implementation, various breeds of MPIs have been developed. For example, the public implementations include

MPICH [25], LAM [48], CHIMP [1], MPICH-NT [49] as well as the vendor implementations - HP and SGI.

In our implementation, we choose MPICH as the underneath function. MPICH is available for several parallel platforms, including massively parallel computers (Cray T3D, Intel machines), shared memory machines (mulit processors servers of SUN, SGI and HP) and even PCs (under Window NT, Free BSD and Linux). Since the API (Application Program Interface) is standardized, it can be ported to other MPI implementations seamlessly.

2.3 Related Research

In this section, we review related research that done by other research teams. More specifically, *SNAP* and *PARKA*. Because both of them are using the marker passing algorithm as the inference algorithm, we try to explain the algorithm in Section 2.3.1.

2.3.1 Marker Passing Algorithm

The marker passing (or marker propagation) algorithm is broadly used in various semantic network systems. It is a technique for finding connections or relations between nodes in the network. A marker is a data structure indicating the nodes we are interested in. It can be activated, passed to other nodes, modified, operated on with other markers or eliminated. Fig. 2.12, taken from article [29] with minor modification, is an example of marker passing.

Fig. 2.12 implies the scenario of “John is in a restaurant, reads the menu and makes an

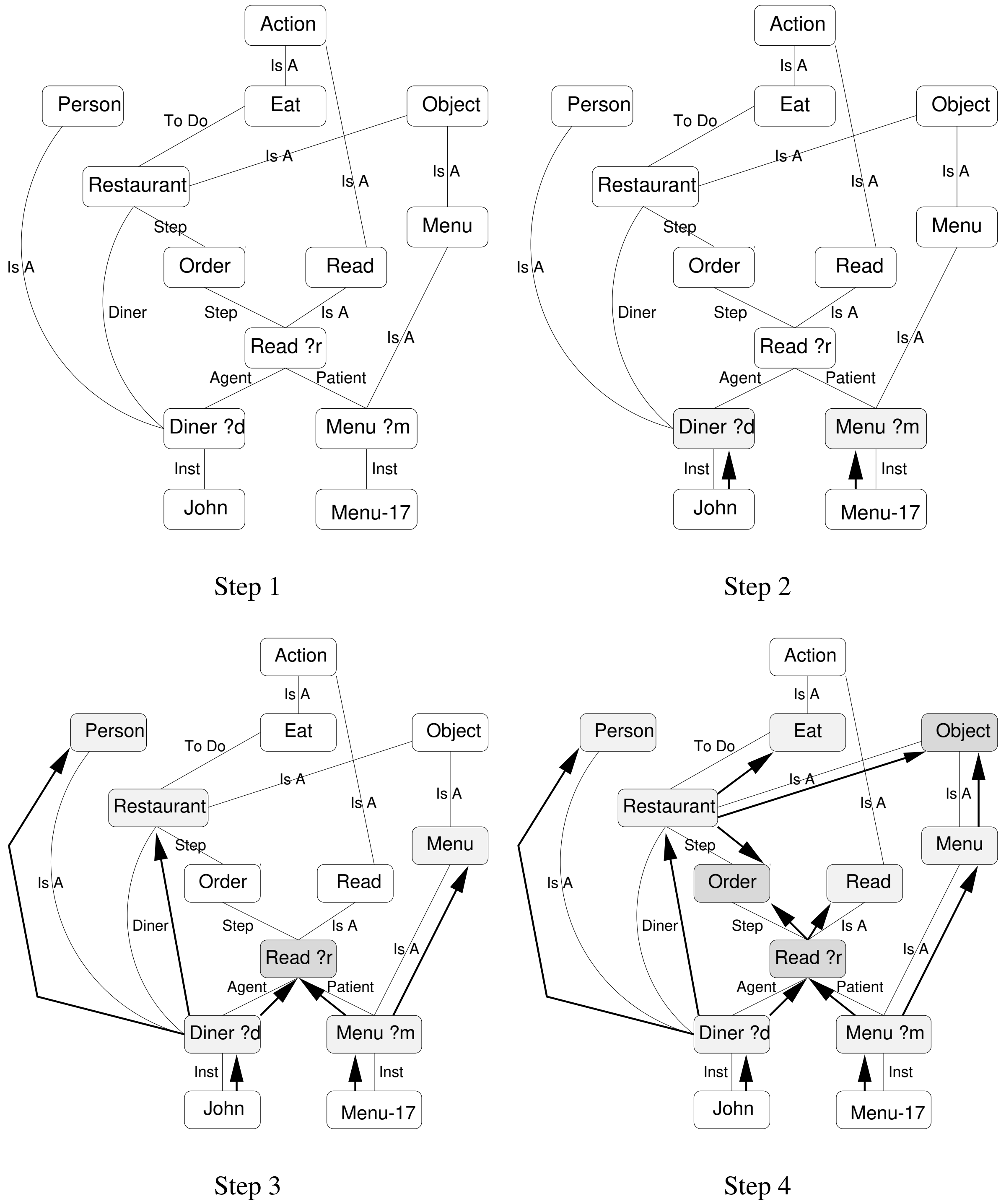

Figure 2.12: The process of marker passing in the semantic network, based on the figure in [29]

order". In the beginning, we install markers into the node "John" and "Menu-17". Then the markers are passed along all links (the dark lines) from these nodes to others. In step 2, the nodes "Diner ?d" and "Menu ?m" accept the marker from the origin and store them inside the nodes (represented by light-gray circles).

The algorithm keeps passing the markers to their neighbors. In step 3, a common node "Read ?r" is found (marked by the deep-gray shadow), which indicates the node can be reached from both origins "John" and "Menu-17".

The marker-passing algorithm will continue to pass the marker until all the paths are checked, which might need exponential time to finish [29]. Two mechanisms are designed to avoid this problem. The first approach is instead of trying to broadcast the markers along all the arcs, several constraints are set to narrowing down the selected paths. The second approach is to try to design a mechanism to "attenuate" the markers, by stopping the markers after certain links have been traversed, for example.

The marker passing algorithm is inherently parallel. Markers can be passed within the network without interfering of other markers.

2.3.2 SNAP, Semantic Network Array Processor

Moldovan and his research team implemented the system called SNAP, Semantic Network Array Processor [14, 46, 8, 9]. SNAP, a marker-passing semantic network, was implemented on several different platforms including the Connection Machine CM-2 [9] and a dedicated parallel machine using the DSP chip TI-TMS320C30 for its processors [14]. SNAP also acted as the base of a natural language processing systems [8]. Since the team

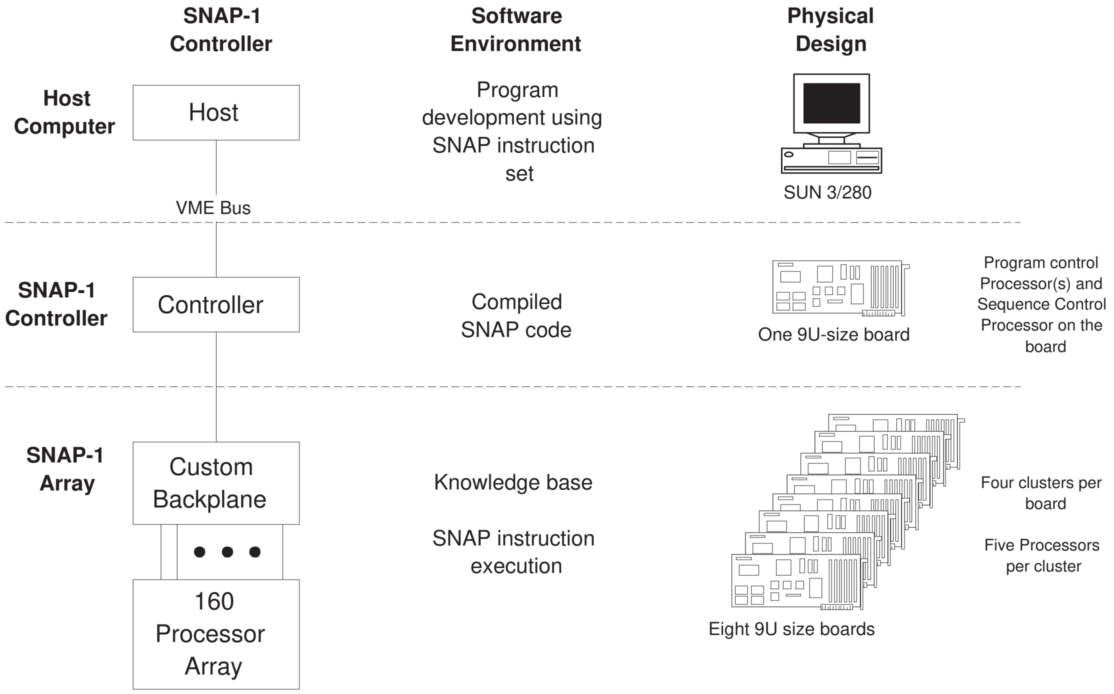

Figure 2.13: SNAP-1 Architecture, excerpted from [14]

concluded that the performance can be achieved only by designing a specialized architecture instead of using general parallel computers, we will concentrate on the dedicated machine implementation, SNAP-1, only.

The architecture of SNAP-1 is shown in Fig. 2.13. SNAP-1 is composed of a host machine SUN 4/280, a multiprocessor array of 144 Texas Instruments TMS320C30 DSP microprocessors and a controller as the control bridge between the host and the PEs. The array is organized as 32 clusters where each cluster handles 1024 semantic network nodes and 10240 links. The SNAP program is written and compiled on the host SUN machine and then downloaded to the controller for execution. The serial portions of the program are executed on the controller. When parallel execution is needed, the controller broadcasts the SNAP instructions to the PEs where the commands are executed simultaneously. Once running on the PEs, markers travel through the networks without the intervention of the controller.

The network structure of SNAP-1 is similar to the original network proposed by Quillian. It has two types of knowledge: permanent knowledge and temporary knowledge. Permanent knowledge is represented by the semantic networks and temporary knowledge is represented by the markers. The reasoning algorithm (marker passing algorithm) is achieved by changing the states of both temporary and permanent knowledge.

Two kinds of parallelism that can be utilized: the *intra-propagation parallelism*, or the *α-parallelism* and the *inter-propagation-parallelism* or the *β-parallelism*. α-parallelism tries to pass the markers in parallel among a single `PROPAGATION` statement, the parallelism implied by the knowledge ontology. β-parallelism tries to execute two or more `PROPAGATION` statements if they are not interfering with each other, or the parallelism implied by the program. The experiments indicate that with $128 \leq \alpha \leq 512$, SNAP-1 can achieve speedup ranges from 18-fold to 33-fold in a 72 processor configuration and with little impact on the performance if $\beta > 16$.

2.3.3 PARKA - PARallel Knowledge and Association

PARKA, designed by M. Evett and J. Hendler and their colleagues, is a semantic network based knowledge representation system on the SIMD Connection Machine CM-2 [15]. Recently the research group has ported PARKA into CM-5 MIMD machines by using the CHAOS parallel library [61].

Knowledge in PARKA is in the form of a general semantic network and the authors concentrated on the parallel inferencing of the inheritance properties, i.e. `is-a` and `has-property` links among the networks.

The inferencing mechanism in PARKA is called *activation wave propagation*, which is similar to the marker-passing algorithm. At the beginning, the system installs an activation seed into the starting nodes, and then the activation nodes start sending *activation waves* to their neighbors of a specified type of link synchronously. The propagation step keeps iterating until no more nodes are activated. Currently, PARKA supports the bottom-up inferencing (questions like "what is the color of Tweety?") as well as top-down inferencing (questions like "what things are red?").

The authors demonstrated that PARKA can accomplish inference in $O(d)$ time, while the sequential systems require $O(B^d)$ time where d is the network depth and B is the average network branchout.

2.4 Differences between TERRESA and Other Systems

Instead of using the marker-passing algorithm, we tried a different approach based on the task-based, message-driven model. The most significant reason is that we believe that there are several drawbacks of the marker-passing algorithm running on MIMD machines.

As described in article [61], the basic form of the parallel marker passing algorithm on MIMD machines can be shown in Fig. 2.14. For every iteration, each PE sends out (`Scatter`) the information acquired from the last cycle, and then tries to collect the information from other PEs (`Gather`). The real computation is performed after the collection

PARALLEL MARKER PASSING - REGULAR()

for $k \leftarrow 1$ **to** *level*
do $Scatter(level(k-1))$;
    $Gather(level(k))$;
    $Compute(level(k))$;

Figure 2.14: Parallel Marker Passing Algorithm on MIMD machines

PARALLEL MARKER PASSING - FUZZY SYNCHRONIZATION()

for $k \leftarrow 1$ **to** *level*
do $id \leftarrow FuzzyScatter(level(k-1))$;
    while $(FuzzyRecv(id, data)) = level\_not\_done)$
    do $Compute(level(k))$;

Figure 2.15: Parallel Marker Passing Algorithm on MIMD machines with fuzzy synchronization

is done. A variant of the original algorithm from the same paper is also given in Fig. 2.15. The difference between these algorithms, the fuzzy algorithm, and the original one is that the fuzzy algorithm is more relaxed than the original version at the synchronous points. It tries to overlap some of the computation with communication. In the fuzzy algorithm, it sends out the messages at the beginning of each cycle and then tries to receive the messages from other PEs. Upon the arrival of any message, the PE tries to calculate it until this level of computation is done. Despite its simplicity and elegance, the marker passing algorithm poses several potential problems when running on MIMD machines. In the following paragraph, two potential problems of the marker passing algorithm are described.

Synchronization

It is a general belief that in parallel computation, synchronization among different PEs is one of the major contributors to generating idle time. The marker passing algorithm described above, no matter in Fig. 2.14 or Fig. 2.15, is a synchronous algorithm. As we can see from Fig 2.14, there are two synchronous points for each iteration, `Scatter` and `Gather`. Although the synchronous effect can be alleviated by modifying the algorithm into fuzzy send/receive, the algorithm still needs to wait for all other PEs to finish their previous cycle before it can move to the next cycle. Therefore the fuzzy algorithm is still more or less a synchronous algorithm from a certain perspective.

Load Balance

In the marker passing algorithm on MIMD machines, all the computations are done locally, and messages are sent to other PEs for activating other nodes. Because there is no semantic network exchanging physically among PEs, the running time of each PE highly depends on the initial distribution and the subsequent activation pattern of the semantic nodes. If, during the computation, load imbalance does occur, there is nothing that can be done to balance the load by using the marker-passing algorithm, and therefore computing resource is wasted. In the worst case scenario, all the activated nodes are concentrated at one PE. This PE executes all the tasks while others sit idle.

Because of these these reasons, we feel a different design is needed for the TERRESA implementation. In the next chapter, the description of TERRESA design is described.

Chapter 3

TERRESA Design

In this chapter, the TERRESA design goals and considerations are stated. At the second part of the chapter, an overview of the two major TERRESA modules, the *host module* and *slave module*, is given.

3.1 Design Goals and Considerations

The current TERRESA design focuses on building an efficient path-based inference mechanism for various parallel computers while retaining the possibility of future extension to other functionalities, such as node-based inference. The most obvious goal in making a parallel system is to construct a system that runs faster than its single processor version. A well designed system should achieve good scalability, either in speedup or in memory capacity. Secondly, unless the systems are designed to be portable, parallel applications often are system dependent. Thus portability is also an important issue in the TERRESA implementation.

Some of the design goals are TERRESA specific. For example, in order to overcome the load balancing issue, both the knowledge and tasks must be migratable. All these issues are discussed in the following paragraphs.

3.1.1 Performance: Latency Hiding/Multithreading

Speedup in parallel computation is defined by the following formula [58]:

$$S = T_s / T_p$$

where S is the speedup of the parallel program, T_s is the running time on the sequential machine and T_p is the running time of the parallel version. If the number of PEs is N, the total parallel time $N \cdot T_p$ can be further decomposed into 3 parts: the actual *computation time* T_{cp}, *overhead time* T_o and *idle time* T_i. In most cases, the quantity of actual computation should be identical to the sequential time T_s. Therefore the formula can be organized into the following format:

$$N \cdot T_p = T_s + T_o + T_i$$

The overhead time T_o is the time consumed which is irrelevant to problem solving. The overhead T_o includes the bookkeeping time T_b, communication time T_c and load balancing/scheduling time T_l.

In decomposing the communication time, $T_c = \sum_{i=1}^{k} T_{c_i}$ for overall k communication calls. The time for the communication T_{c_i} is formed by the start up time $T_{c_{st}}$, and the actual transmission time T_{ct_i}. The start up time $T_{c_{st}}$ is more or less invariant to the message itself, but depends on the hardware and communication software implementation. The actual transmission time T_{ct_i} is usually proportional to the size of the message. Therefore the time for the communication is

$$T_c = k \cdot T_{c_{st}} + R \cdot \sum_{i=1}^{k} M_i$$

where R is the message transfer rate among PEs, and M_i is the message size of the ith message.

As the formula above suggests, a general principle of getting good performance (time-wise) is to increase the granularity of the computation (time ratio of computation versus communication), reduce idle time, fully utilize each processor, reduce overhead incurred by the parallelization and bookkeeping, reduce the number of communications and hide latencies which include the techniques of prefetching, overlap the communication and computation, etc.

One way to hide the latency of the computation is through the multithreading. A thread is an execution stream through a process and a multithreaded program is a program with multiple execution streams all of which can access a shared address space. The execution stream can be switched to other tasks when the current execution task is idle or blocked. Usually, multithreaded programming needs the support of either hardware or the operating system. Unfortunately in the design of TERRESA, we cannot make such an assumption unless the implementation is platform bounded.

3.1.2 Portability

Parallel computer environments vary greatly, both in the software environments and the hardware architectures. Some of the parallel computers like Tera's Multi-threaded Architecture (MTA) [2], support parallel multithreading by the hardware. Some of them like Intel iPSC-2 won't even allow process forking; some parallel computers have a front-end computer as the control host while others are hostless. Some of them only support basic languages like C and Fortran, while some of them support a great variety of languages and libraries. Similarly, the communication model supported by parallel computers also change from system to system. Without careful design, a portable system is unlikely, if not impossible.

Usually there are two approaches in implementing a portable system: the first approach is to hide all the system dependent functions into modules and reimplement them for every different machine. The second approach is trying to use a cross-platform library as its underneath support and let the library developer implement the system dependent codes.

In designing a portable system, the language adopted must be supported by all systems, otherwise it is questionable to build a portable system at all. This is also true for the communication model. The communication model has to be supported by all parallel machines, or the developers must implement the mechanisms that the system does not provide, which increases the effort of system development. Among all the parallel systems, message passing model is one of the most widely used models in programming a parallel application. Almost all the parallel systems support message passing mechanism to certain extent; for most of the commercial available shared memory systems, message passing mechanism can be achieved by emulation or library calls. [1]

3.1.3 Task Migration and Shared Knowledge

As stated in Section 2.4, one of the difficulties that does not exist in other parallel semantic network systems is the issue of load balancing. In designing a system capable of load balancing, task migration is essential. Furthermore, in order to execute the tasks in the remote PEs, any SNePS node must be accessible, no matter where the task is located. Borrowing the notion from the distributed shared memory (DSM) system, shared knowledge is embedded into the design of TERRESA.

3.1.4 TERRESA Design Preferences

TERRESA is written in ANSI C and the MPI (Message Passing Interface) library with the help of *flex* lexical analyzer and *yacc* parser generator. All of these are commonly supported tools in most of the parallel platforms, and thus the portability can be achieved.

Also in the TERRESA design, a multithreading environment, or "user level multithreading", embedded in the TERRESA system is developed. The goal of the multithreading scheme in TERRESA is to execute several different tasks "at the same time" or to have the capability to swap between different tasks once one a task is waiting for other information. The benefit of doing this is that communication can be overlapped with computation, and this approach can increase the performance of TERRESA.

[1] `MPICH` supports many shared memory parallel systems including SGI Onyx, Challenge, Power Challenge, and Power Challenge Array machines, IBM SMP's (symmetric multiprocessors), the Convex Exemplar, and the Sequent Symmetry. [26]

3.2 TERRESA Model

In contrast to other parallel semantic network systems, which uses a data parallel scheme as their fundamental parallelizing strategy, TERRESA uses the mixed strategy of task and data parallelism a task-based, message-driven model to accomplish the given query.

In the task-based message-driven model, the query is decomposed into tasks and arranged into PE for execution. Other system support activities are broken down into system tasks as well. When any tasks are done, a message is sent back to the system. Depending on the property of the task and the current status, either new tasks might be spawned or another message is sent back to the system again for further processing.

TERRESA can be split into two collaborating components: the host module and the slave module. The host module interacts with the user and processes the information for the slave modules, while the slave modules perform task execution. Communication between modules can be accomplished through done by the object-oriented packing system, if the designated information is a composite data structure or a direct communication call, if the designated information is a simple informative message.

3.2.1 Host Module

In our design, the host module is responsible for interacting with users (*language front-end*), decomposing the commands into either knowledge or tasks and dispatching them to PEs. All the preprocessing and distributing are enclosed in the *command processing module*. The *object-oriented packing system* is the communication channel between PEs.

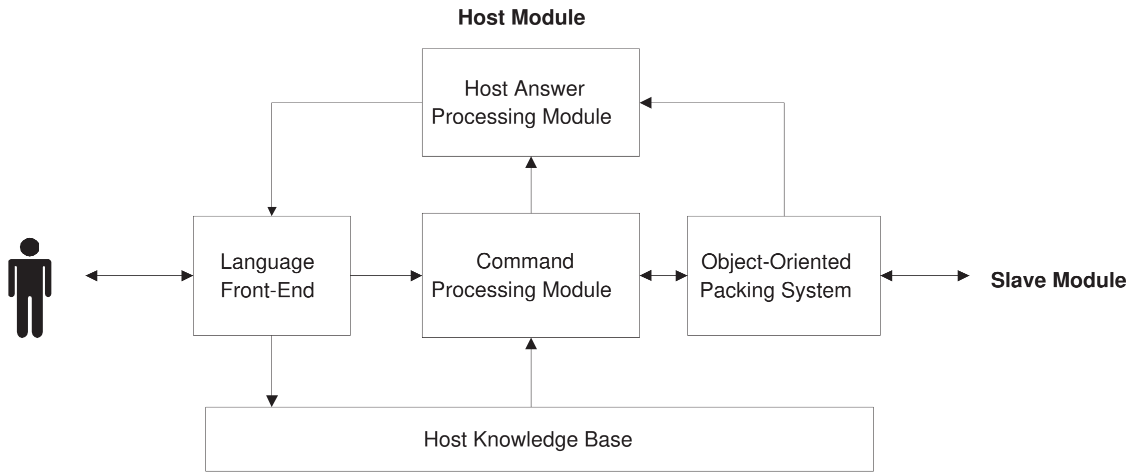

Figure 3.1: Host Architecture - Abstract Model

When the slave module is done with the query, it sends the answer messages back to the host module. After receiving the messages, the TERRESA system tries to merge the answers into a final inference conclusion. These actions are done inside the *host answer processing module*. Some of the knowledge is kept in the *host knowledge base* for simple queries. Fig. 3.1 demonstrates the host module architecture. The detailed design of the host module is discussed in Chapter 4.

3.2.2 Slave Module

There are several modules residing in the slave module as well. The *shared knowledge module*, which will be discussed in Chapter 6, is responsible for storing and exchanging knowledge in the *shared knowledge base*. The *task execution module* is the main execution core of the system, as all the tasks are executed through here. Several modules embedded inside the task execution module are not shown here, including the *kernel message modules*, the *task execution engine*, the *load balancing module*, etc. All of the task related

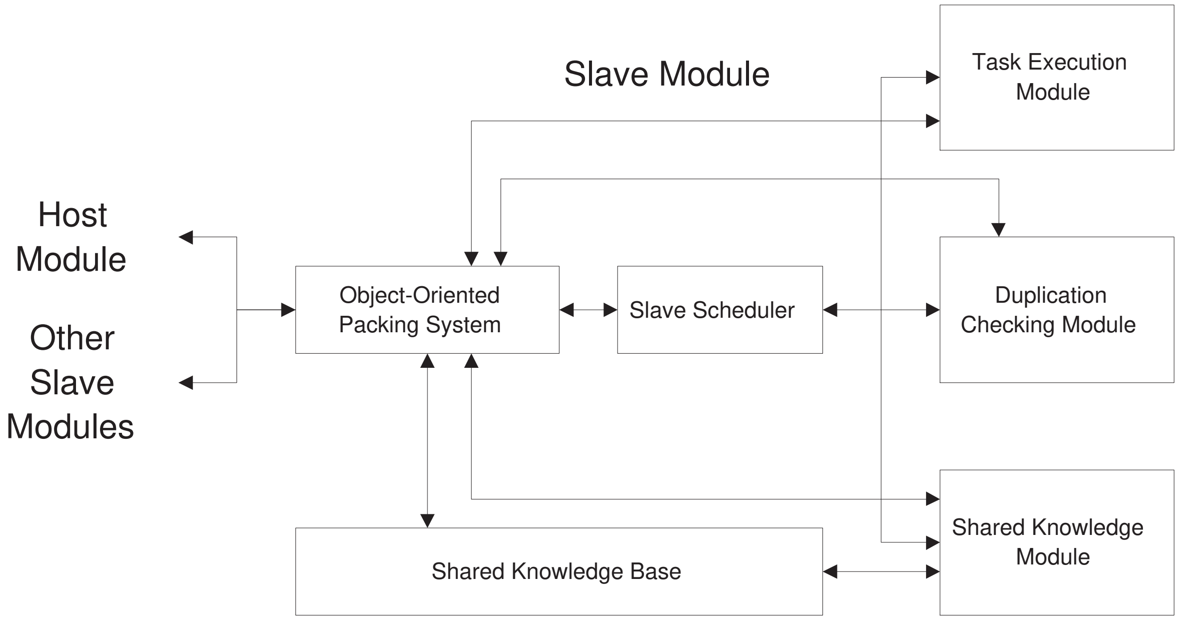

Figure 3.2: Slave Architecture - Abstract Model

module interactions will be discussed in Chapter 5. The *duplicate checking module*, scrutinized in Chapter 7, memorizes the answers that have been reached, and thus avoiding repetitive execution. The *slave scheduler* is a module which controls module execution, and tries to swap between concurrent execution modules. The *object-oriented packing system* is similar to the one in the host module. Both these modules are described in Chapter 8. Fig. 3.2 illustrates the abstract model and interactions of the slave module.

Chapter 4

Host System

As indicated in the previous chapter, the host module is composed of four interconnecting submodules, the *language front end*, the *command processing module*, the *host answer processing module* and the *object oriented packing system*. The object oriented packing system which will be discussed in Sec. 8.1, the other three modules are examined here. Fig. 4.1 illustrates the blueprint of the host module in detail.

4.1 Language Front End

The SNePS user interface, either SNePSUL or SNePSlog, follows a specific context-free grammar [57], and therefore a language front-end is needed to translate the syntactically correct user input to either internal data representation or information for the queries.

In the TERRESA implementation, part of the SNePSUL interface is implemented. (For

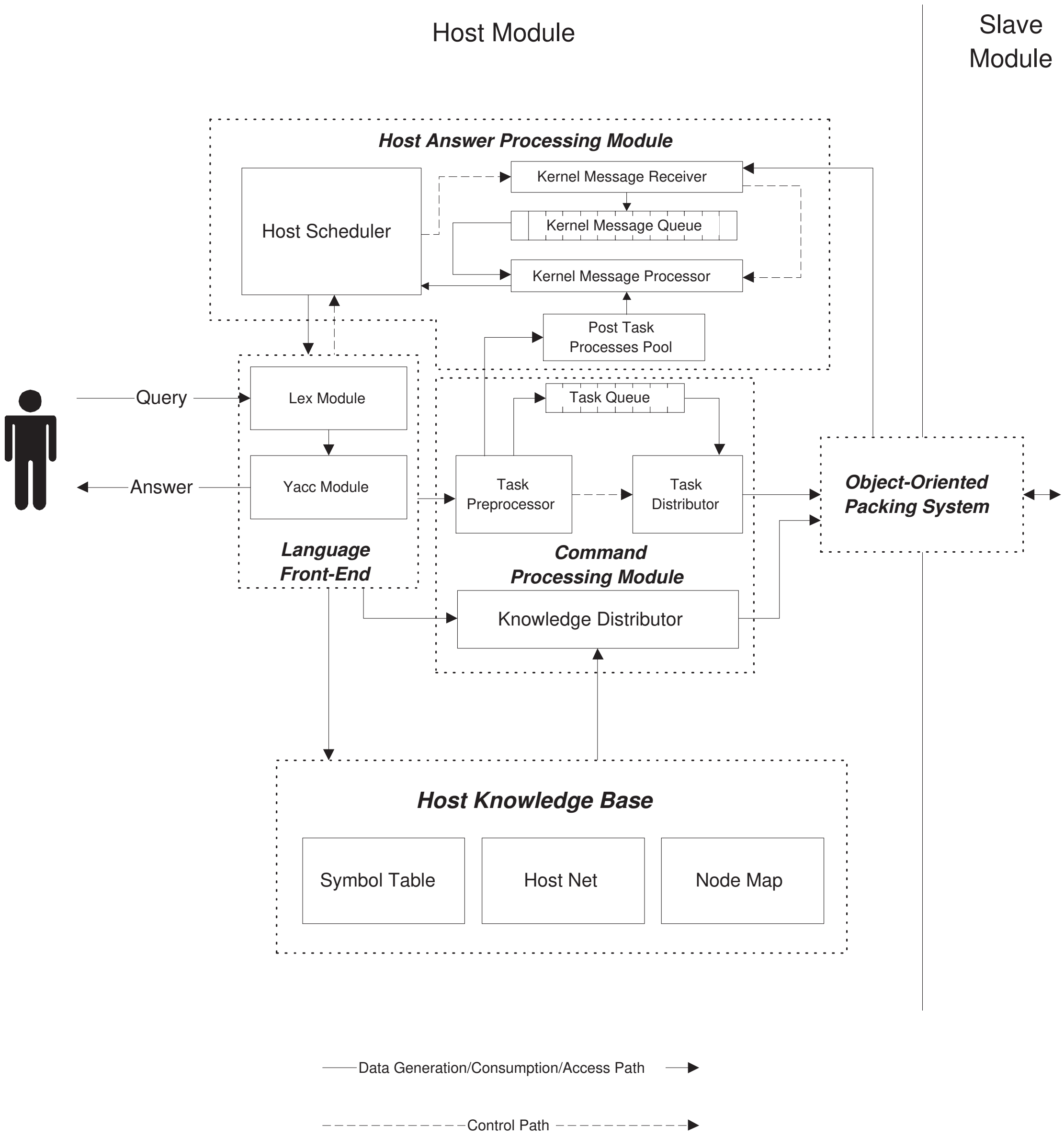

Figure 4.1: Detail Host Architecture

details please see Appendix A) The language front-end is composed of the lexical analyzer `sneps.f`, written in lex, and the parser `sneps.y`, written in yacc. Similar to compilers or interpreters, the lexical analyzer gets input either from the standard input or file and then converts them into a series of tokens. On the other hand, the parser takes the tokens from the lexical analyzer and then parses it according to the predefined grammar. If the parsing is successful, proper actions are performed, otherwise the sentence is rejected. In the meantime, if the commands are not related to the slave PEs, some of the host functions are called to solve them directly on the host PE.

4.2 Command Processing Module

The duty of the command processing module is to receive the commands from the parser, reshape them into proper data structures, dispatch them to the designated slave PEs and update the associated data structures in the host knowledge base.

By examining the command's characteristic, they can be categorized into three major groups: the commands related to building SNePS networks (for example, `build` and `assert`, etc.), the commands related to inference (like `find`, `findassert`, etc.) and all other commands. Frequently, the commands in the third category are answered directly inside the host module. The commands belonging to the first and the second category usually need to talk with slave PEs to finish the given task.

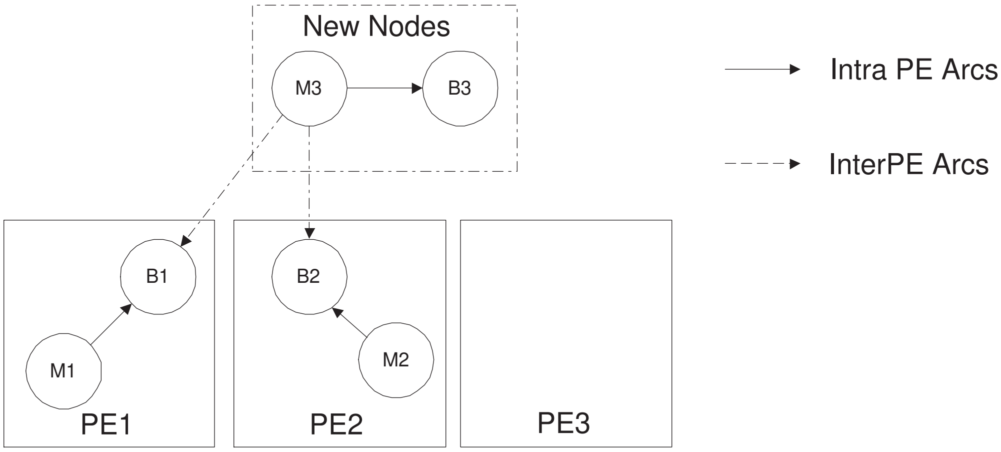

Figure 4.2: Node Distribution, in the figure, M3 and B3 are the new nodes and will be distributed into PE_3, the arcs between $\langle$B1, M3$\rangle$ and $\langle$B2, M3$\rangle$ will be dispatched to PE_1 and PE_2, respectively

4.2.1 Network Building Commands and Node Distribution

At this moment, TERRESA supports two network building commands: *assert* and *build*. Building a network involves installing new nodes and making connections between existing nodes. Both the nodes and the arcs need to be transferred to the slave PEs.

The node distribution algorithm is deals with the issue of splitting the SNePS network among slave PEs. Theoretically the goal of node distribution or graph splitting is to minimize the arcs across different partitions while keeping the the number of nodes balanced in each PE in balance. The optimal graph partition problem belongs to the *NP*-hard class, even for two PEs [22]. So, in the real world implementation, heuristics are usually adopted.

In the TERRESA implementation, the *cyclical block distribution* algorithm is adopted. Starting from PE_1, when a network construction command is issued, the host tries to send the newly built nodes to the target PE until a certain number of nodes have accumulated within it. Then the target PE is shifted to the next PE in a cyclical fashion until all the nodes are in the system or no space is left in the shared knowledge base.

After the target PE is assigned, the host starts the nodes and edges distribution. As illustrated in Fig. 4.2, there are two kinds of information that need to be distributed: the first type of messages are the newly built nodes and the arcs among them, the second type of messages are the arcs between the new nodes and existing nodes residing in somewhere other than the target PE. Therefore a two-stage communication is needed. At the first stage the host puts the newly built nodes into the target PE, and at the second stage the host adds the interPE arcs to the proper PEs.

4.2.2 Inference Commands

Several path-based inference commands are provided by the TERRESA system, including the `find` family (`find`, `findassert`, `findbase`, `findconstant`, `findpattern` and `findvariable`). The tasks are generated by the *task preprocessor* under the command of the parser in the language front end. It is then stored in the host *task queue* temporarily as if waits to be the dispatched by the *task distributor*. The query tasks are split according to the parallelism implied by the command.

For example, when a path-based query is issued, the command is in the format of

$$\texttt{(find } \{\textit{path nodeset}\}^{*}\texttt{)}$$

which is equivalent to

$$\bigcap_{i=1}^{n} \bigcup_{j=1}^{f(i)} (\texttt{find}\; path_i\; node_{i,j})$$

where the $\bigcap$ and $\bigcup$ are the nodeset intersection and union, respectively.

According to the syntax of `find`, the *nodeset* can be either a set of actual nodes, another SNePS commands which returns a nodeset, or a *?symbol* (which stands for "any node"). The query with the *?symbol* is called *wildcard query* in this dissertation. In SNePS, the same symbol in the same query is bound to the same nodeset. TERRESA does not support this feature, each wildcard symbol is consider as a distinct variable even though they have multiple occurences in the same query.

When distributing the tasks into the slave PEs, a two-stage distribution is adopted. Because it is not economically sound in terms of communication time if the instantiation of *?symbol* to all nodes is done in the host PE, the first stage of task distribution is to identify the queries related to the wildcard query. If there are any, the host sends a *wildcard query task* to all the slave PEs and lets the slave PEs instantiate it within the slaves and enqueues them into the slave task queue.

For the query other than wildcard query, all other SNePS commands need to be instantiated into nodesets before they participate in any other computations. In the second stage of the distribution, the tasks related to actual nodes are distributed among the slave PEs. Over this stage, if we assume all the wildcard queries have been handled during stage one, the tasks in the $\bigcap_{i=1}^{n} \bigcup_{j=1}^{f(i)}$ $(\texttt{find}\, path_i\ node_{i,j})$ can be decomposed into $\sum_{i=1}^{n} f(i)$ `find` tasks, $\sum_{i=1}^{n} f(i)$ nodeset union and n nodeset intersect. The uniform distribution scheme is adopted when actually distributing the tasks, so each PE will get either $\lceil \sum_{i=1}^{n} f(i)/NP \rceil$ or $\lfloor \sum_{i=1}^{n} f(i)/NP \rfloor$ tasks if NP is the number of PEs.

When the command processing module generates tasks for the slave PEs, it also generates the corresponding *post task processes (ptasks)* and puts them into the *host post task pool*. A ptask is a tagged data structure which represents what should be done after a task is finished. It can be triggered by a *kernel message (kmesg)*. In the case described above, all union and intersect tasks are transformed into ptasks. A detailed description of the task-ptask-kmesg relations will be offered in Chapter 5. In reflecting the jobs of union and intersection, $n+1$ of the ptasks are installed into the host ptask pool. There are n OR ptasks, each of which accepts $f(i)$ kmesgs depending on which nodeset union the ptask is corresponding to and one special ptasks, the MASTER-TASK ptask.

The special ptask MASTER-TASK, which is the ancestor of all the ptasks and performs the similar action to AND ptask, is installed at the beginning of the query. The function of the MASTER-TASK is to accept the kmesg from the host OR ptask, intersect them and summarize the answer for user.

4.2.3 Other Commands

TERRESA also support other miscellaneous commands, including *nodeset operation commands* and *SNePS variable commands*. Appendix A illustrates the details of what TERRESA has implemented.

4.3 Host Answer Processing Module

Several control modules and data structures exist in the *host answer processing module*, which include a *kernel message receiver*, a *kernel message queue*, a *kernel message processor*, a *post task pool* and a *host scheduler*.

Post Task Pool and Kernel Message Related Modules

These modules are similar to the one in the slave module with stripped down functionalities. Only two kinds of post tasks can be executed in the host module: the OR, and the MASTER-TASK. The way these modules cooperate is that the *kernel message receiver* receives the messages originating from different slave PEs and puts them into the *kernel message queue*; the *kernel message processor* dequeues it and finds the matched post task in the *post task pool*. A nodeset merge, either union or intersect is performed when the match is found. A new kernel message is sent to its parent ptask if the ptask receives sufficient kmesgs. As stated, the MASTER-TASK collects incoming kernel messages from all of the OR ptasks and finalize the query.

Host Scheduler

The host scheduler does not schedule the modules' execution like the slave scheduler does. The name "host scheduler" is a rhetorical term merely reflecting the reciprocal function unit in the slave PEs. The primary function of the host scheduler is to monitor the behavior of the kernel message queue and the post task pool. The host scheduler repeats these actions until the MASTER-TASK is terminated.

4.4 Host Knowledge Base

Two major repositories comprise the host knowledge base: the *host net* and the *node map*. These data structures are updated based on the needs of certain SNePS commands.

Symbol Table

SNePSUL allows variables to be stored in the system. In the TERRESA implementation, a *symbol table* (SBT) is used to store the related information like the variable name, its type and value. A hash table is used as the symbol table's underlying data structure.

Host Net

As the name implies, the data structure *host net* (HN) retains a copy of the knowledge base on the host. Its primary function is to provide the answer for non-inferential queries.

Node Map

Instead of using string names to distinguish the SNePS nodes, TERRESA uses the tuple of node address $\langle type, id \rangle$ internally to identify the nodes, where *type* could be: `BASE`, `MOLECULE`, `VARIABLE`, or `PATTERN`, and *id* is a serial number given by the system. The data structure *node map* (NM) maps the node address into the node name (node name is the alphanumerical value which represents a node given by users like X, Y, Z, SOCRATES, HUMAN, etc). The node map is used as the mapping mechanism when the host receives replies back from the slave PE.

Chapter 5

Abstract Model and Task Execution Engine

The architecture of the slave module is illustrated in the Fig 5.1.

As can be seen from the figure, various modules are interwoven together to form the system supporting the SNePS inference. This chapter focuses on the components directly related to the TERRESA tasks: The *task execution module* and its related components. In the task execution module, the actual inference procedures are being carried out.

The plan of this chapter is as follows: The model related to the task execution module are described in Section 5.1. Next, an overview of the task and related data structures, the task transitions, and the abstract models are given. The load balance module is discussed in the last section of this chapter.

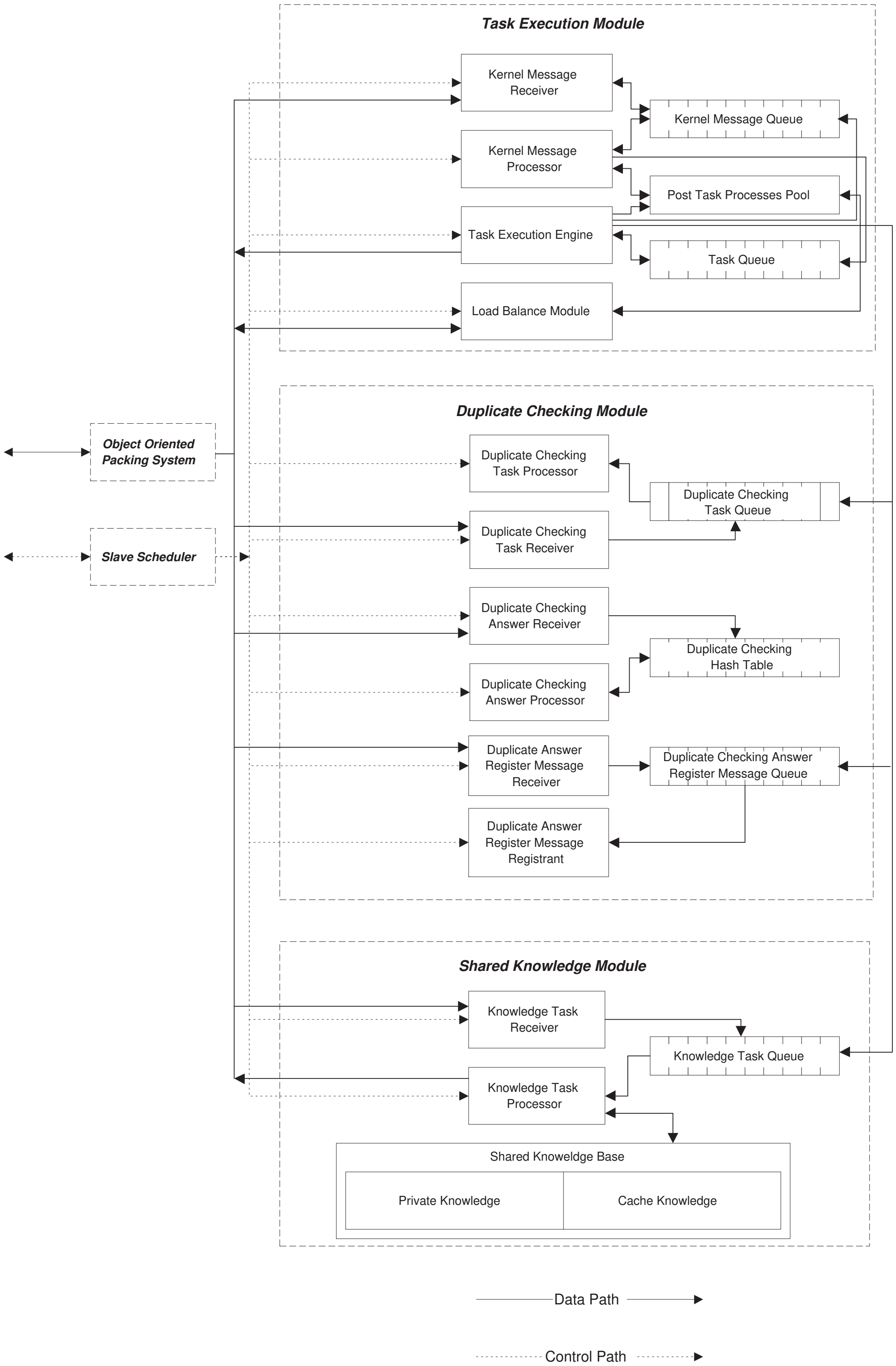

Figure 5.1: The Slave Module Architecture

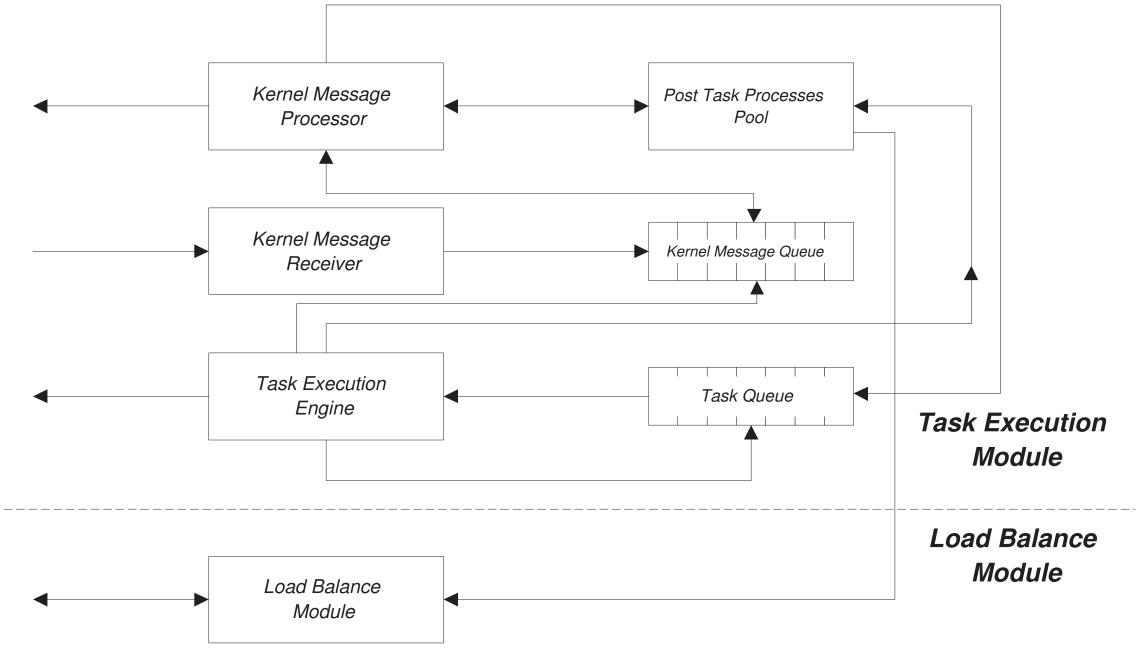

Figure 5.2: Task Execution Module

5.1 The Computational Model

The computational model of the TERRESA task execution module is illustrated in Fig. 5.2. The task execution module consists of several smaller components: the *kernel message queue*, the *kernel message processor*, the *kernel message receiver*, the *post task processes pool*, the *task execution engine*, and a *task queues*. Furthermore, the *load balance module* is hooked up with the ptask pool for the load balance among different PEs.

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

The *kernel message receiver*, acts as a daemon for remote kernel messages. It receives them from other PEs and puts them into the *kernel message queue*. The *kernel message processor*, in the mean time, dequeues them from the kernel message queue, tries to find the matching ptask from the *post task processes pool* and executes them. The execution of the ptask might generate new kmesgs, new tasks, or new ptasks, all of which will be put into proper places to wait for being picked up and executed again.

At the same time, the *load balance module* is invoked periodically to balance the load difference between different PEs. It transmits the ptasks to the remote PE and places them back in the proper location when the incoming ptasks are received.

5.2 Definitions

Some terminologies used in this chapter are defined in advance. As described in the SNePS manual, a *path* can be either a *unit path* or a *composite path* starting with one of the 10 symbols: `COMPOSE`, `KSTAR`, `KPLUS`, `OR`, `AND`, `NOT`, `RELATIVE-COMPLEMENT`, `IRREFLEXIVE-RESTRICT`, `DOMAIN-RESTRICT` or `RANGE-RESTRICT`. The symbol `EXCEPTION` is not implemented due to the different nature of this command.

For convenience, the path header, path body and path tail are defined below:

Definition 1 (Path header (PH))

A **path header** of a path P, $PH(P)$, is the first symbol in the path. It could be the relation/relation- if the path is a unit path or the heading symbol if the path P is a composite path. In Lisp terms, it is equivalent to `(car P)`. A special case is the path of *(path*)* $\equiv$ (P_1 P_2 P_3 ...

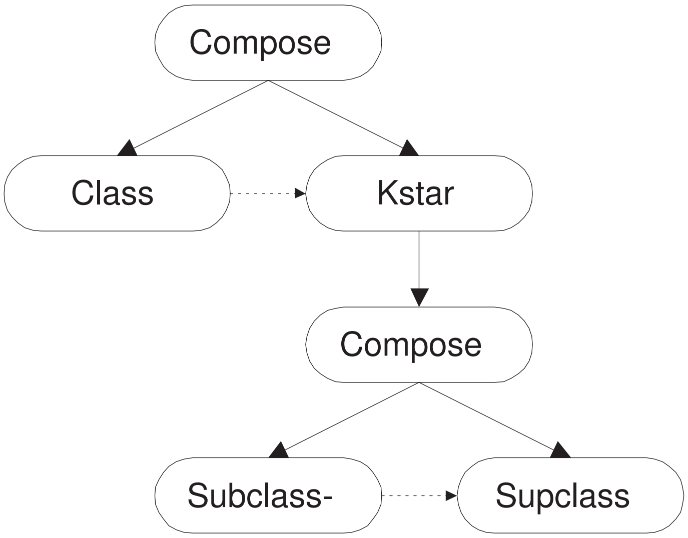

Figure 5.3: A path, the tree representation

P_n), the path header COMPOSE is added automatically to reflect the correct interpretation of the path definition.

Definition 2 (Path body (PB))

A **path body** of a path P, $PB(P)$, is the path stripping off the path header from a path. In Lisp terms, it is equivalent to `(cdr P)`.

Definition 3 (Path tail (PT))

A **path tail** of a path P, $PT(P)$, is the last entry in the path. In Lisp terms, it is equivalent to `(first (last P))`.

For example, for the path

P = `(COMPOSE class (KSTAR (subclass- supclass)))`

PH(P) = `COMPOSE`, PB(P) = `(class ( KSTAR ( subclass- supclass ) ) )` and PT(P) = `(KSTAR ( subclass- supclass ) )`.

From the programmer's point of view, the data structure of a path P defined in the system is represented by a tree of multiple children with PH as the root and each element in

PB as its child. As illustrated, the path of *P* defined above can be represented in the form of a tree in Fig. 5.3. The dash lines between two adjacent horizontal nodes are the sibling links.

In the path-based inference, every task can be viewed as a miniature capsule carrying a starting node *N* and a path *P*, as well as other bookkeeping information. Yet the task alone cannot solve the TERRESA queries. Unlike many other parallel systems, while tasks in other parallel systems are so called *"fire-and-forget"* tasks, there are three major components in the TERRESA task execution model: the *tasks*, the *post task processes* (ptask in abbreviation) and the *kernel messages* (kmesg in abbreviation). When a query is issued by the user, TERRESA will decompose it into two related components: the task, as stated above, carries the information for realizing the query and the ptask, as the base of further processing, indicates current status and what needs to be done once the tasks are solved. During the process of the task, it might generate new tasks/ptask or be solved on the spot. If the task is solved, a kmesg is sent back to its corresponding ptask. The ptask, triggered by the kmesg, might generate new tasks or send another kmesg back to its parent ptask depending on the status itself.

This *task-ptask-kmesg* model can be looked at as a generalized function executions, where the functions can be executed in parallel if possible. When a function is called, the *task* part is being performed. Upon completion, a *kmesg* containing the answer is returned to where it was from, and the *ptask* decides whether any further processing of the answer should be considered. In the traditional programming model, by using the stack as the underneath support mechanism (which is inherently sequential in terms of growth and shrink), we are limited to using the sequential computers. It is therefore the motivation to explicitly design a model expressing the function behavior while maintaining the capability of being executed in parallel even migrating to the remote PEs.

Now the terminologies related to the tasks, ptasks and kmesgs are defined. Formally, the task, ptask and kmesg can be defined in the following:

Definition 4 (Task)

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

Definition 5 (Post Task Process (ptask))

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

Definition 6 (Kernel Message (kmesg))

A *Kernel Message* (kmesg) is defined as a tuple of $\langle Id, NS, Aid \rangle$ where Id is an identification tag (for ptask) and NS is the body of the message, which is a nodeset in the TERRESA implementation. Aid is an auxiliary data structure which will be useful only for the inference of `RELATIVE-COMPLEMENT` and `RANGE-RESTRICT`.

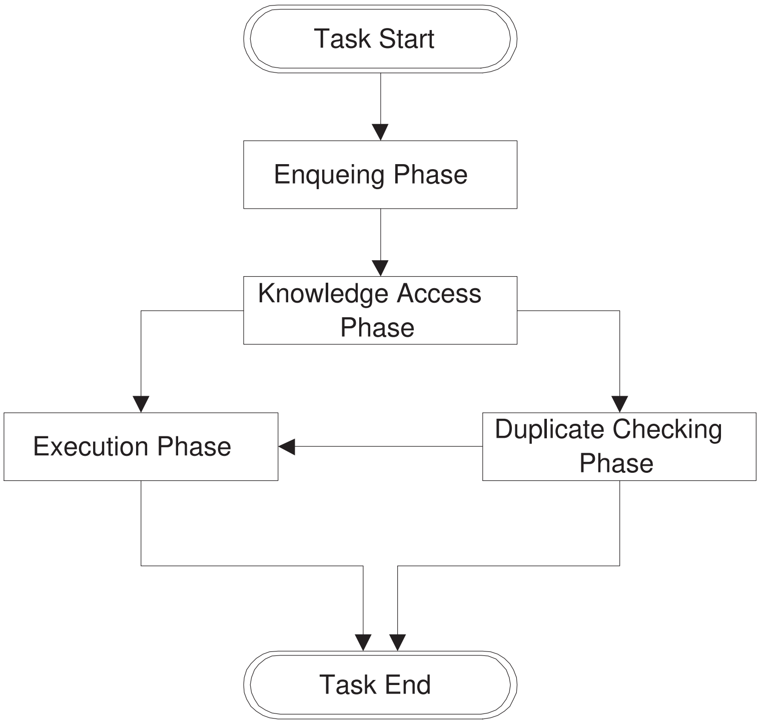

Figure 5.4: Task Phase Transition

5.3 Phase Transition

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

The life cycle of a task starts from the *enqueuing phase* and goes through the following phases until the task is terminated: the *knowledge access phase*, the *execution phase* and the optional *duplicate checking phase*. Fig. 5.4 illustrates the phase transition of the tasks.

During the enqueuing phase, the task is spawned and allocated into the task queue.

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

5.3.1 Enqueuing Phase and Knowledge Access Phase

The enqueuing phase is the prelude of all other phases. As implied by the name, the tasks in the enqueuing phase are basically waiting to enter the knowledge access phase by the task execution engine. The enqueuing phase of a task is incurred by the birth of a new task. All enqueuing phase tasks are residing in the *task queue (TQ)*.

In the knowledge access phase, the system ensures that the contents of the SNePS node N of the task T_k is actually residing in the local PE before it enters the execution phase.

The algorithm listed in Fig. 5.5 is a part of the task execution engine related to these two phases. The task enters the phase with `TASK_ENQUEUE`, if the node is not local, a knowledge access task is issued to the remote PE and the task status is shifted to `TASK_WAIT_KNOWLEDGE` until the reply is received, once the node information is local, the task status is transformed to `TASK_READY`: ready for execution or performing the duplicate checking. The details of knowledge task operation are discussed in Chapter 6.

KNOWLEDGE ACCESS PHASE ALGORITHM(T_k)

switch $T_k.Status$
  case `TASK_ENQUEUE :`
      if $T_k.N$ is in local PE
        then $T_k.Status \leftarrow$ `TASK_READY`
        else $T_K.Status \leftarrow$ `TASK_WAIT_KNOWLEDGE`
              Issues the knowledge task K_{tk} with $K_{tk}.N \leftarrow T_k.N$
       Enqueue T_k into TQ
  case `TASK_WAIT_KNOWLEDGE :`
      if $T_k.N$ is in local PE
        then $T_k.Status \leftarrow$ `TASK_READY`
       Enqueue T_k into TQ
  case `TASK_READY :`
      ...

Figure 5.5: Task Enqueuing Phase and Knowledge Access Phase Algorithm

5.3.2 Execution Phase

At this point, the task has determined whether it should go through the duplicate checking phase as depicted in the next section or to be executed directly. When a task decides to go through the execution phase instead of duplicate checking phase, some predefined actions are performed. The action of a task depends on the path $T_k.P$ and the node $T_k.N$.

When executing a task, the task can either generate new tasks or finish up the execution and send a message back to the corresponding ptask but not both. In the following paragraph, the algorithms of each predefined actions are introduced.

Action 1

For a task $T_k = \langle P_{id}, P, N \rangle$, if $T_k.P$ = NULL, generate a kmesg($T_k.P_{id}$, $T_k.N$, `D/C`)

Action 2

For a task $T_k = \langle P_{id}, P, N \rangle$, if $T_k.N$ = NULL, then generate a kmesg($T_k.P_{id}$, NULL, `D/C`)

The above rules indicate two extreme cases in the path-based inference. Action 1 indicates that when there is no path to be tracked, by default the node is the answer. Action 2 indicates that the task is in the middle of following a path P but it consumes all the possible nodes in the knowledge base, therefore a kmesg containing the answer the NULL should be sent back to its corresponding ptask. For the term "D/C" indicates the field does not carry any important information or a "Don't Care" condition.

Action 3

For Task T_k= $\langle P_{id}, P, N\rangle$, r is a unitpath (relation or relation-), and suppose $n = T_k.N$

if $T_k.P = r$ and $r = (n,n_1), (n,n_2), \dots\ (n,n_k)$, then the task execution engine should generate a kmesg(P_{id}, (n_1, n_2, ..., n_k), D/C)

if $T_k.P = r$ and $r \neq (n,n_i), \forall i$, the task execution engine should generate kmesg(P_{id}, NULL, D/C)

Action 3 indicates that whenever a task is asking for tracking a unitpath, a kernel message is generated indicating the result, whether there is a feasible result (in the case above $n_1, n_2 \dots n_k$) or not (NULL).

The following rules from Action 4 to Action 13 reflect the actions that occur when the task execution engine sees a path header belonging to the predefined SNePS path symbols.

Action 4 (COMPOSE)

For task $T_k = \langle P_{id}, P, N\rangle$, if $PH(T_k.P)$ = COMPOSE and $P_n = PT(PB(T_k.P))$, then the execution engine needs to generate a new task NT_k and a new ptask NP_{tk} where the ptask NP_{tk} = $\langle Id$, $T_k.P_{id}$, COMPOSE, NULL $\rangle$ and the new task $NT_k = \langle NP_{tk}.Id, P_n, T_k.N\rangle$.

As shown in Fig. 5.6, when a task contains the predefined symbol COMPOSE, the system generates a new task NT_k with the path pointing to its last child, and its P_{id} is labelled

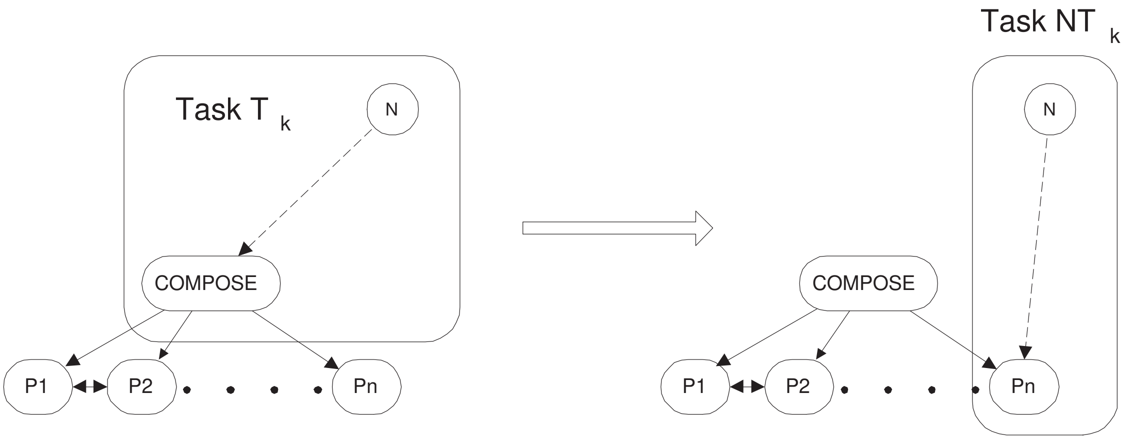

Figure 5.6: Task transition of the symbol COMPOSE

as the id of the new born ptask NP_{tk}. This operation conforms the definition of COMPOSE defined by SNePS. By giving the terminal node x_n and a path P, it tries to find out the node (or nodeset) x_1: If $x_1, x_2, \ldots, x_n$ are nodes and P_i is a path from x_i to x_{i+1}, then (compose $P_1 \ldots P_{n-1}$) is a path from x_1 to x_n.

Action 5 (KSTAR)

For a task $T_k = \langle P_{id}, P, N \rangle$, if $PH(T_k.P)$ = KSTAR, we assume that $P' = PH(PB(T_k.P)) = PT(PB(T_k.P))$ since the syntax of KSTAR only allows one path after it. The task execution engine will generate a new task NT_k and a new ptask NP_{tk} where task $NT_k = \langle NP_{tk}.Id, P', T_k.N \rangle$ and ptask $NP_{tk} = \langle Id, T_k.P_{id},$ KSTAR, $T_k.N \rangle$

Action 6 (KPLUS)

For a task $T_k = \langle P_{id}, P, N \rangle$, if $PH(T_k.P)$ = KPLUS, the definition of $P' = PH(PB(T_k.P))$ $= PT(PB(T_k.P))$ is similar to KSTAR. The task execution engine will generate a new task NT_k and a new ptask NP_{tk} where task $NT_k = \langle NP_{tk}.Id, P', T_k.N \rangle$ and ptask $NP_{tk} = \langle Id, T_k.P_{id},$ KSTAR, NULL $\rangle$

The actions of KPLUS and KSTAR are similar to each other except for the answer sec-

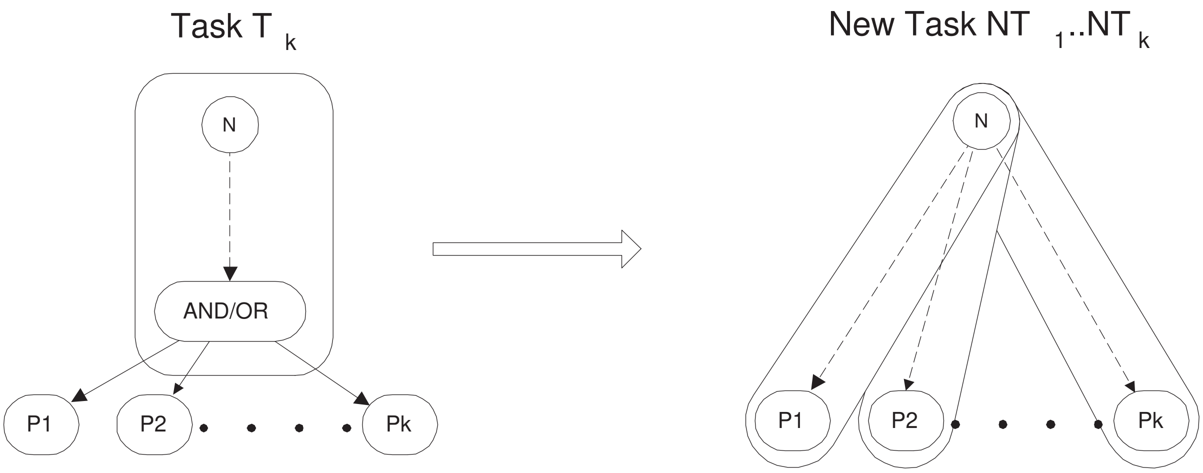

Figure 5.7: AND/OR tasks transition

tion of NP_{tk} whereas the KSTAR contains the original starting node $T_k.N$, and the KPLUS sets to empty set NULL. Therefore the ptask of KPLUS is actually set to KSTAR since the starting node has been excluded from the KPLUS inference.

The following two actions are also similar to each other. One is the AND and the other one is the OR.

Action 7 (AND)

For a task $T_k = \langle P_{id}, P, N\rangle$, if $PH(T_k.P)$ = AND and $P_i \in PB(T_k.P), i = 1 \ldots k$, then the task execution engine will generate a ptask NP_{tk} and k new tasks $NT_{k_1}, NT_{k_2}, \ldots, NT_{k_k}$ where $NP_{tk} = \langle Id, T_k.P_{id}$, AND, NULL, $\rangle$ and $NT_{k_i} = \langle NP_{tk}.Id, P_i, T_k.N\rangle$, for all $i = 1 \ldots k$.

Action 8 (OR)

For a task $T_k = \langle P_{id}, P, N\rangle$, if $PH(T_k.P)$ = OR and $P_i \in PB(T_k.P)$, $i = 1 \ldots k$, then the task execution engine will generate a ptask NP_{tk} and a series of new tasks $NT_{k_1}, NT_{k_2}, \ldots, NT_{k_k}$ where $NP_{tk} = \langle Id, T_k.P_{id}$, OR, NULL $\rangle$ and $NT_{k_i} = \langle NP_{tk}.Id, P_i, T_k.N \rangle$, for all $i = 1 \ldots k$.

The commands AND and OR are similar to each other except that AND is expected to find the intersection of the inference result while OR is trying to find the union of the result. The new tasks generated from the paths underneath the symbol AND and OR can be executed in parallel. Fig. 5.7 illustrates the transition.

Action 9 (NOT)

For a task $T_k = \langle P_{id}, P, N\rangle$, if $PH(T_k.P)$ = NOT and if we define $P' = PH(PB(T_k.P))$, there will be a new task $NT_k = \langle NP_{tk}.Id, P', T_k.N\rangle$ as well as a new ptask $NP_{tk} = \langle Id, T_k.P_{id}$, NOT, NULL $\rangle$ generated by the task execution engine.

As described in Chapter 2, if there is no path P from node x to node y, then (NOT P) is a path from x to y. Or equivalently, if there is a path P from x to y, then there will be a path (NOT P) from x to V - y where V is the nodeset composed of all the nodes in the knowledge base. It is therefore our intention to generate the task described above first, and then try to figure out the node y. The computation of V - y will be done in the ptask process session.

The task decomposition of the command RELATIVE-COMPLEMENT is somehow different from all other path symbols because of semantic differences between the two paths defined in the command (RELATIVE-COMPLEMENT P Q).

Action 10 (RELATIVE-COMPLEMENT)

For a task $T_k = \langle P_{id}, P, N\rangle$, if $PH(T_k.P)$ = RELATIVE-COMPLEMENT and if we define P' = $PH(PB(T_k.P))$, and $Q' = PT(PB(T_k.P))$. There will be two new tasks generated by the system where $NT_{k_1} = \langle NP_{tk_1}.Id, P', T_k.N\rangle$ and $NT_{k_2} = \langle NP_{tk_2}.Id, Q', T_k.N\rangle$. Unlike previous actions, which only generate one ptask, the symbol RELATIVE- COMPLEMENT will generate three ptasks. The first ptask $NP_{tk_0} = \langle Id, T_k.P_{id}$, RELATIVE- COMPLEMENT, NULL $\rangle$, the second ptask $NP_{tk_1} = \langle Id, NP_{tk_0}.P_{id}$, FORWARD-MESSAGE, NULL $\rangle$, and the third ptask $NP_{tk_2} = \langle Id, NP_{tk_0}.P_{id}$, FORWARD-MESSAGE, NULL $\rangle$.

Action 11 (IRREFLEXIVE-RESTRICT)

For a task $T_k = \langle P_{id}, P, N\rangle$, if $PH(T_k.P)$ = IRREFLEXIVE-RESTRICT, and $P' = PH$ (PB ($T_k.P$)), then the system will generate a new task $NT_k = \langle NP_{tk}.Id, P', T_k.N\rangle$ and a new ptask $NP_{tk} = \langle Id, T_k.P_{id}$, IRREFLEXIVE-RESTRICT, NULL $\rangle$.

By the definition listed in Chapter 2, the query ((IRREFLEXIVE-RESTRICT P) y) is X - y if we can find X as the answer for the query (P y). The task listed above tries to derive (P y). The actual answer for the IRREFLEXIVE- RESTRICT X - y will be done during ptask processing.

Action 12 (DOMAIN-RESTRICT)

For a task $T_k = \langle P_{id}, P, N\rangle$, if $PH(T_k.P)$ = DOMAIN-RESTRICT, we define $P' = PH$ (PH ($PB\,(T_k.P)$)), $n = PT(PH(PB(T_k.P)))$ and $Q' = PT(PB(T_k.P))$. Or in the DOMAIN-RESTRICT path definition (DOMAIN-RESTRICT ($path_1$ $node$) $path_2$), P' is the $path_1$, Q' is the $path_2$ and the $node$ is denoted by n'. The the system will generate one ptask $NP_{tk} = \langle Id, T_k.P_{id}$, DOMAIN-RESTRICT, NULL $\rangle$ and two new tasks of $NT_{k_1} = \langle NP_{tk}.P_{id}, P', T_k.N\rangle$ and $NT_{k_2} = \langle NP_{tk}.P_{id}, Q', n'\rangle$

By the definition of DOMAIN-RESTRICT, which is listed in Chapter 2, if P is a path from x to y and Q is a path from x to z, then (DOMAIN-RESTRICT (Q z) P) is a path from x to y. When given y and the (DOMAIN-RESTRICT (Q z) P), it is not hard to derive that $x = Q_z \cap P_y$ where Q_z is the result of the query (Q z) and P_y is the answer of the query (P y). Therefore two tasks NT_{k_1} and NT_{k_2} are denoted as above.

Also, as illustrated in Fig. 2.6, if there is a path P from x to y and Q is a path from y to

z, then there is a path (RANGE−RESTRICT P (Q z)) from x to y. Or in the path-based inference query, if $y \in Q_z$ which is the answer of (Q z), then ((RANGE−RESTRICT P (Q z)) y) $\equiv$ (P y). Therefore in the real implementation, two tasks are generated which represents the query of (Q z) and (P y), respectively. The merging of the answers will be processed during ptask processing. However, due to the discriminatory nature of these two tasks, two ptasks are inserted into the system to facilitate the identification of the answers. Furthermore, in each of the new ptasks, when the action is FORWARD−MESSAGE, a variable aid is set corresponding with the accompanied task.

Action 13 (RANGE-RESTRICT)

For a task $T_k = \langle P_{id}, P, N\rangle$, if $PH(T_k.P)$ = RANGE−RESTRICT, $P' = PH(PB(P))$, $Q' = PH(PT(PB(P)))$, and $n' = PT(PT(PB(P)))$. Or in the path of (RANGE−RESTRICT $path_1$ ($path_2$ $node$)), P' is the $path_1$, Q' is the second path - $path_2$ and the $node$ is denoted by n'. Then the system will spawn three ptasks: $NP_{tk_0} = \langle Id, T_k.P_{id}$, RANGE− RESTRICT, NULL $\rangle$, $NP_{tk_1} = \langle Id, NP_{tk_0}.Id$, FORWARD−MESSAGE, NULL $\rangle$, and $NP_{tk_2} = \langle Id, NP_{tk_0}.Id$, FORWARD−MESSAGE, NULL $\rangle$. Two new born tasks are queued into the system as well $NT_{k_1} = \langle NP_{tk_1}.Id, P', Tk.N\rangle$ and $NT_{k_2} = \langle NP_{tk_2}.Id, Q', n \rangle$.

5.3.3 Duplicate Checking Phase

Depending on the knowledge network topology and query, task executed on the different PEs might be identical, thus duplication occurs. If the user decides a task need to be checked for the duplication, the task enters the duplicate checking phase. Because duplicate checking involves a lot of network traffic, and, unlike the knowledge access, where the answers can be cached, this action is potentially expensive. Therefore, it is under the users discretion whether he/she prefers to turn this facility on.

Fig. 5.8 illustrates a portion of the status transition related to the duplicate checking. First the task issues a *duplicate checking task* D_{tk} to the duplicate checking system and then shifts into the pending status `TASK_WAIT_ DUP_RESPONSE` until a reply from duplicate checking system D_{rpy} is back. The reply will be one of the following:

- `DUP_CHK_SELF_SOLVE`: This status indicates that no similar task is issued to the duplicate checking system, therefore it is the responsibility of local PE to solve it and register the answer afterward if the answer is obtained.

- `DUP_CHK_WAIT`: This status indicates that some PE other than itself is solving the task. An answer will be fed back by the duplicate checking system in the future, hence the task retains the pending status and goes back to the task queue TQ.

- `DUP_CHK_SOLVED`: This status indicates that the answer is stored in the duplicate checking hash table. The answer accompanied with the reply D_{rpy} can be sent back to its parent ptask by a kmesg.

A detailed description of the duplicate checking module can be seen in Chapter 7.

5.3.4 Ptasks and Kmesgs Processing

When a kmesg is dequeued and presented to the system, the TERRESA system tries to find the matched ptask according to its identification ($K_{mesg}.P_{id}$, v.s. $P_{tk}.Id$). After the match

DUPLICATE CHECKING PHASE ALGORITHM(T_k)

```
switch (T_K.Status)
  case TASK_DUP_CHECK :
        Tk.Status ← TASK_WAIT_DUP_RESPONSE
        Issues the duplicate checking task D_tk with D_tk.Id = T_k.Id
        Enqueue T_k into TQ
  case TASK_WAIT_DUP_RESPONSE :
        if No such D_rpy such that Tk.Id = D_rpy.Id
          then Enqueue T_k into TQ
          else switch (D_rpy.Status)
                 case DUP_CHK_SELF_SOLVE :
                      /* No one answer the task, solve itself */
                      T_k.Status ← TASK_ENQUEUE_PASSTHRU
                      Enqueue T_k into TQ
                 case DUP_CHK_WAIT :
                      /* Some PE is solving the task but not */
                      /* yet finished, waiting for the answer */
                      /* from remote */
                      Enqueue T_k into TQ
                 case DUP_CHK_SOLVED :
                      /* The task has been solved already, use */
                      /* the answer directly */
                      Generate Kmesg⟨Tk.P_id, D_rpy.ans, D/C⟩
```

Figure 5.8: Duplication Checking Phase Algorithm

is found, then one of the following ptask algorithms is executed according to the action of the ptask ($P_{tk}.Act$).

COMPOSE

The algorithm of the ptask action of `COMPOSE` is shown in Fig 5.9. Over the algorithm, the notation p of lines 3, 4 and 9 indicates the path being tracked currently, which is stored in the ptask itself but not mentioned in the Action 4 because it is specifically associated with `COMPOSE`.

Lines 1-2 of the algorithm indicates the interruption of the inference along the path (no possible node can be produced), so it generates a kmesg with NULL as its answer. Lines 3-4 indicate the end of the inference (no further paths need to be tracked), where the system produces the answer and generates a kmesg containing the answer. Lines 5-9 indicate that the system is in the middle of a `COMPOSE` inference, so for each node in the $K_{mesg}.ans$, it generates a new task N_{tk}, and a new ptask NP_{tk} which represents the new inference. The original ptask then transforms its action from `COMPOSE` to `OR` for the answer collection of these new generated tasks and ptasks.

KSTAR/KPLUS

As we have pointed out previously, there is no ptask whose action is marked `KPLUS`, as all `KPLUS` ptasks are transformed into the action `KSTAR` during the initiation of the ptask.

In the `KSTAR` ptask process, the system checks the kmesg nodeset one by one to see if it is in the answer set already or not. If not, a new task is spawned (lines 1-8). This

```
COMPOSE PTASK EXEC(P_tk, K_mesg)
if K_mesg.Ans = NULL
  then Generate K_mesg⟨P_tk.P_id, NULL, D/C⟩
  else if p → left_sibling = NULL
         then Generate new kmesg NK_mesg⟨P_tk.P_id, K_mesg.Ans, D/C⟩
         else p ← p → left_sibling ;
              P_tk.Act ← OR ;
              for each node n∈ K_mesg.Ans
              do Generate new ptask NP_tk⟨Id, P_tk.Id, COMPOSE , NULL⟩
                 Generate new task NT_k⟨NP_tk.Id, p, n⟩
```

Figure 5.9: COMPOSE-ptask-exec

```
KSTAR_PTASK_EXEC(P_tk, K_mesg)
 1  for every node n ∈ K_mesg.Ans
 2  do
 3       /* If the node has in the answer already, we can skip it */
 4      if n ∉ P_tk.Ans
 5         then /* The node is not in the answer, we should record */
 6              /* it and spawn new tasks for it */
 7              P_tk.Ans ← P_tk.Ans ∪ n
 8              Generate new task NT_k⟨P_tk.Id, n, p⟩
 9  if no more floating kmesgs for P_tk
10     then Generate new kmesg K_mesg⟨P_tk.P_id, P_tk.Ans, D/C⟩
```

Figure 5.10: KSTAR-ptask-exec

procedure will continue until no more kmesgs are related to this ptask any more (lines 9-10). In the algorithm stated in Fig 5.10, the p in line 8 is the subsequent path after `KSTAR` / `KPLUS` gotten the answer.

OR

Similar to the ptask for `AND`, the ptask for `OR` is relatively simple: it unions the existing result $P_{tk}.ans$ with the answer of the incoming kmesg $K_{mesg}.ans$. In the mean time, it also keeps track of the number of returned kmesgs. If there is no more floating kmesgs, the sys-

OR_PTASK_EXEC(P_{tk}, K_{mesg})
$P_{tk}.Ans \leftarrow P_{tk}.ans \cup k_{mesg}.Ans$
if no more floating $Kmesg$ for P_{tk}
  then Generate new kmesg $NK_{mesg}\langle P_{tk}.P_{id}, P_{tk}.Ans, D/C\rangle$

Figure 5.11: OR-ptask-exec

tem then generates another kmesg with the answer to its parent ptask. Fig. 5.11 illustrates the algorithm for the `OR` ptask.

AND

As shown in Fig. 5.12, given a ptask and a kmesg, the ptask for `AND` tries to intersect the kmesg with the existing answer and obtains the new result. If the kmesg contains an empty set as its result, the result of the whole operation is the empty set and the system takes a short-cut by sending the answer back and ignoring all the subsequent incoming messages (lines 1-3). Otherwise, if the kmesg is the first incoming message (which means there is no answer in the ptask yet) then by default the incoming answer is inherited by the ptask as the initial result (lines 4-5). The routine intersection is performed if none of the cases above are true (line 6). When the ptask receives all the kmesgs, it sends out a kmesg to its parent ptask (lines 7-8).

NOT

The ptask `NOT` simply returns the difference of all the nodes in the system V and the incoming nodeset inside the kmesg $K_{mesg}.ans$. Fig. 5.13 illustrates the algorithm.

AND_PTASK_EXEC(P_{tk}, K_{mesg})

if $K_{mesg}.Ans$ is empty
  then Generate new kmesg $NK_{mesg}\langle P_{tk}.P_{id}, \mathsf{NULL}, D/C\rangle$
    return
  else **if** K_{mesg} is the first incoming kmesg
    then $P_{tk}.Ans \leftarrow K_{mesg}.Ans$;
    else $P_{tk}.Ans \leftarrow P_{tk}.Ans \cap K_{mesg}.Ans$
if no more incoming kmesg for P_{tk}
  then Generate new kmesg $NK_{mesg}\langle P_{tk}.P_{id}, P_{tk}.Ans, D/C\rangle$

Figure 5.12: AND-ptask-exec

NOT_PTASK_EXEC(P_{tk}, K_{mesg})

Generate new kmesg $NK_{mesg}\langle P_{tk}.P_{id}, V - K_{mesg}.Ans, D/C\rangle$

Figure 5.13: NOT-ptask-exec

FORWARD-MESSAGE

The ptask `FORWARD-MESSAGE` always accompanies either the ptask `RELATIVE- COMPLEMENT` or `RANGE- RESTRICT`. As implied by the name, the `FOREARD- MESSAGE` ptask forwards the answer of the incoming messages to its parent ptask while appending the *Aid* number (either 1 or 2) that was recorded in the ptask when it was born. The *Aid* identifies the order of the incoming messages. Fig. 5.14 shows the algorithm.

RELATIVE-COMPLEMENT

When the incoming kmesgs from `FORWARD-MESSAGE` have arrived, the ptask of `RELATIVE-COMPLEMENT` distinguishes them by the the auxiliary id and stores them at a different lo-

FORWARD-MESSAGE_PTASK_EXEC(P_{tk}, K_{mesg})

Generate new kmesg $NK_{mesg}\langle P_{tk}.P_{id}, P_{tk}.Aid, K_{mesg}.Ans\rangle$

Figure 5.14: FORWARD-MESSAGE-ptask-exec

RELATIVE-COMPLEMENT_PTASK_EXEC(P_{tk}, K_{mesg})
1 **if** $K_{mesg}.Aid = 1$
2 **then** $Ans \leftarrow K_{mesg}.Ans$
3 **else** $Ans_2 \leftarrow K_{mesg}.Ans$
4 **if** no more floating K_{mesgs} for this P_{tk}
5 **then** Generate new kmesg $NK_{mesg}\langle P_{tk}.P_{id}, Ans - Ans_2, D/C\rangle$

Figure 5.15: RELATIVE-COMPLEMENT-ptask-exec

IRREFLEXIVE-RESTRICT_PTASK_EXEC(P_{tk}, K_{mesg})
Generate new kmesg $NK_{mesg}\langle P_{tk}.P_{id}, K_{mesg}.Ans - n, D/C\rangle$

Figure 5.16: IRREFLEXIVE-RESTRICT-ptask-exec

cation (lines 1-3) until all the information has been collected. Then a kernel message is generated with the contents of the nodeset difference (lines 4-5).

IRREFLEXIVE-RESTRICT

As described in the previous section, the ptask returns kmesg with $K_{mesg}.Ans - n$ where n is the starting node of the command `IRREFELEXIVE-RESTRICT`. Fig. 5.16 illustrates the algorithm.

DOMAIN-RESTRICT

As indicated in the previous section, the ptask `DOMAIN-RESTRICT` collects the kmesgs from the spawned tasks and generates the intersection of the nodeset.

RANGE-RESTRICT

The ptask `RANGE-RESTRICT` collects two different kmesgs from the spawned `FORWARD-MESSAGE` ptasks. If the node y is one of the node appearing in the result of the inference

DOMAIN-RESTRICT_PTASK_EXEC(P_{tk}, K_{mesg})

if K_{mesg} is the first incoming kmesg
  then $P_{tk}.Ans \leftarrow K_{mesg}.Ans$;
  else $P_{tk}.Ans \leftarrow P_{tk}.Ans \cap K_{mesg}.Ans$
    Generate new $K_{mesg}\langle P_{tk}.Id, P_{tk}.Ans, D/C\rangle$

Figure 5.17: DOMAIN-RESTRICT-ptask-exec

RANGE-RESTRICT_PTASK_EXEC(P_{tk}, K_{mesg})

if $K_{mesg}.Aid = 1$
  then $Ans \leftarrow K_{mesg}.Ans$;
  else $Ans_2 \leftarrow K_{mesg}.Ans$;
if no more floating kmesgs
  then if $y \in Ans_2$
    then Generate new kmesg $NK_{mesg}\langle P_{tk}.Id, Ans, D/C\rangle$

Figure 5.18: RANGE-RESTRICT-ptask-exec

(Q z), then the result of ((`RANGE-RESTRICT` P (Q z)) y) is equivalent to the result of the inference (P y). The algorithm in Fig. 13 illustrates the procedure.

5.4 Load Balance

Load balancing is one of the primary concerns in designing a parallel system. Without proper arrangement, an idle processor will inevitably drags the system performance downward.

From the perspective of problem structure, problems can be classified into static problems and dynamic problems. Static problems are the problems whose task structures are well defined or predictable before runtime. On the contrary, dynamic problems are the problems with irregular task structures or task granularities. Static problems generally use static scheduling algorithms instead of dynamic scheduling algorithm for optimal performance. Likewise, dynamic problems usually use dynamic scheduling algorithms as their load balancing scheme. Most of the static scheduling algorithms are executed prior to the tasks being executed, and usually the scheduling is performed on a single processor machine.

Recently, parallel scheduling algorithms have emerged, and they try to exploit the power of multiple processors to solve the load balancing problem. Parallel scheduling is a dynamic scheduling algorithm that runs on a parallel machine. Generally speaking, parallel scheduling algorithms can be divided into two phases: the system phase and the user phase. All the PEs need to be synchronized at the beginning of the system phase for collecting system information and mapping tasks to PEs. In the user phase, tasks are executed [59]. Fig. 5.19 illustrates the relation between two phases.

In TERRESA, a parallel dynamic scheduling algorithm, DEM (Dimension Exchange Method) [12] is used as our load balancing algorithm. The DEM algorithm was proposed by Cybenko in 1989. The algorithm runs on the hypercube computer but can easily be modified to fit other network topologies.

5.4.1 DEM Algorithm

Fig. 5.20 is the DEM algorithm. In the algorithm, d stands for the dimension of the hypercube and w_i is the number of tasks in PE_i. The basic idea of the algorithm is: in each iteration of DEM, PE_i tries to exchange its load information and then balance the number of tasks with the kth dimension neighbor PE_j. The iteration will not finish until all d dimension neighbors are visited. This algorithm is not an optimal one due to the fact that

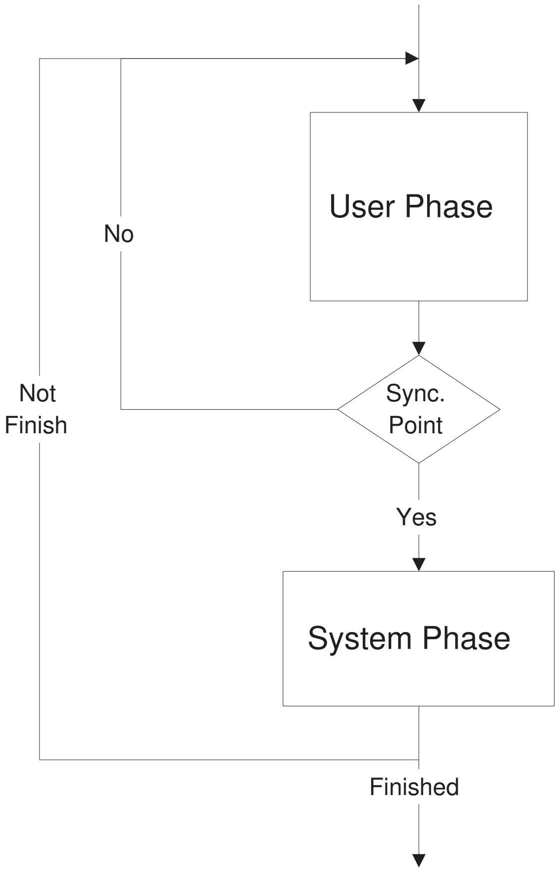

Figure 5.19: The Parallel Scheduling Algorithm

LOAD BALANCE - DEM()

for $k \leftarrow 0$ **to** $d-1$
do
  PE_i exchanges with PE_j the load w_i and w_j , where $j = i \oplus 2^k$
  if $w_i - w_j > 1$
    then PE_i sends $\lfloor (w_i - w_j)/2 \rfloor$ tasks to PE_j
  if $w_j - w_i > 1$
    then PE_i receives $\lfloor (w_j - w_i)/2 \rfloor$ tasks from PE_j
  update the value of w_j

Figure 5.20: DEM Algorithm

the load among the processors is not guaranteed to be equal after the algorithm is applied; nevertheless it is simple, has low overhead, and is efficient.

The DEM algorithm, designing on top of hypercube computers, is a topology dependent algorithm. If the machine connection is constructed in the form other than hypercube, a mapping algorithm is needed. The mapping algorithms from one topology to another have been studied extensively under various circumstance [28, 4, 24, 52]. Fortunately, in current TERRESA implementation, the mapping problem is not significant since the program is run on the SMP workstations. In the TERRESA, the process id of each spawned process is used as the PE id PE_i in the DEM algorithm.

5.4.2 Ptask Migration

Usually the tasks in the TERRESA system are short-lived. They are executed and kernel messages are sent back to the ptasks. If, during the load balance session, only tasks are migrated to the remote PE, it would not be economically sound in terms of time because the communication is significantly slower than the computation in the parallel computers.

On the contrary, ptasks , which generate multiple tasks and kmesgs during their life span, are better candidate for the load balance and migrate to remote PEs.

The migration of the ptasks is not penalty-free, though. When moving a ptask to remote PE, the dependencies of task-ptask-kmesgs in local PE are severed. To keep the dependencies among different objects, some bookkeeping procedures need to be performed so that the inferential correctness can be maintained.

Since the connections between task-ptask-kmesg are maintained by the *Pid* in each of the data structures, the coherent can be maintained if all the *Pid* in the task-ptask-kmesg are updated in reflect the correct scenario after migration. Fig 5.21 shows the ptask migration/updating algorithm used in the TERRESA load balance module. The extra code (lines 7-13, 16-22) are for updating *Pids*.

5.4.3 Affinity DEM

In a parallel system, load balance module moves tasks from one PE to another. Theoretically when load are not balanced, the best way to optimize system performance is to even up the number of tasks among PEs. When moving tasks, it takes time to pack and unpack information. Also, the moving of tasks implies the possibility of introducing new knowledge faults in the destination PE, new overhead might be incurred. When these overheads overshadow the benefits gained from load balancing, a new scheduling strategy should be considered.

One load balance policy considering the overhead is the *affinity based scheduling* [38]. In affinity based scheduling, a task is bound to a specific PE until it is completed. This

PTASK MIGRATION ALGORITHM()

Initialize $T_{id}Set$
for $k \leftarrow 0$ **to** $d-1$
do
    PE_i exchanges with PE_j the load w_i and w_j , where $j = i \oplus 2^k$
    if $w_i - w_j > 1$
      then PE_i send $\lfloor (w_i - w_j)/2 \rfloor$ ptasks to PE_j
          receive $T_{id}Set$ from PE_j
          /* Update pid in the task queue */
          $task\_id\_update(TQ, T_{id}Set)$
          /* Update pid in the kernel message queue */
          $task\_id\_update(KMQ, T_{id}Set)$
          /* Update pid in the ptask pool */
          $task\_id\_update(PTP, T_{id}Set)$
    if $w_j - w_i > 1$
      then PE_i receive $\lfloor (w_j - w_i)/2 \rfloor$ ptasks from PE_j
          for each new arrival ptask P_{tk}
          do
              $oldid = P_{tk}.mid$
              $newid = getnewid()$
              $P_{tk}.Id = newid$
              insert $oldid, newid$ into $T_{id}Set$
              send $T_{id}Set$ to PE_j
    update the value of w_j

Figure 5.21: The Ptask Load Balance Algorithm

strategy saves significant amount of time in not moving information and the faults can be eliminated. However, since tasks cannot be scheduled into another PE, the system might suffer from load imbalance: some PEs are busy executing the tasks while others are sit idle. A looser form of the affinity based scheduling algorithm allows certain amount of tasks being moved to its neighbor. An evaluation function should be established in the algorithm by comparing the benefit gained from migrating tasks to new PE. The affinity DEM (ADEM) algorithm is listed in Fig. 5.22.

Table 5.1 and Fig. 5.23 shows the experiment of controlling maximum ptasks allowed to move from one PE to another. In the experiment, instead even up the ptasks difference between PEs, an upper limit is set to reduce the migration overhead. In this experiment, a query

$$\texttt{(find (subclass- supclass} \ldots \texttt{subclass- supclass)} \ \mathtt{B}_1\texttt{)}$$

with path length 26 is issued on the 8K single inheritance classification tree [1]. The "∞" in the table indicates that the threshold is not set, or the original DEM algorithm. Similarly, "No LB" indicates that no load balance is performed during the execution.

The experiment shows that with a proper control of the number of migrating ptasks, load balance does boost performance. The best experiment result can be achieved by moving either 8 (2 PEs) or 16 (4, 6, and 8 PEs) ptasks during each iteration. In TERRESA, the maximum ptasks allow to move is set to 12, the average of 8 and 16.

[1]The definition of single inheritance classification tree can be seen on page 113

LOAD BALANCE - AFFINITY DEM()

$t \leftarrow Threshold$
for $k \leftarrow 0$ **to** $d-1$
do
    PE_i exchanges with PE_j the load w_i and w_j , where $j = i \oplus 2^k$
    if $w_i - w_j > 1$
        then $m \leftarrow min(\lfloor w_i - w_j/2 \rfloor, t)$
            PE_i sends m tasks to PE_j
    if $w_j - w_i > 1$
        then $m \leftarrow min(\lfloor w_i - w_j/2 \rfloor, t)$
            PE_i receives m tasks from PE_j

Figure 5.22: Affinity DEM Algorithm

Number of ptasks can be migrated	Time (sec.)			
	2 PEs	4 PEs	6 PEs	8 PEs
1	33.85	30.04	42.11	44.20
2	31.81	29.20	36.87	49.29
4	27.65	19.46	17.65	31.43
8	21.02	8.48	6.97	31.33
16	16.45	12.07	20.92	40.74
32	40.25	13.16	14.62	51.92
64	40.97	13.03	23.93	65.23
128	36.02	12.70	22.92	82.19
256	37.86	33.59	32.75	112.37
∞	183.02	73.91	57.55	103.64
No LB	35.33	34.45	37.96	46.12

Table 5.1: Affinity DEM Experiment Result

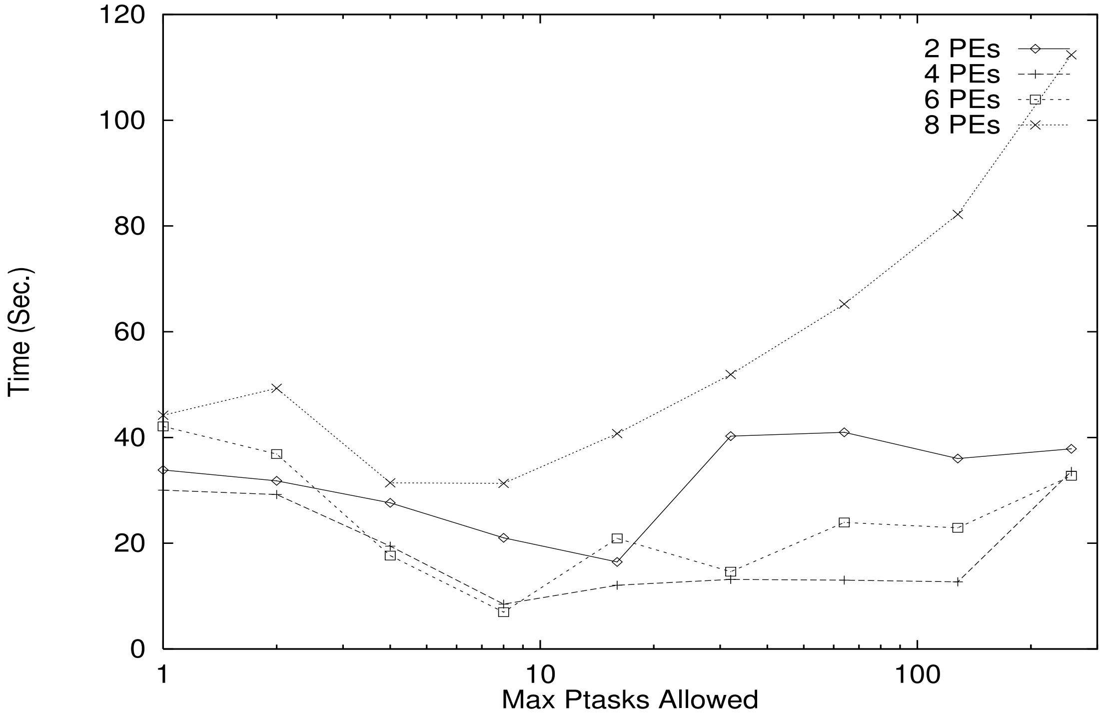

Figure 5.23: Affinity DEM Experiment Result

Chapter 6

Shared Knowledge Management

One of the greatest advantages of distributed memory parallel computers have over sequential machines is that they have the potential to extend their memory capacity virtually without limitation. As the number of PEs increases, so does the memory. However, accessing remote memory in distributed memory machines is both time consuming and hard to code. For a semantic network system implemented on distributed machines, the sharing among PEs have never been an easy task due to the interconnectivity and the irregularity topology of the knowledge base. Without proper system support, accessing the nodes in a remote PE is chaotic in both the system performance and the programming. Similar to distributed shared memory systems, a layer of API for supporting shared knowledge is implemented so that it makes knowledge access transparent and the process design of TERRESA easier.

Research of distributed shared memory system is reviewed in Section 6.1. The architecture of the TERRESA shared knowledge base is then introduced in Section 6.2.1.

6.1 Overview of Distributed Shared Memory

Traditional parallel computers can be categorized into two groups by their memory access methodology. One methodology is shared memory, which includes machines like Encore Multimax, Sequent Symmetry, and BBN Butterfly, which adopt bus (or multiple bus) or switching network architectures with a single address space. The other methodology is distributed memory, which includes machines like the Intel iPSC series, TMC Connection Machine series, which connect tens to hundreds of PEs by various topologies of networks with explicit send/receive remote memory access primitives.

While benefiting from easy prototyping, shared memory machines suffer from their lack of scalability. Generally speaking, the scalability of a shared memory machine is no more than ten to twenty PEs, since over this amount the performance degrades rapidly. On the other hand, distributed memory machines have the merits of good scalability, but the data passing model makes distributed memory machine programs hard to write. Therefore, a new trend - distributed shared memory, which combines the two technologies, has emerged.

Distributed shared memory systems are a trade-off between shared memory systems and distributed memory systems; they support a single virtual address space for all PEs so the programmer can craft a parallel program efficiently while good scalability is maintained by system support. The main issues in implementing distributed shared memory are memory coherence for different memory segments residing in different PEs, and eliminating the system overhead of supporting a distributed shared memory mechanism, or in other words, latency hiding for remote memory access.

There are three approaches to implementing the distributed shared memory:

1. Direct hardware support;

2. An operating system implementation of the kernel or run-time library support by a technique similar to the virtual memory technique in the traditional sequential computer;

3. Compiler translation of shared memory into distributed memory by communication and synchronization primitives insertion.

   Some systems provide a hybrid of these three.

6.1.1 Design Issues of Distributed Shared Memory

There are several things to consider when designing a distributed shared memory system: the unit of shared memory structure and granularity, memory coherence, and the heterogeneity of the system are all important issues.

1. Structure and Granularity: Structure refers to the layout of shared data in memory, while granularity refers to the size of the unit used for sharing. Some systems implement distributed shared memory like virtual memory systems, and therefore memory is transferred in fixed page sizes while other systems design them in an object-oriented style, so the unit being transferred, an object, varies in size.

   Generally speaking, because of locality, distributed shared memory programs favor accessing a large region of their shared address space in a small amount of time.

Therefore, larger page sizes reduce paging overhead. However, from another point of view, larger page sizes reduce the number of pages, so if the memory access pattern is not sequential or regular, the possibility of a page being swapped out increases, which degrades system performance dramatically.

2. Coherence: Distributed shared memory coherence can be divided into several approaches from the strong to weak order:

   - Strict consistency: a read operation returns the most recently written value;
   - Sequential consistency: the result of any execution appears as some interleaving of operations of individual nodes when executed on a multi-threaded sequential machine;
   - Processor consistency: Writes issued by each individual PE are never seen out of order, but the order of writes from two different PEs can be observed differently;
   - Weak consistency: The programmer enforces consistency using synchronization operators guaranteed to be sequentially consistent;
   - Release consistency: Weak consistency with two types of synchronization operators: acquire and release. Each type of operator is guaranteed to be processor consistent.

3. Scalability: The scalability of distributed shared memory is generally limited by two factors: central bottlenecks (such as centralize control over the write operation), and

global common knowledge operations and storage(such as broadcast of invalidation).

4. Heterogeneity: There are several distributed shared systems that provide heterogeneity of computing power, such as *Agora* and *Mermaid*. Although the heterogeneity design of distributed shared memory systems might increase the program performance by adding more processor power, the data format conversion usually overshadows its benefits, and therefore we generally won't consider it as an option.

6.1.2 Algorithms for Distributed Shared Memory

There are four basic algorithms for implementing shared memory in a distributed memory environment. Depending on the data migration strategy, the algorithm can be divided into either a replication or a non-replication algorithm. In the following paragraphs, we discuss the *read* and *write* operations. When a *client*, needs info about a specific variable, it sends a request to a specific PE; the *server*, where the data resides, feeds the data back to the client[63].

Central Server Algorithm

The simplest strategy for implementing distributed shared memory is to use a central server, which is responsible for serving all accesses to shared data and maintaining the copy of the shared data. Both the read and write operations involve sending a request message to the data server. One obvious problem with the algorithm is that the server may become a bottleneck, since the server has to serve the requests from all clients. Alternatives are proposed, as we can have several central servers and partition the shared data among them, so that when a request is issued, we can either broadcast requests or use some mapping mechanisms to locate the correct position of the data.

Migration Algorithm

In the migration algorithm, the system keeps only one copy of the data and always migrates to the site where it is accessed. Because of the data locality and the communication overhead, the data usually migrates in blocks instead of in a single piece.

The second advantage of this algorithm is that it can be integrated with the virtual memory system of the host operating system if the size of the block we choose is the same as the size of the virtual memory page. When the page is stored locally, the application can use normal memory access. When the page is stored on the remote site, the memory access triggers a page fault which brings the page to local memory.

One potential disadvantage of this algorithm is the possibility of page thrashing between two or more PEs, which would result in poor performance.

Read-Replication Algorithm

One disadvantage of the algorithms mentioned above is that only one thread in the system can access any block of data at any given time, even though several threads may want to access the same data while not wanting to modify the data. Storing multiple copies of the data can eliminate this disadvantage. However, the write operation will be more expensive than the algorithms mentioned above since data consistency should be kept.

For the read operation on data items which are not local, a communication with the remote PE is needed to first acquire a read-only copy of the block and try to change the privilege to writable if necessary before the read-operation is complete.

For the write operation to data items which are neither local nor have write permission, all copies of the same block held at other sites must be invalidated before the write can proceed. The algorithm for the write operation is very similar to the cache coherence algorithm.

Full-Replication Algorithm

Full replication allows data blocks to be replicated even while being written to. One possible way to keep the replicated data consistent is to globally sequence the write operations, while only sequencing the read operations relative to the writes that occur local to the site where the reads are executed.

One simple example illustrates as follows: when a process attempts a write to shared memory, the intended modification is sent to the sequencer. This sequencer assigns the next sequence number to the modification and broadcasts the modification with this sequence number to all sites. When a site receives the modification signal, it updates its local memory.

6.2 TERRESA Shared-Knowledge Module Design

Contrary to the DSM system's policy of using pages of bytes as the basic sharing unit, TERRESA takes a more abstract approach. The basic sharing unit in the TERRESA shared-knowledge module is a "block" SNePS nodes. Each block of SNePS nodes is composed of a certain amount of SNePS node information. The shared-knowledge module tries to imitate the full-replication algorithm stated above. But since the path-based inference will not generate new nodes during inferencing, which means there is no writing action, the write coherence checking can be dropped. This property greatly simplifies the current design of the shared knowledge module in TERRESA.

6.2.1 Knowledge Base

Inspired by the architecture of shared memory machines, the TERRESA knowledge base can be split into two sections: *private knowledge* and *cache knowledge*. Figs. 6.1 and 6.2 show the structures of the private knowledge system and the cache knowledge system, respectively. The difference between these two types of knowledge systems is that once a node enters the private knowledge system, it resides there permanently, while the nodes in the cache knowledge system have the possibility of being swapped out due to capacity limitations. For each of the knowledge systems, two major components are embedded: the knowledge base and the control data structure. As indicated by the name, the knowledge base stores the TERRESA nodes and arcs, and the control data structures store control information for accessing the knowledge base.

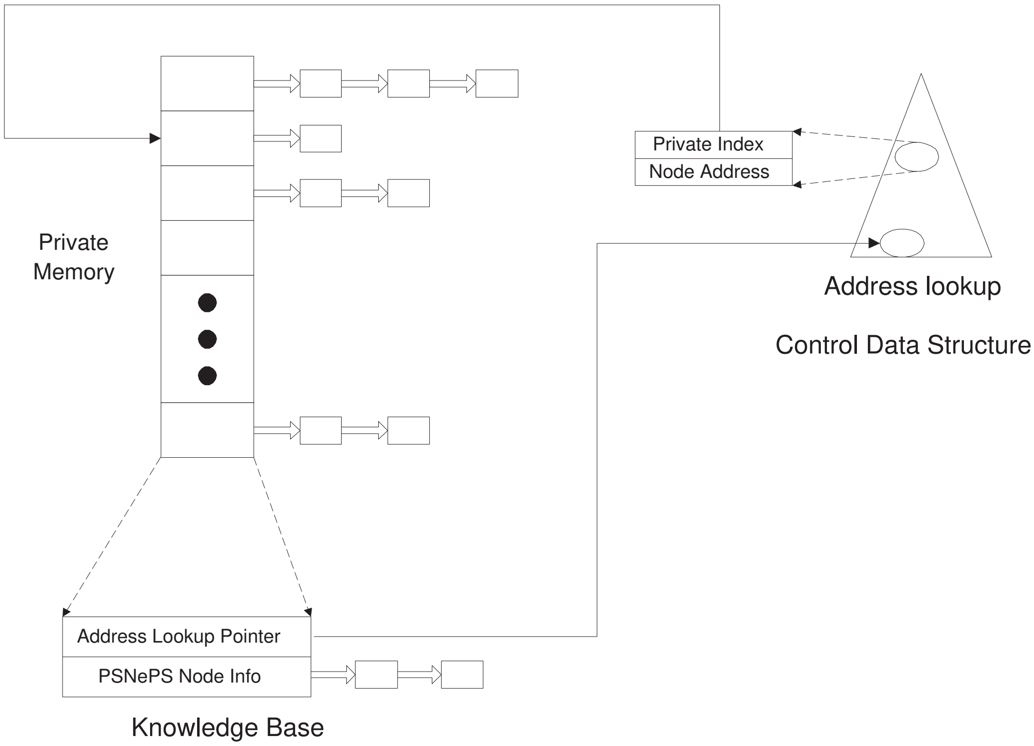

Figure 6.1: Structure of Shared Knowledge System, Private Knowledge

As indicated in the figure, the cells of the private knowledge and the cache knowledge store the same information, and the SNePS cablesets and a pointer which points to the node in the address look-up tree. In addition to these entries, the cache knowledge stores a pointer that points to the priority queue, which maintains the access records of the cache knowledge entries.

As for the control data structure, both private knowledge and cache knowledge keep an address lookup data structure for quick access. In our implementation, a binary tree is used to stores the node addresses and their corresponding index in the knowledge base.

Moreover, each time a knowledge entry in the cache knowledge is accessed, the access record of the priority queue element is updated. When there is no more room in the

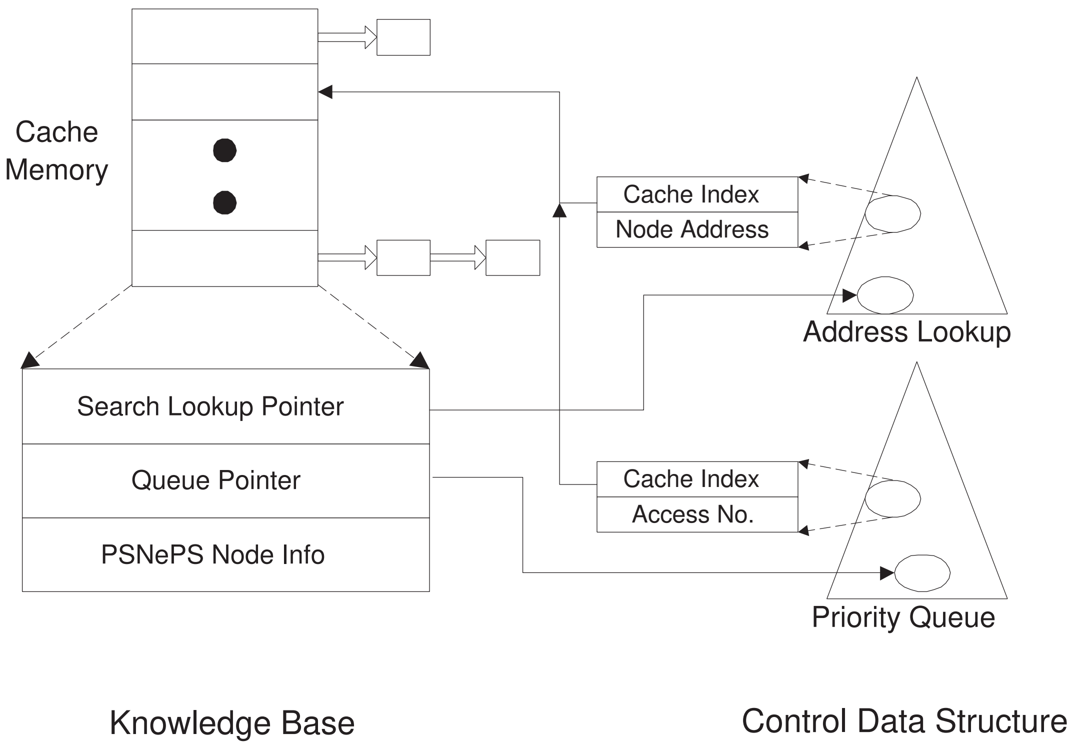

Figure 6.2: Structure of Shared Knowledge System, Cache Knowledge

cache, the control mechanism searches the priority queue and decides which entry should be swapped out. The least recently used (LRU) entry is selected as the target and is replaced in the TERRESA system.

To reduce the number of accesses across different PEs, the node entries are bundled into blocks according to their index within the private knowledge similar to the virtual memory systems of modern computers.

6.2.2 Control Modules

There are two control modules related to knowledge access, both governed by the scheduler: the knowledge task processor (KTP) and the knowledge task receiver (KTR). The

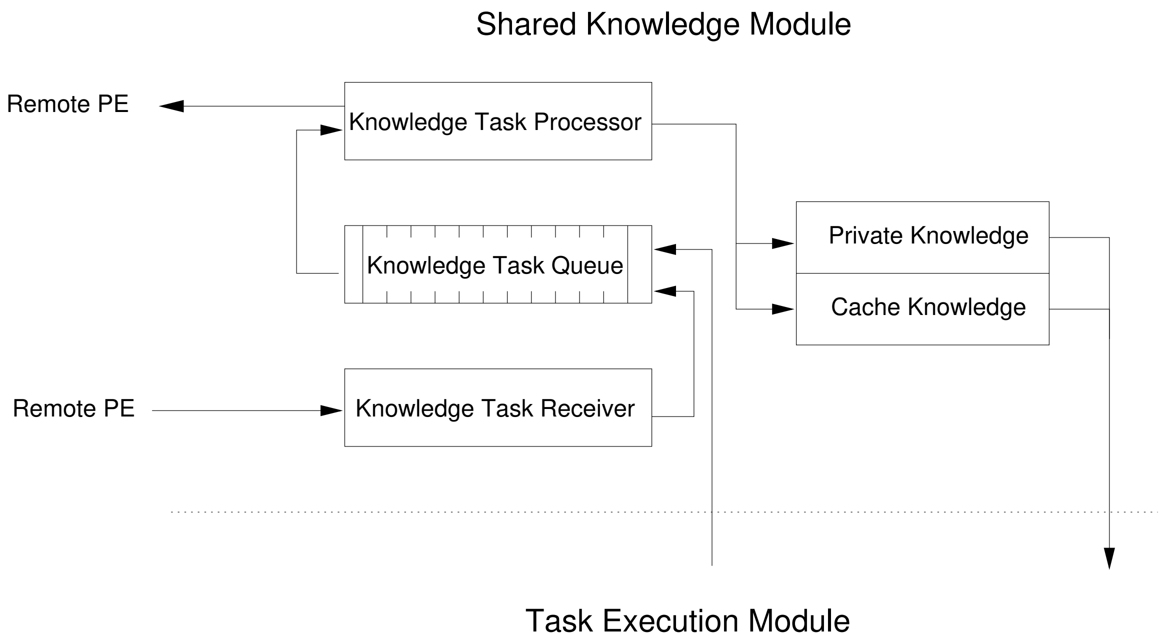

Figure 6.3: Knowledge Modules Architecture

knowledge task processor fetches knowledge tasks from the knowledge task queue (KTQ) and executes them according to the action defined within the knowledge task. The knowledge task receiver checks the incoming system buffer and tries to find any incoming knowledge task from other PEs. If any exist, the knowledge task receiver receives them from the system buffer and then puts them into the knowledge task queue. Fig. 6.3 shows the connections between the control modules and the data structures.

6.2.3 Knowledge Tasks

When a TERRESA task wants to access a node in the knowledge base and it is in a remote PE, a knowledge task is issued. Abstractly, a knowledge task consists of several entries, $\langle Act, Add, PE_s, PE_r, KB, c\rangle$ where Act is the action of the knowledge task, Add is the node address to be operated on (retrieved), PE_s is the task-sending PE, the PE_r is the task receiving PE, and *KB* is a set of nodes which are transferred along with the task. Usually the entry *KB* will not be filled if the task is not a fault reply task, and *c* is the size of *KB*.

The entry *Act* indicates what kind of task it is. The task action could be one of the following values:

1. `KNOWLEDGE_FAULT_REQ`: a knowledge task indicates that the given node address cannot be found in the local knowledge base, so a fault is raised. The KTP sends the knowledge task to all the PEs that participate in the computation, except for itself. Fig. 6.4 shows the algorithm for the action `KNOWLEDGE_FAULT_REQ`.

2. `KNOWLEDGE_FAULT_RPY`: an incoming knowledge access task indicates that a remote PE cannot find the given node in its local knowledge base, so it sends out a query task and waits for the reply. The KTP checks whether the given address is in the private knowledge section or not. If it is not in the private knowledge section, the request is discarded. Otherwise, the KTP bundles a block of the semantic nodes which contains the requested one into a memory task and replies to the query; Fig. 6.5 shows the algorithm for the action `KNOWLEDGE_FAULT_RPY`.

3. `KNOWLEDGE_FAULT_RCV`: an incoming task which indicates the replied task from a remote PE for the knowledge fault query. The KTP tries to allocate a space in the cache knowledge. If the cache runs out of space, the PE looks at the priority queue and determines which entries should be discarded. After allocating space, the incoming nodes are then allocated into the proper location. Fig. 6.6 shows the algorithm for the action `KNOWLEDGE_FAULT_RCV`.

```
KNOWLEDGE_FAULT_REQ(Add)
KT_k.PE_s ← PE_me
KT_k.Add ← Add
KT_k.Act ← MEMORY_FAULT_RPY
for i ← 1 to numprocs
do if i ≠ PE_me
      then KT_k.PE_r ← i;
           send(KT_k, i);
```

Figure 6.4: KNOWLEDGE_FAULT_REQ Algorithm

```
KNOWLEDGE_FAULT_RPY(KT_k)
add = KT_k.Add;
if (idx ← private_search(private, add)) ≠ NULL
  then PE_rpy ← KT_k.PE_s
       KT_k.Act ← KNOWLEDGE_FAULT_RCV
       KT_k.PE_s ← PE_me
       KT_k.PE_r ← PE_rpy
       start_idx ← idx/ KNOWLEDGE_BLOCK_SIZE * KNOWLEDGE_BLOCK_SIZE
       KT_k.c ← 0;
       for i ← 1 to KNOWLEDGE_BLOCK_SIZE
       do if private[start_idx + i] ≠ empty
             then KT_k.c ← KT_k.c + 1
                  KT_k.KB[i] ← private[start_idx + i]
       send(KT_k, PE_rpy);
```

Figure 6.5: KNOWLEDGE_FAULT_RPY Algorithm

```
KNOWLEDGE_FAULT_RCV(KT_k)
if (idx ← cache_alloc(cache)) = NULL
  then idx ← priority_queue_extract_min(cache);
for i ← 1 to KT_k.c
do cache[idx + i] = KT_k.KB[i];
```

Figure 6.6: KNOWLEDGE_FAULT_RCV Algorithm

6.2.4 Prefetching

In distributed shared memory systems, two schemes, prefetching and delay updating, are frequently used as the way to increase system performance. Because there is no write action in the TERRESA system, we focus on the action of prefetching only.

Prefetching tries to fetch data from remote PEs before it is actually used. When memory faults occur, instead of fetching the requested data alone, the DSM system fetches the vicinity data altogether from the remote PEs. Because of the data locality, the communication overhead can be reduced.

In the TERRESA system, the knowledge bases in the slave PEs are divided into *blocks*. A block is a cluster of SNePS nodes allocated in the adjacent location. When a knowledge task is issued, a block of nodes, including the requested one, is moved from the remote private knowledge base to the local cache knowledge base. Yet unlike the data entries in the distributed shared memory system, the nodes are arranged according to the sequence of the appearance presented to the TERRESA system. The adjacent nodes might not be interconnected to each other. Aware of this, it is the users responsibility to arrange the assertions so that related nodes can be clustered into blocks.

Currently TERRESA uses 128 nodes as its block size.

6.2.5 Access Elimination

The TERRESA system is a multithreaded environment, which means the system can execute multiple tasks simultaneously. This indicates that when executing a query, it can ac-

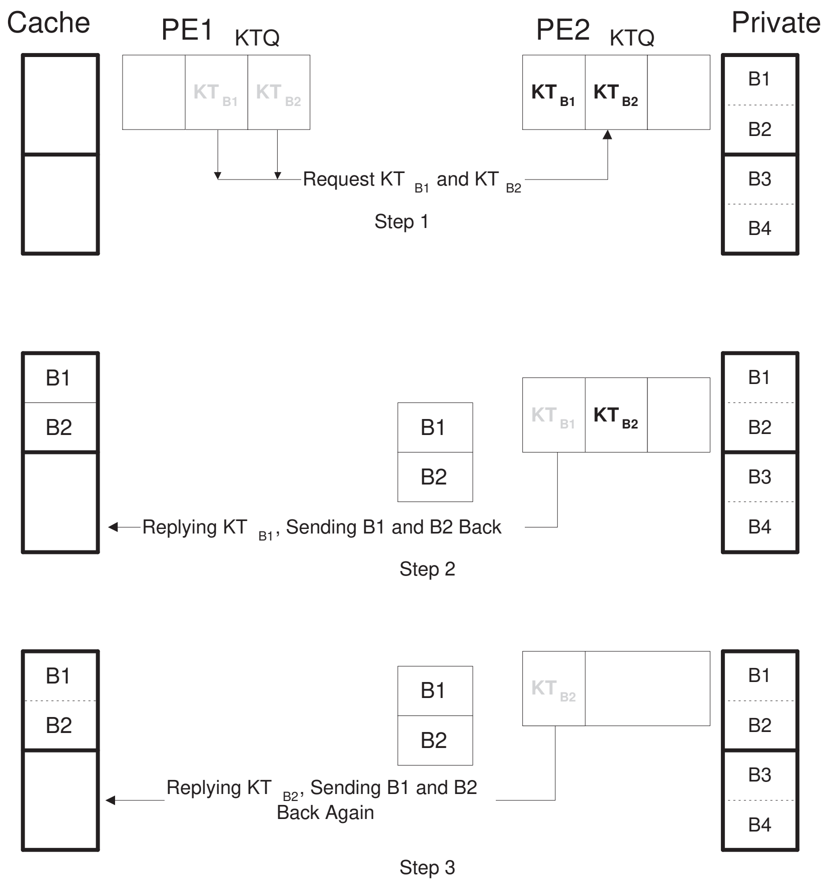

Figure 6.7: Multithreading Knowledge Access

cess multiple nodes at the same time. Because of the latency of remote knowledge access, it is very likely that the PE sends out multiple knowledge tasks for accessing the nodes residing at the same block before the reply task comes back.

As illustrated in Fig. 6.7, at the beginning there are three knowledge tasks trying to access B1 and B2 at PE_1, respectively. During an execution cycle, B1 and B2 requests are sent to PE_2. Inside the private knowledge of PE_2, both B1 and B2 are residing on the same block. After receiving the requests, PE_2 replies with two identical loaded knowledge tasks. All the redundancy communication over the network is considered overhead, which wastes computation resources. According to estimation, the size of a single knowledge task without any nodes attached to it is about 1K bytes. A fully loaded knowledge task could be up to 5K - 6K bytes or more. The shuffle of the duplicate knowledge tasks indeed produces a performance nightmare. Two schemes are used in this system to prevent the overlap of memory block communication: *last-minute checking* and *destination memorization*.

1. Last-Minute Checking: This scheme is to ensure that the request is not in cache before the KTP sends the task to the remote PE. The function of the scheme is to eliminate the redundant tasks due to network latency. Doing this can partially eliminate redundant requests at the receiver side.

2. Destination Memorization: This scheme memorizes the destination PE of each block in private memory. Doing this can eliminate redundant replies at the sender side.

Chapter 7

Duplication Management

7.1 Introduction

Task duplication is an issue that needs to be addressed in parallel systems when tasks are distributed among PEs. Occasionally, tasks generated on different PEs might try to solve the same problem, causing redundancy. When identical tasks are generated on different PEs, not only are the tasks duplicated, but the descendents of the identical tasks are duplicated as well. The duplicate checking system tries to check for this phenomenon and to prevent the performance degradation.

The design of the duplicate checking system is also based the task-based, message-driven model with the assistance of two level hashing. In order to avoid the bottleneck and improve the performance, task informations and their results are distributed among processors instead of being centralized at a specific PE. Every checked task must go through a two-level hashing procedure: the *interprocessor* hashing and the *intraprocessor* hashing, to identify the duplications. The interprocessor hashing procedure decides to which PE the task should go to, and the intraprocessor hashing maps the task to a specific location of the hash table when the duplicate checking task arrives.

Described in Chapter 5, the response of the duplicate checking query, either from a local PE or from remote PEs, could be one of the following conditions: the PE has to solve the query itself (`DUP_CHK_SELF_SOLVE`), the task has been solved (`DUP_CHK_SOLVED`), or some other PE is trying to solve it at the time, but the answer it yet to be achieved (`DUP_CHK_WAIT`). If the response is in the last case, another answer message will be received in the future in responding to this query. This functionality is similar to the *future* in the parallel lisp implementation.

Not every task needs to be checked, though. From the node point of view, only the nodes with multiple fan-in arcs and one or more fan-out arcs need to be checked. From the path point of view, since the inference of an arc can be made relatively quickly, only the predefined path symbols need to be checked. Because duplicate checking could be time-consuming, the system default is set to "off", although users can turn it on if the characteristics of the knowledge base are known in advance.

7.2 The Program Model

Quite a few components comprise the duplicate checking module. There are three data receivers: the *duplicate query receiver*, the *duplicate response receiver* and the *duplicate answer register message receiver*. There are also four storage data structures: the *dupli-*

cate answer register message queue, the *duplicate query task queue*, the *duplicate answer repository*, and a *duplicate task hashing table*. Lastly, there are the execution modules: the *duplicate checking engine* and the *duplicate checking answer registrant*. All the receivers and the duplicate checking engine are governed by the scheduler and activated periodically. Fig. 7.1 shows the interconnection of the these components.

7.2.1 The Receivers

There are three receivers in the duplicate checking module corresponding to the different types of messages: the *duplicate query receiver*, the *duplicate response receiver* and the *duplicate answer register message receiver*. Each of the receivers act as a daemon and receive a specific kind of task or message.

7.2.2 Data Storage Repositories

The data storage repositories serve as temporary storage place for incoming tasks in case the execution thread is not available. Similar to the receivers, there are three data structures for storing the duplicate checking related data and information, either from remote PEs or from local PEs. The *duplicate query task queue*, the *duplicate answer register message queue* and the *duplicate response repository*. Because the *duplicate response repository* is designed to be searchable so that the match between the task and the response can be identical, it is implemented in the form of a binary tree. The other two are designed in the form of FIFO queues.

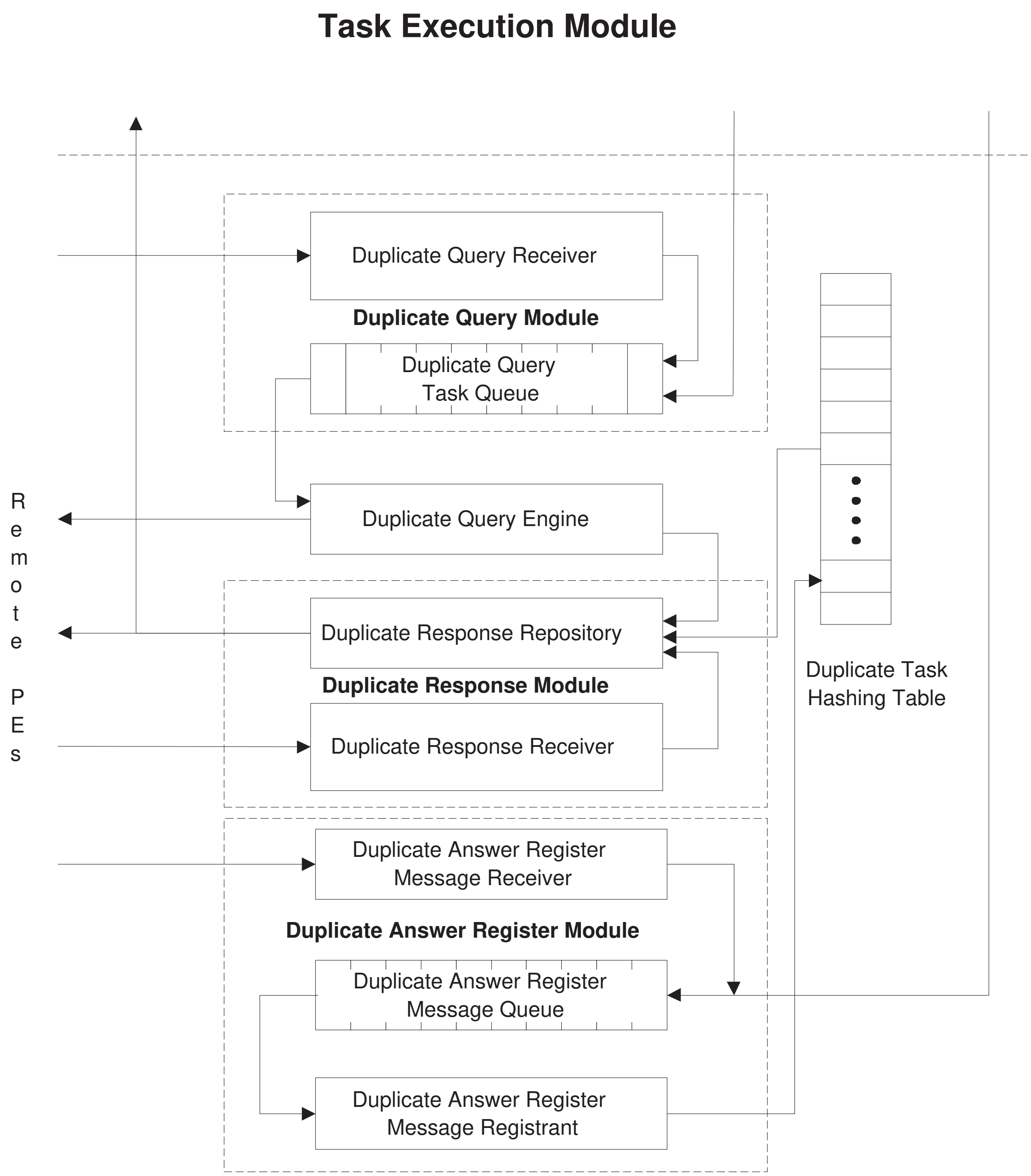

Figure 7.1: Duplicate Checking Module Architecture

7.2.3 Duplicate Checking Hash Table

The primary purpose of the duplicate checking hash table is to store the information of the task and its solution. The duplicate checking hash table stores several kinds of information: the information related to the task in case of hash collision, the status of the entry, either the answer has been obtained or not, and also the answer. Each PE possesses a hash table so that the answer can be distributed among them and avoid being a bottleneck.

7.2.4 Duplicate Checking Engine

The duplicate checking engine fetches the information from the duplicate checking task queue one task at a time. It checks the local hash table and determines if the answer has been reached. A reply message is sent back to the inquirer with a response to the query.

7.2.5 Duplicate Checking Answer Registrant

This unit is activated when a solution is found by the task execution engine that wants to "register" the answer into the hash table. Depending on the number of queries for this task, multiple reply messages may be sent to all the inquirers for the pending duplicate checking tasks.

7.3 The Algorithms

Formally, a duplicate checking task is a tuple of $\langle T_{id}, N, P, H_{id} \rangle$ where T_{id} is the task id recorded in the given task, N is a node, P is a path and H_{id} is a hash table index which contains a pair of addresses $\langle PE_h, Idx_h\rangle$.

Totally there are k duplicate checking hash tables DHT with size $1 \ldots idx$ resides in each $PE_1 \ldots PE_k$. Each entry of $DHT[i]$ is a tuple of $\langle P, N, S, Ans, RSet\rangle$. P and N, as usual, are path and node, respectively. S is the status of this entry, which could be either `DUP_HASH_TABLE_AVAILABLE`, which indicates the hash table entry is not used or `DUP_HASH_ TABLE_IDENTICAL`, which indicates the hash table entry has been visited previously by other duplicate query tasks. Ans records the answer if the task related to P and N has been solved. $RSet$ is a set of task ids which have requested duplicate checking for a specific P and N if the task has not been solved at that moment.

Likewise, a duplicate query reply is a tuple of $\langle T_{id}, S, Ans\rangle$ where T_{id} is the task id similar to the one in the duplicate checking task, Ans is the answer nodeset and S is the status of the answer. The status S can be one of the following: `DUP_CHK_SELF_SOLVE`, `DUP_CHK_WAIT_ANSWER`, or `DUP_CHK_SOLVED`, which represent three different possibilities of the query answer.

The duplicate answer register message is a message generated when the answer of a task is found by the solving PE. Its primary purpose is to serve as a trigger for the answer registration of the hash table and the answer updating for all the requesting PEs. It is composed of a tuple of $\langle H_{id}, Ans \rangle$ where $H_{id} = \langle PE_h, Idx_h\rangle$ is the hash table index and Ans is the answer nodeset it registers.

When a task T_k is scheduled to go through the duplicate checking phase, a duplicate checking task D_{tk} is generated, where $D_{tk}.T_{id} = T_k.id$, $D_{tk}.N = T_k.N$, $D_{tk}.P = T_k.P$ and $D_{tk}.H_{id} = h(Tk.P, Tk.N) = \langle PE_h, Idx_h\rangle$, where h is a hash table index generation function.

By examining PE_h of H_{id}, the duplicate checking task D_{tk} could be sent either to a remote PE (PE_h) or enqueued into the local *duplicate query task queue*. By cross-examining duplicate checking hash table entry $DHT[Idx_h]$, a duplicate checking response D_{rpy} is sent back to the duplicate checking task originator. Fig. 7.2 illustrates the algorithm.

As shown in Section 5.3.3, in the mean time, the execution engine of the task execution module tries to match the returned duplicate reply D_{rpy} in the duplicate reply repository. The algorithm listed in Fig. 5.8 performs the action.

There are three conditions for the duplicate checking reply, D_{rpy}. If the $D_{rpy}.S$ is `DUP_CHK_SELF_SOLVE`, the task T_k is enqueued as a regular task. But when the conclusion is reached, an answer register message is issued to put the answer into the hash table. Upon receiving the message, the *Ans* of $HDT[i]$ is updated, so each of the requesting PEs get a message describing the answer. This part of the action is performed by the *Duplicate Answer Register Module*. Fig. 7.3 describes the algorithm.

The other condition of $D_{rpy}.S$ is `DUP_CHK_SOLVED`, which means the task has been solved by some other PE in the past. The accompanied answer of D_{rpy} can be put into the kmesg as the answer to the task.

The last condition is $D_{rpy}.S$ equals `DUP_CHK_WAIT`. Under this condition, the task keeps its status and return to the task queue for future examination. When the PE of `DUP_CHK_SELF_SOLVE` task finishes its execution, it then submits a duplicate reply message D_{rpy} at line 9 of the algorithm in Fig. 7.3. The incoming message will serve as the solution of the task and a kmesg is back to the parent ptask.

DUPLICATE CHECKING ENGINE(D_{tk})

$h_{idx} \leftarrow D_{tk}.H_{id}.h_{idx}$
$status \leftarrow status\_checking(DHT[h_{idx}])$
switch $(status)$
  case `DUP_HASH_TAB_AVAIL :`
      $DHT[h_{idx}].S \leftarrow DUP\_CHK\_WAIT$
      $D_{rpy}.S \leftarrow DUP\_CHK\_SELF\_SOLVE$
  case `DUP_HASH_TAB_IDENTICAL :`
      if $DHT[h_{idx}].S = DUP\_CHK\_WAIT$
        then
            /* Some PE is solving the task but the answer is yet to be achieved */
            $D_{rpy}.S \leftarrow DUP\_CHK\_WAIT$
            Add $D_{tk}.T_{id}$ to $DHT.RSet$
        else
            $D_{rpy}.S \leftarrow DUP\_CHK\_SOLVED$
            $D_{rpy}.Ans \leftarrow DHT[h_{idx}].Ans$
$D_{cpu} \leftarrow D_{tk}.T_{id}.cpu$
if $D_{cpu} \neq myid$
  then Send D_{rpy} to D_{cpu}
  else Insert D_{rpy} into local *Duplicate Response Repository*

Figure 7.2: Duplicate Checking Engine Algorithm

DUPLICATE CHECKING ANSWER REGISTRANT(D_{AReg})

$h_{idx} \leftarrow D_{AReg}.H_{id}.Idx_h$
$DHT[h_{idx}].Ans \leftarrow D_{AReg}.Ans$
$DHT[h_{idx}].S \leftarrow DUP\_CHK\_SOLVED$
for every $RT_{id} \in DHT[h_{idx}].RSet$
do
    $D_{rpy}.T_{id} \leftarrow RT_{id}$
    $D_{rpy}.S \leftarrow DUP\_CHK\_SOLVED$
    $D_{rpy}.Ans \leftarrow D_{AReg}.Ans$
    $D_{cpu} \leftarrow RT_{id}.cpu$
    if $D_{cpu} \neq myid$
      then Send D_{rpy} to D_{cpu}
      else Insert D_{rpy} into local *Duplicate Response Repository*

Figure 7.3: Duplicate Checking Answer Registrant Algorithm

Chapter 8

Miscellaneous Modules

In this chapter, the focus is put on the modules which have not mentioned previously. Specifically, the *object oriented packing system* and the *slave scheduler*, which were not discussed in the previous chapters.

8.1 Object Oriented Packing System - OOPS

The conventional message-passing systems, either the system calls provided by the vendors or the third party communication software packages such as PVM, only provide essential functions to transfer basic types of variables such as characters, integers, and floating point numbers from one PE to another. Although MPI provides functions to create derived data structures, it won't be easy to create a type template for a dynamic data structure like a linked lists, or composite data structures with various pointers, which unfortunately are quite common in AI program implementations. Secondly, one of the purposes of developing OOPS is to reduce the number of communications. Depending on the complexity of the data structure, without proper packing, sending a sophisticated data structure might involve hundreds, if not thousands of communication calls. Another reason is to eliminate the data format conversions between representation formats (such as *endian*), between heterogeneous computer systems.

The idea of the OOPS system is simple: the programmer should distinguish the pack/unpack function calls from send/receive function calls. Pack/unpack functions are related to data structures and applications, while send/receive functions belong to basic system facilities. In the implementation of OOPS, a sending buffer as well as a receiving buffer is kept. When the user packs data into the system, OOPS translates it into byte codes and stores them in the sending buffer. The buffer is flushed only upon the request of the programmer. Generic communication functions provided by target systems are then used for sending the buffer to the remote PE. After receiving the message at the destination PE, the data is then stored in the OOPS receiving buffer. The remote PE then tries to unpack and restore the data from the system buffer back to its original relative position. Fig. 8.1 shows the concepts of OOPS communication.

For every data structure, four functions are required: `pack`, `unpack`, `send` and `receive`. Before sending a data object, the program must pack all the data elements with the OOPS packing function. Opposite to the sending procedure, the receiving PE receives the data from the communication channel first, and then unpacks it according to its packing sequence.

The advantage of writing four functions instead of two are:

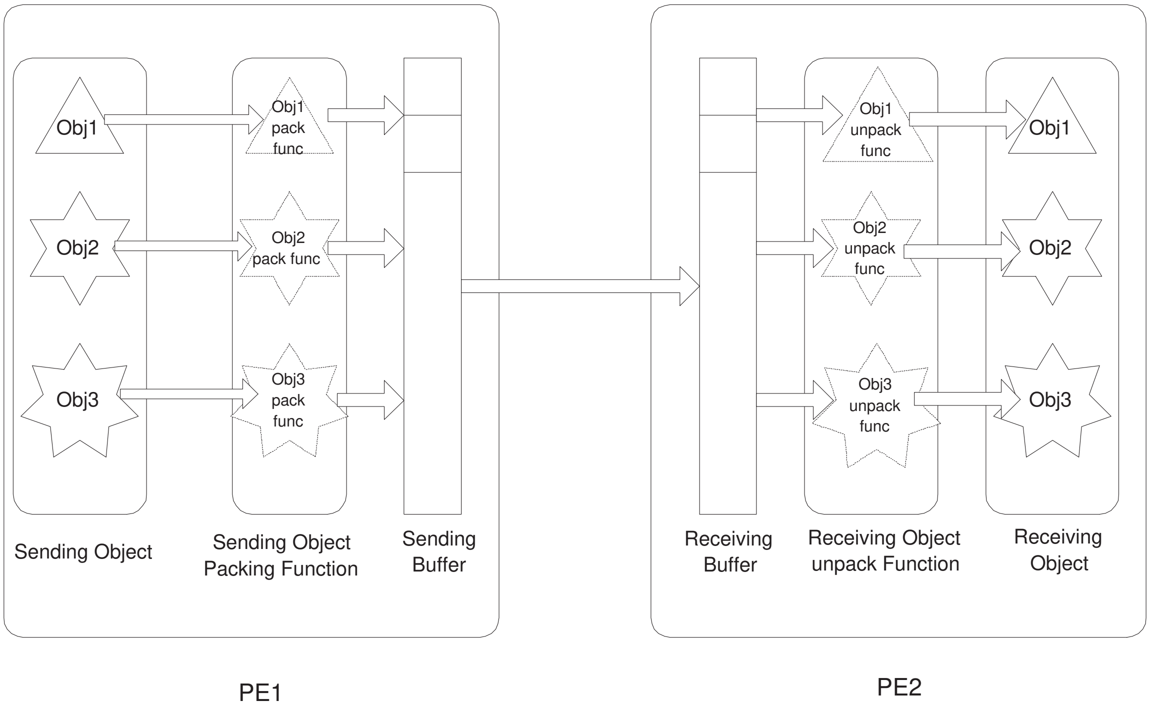

Figure 8.1: OOPS Program Architecture

1. Two extra functions, *pack* and *unpack*, provide us the opportunity to exploit inheritance feature of data objects while maintaining the minimum number of actual sending/receiving function calls. Besides, since the inheritance feature is utilized, writing the communication functions is greatly simplified.

2. Since the OOPS system takes care of endian conversions in different systems, a universal communication function is possible regardless of whether the working environment is homogeneous or heterogeneous.

3. The coding of these functions is relatively systematic. It is conceivable that a program generator can be developed.

OOPS supports five types of data: char, float, double, integer and short, which are the

```
SLAVE_SCHED()
while (!finish)
do /* User Phase */
      Select one of the following task and execute
      MEM_TASK_PROC( ) ;  /* process knowledge access tasks */
      MEM_COMM_PROC( ) ;  /* check for incoming knowledge tasks */
      TASK_QUEUE_PROC( ) ;  /* perform the tasks in the TQ */
      KERNEL_MESG_QUEUE_PROC( ) ;  /* picking up a kmesg and execute */
      KERNEL_MESG_RETRIEVAL( ) ;  /* check for the incoming kmesgs */
      DUP_QUERY_RETRIEVAL( ) ;  /* check for the incoming duplicate check task */
      DUP_QUERY_PROC( ) ;  /* perform the duplicate query tasks */
      DUP_REPLY_RETRIEVAL( ) ;  /* check for the duplicate query reply */
      DUP_ANS_REG_RETRIEVAL( ) ;  /* check for the duplicate ans register msgs */
      DUP_ANS_REG_EXEC() ;  /* register the duplicate answer into hash table */

     if nothing to do
        then Enters the synchronous point

      /* System Phase */
     if at synchronous point
        then
              LOAD_BALANCE( ) ;
              finish ← GLOBAL_TERMINATION( ) ;
```

Figure 8.2: Slave Scheduler

most commonly used data types in the computation.

8.2 Slave Scheduler

The slave scheduler behaves like a switching module during slave PE execution. There are two phases in the scheduler, the *user phase* and the *system phase*. In the user phase, the scheduler selects one module to execute every cycle. The system phase is invoked periodically for load balancing and termination checking. Fig. 8.2 shows the algorithm of the slave scheduler.

8.2.1 User Phase

In the user phase, real query tasks and other supporting tasks are executed. The modules in the user phase can be classified into two groups: *task modules* and *listener modules*. The task modules are the *knowledge task execution module*, the *task execution modules*; which can be further divided into the *task execution module*, the *kernel message execution module*, the *duplicate query execution module*, and the *duplicate query registration module*.

The function of the listener modules is to act like the daemon programs in the TERRESA system. When executing the listener modules, they inspect the communication channels to see if there is any specific incoming message. If there is, the daemon modules try to pick it up and put it into task queue. The daemon modules in the TERRESA system are the *knowledge receiving module*, the *kernel message receiving module*, the *duplication query receiving module*, and the *duplication answer receiving module*.

Currently, all user phase modules are scheduled in round-robin fashion. Modules are executed sequentially, once per cycle.

8.2.2 System Phase

The system phase in the TERRESA scheduler is a synchronous phase, as every slave PE must enter the system phase before any PE can go further. During the system phase, two modules are executed in consecutively: the *load balancing module*, which was mentioned in Section 5.4, and the *termination checking module*, which checks if all the tasks on all the PEs have be solved so that the slave session can be terminated.

Entering the system phase can be either *periodical* or *active*. Periodical entry is when the system phase is entered occasionally during the scheduling routine. If the scheduler enters the system phase only when nothing can be executed in the user phase, the condition is called *active entry*.

Chapter 9

Performance and Experimental Results

Performance of TERRESA, like many parallel applications, is influenced by many factors. Internally, the performance can be influenced by idle time as well as the overhead of parallel execution. Idle time in a parallel system is usually the result of the following circumstances: (1) load imbalance among PEs, i.e. the time between finishing the last task in the local PE and the beginning of the load balancing procedure; (2) blocking send/receive calls (like `MPI_Send` and `MPI_Recv`) and (3) other synchronization behavior among different PEs, such as the collective communication, like `MPI_Barrier` and `MPI_Scatter`, etc. or the synchronization demanded by the designer.

System overhead is incurred by several actions, including system bookkeeping, communication, and scheduling. Because the TERRESA system is a complex program with various modules, it is very difficult to isolate and analyze the overhead incurred by a single factor across modules. Rather, in this dissertation, we try to analyze the overhead incurred by each module.

Externally, the performance is influenced by various issues. From the hardware point of view, it can be influenced by communication bandwidth, the speed of the PE, the number of PEs participating in the computation, the size of cache and primary memory of the machines, and the topologies of the cluster if the computer is connected by network, etc. Moreover, under a timesharing environment like the network of workstations, the load on the system and network congestion are also decisive elements. Unfortunately, almost all the factors mentioned here cannot be controlled by either the programmer or the end users.

As for the TERRESA itself, performance can be influenced by the knowledge network topology and connectivity, the number of nodes in the network, the order of the assertions, the format of the query and the order of the query, etc.

This chapter is organized as follows: an introduction to the test examples adopted in the TERRESA experiments is given in Section 9.1. There are three kinds of test examples: the single inheritance classification tree, the multiple inheritance classification tree, and the relational database. Section 9.2 tries to measure the overhead incurred by different modules. Section 9.3 describes several factors which might influence the performance of the TERRESA system. In Section 9.4, experiments are conducted and the performance evaluation is presented.

The machine used in the experiment, `pegasus.cs.buffalo.edu`, is a SUN server Enterprise 4000. It contains 1GB of main memory and 8 168 MHz UltraSPARC CPUs. Detail specification can be seen on SUN Corp. web documentation [64]. Although it might be less efficient to use message passing scheme on the shared memory machines, yet because for the portability reason, the message passing scheme is adopted as the underneath supporting mechanism. Also, although there is no direct evidence showing how the SUN server/MPI performs, by using the shared memory as the communication device, the communication bandwidth of symmetric multiprocessors machine should be about the same as other parallel machines. According to [26], the communication bandwidth of the SGI Power Challenge/MPI is about 45 to 55 MB/sec., which is comparable to other parallel machines like Intel Paragon (40 $\sim$ 65 MB/sec.), Cray T3D (70 $\sim$ 100 MB/sec.), and out performs IBM SP2 (15 $\sim$ 25 MB/sec.). The performance is definitely better than the networked workstations (0.8 $\sim$ 1.0 MB/sec.). There is no reason to believe when using the same communication device, the performance of symmetric shared memory SUN Sparc server will be far less than desirable.

9.1 Test Examples

Three test examples are used in the TERRESA experiments: the single inheritance classification tree, the multiple inheritance classification tree, and the relational database.

9.1.1 Single Inheritance Classification Tree

Property inheritance question probably is the most basic problem in the knowledge representation research. It has been a part of semantic network since the beginning of the semantic network research [50, 51]. Usually the inheritance relationships is represented in a hierarchical tree, or a classification tree. In the classification tree, every subclass entity exhibits some resemblance to its ancestor or super class entity. The single inheritance classification tree is the basic representation form in the classification. Fig. 9.1 is an animal classification tree. In the figure, base nodes, indicated by oval boxes, represent conceptual entities, either an animal or a class of animals. The assertions, indicated by the molecular nodes, are represented by circles. Each node in the single inheritance classification tree has only one ancestor. One node, above all other nodes, is the common ancestor of the entire population.

There are two types of queries can be asked: *top-down query* and *bottom-up query*. Top-down query asks the questions like "Which entities inherit these properties?" The inference procedure follows the *(supclass- subclass)* arcs down and seeks the possible entries. On the contrary, the bottom-up query asks the question like "What kind of properties does this entity have?" The inference procedure follows the *(subclass- supclass)* arcs up and seeks the possible solutions.

The *binary single inheritance classification tree* is a classification tree whose non-leaf base nodes have two distinct descendents. In our experiments, a tree totally 8192 assertions (16381 nodes) are generated by program as one of the TERRESA test examples. The appearance sequence of the assertions are organized according to the depth-first traversal strategy.

9.1.2 Multiple Inheritance Classification Tree

Unfortunately the single inheritance classification tree does not have the enough expressing power to represent various forms of the hierarchical knowledge. In the real world, usually the knowledge cannot be encoded in such a strict taxonomic hierarchy format. Many

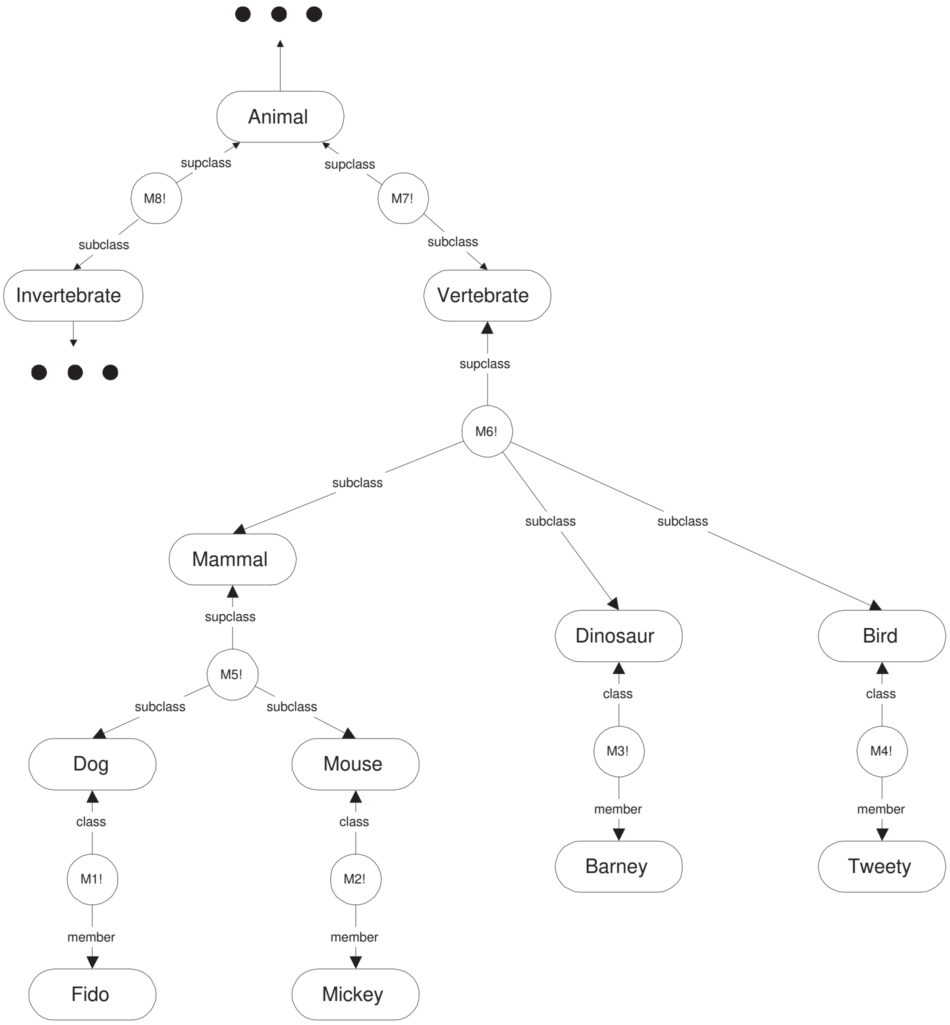

Figure 9.1: Single Inheritance Classification Tree

knowledge entities may belongs to two non-mutually exclusive supersets at the same time. The tangled semantic network forms a "web of knowledge". Fig. 9.2 illustrates another example of an animal classification tree. For simplification, all molecular nodes in the figure are omitted. The classification not only classify animals according to their species, but also the diet habit and whether the animal can be a pet or not. As seen in the figure, in this classification, the nodes *cat*, *dog*, *chameleon*, *rabbit*, *feline*, and *canine* all have more than one parent.

A multiple inheritance classification tree is a classification tree where the non-root nodes in the tree might have two or more parents. In our experiment, a network is built by program by joining two binary trees together as the experiment target of the multiple inheritance classification tree. The tree, where all the molecular nodes are omitted for simplicity, is shown in Fig. 9.3. In this network, each non-root node has two parents and each non-leaf node has four children. The network we tested has 8192 assertions with 12278 nodes. The appearance of the assertions are according to the depth first traversal strategy for each tree and left tree is listed prior the right one.

9.1.3 Relational Database

A relation database is another interesting form of knowledge representation and one of the three test examples used in the experiments. From the user point of view, the database can be seen as a collection of tables. Each row of a table, or a record, is a specific entity representing a concept. Each column of a table, or the field, stores the value of the attribute of that row. In the semantic representation, each attribute value in the data entries are repre-

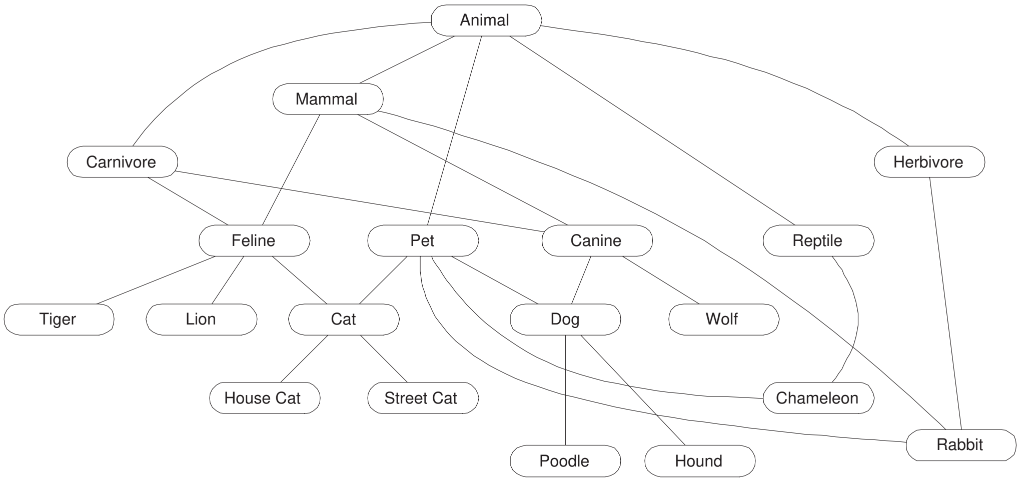

Figure 9.2: Multiple Inheritance Classification Tree

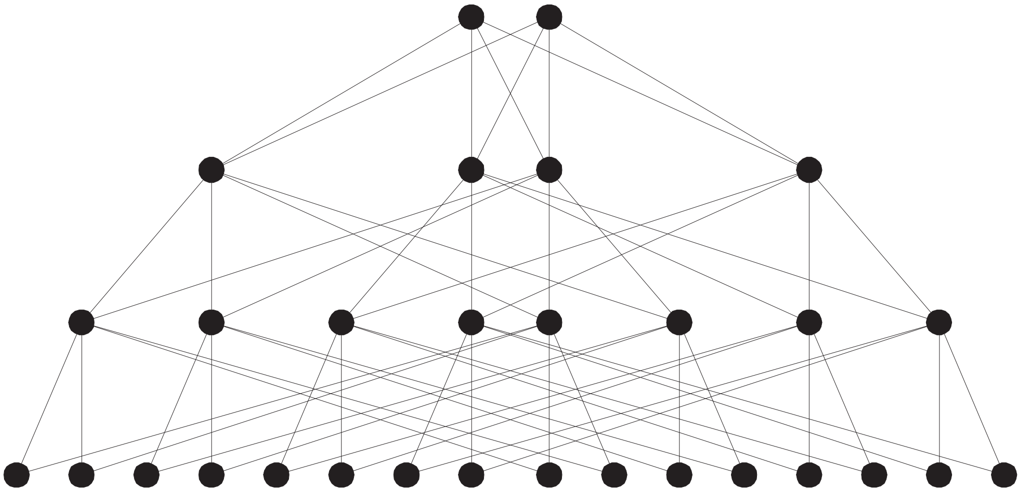

Figure 9.3: Test Example of Multiple Inheritance Classification Tree

sented by base nodes. The tuple, which represents the relationship between data entries in the table, is denoted by molecular nodes, and the attributes in the database are represented by the edges connecting the nodes.

A typical example of database, excerpted from page 124 of [13] can be seen in Table 9.1. Take an example, say the first row of the first table. A molecular node, say M1, represents the row. `S1`, `Smith`, `20`, and `London` are represented by some base nodes. The molecular node and the base nodes above are connected by the edges `SNUM`, `SNAME`, `STATUS`, and `CITY`, respectively.

Two databases, one 5120 assertions (9432 nodes), the other 16384 assertions (25238 nodes) are randomly generated as our test examples.

9.2 Overheads

In this section, overhead is analyzed by the modules. Three major modules incur overhead when TERRESA is executed: the *Shared Knowledge Module*, the *Load Balance Module*, and the optional *Duplicate Checking Module*.

9.2.1 Shared Knowledge Module Overhead

The TERRESA tasks need to access the nodes to finish the inference. Therefore the accessibility of nodes is crucial for the performance of the system. The nodes can be either local - private or cache, or remote. When the nodes are not in local, a knowledge fault occurs.

SNUM	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

PNUM	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

JNUM	JNAME	CITY
J1	Sorter	Paris
J2	Punch	Rome
J3	Reader	Athens
J4	Console	Athens
J5	Collator	London
J6	Terminal	Oslo
J7	Tape	London

SNUM	PNUM	JNUM	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	100
S5	P3	J4	200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

Table 9.1: Database Example

Shared knowledge module overhead is resulted from knowledge faults. Remote knowledge access overhead includes packing, sending, receiving, and unpacking knowledge tasks, knowledge task execution, putting them into the proper cache location, and bookkeeping information update (like access record, address lookup, etc). An remote knowledge access consists two knowledge tasks traffic, request and reply.

The experiment uses the 8K single inheritance classification tree as the test bed with a query of path length 16. The experiment is conducted as follows: first the load balance module and duplicate checking module are turned off; then the same query is executed twice in the same session; when the query is executed the first time, some of the nodes are not local causing knowledge faults to occur; afterward when the query is asked at the second round, because all the nodes that were in remote PEs are now in local cache, the time difference between the two queries is considered overhead caused by remote knowledge access.

The experiment result are listed in Table 9.2 and Fig. 9.4. The overhead varies from approximate 8 msec for 2 PEs to 60 msec for 8 PEs. From the figure we know that the overhead per knowledge task is more or less proportional to the number of PEs in the system due to resource contention.

9.2.2 Load Balancing Overhead

Load balance overhead is caused by performing load balance when the mechanism is on during execution. The load balance overhead includes exchanging load information, shuttling ptasks between corresponding PEs, adding the migrated ptasks into local ptask pools,

No. of PEs	2	4	6	8
Average Knowledge Fault Overhead (Sec.)	0.0078	0.0316	0.0490	0.0601

Table 9.2: Average Remote Knowledge Access Overhead

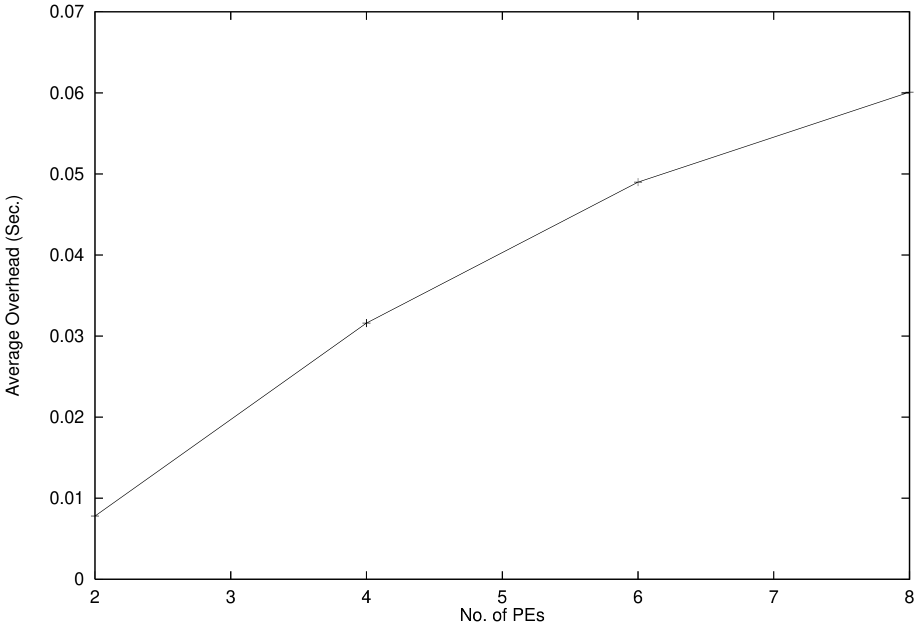

Figure 9.4: Average Remote Knowledge Access Overhead

and bookkeeping information updating.

Unlike remote knowledge access which involving asynchronous information request and reply, load balance is a synchronous event invoked periodically during execution, therefore the load balance overhead can be measured by the entering and exiting time of the load balance procedure. The experiment also uses the 8K single inheritance classification tree as the test example. The query tested in the experiment is in the form:

$$\texttt{(find (subclass- supclass} \ldots \texttt{ subclass- supclass) B}_i\texttt{)}$$

with the path length 10 and one starting node. This query guarantees that PEs will move some of its load to its neighbor. The same query is asked 10 times and average is calculated. Table 9.3 and Fig. 9.5 show the number of PEs versus average load balance overhead. Similar to previous experiment, as the number of PEs increases, so does the average load balance time. Average overhead ranges from 1 msec of 2 PEs to almost 170 msec of 8 PEs. The primary reason of time increase is due to more tasks movement and synchronous session in the load balance module.

9.2.3 Duplicate Checking Overhead

Table 9.4 and Fig. 9.6 illustrates the average overhead per duplicate checking task. The duplicate checking overhead is caused by generating the duplicate checking tasks, sending and receiving the task for checking, generating reply messages, registration of the answers, etc.

The experiment is conducted on the 8K single inheritance classification tree. The procedure of the experiment is as follows: first we turned the duplicate checking off to get the

No. of PEs	2	4	6	8
Average Load Balance Overhead (Sec.)	0.0014	0.0328	0.0925	0.168

Table 9.3: Average Load Balance Overhead

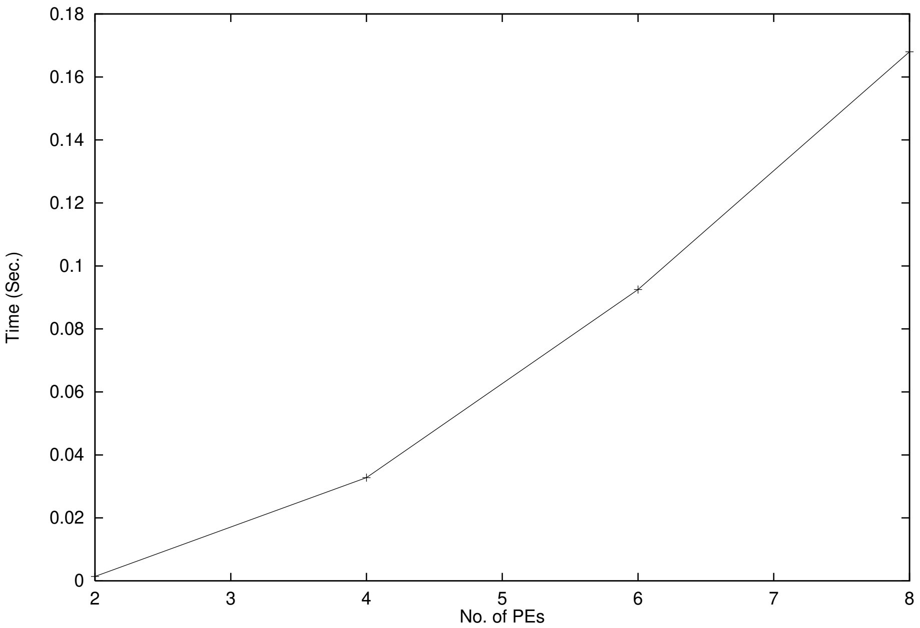

Figure 9.5: Average Load Balance Overhead

No. of PEs	2	4	6	8
Average Duplicate Checking Overhead (Sec.)	0.00422	0.00736	0.03526	0.06498

Table 9.4: Average Duplicate Checking Overhead

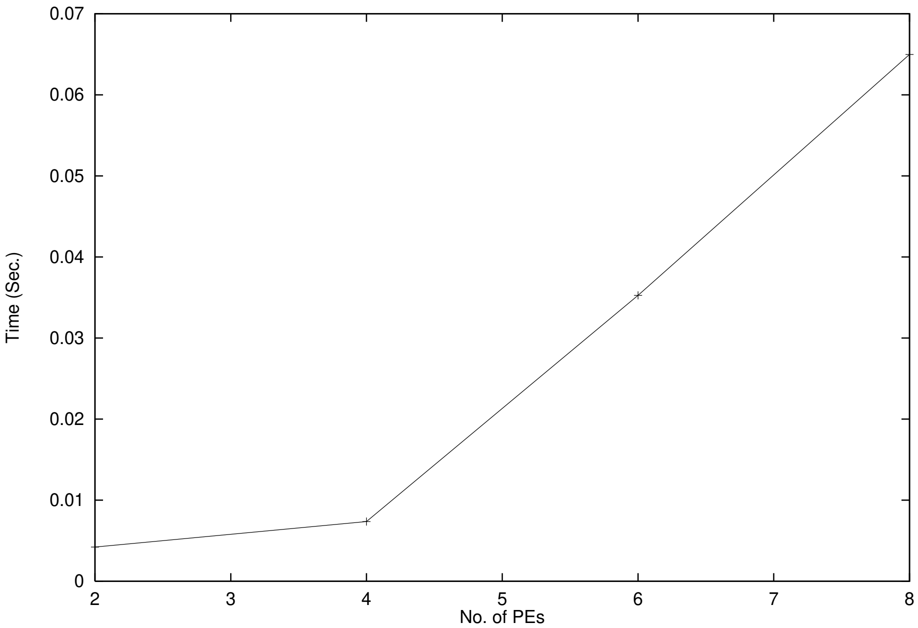

Figure 9.6: Average Duplicate Checking Overhead

time needed for any other services. Then duplicate checking is turned on for another session of time estimation. Because the binary tree is used, there will be no duplication in the entire computation session, the difference between these two experiments is the overhead caused by the duplicate checking tasks. This experiment is repeated 10 times and average is calculated.

As illustrated, when the number of PEs is small (less than 4), the overhead is almost the same (about 5 - 10 msec. per checking). When the number of PE increases, so is the overhead. In our experiment, the overhead for 8 PEs is almost 65 msec.

9.3 Factors that Influence the Performance

TERRESA performance is influenced by many factors, two major factors are discussed in this section.

9.3.1 Assertion Sequence

Due to the two-level knowledge access and the cyclic block graph partition and distribution in TERRESA, the order of assertions in the input sequence has a tremendous influence on the performance of the system. The assertion sequence actually determines where the nodes are placed in the PEs. If all the adjacent nodes are disjoint, the penalties incurred by the knowledge access will overshadow the effect of parallelism. Better performance can be achieved by simply tweaking the order of the assertions.

Take example of a single-inheritance binary tree. The serial number of the molecular nodes represents the order of the assertions entering the system. Suppose the system has 4 PEs and the block size is 16. If the assertions are input in a top-down, left to right fashion, the partition of the nodes are organized in the way shown in Fig. 9.7. As we can observe, although the nodes in the PE 1 are all connected, most nodes in the other three PEs are disjoint. The pattern of distribution is highly unfavorable to the cache access because of its lack of locality.

Instead, if we organize the assertion sequence according to in-order tree traversal, the partition of nodes can be set up in the way shown in Fig 9.8. From the figure, almost all the nodes are connected, thus improving performance. One of the experiments shows that by simply reorganizing the assertions, the same query can be sped up from 87.986 seconds to 4.987 seconds, almost 18 times faster than the poor sequencing example.

9.3.2 Queries

Because of the overhead incurred by the parallel computation, not all problems benefit from the parallelism of TERRESA. Usually, parallel computing focuses on the problems with large test data sets or long computation duration. Shorter problems generally will not gain any performance advantages. Therefore in our experiments with TERRESA, we focus on the queries with either extensive path length or a huge number of starting nodes.

Also, because of the shared knowledge module design in TERRESA, the performance will be different for the same query if they appears in different order. The effect of using the cache is crucial to the performance in the system.

9.4 Experiments

This section examines the result of the experiments conducted on the single inheritance classification tree, multiple inheritance classification tree, and the relational database.

9.4.1 Single Inheritance Classification Tree

Remote Access v.s. Local Access

Fig. 9.9 shows the first experiment on the 8K assertions single inheritance classification tree. The query is in the format of

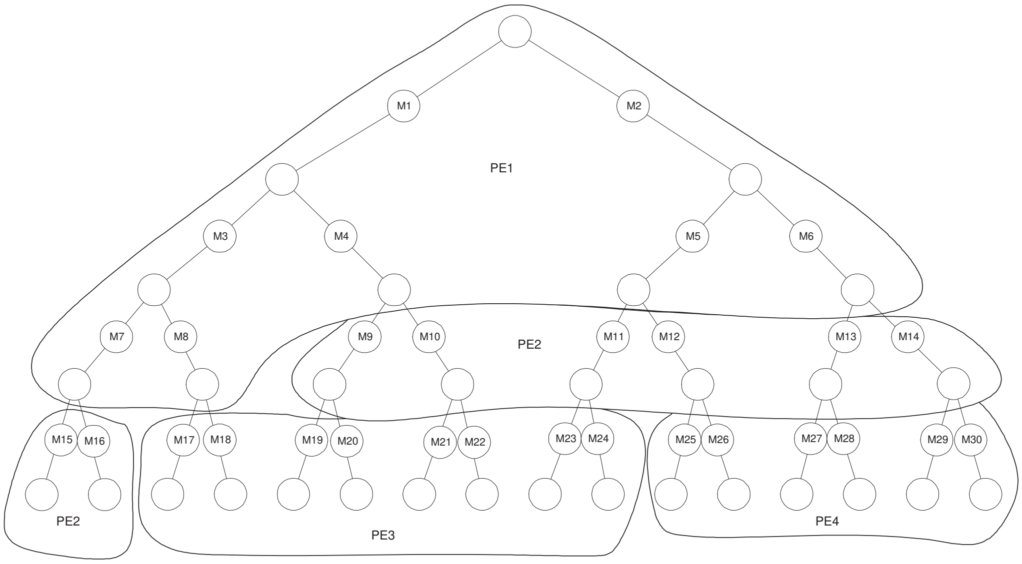

Figure 9.7: Poor Assertions Ordering

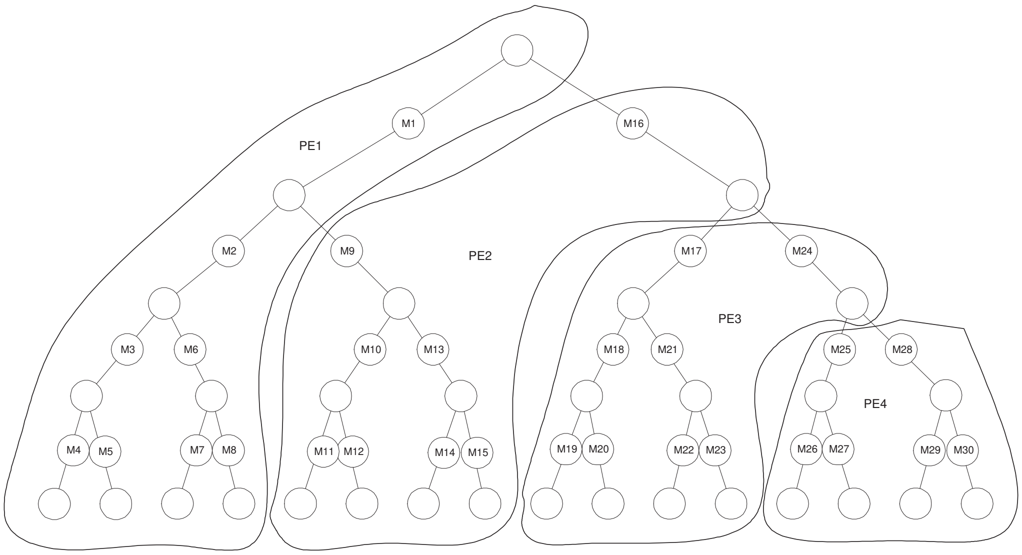

Figure 9.8: Better Assertions Ordering

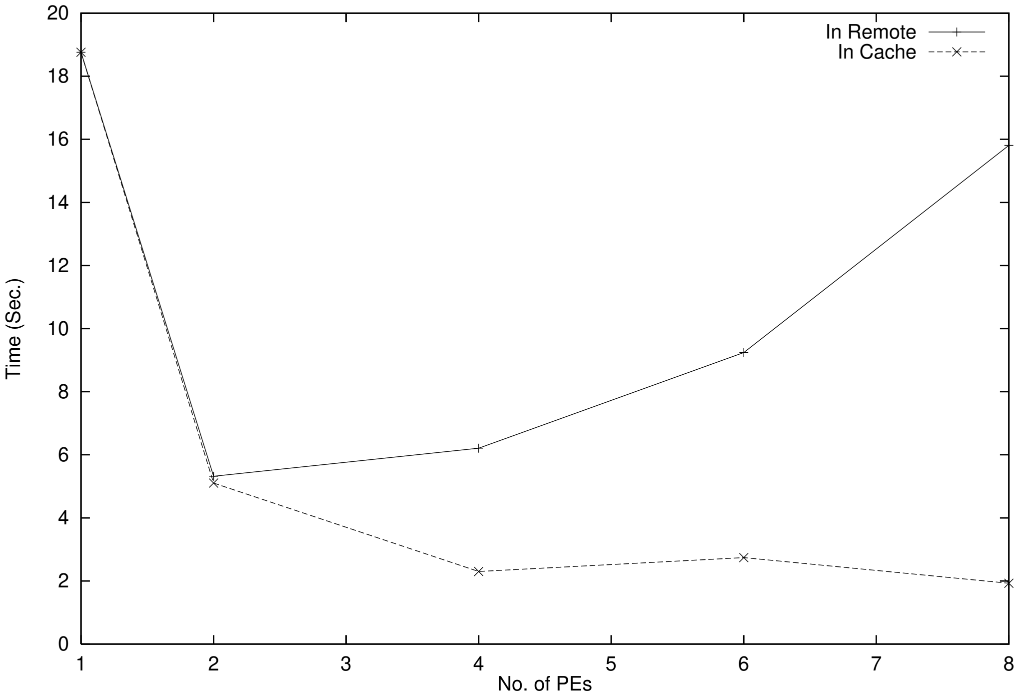

Figure 9.9: Single-Inheritance Classification Tree: Remote Access v.s. Local Access

$$(\texttt{find}\,(\texttt{subclass-}\ \texttt{supclass}\ \ldots\ \texttt{subclass-}\ \texttt{supclass})\,(B_{i_1}\ \ldots\ B_{i_7}\ B_{i_8}))$$

The path length is 16 in the test query. The same query was asked twice to examine the effect of cache knowledge. Intuition tells us that in an SMP parallel system, as the number of PEs increase, so does the competition for computer resources like shared memory and PEs. Also, as the number of PEs increase, the possibility of accessing a node that is not in the local shared knowledge module also increases because of the distribution of the same amount of nodes into a larger of PE pool.

As illustrated, if a query is issued at the first time, some of the nodes are in remote, therefore knowledge access faults occur and knowledge tasks must be issued to transfer them from remote to local PE, and the finish time is inevitably longer than the query on the second time. Even though, the experiments show that we can still benefit from parallel computation and get moderate performance (speedup ranges from 3.52 of 2 PEs to 1.19 of 8 PEs). However, the performance degrades as the number of PEs increases.

But once the nodes are in local (inside the cache), the experiment indicates that a nice performance increase can be achieved. Table 9.5 shows the speedup for the same query the second time. Speedup ranges from 3.67 for 2 PEs to 9.75 for 8PEs.

A special phenomena worth mentioning is the superlinear behavior in the experiment, which will be seen from time to time in the subsequence experiments. When examining the program by the profiler `gprof`, the superlinear speedup phenomena is caused by the dramatic time decrease in the task execution module. Further examination indicates that this phenomena is caused by the diminishing number of node location lookup in the shared knowledge module, in both private knowledge and cache knowledge. Table 9.6 shows the number of comparisons made and time percentage for locating a node in the shared knowledge module during execution. The possible reason for fewer comparisons is because of the imbalance in the private knowledge address lookup tree. As presented previously in Chapter 6, binary search trees are used in the shared knowledge module for address lookup. When faults occur and requests are made, the random appearances of the nodes from remote might generate a more "scrambled" tree index and thus a more balanced binary tree.

Average Performance

Another experiment is conducted to test the average performance on the single inheritance classification tree. Total 25 queries with different query length and number of initial nodes are generated randomly. The result of the experiments are listed in Table 9.7 and Table 9.8.

No. of PEs	2	4	6	8
First query, some of the nodes are in remote	3.52	3.02	2.03	1.19
Second query, all the nodes are local	3.67	8.16	6.84	9.75

Table 9.5: Single-Inheritance Classification Tree: PEs v.s. Speedup

No. of PEs	1	2	4	6	8
No. of Comparison	50,139,072	10,005,217	10,634,716	5,156,929	4,315,222
Time (%) (lookup / `find`)	62.0%	31.3%	35.8%	33.1%	27.1%

Table 9.6: Single-Inheritance Classification Tree: Number of Comparison in address lookup of the Shared-Knowledge Module

In the tables, the *cache hit ratio* is calculated according to the following formula:

$$\text{Hit Ratio} = \frac{\text{Cache Hit}}{\text{Cache Hit + Cache Miss}}$$

where the number of cache miss is equivalent to the number of knowledge faults. The table indicates that usually the cache performs quite well, the hit ratio is kept above 97% in all cases. Yet TERRESA computation is sensitive to the cache hit. From observation, when the hit ratio is less than 98%, undesirable performance could happen.

Fig. 9.10 shows the total number of faults for each query and Fig. 9.11 shows the speedup for each query. Due to competition and overhead, whenever there is a knowledge fault in the computing process, the performance degrades. However, when the cache is filled up with remote information during the computation procedure and queries no longer seek knowledge from other PEs, the speedup, especially for higher number of PEs, improves dramatically. Table 9.9 shows the average speedup for the queries with no faults and the average speedup with faults. The table indicates that the speedup without faults are generally better than the ones with faults. Even so, we can still get speedup about 2.1 to 5.8 with faults.

9.4.2 Multiple Inheritance Classification Tree

Remote Access v.s. Local Access

A similar experiment is conducted on the multiple inheritance classification tree. Because the network has higher connectivity (every non-leaf node has 4 children and every non-root node has 2 parents), the time needed to compute the same query will be more expensive than the one in the single inheritance classification tree. In the experiment, instead of using a combination of 16 unit paths, 12 are used as the initial test. Fig. 9.12 shows the result, which is very similar to the outcome of the single inheritance classification tree except that it takes more time to finish even for a shorter query.

Average Performance

The average performance tests are conducted in a similar way to the experiments in the single-inheritance classification tree. Total 25 randomly generated test cases are fed into the system. The result is listed in Table 9.10 and Table 9.11, respectively. Fig. 9.13 and Fig. 9.14 shows the total number of faults and the speedup for each query, respectively.

Query	1 PE Time	2 PEs Time	2 PEs Faults	2 PEs Cache Hit Ratio	2 PEs Speedup	4 PEs Time	4 PEs Faults	4 PEs Cache Hit Ratio	4 PEs Speedup
1	32.229	11.745	61	0.991	2.744	6.796	141	0.987	4.742
2	51.803	28.099	28	0.997	1.844	8.692	132	0.990	5.960
3	10.313	7.151	50	0.985	1.442	11.808	112	0.977	0.873
4	37.332	13.354	0	1.000	2.796	11.957	113	0.988	3.122
5	18.503	6.854	0	1.000	2.700	1.913	0	1.000	9.672
6	49.835	18.630	0	1.000	2.675	6.597	8	0.999	7.554
7	13.508	5.657	0	1.000	2.388	1.309	0	1.000	10.319
8	50.263	20.463	8	0.999	2.456	9.935	14	0.999	5.059
9	9.515	4.053	0	1.000	2.348	1.504	5	0.999	6.326
10	18.462	6.312	0	1.000	2.925	3.217	0	1.000	5.739
11	18.824	9.226	0	1.000	2.040	5.130	10	0.999	3.669
12	45.980	22.847	0	1.000	2.013	10.295	0	1.000	4.466
13	26.543	11.303	0	1.000	2.348	2.970	0	1.000	8.937
14	19.488	6.754	0	1.000	2.885	2.816	8	0.999	6.920
15	37.219	13.907	0	1.000	2.676	4.413	0	1.000	8.434
16	13.023	5.203	0	1.000	2.503	1.550	0	1.000	8.402
17	31.695	11.011	0	1.000	2.878	2.823	0	1.000	11.227
18	22.447	7.085	0	1.000	3.168	3.208	0	1.000	6.997
19	28.432	11.463	0	1.000	2.480	3.222	0	1.000	8.824
20	10.636	3.907	0	1.000	2.722	2.642	0	1.000	4.026
21	13.695	4.832	0	1.000	2.834	1.387	0	1.000	9.874
22	19.432	6.924	0	1.000	2.806	2.550	0	1.000	7.620
23	37.607	14.090	0	1.000	2.669	5.186	0	1.000	7.252
24	25.331	13.213	0	1.000	1.917	3.915	0	1.000	6.470
25	58.407	21.582	0	1.000	2.706	7.276	0	1.000	8.027
Avg					2.519				6.821

Table 9.7: Single-Inheritance Classification Tree (1 PE to 4 PEs): Time and Speedup

Query	6 PEs				8 PEs			
	Time	Faults	Cache Hit Ratio	Speedup	Time	Faults	Cache Hit Ratio	Speedup
1	7.309	194	0.983	4.409	13.359	288	0.977	2.413
2	7.825	127	0.991	6.620	15.617	255	0.984	3.317
3	16.229	136	0.975	0.635	40.117	161	0.972	0.257
4	18.229	269	0.974	2.048	23.678	262	0.976	1.577
5	2.082	1	1.000	8.887	6.987	77	0.991	2.648
6	7.691	53	0.996	6.480	6.414	0	1.000	7.770
7	1.133	4	0.999	11.922	0.828	0	1.000	16.314
8	11.015	13	0.999	4.563	9.396	23	0.998	5.349
9	1.043	3	0.999	9.123	22.215	106	0.975	0.428
10	1.744	0	1.000	10.586	1.825	16	0.998	10.116
11	3.796	12	0.999	4.959	9.597	31	0.997	1.961
12	6.425	0	1.000	7.156	5.024	0	1.000	9.152
13	2.470	0	1.000	10.746	3.259	0	1.000	8.145
14	1.715	3	1.000	11.363	2.365	6	0.999	8.240
15	3.670	11	0.999	10.141	2.237	0	1.000	16.638
16	0.790	0	1.000	16.485	0.853	0	1.000	15.267
17	2.927	0	1.000	10.828	1.766	0	1.000	17.947
18	2.270	0	1.000	9.889	2.276	0	1.000	9.862
19	2.204	0	1.000	12.900	1.696	0	1.000	16.764
20	3.897	34	0.994	2.729	1.077	0	1.000	9.876
21	2.707	13	0.998	5.059	1.287	8	0.999	10.641
22	1.204	0	1.000	16.140	1.112	0	1.000	17.475
23	3.271	0	1.000	11.497	7.568	36	0.997	4.969
24	3.673	8	0.999	6.897	2.079	0	1.000	12.184
25	3.928	0	1.000	14.869	4.322	0	1.000	13.514
Avg				8.677				8.913

Table 9.8: Single-Inheritance Classification Tree (6PEs and 8 PEs): Time and Speedup

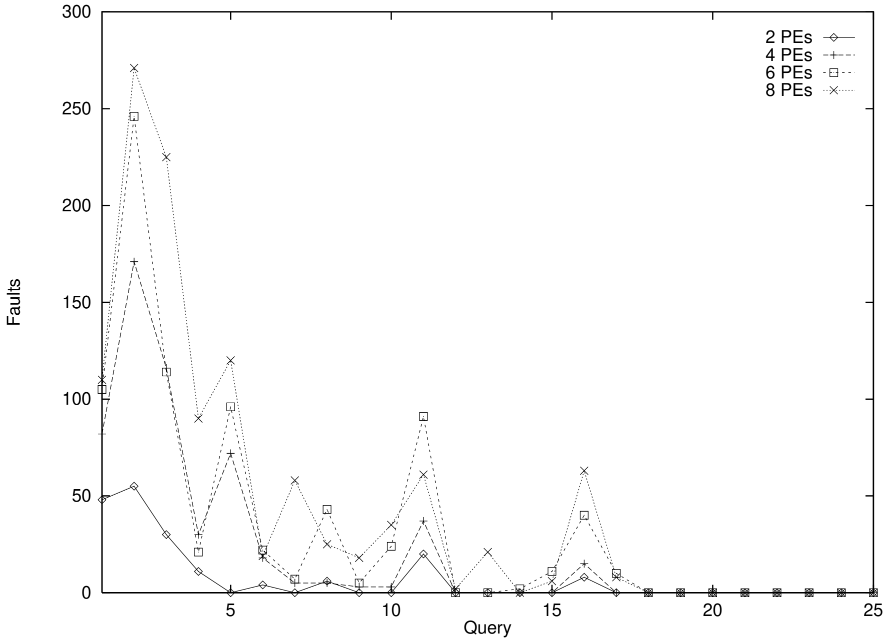

Figure 9.10: Single-Inheritance Classification Tree: Total Number of Knowledge Faults

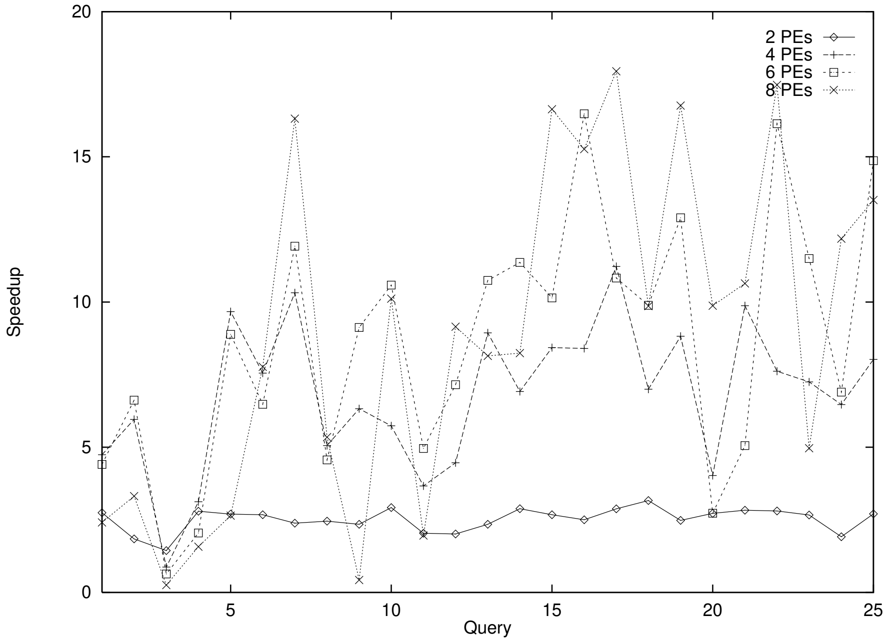

Figure 9.11: Single Inheritance Classification Tree: Speedup

No. of PEs	2 PEs	4 PEs	6 PEs	8 PEs
Speedup with Faults	2.12	4.91	5.81	4.32
Speedup without Faults	2.59	7.42	11.78	13.14

Table 9.9: Single-Inheritance Classification Tree: Average Speedup Comparison

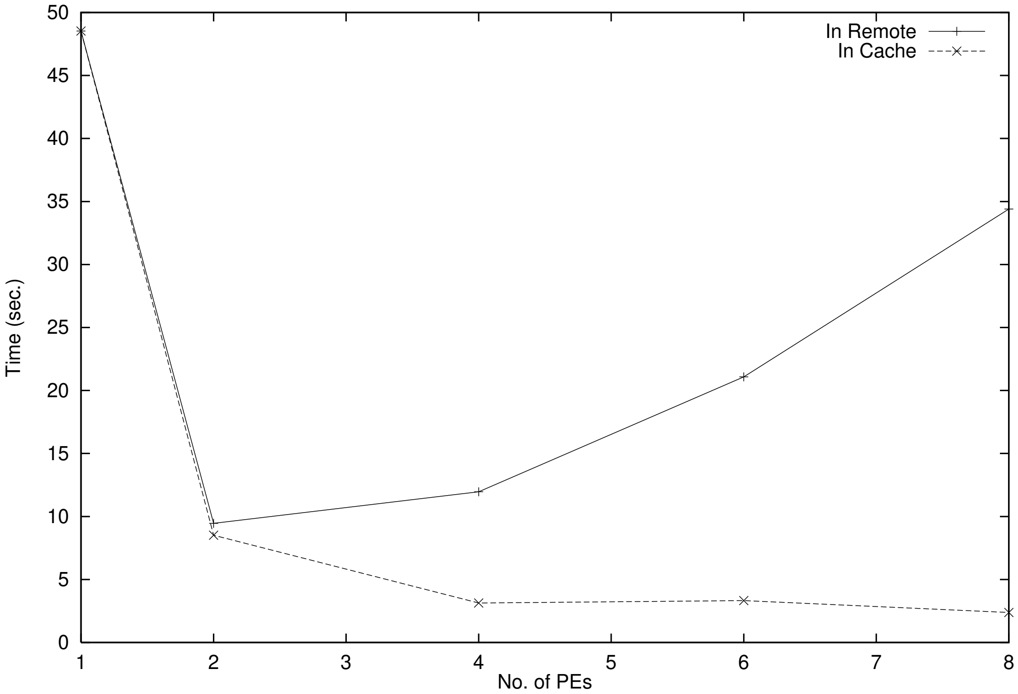

Figure 9.12: Multiple Inheritance Classification Tree: Private v.s. Cache

Query	1 PE Time	2 PEs				4 PEs			
		Time	Faults	Cache Hit Ratio	Speedup	Time	Faults	Cache Hit Ratio	Speedup
1	39.399	9.490	506	0.912	4.152	6.836	757	0.907	5.763
2	11.847	3.093	73	0.973	3.830	3.434	306	0.932	3.450
3	12.351	3.286	118	0.957	3.759	4.065	335	0.918	3.038
4	48.071	9.885	19	0.997	4.863	5.924	356	0.968	8.115
5	39.935	10.226	304	0.947	3.905	3.717	44	0.995	10.744
6	15.264	4.115	91	0.975	3.709	3.569	107	0.979	4.277
7	7.284	1.612	0	1.000	4.519	1.708	18	0.994	4.265
8	11.561	2.638	0	1.000	4.382	1.060	23	0.995	10.907
9	20.410	4.555	0	1.000	4.481	1.708	0	1.000	11.950
10	7.792	1.741	0	1.000	4.476	0.600	0	1.000	12.987
11	17.523	3.748	0	1.000	4.675	1.420	0	1.000	12.340
12	48.206	9.593	0	1.000	5.025	3.868	16	0.999	12.463
13	7.349	1.466	0	1.000	5.013	0.435	0	1.000	16.894
14	29.925	5.837	0	1.000	5.127	2.235	0	1.000	13.389
15	7.749	1.643	0	1.000	4.716	0.544	0	1.000	14.244
16	23.261	4.786	0	1.000	4.860	1.598	0	1.000	14.556
17	36.036	7.474	0	1.000	4.822	4.847	160	0.982	7.435
18	9.428	2.110	0	1.000	4.468	0.938	14	0.997	10.051
19	49.000	10.399	0	1.000	4.712	3.054	0	1.000	16.045
20	50.564	10.775	0	1.000	4.693	3.005	0	1.000	16.827
21	12.510	3.800	0	1.000	3.292	0.880	0	1.000	14.216
22	14.743	3.387	0	1.000	4.353	0.721	6	0.999	20.448
23	23.668	4.866	0	1.000	4.864	1.152	28	0.994	20.545
24	33.138	6.916	0	1.000	4.791	1.634	44	0.992	20.280
25	14.370	4.814	0	1.000	2.985	0.904	0	1.000	15.896
Avg					4.419				12.045

Table 9.10: Multiple Inheritance Classification Tree (1 PE to 4 PEs): Time and Speedup

Query	6 PEs				8 PEs			
	Time	Faults	Cache Hit Ratio	Speedup	Time	Faults	Cache Hit Ratio	Speedup
1	14.159	1025	0.889	2.783	52.485	1465	0.863	0.751
2	8.947	653	0.863	1.324	37.798	751	0.836	0.313
3	5.784	481	0.904	2.135	20.891	401	0.916	0.591
4	18.296	754	0.933	2.627	36.955	633	0.948	1.301
5	8.472	246	0.975	4.714	44.096	521	0.953	0.906
6	5.706	261	0.956	2.675	21.416	596	0.905	0.713
7	0.976	45	0.987	7.463	3.298	63	0.982	2.209
8	4.946	73	0.985	2.337	5.129	78	0.985	2.254
9	4.036	113	0.985	5.057	5.954	109	0.986	3.428
10	0.827	8	0.998	9.422	1.929	25	0.994	4.039
11	2.994	92	0.986	5.853	3.492	40	0.994	5.018
12	5.357	44	0.996	8.999	16.314	283	0.977	2.955
13	0.788	32	0.990	9.326	4.606	17	0.995	1.596
14	1.751	0	1.000	17.090	4.497	47	0.995	6.654
15	1.340	25	0.994	5.783	4.079	45	0.989	1.900
16	2.557	24	0.997	9.097	8.290	100	0.989	2.806
17	2.178	12	0.999	16.545	11.398	232	0.977	3.162
18	1.057	0	1.000	8.920	3.085	33	0.993	3.056
19	3.605	0	1.000	13.592	1.473	0	1.000	33.265
20	3.646	40	0.996	13.868	2.106	32	0.997	24.009
21	0.526	0	1.000	23.783	0.811	0	1.000	15.425
22	0.466	0	1.000	31.637	3.641	39	0.992	4.049
23	1.401	10	0.998	16.894	11.150	148	0.977	2.123
24	3.161	172	0.976	10.483	6.804	44	0.994	4.870
25	0.473	0	1.000	30.381	0.590	0	1.000	24.356
Avg				10.512				6.070

Table 9.11: Multiple Inheritance Classification Tree(6 PEs and 8 PEs): Time and Speedup

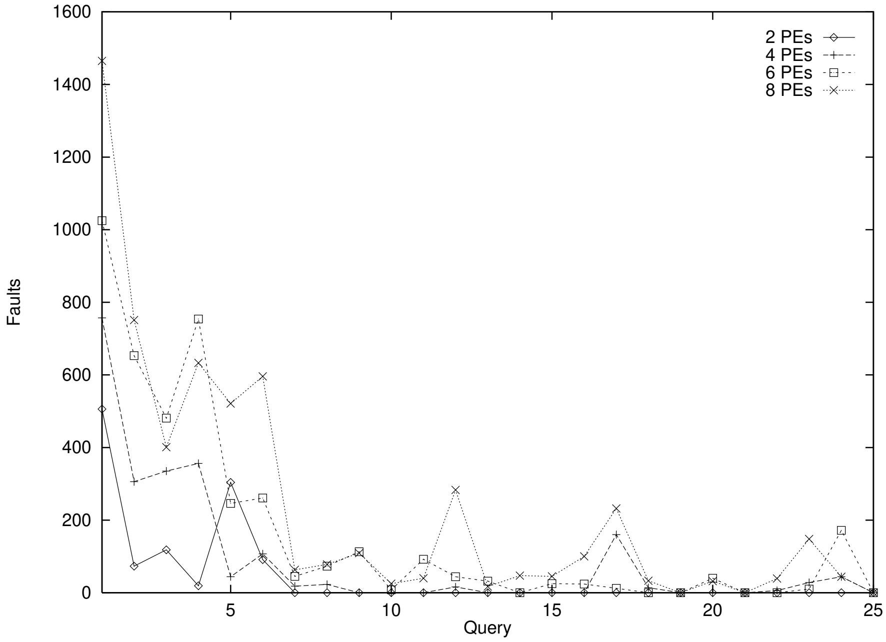

Figure 9.13: Multiple Inheritance Classification Tree: Faults

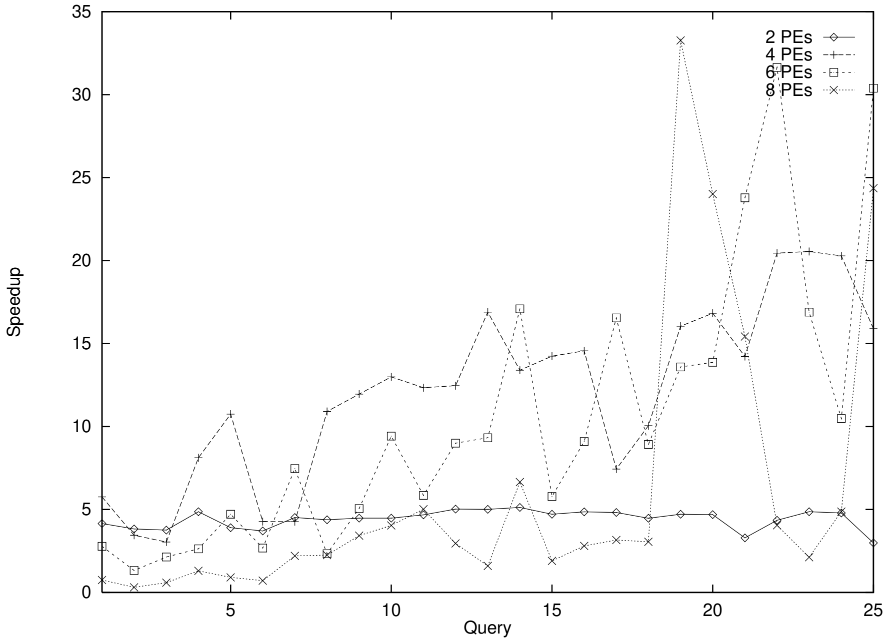

Figure 9.14: Multiple Inheritance Classification Tree: Speedup

Discussion

By observing the results listed, although we can get performance boost no matter how many PEs in the system, unfortunately in this experiment, the more PEs does not necessarily produce better performance. The maximum average speedup is achieved by using 4 PEs in the experiment. One primary reason in the experiment, while cache in the fewer PEs (2 and 4 PEs) systems can be filled up rather quickly, the cache in the more PEs (6 and 8 PEs) systems needs more query steps to be filled. As Table 9.11 shows almost all the queries need more or less nodes from other PEs for 8 PEs. The experiments also indicates that, except 2 PEs which keeps the performance more or less the same through query procedure, the speedup has the tendency of increasing as the query progress, but it also has the tendency of fluctuation.

Fig. 9.15 shows a picture of cache hit ratio v.s. speedup, the figure indicates that the more PEs in the system, the more sensitive the performance is in responding to hit ratio. The reason for this activity is in the lower number PEs system, the penalties are relatively low to access a remote node, but when the number of PEs increases, the penalties is severer. In order to reduce the number of faults and increase the hit ratio, a good node allocation strategy is crucial, specially to the performance gain in higher number PEs system.

Summary of Inheritance Experiments

Table 9.12 shows the statistical information related to the speedup of the above two inheritance experiments. When the number of PEs are small (2 PEs and 4 PEs, in this case), the speedups are relatively invariant for the same number of PEs no matter whether there

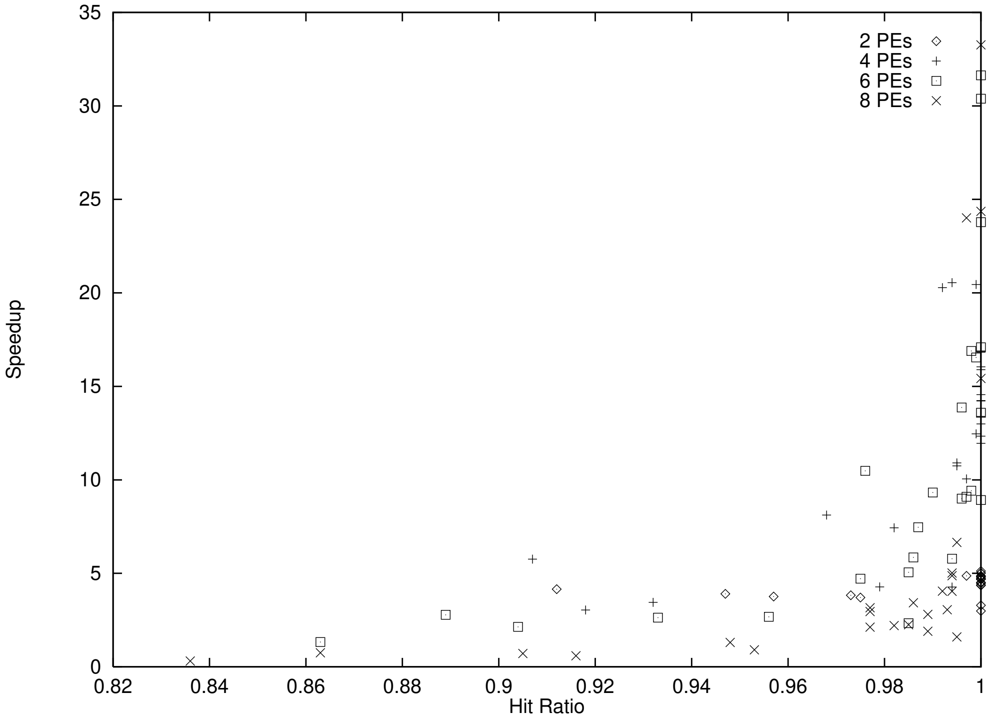

Figure 9.15: Multiple Inheritance Classification Tree: Hit Ratio v.s. Speedup

is a fault or not. The results indicate that the influences of knowledge faults are relatively mild when the number of PEs is low. But when the number of PEs is larger, the hit ratio plays a crucial factor in determining the performance of the system. The table indicates when faults do occur in the inference, the average performance of 6 PEs and 8 PEs are lower than 4 PEs. But when there is no fault, the average performance of 6 PEs and 8 PEs are better than 4PEs. The table indicates that the performance improves significantly for larger PEs (2.226 times faster in 6 PEs as of 3.462 times faster in 8 PEs).

The table also indicates that as the number of PEs increases, chances are the possibility of getting faults also increases. The reason is obvious: when dividing the same amount of semantic network information into more PEs, the less information each PE will get, and the possibility of requesting a node which is not in local also increases.

Faults	Speedup	2 PEs	4 PEs	6 PEs	8 PEs
Yes	Average Speedup	3.27	8.087	6.655	3.842
	Std. Dev. of Speedup	1.094	5.561	4.221	4.404
No	Average Speedup	3.518	10.579	14.819	15.247
	Std. Dev. of Speedup	1.076	3.797	7.057	6.527

Table 9.12: Speedup Distribution

Since the hit ratio is crucial to the performance, especially for the larger number of PEs, therefore the primary goal to increase the system performance is to increase the hit ratio. The general principles of hit ratio increase can be done in the following ways: the first strategy, as indicated in Section 9.3.1, is to rearrange the assertion sequences so that the maximum connectivity of the nodes within each PE can be achieved. Also, when the cache is filled with remote nodes, the larger the cache is, the lower the hit ratio will be. Lastly, as the queries go on, the local cache, which is empty at the beginning of the process, will be filled subsequently by the remote nodes during the journey. The tendency of getting a node which is not in local decreases. Therefore the system is in favor of long term query procedure.

9.4.3 Relational Database

Unlike the inheritance experiments which examine the impact of the remote node access on the performance, the database experiments are trying to illustrate the influences of the number of nodes on the performance. In this section, two wild card queries experiments are examined. Fig. 9.16 and Table 9.13 show the first experiment, which test results on the query (`find snum ?x`) on both the 5K assertions (9432 nodes) and 16K assertions (25238

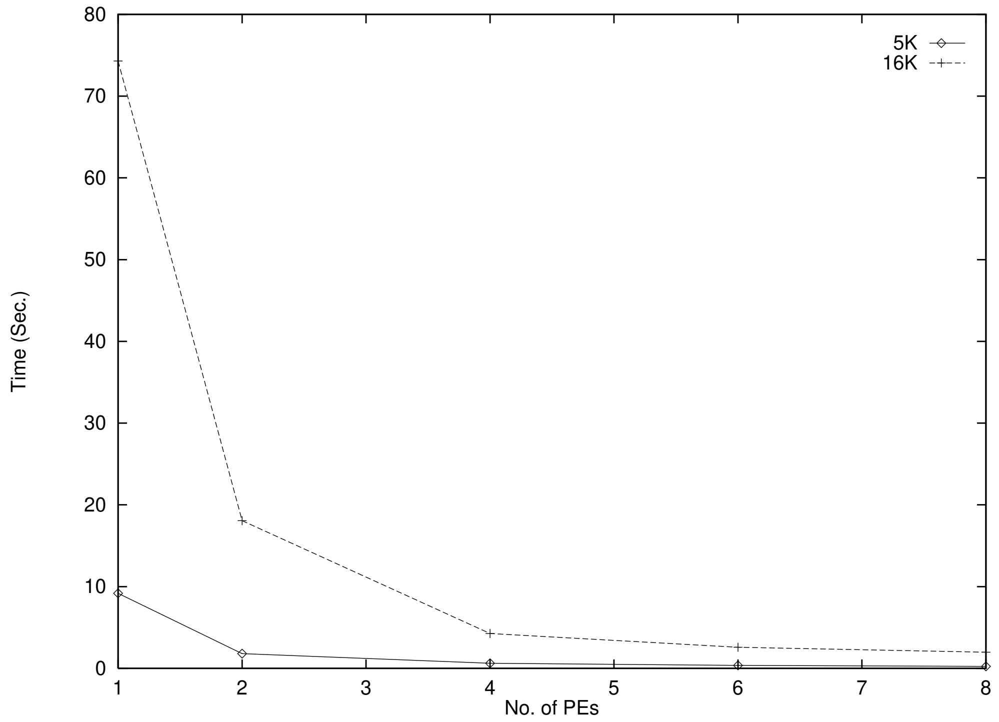

Figure 9.16: Database Experiment: 5K v.s. 16K

nodes) databases. Since TERRESA instantiates the wild card query inside the slave PEs and there is no need to access remote nodes in the most of the inferential procedures, the performance behavior is better than the ones with remote node access such as the experiments in the previous paragraphs. Also, the time of 5K assertions database is strictly less than the 16K ones due to fewer node access. The superlinear speedup phenomenon also occurs in the database test example. Also shown in the Table 9.13, the average number of comparison per PE gets smaller as the number of PEs increases.

Next, the queries of the format (`find` P_1 $?x_1$) or (`find` P_1 $?x_1$ P_2 $?x_2$), where either P_1 or P_2 could be any valid path, are randomly generated 25 times to test the performance of the system on the 5K assertions database. The query list can be seen in Appendix D.5. The results are listed in Table 9.14 and Table 9.15. Value N/A in the hit ratio indicates there is

PE	1	2	4	6	8
5K (Sec.)	9.18	1.8	0.62	0.37	0.25
Speedup	1	5.1	14.8	24.8	36.72
No. of comparison	$7.05 \cdot 10^6$	$1.77 \cdot 10^6$	$4.46 \cdot 10^5$	$2.00 \cdot 10^5$	$1.13 \cdot 10^5$
16K (Sec.)	74.29	18.08	4.26	2.58	1.97
Speedup	1	4.1	17.4	28.8	37.71
No. of comparison	$9.62 \cdot 10^7$	$2.41 \cdot 10^7$	$6.04 \cdot 10^6$	$2.69 \cdot 10^6$	$1.34 \cdot 10^6$

Table 9.13: Database Experiment: 5K v.s. 16K

no cache access at all in the query. In the test, because query seldom accesses the remote knowledge, the overhead of remote knowledge access can be eliminated in the most cases. In this experiment, the more the number of PEs, the less the time is needed. The average speedup ranges from 4.568 of 2 PEs up to 25.28 of 8 PEs.

Query	1 PE Time	2 PEs Time	2 PEs Faults	2 PEs Cache Hit Ratio	2 PEs Speedup	4 PEs Time	4 PEs Faults	4 PEs Cache Hit Ratio	4 PEs Speedup
1	7.355	1.788	0	N/A	4.114	0.512	N/A	0.000	14.365
2	7.445	1.738	0	N/A	4.284	0.598	N/A	0.000	12.450
3	54.854	11.762	318	0.792	4.664	9.738	985	0.576	5.633
4	7.758	1.852	0	N/A	4.189	0.589	N/A	0.000	13.171
5	17.780	4.181	0	N/A	4.253	1.194	N/A	0.000	14.891
6	7.415	1.808	0	N/A	4.101	0.708	N/A	0.000	10.473
7	43.818	7.682	0	N/A	5.704	1.995	N/A	0.000	21.964
8	7.613	1.93	0	N/A	3.945	0.624	N/A	0.000	12.200
9	7.754	1.883	0	N/A	4.118	0.469	N/A	0.000	16.533
10	7.480	1.659	0	N/A	4.509	0.397	N/A	0.000	18.841
11	7.427	1.75	0	N/A	4.244	0.416	N/A	0.000	17.853
12	44.963	8.718	0	1.000	5.157	2.500	0	1.000	17.985
13	8.449	1.937	0	N/A	4.362	0.718	N/A	0.000	11.767
14	64.245	11.472	0	N/A	5.600	3.278	N/A	0.000	19.599
15	41.354	9.122	44	0.972	4.533	3.075	97	0.959	13.448
16	58.249	11.419	9	0.992	5.101	2.653	27	0.981	21.956
17	62.239	13.602	0	1.000	4.576	3.609	0	1.000	17.245
18	41.765	9.124	0	1.000	4.577	2.472	0	1.000	16.895
19	61.457	14.117	0	N/A	4.353	4.365	N/A	0.000	14.079
20	47.720	8.996	0	1.000	5.305	2.246	0	1.000	21.247
21	17.174	4.06	0	N/A	4.230	1.053	0	1.000	16.310
22	62.190	11.561	0	1.000	5.379	3.143	0	1.000	19.787
23	67.777	13.087	0	N/A	5.179	3.507	N/A	0.000	19.326
24	8.714	2.649	0	1.000	3.290	1.973	N/A	0.000	4.417
25	7.677	1.734	0	1.000	4.427	0.565	0	1.000	13.588
Avg					4.568				15.441

Table 9.14: Database Performance (1 PE to 4 PEs)

Query	6 PEs				8 PEs			
	Time	Faults	Cache Hit Ratio	Speedup	Time	Faults	Cache Hit Ratio	Speedup
1	0.424	0	N/A	17.347	0.480	0	N/A	15.323
2	0.543	0	N/A	13.711	0.407	0	N/A	18.292
3	15.144	1550	0.396	3.622	34.716	2232	0.187	1.580
4	0.369	0	N/A	21.024	0.562	0	N/A	13.804
5	0.857	0	N/A	20.747	0.598	0	N/A	29.732
6	0.554	0	N/A	13.384	0.500	0	N/A	14.830
7	1.395	0	N/A	31.411	0.972	0	N/A	45.080
8	0.413	0	N/A	18.433	0.306	0	N/A	24.879
9	0.578	0	N/A	13.415	0.626	0	N/A	12.387
10	0.456	0	N/A	16.404	0.546	0	N/A	13.700
11	0.348	0	N/A	21.342	0.423	0	N/A	17.558
12	1.827	0	1.000	24.610	1.271	0	1.000	35.376
13	0.228	0	N/A	37.057	0.549	0	N/A	15.390
14	2.352	0	N/A	27.315	1.198	0	N/A	53.627
15	3.589	169	0.936	11.522	3.978	218	0.921	10.396
16	3.12	65	0.960	18.670	4.568	114	0.932	12.752
17	2.188	0	1.000	28.446	1.740	0	1.000	35.770
18	1.667	0	1.000	25.054	0.973	0	1.000	42.924
19	2.678	0	N/A	22.949	1.675	0	N/A	36.691
20	1.56	0	1.000	30.590	1.113	0	1.000	42.875
21	0.852	0	N/A	20.157	1.035	0	N/A	16.593
22	2.738	0	1.000	22.714	1.247	0	1.000	49.872
23	2.73	0	N/A	24.827	1.315	0	N/A	51.541
24	2.332	0	1.000	3.737	2.548	0	1.000	3.420
25	0.375	0	1.000	20.472	0.436	0	1.000	17.608
Avg				20.358				25.280

Table 9.15: Database Performance (6 PEs and 8 PEs)

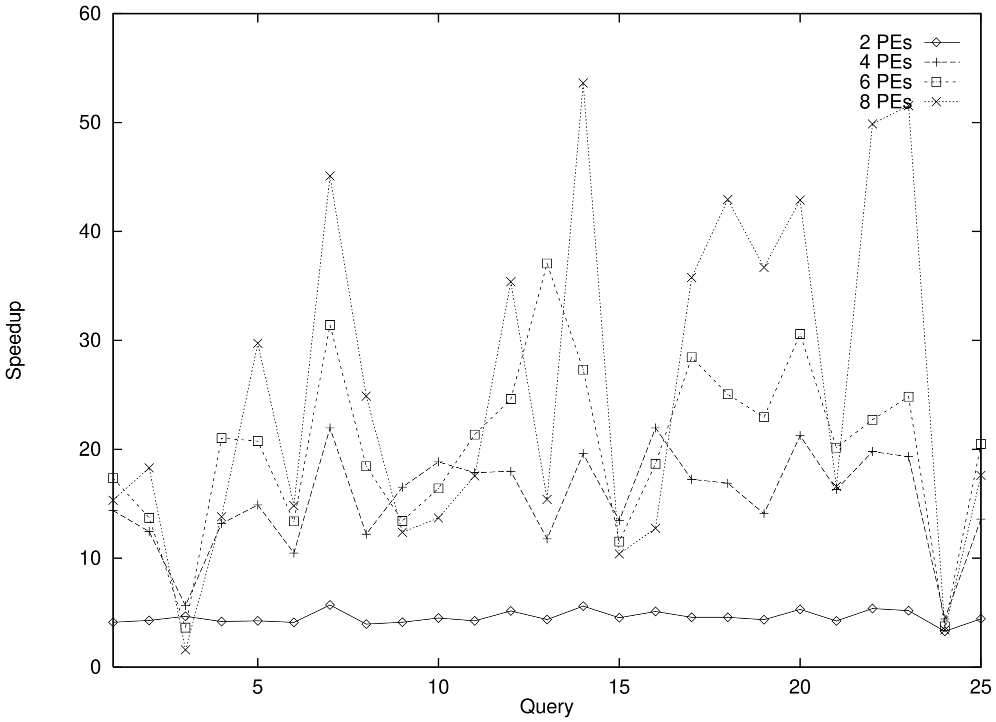

Figure 9.17: Database Speedup

Chapter 10

Conclusion and Future Research

It is obvious that parallel processing is necessary for large knowledge-based systems. A well-designed parallel system can deliver answers much faster than a sequential system, and the amount of knowledge stored in the system can be augmented by increasing the number of processing elements (PEs). Several parallel semantic network systems have been implemented previously, but all these previous systems used the "marker passing algorithm" as their inference mechanism. As described in Chapter 2, the marker passing algorithm has some limitations when implemented on MIMD distributed memory machines. Since the marker passing algorithm is a synchronous algorithm, it is easy to implement on SIMD machines but not on MIMD machines since most MIMD machines are asynchronous systems. Furthermore, the nodes of the semantic network are statically assigned to processors in the marker passing algorithm, and hence the load can become extremely imbalanced. When the marker passing algorithm is implemented on fine-grain SIMD machines, efficiency deteriorates due to relatively large overhead. Finally, implementing the semantic network on a medium or coarse-grain MIMD machine is very difficult. Firstly, since the semantic network is a fine-grain knowledge representation scheme, only a few instructions are needed for each basic operation. Second, its computation structure is highly irregular and dynamic. Therefore, efficient runtime load balance is necessary. Third, its heavy communication and random communication structure may offset any benefit of parallelization.

The primary contribution of this research is the design and implementation of a parallel semantic network system on MIMD distributed memory machines that overcomes the limitations described above. We tackled these problems by using a task-based message-driven model. This is an asynchronous model for MIMD machines. In order to balance the loads, queries must be decomposable and the decomposed queries must be capable of migration. In our implemented system, TERRESA, the query is decomposed into task and post task process (ptask). This mechanism ensures that either the task or the ptask can be migrate to other PEs so that the goal of load balance can be achieved. The communication overhead can be reduced by using an object-oriented packing system which packs many small messages into a larger message. Also, a significant amount of the communication demand can be eliminated by caching the shared knowledge. By using these advanced parallelization techniques, an efficient implementation on the MIMD machine becomes possible. TERRESA has been developed and tested on the SUN SPARC symmetric multiprocessors server. With portability in mind, TERRESA is written in MPI, ANSI C, lex, and yacc and can be easily ported to other parallel platforms.

In TERRESA, two collaborating modules, the host module and the slave module, are integrated together to finish the given queries. The host module is responsible for interacting with users and dispatching tasks to the slave modules, and the slave module performs actual computation. The host module is composed of several functional modules including the language front end, the host knowledge base, the command processing module, the host answer processing module, and the object-oriented packing system. Apart from the module for task execution, the slave module is composed of several smaller functional modules like the shared knowledge module, the duplicate checking module, slave scheduler, and object-oriented packing system, etc.

Our experimental results indicate that when the network building commands are proper arranged and after the cache is filled, queries benefit when inference procedures are run in parallel and speedup can be achieved. Analysis indicates that when the remote access is necessary, the average speedup ranges from 3.27 to 8.08 in our experiment. Unfortunately under this condition, the more PEs do not guarantee the more power you can squeeze out of the parallel box. However, when there is no remote access during the execution, the more PEs you have, the faster the inference will be. Under the circumstance, the average speedup ranges from 3.51 to 15.24. The maximum average speedup can be achieved at 8 PEs. The experiments also indicates that the superlinear speedup occurs from time to time, which is the result of search tree in the address lookup procedures in the shared knowledge module.

The primary goal of parallelizing an application is performance. Of course the current system implementation needs further polish and optimization to boost its speed. But how can we increase performance without attuning the source code? Our main conclusions from these experiments indicated the following rules:

- Rearrange the assertion sequence: As indicated previously, the assertion sequence affects the partitioning of the semantic network among PEs and the partitioning has a profound influence on the number of remote knowledge access as well as performance.

- Increase cache knowledge capacity: With larger cache, the possibility of getting faults can be reduced after the cache is filled. Moreover, if the physical memory on the machine is sufficient, the optimum case will be to duplicate the whole knowledge base on each PE.

- In favor of long term execution: Knowledge faults are inevitable as the system starts. As the queries proceeds, the cache is filled with the nodes retrieved from remote PEs, so the possibility of fault occurrences reduced as the queries go on.

10.1 Future Research

There are many intriguing possibilities for research involving TERRESA. Here are some potential interesting exploration directions:

1. **Node Allocation Strategy**: One of the most influential factors in TERRESA performance is the hit ratio of the cache knowledge, and the hit ratio depends on how the nodes distributes among PEs. As we mentioned in Chapter 9, poor allocation strategy produces disastrous performance, although the phenomena can be alleviated as

the query goes on, how to revise the partition the network and get better performance is an interesting research and worth more investigation.

2. **Node-based Inference**: One immediate goal is to integrate SNePS node-based inference scheme into TERRESA. Node-based inference involves unification and backward chaining inference, which are much more complicated than path-based inference. A careful design must be taken to integrate these functionalities into the task-based message-driven model. In addition, since node-based inference could introduce new knowledge into the system, the shared-knowledge system needs to extend its capability to be writable, therefore the data coherence problem needs to be taken care of as well.

    (a) Shared knowledge module: When node-based inference mechanism is added into the system, the shared knowledge module needs to be augmented to writable. The write module can be further split into several sub-procedures: including existence verification (check whether the node is in the system already), location assignment (assign which PE should this node belongs to), etc. Distributed shared memory locking mechanism and coherence protocol can be used in designing this module.

    (b) Inference: To achieve node based inference, several new tasks and ptasks must be generated. For example, take a look at Fig. 10.1 which excerpted from SNePS UVBR (Unique Variable Binding Rule) demo. The nodes indicate the rule of "If x and y are brothers, then x likes y" and the fact of "Bob, Joe and Harry are

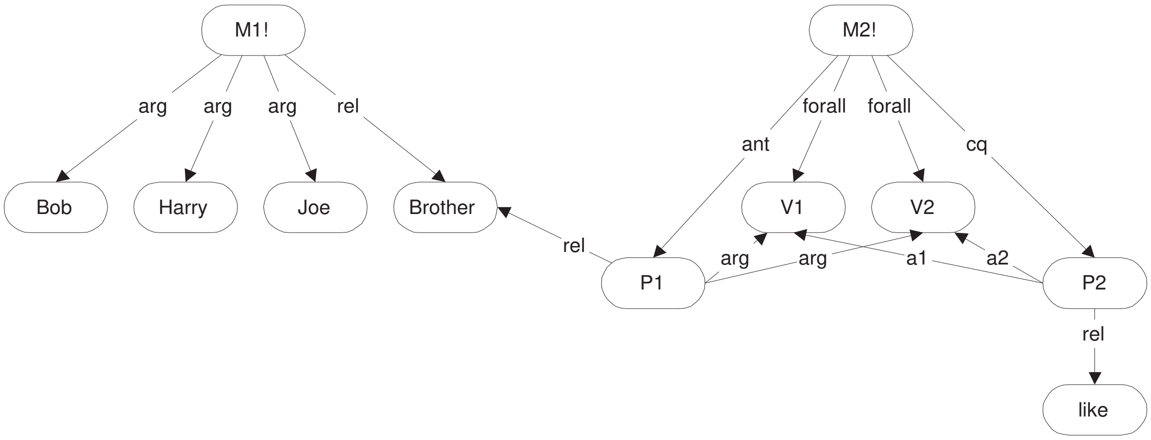

Figure 10.1: A Semantic Network Example representing the concepts of "If x and y are brothers, then x likes y" and the fact of "Bob, Joe and Harry are brothers".

brothers". When the query (`deduce rel like a1 *brother a2 *brother`) is issued, the preliminary plan for node-based inference can be seen as follows: TERRESA generates a task and ptask for initial subnetwork matching. When the subnetwork is found (in the example is (`P2 (rel like) (a1 v1) (a2 v2)`) and returned to the ptask, a new unification task/ptask pair is generated, which performs unification, UVBR can be applied at the stage. After getting the unifiers, since this is a consequence of some rules, therefore for each rule, a backward chaining task/ptask pair is generated to process this scenario. `AND`, `OR`, and numerical entailment can be handled at this stage. If the antecedents are not any part of the rule, a message is sent back to report the finding, otherwise the unification/backward chaining procedure is executed repeatedly until no more rules can be triggered.

3. **Middleware**: The parallel library we are using, MPI, though versatile and powerful, is still too primitive to provide an elegant solution for TERRESA implementa-

tion. A higher level and more abstract middleware, especially the object oriented parallel environments or languages, might better facilitate the TERRESA modeling and program development. One possible interesting research topic in the future is to build a run time support system which supports a multithreading task-based, message-driven model. By doing this the entire system design can be simplified.

4. **Database Technologies**: Database research possesses similar yet distinct features with the knowledge base system. Some of the research teams, for example, PARKA, has begun to integrate database technologies into their system [62]. Yet to integrate database technologies into the TERRESA might generate new challenges. For example, in the current TERRESA design, there are only two levels of knowledge storage, private and cache. In order to keep vast amount of knowledge, the disks must be included into the the system design. When we introduce disks as the third level of storage, the accessibility and coherence may become even more complex. How to integrate these technologies together while maintaining good performance is a very interesting topic to be explored.

Appendix A

TERRESA Grammars

Appendix A lists the grammar of TERRESA.

$\langle$sneps_command$\rangle$ ::$\Rightarrow$ ($\langle$path_defn_command$\rangle$)
| ($\langle$file_command$\rangle$)
| ($\langle$delete_command$\rangle$)
| ($\langle$misc_command$\rangle$)
| $\langle$multi_node_command$\rangle$
| $\langle$snepsul_var$\rangle$ ■

$\langle$path_defn_command$\rangle$ ::$\Rightarrow$ DEFINE $\langle$unitpath$\rangle^{+}$
| UNDEFINE $\langle$unitpath$\rangle^{+}$ ■

$\langle$file_command$\rangle$ ::$\Rightarrow$ INNET "$\langle$string$\rangle$"
| OUTNET "$\langle$string$\rangle$" ■

$\langle$delete_command$\rangle$ ::$\Rightarrow$ RESETNET 'T
| RESETNET 'NIL
| RESETNET ■

$\langle$misc_command$\rangle$ ::$\Rightarrow$ LISP
| $\langle$dup_check_command$\rangle$
| $\langle$load_bal_command$\rangle$
| $\langle$knowledge_base_command$\rangle$ ■

⟨dup_check_command⟩ ::⇒ DUPCHECK 'T [1]
| DUPCHECK 'NIL
| DUPCHECK ■

⟨load_bal_command⟩ ::⇒ LOADBAL 'T [2]
| LOADBAL 'NIL
| LOADBAL ■

⟨knowledge_base_command⟩ ::⇒ PRIVATE ⟨integer⟩[3]
| PRIVATE ALL
| CACHE ⟨integer⟩
| CACHE ALL ■

⟨multi_node_command⟩ ::⇒ (⟨multiple_nodes⟩)
| ⟨node_command⟩ ■

⟨multiple_nodes⟩ ::⇒ ⟨node_command⟩$^+$ ■

⟨node_command⟩ ::⇒ (⟨inference_command⟩)
| (⟨display_command⟩)
| (⟨net_build_command⟩)
| (⟨nodeset_op_command⟩)
| ⟨snepsul_var⟩
| ⟨multi_dollar_node⟩
| ⟨multi_hash_node⟩
| ⟨node_name⟩ ■

⟨inference_command⟩ ::⇒ FIND ⟨multi_path_nodeset⟩
| FINDASSERT ⟨multi_path_nodeset⟩
| FINDCONSTANT ⟨multi_path_nodeset⟩
| FINDBASE ⟨multi_path_nodeset⟩
| FINDVARIABLE ⟨multi_path_nodeset⟩
| FINDPATTERN ⟨multi_path_nodeset⟩ ■

⟨display_command⟩ ::⇒ DUMP ⟨multi_node_command⟩
| DESCRIBE ⟨multi_node_command⟩ ■

⟨net_build_command⟩ ::⇒ ASSERT ⟨multi_relation_nodeset⟩
| BUILD ⟨multi_relation_nodeset⟩ ■

---

[1] Duplicate checking switch and status display
[2] Load balance switch and status display
[3] Slave knowledge base display, either specific PE of all PEs

⟨nodeset_op_command⟩ ::⇒ & ⟨multi_node_command⟩ ⟨multi_node_command⟩
| + ⟨multi_node_command⟩ ⟨multi_node_command⟩
| - ⟨multi_node_command⟩ ⟨multi_node_command⟩
| = ⟨multi_node_command⟩ ⟨symbol⟩
| _ ⟨multi_node_command⟩ ⟨unitpathset⟩
| > ⟨unitpathset⟩ ⟨symbol⟩ ■

⟨snepsul_var⟩ ::⇒ * ⟨symbol⟩ ■

⟨multi_dollar_node⟩ ::⇒ ⟨dollar_node⟩
| (⟨dollar_nodeset⟩) ■

⟨dollar_nodeset⟩ ::⇒ ⟨dollar_node⟩$^+$ ■

⟨dollar_node⟩ ::⇒ $ ⟨symbol⟩ ■

⟨unitpathset⟩ ::⇒ ⟨unitpath⟩
| (⟨multi_unitpath⟩) ■

⟨multi_unitpath⟩ ::⇒ ⟨unitpath⟩$^+$ ■

⟨node_name⟩ ::⇒ ⟨symbol⟩
| ⟨integer⟩ ■

⟨multi_hash_node⟩ ::⇒ ⟨hash_node⟩
| (⟨hash_nodeset⟩) ■

⟨hash_nodeset⟩ ::⇒ ⟨hash_node⟩ $^+$ ■

⟨hash_node⟩ ::⇒ # ⟨symbol⟩ ■

⟨question_node⟩ ::⇒ ? ⟨symbol⟩ ■

⟨multi_relation_nodeset⟩ ::⇒ ⟨relation_nodeset⟩$^+$ ■

⟨relation_nodeset⟩ ::⇒ ⟨relation⟩ ⟨multi_node_command⟩ ■

⟨multi_path_nodeset⟩ ::⇒ ⟨path_nodeset⟩$^+$
| ⟨path_question_node⟩
| ⟨multi_path_nodeset⟩ ⟨path_question_node⟩ ■

⟨path_question_node⟩ ::⇒ ⟨path⟩ ⟨question_node⟩ ■

$$\langle \text{path\_nodeset} \rangle ::\Rightarrow \langle \text{path} \rangle \ \langle \text{multi\_node\_command} \rangle \ \blacksquare$$

$$
\begin{aligned}
\langle \text{path} \rangle ::\Rightarrow\ & \langle \text{unitpath} \rangle \\
& |\ (\ \mathsf{COMPOSE}\ \ \langle \text{multi\_path} \rangle\ ) \\
& |\ (\ \mathsf{KSTAR}\ \ \langle \text{path} \rangle\ ) \\
& |\ (\ \mathsf{KPLUS}\ \ \langle \text{path} \rangle\ ) \\
& |\ (\ \mathsf{OR}\ \ \langle \text{multi\_path} \rangle\ ) \\
& |\ (\ \mathsf{AND}\ \ \langle \text{multi\_path} \rangle\ ) \\
& |\ (\ \mathsf{NOT}\ \ \langle \text{path} \rangle\ ) \\
& |\ (\ \mathsf{RELATIVE\_COMPLEMENT}\ \ \langle \text{path} \rangle\ \langle \text{path} \rangle\ ) \\
& |\ (\ \mathsf{IRREFLEXIVE\_RESTRICT}\ \ \langle \text{path} \rangle\ ) \\
& |\ (\ \mathsf{DOMAIN\_RESTRICT}\ \ (\ \langle \text{path} \rangle\ \langle \text{node\_name} \rangle\ )\ \langle \text{path} \rangle\ ) \\
& |\ (\ \mathsf{RANGE\_RESTRICT}\ \ \langle \text{path} \rangle\ (\ \langle \text{path} \rangle\ \langle \text{node\_name} \rangle\ )\ ) \\
& |\ (\ \langle \text{multi\_path} \rangle\ )\ \blacksquare
\end{aligned}
$$

$$\langle \text{multi\_path} \rangle ::\Rightarrow \langle \text{path} \rangle^{+}\ \blacksquare$$

$$
\begin{aligned}
\langle \text{relation} \rangle ::\Rightarrow\ & \langle \text{unitpath} \rangle \\
& |\ \langle \text{unitpath} \rangle\text{-}\ \ \blacksquare
\end{aligned}
$$

$$\langle \text{unitpath} \rangle ::\Rightarrow \langle \text{symbol} \rangle\ \blacksquare$$

$$\langle \text{digit} \rangle ::\Rightarrow [\mathsf{0\text{-}9}\ ]\ \blacksquare$$

$$\langle \text{non\_neg\_char} \rangle ::\Rightarrow [\mathsf{0\text{-}9a\text{-}zA\text{-}Z\_}\ ]\ \blacksquare$$

$$\langle \text{integer} \rangle ::\Rightarrow \langle \text{digit} \rangle^{+}\ \blacksquare$$

$$\langle \text{symbol} \rangle ::\Rightarrow \langle \text{non\_neg\_char} \rangle^{+}\ \blacksquare$$

$$\langle \text{string} \rangle ::\Rightarrow \langle \text{char} \rangle^{+}\ \blacksquare$$

Appendix B

Test Semantic Networks Generation Program

The Appendix B lists the knowledge generation programs. Three of the generation programs are listed, the single inheritance tree, the multiple inheritance classification tree and the relational database.

B.1 Single Inheritance Classification Tree

The following C program generates a single inheritance classification tree knowledge base file. Two adjustable parameters: `HEIGHT` limits the height of the classification tree, or `TOTAL` limits total number of assertions in the knowledge base. The assertions are generated according to the depth first traversal of the tree.

```
#define HEIGHT 7
```

```
#define TOTAL  20000

#include <stdio.h>
#include <math.h>

main()
{
  int i, j, tot;

  printf("(define subclass supclass)\n");
  tot = 0;
  rec_gen(1, 2);

  printf("(outnet \"bintree.img\")\n");
  printf("(lisp)\n");
}

rec_gen(int i, int depth)
{
  if(depth > HEIGHT) return;
  else {
    if(2 * i <= TOTAL) {
      printf("(assert supclass B%d subclass B%d)\n", i, 2*i);
      rec_gen(i*2, depth+1);

      printf("(assert supclass B%d subclass B%d)\n",
             i, 2*i + 1);
      rec_gen(i*2+1, depth+1);
    }
  }
}
```

B.2 Multiple Inheritance Classification Tree

The following C program generates a multiple inheritance classification tree knowledge base file. `HEIGHT` is the tree height, This program first generates two single inheritance classification trees independently, then it creates interconnection edges between two classification trees. The assertions within a tree are generated according to the depth first traversla strategy.

```
#define HEIGHT 5
#include <stdio.h>
#include <math.h>

main()
{
  printf("(define subclass supclass)\n");
  rec_gen(1, 1, 2);
  rec_gen(2, 1, 2);
  inv_rec_gen(1, 2);
  inv_rec_gen(2, 1);

  printf("(outnet \"rec-fattree.img\")\n");
  printf("(lisp)\n");
}

rec_gen(int treeidx, int i, int depth)
{
  if(depth > HEIGHT) return;
  else {
    printf("(assert supclass B%d-%d subclass B%d-%d)\n",
           treeidx, i , treeidx, 2*i);
    rec_gen(treeidx, i*2, depth+1);

    printf("(assert supclass B%d-%d subclass B%d-%d)\n",
           treeidx, i , treeidx, 2*i + 1);
    rec_gen(treeidx, i*2+1, depth+1);
  }
}

inv_rec_gen(int treeidx, int inv_idx)
{
  int tot, i;

  tot = pow(2, HEIGHT);
  for(i = 2; i < tot; i++)
    printf("(assert supclass B%d-%d subclass B%d-%d)\n",
           treeidx, i/2 , inv_idx, i);
}
```

B.3 Relational Database

The following C program generates a test example for relational database randomly. The ratio of the number of the rules (`assert snum` xxx `sname` ...) : (`assert pnum` yyy `pname` ...) : (`assert jnum` zzz `jname` ...) : (`assert snum` xxx `jnum` yyy `pnum` zzz `qty` uuu) is 1 : 1 : 1 : 5. Total number of assertions can be input at the command line.

```
#define SNUMRATE 1
#define PNUMRATE 1
#define JNUMRATE 1
#define SPJRATE  5
#define CITYRATE 1
#include <stdlib.h>
#include <stdio.h>

main(int argc, char **argv)
{
  int i, count, SNUMSIZE, PNUMSIZE, JNUMSIZE, SPJSIZE, CITYSIZE;
  float TOT;

  if(argc != 2) {
    printf("Incorrect number of parameter\n");
    exit(-1);
  } else {
    count = atoi(argv[1]);
    TOT = SNUMRATE + PNUMRATE + JNUMRATE + SPJRATE;
    SNUMSIZE = SNUMRATE / TOT * count;
    PNUMSIZE = PNUMRATE / TOT * count;
    JNUMSIZE = JNUMRATE / TOT * count;
    SPJSIZE =  SPJRATE  / TOT * count;
    CITYSIZE = CITYRATE / TOT * count;
    if(TOT > count) {
      printf("Less then the total size\n");
      exit(-1);
    }
  }

  srand(7);
  printf("(define snum sname status city pnum pname color
```

```
  weight jnum jname qty)\n");
  for(i = 0; i < SNUMSIZE; i++) {
    printf("(assert snum s%d sname sname%d status %d city
           city%d)\n", i, i, (rand()%10+1)*10,
           rand()%CITYSIZE+1);
  }

  for(i = 0; i < PNUMSIZE; i++) {
    printf("(assert pnum p%d pname pname%d color color%d
           weight %d city city%d)\n", i, i, (rand()%50+1)*10,
           rand() % 20 + 1, rand()%CITYSIZE+1);
  }

  for(i = 0; i < JNUMSIZE; i++) {
    printf("(assert jnum j%d jname jname%d city city%d)\n",
           i, i, rand()%CITYSIZE+1);
  }

  for(i = 0; i < SPJSIZE; i++) {
    printf("(assert snum s%d pnum p%d jnum j%d qty %d)\n",
           rand()%SNUMSIZE+1, rand()%PNUMSIZE+1,
           rand()%JNUMSIZE+1, (rand()%10+1) * 100);
  }

  printf("(outnet \"db.sneps\")\n(lisp)\n");
}
```

Appendix C

Query Generation Program

The following programs are the query generation programs for the single / multiple inheritance classifitation trees, and relational database. All the programs are written in `PERL`.

C.1 Single Inheritance Classification Tree

The program generates the queries for the single inheritance classification tree in the format

(`find` (`subclass- supclass` ... `subclass- supclass`) (n_{i1} n_{i2} ... n_{ik}))

The number of path pair (`subclass- supclass`) ranges from 8 to 13 and the number of nodes k ranges from 8 to 20.

```
#!/util/bin/perl

srand(time);
print "(innet \"bintree-8k-2pe.img\")\n";
for($i = 0; $i < 25; $i++) {
    $path_len = int(rand(5) + 8); #8..13
```

```
    $node_size = int(rand(12) + 8); #8..20

    print "(find (";
    for($j = 0; $j < $path_len; $j++) {
        print "subclass- supclass ";
    }
    printf ") (";

    for($k = 0; $k < $node_size; $k++) {
        $m = int(rand(96)+1);
        $rand_node[$k] = $m;
    }
    @kk = sort {$a <=> $b} @rand_node;

    for($k = 0; $k < $node_size; $k++) {
        print "B", $kk[$k], " ";
    }

    print "))\n";
}
print "(lisp)\n";
```

C.2 Multiple Inheritance Classification Tree

The program generates the queries for the multiple inheritance classification tree in the format

(find (subclass- supclass ... subclass- supclass) ($n_{1,1}$ $n_{1,2}$ $\ldots$ $n_{1,i}$ $n_{2,1}$ $\mathrm{n}_{2,2}$ $\ldots$ $n_{2,j}$))

where $n_{1,k}$, $k = 1 \ldots i$ is the node in the first tree, and $n_{2,l}$ $l = 1 \ldots j$ is the node in the second tree.

The number of path pair (subclass- supclass) ranges from 6 to 10 and depending on the path length, the number of initial nodes ranges from 8 to 50.

```
#!/util/bin/perl

srand(time);
print "(innet \"rec-fattree-8k-2pe.img\")\n";
for($i = 0; $i < 10; $i++) {
    print "(find (";
    $path_len = int(rand(5) + 6); #6 .. 10
    if($path_len % 2 == 0) {
        $start = 0;
        $fin = $path_len;
    } else {
        $start = 1;
        $fin = $path_len + 1;
    }
    for($j = $start; $j < $fin; $j++) {
        $k = $j % 2;
        if($k == 0) {
            print "subclass- ";
        } else {
            print "supclass ";
        }
    }
    printf ") (";

    if($path_len == 6) {
        $totnode = 10 + int(rand(40));
    } elsif($path_len == 7) {
        $totnode = 10 + int(rand(30));
    } elsif($path_len == 8) {
        $totnode = 10 + int(rand(20));
    } elsif($path_len == 9) {
        $totnode = 10 + int(rand(10));
    } else {
        $totnode = 8 + int(rand(2));
    }

    $pe1node = $totnode / 2;
    $pe2node = $totnode / 2;

    for($k = 0; $k < $pe1node; $k++) {
        $rand_node[$k] = int(rand(64)+1);
    }
    @kk = sort {$a <=> $b} @rand_node;
```

```
    for($k = 0; $k < $pe1node; $k++) {
        print "B1-", $kk[$k], " ";
    }

    # --------------------------------

    for($k = 0; $k < $pe2node; $k++) {
        $rand_node[$k] = int(rand(64)+1);
    }
    @kk = sort {$a <=> $b} @rand_node;

    for($k = 2; $k < $pe2node; $k++) {
        print "B2-", $kk[$k], " ";
    }
    print "))\n";
}

print "(lisp)\n";
```

C.3 Relational Database

This PERL program generates the queries for the relational database.

```
#!/util/bin/perl
@symbol=("compose", "kstar", "kplus", "and", "or", "not",
"relative-complement", "irreflexive-restrict", "domain-restrict",
"range-restrict", "snum", "sname", "status", "city", "pnum",
"pname", "color", "weight", "jnum", "jname", "qty", "snum-",
"sname-", "status-", "city-", "pnum-", "pname-", "color-",
"weight-", "jnum-", "jname-", "qty-");

srand(time());
print "(innet \"db5k-2pe.sneps\")\n";
for($i=0; $i < 25; $i++) {
    $k = intrand(2) + 1;
    print "(find ";
    for($j = 0; $j < $k; $j++) {
        print_path();
```

```
        print " ?x$j ";
    }
    print ")\n";
}
print "(lisp)\n";

sub print_path {
    $thisi = intrand(32);
    if($thisi < 10) {
        print "(";
        print $symbol[$thisi], " ";
        if($thisi == 0) {
            # COMPOSE
            $j = intrand(3) + 2;
            for($count = 0; $count < $j ; $count++) {
                print_path();
            }
        } elsif($thisi == 1) {
            #KSTAR
            print " ("; print_path(); print ")";
        } elsif($thisi == 2) {
            #KPLUS
            print_path();
        } elsif($thisi == 3) {
            #AND
            $j = intrand(3) + 2;
            for($count = 0; $count < $j ; $count++) {
                print_path();
            }
        } elsif($thisi == 4) {
            #OR
            $j = intrand(3) + 2;
            for($count = 0; $count < $j ; $count++) {
                print_path();
            }
        } elsif($thisi == 5) {
            #NOT
            print_path();
        } elsif($thisi == 6) {
            #RELATIVE-COMPLEMENT
            print_path();
            print_path();
        } elsif($thisi == 7) {
```

```
            #IRREFLEXIVE-RESTRICT
            print_path();
        } elsif($thisi == 8) {
            #DOMAIN-RESTRICT
            print "(";  print_path(); print " s1)";
            print_path();
        } elsif($thisi == 9) {
            #RANGE-RESTRICT
            print_path();
            print "(";  print_path(); print " s1)";
        }
        print ")";
    } else {
        print" "; print $symbol[$thisi];  print" ";
    }
    return;
}

sub randnode()
{
    $SUMRATE = 1.0;
    $PNUMRATE = 1.0;
    $JNUMRATE = 1.0;
    $SPJRATE = 5.0;
    $CITYRATE = 1.0;
    $count = 16384.0;

    $TOT = $SNUMRATE + $PNUMRATE + $JNUMRATE + $SPJRATE;
    $SNUMSIZE = $SNUMRATE / $TOT * $count;
    $PNUMSIZE = $PNUMRATE / $TOT * $count;
    $JNUMSIZE = $JNUMRATE / $TOT * $count;
    $SPJSIZE =  $SPJRATE  / $TOT * $count;
    $CITYSIZE = $CITYRATE / $TOT * $count;

    $i = intrand(10);

    if($i == 0) {
        print "?x";
    } elsif($i == 1) {
        print "s"; print intrand($SNUMSIZE)+1;
    } elsif($i == 2) {
        print "sname"; print intrand($SNUMSIZE)+1;
    } elsif($i == 3) {
```

```
        print intrand(10)+1;
    } elsif($i == 4) {
        print "p"; print intrand($PNUMSIZE)+1;
    } elsif($i == 5) {
        print "pname"; print intrand($PNUMSIZE)+1;
    } elsif($i == 6) {
        print "color"; print intrand(50)+1;
    } elsif($i == 7) {
        print "weight"; print intrand(20)+1;
    } elsif($i == 8) {
        print "city"; print intrand($CITYSIZE);
    }
}

sub intrand()
{
    my($i) = @_;
    return (int(rand($i)));
}
```

Appendix D

Test Example

D.1 Single Inheritance Classification Tree - Remote Access v.s. Local Access

Queries for the first experiment in Sec. 9.4.1.

```
(innet "bintree-8k-3pe.img")
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass) (b8 b9 b10 b11 b12 b13 b14 b15))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass) (b8 b9 b10 b11 b12 b13 b14 b15))
(lisp)
```

D.2 Single Inheritance Classification Tree

Queries for the second experiment in Sec. 9.4.1. The test is generated by the program in Appendix C.1.

```
(innet "bintree-8k-3pe.img")
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass) (B4 B4 B8 B16 B22 B24 B28 B37 B39
B43 B54 B57 B61 B63 B73))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass) (B5 B7 B8 B13 B18 B27 B34 B46 B50
B51 B64 B65 B66 B68 B73 B96))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass) (B1 B4 B6 B8 B21 B25 B36 B36 B40 B52 B53))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B1 B3 B7 B14 B16 B20 B24
B27 B30 B31))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B2 B8 B10 B13 B14 B18
B23))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass) (B4 B6 B8 B14 B16 B21 B25 B26 B32
B36 B38 B40))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
```

```
supclass subclass- supclass subclass- supclass
subclass- supclass) (B17 B19 B22 B26 B35 B36 B40 B55
B58 B61 B73 B77 B95))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass) (B2 B8 B9 B14 B15 B17 B19 B30))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B3 B4 B5 B8 B14
B15 B19 B33 B42 B53 B60 B61 B64))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B4 B6 B8 B12 B19 B42 B44
B47 B50 B62 B70))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass ) (B5 B8 B8 B19 B20 B32 B36 B42 B44
B46 B55))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B4 B8 B9 B10
B16 B19 B24 B31 B44 B47 B53 B56 B57 B58 B77))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B4 B10 B19 B26
B30 B36 B44 B45 B49 B51 B56 B69))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B8 B17 B22 B24
B30 B41 B44 B51 B59 B64 B71 B81 B82 B84 B88 B89 B91))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
```

```
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B2 B7 B8 B10 B13 B21 B23
B24 B25 B27 B44 B46 B69 B70 B80))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass) (B8 B18 B24 B39 B44 B48 B50 B51 B52 B53 B56
B69 B70 B77 B87 B91))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B1 B11 B14 B20 B24 B28
B33 B38 B54 B56 B57 B59 B66 B81 B89 B92))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass ) (B4 B11 B21 B24 B38 B39 B42 B53
B54 B55 B66))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass) (B11 B15 B17 B20 B24 B31 B36 B39
B40 B41 B54))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B10 B10 B11 B22
B24 B34 B36))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass) (B1 B5 B9 B20 B27 B41 B42 B47 B51 B62 B66 B79
B90 B93 B94))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass) (B9 B20 B26 B27 B32 B35 B36 B44 B49
B51 B56 B62 B78))
(find (subclass- supclass subclass- supclass subclass-
```

```
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass) (B5
B13 B15 B20 B35 B44 B50 B52 B60 B62 B63 B66 B84 B85 B88
B90))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B12 B24 B29 B30
B40 B49 B58 B59 B61 B65 B74 B82 B85 B87 B91 B92))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass) (B4 B6
B8 B12 B15 B21 B25 B39 B40 B44 B46 B52 B54 B57 B70 B71
B75 B87))
(lisp)
```

D.3 Multiple Inheritance Classification Tree - Remote Access v.s. Local Access

Queries for the first experiment in Sec. 9.4.2.

```
(innet "rec-fattree-8k-3pe.img")
(loadbal 'nil)
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass) (b1-4 b1-5 b1-6 b1-7 b2-4 b2-5 b2-6
b2-7))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass
subclass- supclass) (b1-4 b1-5 b1-6 b1-7 b2-4 b2-5 b2-6
b2-7))
(lisp)
```

D.4 Multiple Inheritance Classification Tree

Queries for the second experiment in Sec. 9.4.2. The test is generated by the program in Appendix C.2.

```
(innet "rec-fattree-8k-3pe.img")
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass) (B1-21
B1-22 B1-36 B1-43 B1-48 B2-26 B2-26 B2-42))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B1-7 B1-17 B1-19 B1-30
B1-34 B1-42 B1-58 B1-60 B2-33 B2-33 B2-47 B2-49 B2-50
B2-60))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B1-26 B1-29 B1-30 B1-35
B1-46 B1-48 B1-62 B2-12 B2-21 B2-29 B2-43 B2-56 B2-63))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass) (B1-7
B1-12 B1-21 B1-43 B2-8 B2-12 B2-14 B2-16))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass) (B1-7
B1-12 B1-21 B1-28 B2-7 B2-12 B2-12 B2-16))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B1-1 B1-9 B1-22 B1-27
B1-33 B1-45 B1-51 B1-53 B1-56 B2-9 B2-19 B2-29 B2-52
B2-53 B2-57 B2-64))
(find (subclass- supclass subclass- supclass subclass-
supclass) (B1-4 B1-6 B1-12 B1-14 B1-18 B1-25 B1-26
B1-39 B1-40 B1-41 B1-42 B1-47 B1-48 B1-52 B1-53 B1-54
B1-55 B1-59 B1-63 B2-5 B2-6 B2-8 B2-16 B2-21 B2-22
B2-24 B2-33 B2-34 B2-35 B2-42 B2-46 B2-50 B2-53 B2-55
B2-56 B2-59 B2-60 B2-62))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B1-1 B1-2 B1-5
B1-8 B1-13 B2-5 B2-7 B2-8 B2-11))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B1-1 B1-3 B1-4
B1-7 B1-8 B1-9 B1-20 B2-3 B2-4 B2-7 B2-8 B2-9 B2-14))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass) (B1-1 B1-8 B1-9 B1-12 B1-13 B1-16
B1-19 B1-22 B1-25 B1-26 B1-27 B1-29 B1-30 B1-31 B1-33
```

```
B2-4 B2-7 B2-8 B2-9 B2-12 B2-13 B2-20 B2-22 B2-24 B2-33
B2-40 B2-46 B2-60))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B1-1 B1-4 B1-5
B1-7 B1-8 B1-9 B1-12 B2-6 B2-7 B2-8 B2-9 B2-12))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass) (B1-1
B1-4 B1-6 B1-7 B1-8 B2-6 B2-7 B2-8))
(find (subclass- supclass subclass- supclass subclass-
supclass) (B1-3 B1-4 B1-5 B1-12 B1-14 B1-19 B1-20 B1-26
B1-28 B1-35 B1-36 B1-40 B1-46 B1-47 B1-48 B1-49 B1-50
B1-58 B1-60 B2-6 B2-8 B2-12 B2-13 B2-14 B2-19 B2-22
B2-23 B2-24 B2-26 B2-33 B2-46 B2-47 B2-52 B2-54 B2-55
B2-62))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B1-3 B1-8 B1-12
B1-14 B1-18 B1-19 B1-21 B1-22 B1-23 B1-26 B2-9 B2-12
B2-14 B2-19 B2-22 B2-23))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass) (B1-1 B1-3 B1-11 B1-12 B1-16 B1-18
B1-21 B1-22 B1-23 B1-33 B1-34 B1-37 B1-40 B1-42 B1-44
B2-5 B2-12 B2-16 B2-20 B2-21 B2-22 B2-23 B2-25 B2-33
B2-37 B2-39 B2-40 B2-46))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass ) (B1-3 B1-12
B1-14 B1-18 B1-21 B1-22 B1-23 B1-24 B2-17 B2-21 B2-22
B2-23 B2-24 B2-25))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B1-2 B1-3 B1-4 B1-7 B1-12
B1-22 B1-23 B1-25 B1-30 B1-31 B1-33 B1-38 B1-39 B1-40
B2-7 B2-8 B2-12 B2-19 B2-22 B2-23 B2-24 B2-25 B2-30
B2-33 B2-34 B2-38 B2-43))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B1-2 B1-7 B1-8
B1-12 B1-19 B2-8 B2-12 B2-19))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass) (B1-5
B1-7 B1-9 B1-12 B1-18 B2-7 B2-9 B2-12))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass subclass- supclass) (B1-3
B1-7 B1-9 B1-12 B2-9 B2-12 b2-14 b2-30))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass) (B1-3 B1-10 B1-15 B1-17 B1-18 B1-18
```

```
B1-20 B1-42 B1-52 B1-55 B1-56 B1-58 B1-59 B1-60 B1-63
B2-15 B2-18 B2-22 B2-24 B2-31 B2-38 B2-49 B2-50 B2-52
B2-55 B2-57 B2-64))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass subclass- supclass) (B1-8 B1-9 B1-11
B1-14 B1-18 B2-8 B2-9 B2-14))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B1-2 B1-4 B1-18 B1-23
B1-25 B1-28 B1-29 B1-32 B1-37 B2-17 B2-18 B2-24 B2-25
B2-26 B2-32 B2-34))
(find (subclass- supclass subclass- supclass subclass-
supclass subclass- supclass) (B1-1 B1-2 B1-8 B1-9 B1-11
B1-18 B1-25 B1-29 B1-45 B1-49 B1-50 B2-14 B2-15 B2-18
B2-22 B2-24 B2-26 B2-34 B2-38))
(find (supclass subclass- supclass subclass- supclass
subclass- supclass) (B1-4 B1-6 B1-10 B1-18 B1-25 B1-28
B1-31 B1-36 B1-38 B1-40 B1-44 B1-46 B1-48 B1-49 B1-52
B1-56 B1-61 B2-9 B2-11 B2-14 B2-17 B2-18 B2-21 B2-27
B2-28 B2-29 B2-30 B2-33 B2-38 B2-61 B2-62))
(lisp)
```

D.5 Relational Database

Queries for the experiment in Sec. 9.4.3. The test is generated by the program in Appendix C.3.

```
(innet "db5k-2pe.sneps")
(find status ?x0)
(find status- ?x0)
(find (compose snum- jnum) ?x0)
(find city ?x0)
(find jnum ?x0 pnum ?x1)
(find weight ?x0)
(find (irreflexive-restrict city) ?x0)
(find pname- ?x0)
(find qty- ?x0)
```

```
(find weight- ?x0)
(find sname- ?x0)
(find (kplus jnum) ?x0)
(find snum ?x0)
(find (and qty color-) ?x0)
(find (kplus pnum) ?x0)
(find (compose jnum jnum- city) ?x0)
(find (compose snum- snum) ?x0)
(find (kplus status) ?x0)
(find (or pnum pnum-) ?x0)
(find (compose jnum- pname color) ?x0)
(find pnum ?x0 snum ?x1)
(find (compose jnum- snum) ?x0)
(find (and snum jnum) ?x0)
(find city- ?x0)
(find weight- ?x0)
(lisp)
```

Appendix E

Sample Runs

This sample run creates a network as illustrated in Fig. E.1. Different queries are issued to test the functionalities of TERRESA.

```
Script started on Fri Oct 09 00:25:59 1998
Fri Oct  9 00:25:59 EDT 1998
pegasus:lee-d{/u1/grads/lee-d/lee-d/src/Host}51>mpirun -np 3 terresa
Process 1 on pegasus.cs.Buffalo.EDU
Process 2 on pegasus.cs.Buffalo.EDU


    Entering TERRESA System, Fri Oct  9 00:26:07 EDT 1998


Process 0 on pegasus.cs.Buffalo.EDU
(1)* (define a b c d)
(A B C D)

  CPU time :   0.000

(2)* (assert a r b x)
(M1!)
  CPU time :   0.001

PE2: Time    0.000
PE1: Time    0.001
```

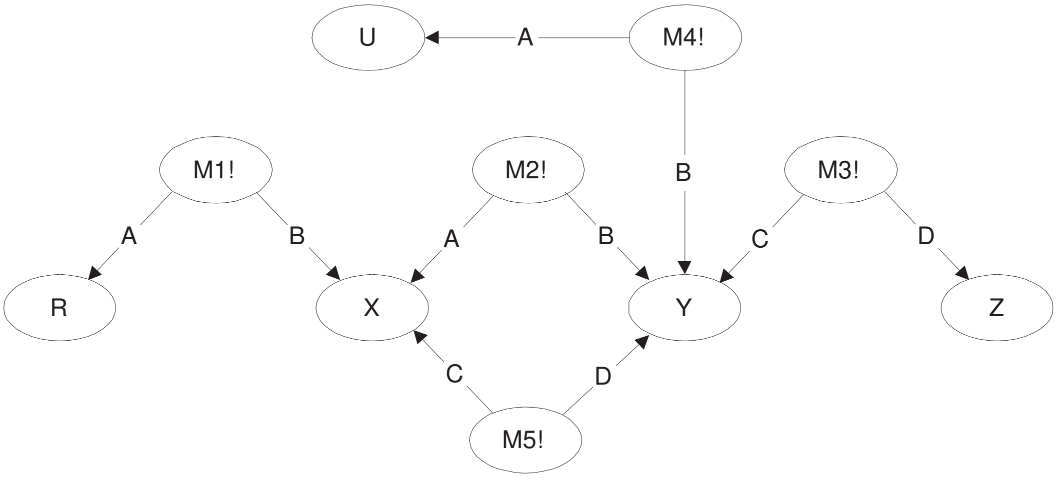

Figure E.1: Test Example of Semantic Network

```
PE2 done  PE1 done

(3)* (assert a x b y)
(M2!)
  CPU time :   0.000

PE2: Time    0.000
PE1: Time    0.001
PE2 done  PE1 done

(4)* (assert c y d z)
(M3!)
  CPU time :   0.000

PE2: Time    0.000
PE1: Time    0.000
PE1 done  PE2 done

(5)* (assert a u b y)
(M4!)
  CPU time :   0.000

PE2: Time    0.000
PE1: Time    0.001
PE2 done  PE1 done

(6)* (assert c x d y)
(M5!)
```

```
  CPU time :   0.000

PE2: Time    0.000
PE1: Time    0.000
PE2 done  PE1 done

(7)* (describe *nodes)
(R)
(X)
(Y)
(Z)
(U)
(M1! (A R) (B X))
(M2! (A X) (B Y))
(M3! (C Y) (D Z))
(M4! (A U) (B Y))
(M5! (C X) (D Y))
(R U X Y Z M1! M2! M3! M4! M5!)
  CPU time :   0.002

(8)* (dump *nodes)
(R  (A- (M1!)))
(U  (A- (M4!)))
(X  (B- (M1!)) (A- (M2!)) (C- (M5!)))
(Y  (B- (M2! M4!)) (C- (M3!)) (D- (M5!)))
(Z  (D- (M3!)))
(M1 ! (A (R)) (B (X)))
(M2 ! (A (X)) (B (Y)))
(M3 ! (C (Y)) (D (Z)))
(M4 ! (A (U)) (B (Y)))
(M5 ! (C (X)) (D (Y)))
(R U X Y Z M1! M2! M3! M4! M5!)
  CPU time :   0.002

(9)* (find a x b y)
Avg Fin Time =    0.050, Tot Private = 1, Tot Cache = 0, Tot
Remote = 1, HRatio =    0.000
(M2!)
  CPU time :   0.153

PE2 done  PE1 done

(10)* (find (and (compose a- b) (compose c- d)) y)
```

```
Avg Fin Time =    0.019, Tot Private = 8, Tot Cache = 0, Tot
Remote = 0, HRatio =      NaN
(X)
  CPU time :   0.078

PE2 done  PE1 done

(11)* (find (or (compose a- b) c) y)
Avg Fin Time =    0.012, Tot Private = 6, Tot Cache = 0, Tot
Remote = 0, HRatio =      NaN
(U X M3!)
  CPU time :   0.094

PE2 done  PE1 done

(12)* (find (kplus (compose a- b)) y)
Avg Fin Time =    0.014, Tot Private = 12, Tot Cache = 0, Tot
Remote = 0, HRatio =      NaN
(R U X)
  CPU time :   0.094

PE2 done  PE1 done

(13)* (find (kstar (compose a- b)) y)
Avg Fin Time =    0.019, Tot Private = 12, Tot Cache = 0, Tot
Remote = 0, HRatio =      NaN
(R U X Y)
  CPU time :   0.078

PE2 done  PE1 done

(14)* (find (relative-complement (compose a- b) (compose c- d)) y)
Avg Fin Time =    0.013, Tot Private = 8, Tot Cache = 0, Tot
Remote = 0, HRatio =      NaN
(U)
  CPU time :   0.094

PE2 done  PE1 done

(15)* (find (irreflexive-restrict (compose d- c a- b)) y)
Avg Fin Time =    0.017, Tot Private = 8, Tot Cache = 0, Tot
Remote = 0, HRatio =      NaN
```

```
  CPU time :   0.094

PE2 done  PE1 done

(16)* (find (irreflexive-restrict (compose c a- b)) y)
Avg Fin Time =    0.016, Tot Private = 7, Tot Cache = 0, Tot
Remote = 0, HRatio =      NaN
(M5!)
  CPU time :   0.094

PE2 done  PE1 done

(17)* (find (domain-restrict ((compose c- d) y) (compose a- b)) y)
Avg Fin Time =    0.018, Tot Private = 8, Tot Cache = 0, Tot
Remote = 0, HRatio =      NaN
(X)
  CPU time :   0.094

PE2 done  PE1 done

(18)* (find (range-restrict (compose a- b) ((compose c- d) z)) y)
Avg Fin Time =    0.019, Tot Private = 8, Tot Cache = 0, Tot
Remote = 0, HRatio =      NaN
(U X)
  CPU time :   0.095

PE2 done  PE1 done

(19)* (outnet "exper.sneps")

  CPU time :   0.134

PE1: Time    0.209
PE2: Time    0.226
PE1 done  PE2 done

(20)* (resetnet 't)

  CPU time :   0.126

PE1: Time    0.345
PE2: Time    0.345
PE1 done  PE2 done
```

```
(21)* *nodes

  CPU time :   0.000

(22)* (innet "exper.sneps")
Importing misc infos ...
Importing relations ...
Importing symbol tables ...
PE2, Num of SNePS node=0
VARIABLES PE2: Time    0.004
VARNODES COMMANDS ASSERTIONS RELATIONS PE1, Num of SNePS node=10
PE1: Time    0.005
PATTERNS NODES
Importing host nets ...

Importing node maps ...

  CPU time :   0.135

PE2 done  PE1 done

(23)* *nodes
(R U X Y Z M1! M2! M3! M4! M5!)
  CPU time :   0.000

(24)* (define aa bb cc)
(AA BB CC)

  CPU time :   0.000

(25)* *relations
(&ANT A AA ACT ACTION ANT ARG B BB C CC CONDITION CQ D DCQ DEFAULT
DO EFFECT ELSE EMAX EMIN ETOT EXISTS FORALL GOAL IF MAX MIN
OBJECT1 OBJECT2 PEVB PLAN PRECONDITION SUCHTHAT THEN THRESH
THRESHMAX VARS WHEN)

  CPU time :   0.001

(26)* (undefine bb)
(BB)

  CPU time :   0.000
```

```
(27)* *relations
(&ANT A AA ACT ACTION ANT ARG B C CC CONDITION CQ D DCQ DEFAULT
DO EFFECT ELSE EMAX EMIN ETOT EXISTS FORALL GOAL IF MAX MIN
OBJECT1 OBJECT2 PEVB PLAN PRECONDITION SUCHTHAT THEN THRESH
THRESHMAX VARS WHEN)

  CPU time :   0.001

(28)* (private 1)

  CPU time :   0.000

PE2: Time    0.000
PE1 index : 9
Addr : <<idx = 9>> [1 5]
Dirty bit : FALSE, clean_holder : 1, data holder : (001) [1 5]
Assertion : (+!)
Forward link : (C ([0 2])) (D ([0 3]))
Reverse link : (C- ([0 2])) (D- ([0 3]))
-------------------------------------------
PE1 index : 8
Addr : <<idx = 8>> [1 4]
Dirty bit : FALSE, clean_holder : 1, data holder : (001) [1 4]
Assertion : (+!)
Forward link : (A ([0 5])) (B ([0 3]))
Reverse link : (A- ([0 5])) (B- ([0 3]))
-------------------------------------------
PE1 index : 7
Addr : <<idx = 7>> [0 5]
Dirty bit : FALSE, clean_holder : 1, data holder : (001) [0 5]
Assertion : (-!)
Forward link : (A- ([1 4]))
Reverse link : (A ([1 4]))
-------------------------------------------
PE1 index : 6
Addr : <<idx = 6>> [1 3]
Dirty bit : FALSE, clean_holder : 1, data holder : (001) [1 3]
Assertion : (+!)
Forward link : (C ([0 3])) (D ([0 4]))
Reverse link : (C- ([0 3])) (D- ([0 4]))
-------------------------------------------
PE1 index : 5
```

```
Addr : <<idx = 5>> [0 4]
Dirty bit : FALSE, clean_holder : 1, data holder : (001) [0 4]
Assertion : (-!)
Forward link : (D- ([1 3]))
Reverse link : (D ([1 3]))
-------------------------------------------
PE1 index : 4
Addr : <<idx = 4>> [1 2]
Dirty bit : FALSE, clean_holder : 1, data holder : (001) [1 2]
Assertion : (+!)
Forward link : (A ([0 2])) (B ([0 3]))
Reverse link : (A- ([0 2])) (B- ([0 3]))
-------------------------------------------
PE1 index : 3
Addr : <<idx = 3>> [0 3]
Dirty bit : FALSE, clean_holder : 1, data holder : (001) [0 3]
Assertion : (-!)
Forward link : (B- ([1 2] [1 4])) (C- ([1 3])) (D- ([1 5]))
Reverse link : (B ([1 2] [1 4])) (C ([1 3])) (D ([1 5]))
-------------------------------------------
PE1 index : 2
Addr : <<idx = 2>> [1 1]
Dirty bit : FALSE, clean_holder : 1, data holder : (001) [1 1]
Assertion : (+!)
Forward link : (A ([0 1])) (B ([0 2]))
Reverse link : (A- ([0 1])) (B- ([0 2]))
-------------------------------------------
PE1 index : 1
Addr : <<idx = 1>> [0 2]
Dirty bit : FALSE, clean_holder : 1, data holder : (001) [0 2]
Assertion : (-!)
Forward link : (A- ([1 2])) (B- ([1 1])) (C- ([1 5]))
Reverse link : (A ([1 2])) (B ([1 1])) (C ([1 5]))
-------------------------------------------
PE1 index : 0
Addr : <<idx = 0>> [0 1]
Dirty bit : FALSE, clean_holder : 1, data holder : (001) [0 1]
Assertion : (-!)
Forward link : (A- ([1 1]))
Reverse link : (A ([1 1]))
-------------------------------------------
PE1: Time    0.008
PE2 done  PE1 done
```

```
(29)* (cache 1)

  CPU time :   0.000

PE2: Time    0.000
PE1: Time    0.081
PE2 done  PE1 done

(30)* (cache 2)

  CPU time :   0.000

PE2 index : 0
PE1: Time    0.000
Addr : NULL
Dirty bit : TRUE Queue_idx(for swapping out):0 owner : 1,
[UNKNOWN,UNKNOWN] Assertion : (-!)
Forward link :
Reverse link :

-------------------------------------------
PE2 index : 1
Addr : NULL
Dirty bit : TRUE Queue_idx(for swapping out):0 owner : 1,
[UNKNOWN,UNKNOWN] Assertion : (-!)
Forward link :
Reverse link :

-------------------------------------------
PE2 index : 2
Addr : NULL
Dirty bit : TRUE Queue_idx(for swapping out):0 owner : 1,
[UNKNOWN,UNKNOWN] Assertion : (-!)
Forward link :
Reverse link :

-------------------------------------------
PE2 index : 3
Addr : NULL
Dirty bit : TRUE Queue_idx(for swapping out):0 owner : 1,
[UNKNOWN,UNKNOWN] Assertion : (-!)
Forward link :
```

```
Reverse link :

-------------------------------------------
PE2 index : 4
Addr : NULL
Dirty bit : TRUE Queue_idx(for swapping out):0 owner : 1,
[UNKNOWN,UNKNOWN] Assertion : (-!)
Forward link :
Reverse link :

-------------------------------------------
PE2 index : 5
Addr : NULL
Dirty bit : TRUE Queue_idx(for swapping out):0 owner : 1,
[UNKNOWN,UNKNOWN] Assertion : (-!)
Forward link :
Reverse link :

-------------------------------------------
PE2 index : 6
Addr : NULL
Dirty bit : TRUE Queue_idx(for swapping out):0 owner : 1,
[UNKNOWN,UNKNOWN] Assertion : (-!)
Forward link :
Reverse link :

-------------------------------------------
PE2 index : 7
Addr : NULL
Dirty bit : TRUE Queue_idx(for swapping out):0 owner : 1,
[UNKNOWN,UNKNOWN] Assertion : (-!)
Forward link :
Reverse link :

-------------------------------------------
PE2 index : 8
Addr : NULL
Dirty bit : TRUE Queue_idx(for swapping out):0 owner : 1,
[UNKNOWN,UNKNOWN] Assertion : (-!)
Forward link :
Reverse link :

-------------------------------------------
```

```
PE2 index : 9
Addr : NULL
Dirty bit : TRUE Queue_idx(for swapping out):0 owner : 1,
[UNKNOWN,UNKNOWN] Assertion : (-!)
Forward link :
Reverse link :

-------------------------------------------
PE2: Time    0.088
PE1 done  PE2 done

(31)* (= (x y z) newnodes)
(X Y Z)
  CPU time :   0.000

(32)* *newnodes
(X Y Z)
  CPU time :   0.000

(33)* (lisp)
Host:End of SNePS

  CPU time :   0.000

PE1: Time    0.000
PE2: Time    0.000
PE1:Private Hit = 78, memory fault (cache request) = 0,
Cache Hit = 0, Hit Ratio =      NaN, Remote Found=0 dupchk =0
PE1 done  PE2 done

PE2:Private Hit = 0, memory fault (cache request) = 1,
Cache Hit = 0, Hit Ratio =    0.000, Remote Found=0 dupchk =0
-----------------------------------------------------
TERRESA Total Running Time = 0 hr 10 min   59.225 sec
-----------------------------------------------------
pegasus:lee-d{/u1/grads/lee-d/lee-d/src/Host}52>exit
exit

script done on Fri Oct 09 00:37:14 1998
```

Appendix F

Sample Runs - Database Example

This sample run tests the database example similar to the one in SNePS demonstration.

Because of the wildcard query functionalities, not all the queries in the SNePS are tested by TERRESA.

```
Script started on Thu Dec 24 15:02:34 1998
Thu Dec 24 15:02:35 EST 1998
pegasus:lee-d{/u1/grads/lee-d/lee-d/src/Host}51>mpirun -np 3 terresa

Process 2 on pegasus.cs.Buffalo.EDU
Process 1 on pegasus.cs.Buffalo.EDU


  Entering TERRESA System, Thu Dec 24 15:02:51 EST 1998


Process 0 on pegasus.cs.Buffalo.EDU
(1)* (innet "dbmsdemo")
Importing misc infos ...
Importing relations ...
Importing symbol tables ...
VARIABLES VARNODES COMMANDS ASSERTIONS RELATIONS PATTERNS NODES
PE2, Num of TERRESA node=0
PE2: Time  0.006
```

```
Importing host nets ...
PE1, Num of TERRESA node=102
PE1: Time  0.036

Importing node maps ...

 CPU time :  0.174

PE2 done PE1 done

(2)* (dump *nodes)
(S1 (SNUM- (M1! M19! M20!)))
(SMITH (SNAME- (M1!)))
(20 (STATUS- (M1! M4!)))
(LONDON (CITY- (M1! M4! M6! M9! M11! M16! M18!)))
(S2 (SNUM- (M2! M21! M22! M23! M24! M25! M26! M27! M28!)))
(JONES (SNAME- (M2!)))
(10 (STATUS- (M2!)))
(PARIS (CITY- (M2! M3! M7! M10! M12!)))
(S3 (SNUM- (M3! M29! M30!)))
(BLAKE (SNAME- (M3!)))
(30 (STATUS- (M3! M5!)))
(S4 (SNUM- (M4! M31! M32!)))
(CLARK (SNAME- (M4!)))
(S5 (SNUM- (M5! M33! M34! M35! M36! M37! M38! M39! M40! M41! M42!)))
(ADAMS (SNAME- (M5!)))
(ATHENS (CITY- (M5! M14! M15!)))
(P1 (PNUM- (M6! M19! M20! M38!)))
(NUT (PNAME- (M6!)))
(RED (COLOR- (M6! M9! M11!)))
(12 (WEIGHT- (M6! M10!)))
(P2 (PNUM- (M7! M33! M34!)))
(BOLT (PNAME- (M7!)))
(GREEN (COLOR- (M7!)))
(17 (WEIGHT- (M7! M8!)))
(P3 (PNUM- (M8! M21! M22! M23! M24! M25! M26! M27! M29! M39!)))
(SCREW (PNAME- (M8! M9!)))
(BLUE (COLOR- (M8! M10!)))
(ROME (CITY- (M8! M13!)))
(P4 (PNUM- (M9! M30! M40!)))
(14 (WEIGHT- (M9!)))
(P5 (PNUM- (M10! M28! M35! M36! M41!)))
```

```
(CAM (PNAME- (M10!)))
(P6 (PNUM- (M11! M31! M32! M37! M42!)))
(COG (PNAME- (M11!)))
(19 (WEIGHT- (M11!)))
(J1 (JNUM- (M12! M19! M21! M29!)))
(SORTER (JNAME- (M12!)))
(J2 (JNUM- (M13! M22! M28! M30! M33! M37!)))
(PUNCH (JNAME- (M13!)))
(J3 (JNUM- (M14! M23! M31!)))
(READER (JNAME- (M14!)))
(J4 (JNUM- (M15! M20! M24! M34! M38! M39! M40! M41! M42!)))
(CONSOLE (JNAME- (M15!)))
(J5 (JNUM- (M16! M25! M35!)))
(COLLATOR (JNAME- (M16!)))
(J6 (JNUM- (M17! M26!)))
(TERMINAL (JNAME- (M17!)))
(OSLO (CITY- (M17!)))
(J7 (JNUM- (M18! M27! M32! M36!)))
(TAPE (JNAME- (M18!)))
(200 (QTY- (M19! M22! M23! M29! M33! M37!)))
(700 (QTY- (M20!)))
(400 (QTY- (M21! M26! M41!)))
(500 (QTY- (M24! M30! M35! M42!)))
(600 (QTY- (M25!)))
(800 (QTY- (M27! M40!)))
(100 (QTY- (M28! M34! M36!)))
(300 (QTY- (M31! M32!)))
(1000 (QTY- (M38!)))
(1200 (QTY- (M39!)))
(M1! (SNUM (S1)) (SNAME (SMITH)) (STATUS (20)) (CITY (LONDON)))
(M2! (SNUM (S2)) (SNAME (JONES)) (STATUS (10)) (CITY (PARIS)))
(M3! (SNUM (S3)) (SNAME (BLAKE)) (STATUS (30)) (CITY (PARIS)))
(M4! (SNUM (S4)) (SNAME (CLARK)) (STATUS (20)) (CITY (LONDON)))
(M5! (SNUM (S5)) (SNAME (ADAMS)) (STATUS (30)) (CITY (ATHENS)))
(M6! (PNUM (P1)) (PNAME (NUT)) (COLOR (RED)) (WEIGHT (12))
(CITY (LONDON)))
(M7! (PNUM (P2)) (PNAME (BOLT)) (COLOR (GREEN)) (WEIGHT (17))
(CITY (PARIS)))
(M8! (PNUM (P3)) (PNAME (SCREW)) (COLOR (BLUE)) (WEIGHT (17))
(CITY (ROME)))
(M9! (PNUM (P4)) (PNAME (SCREW)) (COLOR (RED)) (WEIGHT (14))
(CITY (LONDON)))
(M10! (PNUM (P5)) (PNAME (CAM)) (COLOR (BLUE)) (WEIGHT (12))
```

```
(CITY (PARIS)))
(M11! (PNUM (P6)) (PNAME (COG)) (COLOR (RED)) (WEIGHT (19))
(CITY (LONDON)))
(M12! (JNUM (J1)) (JNAME (SORTER)) (CITY (PARIS)))
(M13! (JNUM (J2)) (JNAME (PUNCH)) (CITY (ROME)))
(M14! (JNUM (J3)) (JNAME (READER)) (CITY (ATHENS)))
(M15! (JNUM (J4)) (JNAME (CONSOLE)) (CITY (ATHENS)))
(M16! (JNUM (J5)) (JNAME (COLLATOR)) (CITY (LONDON)))
(M17! (JNUM (J6)) (JNAME (TERMINAL)) (CITY (OSLO)))
(M18! (JNUM (J7)) (JNAME (TAPE)) (CITY (LONDON)))
(M19! (SNUM (S1)) (PNUM (P1)) (JNUM (J1)) (QTY (200)))
(M20! (SNUM (S1)) (PNUM (P1)) (JNUM (J4)) (QTY (700)))
(M21! (SNUM (S2)) (PNUM (P3)) (JNUM (J1)) (QTY (400)))
(M22! (SNUM (S2)) (PNUM (P3)) (JNUM (J2)) (QTY (200)))
(M23! (SNUM (S2)) (PNUM (P3)) (JNUM (J3)) (QTY (200)))
(M24! (SNUM (S2)) (PNUM (P3)) (JNUM (J4)) (QTY (500)))
(M25! (SNUM (S2)) (PNUM (P3)) (JNUM (J5)) (QTY (600)))
(M26! (SNUM (S2)) (PNUM (P3)) (JNUM (J6)) (QTY (400)))
(M27! (SNUM (S2)) (PNUM (P3)) (JNUM (J7)) (QTY (800)))
(M28! (SNUM (S2)) (PNUM (P5)) (JNUM (J2)) (QTY (100)))
(M29! (SNUM (S3)) (PNUM (P3)) (JNUM (J1)) (QTY (200)))
(M30! (SNUM (S3)) (PNUM (P4)) (JNUM (J2)) (QTY (500)))
(M31! (SNUM (S4)) (PNUM (P6)) (JNUM (J3)) (QTY (300)))
(M32! (SNUM (S4)) (PNUM (P6)) (JNUM (J7)) (QTY (300)))
(M33! (SNUM (S5)) (PNUM (P2)) (JNUM (J2)) (QTY (200)))
(M34! (SNUM (S5)) (PNUM (P2)) (JNUM (J4)) (QTY (100)))
(M35! (SNUM (S5)) (PNUM (P5)) (JNUM (J5)) (QTY (500)))
(M36! (SNUM (S5)) (PNUM (P5)) (JNUM (J7)) (QTY (100)))
(M37! (SNUM (S5)) (PNUM (P6)) (JNUM (J2)) (QTY (200)))
(M38! (SNUM (S5)) (PNUM (P1)) (JNUM (J4)) (QTY (1000)))
(M39! (SNUM (S5)) (PNUM (P3)) (JNUM (J4)) (QTY (1200)))
(M40! (SNUM (S5)) (PNUM (P4)) (JNUM (J4)) (QTY (800)))
(M41! (SNUM (S5)) (PNUM (P5)) (JNUM (J4)) (QTY (400)))
(M42! (SNUM (S5)) (PNUM (P6)) (JNUM (J4)) (QTY (500)))
(10 12 14 17 19 20 30 J1 J2 J3 J4 J5 J6 J7 M1! M2! M3! M4!
M5! M6! M7! M8! M9! P1 P2 P3 P4 P5 P6 S1 S2 S3 S4 S5 100
200 300 400 500 600 700 800 CAM COG M10! M11! M12! M13! M14!
M15! M16! M17! M18! M19! M20! M21! M22! M23! M24! M25! M26!
M27! M28! M29! M30! M31! M32! M33! M34! M35! M36! M37! M38!
M39! M40! M41! M42! NUT RED 1000 1200 BLUE BOLT OSLO ROME
TAPE ADAMS BLAKE CLARK GREEN JONES PARIS PUNCH SCREW SMITH
ATHENS LONDON READER SORTER CONSOLE COLLATOR TERMINAL)
 CPU time : 0.071
```

```
(3)* (dump (find jname ?x))

Avg Fin Time = 0.020, Tot Private = 102, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.178

(M12! (JNUM (J1)) (JNAME (SORTER)) (CITY (PARIS)))
(M13! (JNUM (J2)) (JNAME (PUNCH)) (CITY (ROME)))
(M14! (JNUM (J3)) (JNAME (READER)) (CITY (ATHENS)))
(M15! (JNUM (J4)) (JNAME (CONSOLE)) (CITY (ATHENS)))
(M16! (JNUM (J5)) (JNAME (COLLATOR)) (CITY (LONDON)))
(M17! (JNUM (J6)) (JNAME (TERMINAL)) (CITY (OSLO)))
(M18! (JNUM (J7)) (JNAME (TAPE)) (CITY (LONDON)))
(M12! M13! M14! M15! M16! M17! M18!)

(4)* (dump (find jname ?x city london))

Avg Fin Time = 0.013, Tot Private = 103, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.178

(M16! (JNUM (J5)) (JNAME (COLLATOR)) (CITY (LONDON)))
(M18! (JNUM (J7)) (JNAME (TAPE)) (CITY (LONDON)))
(M16! M18!)

(5)* (find (snum- jnum) j1)
Avg Fin Time = 0.012, Tot Private = 6, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.098

(S1 S2 S3)

PE2 done PE1 done

(6)* (find snum- (find jnum j1 pnum p1))

 CPU time : 0.158

Avg Fin Time = 0.071, Tot Private = 1, Tot Cache = 0,
```

```
Tot Remote = 1, HRatio = 0.000
Avg Fin Time = 0.017, Tot Private = 1, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.084

(S1)

PE2 done PE1 done

(7)* (find (jname- jnum jnum- snum) s1)
Avg Fin Time = 0.015, Tot Private = 20, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.098

(SORTER CONSOLE)

PE2 done PE1 done

(8)* (find (color- pnum pnum- snum) s1)
Avg Fin Time = 0.016, Tot Private = 15, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.098

(RED)

PE2 done PE1 done

(9)* (find (snum- jnum) j1 (snum- jnum) j2)
Avg Fin Time = 0.021, Tot Private = 6, Tot Cache = 8,
Tot Remote = 0, HRatio = 1.000

 CPU time : 0.160

(S2 S3)

PE2 done PE1 done

(10)* (find snum- (find jnum j1 (pnum pnum- color) red))
Avg Fin Time = 0.013, Tot Private = 1, Tot Cache = 8,
Tot Remote = 0, HRatio = 1.000
```

```
 CPU time : 0.158

Avg Fin Time = 0.013, Tot Private = 1, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.081

(S1)

PE2 done PE1 done

(11)* (find (pnum- jnum jnum- city) london)
Avg Fin Time = 0.013, Tot Private = 18, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.082

(P3 P5 P6)

PE2 done PE1 done

(12)* (find snum- (find (jnum jnum- city) (london paris)
      (pnum pnum- color) red))
Avg Fin Time = 0.020, Tot Private = 19, Tot Cache = 8,
Tot Remote = 0, HRatio = 1.000

 CPU time : 0.160

Avg Fin Time = 0.020, Tot Private = 1, Tot Cache = 1,
Tot Remote = 0, HRatio = 1.000

 CPU time : 0.158

(S1 S4)

PE2 done PE1 done

(13)* (find pnum- (find (jnum jnum- city) london
      (snum snum- city) london))

 CPU time : 0.156
```

```
Avg Fin Time = 0.154, Tot Private = 11, Tot Cache = 11,
Tot Remote = 0, HRatio = 1.000

 CPU time : 0.117

(P6)

PE2 done PE1 done
Avg Fin Time = 0.019, Tot Private = 1, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

(14)* (- (find jnum- ?x)
    (find jnum- (find (pnum pnum- color) red
         (snum snum- city) london)))
Avg Fin Time = 0.015, Tot Private = 102, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.177


Avg Fin Time = 0.021, Tot Private = 8, Tot Cache = 11,
Tot Remote = 0, HRatio = 1.000

 CPU time : 0.159

Avg Fin Time = 0.013, Tot Private = 2, Tot Cache = 2,
Tot Remote = 0, HRatio = 1.000

 CPU time : 0.156

(J2 J5 J6)

(15)* (find (snum- pnum pnum- snum snum- pnum pnum- color) red)
Avg Fin Time = 0.061, Tot Private = 379, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.099

(S1 S2 S3 S4 S5)

PE2 done PE1 done

(16)* (find (jnum- pnum pnum- snum) s1)
```

```
Avg Fin Time = 0.043, Tot Private = 15, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.080

(J1 J4)

PE2 done PE1 done

(17)* (find jnum- ?x)

 CPU time : 0.181

(J1 J2 J3 J4 J5 J6 J7)

PE2 done PE1 done
Avg Fin Time = 0.163, Tot Private = 102, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

(18)* (find (jnum- city) london)

 CPU time : 0.079

(J5 J7)

PE2 done PE1 done
Avg Fin Time = 0.181, Tot Private = 9, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

(19)* (find (pnum- jnum) j5 (pnum- jnum) j7)
Avg Fin Time = 0.021, Tot Private = 5, Tot Cache = 6,
Tot Remote = 0, HRatio = 1.000

 CPU time : 0.163

(P3 P5)
PE2 done PE1 done

(20)* (find (pnum- snum) s1)
Avg Fin Time = 0.044, Tot Private = 5, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.080
```

```
(P1)

PE2 done PE1 done

(21)* (find (jnum- pnum) p1)
Avg Fin Time = 0.012, Tot Private = 6, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.080

(J1 J4)
PE2 done PE1 done

(22)* (find (pnum- snum) s1)
Avg Fin Time = 0.014, Tot Private = 5, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.100

(P1)

PE2 done PE1 done

(23)* (find jnum- (find snum s1 pnum p1))
Avg Fin Time = 0.013, Tot Private = 1, Tot Cache = 1,
Tot Remote = 0, HRatio = 1.000

 CPU time : 0.163

Avg Fin Time = 0.020, Tot Private = 1, Tot Cache = 1,
Tot Remote = 0, HRatio = 1.000

 CPU time : 0.163

(J1 J4)PE2 done PE1 done

(24)* (find (snum- pnum pnum- color) red)
Avg Fin Time = 0.015, Tot Private = 20, Tot Cache = 0,
Tot Remote = 0, HRatio =  NaN

 CPU time : 0.099
```

```
(S1 S3 S4 S5)

PE2 done PE1 done

(25)* (find (jnum- snum) s1 (jnum- snum) s3
   (jnum- snum) s4 (jnum- snum) s5)
Avg Fin Time = 0.017, Tot Private = 10, Tot Cache = 18,
Tot Remote = 0, HRatio = 1.000

 CPU time : 0.156

PE2 done PE1 done

(26)* (lisp)
Host:End of TERRESA
PE1: Time 0.000

 CPU time : 0.000

PE2 done PE1 done

PE2: Time 0.000
PE1:Private Hit = 976, memory fault (cache request) = 0,
Cache Hit = 0, Hit Ratio =  NaN, Remote Found=0 dupchk =0
-----------------------------------------------------
TERRESA Total Running Time = 0 hr 3 min 33.202 sec
-----------------------------------------------------
PE2:Private Hit = 0, memory fault (cache request) = 1,
Cache Hit = 75, Hit Ratio = 0.987, Remote Found=0 dupchk =0

pegasus:lee-d{/u1/grads/lee-d/lee-d/src/Host}52>exit
exit

script done on Thu Dec 24 15:06:27 1998
```

Bibliography

[1] ALASDIAR, R., BRUCE, A., MILLER, J., AND SMITH, G. *CHIMP/MPI User Guide*. Edinburgh Parallel Computing Centre, Edinburgh, UK, 1994.

[2] ALVERSON, G., BRIGGS, P., COATNEY, S., KAHAN, S., AND KORRY, R. Tera Hardware-Software Cooperation. In *Proceedings of Supercomputing 1997* (San Hose, CA, Nov. 1997), ACM/IEEE.

[3] BBN ADVANCED COMPUTERS INC. *TC2000 Technical Production Summary*. Cambridge, MA, Nov. 1989.

[4] BHATT, S., GREENBERG, D., LEIGHTON, T., AND LIU, P. Efficient embeddings of trees in hypercubes. *SIAM Journal on Computing* (1991).

[5] BRACHMAN, R. J., AND SCHMOLZE, J. G. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Sci. 9* (1985), 171–216.

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

[7] CHAN, C. M. Forward Path-Based Inference in SNePS. Tech. Rep. SNeRG Technical Note No. 17, Department of Computer Science at SUNY Buffalo, Feb. 1987.

[8] CHUNG, M., AND MOLDOVAN, D. Parallel Memory-Based Parsing on SNAP. In *7th International Parallel Processing Symposium* (Apr. 1993), pp. 680–684.

[9] CHUNG, S., AND MOLDOVAN, D. I. Modeling Semantic Networks on the Connection Machine. *Journal of Parallel and Distributed Computing 17* (1993), 152–163.

[10] CLARK, K., AND GREGORY, S. Parlog: parallel programming in logic. *ACM Transactions on Programming Languages and Systems 8*, 1 (Jan. 1986), 1–49.

[11] CRAY RESERACH INC.

. *Cray T3E Architecture Overview*. http://www.cray.com/products/systems/crayt3e/.

[12] CYBENKO, G. Dynamic Load Balancing for Distributed Memory Multiprocessors. *J. of Parallel Distrib. Computing 7* (1989), 279–301.

[13] DATE, C. J. *An Introduction to Database System.* Addison-Wesley Publishing Company, 1986.

[14] DEMARA, R. F., AND MOLDOVAN, D. I. The SNAP-1 Parallel AI Prototype. *IEEE Trans. on Parallel and Distributed Systems 4*, 8 (Aug. 1993), 841–854.

[15] EVETT, M. P., HENDLER, J. A., AND SPECTOR, L. Parallel Knowledge Representation on the Connection Machine. *Journal of Parallel and Distributed Computing 22* (1991), 168–184.

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

[17] FERREIRA, C. P., MAMEDE, N. J., AND MARTINS, J. P. SNIP 2.1, The SNePS Inference Package. Tech. rep., Instituto Superior Técnico, Av. Rovisco Pais, 1000 Lisboa, Portugal, Jun. 1989.

[18] FLYNN, M. Some Computer Organization and Their Effectiveness. *IEEE Trans. on Computers 21*, 9 (1972), 948–960.

[19] FOSTER, I., AND TAYLOR, S. *Strand: New Concepts in Parallel Programming*. Prentice Hall, 1990.

[20] FURSE, E., AND SEWELL, K. H. *Parallel Processing for Atrtificial Intelligence*, vol. 2. North-Holland, 1994, ch. 7, Automatic Parallelisation of LISP programs.

[21] GABRIEL, R., AND MCCARTHY, J. Qlisp. In *Parallel Computation and Computers for Artificial Intelligence*, J. Kowalik, Ed. Kluwer, Boston, 1988.

[22] GAREY, M. R., JOHNSON, D. J., AND STOCKMEYER, L. Some Simplified NP-Complete Graph Problems. *Theoret. Comput. Sci. 1* (1976), 237–267.

[23] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., AND SUNDERAM, V. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[24] GREENBERG, D., HEATH, L., AND ROSENBERG, A. Optimal Embeddings of Butterfly-like Graphs in the Hypercube. *Mathematical Systems Theory 23*, 1 (1990), 61–77.

[25] GROPP, W., LUSK, E., DOSS, N., AND SKJELLUM, A. A High-Performance, Portable Implementation of MPI Message-Passing Interface Standard. Tech. rep., Argonne National Lab., 1996.

[26] GROPP, W., LUSK, E., DOSS, N., AND SKJELLUM, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing 22*, 6 (Sept. 1996), 789–828.

[27] GROPP, W., LUSK, E., AND SKJELLUM, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, MA, 1994.

[28] HAVEL, I., AND LIEBL, P. Embedding the Polytomic Tree into the n-cube. *Časopis pro Pěstovan ́ı matematiky 98* (1973), 304–317.

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

[30] HIGUCHI, T. S., KITANO, H., FURYUA, T., KUSUMOTO, H., HANDA, K., AND KOKUBU, A. IXM2: A parallel associate processor for knowledge processing. In *Proceedings of the National Conference on Artificial Intelligence* (1991).

[31] HIGUCHIM, T. S., FURUYA, T., AND KUSUMOTO, H. The Prototype of Semantic Network Machine IXM. In *Proceeding 1989 Int'l Conf. Parallel Processing* (1989), vol. 1, IEEE, pp. 217–224.

[32] HILLS, W. D. *The Connection Machine*. MIT Press, 1985.

[33] HSU, F. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1989.

[34] HSU, F. H., CAMPBELL, M. S., AND HOANE, A. J. J. Deep Blue System Overview. In *Proceedings of the 9th ACM International Conference on Supercomputing* (1995), ACM, pp. 240–244.

[35] JR., R. H. Parallel Computing using Multilisp. In *Parallel Computation and Computers for Artificial Intelligence*, J. Kowalik, Ed. Kluwer, Boston, 1988.

[36] KAUFFMANN, W. J., AND SMARR, L. L. *Supercomputing and the Transformation of Science*. Scientific American Library, 1993.

[37] KITANO, H., AND HENDLER, J. A. *Massively Parallel Artificial Intelligence*. MIT Press, Cambridge, MA, 1994.

[38] LAZOWSKA, E., AND SQUILLANTE, M. Using Processor-Cache Affinity in Shared-Memory Multiprocessor Scheduling. Tech. rep., Uinv. of Wa., 1989.

[39] LEANT, D. CYC: A Large-Scale Investment in Knowledge Infrastructure. *Communication of ACM 38*, 11 (Nov. 1995), 34–38.

[40] LEE, C.-W., AND WU, M.-Y. Parallelization of SNePS : A Parallel Semantic Network Algorithm. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing* (Minneapolis, MN, 1997).

[41] LEE, C.-W., AND WU, M.-Y. PSNePS : A Parallel Semantic Network. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (Las Vegas, NV, 1998).

[42] LEE, K.-F., AND HON, H. W. Large-Vocabulary Speaker-Independent Continuous Speech Recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing* (1988).

[43] LI, L., AND WANG, P. *The CL-PVM Package*. Kent State University, 1995.

[44] MASPAR COMPUTER CORPORATION. *The MasPar Family Data-Parallel Computer*. Sunnyvale, CA, 1991.

[45] MESSAGE PASSING INFTERFACE FORUM. *MPI: A Message-Passing Interface Standard*, Jun. 1995.

[46] MOLDOVAN, D., LEE, W., LIN, C., AND CHUNG, M. SNAP, Parallel Processing Applied to AI. *IEEE Computer* (May 1992), 39–49.

[47] MOORE'S LAW. http://www.intel.com/intel/museum/25anniv/hof/moore.htm.

[48] OHIO SUPERCOMPUTER CENTER. *MPI Primer/Developing with LAM*. The Ohio State University, Nov. 1995.

[49] PROTOPOPOV, B. MPI Implementation for Window NT. Tech. rep., Mississippi State University, 1996.

[50] QUILLIAN, R. Semantic Memory. In *Semantic Information Processing*, M. Minsky, Ed. MIT Press, 1968.

[51] QUILLIAN, R. The Teachable Language Comprehender: A Simulation Program and Theory of Language. *Comm. ACM* (Aug. 1969).

[52] SCHWABE, E. *Efficient Embeddings and Simulations for Hypercube Networks*. PhD thesis, MIT, Jun. 1991.

[53] SHAPIRO, E. *Concurrent Prolog: Collected Papers*. Logic Programming Series. MIT Press, 1988.

[54] SHAPIRO, S. C. The SNePS Semantic Network Processing System. In *Associative Networks: The Representation and Knowledge by Computers*, N. V. Findler, Ed. Academic Press, New York, 1979, pp. 179–203.

[55] SHAPIRO, S. C. *Cables, paths and "subconsious" reasoning in propositional semantic networks*. Morgan Kaufmann, San Mateo, CA, 1991, ch. 4, pp. 137–156.

[56] SHAPIRO, S. C., AND RAPAPORT, W. J. The SNePS Family. *Computers Math. Application 23*, 2-5 (1992), 243–275.

[57] SHAPIRO, S. C., AND THE SNEPS IMPLEMENTATION GROUP. *SNePS-2.4 User's Manual*. Department of Computer Science at SUNY Buffalo, 1998.

[58] SHU, W. *Chare Kernel and Its Implementation on Multicomputers*. PhD thesis, University of Illinios, Urbana, Champaign, 1990.

[59] SHU, W., AND WU, M. Runtime Incremental Parallel Scheduling (RIPS) on Distributed Memory Computers. *IEEE Trans. on Parallel and Distributed Systems 7*, 6 (Jun. 1996), 637–649.

[60] SNEPS HOME PAGE. http://www.cs.buffalo.edu/pub/www/sneps.

[61] STOFFEL, K., HENDLER, J., SALTZ, J., AND ANDERSON, B. Parka on MIMD-Supercomputers. Tech. Rep. CS-TR-3672, Computer Science Dept., UM Institute for Advanced Computer Studies, University of Maryland, College Park, 1996.

[62] STOFFEL, K., TAYLOR, M., AND HENDER, J. Efficient Management of Very Large Ontologies. In *AAAI-97* (1997), MIT Press.

[63] STUMM, M., AND ZHOU, S. Algorithms Implementing Distributed Shared Memory. *IEEE Computer Magazine* (May 1990), 54–64.

[64] SUN ENTERPRISE SERVER SPECIFICATION. http://www.sun.com/servers/enterprise/e4000/index.html.

[65] TAN, C. J. Deep Blue: Computer Chess and Massively Parallel Systems (extended abstract). In *Proceedings of the 9th ACM International Conference on Supercomputing* (1995), ACM, pp. 237–239.

[66] THINKING MACHINE CORP. **Lisp Reference Manual*. Cambridge, MA, Sep. 1988.

[67] THINKING MACHINE CORP. The *Lisp Programming Language. In *CM5 Technical Summary*. The Thinking Machine Corporation, http://grover.think.com/ index.html/ html/ products/ hardware/ cm5_tech.htm, Nov. 1992, ch. 11.

[68] THINKING MACHINES CORPORATION. *The CM-2 Technical Summary*. Cambridge, MA, 1990.

[69] YAO, C., AND GOLDBERG, B. Pscheme: Extending Continuations to Express Control and Synchronization in a Parallel LISP. Tech. Rep. 655, New York University, 1990.