MICROCOPY RESOLUTION TEST CHART

AD-A215 762

Hypercube Expert System Shell -
Applying Production Parallelism

THESIS

William Arthur Harding
Captain, USAF

AFIT/GCS/ENG/89D-6

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

89 12 26 147

AFIT/GCS/ENG/89D-6

Hypercube Expert System Shell - Applying Production Parallelism

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

William Arthur Harding, B.A.

Captain, USAF

Preface

This study was prompted by the need for expert system software to produce required results in real-time for systems like the Robotic Air Vehicle. The expert system processing speedups realized on serial machines due to state-saving match algorithms like Rete are impressive, but they still fall short of real-time processing. This research investigation focuses on parallel processing of such state-saving algorithms with the goal of achieving real-time expert system processing.

William Arthur Harding

Acknowledgments

This work would not have been possible without the support of my wife, Carol. I thank my thesis advisor, Dr. Gary Lamont, for his insight and direction. I am grateful to Captain Donald Shakley and Lieutenant Jesse Fanning for providing the background information from which I could proceed with my research. I am also endebted to my many AFIT classmates, especially Captain R. Andrew Beard, Captain Mark Huson, Captain William Koch, and Captain Michael Proicou, for their guidance and support at key points during this research investigation.

I lovingly dedicate this effort to my daughter, Erin Marie.

<div align="right">William Arthur Harding</div>

Table of Contents

List of Figures

AFIT/GCS/ENG/89D-6

Abstract

This research investigation proposes a hypercube design (hardware and software) which supports efficient symbolic computing to permit real-time control of an air vehicle by an expert system. Real-time processing requirements motivate the researcher to alleviate common expert system bottlenecks. Examples of these bottlenecks include the inefficiency of symbolic programming languages like Lisp and the disproportionate amount of computation time commonly spent in the match phase of the expert system match select-act cycle.

In the design presented in this research investigation, faster processing of the Defense Advanced Research Projects Agency's (DARPA) Robotic Air Vehicle (RAV) expert system software is approached through 1) fast production matching using an expert system shell which employs Rete, a state-saving match algorithm, 2) more efficient expert system shell program execution due to implementation using the C-Programming Language and 3) parallel processing of the RAV expert system production cycle using multiple copies of the serial expert system shell. For this design, the serial C-Language Integrated Production System (CLIPS) shell, which uses the Rete match algorithm, is modified to execute in parallel on the iPSC/2 Hypercube. Although the RAV expert system is the application of interest in this investigation, the parallel expert system shell is capable of processing any CLIPS-syntax software.

Speedups achieved using this architecture are quantified through theoretical timing analysis, and comparison with serial architecture performance results, with earlier parallel architectures' performance results, with best case theoretical analysis performance results, and with the "real-time" goal performance. This performance quantification approach introduces the concept of a performance spectrum which exposes the level of maturity of RAV expert system processing in particular and the level of maturity of parallel expert system shell processing on a multicomputer in general.

Hypercube Expert System Shell - Applying Production Parallelism

I. Introduction

The feasibility of improving the performance of production system software (particularly, expert system software) running on a parallel architecture is an area of current interest in artificial intelligence research. One sponsor of research in this area, the Air Force Wright Aeronautical Laboratories (AFWAL), requires a fast multiprocessor architecture to process an expert system capable of piloting a robotic air vehicle (RAV) (28:1326). The Strategic Defense Initiative Organization (SDIO) is another sponsor of faster expert system processing research. This research investigation is performed in support of the RAV expert system project and is a follow-on to the investigation by Captain Donald Shakley (37). Some of the areas recommended for future research in Shakley's thesis are pursued.

1.1 System Requirement

The concept of the RAV project is to create an unmanned air vehicle capable of autonomous operation (see Appendix A). To develop this robotic capability, AFWAL contracted Texas Instruments, Inc. to produce RAV software which, when executed, could perform all of the basic piloting skills as well as various navigation and obstacle avoidance functions (28:1326). AFWAL and TI jointly chose an expert system as the preferred RAV software implementation method for several reasons.

First and foremost, AFWAL uncovered through literature reviews that efforts to apply traditional software implementation methods to intelligent vehicle control had failed due in part to unmanageability of code (37:77). Second, researchers increasingly are discovering that applications that humans currently do better than machines (like the task of piloting an aircraft) lend themselves to solution using an artificial intelligence approach. Because expert system technology is currently one of the most successful branches of artificial intelligence, AFWAL and TI chose the expert system approach to the RAV software construction (18:4).

Unfortunately, the RAV expert system has proven to be too compute-intensive to yield results in real-time on any serial or parallel computer architecture (hardware and software) developed to date. Real-time means "the time needed to make a calculation has to be less than the time from when the need for the calculation is recognized until the time when the response is needed to take action" (31:10). The RAV concept is not feasible until a computer architecture that produces real-time results processing the RAV expert system can be developed.

1.2 Related Work

The only parallel architectures applied to the RAV expert system were developed by Shakley (37). These include parallel processing designs implemented on a network of Texas Instruments (TI) Explorer Lisp machines and on a first generation Intel Personal Supercomputer (iPSC). Although this study showed that the increased parallelism of the iPSC design could in fact produce processing speedup compared to the TI Explorer design, the amount of speedup realized was hampered by the effects of interprocessor communication overhead and load imbalance. These effects were caused mainly by the simple data decomposition employed on the iPSC design (37:72). Furthermore, comparing the iPSC design to the TI Explorer design may have allowed factors unique to each of these products to skew the analysis. That is, the difference in the observed processing speeds of the two systems may have resulted from factors other than just the difference in the number of parallel processors applied.

1.3 Problem Statement

The goal of this research investigation is the design, implementation, and analysis of a parallel processing architecture implemented on a second generation Intel Personal Supercomputer (iPSC/2) and the quantification of the processing speedup realized by this design as applied to the RAV expert system.

1.4 Research Objectives

The following are the objectives of this research:

- implement an efficient and effective parallel RAV expert system architecture on the iPSC/2 configured to use varying numbers of processors to gather data on the speedups realizable through parallelization

- quantify the performance results for this parallel RAV design through theoretical order-of timing analysis and through comparisons with past research results, with serial architecture results, with best-case theoretical analysis results, and with the real-time goal performance

1.5 Scope

The Intel Personal Supercomputer (iPSC/2) is the multicomputer employed in this research investigation (see Appendix B). This supercomputer, which can be configured with up to 128 available processors, is chosen due to availability as well as to research interest in the performance of artificial intelligence application software executing on a multicomputer. Limiting the analysis to designs implemented on this one supercomputer eliminates the possibility of machine-unique factors skewing the results, thus yielding more quantifiable comparisons of the performance differences between designs.

Software coding is done in the C Programming Language. C was chosen over the major AI programming languages, Lisp and PROLOG, to facilitate better efficiency and portability (27:190). C is also the only one of these languages available for use on the AFIT iPSC/2 at the time of this research investigation. Shakley's parallel RAV design, however, was implemented not in C but in Lisp on the iPSC. Shakley's design is not reimplemented in C as part of this research investigation, because the potential execution speedups attainable through reimplementation in C on the iPSC/2 are derived much more easily using theoretical analysis.

Because the RAV expert system is the application of interest, it is the only application to which research designs are applied for performance comparison purposes. Pertinent

1-3

RAV computation data is acquired from and assumed validated by AFWAL RAV project managers.

1.6   Constraints

The decision to use the iPSC/2 to implement an RAV parallel expert system drives many program design decisions. Certain features of iPSC/2 hardware configuration, such as a local memory for each processor node, exclude many program design options. For example, algorithms that perform relatively few operations between synchronizations (i.e., have a small grain size) usually exhibit poor efficiency on multicomputers that employ multiple local memories as does the iPSC/2 (34:53). Consequently, small grain algorithms are excluded from consideration in this study. In fact, the iPSC/2's multicomputer design prevents this research investigation from taking advantage of extensive production system research performed to date using shared-memory multiprocessors (14:63).

The RAV expert system is the application of interest to this research investigation. Although the research methodology and design factors are applicable to any parallel architecture research, the performance results realized are unique to the RAV expert system. That is to say, even though the form of the RAV expert system is not unlike that of any other expert system program, the application of the architecture implemented in this investigation to other expert systems will not necessarily produce the same performance effects. This fact is significant when analyzing the expert system shell execution speeds observed for an application and when using these execution speed analysis metrics to quantify the merit of the expert system shell.

1.7   Summary

This research investigation addresses improving the performance of production system software by executing this software on a parallel architecture. Such an architecture is applied to the Robotic Air Vehicle (RAV) expert system. The parallel expert system architecture is implemented on the iPSC/2 supercomputer using the C programming language. Realization and quantification of performance speedups using this parallel architecture are the objectives of this investigation.

Chapter II provides background information on the key topics underlying this research effort, including expert systems and parallel processing. Chapter III describes the Rete match algorithm and issues regarding parallelization of Rete. Chapter IV details the fundamental approach to this research and introduces performance quantification using a performance spectrum. In Chapters V and VI, the lower and upper performance bounds of processing the RAV software are derived. Capt Shakley's parallel RAV design and its performance are related in Chapter VII. The parallel Rete expert system shell design and its application to the RAV expert system are presented in Chapter VIII. The many performance metrics produced in Chapters V through VIII are compared using the performance spectrum method in Chapter IX. Finally, the research conclusions and some recommendations stemming from research findings are offered in Chapter X. Appendices are available for those seeking deeper insight into the RAV project itself (Appendix A), parallel architectures (Appendix B), the theoretical and actual realizations of the parallel Rete expert system shell (Appendix C and Appendix D, respectively), and programmers' and users' manuals for the expert system shell (Appendix E and Appendix F, respectively).

II. Background

This chapter summarizes some of the underlying concepts, uncovered through literature search, that form the basis of this research, including production systems, expert systems, parallel processing, communication overhead, and load imbalance.

2.1 Production System

A *production system* is a pattern recognition formalism based on string replacement rules. An order is imposed on these rules to decide which applicable rule to apply next. Production system computation proceeds as a string-resolution-based search. A control strategy is used with string-modifying production rules to model certain types of human problem-solving behavior (30:48).

The major elements of a production system are a global database of *facts*, a set of *production rules*, and a *control system*.

A *fact* is an assertion which represents a specific item of knowledge. A fact is generally of the following abstract form (14:8):

$$(< object >< attribute1 >< value1 >< attribute2 >< value2 > ...)$$

Written in this form, a fact is a parenthesized list consisting of a constant symbol, commonly called the *object*, and zero or more *attribute-value* pairs. An object represents an entity within a problem's domain that is of significance to the solution of that problem. An attribute *represents* a specific characteristic of its associated object. A value is the parameter instantiated for a given attribute (36:75).

The global database of facts, also called *working memory* (WM), is the central data structure used by a production system. Because facts are held in working memory, the facts are often referred to as working memory elements (WME). At the start of production system computation, WM contains a set of initial facts about the problem domain. As production system computation proceeds, facts are added and/or deleted from WM.

A *production rule* is expressed as strings that represent general knowledge about a particular subject area. A production rule generally appears in the form of an "IF condition THEN action" implication. By convention, the condition part of a rule is called the Left Hand Side (LHS) and the action part of a rule is called the Right Hand Side (RHS). The LHS is consists of one or more *condition elements* (CEs) that are compared to the actual state of referenced facts in WM at a given point in computation. The RHS consists of an unconditional sequence of *actions* which can add facts to and/or delete facts from WM. The following is a sample production rule (14:12):

(*rule* sample :

     if   ((*object2*   *attribute*1   15   *attribute*2   y)

        (*object4*   *attribute*1   y)

)

    then

        (*add*   *object3*   *attribute*1   12))

This production rule, named "sample", consists of two condition elements in its LHS and one action, an add to WM, in its RHS. Note that this production rule form can be generalized in the following first-order predicate calculus form:

    if (($P_1(x_1, x_2)$) +

        ($P_2(x_3, x_4)$))

)

    then

        ($P_3(x_1, x_3)$)

In this example, both predicate P_1 and predicate P_2 must be *true* for predicate P_3 to be *true*.

The production rules, known as the *production memory* (PM), are matched against the global database of facts. All rules in PM can match against and alter any facts in WM. The LHS of each rule in PM is either satisfied or is not satisfied by one or more facts in the global database in WM at any given discrete step in production system computation. If a rule's LHS is satisfied, that rule may be applied. Application, or *firing*, of a rule adds facts to and/or deletes facts from WM as dictated by that rule's RHS.

2-2

For example, the production rule "sample" shown above is satisfied when there exists one or more facts in WM that meet all of the following conditions: *i*) the fact's object matches *object2*, *ii*) the fact's current value *y* for *attribute1* is "15" and *iii*) the fact's current value *y* for *attribute2* is the same as the current value *y* saved for *attribute1* in some other fact whose object matches *object4*. If all three of the above conditions exist in WM, the "sample" rule is then eligible to fire, adding a new fact with object *object3* and value "12" for *attribute1* to WM.

It is possible during production system computation that more than one rule's LHS is satisfied by the state of the facts in WM. The list of satisfied production rules in PM is commonly called the *conflict set*. The *control system* chooses which among all of the satisfied rules is to be applied. The choice of which rule to fire may be based on some firing priority assigned to each rule, on some characteristic of the rule string itself, or on some arbitrary ordering. The control system may employ an *irrevocable* control strategy or a *tentative* control strategy. In an irrevocable strategy, a satisfied rule is selected and applied without provision for reconsideration later. In a tentative strategy, a satisfied rule is selected and applied, but with provision made to return later to that point in the computation to apply some other satisfied rule instead. The control system halts computation when a predefined termination condition, or *goal* condition, exists in the current contents of WM (30:18).

The basic production system algorithm is illustrated in Figure 2.1. This algorithm is executed as a Match-Select-Act cycle, usually in the following order (11:36):

1. Match - evaluate the LHSs of the production rules to determine which are satisfied given the current contents of WM

2. Select - choose one production rule with a satisfied LHS from the conflict set; if no production rules have satisfied LHSs, return control to the user

3. Act - perform the actions specified in the RHS of the selected production rule

4. If a termination condition is detected, then return control to the user; otherwise go to Step 1 (Match)

```
Procedure PRODUCTION

    1. DATA — initial fact database

    2. until DATA satisnes the termination condition, do;

    begin

            select some rule R in the set of rules
                        that can be applied to DATA

            DATA — result of applying R to DATA

    3. end
```

Figure 2.1. Production System Algorithm (30:21)

2.2 Expert System

One class of production systems is that of expert systems. Expert systems are formal computing systems, or programs, that use the production system paradigm to offer advice or solve problems by reasoning with bodies of knowledge highly specific to a particular domain (4:105). The bodies of knowledge are generally extracted from human experts in the domain. Using this knowledge base, the expert system attempts to emulate the experts' methodology and performance toward solving a problem (27:291).

Knowledge engineering is the interdisciplinary AI field concerned with the extraction of knowledge from domain experts and the transfer of this knowledge into hardware and software representation. After the knowledge engineer has developed the basic expert system, the acquired expertise is refined through a process of giving the system example problems to solve. Domain experts criticize the system's behavior and make any required changes or modifications to its knowledge. This process is repeated until the system has achieved the desired level of performance (27:16). Because the knowledge engineering task is difficult and expensive, expert systems typically emulate problem solving over a very limited domain.

The principal components of a rule-based expert system are a *knowledge base*, an *in-*

ference engine, and a *man-machine interface* (see Figure 2.2). The knowledge base contains the global database of facts and the production rules that embody an expert's expertise. The inference engine is the control system that serves as a reasoning mechanism and search controller. It is the inference engine that performs the match-select-act cycle. These two expert system components represent the major elements present in any production system. The man-machine interface simply produces dialog (string, graphics, etc.) between the computer and the user (18:8-13).

```
┌──────────┐    ┌──────────┐    ┌──────────┐    ┌──────────┐    ┌──────────┐
│ User     │───▶│ Knowledge│───▶│Explanation│──▶│ Inference│───▶│ Output   │
│ Interface│    │ Base     │    │ Facility │    │ Engine   │    │          │
└──────────┘    └──────────┘    └──────────┘    └──────────┘    └──────────┘
                                      ▲
                                ┌──────────┐
                                │  Data    │
                                └──────────┘
```
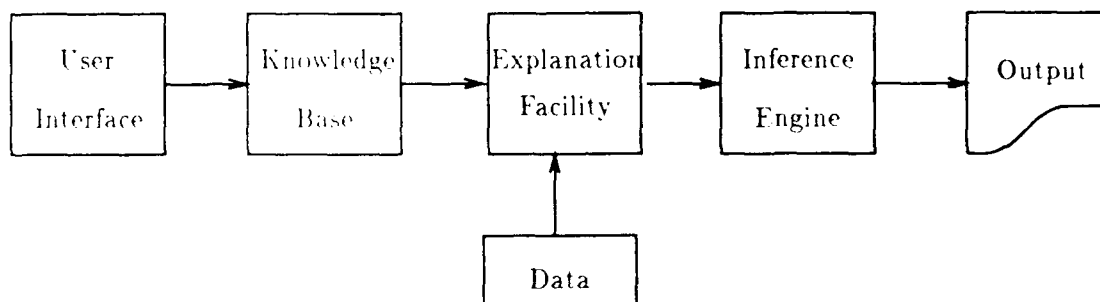
Figure 2.2. Expert System Components (18)

2.3 Speedup and Parallel Processing

Compute-intensive applications, such as the RAV expert system (28), require processing beyond the performance ability of conventional, single-processor machines. Ishida and Stolfo suggest that although speed improvements in single-processor machines have occurred, "further speed improvements are required for very large [expert] systems with severe time constraints." (23:568) *Parallel processing*, which means applying several processors to run the solution algorithm for a single problem, is one approach to achieving such speed improvements.

The degree to which parallel processing improves processing speed depends upon the efficient use of available processors. Two main obstacles to achieving peak performance using parallel processing are *communication overhead* and *load imbalance* (37:72). One key

to overcoming both obstacles is the proper choice of a problem decomposition approach. A decomposition algorithm divides the overall expert system into independent subunits, or *tasks*, each of which is assigned to one of the available processors for execution.

2.4 Communication Overhead

Communication, or the passing of required information between processors, seriously degrades expert system performance because the overhead can become so intense that "the [processors] spend more of their time communicating than computing." (6:72) Obviously, chosing the most current technology parallel processor that performs interprocessor communication as fast as possible is one way of reducing communication time. An algorithmic approach to reducing the detrimental effects of communication overhead is to decompose and distribute the expert system tasks in such a way that the interprocessor communication time incurred by parallelizing the expert system is less than the computation time saved through parallelization of the expert system. The benefits of a "good" expert system decomposition are realized regardless of the particular parallel processor's communication speed. But an extremely fast interprocessor communication capability cannot be expected to always overcome the effects of a poor decomposition.

The number of processors over which an expert system's tasks are decomposed also affects communication overhead. A phenomenon called the *Amdahl effect* dictates that any parallel algorithm shows constrained speedup if there is not enough work to be done by the number of processors available (34:60). The Amdahl effect suggests there exists an optimal number of processors upon which a parallel program can be run. Applying more than this optimal number of processors adds communication overhead that overcomes some of the computation time savings of parallelizing the system. Figure 2.3 shows how processing can slow down when more than the optimal number of processors are applied to a problem. Proponents of parallel processing hasten to point out that Amdahl's effect occurs under the assumption that some number of necessarily sequential operations exist to interrupt the parallel execution of an algorithm. Hence, Amdahl's argument serves as a way of determining whether an algorithm is a good candidate for parallelization, rather than as a provable limit to speedup for all algorithms (34:19).

2-6
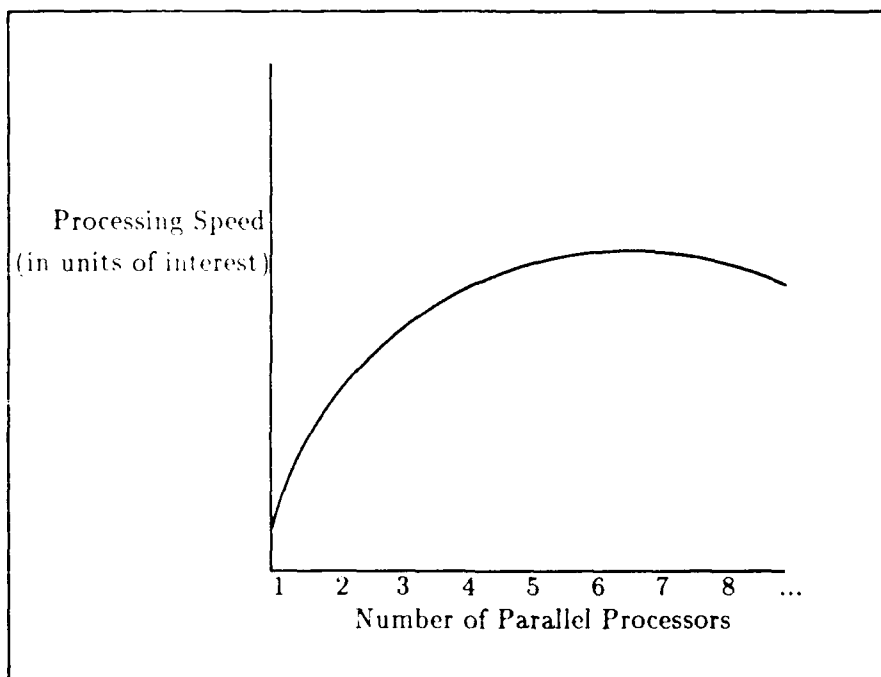
Figure 2.3. Example Parallel Speedup Chart

2.5 Load Imbalance

An imbalance of the task load among the processors also severely degrades performance. The goal is to ensure that "the tasks being executed [are] uniformly distributed amongst the various processing nodes in a manner which maximizes resource utilization to enhance the total throughput of the system." (2:189) Achieving load balance, like achieving low communication overhead, also depends on choosing a good problem decomposition. A good decomposition algorithm maps the expert system tasks to the available processors in such a way as to keep as many of the processors busy doing useful work as possible.

There are two general task allocation policies: *static* decomposition and *dynamic* decomposition. Static decomposition assumes that tasks and their precedence relations are known before execution. Dynamic decomposition assumes that tasks are generated during program execution. The advantage of static decomposition is that it allows the preallocation of tasks to processors, thus reducing the amount of interprocessor communication. The advantage of dynamic decomposition is that it makes it easier to keep all the processors busy because tasks requiring processing are assigned to the first available processor (34:62). But dynamic decomposition adds communication overhead to distribute tasks to available processors.

2.6 Summary

A production system consists of a global database of facts and a set of production rules (constituting the knowledge base), and a control system (inference engine). The basic production system execution algorithm is the Match-Select-Act cycle. The rule LHSs are matched against the current facts in the knowledge database, one of the satisfied rules is selected for firing, and the RHS actions of the selected rule are performed to update the fact database. An expert system applies the production system paradigm to problem solving using knowledge that is specific to a particular domain. Parallel processing, which means applying several processors to the solution of a single problem, is employed to speed up processing of a wide range of applications, including expert systems. Two main obstacles to realizing speedup through parallelization are communication overhead and load imbalance.

III. Production System Performance Improvement Concepts

This chapter offers some production system parallelism concepts and introduces the serial Rete match algorithm. The parallelization of the Rete algorithm lays the groundwork for a new distributed processor parallel RAV expert system design.

3.1 Potential Production System Parallelism

As discussed in Chapter II, the three steps that are repeatedly performed to execute a production system algorithm are *match, select,* and *act.* Figure 3.1 illustrates the information flow among these three steps. Note that a synchronization point exists after the select step and before the subsequent act step. This synchronization point has a serializing effect on the match-select-act cycle. The select step must finish completely before the next production rule to fire can be determined and its RHS evaluated. Without this synchronization, a potential race condition exists in which the WM change inputs to a match-select-act cycle may be corrupted by outputs of that same cycle. No other mandatory synchronization points exist in the cycle.
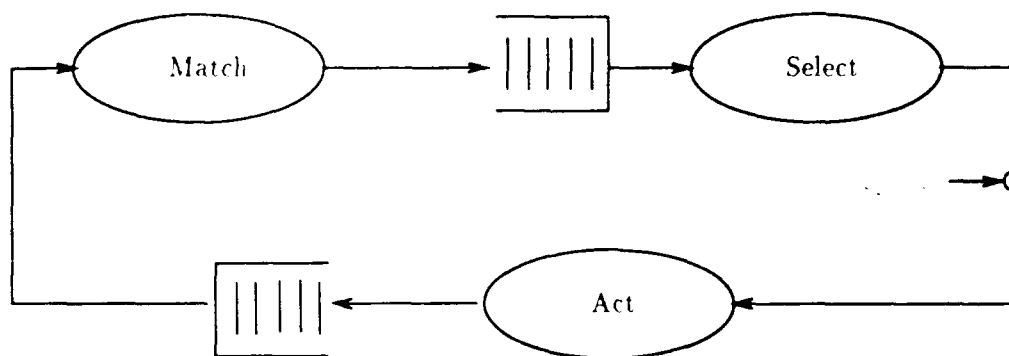
Figure 3.1. Match-Select-Act Flow Graph (14:46)

This feature of production systems suggests areas of potential parallelism. For example, it is possible to use parallelism within the match step, within the select step, and within the act step. It is further possible to overlap the processing performed within the

match step and the select step of the same cycle and the processing performed within the act step of one cycle and the match step of the next cycle (14:45). These potential inter-step overlaps are represented by the queue symbols in Figure 3.1. But as noted above, it is not possible to overlap the processing within the select step of one cycle and the subsequent act step.

3.2   Rete Match Algorithm

All of the information presented in this section is taken from (15) except items noted from other sources.

The most time consuming step in the execution of production systems is the *match*. Matching is a pattern recognition activity which involves matching the left hand sides (LHSs) of all productions against all facts in working memory (WM). Even with special-ized algorithms, the match step constitutes around 90% of the interpretation time (13:4). Consequently, speeding up the match step in production systems is an area of intense research.

The Rete (pronounced "reet") algorithm is among the most efficient algorithms for match yet developed. To achieve this efficiency, Rete exploits two features common to most production systems: first, only a small fraction of working memory typically changes every match-select-act cycle; and, second, similar condition elements often appear repeatedly among the productions in production systems. Rete exploits the first feature by storing results of match from previous cycles and using them in subsequent cycles. Rete exploits the second feature by recognizing condition elements referenced in multiple productions and performing common tests only once (13:4).

The Rete algorithm uses an augmented discrimination network compiled from the LHSs of the productions to perform the match. In fact, the name "Rete" comes from the latin word for "network." (29) Figure 3.2 shows such a network for productions p1 and p2 which appear in the top part of the figure. In this figure, lines have been drawn between nodes to indicate the paths along which information flows. Information flows from the top node down along these paths.

(rule p1 (C1 attr1 ?x attr2 12)
    (C2 attr1 15 attr2 ?x)
    (C3 attr1 ?x)
    →
    (remove 2))

(rule p2 (C2 attr1 15 attr2 ?y)
    (C4 attr1 ?y)
    →
    (modify 1 attr1 12))
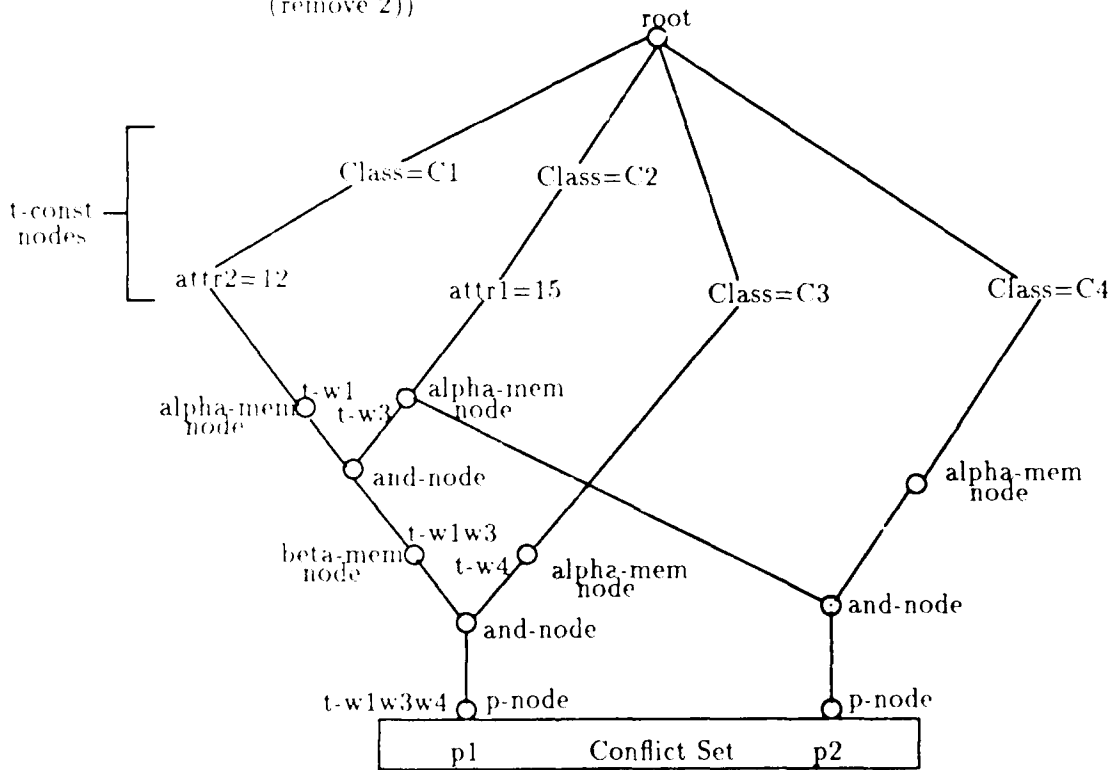
Add to Working Memory

fact1: (C1 attr1 12 attr2 12)
fact2: (C2 attr1 12 attr2 15)
fact3: (C2 attr1 15 attr2 12)
fact4: (C3 attr1 12)

Note: t-wn represents a token
and its contents

Figure 3.2. Sample Rete Graph (14:12)

To generate the network for a particular expert system application, the production rules are parsed and "translated" by a *network compiler*. Note that the Rete network compiler is not a compiler in the conventional sense of computing. The Rete compiler builds a network which serves as the data structure acted upon by the inference engine during execution of the expert system.

The network compiler proceeds first with the individual condition elements in the rules' LHSs. For each condition element, the compiler chains together test nodes that check the following:

- if the attributes in the condition element that have a constant as their value are satisfied

- if the attributes in the condition element that are related to a constant by a predicate are satisfied

- if two occurrences of the same variable within the condition element are consistently bound (i.e. working memory elements with the same value for the specified attribute exist)

Each node in the chain performs one such test. These three tests are called *intra-condition* tests because they correspond to individual condition elements. In Figure 3.2, the nodes with a single predecessor (near the top of the network) are the ones that are concerned with individual condition elements.

Once the network compiler has finished with the individual condition elements, it adds nodes that check for consistency of variable bindings across the multiple condition elements in the LHS. These tests are called *inter-condition* tests because they refer to multiple condition elements. The nodes with two predecessors are the ones that check for consistency of variable bindings between condition elements.

Finally, the compiler adds a special terminal node to represent the successful matching of the production to which this part of the network corresponds. The terminal nodes are at the bottom of the figure.

Note that when two LHSs require identical nodes, the compiler builds a shared set of nodes in the network rather than duplicate nodes. This feature of Rete ensures that the same test is not performed repeatedly for multiple rules within a single match step.

To avoid performing all of the same tests completed during the previous match step, the Rete algorithm stores the result of a match with working memory as *state* within the network. Only changes made to the working memory by the most recent production firing have to be processed every match-select-act cycle. That is, the input to the Rete network consists of the most recent changes to the working memory. These changes filter through the network and, where relevant, the state stored in the network is updated. Due to the Rete algorithm's state-saving feature, the amount of effort expended by the matcher depends primarily on the rate of change of working memory rather than the absolute size of working memory (11:37). The output of the network consists of a specification of changes to the list of rules eligible to be fired. This list of rules is called the *conflict set* because only one of these rules may be allowed to fire under the production system paradigm. Consequently, the rules can be said to be in conflict over the right to be fired in the current cycle.

The objects that are passed between nodes in the network are called *tokens*, which consist of a tag and a list of working memory elements. The tag can be either a +, indicating that an element has been *added* to the working memory, or a -, indicating that an element has been *deleted* from working memory. No special tag for working memory element modification is needed because a *modify* is treated as a delete followed by an add. The list of working memory elements associated with a token corresponds to the permutation of those elements that the system is trying to match or has already matched against a subsequence of condition elements in the LHS.

The discrimination network produced by the Rete network compiler consists of a number of the following types of nodes:

- *Root Node*: This node forms the root of the discrimination net. It broadcasts tokens corresponding to any change in the working memory to all its successor nodes. In Figure 3.2, the root node is shown at the top.

- *Constant Test (t-const) Nodes:* These nodes are used in the network to perform intra-condition tests, for example, to check if condition attributes that have constant symbols or numbers as their values are satisfied. Each t-const node checks for one feature. Whenever the token arriving at the input of a t-const node satisfies the associated test, it is passed on to the successors of the t-const node. If the token does not satisfy the test, it is not passed on to the successors. In Figure 3.2, the nodes towards the top of the network are t-const nodes. Because the second condition element of production p1 is similar to the first condition element of production p2, t-const nodes "Class=C2" and "attr1=15" are shared in the network for rules p1 and p2.

- *Alpha Memory (alpha-mem) Nodes:* If a working memory element satisfies all intra-condition tests for a condition element, the working memory element is said to *partially match* the condition element. Note that it may not, as yet, satisfy all the inter-condition tests. Tokens corresponding to working memory elements that partially match a condition element are stored in the alpha-mem node for that condition element. When a token arrives at an alpha-mem node with a + tag, the token is stored in the alpha-mem node and a copy of the token is passed to the node's successors. If the tag is -, a corresponding token with a + tag must already exist in the alpha-mem. The corresponding + token is deleted from the alpha-mem node and the incoming token is passed down to the successors of the alpha-mem node. If two condition elements in the same or different productions have exactly the same tests for a successful partial match, the network compiler generates a shared alpha-mem node for the two. This sharing of an alpha-mem node can be seen in Figure 3.2.

- *Beta Memory (beta-mem) Nodes:* Just as alpha-mem nodes store tokens that partially match individual condition elements, so beta-mem nodes store tokens that partially match two or more condition elements in the LHS of a production. The list of working memory elements in beta-mem tokens has length two or more. The response of beta-mem nodes to the arrival of tokens at their inputs is exactly the same as that of alpha-mem nodes. The beta-mem nodes form the left input of and-nodes and not-nodes.

- *And-Nodes:* The and-nodes are the first of the two-input node types. The primary function of an and-node is to check for consistency of variable bindings between the partially matched tokens it receives on its left and right inputs. The right input of an and-node always comes from an alpha-mem node, while its left input can come from an alpha-mem or a beta-mem node. Whenever a token arrives at the left input of an and-node, the and-node compares the incoming token to each token stored in the mem-node connected to its right input to check if they are consistent. For every right-token which is consistent with the left-token, a new token is constructed and sent down to the successor nodes. The new token has the same tag as that of the left-token, and the list of working memory elements is the concatenation of the working memory element lists for the left and right tokens. The case when a token arrives at the right input of an and-node is processed exactly as above, with left and right interchanged.

- *Not-Nodes.* The not-nodes are the second of the two-input node types. They also have a left and a right input. The not-nodes are used by the network to implement negated condition elements. Their functionality differs from that of and-nodes only in minor ways. One difference is that not-nodes keep reference counts with tokens in left memory to find when there are no tokens in the right memory that are consistent with them.

- *Production Nodes (p-nodes):* These are the terminal nodes in the network. There is one such node associated with each production. Whenever a token with a + tag flows into a p-node, it adds an instantiation (corresponding to the token) of the associated production into the conflict set. The arrival of a token with a - tag leads to the deletion of the corresponding production instantiation from the conflict set.

- Other Nodes: Other than the node types mentioned above, the network uses two more node types. These are the Two-Nodes and the Any-Nodes. The two-node is used as a place filler in some circumstances, and the any-node is used when the value of an attribute is to be one of a number of alternatives.

The match step in a Rete network interpreter can itself be divided into two parts: the

selection phase, which consists of evaluating the t-const nodes; and the *state-update phase*, which consists of evaluating the alpha-mem nodes, beta-mem nodes, and-nodes, not-nodes, and p-nodes. Comparing these two phases, about 75% to 95% of the total processing time is spent performing the state-update phase (14:47).

3.3 Parallelizing Rete

The Rete match algorithm is suitable for parallel implementations. The data-flow like organization of the Rete network makes it possible to evaluate the activations of different nodes in the network in parallel. It is also possible to evaluate multiple activations of the same node in parallel and to process multiple changes to working memory in parallel (14:20). Of the many sources of production system parallelism, the following three are particularly important in the parallelization of Rete and, specifically, of the state-update phase of the Rete match step: *production parallelism*, *node parallelism*, and *action parallelism*.

Production parallelism is accomplished by dividing the productions in a program into several partitions and performing the match for each of the partitions in parallel. Figure 3.3 illustrates the partitioning of productions of an expert system. Production partitioning is a static task decomposition approach. Consequently, the main advantage of using production parallelism is that no communication is required between the processes performing match for different productions or different partitions. That is, it is large-grain parallelism. Disadvantages of production parallelism are that it is limited by 1) the typically small number of productions affected per change to working memory, 2) the large variance in the amount of processing required by the affected productions, and 3) the loss of sharing in the overall Rete network as a result of production partitioning (14:48-49).

Node parallelism, which is unique to the Rete algorithm, means that activations of different two-input nodes in the Rete network are evaluated in parallel. An advantage of node parallelism is that both activations of two-input nodes belonging to different productions (corresponding to production parallism) and activations of two-input nodes belonging to the same production (resulting in extra parallelism) are processed in parallel. Node parallelism is implemented at a finer granularity than production parallelism to 1) reduce the effect of large variance in the amount of affected productions processing, and 2) to recover
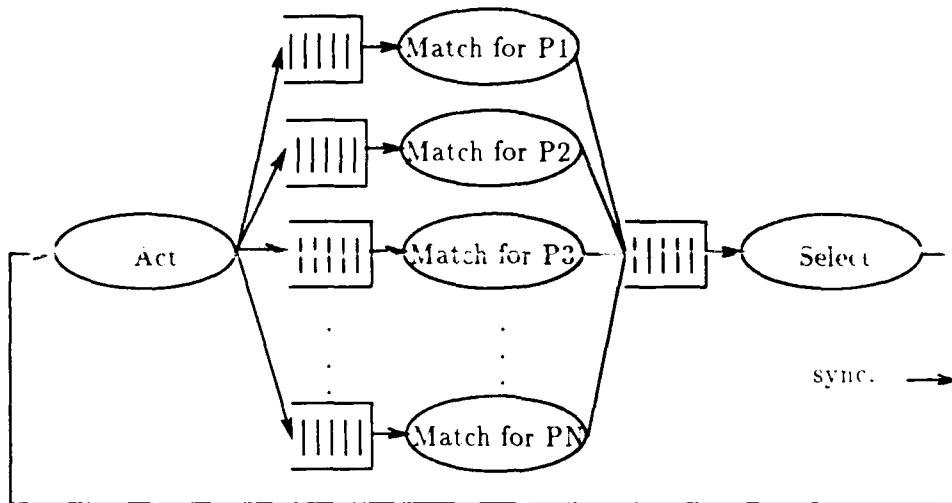
Figure 3.3. Production Parallelism (17:48)

some of the sharing lost in the overall Rete network when using production parallelism. This fine granularity, however, leads to increased communication requirements between processes evaluating the nodes in parallel (14:51).

Action parallelism refers to the concurrent processing of changes made to working memory when a production fires. Action parallelism enhances the speedup obtainable using production, node, and other forms of parallelism (14:54).

A significant amount of research has been performed toward implementing parallel Rete on multiprocessors (14, 25). Fine-grain node parallelism, enhanced with action parallelism wherever possible, is the preferred implementation method when using a multiprocessor architecture. This method is attractive because multiprocessors have the advantage of shared memories (see Appendix B), which offsets much of the cost of increased communication associated with node parallelism (14:58).

In a typical multiprocessor design, a single copy of the Rete network is held in shared memory. The match is broken into fairly small units of work called *tasks*, where a task is an independently schedulable unit of work that may be executed in parallel with other

tasks (17:103). Each task is represented by a token. This token is essentially the same as that described for the sequential Rete matcher, except that it has two extra items of information: the *address* of the node to which the token is to be sent; and. if that node is a two-input node, an indication of whether to send it to the *left* or *right* input. The list of tokens that are awaiting processing is held in a central data structure called a *task queue*. When a processor in the multiprocessor becomes available, it removes a token from the task queue. If, during processing of a token, new tokens are to be sent out, these are entered into the task queue for subsequent processing. See (17) for details of a highly successful parallel production system implementation employing Rete on a shared-memory multiprocessor.

Unlike the parallel Rete research performed on multiprocessors, implementation of a parallel Rete matcher on a multicomputer architecture remains relatively unexplored for two basic reasons. First, the most natural approach to implementing production systems on a multicomputer is production parallelism, enhanced with action parallelism. But preliminary simulation analysis of parallel processing using only production parallelism and action parallelism indicates that the speedups attainable are very low (32:92). Second, the node parallelism approach was shown theoretically and through simulation to be superior to production parallelism. But the fine granularity of node parallelism adds communications costs that may restrict the class of suitable architectures to shared-memory multiprocessors (14:58).

Limited theoretical analysis of a parallel architecture which implements Rete in an object-oriented manner on a multicomputer has been performed (16). This research draws heavily from work performed on multiprocessor designs. At the time of this research investigation, the only parallel multicomputer implementation of a production system interpreter that employs Rete is II CLIPS, developed for the FLEX/32 (MIMD) multicomputer (35).

3.4 Summary

Production systems lend themselves to parallel execution. The only mandatory synchronization point in the match-select-act cycle exists after the select step and before the subsequent act step. Speeding up processing of the match step is critical, as it typically

constitutes around 90% of the production cycle time. The Rete match algorithm speeds match time in two ways: first, Rete saves WM search time by saving state between cycles, a feature that takes advantage of the small fraction of WM typically changed per cycle; second, Rete recognizes condition element references common to multiple rules and executes common tests only once. The data-flow like organization of the Rete network makes it suitable for parallel implementations to take particular advantage of production parallelism, node parallelism, and act parallelism during execution. Parallel multicomputer implementation of a production system interpreter that employs Rete is an area of research still in its infancy.

IV. Research Methodology

This chapter presents and justifies the research methodology applied to parallelization of the RAV expert system during this research investigation. The following are the restated and expanded goals of this research:

- Present the fastest processing of the RAV expert system achieved to-date using the current state-of-the-art parallel design and determine this design's expected performance if implemented on the iPSC/2.

- Design and implement a new parallel design on the iPSC/2 using a more efficient and effective match-select-act algorithm (Rete) to achieve speedup over the current state-of-the-art design.

- Ensure valid performance comparisons between these parallel designs by considering implementations of the designs that use the same tools (e.g. hardware, language) and the same input data wherever possible.

- Determine the relative usefulness of these parallel designs by showing their relation to the expected lower and upper bounds of parallel processing performance and to the desired "real-time" performance of the RAV expert system.

4.1 Justification of Method Selected

It is common practice in parallel computer architecture research to compare the performance of one's *new design* with that of the current *state-of-the-art* design applied to the same problem. Speed and correctness of processing are the key performance criteria analyzed. The comparison of designs is necessary to show the advancement of knowledge in the field of application. But this approach by itself offers only a very limited analysis of the merit of the new design for two main reasons.

First, the new parallel design is often configured to run on a different machine than that used by the previous parallel design. Consequently, the performance speedups attributable to the different hardware are not distinguishable from the speedups attributable solely to the new program design. It is imperative that architectures being compared use

identical, or at least very similar, hardware to isolate the performance differences attributable to program design.

Second, the true merit of the current state-of-the-art parallel design is often a mystery. This is especially true when current research into a particular area of application is relatively immature. In this case, it is not enough that a new parallel design outperforms the previous "best" design, because both designs' performances may still fall far short of the theoretical performance potential for such an application.

4.2 Performance Spectrum

To determine the true merit of a parallel design for a particular application, one must determine where on a *performance spectrum* this parallel design lies in terms of processing speed. The processing speed is defined in terms that are significant to the particular application (e.g., for expert systems, the performance metric of interest is typically the average number of rules fired per second). Two logical and essential metrics on the performance spectrum are a good serial design's processing speed and the required, or goal, processing speed.

Performance data on a serial, or single-processor, design for an application is often available to the researcher. In fact, it is sometimes the failure of a serial approach to solve a problem in what the user defines as "real-time" that leads to attempts at parallel solutions. Although the "goodness" of a particular serial design is difficult to quantify, still the performance of some serial design is useful to quantify the performance payback realized by parallelizing a solution in the new design. The serial design's performance serves as the *lower bound* on the performance spectrum.

The other key metric on the performance spectrum is the *goal* processing speed. The maturity level of computer architecture research with respect to a particular application can only be determined by comparing the performance of a new design with the goal performance. For example, the "real-time" performance requirement is the goal driving most research in parallel processing of expert systems. The real-time metric defines the speed at which computational results must be produced for the particular application to

4-2

be successful.

An additional metric on the performance spectrum is needed to show if it is reasonable for the researcher to expect to meet the performance goal within the limitations of research assumptions. For example, although processing speed comparisons among a serial design, a state-of-the-art design, and a new design may all show progress toward the goal processing speed, the hardware of choice may not be physically capable of ever actually achieving the goal performance. Consequently, there exists a research need for an *upper bound* performance metric under given application and architecture constraints. Of course, because the upper bound is not readily attainable like the other performance spectrum metrics, theoretical and/or simulation methods must be applied.

Figure 4.1 is an example performance spectrum chart illustrating the above metrics and their interaction for a fictional application. Note that interpretation of this figure suggests the following:

1. The performance of the serial design applied to this problem falls far short of the goal performance. This poor serial design performance leads the researcher to consider the possibility of attempting a parallel solution.

2. The upper bound performance metric suggests that the architectural approach being taken, under ideal parallel conditions, has the potential to achieve the goal performance. The upper bound performance findings encourage continued research on the current architecture.

3. The state-of-the-art design realizes only limited performance improvement over the serial design. This limited performance improvement suggests to the researcher that perhaps a whole new design approach, rather than enhancements to the previous design, should be pursued.

4. The new design performance produces significant speedup over the serial design and approaches the goal performance, lending merit to the research contribution of this new design.

Figure 4.1. Example Performance Spectrum Chart

Processing Speed
(in units of interest)

Number of Parallel Processors
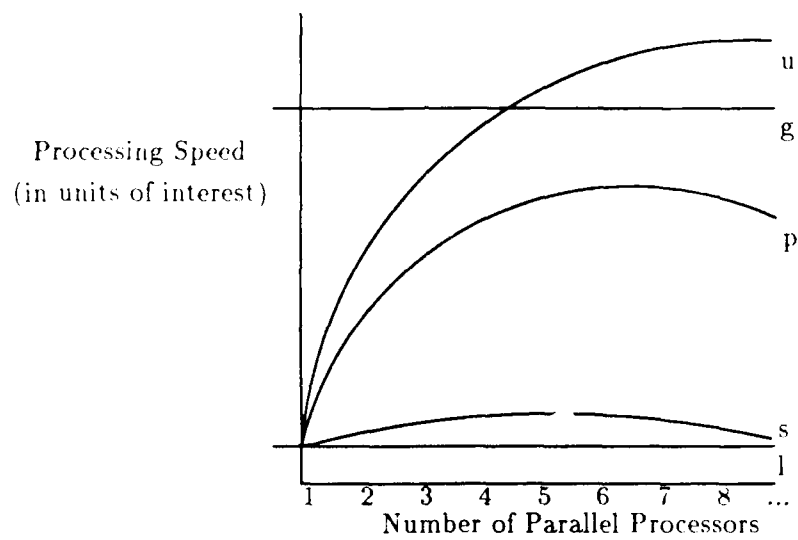
Legend: l - lower bound performance metric
s - current state-of-the-art parallel performance
p - proposed parallel architecture performance
g - goal performance
u - upper bound on proposed architecture performance

4.3 Research Investigation Steps

Based on the methodology detailed above, this research is comprised of the following steps:

- Step 1: Determine the *lower bound* performance expected in processing the RAV expert system. To do this, a "good" serial design is implemented on the iPSC/2 configured to use only one of its processors. The performance metric observed during the processing of the RAV expert system on a single processor is recorded as the lower bound (worst case) performance metric. This metric is needed to determine the speedup achieved by any of the parallel designs (e.g. Speedup = Time for Parallel Processors to Compute Result / Time for Single Processor to Compute Result).

- Step 2: Determine the *upper bound* performance expected in parallel processing of the RAV expert system. The upper bound is needed to estimate the maximum speedup achievable under a given parallel design. The theoretical maximum speedup of N (where N is the number of parallel processors) requires the very unlikely pairing of perfect load balance and no communication among processors. A more realistic estimate of the minimal communication and optimal load balance achievable is developed theoretically. Lamanna's hypercube model is adjusted to describe the performance of the RAV expert system over several iPSC/2 configurations of differing numbers of parallel processors (26:4).

- Step 3: Analyze theoretically the expected performance of the *current state-of-the-art* parallel architecture as implemented on the iPSC/2. This analysis entails considering the likely effects on the Shakley design's performance due to any upgrades added to date to the iPSC/2 as compared to the iPSC/1. Analysis results for the Shakley design over several configurations of differing numbers of parallel processors are recorded.

- Step 4: Design, implement, and analyze the performance speedup (if any) of a *new parallel architecture* on the iPSC/2 using a better match-select-act algorithm and an appropriate decomposition algorithm. This design is exercised over several iPSC/2

configurations employing different numbers of parallel processors and the performance metrics are recorded.

- Step 5: Compare the performance results produced in previous steps with respect to the RAV *real-time requirement.* AFWAL project managers are interviewed to determine how fast the RAV expert system must be processed in order for the RAV to be feasible (e.g. how fast is "real-time"?). This real-time performance metric is plotted in relation to the plotted performance metrics determined in Steps 1 through 4. Analysis of the performance metrics shows whether any one of the actual designs implemented on the iPSC/2 meets the RAV real-time performance requirement, or whether the simulated model of an optimal design can support the RAV expert system in real-time. In the latter case, such findings suggest how many iPSC/2-like processors need to be applied to achieve real-time processing of the RAV expert system given minimal communication overhead and perfect load balance. This information could add insight into the feasibility of the RAV project.

4.4 Statistical Procedures

All RAV expert system designs operate on the same data sets. The designs are validated by direct inspection of the results and of performance metrics compiled during expert system processing.

The expert system performance metric of particular interest is processing speed attained, defined in terms of the average number of rules fired per second. Timing data are also collected on the average time spent in each of the match, select, and act steps during processing.

4.5 Summary

The research methodology applied in this investigation stresses not only the development of a new parallel architecture for the RAV expert system but also the quantification of this design's contribution to RAV expert system research. The following is a summary of the RAV expert system research investigation steps:

- **Step 1**: Determine the lower bound performance metric realized by a good serial RAV expert system design.

- **Step 2**: Determine theoretically the upper bound performance metrics achievable by a parallel design implemented on the chosen architecture.

- Step 3: Determine theoretically the current state-of-the-art parallel design's performance metrics realized when running on the chosen architecture.

- Step 4: Determine the actual performance metrics realized by a new parallel design implemented on the chosen architecture.

- Step 5: Compare these performance metrics to the real-time performance requirement.

V. Step 1: Lower Bound Performance

5.1 System Design

To define the processing speedup attributable to parallel processing of a program, one must first establish the processing speed of a "good" serial design as a base of comparison (e.g. Speedup = Parallel Processing Time / Serial Processing Time). For this research, the processing speed of a "good" serial design is needed to delineate a *lower bound* of performance expected in processing the RAV expert system. Furthermore, the serial design must be implemented on hardware similar to the hardware upon which subsequent parallel designs are implemented if comparison of their processing speeds is to be valid.

The proposed serial design employs an existing serial expert system interpreter, or *shell*, which uses the Rete algorithm when performing the match-select-act cycle on any input set of rules and initial facts. The design decision to use an existing shell to support the RAV expert system allows this research investigation to take full advantage of the Rete algorithm optimization efforts afforded during the implementation of the expert system shell. Expert system shells considered as alternatives for this implementation include Inference Corporation's Automated Reasoning Tool (ART) (1), Carnegie-Mellon University's OPS5 and paraOPS5 (9), and NASA's C-Language Integrated Production System (CLIPS) (5).

The original RAV expert system was developed using ART (see Appendix A), making this expert system shell a good candidate from an RAV knowledge-base portability perspective. But ART is not an attractive alternative for this design effort for several reasons. ART is currently available only in Lisp-based and Bliss-based versions. A C-based interpreter is desired for this investigation, rather than a shell based in a symbolic language, for both program efficiency and program portability reasons (26:190). Also, the cost of acquiring new versions of ART for the purpose of this research proved prohibitive.

Carnegie-Mellon's OPS5 expert system shell series was developed and optimized by the originators of the Rete algorithm (10). But, again, the early versions of OPS5 are Lisp-based. A later parallel version, called paraOPS5, can execute in serial mode and is C-based at a macro-level. But paraOPS5 is only partially C-coded, with the Rete network

5-1

embedded directly in the National Semiconductor NS32032 machine code for realization of more speedup (17:96). Neither of these existing designs lend themselves to convenient rehosting into a serial, fully C-based OPS5 to execute on the iPSC/2's Intel 80386 chip.

NASA's C-Language Integrated Production System (CLIPS) interpreter is chosen as the expert system shell for this research investigation (33:743). As the shell's name suggests, the serial CLIPS is written in C specifically for the purposes of efficiency and portability (3:71). Of course, to exercise the RAV expert system using the CLIPS interpreter, the RAV knowledge base (rules and facts) is transliterated from its original Automated Reasoning Tool (ART) syntax to CLIPS syntax with no loss of functionality.

5.2 Detailed Design

A full CLIPS interpreter executes on the host processor of the iPSC/2. The source code of the CLIPS program, written in the C-Programming Language, is compiled without modification using the Greenhill C compiler under the UNIX/System V operating system. At system initialization, the processor is loaded with all of the production rules in the RAV knowledge base from which to build a Rete network. Then the initial facts are asserted in working memory, after which the RAV expert system is ready to execute. When production system execution is complete, performance data are collected and displayed by CLIPS (e.g. rules fired, execution time). The high-level algorithm employed by the serial design is illustrated in Figure 5.1.

Note that this algorithm is the same as the match-select-act cycle algorithm described in Chapter II. Of course, no interprocessor communication is required because the iPSC/2 is configured as an SISD computer (see Appendix B).

5.3 Implementation

Because the expert system portions of the RAV constitute the scope of this study, only the Piloting Expert System (PES) and Vehicle Control Expert System (VCES) are executed under CLIPS (see Appendix A). That is, the conventionally programmed subsystems of the prototype RAV design, such as the Route Planner and the Intelligent Vehicle

```
Procedure RETE:

    1. do while (termination state not detected);

    2. match - update the Rete network with WME

                    change applied during the last cycle

    3. select - select a production from conflict set

    4. act - apply WME change specified by selected
                    production's RHS

    5. end do;
```

Figure 5.1. RAV Serial Rete Design

Workstation, are not present to provide inputs to the PES and VCES. Consequently, the RHSs of key rules are altered to artificially introduce the values normally produced by one or more of the missing conventional subsystems. In this way, the RAV rules are kept firing to simulate progression through a reasonable air mission. The benchmark air mission consists of the execution of the entire RAV takeoff sequence of rules, the initiation of all possible RAV air maneuver rule sequences in the knowledge base, and completion of the entire RAV landing sequence of rules.

The initial facts are asserted into working memory by the firing of a *startup rule*. The startup rule has no conditions in its LHS, meaning it satisfied regardless of the state of WM. The RHS of the startup rule consists of a set of fact assertions that, when the rule is fired, load all of the facts required to activate the desired set of initial RAV rules. The subsequent RAV rule firings simulate the guidance of an aircraft through the entire takeoff sequence, a series of air maneuvers, and the entire landing sequence. It is for these subsequent rule firings that timing data are collected. For this study's benchmark RAV execution, a total of 73 RAV rules are fired in approximately 3.5 seconds by the CLIPS interpreter for an average of 20.9 rules per second. The entire RAV rule set consists of 273 rules. Therefore, the 73 rule firings observed, representing nearly 27% of the RAV rule

base, is considered a valid number to show performance difference among designs.

5.4 Summary

NASA's serial C-Language Integrated Production System (CLIPS) shell is used to execute the RAV expert system. The original ART-syntax RAV knowledge base is transliterated into CLIPS-syntax and is adapted to run without external input to allow execution of the RAV using CLIPS. A full CLIPS interpretter executes on the host processor of the iPSC/2 under the UNIX/System V operating system. Using this serial design, an average RAV processing rate of 20.9 rules fired per second is observed. This processing rate serves as the lower bound performance for execution of the RAV expert system.

VI. Step 2: Upper Bound Performance

6.1 System Design

A critical metric in any parallel architecture design is the estimate of the maximum speedup achievable. The theoretical maximum speedup of N (where N is the number of parallel processors) assumes the unlikely pairing of perfect load balance and no communication overhead. A more realistic estimate of the minimal communication overhead and near optimal load balance achievable within a given design must be developed theoretically and/or through simulation.

As Lamanna points out in her Performance Study of the Hypercube Architecture, evaluating the performance of an architecture cannot be divorced from the algorithm used (25:10). Consequently, the theoretical upper bound performance metric in this research investigation represents the maximum potential RAV expert system processing speedup realizable using the parallel expert system algorithm proposed under ideal communication and load balance conditions. The units of speedup of interest regarding the RAV expert system are the number of rules fired per second.

6.2 Detailed Design

The upper bound performance analysis presented here follows closely the timing analysis detailed in Appendix C.

Certain assumptions are made at the onset of this analysis to present an ideal computing environment for the RAV expert system executing on the proposed parallel expert system shell which employs mainly production parallelism. First, the optimal load balance is defined as an even distribution of the workload experienced by the serial algorithm amongst the parallel processors available to the parallel program. No computational overhead is introduced through parallelization. Second, the only activity other than computation on a processor that produces a time cost is interprocessor communication. No system interface overhead, such as input or output (I/O), is allowed to degrade optimal performance.

From Appendix C, the time required to complete one match-select-act cycle under the **proposed** parallel design is defined as follows:

(*Equation* 6.0)

$$O([max_{PE}([sum^{PE'sCEs} \text{ (match filter time)}] + \text{local select time})] +$$

$$\text{select compare/exchange time} +$$

$$\text{act broadcast time} + \text{local act time})$$

This equation states that the cycle time consists of 1) the maximum time spent by one of the processors updating its local Rete network and selecting a candidate rule to fire from its local conflict set, plus 2) the time for the processors to determine, through a gray-code compare/exchange, which processor has the best candidate rule to fire , plus 3) the time to broadcast the actions specified in the RHS of the rule to fire. Under the ideal condition assumptions described above, the time spent by each processor to update its Rete network, select from its conflict set, and fire the best rule's RHS actions is uniform across all processors (e.g. perfect load balance). Furthermore, the sum of the times spent processing these uniform task loads equals the total time spent processing the entire workload serially. Thus Equation 6.0 simplifies to the following, with N being the number of available processors:

(*Equation* 6.1)

$$O((\text{ total serial processing time / N }) +$$

$$\text{select compare/exchange time} +$$

$$\text{act broadcast time})$$

6.3 Implementation

The task of determining the upper bound performance for the parallel design proposed in this investigation now becomes that of acquiring actual and/or expected times for the total RAV expert system serial processing time, the average select compare/exchange

time, and the typical act broadcast time experienced on the iPSC/2 hardware. Substituting these time values into the equation presented in the previous section yields the upper bound of expected performance in terms of the total processing time required to complete one match-select-act cycle.

The total serial processing time, determined empirically using the serial CLIPS design described in Chapter V, is 3.5 seconds to fire 73 rules. The select compare/exchange time and act broadcast time each depend on the data rate of the interprocessor communication lines on the iPSC/2 and on the size of the data structure sent as a message. Because the minimum message data structures for both types of messages were known prior to actual parallel design implementation, the processing times for these activities are also determined empirically on the iPSC/2 hardware.

The data structure passed during a gray-code compare/exchange consists, as a minimum, of the integer ID of the processor passing the message and an integer value representing the firing priority, or *salience*, of its candidate rule. A total of d communications of such a structure (where d is the dimension of the hypercube) is required to ensure the structure representing the best rule-to-fire candidate is at the base processor, say node 0. The last communication required in the select step is the broadcast of the best rule-to-fire structure from the base node 0 to the other processors. The further assumption is made that a broadcast requires the same amount of time as does a node-to-node communication. A total of $d+1$ communications during the select step add cost to the total program execution time.

The data structure broadcasted during the act represents the RHS actions of the rule selected for firing. These RHS actions can consist of any number of fact assertions, fact retractions, and interface actions (such as I/O). Again, to produce ideal computing conditions for optimal processing speed, only fact assertions and fact retractions are considered in this analysis. Another simplification is that all of a selected rule's RHS actions are passed in a single data structure large enough to contain the average number of assertions and retractions specified by a typical rule in the RAV rule set. The assertions are assumed sent in the form of a typical RAV fact string and the retractions are assumed sent in the form of an index to the fact in WM to be retracted.

Inspection of the RAV knowledge base suggests that the average rule's RHS specifies approximately two fact assertions (1.46 average observed) and about two fact retractions (1.18 average observed). The typical length of a fact string to be asserted is approximately 231 characters, which represents a message size of 231 bytes for each asserted string. Integer IDs of facts to be retracted add 8 bytes each to the message size. Under the above assumptions, the typical single message broadcasted during the act step is about 470 bytes long.

The communication times required for passing of select step and act step messages on the iPSC/2 are determined empirically using a simple ring communication program that sends messages of the specified size around the nodes of the hypercube, configured as a ring. Timing data are collected as message passing proceeds. Each select message communication can be completed in 0.00424 seconds. The single act message can be broadcasted in 0.00776 seconds.

Summing the times derived above, the upper limit on the time required for the proposed parallel design to process the 73 rules fired in the RAV benchmark follows Equation 6.1:

$$(\ 3.5 \ / \ N \ ) + ((d+1) \ * \ 0.00424) + (0.00776) \ \text{seconds}$$

where $N = 2^d$ is the number of parallel processors used. Dividing the 73 rules fired by the result of this equation yields the upper bound performance, in rules per second, for the proposed parallel design.

6.4 Summary

The theoretical upper bound performance metric represents the maximum potential RAV expert system processing speedup realizable using the proposed design under ideal communication and load balance conditions. Assumptions made to simulate ideal processing conditions include the following:

- Ideal load balance suggests even distribution of the serial workload amongst available parallel processors.

- Neither computational overhead due to parallelization nor system interface overhead due to I/O is considered.

- The minimum-size data structures are assumed passed whenever communication is required, and both node-to-node and broadcast communication times are uniform and equal.

- All of the actions specified in a rule's RHS can be contained in a single data structure for communication purposes.

Communication times required for passing messages of sizes typical to the RAV are determined empirically. The equation for the upper limit on processing performance is

$$(3.5 / N) + ((d+1) * 0.00424) + (0.00776) \text{ seconds}$$

where $N = 2^d$ is the number of parallel processors used.

VII. Step 3: Current Best Performance

7.1 System Design

The performance of any new parallel architecture must be compared to that of any existing parallel architecture that is considered *state-of-the-art*. This comparison is necessary to show advancement in knowledge for fast processing of a particular application. But for the comparison between parallel designs to be valid, the designs being compared must experience similar support environments (e.g. hardware, compilers, languages). Otherwise, it is difficult to discern whether performance differences observed are due to the designs or rather to their individual support environments.

The latest architecture applied to the RAV expert system was designed by Shakley (37). The processing speed of the new parallel architecture proposed in this research investigation is compared to the speed achieved by Shakley's architecture. Shakley's program design was implemented in Lisp and on the first generation Intel Personal Super Computer (iPSC/1), a support environment different from that of the new architecture. Because actual reimplementation of the Shakley design in C and on the iPSC/2 is neither within the scope of this research nor desired, a theoretical "reimplementation" is offered instead. That is, the likely effects on the Shakley design's performance due to any upgrades to the iPSC/2 as compared to the iPSC/1 are analyzed theoretically. The theoretical performance results for Shakley's design are then used for comparison to the new parallel design's performance results.

7.2 Detailed Design

The purpose of Shakley's research investigation was to analyze and explore the feasibility of translating the RAV expert system written in ART for the TI Explorer into CCLISP for rehosting onto the Intel iPSC/1 Hypercube (see Appendix A). The focus of Shakley's study was search parallelism within a production system (37:10-12).

In his parallel RAV expert system design, Shakley exploits production parallelism. Production rules are equally distributed in a round-robin fashion across available proces-

sors in the iPSC/1 hypercube multicomputer. The rules constituting a processor's local production memory (PM) are formed into a linked-list data structure (37:26).

In ART representation, each RAV fact comprises part of a frame-like structure of facts, called a *schema*. Shakley preserves support for schemata in his design (37:41). Schemata facilitate indexing into facts in working memory (WM), thus shortening the time required to find and check the value of a given fact during the match step. Each processor hosts a copy of the entire RAV WM.

For his parallel RAV expert system design, Shakley implements on each processor an enhanced version of a serial inference engine developed by Winston and Horn (38). Enhancements to the Winston and Horn serial engine include support for schemata, for salience (priority) selection of rules, and for selection from a conflict set (or *agenda*) of rules (37:50). The ART rules and schemata are translated into a form useable by the new inference engine. The high-level algorithm employed in Shakley's design is illustrated in Figure 7.1.

Shakley organizes the iPSC/1 processors into a spanning tree for interprocessor communication (37:56). A spanning tree connection pattern is another name for the gray-code definition of near-neighbor processors in a hypercube network (24:F-1). This spanning tree connection pattern defines the parent-child relationships among processors referred to in Figure 7.1.

Shakley uses a test suite of small, prearranged sets of facts to trigger firings of subse's of rules in PM. From these firings, results are traced to confirm correctness of operation and to yield performance metrics on the speedup achieved by the parallel RAV expert system design (37:58).

Shakley acknowledges two shortcomings in his design. First, the round-robin assignment of production rules to processors creates a load imbalance. Although all processors host the same total number of production rules, these rules are of varying length and computational complexity, thus causing an imbalance. Second, the test suites of prearranged facts are too small to take significant advantage of parallelism. The small test sets have too little span of effect on the production rules in PM, thus limiting the potential production

```
Procedure HYPER:

    1. do while (termination state not detected);


    2. parallel match
                    - each processor waits to receive WME change
                        from its parent (except root node)

                    - each processor sends WME change to its child
                        in the tree (if any)

                    - each processor adds WME change to its local
                        copy of the WM

                    - each processor matches the rules in its PM
                        against the facts in its WM


    3. global select
                    - each processor waits to receive the selected rule
                        from its child (if any)

                    - each processor adds the rule received from its
                        child to its conflict set

                    - each processor sends selected rule to its parent
                        on tree (not root);

                    - root processor holds production to fire after its
                        select is done


    4. global act
                    - root processor sends to its child the WME change
                        specified by selected rule's RHS


    5. end do;
```

Figure 7.1. RAV iPSC/1 Hypercube Design (37:47)

parallelism.

7.3 Implementation

On the iPSC/1, Shakley's parallel design is exercised using one of two subsets of the RAV rule base, which he terms "small" and "large" rule bases. For comparison purposes, the faster performing small rule base configuration is analyzed. Shakley's expert system, using the updated Winston and Horn inference engine, fires an average of 1 rule every 11 seconds in a serial mode. Speedups are realized in parallel mode, with a peak performance of about 0.5 rules per second experienced in a 16-node configuration (see Figure 7.2).
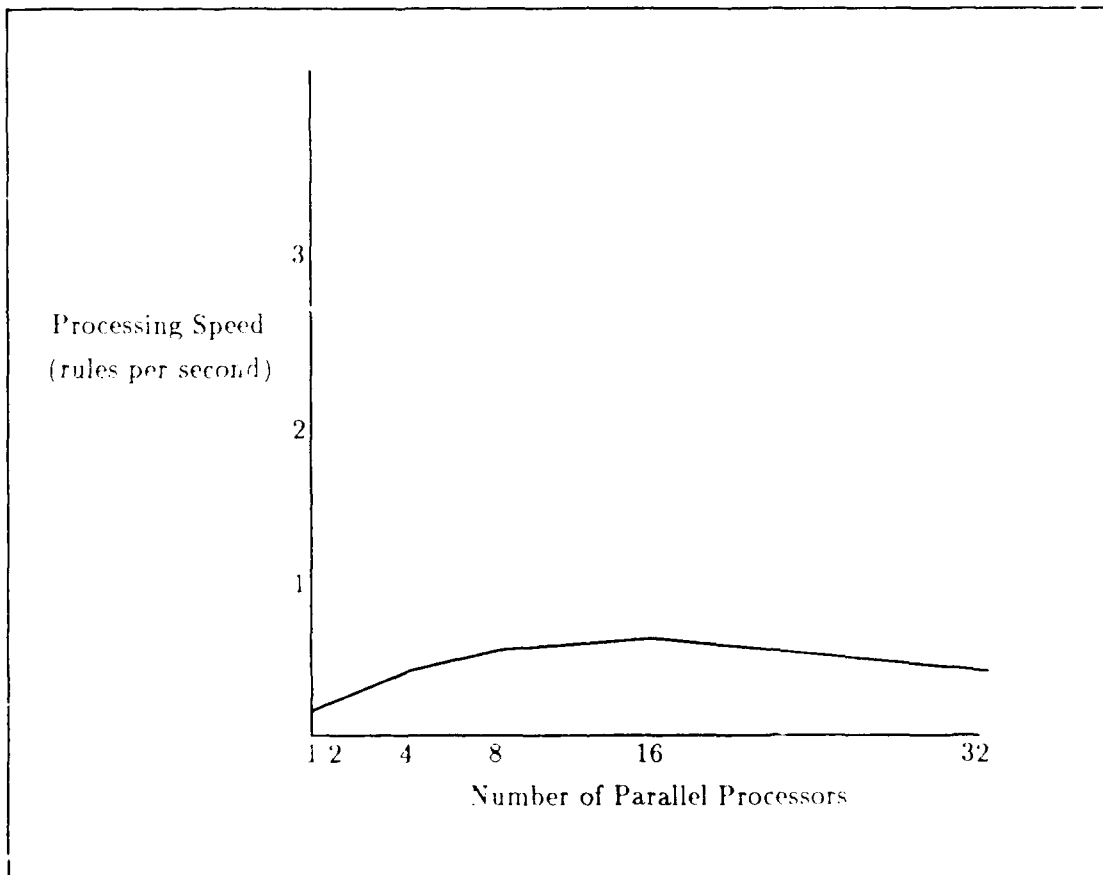
Figure 7.2. iPSC/1 RAV Performance Results (37)

Uploaded to the iPSC/2, the Shakley design will experience immediate performance improvement due to the raw processing power of the iPSC/2 chip technology compared to

that of the iPSC/1. At best, a processing speedup of 4 times can be expected due to use of the iPSC/2's 80386 chip versus the iPSC/1's 80286. A speedup can also be expected in communications capability. Even though the iPSC/2's peak message passing rate of 2.8 Mbytes per second ((21:1-11)) is hardly discernable from the iPSC/1's rate of 2.5 Mbytes per second ((22:1-22)), the iPSC/2 communications scheme will generate some speedup, because the passing of a message between two nodes on the iPSC/2 does not interrupt the processing on intermediate nodes, as is the case on the iPSC/1.

Experiencing the maximum possible beneficial effect of the processing and communications upgrades in the iPSC/2, the Shakley parallel design's performance can be expected to improve to 8 rules per second, at best. This analysis is abandoned at this point, because the further possible performance improvements in the Shakley design realizable due to the iPSC/2's broadcast capability and to a hypothetical reimplementation of the design in C will certainly not be enough to bring the design's performance up within the 20-rules-per-second range of the serial CLIPS lower bound performance. It is apparent that the simple Winson and Horn inference engine, even executed in parallel, cannot compete, in terms of processing speed, with the state-saving Rete algorithm employed in CLIPS.

VIII. Step 4: Parallel Rete Performance

This chapter describes in detail the proposed hypercube expert system shell, called HyperCLIPS. The source code (Appendix D), programmer's manual (Appendix E), and user's manual (Appendix F) offer more detail for the interested reader.

8.1 System Design

The proposed design implements a parallel production system interpreter which uses the Rete match algorithm. A C-based version of the serial C-Language Integrated Production System (CLIPS) interpreter is adapted to run in parallel on the Intel iPSC/2 (33:7:13). The system is configured to take advantage of production parallelism, enhanced by action parallelism. The goals of this system are the following:

- support the speedup features inherent in the Rete network as much as possible (e.g. state saving, one-time comparisons)

- minimize detrimental communication overhead, especially in the match step

- distribute productions in such a way that the workload is well balanced among the available processors

- assign productions that are expected to be activated at the same time to different processors to enhance parallelism

8.2 Detailed Design

Each active processor supports a full production system interpreter. At system initialization, each processor is parsed a subset of the productions in the RAV knowledge base from which to build a local Rete network. With this static decomposition approach, no interprocessor communication is required during the match step because all of the nodes needed to process state updates of a production are local to the production's host processor. Furthermore, each processor's interpreter performs a local select step, picking its local candidate production for overall system firing. The local select step is performed without interprocessor communication.

The first interprocessor communication occurs when the processors must compare and exchange their locally selected productions to determine which of these productions is to be selected for firing. Using the gray-code compare/exchange paradigm for 2^d processors connected in a hypercube (see Appendix B), a total of d compare/exchanges must be performed before the best candidate production is guaranteed to be at the root processor (24:F-1).

Once the root processor has the globally selected production, the working memory element (WME) change specified by the RHS of that production is broadcasted to all processors. This WME change is the input to each of the processors that triggers the subsequent match step. The entire match-select-act cycle repeats in this fashion until a termination condition is detected or no productions are matched. Figure 8.1 illustrates the high-level algorithm implemented in this design. Note that the few communications required in this design occur only in the select and act steps.

This design exploits all match-select-act cycle parallelism discussed in Chapter III. Parallelism within the match step is achieved when all processors update their Rete subgraphs concurrently. Select step parallelism is possible because each processor performs its select on its local conflict set upon completion of its local match, thus creating the potential for multiple processors to be in their respective local select steps concurrently. The processing within the match step and the select step of the same cycle can overlap when the local match on one processor completes and triggers the start of the local select step before one or more other processors complete that same match step locally. Act step parallelism occurs in that, when the globally selected production is fired, the working memory element change is broadcast to the waiting processors, triggering their concurrent match steps. Overlap of the act step of one cycle and the match step of the next cycle is conceivable. That is, the WME change is allowed, by design, to arrive at some processors before it arrives at others, although "broadcast" implies the change arrives at all processors simultaneously. Whether or not changes arrive simultaneously, local match steps begin as soon as the WME change is received.

Procedure HYPER-RETE:

1. do while (termination state not detected);

2. parallel match
   - each processor receives WM change from root processor
   - each processor updates its local Rete network

3. parallel local select
   - each processor selects a production from its local conflict set

4. global select
   - processors perform gray-code compare/exchange
   - root processor holds production to fire when compare/exchange done

5. broadcast global act
   - root processor broadcasts WM change specified by selected production's RHS

6. end do;

Figure 8.1. RAV Hypercube Rete Design

8.3 Implementation

A host program executing on the front-end host processor of the iPSC/2 provides the user interface to the HyperCLIPS shell. The host prompts the user for the desired cube dimension, the application knowledge base, and the desired run time options (see Appendix F). The node program, which is executed on each active hypercube processor, downloads from the host processor the entire initial working memory fact set and its partition of the total production rule set. Each node then initializes and executes its local version of CLIPS.

To implement HyperCLIPS, two adaptations to the serial CLIPS shell are required: 1) the global select gray-code compare/exchange capability and 2) the global act broadcast capability (from Figure 8.1). For both of these communication activities, the main design challenge is the choice of data structure to pass as a message.

From an implementation-independent perspective, it seems that the typical message consists of a structure representing one complete rule. For the select compare/exchange, the structure passed by a processor node represents the top rule on the node's local conflict set. For the act broadcast, the structure passed represents the one rule selected globally for firing.

Unfortunately, the data structure for a CLIPS rule is a multi-directional, multipli-linked list of multivariate structures. Because the many pointers employed in such a structure on a given processor have meaning only in the context of that processor's local memory, the rule structure must be par d and the actual structure values put into an array to be sent to other processors. Once received, such an array must again be parsed, reassembled into CLIPS rule form, and processed.

Parsing, passing, and reassembing of rules is not chosen for the HyperCLIPS implementation for several reasons. First, the raw complexity of parsing and reassembling a rule structure is prohibitive. Second, the communication time required to send such a potentially large message, added to the computation overhead time required for parsing and reassembling the rule structure, does not suggest efficient use of processing time. Third, a rule structure in CLIPS is tightly entwined in the local Rete network, including data

describing the rule's dependencies among other rules in its network. Passing of a complete rule, then, entails passing large portions of a processor's local Rete network, including data that is probably unneeded for processing at another processor and possibly even incorrect and corrupting if processed at another processor.

The chosen select compare/exchange design approach is to pass not the entire rule, but a structure consisting only of the priority, or *salience*, of the top rule on a processor's conflict set and the ID of the processor holding the particular candidate rule for firing. After the compare/exchange is complete, processor 0 holds the salience of the rule to fire and the ID of the processor where that rule resides. Processor 0 then broadcasts this information to all processors, triggering the processor holding the globally selected rule to enter a *master* processing state while the other processors enter a *slave* processing state for the upcoming global act broadcast.

The act broadcast design approach is for the master processor to send as messages only the *RHS actions* of the globally selected rule, rather than the entire rule structure. Because the firing of any rule ultimately results in either no action on WM or a series of working memory element additions and/or retractions (see Chapter III), each fact addition and retraction is broadcast as it is about to be processed on the master processor. Consequently, the same addition or retraction is received and processed on the slave processors, lagging the length of the broadcast time behind the master's processing state.

A fact string specified for assertion in the selected rule's RHS must be built by parsing the RHS's CLIPS fact structure, but this parsing task is much less complex than parsing the entire rule structure. An assert message consists of this fact string and a tag specifying an assert operation is required. A retraction specification need only consist of the integer ID of the fact to be retracted, because all processors maintain identical copies of working memory with identical working memory element IDs. A retract message consists of this working memory element ID and a tag specifying a retract operation is to be performed.

Unfortunately, execution of the RAV expert system on HyperCLIPS resulted in slow down compared to the serial CLIPS implementation. A two-node configuration produced an average of only 12.6 rules fired per second, followed by 5.11 rules per second using four

nodes and 2.01 rules per second using eight nodes. These results, although discouraging, are in keeping with the timing analysis conclusions presented at Appendix C.

Note from Appendix C that, for the HyperCLIPS design to produce speedup, the following two conditions must exist:

1. The RHSs of all production rules must affect many condition elements (CEs), or predicates, in the LHSs of many other rules. Ideally, the average number of CEs affected would equal the number of processors available.

2. The production rules must lend themselves to fortuitous assignments to unique processors. Specifically, the production rules containing CEs that initiate processing along non-interacting Rete network paths should be assignable to unique processors.

Inspection of the RAV expert system execution suggests that the first condition is not adequately met. The average number of facts affected per rule firing is observed to be less than three, with an observed range between 1 and 13. These few affected facts represent less than 0.5% of a WM that averages approximately 700 total facts during execution. Furthermore, the facts changed due to rule firings are in turn observed to affect an average of just over four rules each, with an observed range between 1 and 18. These four rules represent only 1.5% of the 273 rules present in the RAV knowledge base. The small span of effect per rule firing severely limits the benefits attainable through parallel processing of the RAV expert system using production parallelism.

HyperCLIPS, as implemented in this research investigation, has no mechanism to take full advantage of the second condition for producing speedup described above and in Appendix C. To the extent that the RAV rule base lends itself to fortuitous assignment of rules to unique processors, HyperCLIPS leaves the burden of rule assignment to the user. No algorithmic process is currently available to recognize dependencies and relationships among rules in an expert system's knowledge base and to use dependency data to drive near-optimal assignment of rules to processors. Consequently, the load imbalance introduced by the user's inability to assign rules to processors in a way that takes advantage of production parallelism further limits the processing speed observed for the RAV expert system using HyperCLIPS.

8.4   Summary

The proposed hypercube expert system shell is called HyperCLIPS. HyperCLIPS employs a full serial CLIPS interpretter executing on each available processor in the iPSC/2 multicomputer. Each iPSC/2 processor is parsed a partition of an application's total rule set, and performs a normal CLIPS match step on the local rule set. Then a gray-code compare/exchange of each processor's highest-salience rule (tagged with the rule's home processor) transfers the salience of the rule to fire to the root node. After the root broadcasts the salience and identification information for selected rule, the home processor for the selected rule becomes the action master processor. The master drives the other slave processors to mimic its fact assertions and retractions as the selected rule's RHS is fired. Execution of the RAV expert system using HyperCLIPS results in slow down compared to the serial CLIPS implementation. The slow down results 1) from the small span of effect produced when the RHSs of typical RAV rules are fired and 2) from the lack of a mechanism which recognizes RAV rule inter-dependency data and uses this data to drive assignment of rules to processors.

IX. Step 5: Performance Comparison Findings

In this chapter, the performance results produced in Chapters V through VIII are compared to the RAV expert system real-time processing requirement.

Real-time processing of the RAV is defined through interview with AFWAL project managers (7). The current ART RAV implementation on the TI Explorer Lisp machines executes at a rate of 15 to 30 rules per *simulated vehicle second*. The simulated vehicle second is a unit derived to account for the processing delay introduced when the expert system must wait for the vehicle simulator to produce needed vehicle control parameters. One simulated vehicle second equals approximately 2.5 actual seconds. Therefore, real-time processing of the RAV entails firing rules at a rate roughly between 37 and 75 rules per second. An estimate of 50 rules per second is plotted as the real-time RAV performance requirement in Figure 9.1.

The plotted performance metrics produced during analysis of lower bound performance, upper bound performance, and parallel Rete performance complete the remainder of Figure 9.1. Note that the current best performance is omitted from Figure 9.1 because it falls entirely below the lower bound.

The lower bound performance experienced by the serial CLIPS design is impressive, although it does fall short of real-time. CLIPS performs well for the RAV expert system application because of the Rete state-saving feature (from Chapter III). CLIPS's state-saving takes full advantage of the very small rate of change of working memory observed in Chapter VIII, which is why CLIPS soundly outperformed the Shakley parallel design in terms of rules fired per second.

The upper bound performance metric is plotted for the shown number of processors ($N = 2^d$) and derived from the equation for upper bound offered in Chapter VI. The upper bound suggests that it is reasonable to expect to achieve real-time processing of the RAV expert system on the iPSC/2 using HyperCLIPS under ideal conditions. However, the model in Chapter VI does not take into account the dependencies between rules in the RAV knowledge base that limit the ability to achieve perfect load balance.
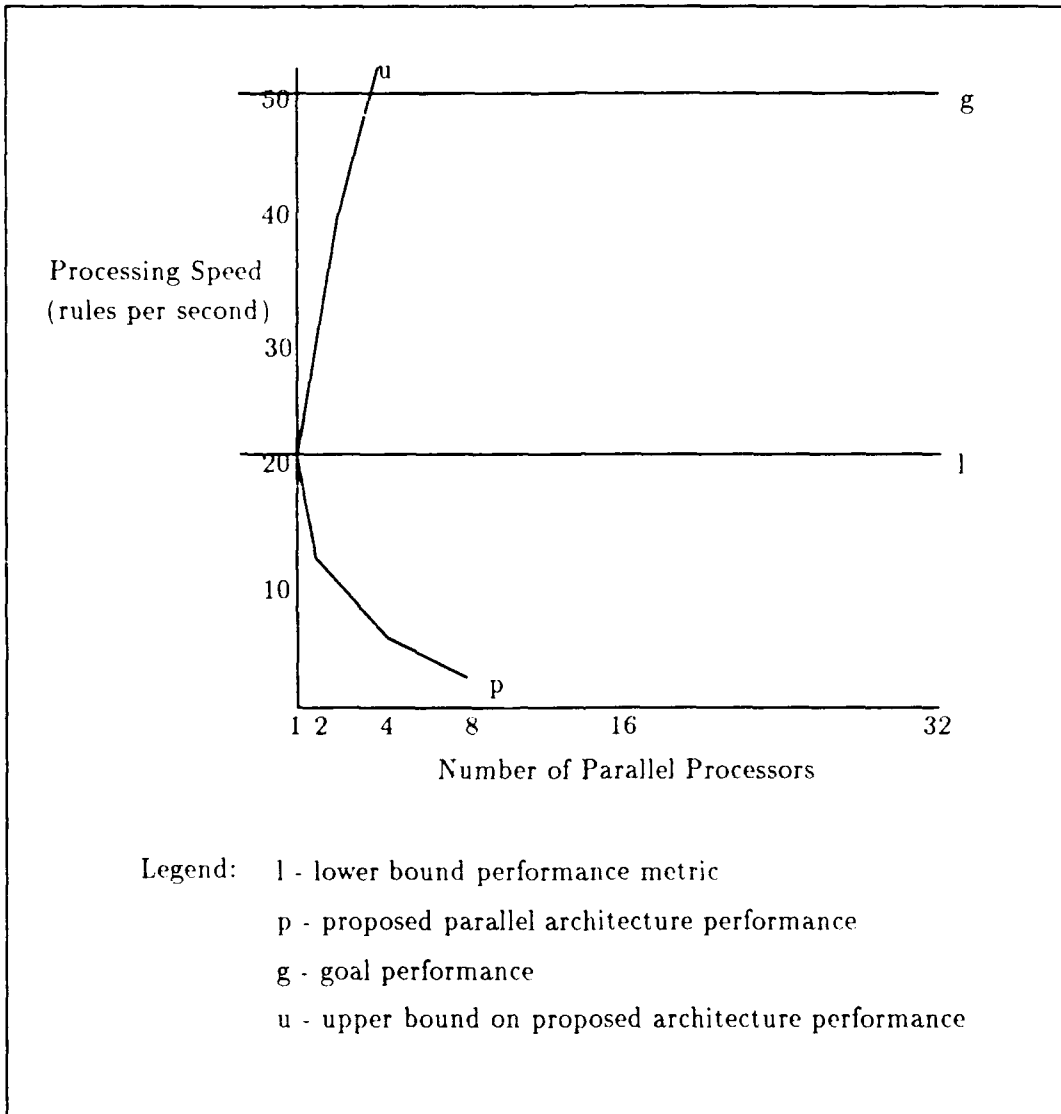
Figure 9.1. RAV Expert System Performance Spectrum

The performance of HyperCLIPS executing the RAV expert system is also plotted in Figure 9.1. The slow down experienced by HyperCLIPS suggests, at best, that an algorithmic mechanism is required to assign input rules to available processors in a way that optimizes potential production parallelism. Worst case, the slow down suggests that the RAV expert system itself does not exhibit the inter-rule dependencies necessary to allow a significant amount of production parallelism.

X. Conclusions and Recommendations

10.1 Research Conclusions

Parallel processing is a promising approach to achieving real-time processing of expert system software. The keys to improving parallel processing performance are reducing communications overhead and balancing task load. The major factor in both of these goals is the proper choice of a problem decomposition.

This research goes beyond just producing a new parallel architecture design. The performance results of this design are quantified in relation to the lower and upper performance bounds, the current state-of-the-art design's performance, and the required real-time performance. This approach not only adds validity to the performance results of the new design but also exposes the level of maturity the RAV expert system research has achieved as a consequence of this design. This performance quantification methodology serves as a template for other researchers performing parallel computer architecture design as applied to any application.

10.2 Research Recommendations

The findings analyzed in Chapter IX suggest that research into parallel processing of the RAV expert system is still in its infancy. The speedups realized using serial CLIPS (Chapter V) support the continued use of state-saving algorithms, such as Rete, to process the RAV. But the characteristics of the RAV observed in Chapter VIII suggest that the system lends itself to very limited potential speedup due to production parallelism. Therefore, RAV expert system research is perhaps better served by approaching parallel processing from the standpoint of node parallelism, possibly using a shared-memory multiprocessor (see Chapter III).

The critical component missing during this research investigation is an algorithmic mechanism to assign production rules to available parallel processor nodes. Such a mechanism will parse rules, recognize dependencies among rules that promote production parallelism during rule firings, and use this information to assign rules to available processors.

10-1

The results of the timing analysis in Appendix C suggest that the assignment algorithm is driven by the span of effect of actions in the RHSs of all available rules.

An "optimal" assignment algorithm potentially poses an NP-complete problem, an fast execution of the algorithm may require parallel processing itself! But an assignment algorithm is critical for designs, such as HyperCLIPS, that depend on production parallelism to realize processing speedup.

Further exercise of the HyperCLIPS expert system shell, using applications more amenable to production parallelism, is recommended. HyperCLIPS also serves as a possible tool for the development of an optimal assignment algorithm described above.

10.3 Summary

Parallel processing of the RAV expert system is still in its infancy as an area of research. Processing of the RAV expert system using serial CLIPS executes at an impressive speed due to the state-saving Rete algorithm. But the HyperCLIPS implementation performs poorly for the RAV application due to the limited potential production parallelism characteristic of the RAV and due to the need for a rule assignment mechanism. Further exercise of HyperCLIPS using applications with significant potential production parallelism is recommended.

Appendix A. *Robotic Air Vehicle Background*

The Robotic Air Vehicle (RAV) is a concept under exploration by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Wright Aeronautical Laboratories (AFWAL). The concept is to create an unmanned air vehicle capable of autonomous operation. The RAV must be able to perform basic piloting skills as well as passive terrain following, terrain avoidance, obstacle avoidance, and autonomous navigation. The mission of such a vehicle would consist of intelligent reconnaissance or attack of high risk, heavily defended targets. A contract was awarded to Texas Instruments Incorporated (TI) in September 1985 to develop a system architecture and to demonstrate the feasibility of some of the key components of such a system (28). TI completed its system development and demonstration in June 1988 (12:1).

The final system architecture developed by TI is shown in Figure A.1. The RAV system software consists of six top level modules: the Route Planner, the Piloting Expert System (PES), the Vehicle Control System (VCS), the Spatial Database, the Intelligent Vehicle Workstation (IVW), and the Natural Language Menu (NLMenu) system (12:6). This architecture includes several expert subsystems linked to a central controlling agent, the Piloting Expert System (PES). The expert subsystems consist of the following: a Passive Navigation module for estimating the current platform location on a digital map and for generating a terrain model for terrain following; an Airspace Expert System for three dimensional situation awareness; a Route Planner for generating the RAV flight path; a Vehicle Control System for translating high level flight directives into stick and throttle manipulations. The PES coordinates the activities of these subsystems and is the final arbiter of RAV responses to the environment (12:15).

During typical system operation, mission and flight data is entered via NLMenu or Voice and sent to the PES. The PES sends the waypoint coordinates, including target location, to the Route Planner, which calculates a path through the waypoints. As the PES executes the route plan, it responds to inputs from NLMenu and IVW vehicle simulation and queries the Spatial Database. Spatial Database queries activate both the Passive Navigation subsystem and the Airspace Expert System. These inputs are used to pilot
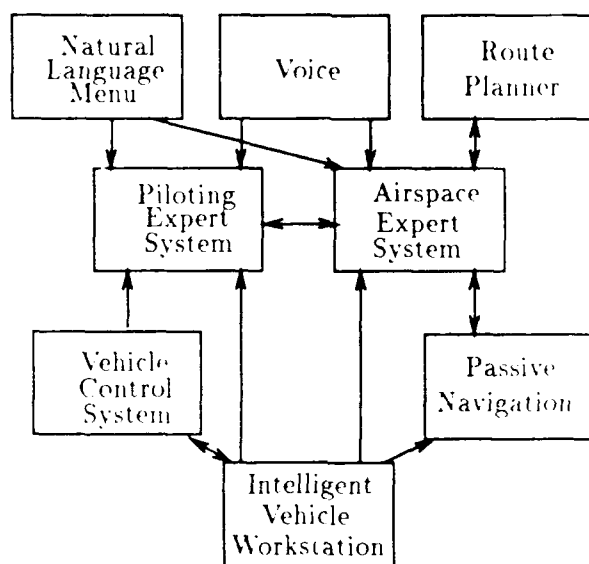
Figure A.1. RAV System Architecture (11:7)

the vehicle and to execute the mission plan. To perform complex maneuvers such as steep turns and rolls, the PES issues airspeed, altitude and heading commands to the VCS.

Figure A.2 shows the physical placement of the RAV subsystems with respect to their hardware requirements. The RAV system software resides in a hardware configuration of four Texas Instrument Explorer II Lisp Machines and one Digital Equipment Corporation (DEC) MicroVax II. All machines are linked together via a Local Area Network (LAN). A message scheme, known as Post Office, facilitates communication among the machines and their resident software subsystems. The Explorers support all of the AI software and the vehicle control and simulation software. The MicroVax II performs the mathematic computations required for con entional software systems (12:9).

Knowledge bases for these expert systems were developed using conventional knowledge engineering techniques. The methodology was based on the approach used to train fighter pilots. The knowledge acquisition process followed a series of qualification training sessions and evaluation check-rides. Using this technique, the system capabilities increased

Note: "En" denotes Explorer machine
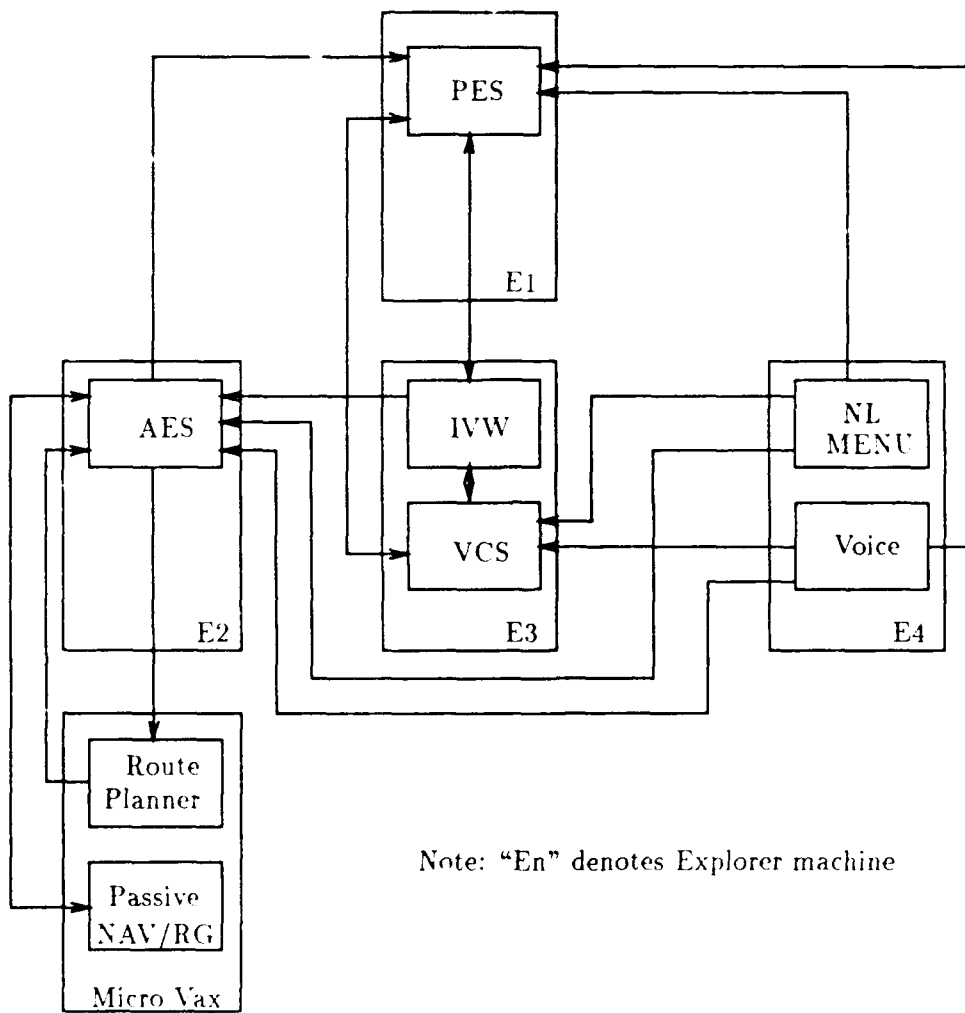
Figure A.2. RAV System Configuration (11:10)

A-3

in an ordered fashion. The knowledge bases became hierarchical as new concepts were built on old skills. Each level of competency was validated before moving to the next training level (12:28).

The knowledge representation used to develop the knowledge bases for the expert systems is a combination of TI Dallas Inference Engine (TIDIE), Automated Reasoning Tool (ART), and/or Lisp representations. These three tools are used to represent the knowledge bases at three different levels of abstraction.

TIDIE was developed for the RAV project to provide a readable, high-level knowledge representation for piloting rules. The TIDIE representation consists of OBJECTS, representing the aircraft state variables; NEEDS, representing the goal to be met; and PLANS, representing how the goal is to be met (12:12). NEED and PLAN macros expand these high level structures into lower-level ART components.

Inference Corporation's ART is a commercial artificial intelligence shell. It provides an inference engine for rule-based reasoning. ART also uses the Rete algorithm for pattern matching. The ART representation consists of typical expert system RULES and facts organized into frame structures, called SCHEMAS (12:13). A Lisp-based version of ART is employed in the RAV architecture.

All three levels of knowledge representation are compatible. TIDIE is implemented in ART rules and Lisp functions, thus allowing free intermixing of TIDIE and ART constructs in a single knowledge base source file. Lisp was used directly for low level command modules like the VCS (12:12).

The TI architecture has demonstrated the feasibility and practicality of a Robotic Air Vehicle (RAV). The entire ensemble of subsystems has proven the effectiveness of distributed cooperating expert systems. The RAV provides a performance benchmark for a near real-time control system (12:35). Unfortunately, current hardware support for symbolic computing such as that used by the PES is not adequate to permit real-time control of a vehicle by an expert system (12:33).

In the final report, TI recommended three main follow-on research directions to be pursued: first, investigate how the RAV prototype would handle a more high fidelity

simulation; second, investigate real-time expert systems, distributed expert systems, and maintenance and verification of expert systems; and, third, extend the piloting capabilities currently employed by the RAV prototype (12:35).

Appendix B. *Parallel Processing Architectures*

Computer architectures may be classified by many methods, or *taxonomies*. Flynn's Taxonomy is based on the concepts of instruction stream and data stream (8:1901). An instruction stream is a sequence of instructions performed by a computer. A data stream is a sequence of data used to execute an instruction stream.

The Single-Instruction stream, Single Data stream (SISD) category includes most serial computers. The Single-Instruction stream, Multiple Data stream (SIMD) category includes processor arrays. The Multiple-Instruction stream, Multiple Data stream (MIMD) category contains most multiprocessor systems. Finally, no computers in common use today belong in the Multiple-Instruction stream, Single Data stream (MISD) category (34:16). This leaves SIMD and MIMD as the two main categories of parallel processors.

As implied by the label, an SIMD parallel system has multiple processors operating the same instruction synchronously on separate data streams. Examples of SIMD architectures are the ILLIAC IV and Connection machines. The MIMD parallel system has multiple processors capable of operating on multiple data streams with different operations asynchronously. Examples of MIMD architectures are the Butterfly and the iPSC hypercube (20).

There are many possible processor organizations, or *interconnection methods*, for parallel architectures. The following are examples of commonly used interconnection methods (34:25-29):

- *Mesh network* - Processors are arranged into a q-dimensional lattice. Communication is possible only between neighboring nodes, thus interior nodes can communicate with 2q other processors (see Figure B.1).

- *Pyramid network* - Processors in a pyramid network of size p form a complete 4-ary rooted tree of height $log_4 p$ augmented with additional interprocessor links that allow processors in every tree level to form a two-dimensional mesh network (see Figure B.2).

- *Shuffle-Exchange Network* - This network consists of $n = 2^k$ processors and two kinds of connections: shuffle and exchange. Exchange connections link processors whose numbers differ in their least significant bit. The shuffle connection links processor i with processor 2i modulo (n-1) (see Figure B.3).

- *Butterfly Network* - $(k+1)2^k$ processors are divided into k+1 rows, or ranks, containing $n = 2k$ processors each. Each processor has four connections to other processors (see Figure B.4).

- *Hypercube (Cube-Connected) Network* - A cube-connected network is a butterfly with its columns collapsed into single processors. The network consists of $n = 2^k$ processors forming a k-dimensional hypercube. Two processors are adjacent if their labels differ in exactly one bit position (see Figure B.5).
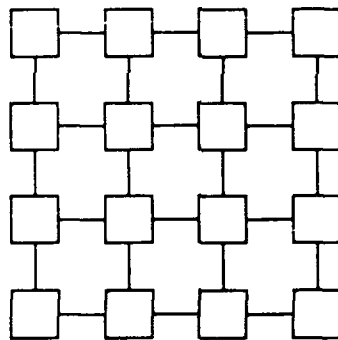
Figure B.1. 2-D Mesh Network (34:26)

Another important feature of a parallel architecture is the memory organization employed. Most reported parallel SIMD architectures assume a shared global memory among all processors (34:30). MIMD architectures, however, can be further classified as multiprocessors or multicomputers based on the memory organization.

An MIMD multiprocessor is characterized by shared memory among the processors. In a tightly coupled multiprocessor, these processors work through a central switching mechanism to reach the shared global memory. A loosely coupled multiprocessor is also
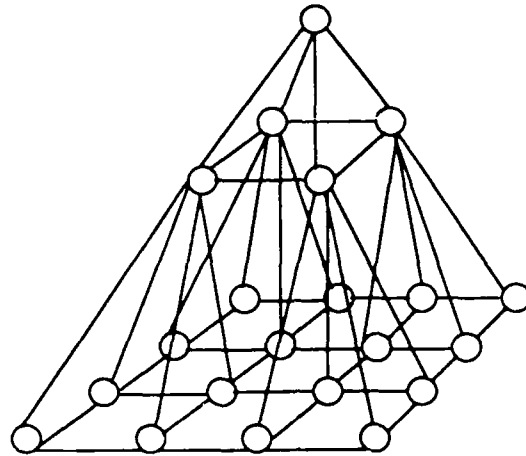
Figure B.2. Size 16 Pyramid Network (34:27)
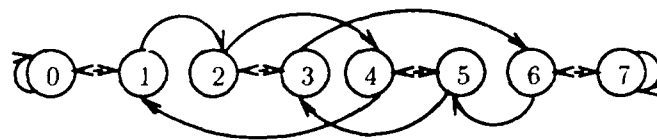
Figure B.3. 8-Node Shuffle-Exchange Network (34:27)
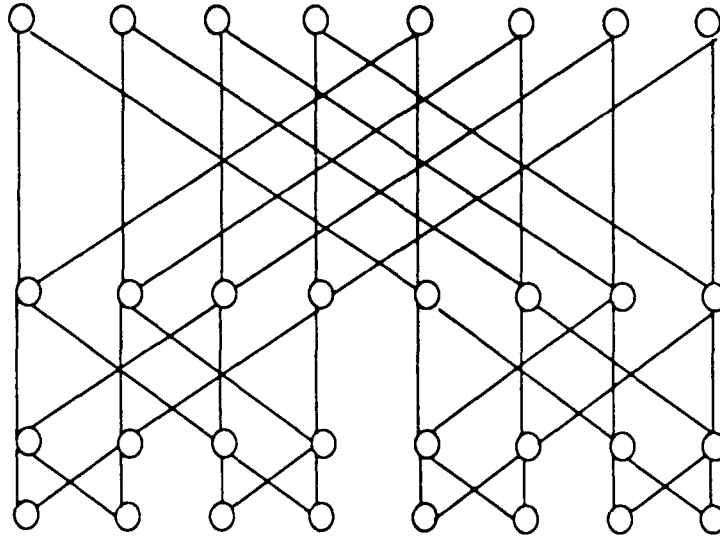
Figure B.4. 32-Node Butterfly Network (34:28)
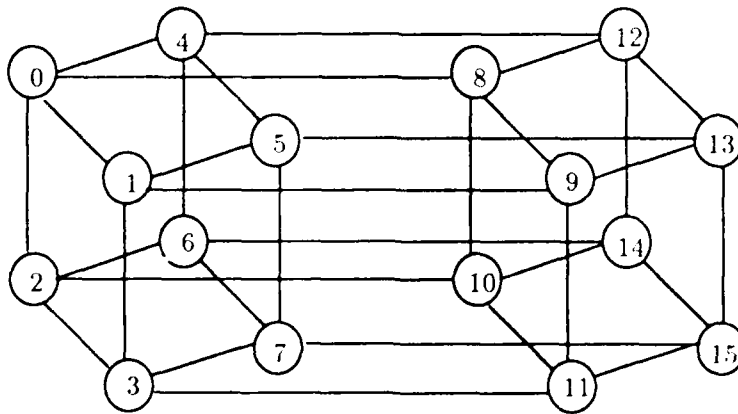
Figure B.5. 4-D Hypercube Network (34:28)

characterized by a shared address space, but this shared address space is formed by combining the local memories of the processors (34:35-38).

An MIMD multicomputer has no shared global memory. Instead, each processor has its own local memory. Process cooperation occurs either through message passing or through memory shared between pairs of processors (34:41).

Two parallel architectures are of interest in this research investigation. The first is a network of four Texas Instruments (TI) Explorer II Lisp machines. The second is the second generation Intel Personal Supercomputer (iPSC/2). Each is an MIMD architecture.

The first host architecture of the RAV expert system consists of a network of four TI Explorer II Lisp machines. They form a loosely coupled system. Each system has its own local memory and a common bus structure connects the systems. All systems share a central file server. Each of the four systems are very powerful Lisp processors.

The second architecture of interest is the Intel iPSC/2 hypercube. This system can be configured with up to 128 processor elements (PE). The iPSC/2 is a multicomputer made up of Intel 80386 processors. The flexibility of this system, along with its ready accessibility, make it a useful tool for this study.

Examples of the previously discussed architectures are given in Figure B.6. Only the TI Explorer Lisp machine and the iPSC/2 are evaluated as part of this research investigation due to availability.

```
SIMD Processor Arrays:      ILLIAC IV
                            Connection Machine
                            Burroughs PEPE
                            IBM GF-11
                            ICL/DAP


MIMD Multiprocessors:       C.mmp (Carnegie-Mellon)
   (tightly coupled)        Encore Multimax
                            Sequent Balance 8000


MIMD Multiprocessors:       Cm* (Carnegie-Mellon)
   (loosely coupled)        BBN Butterfly


MIMD Multicomputers:        Intel iPSC
                            Ametek S/14
                            NCUBE/10
```

Figure B.6. Summary of Architectures (19, 34:1350,31-41)

Appendix C. *Timing Analysis of RAV Expert System*

One key to the amount of processing required to perform a state update in a Rete network during the match step is the number of co..dition elements (CEs) present near the top of the network. Each CE may be considered an initiator of a path or set of paths into the Rete network. As a network path(s) initiator, a CE defines a fixed number of productions that can potentially be affected by the information flow down that path(s). A CE's span of effect is important to this timing analysis because match speed-up available from production parallelism is proportional to the average number of affected productions (14:54).

When a working memory element (WME) change is input at the root node of a Rete network, the information flow resulting from the processing of that WME change passes through one or more CEs and down through some number of nodes in the network. Call the time from the initial receipt of a WME change at the root node to the termination of information flow caused by that WME change the *match filter time*. Call the time required by the interpreter to select a production from the updated conflict set the *local select time*. Similarly, call the time required by the interpreter to evaluate the RHS of the selected rule and to send it to the root node in the network the *local act time*.

The term "local" implies that the specified activity is performed using the data (i.e. conflict set, Rete network) that are saved in the memory that is local to the processor running the interpreter. Although this definition is assumed for serial processors, it is useful later in the context of multicomputers.

Consider the very simple exa.nple case in which the WME change input at the root node of the Rete network matches only one CE at a subsequent t-const node. A production system interpreter implemented on a serial processor will complete a match-select-act cycle in the amount of time defined by the following (in order-of notation):

$$O \ (\ \text{match filter time} + \text{local select time} + \text{local act time}\ )$$

In the opposite extreme case, assume a WME change input at the root node matches all of the CEs at the subsequent t-const nodes. A serial production system interpreter will complete one match-select-act cycle in the following time:

$$O([sum^{allCEs} \text{ (match filter time)}] + \text{local select time} + \text{local act time })$$

Now cons er the production system implemented on a multicomputer configured as in the design presented in this research investigation. Call the time required by processors to perform the gray-code compare/exchange of productions in the select step the *select compare/exchange time*. Call the time required to broadcast to all other processors the production selected for firing the *act broadcast time*.

Returning to the simple example case in which the WME change input matches only one CE, the parallel production system interpreter will complete one cycle in the following time:

$$O([max_{PE} \text{ (match filter time + local select time)}] +$$

$$\text{select compare/exchange time} +$$

$$\text{act broadcast time} + \text{local act time })$$

This equation states that the processor, or *processing element* (PE), that requires the most time to update its local Rete subnetwork and to select a production from its local conflict set will make the processors that have completed these steps wait before entering the select compare/exchange step. Of course, the processor that hosts the Rete subnetwork that holds the single affected CE will experience the maximum match filter time. Note that the local act time does not appear as max_{PE} (*local act time*) because all processors are assumed to enter the local act step synchronously as a result of the act broadcast and because the local act step will require the same amount of time on each processor.

omparing the results of serial versus parallel processing of the single-affected-CE case, both systems experience the same match filter time and the same local act time. The parallel system could experience a slight savings in local select time because the affected Rete subnetwork will always have a conflict set smaller than or equal in size to

that of the whole Rete network on the serial system. Consequently, the processor on the parallel system that hosts the affected Rete subnetwork could potentially perform its local select faster than could the processor in the serial design. But this potential local select time savings is sure to be overcome by the communication cost incurred during select compare/exchange and act broadcast. This finding suggests that the multicomputer design is not as fast as the serial processor design when processing WME changes that affect very few CEs.

Considering the example case in which the WME change input matches all CEs, the parallel production system interpreter will complete one cycle in the following time:

$$O(\{max_{PE}[sum^{PE'sCEs} \text{ (match filter time + local select time)]}\} +$$

$$\text{select compare/exchange time } +$$

$$\text{act broadcast time + local act time)}$$

Again, some processor (PE) will require more time than the others to process all of the Rete subnetwork updates required by its many CE matches and to select a production from its local conflict set The other processors will wait for that processor to finish before entering the select compare/exchange step.

Comparing the results of serial versus parallel processing of the all-CE-affected case, both systems experience the same local act time. The parallel system could experience a substantial savings in match filter time if the workloads distributed to the available processors are of equal computational complexity. Assuming this good load balance, the sum of the match filter time saved and the local select time saved in parallel design can reasonably be expected to be greater than the sum of the select compare/exchange time incurred and the act broadcast time incurred. This finding suggests that the multicomputer design can be expected to perform faster than the serial processor design when processing WME changes that affect many uniquely assignable CEs.

The conclusion reached by this analysis is that the performance of the parallel production system design proposed in this research investigation depends upon the knowledge

base of the application of interest. For this design to perform well, the application production system must exhibit the following qualities:

1. The RHSs of all production rules must, on the average, affect many CEs. Ideally, the average number of CEs affected would equal the number of processors available.

2. The production rules must lend themselves to fortuitous assignments to unique processors. Specifically, the production rules containing CEs that initiate concurrently filterable Rete network paths should be assignable to unique processors.

Appendix D. *Parallel RAV Expert System Program*

The appendix presents the actual C language code implementing the HyperCLIPS shell. Note that only the original CLIPS routines adapted for HyperCLIPS are included here. Routines not shown in this appendix are used in HyperCLIPS in their original CLIPS form.

```
/*************************************************************************
--   DATE: 11/17/89
--   VERSION: 1.0
--
--   TITLE:  CLIPS Ioader for iPSC/2 Hypercube
--   FILENAME: HOST.C
--   AUTHOR: Capt William A. Harding
--   COORDINATOR:  R. Norris
--   PROJECT: Hypercube Expert System Shell - Application of
--           Production Parallelism (Thesis)
--   OPERATING SYSTEM:  UNIX System V/386 R3.0 (for iPSC/2)
--   LANGUAGE: C
--   FILE PROCESSING:  Compile and link with chost.def, stdio.h
--   CONTENTS:
--       main      - executive module
--       power     - exponentiation subroutine
--
--   FUNCTION:
--
--   This program prompts the user for the dimension of the cube to be
--   used for the expert system shell.  It then loads processors with
--   the knowledge base represented in user-entered files.  The host
--   then prompts for the run limit (number of rules to fire before
--   stopping) and the watch option desired (to display rules, facts,
```

```
        fp = fopen ("watch.out","w");
        fprintf (fp,"\nCLIPS EXPERT SYSTEM SHELL FOR THE iPSC/2 HYPERCUBE\n\n"
                );
        fprintf (fp,"      Written by Capt William A. Harding");


/*************************************************************************/
/* Get cube dimension from user and send to nodes                      */
/*************************************************************************/


        printf  (" Enter cube dimension (1-5): ");
        scanf   ("%d",&dim);
        csend (DIM_TYPE, &dim, sizeof(dim), ALL_NODES, NODE_PID);
        fprintf (fp,"\n\n** Dimension %d\n",dim);


/*************************************************************************/
/* WM loop:  Prompt for the names of files holding facts and templates */
/* and send these so each node can initialize its copy of working memory*/
/* The loop waits for the nodes to load a file before requesting another*/
/*************************************************************************/


        dummy = 1;  /* dummy - no meaningful value */
        num_nodes = power(2,dim);


        printf ("\n Enter fact file names for all processors: ");
        printf ("\n -> ");
        scanf ("%s", sname);
        while ((infile = fopen(sname, "r")) != NULL) {
           fclose(infile);
           for (i = 0; i < num_nodes; i++) {
```

D-4

```
--    or all affected during the run).  Run times are then collected from
--    the nodes, as well as any "watch" results, and these are displayed.
*************************************************************************/
#include "/usr/ipsc/lib/chost.def"
#include <stdio.h>


#define NODE_PID      1          /* Node process id                     */
#define HOST_PID      1          /* host process id                     */
#define ALL_NODES     -1         /* all active nodes in the cube         */
#define ALL_PIDS      -1         /* all active processes in the cube     */
#define DONE_TYPE     0          /* type of done message                */
#define DIM_TYPE      10         /* type of dimension message            */
#define FILE_TYPE     20         /* type of filename message             */
#define LIM_TYPE      30         /* type of run limit message            */
#define ITEM_TYPE     40         /* type of watch item message           */
#define ACT_TYPE      50         /* type of watch action value message   */
#define FIRE_TYPE     60         /* type of rules fired message          */
#define TIME_TYPE     70         /* type of time message                */
#define REPORT_TYPE   80         /* type of report signal message        */
#define GO_TYPE       90         /* type of synch start message          */
#define TIME_SIZE     (sizeof(long)) /* size of time message in bytes   */


main ()
{
int     dim,                    /* dimension of cube                   */
        num_nodes,              /* number of nodes in current cube dim */
        i,                      /* iteration counter (thru processors) */
        dummy,                  /* dummy variable (done messages)      */
        run_limit,              /* run limit for desired run           */
        watch_action_value,     /* turns watch option ON or OFF        */
```

```c
        check_firings,          /* checks consistency of rules fired    */
        start,                  /* signifies 1st node's responses coming*/
        rules_fired;            /* number of rules fired during run     */

float   longest_time,           /* maximum run time of all nodes        */
        rules_per_sec;          /* run's average rules fired per second */

long    run_time;               /* run time returned from a node        */

char    fname[15],              /* loadfile name                        */
        sname[15],              /* setfile name                         */
        watch_item[15];         /* item to display using watch option   */

FILE    *fp,                    /* file pointer                         */
        *infile;                /* input file pointer                   */


/****************************************************************************/
/* Load the cube                                                          */
/****************************************************************************/

    setpid (HOST_PID);
    load   ("node", ALL_NODES, NODE_PID);


/****************************************************************************/
/* Print welcome message                                                  */
/****************************************************************************/

    printf ("\n    CLIPS EXPERT SYSTEM SHELL FOR THE iPSC/2 HYPERCUBE\n\n"
            );
    printf ("            Written by Capt William A. Harding\n\n");
```

```c
        fp = fopen ("watch.out","w");
        fprintf (fp,"\nCLIPS EXPERT SYSTEM SHELL FOR THE iPSC/2 HYPERCUBE\n\n"
                 );
        fprintf (fp,"    Written by Capt William A. Harding");


/**********************************************************************/
/* Get cube dimension from user and send to nodes                    */
/**********************************************************************/

        printf  (" Enter cube dimension (1-5): ");
        scanf   ("%d",&dim);
        csend (DIM_TYPE, &dim, sizeof(dim), ALL_NODES, NODE_PID);
        fprintf (fp,"\n\n** Dimension %d\n",dim);


/**********************************************************************/
/* WM loop:  Prompt for the names of files holding facts and templates */
/* and send these so each node can initialize its copy of working memory*/
/* The loop waits for the nodes to load a file before requesting another*/
/**********************************************************************/

        dummy = 1;  /* dummy - no meaningful value */
        num_nodes = power(2,dim);

        printf ("\n Enter fact file names for all processors: ");
        printf ("\n -> ");
        scanf ("%s", sname);
        while ((infile = fopen(sname, "r")) != NULL) {
          fclose(infile);
          for (i = 0; i < num_nodes; i++) {
```

```c
            csend(FILE_TYPE, sname, sizeof(sname), i, NODE_PID);
            crecv(DUNE_TYPE, &dummy, sizeof(dummy));
        }
        printf (" -> ");
        scanf ("%s", sname);
    }
    fclose(infile);
    for (i = 0; i < num_nodes; i++) {
        csend(FILE_TYPE, sname, sizeof(sname), i, NODE_PID); }


/**************************************************************************/
/* PM loop:  Prompt for each node's rule file names and send these to   */
/* the nodes so they can access the files to load their rule bases.     */
/* The loop waits for the node to load a file before requesting another.*/
/**************************************************************************/

    for (i = 0; i < num_nodes; i++) {
        printf ("\n Enter rule file names for processor %d : ", i);
        printf ("\n -> ");
        scanf ("%s", fname);
        while ((infile = fopen(fname, "r")) != NULL) {
            fclose(infile);
    csend(FILE_TYPE, fname, sizeof(fname), i, NODE_PID);
            crecv(DONE_TYPE, &dummy, sizeof(dummy));
            printf (" -> ");
            scanf ("%s", fname);
        }
        fclose(infile);
        csend(FILE_TYPE, fname, sizeof(fname), i, NODE_FID);
    }
```

```
/**************************************************************************/
/* Prompt for run options (run_limit, watch_item, watch_action_value)  */
/* and send these to all nodes.  This serves as the nodes' "GO" signal. */
/**************************************************************************/


    printf ("\n Enter run limit (-1 for no limit): ");
    scanf ("%d", &run_limit);
    csend(LIM_TYPE, &run_limit, sizeof(run_limit), ALL_NODES, NODE_PID);


    printf ("\n Enter watch item (lower case): ");
    scanf ("%s", watch_item);
    csend(ITEM_TYPE, watch_item, sizeof(watch_item), ALL_NODES, NODE_PID);


    printf ("\n Enter watch_action_value (1-ON, 0-OFF): ");
    scanf ("%d", &watch_action_value);
    printf ("\n\n Executing... \n\n");
    csend(ACT_TYPE, &watch_action_value, sizeof(watch_action_value),
          ALL_NODES, NODE_PID);


    for (i = 0; i < num_nodes; i++) {

        crecv(DONE_TYPE, &dummy, sizeof(dummy));}


    csend(GO_TYPE, dummy, sizeof(dummy), ALL_NODES, NODE_PID);


/**************************************************************************/
/* Get run data from nodes, compute and display the results.           */
/**************************************************************************/
```

```c
        longest_time = 0.00;
        start = 1;  /* true - first time thru */


        for (i = 0; i < num_nodes; i++) {

            csend(REPORT_TYPE, dummy, sizeof(dummy), i, NODE_PID);


            crecv(FIRE_TYPE, &rules_fired, sizeof(rules_fired));
            if (start == 1) {
               check_firings = rules_fired;
               start = 0;  /* false - first time only */
            }
            else {
               if (check_firings != rules_fired)
                  printf ("\n** ERROR -> rules fired discrepancy **\n");
            }


            crecv(TIME_TYPE, &run_time, TIME_SIZE);
            if (longest_time < (float)run_time/1000.00)
               longest_time = (float)run_time/1000.00;


        }


        rules_fired--;  /* subtract out setup rule firing */
        rules_per_sec = (float)rules_fired / longest_time;

        printf ("\n** Run Completed after %d rule firings.\n", rules_fired);
        fprintf (fp,"\n** Run Completed after %d rule firings.\n",
                 rules_fired);
```

```c
        printf ("\n** Total Run Time = %0.5f seconds.\n", longest_time);
        fprintf (fp,"\n** Total Run Time = %0.5f seconds.\n", longest_time);


        printf ("\n** Average for Run = %0.5f rules/second.\n",
                                        rules_per_sec);
        fprintf (fp,"\n** Average for Run = %0.5f rules/second.\n",
                                           rules_per_sec);


        fclose (fp);


/**************************************************************************/
/*  Close channels to the cube for this run                             */
/**************************************************************************/


        killcube (ALL_NODES, ALL_PIDS);
}
/**************************************************************************/
/*  Subroutine Power                                                    */
/**************************************************************************/
power(base, exp)
int base,exp;
{
        int answer;
        for (answer = 1;  exp > 0; exp--)
            answer = answer * base;
        return(answer);
}



/**************************************************************************
```

```
--    DATE: 11/17/89

--    VERSION: 1.0

--

--    TITLE:  CLIPS Node for iPSC/2 Hypercube

--    FILENAME:  NODE.C

--    AUTHOR: Capt William A. Harding

--    COORDINATOR:  R. Norris

--    PROJECT: Hypercube Expert System Shell - Applying

--             Production Parallelism (Thesis)

--    OPERATING SYSTEM:  NX/2 Node eXecutive (for iPSC/2)

--    LANGUAGE: C

--    FILE PROCESSING:  Compile and link with host.c, cnode.def, stdio.h,

--                      msgs.h, and clips.h

--    CONTENTS:

--        main      - executive module

--        power     - exponentiation subroutine

--

--    FUNCTION:

--

--    This program receives the dimension of the cube to be used for the

--    expert system shell and determines which nodes will remain active.

--    It then initializes the CLIPS expert system shell and loads the

--    knowledge base from user-entered files (at the host).  Each node

--    then receives the run limit (number of rules to fire before

--    stopping) and the watch option desired (to display rules, facts,

--    etc. affected during the run).  Run times are sent to the host,

--    as well as any "watch" results (saved as a file at the host).

**************************************************************************/


#include "/usr/ipsc/lib/cnode.def"
```

```c
#include <stdio.h>
#include "clips.h"
#include "msgs.h"


#define HOST_NID   myhost()   /* host node id                          */
#define HOST_PID      1        /* host process id                       */
#define NODE_PID      1        /* Node process id                       */
#define ALL_NODES    -1        /* All nodes' ids                        */
#define DONE_TYPE     0        /* type of done message                  */
#define DIM_TYPE     10        /* type of dimension message             */
#define FILE_TYPE    20        /* type of filename message              */
#define LIM_TYPE     30        /* type of run limit message             */
#define ITEM_TYPE    40        /* type of watch item message            */
#define ACT_TYPE     50        /* type of watch action value message    */
#define FIRE_TYPE    60        /* type of rules fired message           */
#define TIME_TYPE    70        /* type of time message                  */
#define REPORT_TYPE  80        /* type of report signal message         */
#define GO_TYPE      90        /* type of synch start message           */
#define TIME_SIZE    (sizeof(long)) /* size of time message             */


/*************************************************************************/
/* Global Variables (seen also in files: engine.c and factmngr.c)      */
/*************************************************************************/


int     dim,                 /* cube dimension                        */
        my_node,             /* my node number                        */
        my_pid,              /* my node process id                    */
        num_nodes,           /* number of nodes in current cube dim   */
        RHS_type,            /* type of action message                */
        RUNNING;             /* global flag - master rule firing node */
```

```c
long     start_time;                 /* start time of run                    */


RHS_msg my_RHS_buf;                   /* global msg -rule firing mechanism    */


main()


/***********************************************************************/
/* Local Variables                                                     */
/***********************************************************************/


{
int     dummy,                        /* dummy variable (done messages)       */
        run_limit,                    /* run limit for desired run            */
        watch_action_value,           /* turns watch option ON or OFF         */
        msg_type,                     /* message type buffer                  */
        rules_fired;                  /* number of rules fired during run     */


long    /* start_time,                   start time of run                    */
        run_time;                     /* total time of run                    */


char    fname[15],                    /*    dfile name                        */
        sname[15],                    /* setfile name                         */
        watch_item[15];               /* item to display using watch option   */


FILE    *f1;                          /* file pointer                         */


/***********************************************************************/
/* each node define its node number and pid                            */
/***********************************************************************/
```

```
    my_node = mynode();
    my_pid = mypid();


/**********************************************************************/
/* receive dimension from host, compute number of nodes, & start CLIPS  */
/**********************************************************************/


    crecv (DIM_TYPE, &dim, sizeof(dim));
    num_nodes = power(2,dim);
    if (my_node >= num_nodes) exit (0);


    init_clips();


    RUNNING = FALSE;  /* no nodes firing rules yet */


/**********************************************************************/
/* WM loop: Receive from the host names of files holding facts/templates*/
/* and read these to initialize the local copy of working memory (facts)*/
/* Continue this loop for all fact file names sent from the host.       */
/**********************************************************************/


    /* set_conserve("on");   on - pprule info not kept */
    dummy = 1;  /* dummy - no meaningful value */


    crecv(FILE_TYPE, sname, sizeof(sname));
    while ((fl = fopen(sname, "r")) != NULL) {
        fclose(fl);
        load_rules(sname);
        csend(DONE_TYPE, dummy, sizeof(dummy), HOST_NID, my_pid);
```

```
            crecv(FILE_TYPE, sname, sizeof(sname));
    }
    fclose(fl);


/**************************************************************************/
/* PM loop: Receive rule file names from the host and access the files  */
/* at the host to load the local rule base.  Continue this loop for     */
/* all rule file names sent from the host.                              */
/**************************************************************************/

    crecv(FILE_TYPE, fname, sizeof(fname));
    while ((fl = fopen(fname, "r")) != NULL) {
        fclose(fl);
        load_rules(fname);
        csend(DONE_TYPE, dummy, sizeof(dummy), HOST_NID, my_pid);
        crecv(FILE_TYPE, fname, sizeof(fname));
    }
    fclose(fl);


/**************************************************************************/
/* Receive run options from host (run_limit, watch_item,                */
/* watch_action_value).  Then reset CLIPS environment and start the run.*/
/**************************************************************************/

    crecv(LIM_TYPE, &run_limit, sizeof(run_limit));
    crecv(ITEM_TYPE, watch_item, sizeof(watch_item));
    crecv(ACT_TYPE, &watch_action_value, sizeof(watch_action_value));


    set_watch(watch_item, watch_action_value);
```
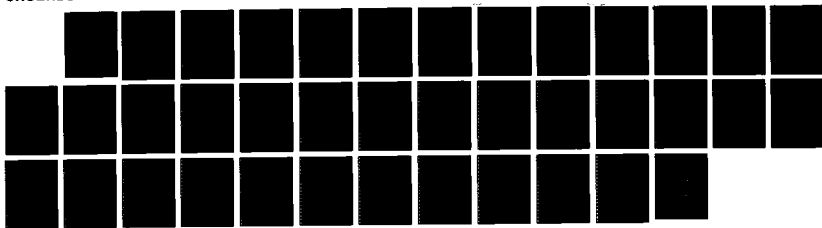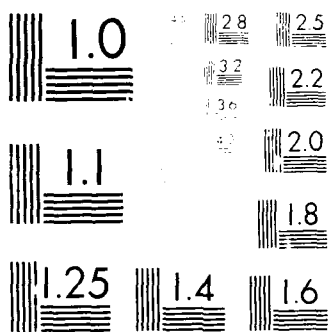
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS

```c
    reset_clips();

    csend(DONE_TYPE, dummy, sizeof(dummy), HOST_NID, my_pid);
    crecv(GO_TYPE, &dummy, sizeof(dummy));

    rules_fired = run(run_limit);
    run_time = mclock() - start_time;
    /* displayfacts(); * enable if test desired */


/**************************************************************************/
/* Send run data for this node to the host                             */
/**************************************************************************/

    crecv(REPORT_TYPE, &dummy, sizeof(dummy));
    csend(FIRE_TYPE, &rules_fired, sizeof(rules_fired), HOST_NID, my_pid);
    csend(TIME_TYPE, &run_time, TIME_SIZE, HOST_NID, my_pid);


}
/**************************************************************************/
/* Subroutine Power                                                     */
/**************************************************************************/
power (base, exp)
int base, exp;
{
    int answer;
    for (answer = 1; exp > 0; exp--)
        answer = answer * base;
    return(answer);
}
```

```
/***************************************************************/
/* USRFUNCS:  The function which informs CLIPS of any user     */
/*   defined functions.  In the default case, there are no     */
/*   user defined functions.  To define functions, either      */
/*   this function must be replaced by a function with the     */
/*   same name within this file, or this function can be       */
/*   deleted from this file and included in another file.      */
/*   User defined functions may be included in this file or    */
/*   other files.                                              */
/*   Example of redefined usrfuncs:                            */
/*     usrfuncs()                                              */
/*       {                                                     */
/*         define_function("fun1",'i',fun1,"fun1");            */
/*         define_function("other",'f',other,"other");         */
/*       }                                                     */
/***************************************************************/
usrfuncs()
  {
  }
```

```
/*   HyperCLIPS Version 1.0   11/17/89 */


   /*********************************************************/
   /* NOTE! : the following routines are compiled within  */
   /* "engine.c".  Only new routines added for HyperCLIPS */
   /* or original CLIPS routines adapted for HyperCLIPS   */
   /* are shown here.  Other routines not shown here must */
   /* still be compiled in their original CLIPS form      */
   /* within "engine.c"                                   */
   /*********************************************************/


   /*********************************************************/
   /*      "C" Language Integrated Production System      */
   /*                  ENGINE MODULE                      */
   /*********************************************************/


#include <stdio.h>

#include "clips.h"
#include "engine.h"
#include "msgs.h"
#define ALL_NODES    -1                 /* All nodes' ids         */
#define BUFFER_SIZE  (sizeof(gray_msg)) /* size of message buffer */
#define RHS_SIZE     (sizeof(RHS_msg))  /* size of RHS msg buffer  */


/****************************************/
/* GLOBAL INTERNAL FUNCTION DEFINITIONS */
/****************************************/
```

```
int             run();
int             add_activation();
int             remove_all_activations();
int             set_agenda_count();
int             set_activations_watch();
int             get_activations_watch();
int             purge_agenda();
int             clear_rule_from_agenda();
int             print_activation();
struct activation   *get_next_activation();
int             set_execution_error();
int             get_execution_error();
int             get_change_agenda();
int             set_change_agenda();
int             global_select();
int             get_partner_node();
int             slave_run();
struct fact     *ptr_to();


/******************************************/
/* LOCAL INTERNAL VARIABLE DEFINITIONS */
/*********************** ****************/


static struct activation   *AGENDA = NULL;
static long int             AGENDA_COUNT = 0;
static int                  watch_activations = OFF;
static int                  EXECUTION_ERROR = FALSE;
static char                 *currentrule = NULL;
static struct exec_func     *exec_list = NULL;
static int                  change_agenda = FALSE;
```

```
        static int                executing = FALSE;

        static int                NULL_SALIENCE = -10001;

        static int                i = 0;

        gray_msg                  my_msg_buf,

                                  part_msg_buf;

        int                       rules_fired = 0;


/****************************************:****/
/* GLOBAL EXTERNAL VARIABLE DEFINITIONS */
/*******************************************/


        extern RHS_msg            my_RHS_buf;

        extern struct fbind       *gbl_lhs_binds;

        extern struct fbind       *gbl_rhs_binds;

        extern int dim,           /* dimension of cube        */

                   my_node,       /* node number              */

                   my_pid,        /* node pid                 */

                   num_nodes,     /* number of nodes in cube  */

                   RHS_type,      /* type of action message   */

                   RUNNING;       /* node running flag        */

        extern long start_time;   /* start time hack          */


/*****************************************************************/
/* RUN: Begins execution of rules.  If run limit is less than */
/*   zero, then rules will be executed until all nodes'agendas*/
/*   are empty.  If run limit is greater than zero, then rules*/
/*   will be executed until either all agendas are empty, the */
/*   run limit has been reached, or a rule execution error    */
/*   has occured.  Returns the number of rules fired.         */
/*****************************************************************/
```

```
run(run_limit)
  int run_limit;
  {
   struct test *commands;
   struct fbind *local_vars;
   struct activation *rule_to_fire;
   char print_space[20];
   struct values result;
   struct exec_func *exec_ptr;
   RHS_msg done_RHS_buf;


   done_RHS_buf.type = DONE;
   strcpy(done_RHS_buf.fact_str, EOS);
   done_RHS_buf.index = -1;


   /*=======================================================*/
   /* Fire rules until all agendas empty, the run limit    */
   /* has been reached, or a rule execution error occurs. */
   /*=======================================================*/


   EXECUTION_ERROR = FALSE;
   executing = TRUE;

 global_select();

 while ((my_msg_buf.salience_of_rule != NULL_SALIENCE) &&
        (run_limit != 0) &&
        (EXECUTION_ERROR == FALSE))
   {
```

D-19

```
    rules_fired++;
  if (run_limit > 0) { run_limit--; }


  if (my_msg_buf.chosen_node == my_node) {

      RUNNING = TRUE;
      RHS_type = my_node + num_nodes;


      /*=============================*/
      /* Bookkeeping and Tracing. */
      /*=============================*/


      currentrule = AGENDA->rule;


      if (get_rules_watch() == ON)
        {
         sprintf(print_space,"FIRE %4d ",rules_fired);
         cl_print("wtrace",print_space);
         cl_print("wtrace",currentrule);
         cl_print("wtrace",": ");
         print_fact_basis("wtrace",AGENDA->basis->binds);
         cl_print("wtrace","\n");
        }


      if (get_crsv_trace_watch() == ON)
        {
         sprintf(print_space,"F    %-4d   ",rules_fired);
         cl_print("wcrsv_tr",print_space);
         cl_print("wcrsv_tr",currentrule);
cl_print("wcrsv_tr","\n");
```

```
      }


      /*=============================================*/
      /* Execute the rule's right hand side actions. */
      /*=============================================*/


      change_agenda = TRUE;

      rule_to_fire = AGENDA;

      commands = AGENDA->actions;

      local_vars = AGENDA->basis->binds;

      AGENDA = AGENDA->next;


      gbl_lhs_binds = local_vars;

      gbl_rhs_binds = NULL;


      commands = commands->arg_list;

      while ((commands != NULL) && (EXECUTION_ERROR == FALSE))
        {
          generic_compute(commands,&result);
          commands = commands->next_arg;
        }
      commands = NULL;


      /*=====================================*/
      /* Return the agenda node to free memory. */
      /*=====================================*/


      returnbinds(local_vars);

      rtn_struct(flink,rule_to_fire->basis);

      rtn_struct(activation,rule_to_fire);
```

```
/*=========================================*/
/* Tell other nodes my RHS firing is done */
/*=========================================*/


csend(RHS_type, &done_RHS_buf, RHS_SIZE, ALL_NODES, my_pid);
RUNNING = FALSE;
}


else {

  slave_run();

}


  /*===========================================*/
  /* Remove retracted facts, ephemeral symbols, */
  /* variable bindings, and temporary segments. */
  /*===========================================*/


  rmv_old_facts();
  rem_eph_symbols();
  flush_bind_list();
  flush_segments();


  /*===========================================*/
  /* Execute exec list after performing actions. */
  /*===========================================*/


  exec_ptr = exec_list;
  while (exec_ptr != NULL)
    {
```

```
        (*exec_ptr->ip)();
        exec_ptr = exec_ptr->next;
    }


    global_select();
}


    executing = FALSE;
    EXECUTION_ERROR = FALSE;
    return(rules_fired);
}


/*********************************************************/
/* Global_Select: Sends salience of top rule on local   */
/*    AGENDA for comparison.  Receives salience of RHS   */
/*    to compute and id of node holding selected rule    */
/*********************************************************/
global_select()
{
    int partner_node,
        count;

    if (rules_fired == 1) { start_time = mclock(); } /* start time hack */

    if (AGENDA == NULL) {
        my_msg_buf.salience_of_rule = NULL_SALIENCE;
    }
    else {
        my_msg_buf.salience_of_rule = AGENDA->salience;
    }
```

```
my_msg_buf.chosen_node = my_node;


/*====-  .================================================*/
/* Do gray-code compare-exchange down to node 0         */
/*======================================================*/


for (count = 0; count < dim; count++ ) {
    partner_node = get_partner_node(my_node, (power(2,count)));


    if (my_node > partner_node)
       {
         csend'my_node, &my_msg_buf,
              BUFFER_SIZE,
              partner_node, my_pid);
       }
    else {
         crecv(partner_node, &part_msg_buf,
              BUFFER_SIZE);
         if (part_msg_buf.salience_of_rule >
                      my_msg_buf.salience_of_rule)
           { my_msg_buf.salience_of_rule =
                      part_msg_buf.salience_of_rule;
             my_msg_buf.chosen_node = part_msg_buf.chosen_node;
           }
         }
    }


/*======================================================*/
/* broadcast overall best msg to all nodes              */
/*======================================================*/
```

D-24

```
     if (my_node != 0) {
         crecv(0, &my_msg_buf, BUFFER_SIZE);
         }
     else {
         csend(0, &my_msg_buf, BUFFER_SIZE, ALL_NODES, my_pid);
         }


  }


/**************************************************************************/
/* Get_Partner_Node: Returns gray-code partner node for message passing */
/**************************************************************************/
get_partner_node (this_node, xor_value)
int this_node,
    xor_value;
{
    int binary_code;
    binary_code = (gray(this_node)) ^ (gray(xor_value));
    return(ginv(binary_code));
}


/**************************************************************************/
/* SLAVE_RUN: Receives assertion/retraction to perform from chosen node */
/**************************************************************************/
slave_run()
{
    struct fact *ptr;
    int my_RHS_type;
```

```c
        my_RHS_type = my_msg_buf.chosen_node + num_nodes;
        crecv(my_RHS_type, &my_RHS_buf, RHS_SIZE);


        while (my_RHS_buf.type != DONE) {
            switch (my_RHS_buf.type) {
                case ASSERT  : assert(my_RHS_buf.fact_str);
                               if (rules_fired == 1) {
                                   printf("I%d:\"%s\"\n", my_node,
                                           my_RHS_buf.fact_str);
                                   }
                               break;


                case RETRACT :if ((ptr = ptr_to(my_RHS_buf.index)) != NULL)
                               { retract_fact(ptr); }
                               break;
                default : printf("ERROR in switch stmt type\n"); break;
            }
            crecv(my_RHS_type, &my_RHS_buf, RHS_SIZE);
        }
}


/***************************************************************************/
/* PTR_TO: Returns pointer to the fact with the specified index          */
/***************************************************************************/
struct fact *ptr_to(index)
FACT_ID index;
{
    struct fact *ptr;


    ptr = get_next_fact(NULL);
```

```c
    while (ptr != NULL)
      {
        if (ptr->ID == index)
           { return(ptr); }
        ptr = get_next_fact(ptr);
      }
    return(NULL); /* fact not found */
}


/*   HyperCLIPS Version 1.0    11/17/89 */


  /**********************************************************/
  /* NOTE! : the following routines are compiled within  */
  /* "factmngr.c".  Only new routines added for           */
  /* HyperCLIPS or original CLIPS routines adapted for    */
  /* HyperCLIPS are shown here.  Other routines not shown*/
  /* here must still be compiled in their original CLIPS */
  /* form within "factmngr.c"                             */
  /**********************************************************/


  /**********************************************************/
  /*      "C" Language Integrated Production System       */
  /*                FACT MANAGER MODULE                   */
  /**********************************************************/


#include <stdio.h>
#include <string.h>

#include "setup.h"
#include "msgs.h"
```

```c
#include "clips.h"
#include "scanner.h"


#define ALL_NODES     -1                  /* All nodes' ids         */
#define RHS_SIZE      (sizeof(RHS_msg)) /* size of RHS msg buffer */


/***************************************/
/* GLOBAL INTERNAL FUNCTION DEFINITIONS */
/***************************************/


   struct fact          *get_el();
   struct fact          *add_fact();
   char                 *build_str();


/***************************************/
/* GLOBAL EXTERNAL FUNCTION DEFINITIONS */
/***************************************/


   extern struct draw    *add_symbol();
   extern struct element *fast_gv();
   extern struct pat_node *network_pointer();
   extern char           *symbol_string();


/***************************************/
/* LOCAL INTERNAL VARIABLE DEFINITIONS */
/***************************************/


   static struct fact    *garbage_facts = NULL;
   static struct fhash   *fact_hashtable[SIZE_FACT_HASH];
   static int             watch_facts;
```

```c
    static struct fact    *factlist;
    static struct fact    *last_fact;
    static long int        ID;
    static int             change_facts = FALSE;


/***************************************/
/* GLOBAL EXTERNAL VARIABLE DEFINITIONS */
/***************************************/


    extern int            my_node,
                          my_pid,
                          RHS_type,
                          RUNNING;
    extern RHS_msg        my_RHS_buf;
    extern struct fbind   *gbl_lhs_binds;
    extern struct fbind   *gbl_rhs_binds;
    extern struct funtab  *PTR_GET_VAR;




/*****************************************************************/
/* RETRACT_FACT:  Retracts a fact from the fact list given a */
/*    pointer to the fact.                                   */
/*****************************************************************/
retract_fact(fact_ptr)
  struct fact *fact_ptr;
  {
  FACT_ID fact_num;
  struct fact *temp_ptr;
  struct match *match_list;
  char print_space[20];
```

```
/*=============================================================*/
/* Check to see if the fact has already been retracted. */
/*=============================================================*/

if (RUNNING == TRUE)
  {
   temp_ptr = fact_ptr;
   my_RHS_buf.type = RETRACT;
   my_RHS_buf.index = temp_ptr->ID;
   strncpy(my_RHS_buf.fact_str,EOS,
                     strlen(my_RHS_buf.fact_str));/*not used*/
   isend(RHS_type, &my_RHS_buf, RHS_SIZE, ALL_NODES, my_pid);
  }


temp_ptr = garbage_facts;
while (temp_ptr != NULL)
  {
   if (temp_ptr == fact_ptr)
     { return(0); }
   temp_ptr = temp_ptr->next;
  }


/*=========================================*/
/* Show retraction if facts being watched. */
/*=========================================*/

fact_num = fact_ptr->ID;

if (watch_facts == ON)
```

```
      {
       cl_print("wtrace","<== ");
       show_fact("wtrace",fact_ptr);
       cl_print("wtrace","\n");
      }


  if (get_crsv_trace_watch() == ON)
     {
cl_print("wcrsv_tr","R     ");
      sprintf(print_space,"%-51d  ",fact_ptr->ID);
      cl_print("wcrsv_tr",print_space);
print_element("wcrsv_tr",&(fact_ptr->atoms[0]));
      cl_print("wcrsv_tr","\n");
     }


  change_facts = TRUE;


  /*======================================*/
  /* Delete the fact from the fact list. */
  /*======================================*/


  del_hash_fact(fact_ptr);
  /* Save the list of pattern matches. */
  match_list = fact_ptr->list;


  if (fact_ptr == last_fact)
    { last_fact = fact_ptr->previous; }


  if (fact_ptr->previous == NULL)
     {
```

```
        /* Delete the head of the fact list. */
        factlist = factlist->next;
        if (factlist != NULL)
          { factlist->previous = NULL; }
    }
    else
      {
        /* Delete a fact other than the head of the fact list. */
        fact_ptr->previous->next = fact_ptr->next;
        if (fact_ptr->next != NULL)
          { fact_ptr->next->previous = fact_ptr->previous; }
      }


temp_ptr = garbage_facts;
garbage_facts = fact_ptr;
fact_ptr->next = temp_ptr;


/*====================================================*/
/* Loop through the list of all the patterns that */
/* matched the fact.                               */
/*====================================================*/


match_retract(match_list,fact_num);


/*====================================================*/
/* Remove all activations that contain this fact */
/* from the agenda.                               */
/*====================================================*/


purge_agenda(fact_num);
```

```
    return(1);
  }



/*******************************************************************/
/* ADD_FACT: Places a fact onto the end of the fact list and calls   */
/*   compare to filter the fact through the pattern network. Returns */
/*   null if the fact was already in the knowledge base, and a       */
/*   pointer to the fact if it was not in the knowledge base.        */
/*******************************************************************/
struct fact *add_fact(new_fact)
  struct fact *new_fact;
  {
   int hash_value;
   struct fact *temp_fact;
   char print_space[20];


   /*===========================================================*/
   /* If fact assertions are being checked for duplications, */
   /* then search the fact list for a duplicate fact.        */
   /*===========================================================*/


   hash_value = hash_fact(new_fact);

   if (fact_exists(new_fact,hash_value) == 1)
     {
      rtn_el(new_fact);
      return(NULL);
     }
```

```
if (RUNNING == TRUE)
  {
    temp_fact = new_fact;
    build_str(temp_fact);
    my_RHS_buf.type = ASSERT;
    my_RHS_buf.index = -1;   /* not used */
    isend(RHS_type, &my_RHS_buf, RHS_SIZE, ALL_NODES, my_pid);
  }


add_hash_fact(new_fact,hash_value);


/*=====================================================*/
/* Add the fact to the fact list. Set the ID for the */
/* fact and install the symbols used by the fact in  */
/* the symbol table.                                 */
/*=====================================================*/


new_fact->next = NULL;
new_fact->list = NULL;
new_fact->previous = last_fact;
if (last_fact == NULL)
  { factlist = new_fact; }
else
  { last_fact->next = new_fact; }
last_fact = new_fact;


ID++;
new_fact->ID = ID;
fact_install(new_fact);
```

```
/*====================================================*/
/* Indicate the addition of the fact to the fact */
/* list if facts are being watched.              */
/*====================================================*/


if (watch_facts == ON)
  {
   cl_print("wtrace","==> ");
   show_fact("wtrace",new_fact);
   cl_print("wtrace","\n");
  }


if (get_crsv_trace_watch() == ON)
  {
   cl_print("wcrsv_tr","AS   ");
   sprintf(print_space,"%-5ld (",new_fact->ID);
   cl_print("wcrsv_tr",print_space);
   show_elements("wcrsv_tr",new_fact);
   cl_print("wcrsv_tr",")");
   cl_print("wcrsv_tr","\n");
  }


change_facts = TRUE;


/*====================================================*/
/* Filter the fact through the pattern network. */
/*====================================================*/


compare(new_fact,new_fact->atoms,network_pointer(),1,0,NULL,NULL);
```

```c
    return(new_fact);
  }




/*****************************************************/
/* Build_Str:                                        */
/*****************************************************/
char *build_str(fact_ptr)
  struct fact *fact_ptr;
  {
  struct element *sublist, *elem_ptr;
  char *num_to_string();
  int length, i;


  length = fact_ptr->fact_length;
  sublist = fact_ptr->atoms;


  strncpy(my_RHS_buf.fact_str,"",strlen(my_RHS_buf.fact_str));


  for (i = 0; i < length; i++)
    {
      elem_ptr = &sublist[i];


      if (elem_ptr->type == NUMBER)
        {
          strcat(my_RHS_buf.fact_str,
                 num_to_string(elem_ptr->val.fvalue));
        }
      else if (elem_ptr->type == WORD)
```

D-36

```c
                {
                    strcat(my_RHS_buf.fact_str,
                            symbol_string(elem_ptr->val.hvalue));
                }
        else if (elem_ptr->type == STRING)
                {
                    strcat(my_RHS_buf.fact_str, "\'");
                    strcat(my_RHS_buf.fact_str,
                            symbol_string(elem_ptr->val.hvalue));
                    strcat(my_RHS_buf.fact_str, "\"");
                }
        if (i + 1 != length)
            { strcat(my_RHS_buf.fact_str, " ");  }
    }

}
```

E.1 *General Overview*

HyperCLIPS is developed using serial CLIPS as an embedded program as detailed in the Advanced Programming Guide section of the CLIPS Reference Manual (Version 4.3). This manual, which comes as part of CLIPS Version 4.3, is available through the Computer Software Management and Information Center (COSMIC), the distribution point for NASA software. Further information can be obtained from

```
COSMIC
382 E. Broad St.
Athens, GA   30602
(404) 542-3265
```

The HyperCLIPS programmer may see the CLIPS Reference Manual for an overview of CLIPS operation, syntax, and programming.

E.2 *HyperCLIPS Initialization*

Executing HyperCLIPS as a stand-alone expert system shell, the iPSC/2 front-end host processor provides the user interface to the system and loads the node program on all active node processors. The node program on each node processor initializes HyperCLIPS for that processor, loads facts and templates from host files constituting the initial working memory for that processor (same on all), and loads the node processor's rules from specified host files (each rule partition should be unique). After then resetting HyperCLIPS and activating desired run-time watch options, the nodes start off synchronously executing the match step in the match-select-act cycle. The identical working memory is maintained on all node processor's throughout execution.

E.3 **HyperCLIPS Basic Cycle of Execution**

The basic production system algorithm is executed by HyperCLIPS as a Match-Select-Act cycle in the following order:

1. Match - Each processor evaluates the LHSs of the local production rules to determine which are satisfied given the current contents of its local copy of working memory. Satisfied production rules are entered into the local conflict set for that processor. The start rule is matched first because it has an empty LHS.

2. Select - Each processor chooses the rule with the highest salience from the local conflict set. If none of a processor's production rules have satisfied LHSs, a salience less than the minimum possible HyperCLIPS salience is used (e.g. null salience). The processors perform a gray-code compare/exchange of each processor's chosen rule's salience and that processor's node ID. For a 2^d hypercube, the root processor is guaranteed to hold the highest salience of any candidate rule and the ID of the processor that hosts that rule after d compare/exchanges. The root broadcasts to all processors the selected rule's salience and hosting processor's ID, setting the rule hosting processor's state to that of action master and setting the states of the other processors' to that of action slave.

3. Act - The action master processor executes a normal, serial CLIPS computation of the selected rule's RHS, resulting in some combination of fact assertions (by story) and retractions (by fact ID). These assertions and retractions are broadcasted from the master processor to all slave processors, each of which then asserts/retracts the desired fact into/out of its local working memory.

4. If a termination condition is detected by all processors, then control is returned to the user. Possible termination conditions include the highest rule salience selected on any processor being the null salience (i.e. no rules are satisfied on any processor) or the run limit having been reached. If no termination condition is detected, go to Step 1 (Match)

E.4 *Detailed Design*

As mentioned above, HyperCLIPS is developed using serial CLIPS as an embedded **program.** The host and node programs replace the original main program defined for serial CLIPS. To maintain as much of the serial CLIPS modularity, functionality, and integrity as

possible, HyperCLIPS is accomplished using minimal adaptation of original CLIPS source code. Specifically, only the *run()* routine (in CLIPS file "engine.c") and the *add_fact()* and *retract_fact()* routines (in CLIPS file "factmngr.c") are altered from their original serial representation. Some extra service routines are added to these same two files to implement HyperCLIPS (see the HyperCLIPS source code in these two files for details).

E.5 Embedding HyperCLIPS

Like serial CLIPS, the HyperCLIPS shell may be used as an embedded program. The basic approach to embedding HyperCLIPS for use by another application program on the iPSC/2 is to 1) edit the host and node programs as needed to perform the higher-level application and 2) call HyperCLIPS using the function calls normally used to control embedded CLIPS.

A certain minimum set of function calls must be made when executing HyperCLIPS as an embedded program. First, *init_clips()* must be called prior to any other CLIPS function to initialize the HyperCLIPS environment. Second, all deffact, deftemplate, and defrule statements (in order) must be loaded as the initial application knowledge base using the *load_rules()* function. Third, *reset_clips()* must be called to reset the HyperCLIPS environment, thus removing all activations from the conflict set and all facts from working memory, and asserting all facts listed in deffacts statments into working memory. Finally, the top-level program must call *run()* to initiate expert system execution (allowing rules to fire). All other CLIPS functions available when embedding CLIPS are available when embedding HyperCLIPS.

F.1 *HyperCLIPS Overview*

HyperCLIPS is a parallel expert system shell designed to execute on the Intel Second Generation Personal Supercomputer (iPSC/2). HyperCLIPS is an adaptation of the serial C-Language Integrated Production System (CLIPS Version 4.3) developed by the Artificial Intelligence Section (AIS) at NASA/Johnson Space Center (JSC). HyperCLIPS is designed as a research tool for determining the processing speedup attainable through parallel processing of a given expert system application.

Much of the original source code for the serial version of CLIPS is used in the HyperCLIPS implementation. Therefore, those interested in using HyperCLIPS must first acquire CLIPS. CLIPS Version 4.3 is available through the Computer Software Management and Information Center (COSMIC), which is the distribution point for NASA software. Further information can be obtained from

> COSMIC
>
> 382 E. Broad St.
>
> Athens, GA 30602
>
> (404) 542-3265

The additional source code and setup files for HyperCLIPS are available upon request to

> HyperCLIPS
>
> c/o Dr. Gary Lamont
>
> Department of Computer and Electrical Engineering
>
> Air Force Institute of Technology
>
> Wright-Patterson AFB, OH 45433

F.2 *Requirements for Running HyperCLIPS*

To prepare the HyperCLIPS program for execution, you must first compile it using the Greenhills C (or compatible) Compiler and link it with the iPSC/2's run-time system

and libraries. The iPSC/2 currently uses the UNIX System V/386 operating system on the front-end host processor and Node eXecutive (NX/2) for each of the node processors in the hypercube. Using the provided setup files mentioned above to compile and link, the executable file produced for the HyperCLIPS shell is labelled *host*. Typing *host* at the UNIX system prompt will begin execution of HyperCLIPS.

Your own CLIPS-syntax expert system program must satisfy a few requirements before it can be compiled and executed by HyperCLIPS. All fact and template declarations and all productions must be contained in one or more files, and no declarations and productions can exist in the same file. All declarations files must be input before any productions files are input. To initialize the working memory for your execution, one of your productions must be a *start* production which has no conditions in its LHS and which has your desired list of initializing assertions in its RHS. The following is a sample start production:

$$(defrule \quad start :$$

$$\rightarrow$$

$$(assert \quad (SWITCH \quad (STATUS \quad ON)))$$
$$(assert \quad (SORTIE \quad (SET \quad MCCHORD)))$$
$$(assert \quad (TAKEOFF \quad (PLAN \quad ROGER))))$$

Although this sample start production happens to also be named "start", any valid CLIPS production name will do. When execution of your program begins, the start production will match and fire before any other productions. If you desire not initialize your working memory with the start production, simply leave the start production's RHS empty.

F.3 Interface to HyperCLIPS

HyperCLIPS does not maintain the high-level user interface provided by CLIPS Version 4.3. Instead, HyperCLIPS leads the user through a series of prompts to allow input of some key expert system shell commands required for execution to proceed.

After the startup welcome message, the first HyperCLIPS prompt requests the desired dimension of the hypercube:

```
Enter cube dimension (1-5):
```

This prompt allows the user to specify the number of processor nodes to be applied to solution of his expert system application. Note, however, that the response to this prompt specifies the *dimension* of the cube, not the *number* of nodes in the cube. For example, a response of "3" causes the activation of 2^3, or eight, processors in the cube.

Next, HyperCLIPS asks for the list of file names containing the fact and template declarations files. The prompt is of the following form:

```
Enter fact file names for all processors:
->
```

The user can enter any number of file names for loading, terminating entry of declarations files with an "x" (or any invalid file name). The facts and templates specified in these files are loaded onto each of the active processors.

Loading of the productions files follows a process similar to that of loading declarations files, except that HyperCLIPS prompts for the files to be loaded to each active processor individually, as follows:

```
Enter rule file names for processor P :
->
```

During actual execution of HyperCLIPS, the "P" shown in this sample prompt is replaced with the ID of the processor currently being loaded with productions files. HyperCLIPS does not protect against the user entering the same productions file into more

than one processor, nor does it check that no single production is loaded into more than one processor. Parsing of production rules into production files, and the subsequent assignment of production files to specific processors, is left to the complete discretion of the user. Any number of file names may be loaded, entry of files again being terminated with an "x" (or any invalid file name).

The next prompt requests the desired run limit for execution. The user can specify the number of rules to fire before suspending execution, or the user may enter -1 to set HyperCLIPS to run until no more rules remain to fire or until the program terminates (whichever occurs first):

```
Enter run limit (-1 for no limit):
```

The final two prompts allow setting of the CLIPS watch options for debugging and program verification purposes. The prompts appear in the following form:

```
Enter watch item (lower case):
```

```
Enter watch_action_value (1-ON, 0-OFF):
```

The user first enters a watch item, which specifies that "rules", "facts", "activations", "compilations", or "all" be displayed during expert system shell execution. The watch action value prompt turns the desired watch option ON (1) or OFF (0).

The HyperCLIPS program terminates after a single execution, requiring that responses to the entire sequence of prompts described above be reaccomplished for subsequent runs. This limitation is very inconvenient when a large number of declarations files and productions files must be entered for each execution. The workaround to avoid retyping of prompt responses is to use an executable macro file to initiate HyperCLIPS and provide responses to the user prompts.

F-4

F.4 HyperCLIPS Limitations

HyperCLIPS is designed as a parallel processing research tool. As such, its simple interface environment is not adequate for expert system application program debugging. The user is encouraged to debug his expert system application using serial CLIPS Version 4.3 before attempting to process the application using HyperCLIPS.

To achieve processing speedups using HyperCLIPS, two conditions must exist:

1. The user's application must lend itself to fortuitous assignment of production rules to available processors so as to attain some benefit from production parallelism.

2. The user must employ some algorithmic mechanism to recognize dependencies and relationships among the production rules, upon which fortuitous assignment to available processors depends.

In regard to condition 1, HyperCLIPS can only take advantage of the production parallelism inherent in the productions parsed to its separate processors. Concerning condition 2, HyperCLIPS offers no algorithmic inter-production dependency recognition mechanism, as the burden of assignment of productions to processors is left to the user.

Vita

Captain William A. Harding was born 22 May 1963 in Harvey, Illinois. He graduated from high school in Wilmington, Illinois, in 1981. He attended Southern Illinois University - Carbondale, where he received the degree of Bachelor of Arts in Computer Science in May 1985. Upon graduation, he received a regular commission in the United States Air Force through the Reserve Officer Training Corps program. He served as an Information System Programs and Analysis Officer with Headquarters Air Force Space Command, Peterson AFB, CO, until entering the School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, in May 1988.

Permanent address: 8278 Wold Den Court
                      Springfield, VA 22153

Bibliography

1. Automated Reasoning Tool (ART) Reference Manual. ART Version 2.0; Inference Corporation Publication. April 1986.

2. Barhen, Jacob, Sandeep Gulati, and S. Sitharama Iyengar. "The Pebble Crunching Model for Load Balancing in Concurrent Hypercube Ensembles." In *The Third Conference on Hypercube Concurrent Computers and Applications*, Volume 1, pages 189–199. The Association for Computing Machinery, 1988.

3. Brooke, Thomas M. "Cosmic: CLIPS," *AI Expert*, pages 71–73 (April 1988).

4. Chabris, Christopher F. *Artificial Intelligence and Turbo Pascal*. Homewood, Illinois: Multiscience Press, Inc, 1987.

5. CLIPS Reference Manual. Version 4.3 of CLIPS; Artificial Intelligence Section, Lyndon B. Johnson Space Center. June 1989.

6. Douglass, Robert J. "A Qualitative Assessment of Parallelism in Expert Systems," *IEEE Software*, 2:70–81 (May 1985).

7. Fanning, 1Lt Jesse. AFWAL Robotic Air Vehicle (RAV) Project Member. Telephone interview. Air Force Wright Aeronautics Laboratory, Wright-Patterson AFB, OH. 22 November 1989.

8. Flynn, M. J. "Very High-Speed Computing Systems." In *Proceedings of the IEEE*, pages 1901–1909, IEEE, December 1966.

9. Forgy, Charles L. OPS5 User's Manual; Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. July 1981.

10. Forgy, Charles L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, 19:17–37 (September 1982).

11. Forgy, Charles L. "Rete: A Fast Match Algorithm," *AI Expert*, pages 34–40 (January 1987).

12. Graham, J. M., et al. *Multiple-Function Forward-Looking Infrared (FLIR) Robotic Air Vehicle (U) Volume III*. Technical Report, Texas Instruments Inc, 1988. AFWAL-TR-87-1109; Public Domain Portion Only.

13. Gupta, Anoop. *Parallelism in Production Systems: The Sources and the Expected Speedup*. Technical Report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1984. Contract F33615-81-K-1539.

14. Gupta, Anoop. *Parallelism in Production Systems*. MS thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March 1986.

15. Gupta, Anoop and Charles L. Forgy. *Measurements on Production Systems*. Technical Report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1983. Contract F33615-81-K-1539.

16. Gupta, Anoop and Milind Tambe. "Suitability of Message Passing Computers for Implementing Production Systems." In *Proceedings of the National Conference on Artificial Intelligence (To Appear)*, August 1988.

17. Gupta, Anoop, Milind Tambe, Dirk Kalp, Charles Forgy, and Allen Newell. "Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis," *International Journal of Parallel Programming*, 17(2):95–124 (April 1988).

18. Hu, David. *C++ for Expert Systems*. Portland, Oregon: Management Information Source, Inc., 1989.

19. Hwang, Kai. "Advanced Parallel Processing with Supercomputer Architectures." In *Proceedings of the IEEE*, pages 1348–1379, IEEE, October 1987.

20. Hwang, Kai and Faye A. Briggs. *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, Inc., 1984.

21. iPSC User's Guide. Intel Corporation Publication. April 1986.

22. iPSC/2 User's Guide. Intel Corporation Publication. March 1988.

23. Ishida, Toru and Salvatore J. Stolfo. "Towards the Parallel Execution of Rules in Production System Programs." In *Proceedings of the International Conference on Parallel Processing*, pages 568–575, 1985.

24. Jackson, Victor and Paul Kautz. Class handout distributed in iPSC Concurrent Programming Workshop. Intel Scientific Computers, 1986.

25. Kelly, Michael A. and Rudolph E. Seviora. "A Multiprocessor Architecture for Production System Matching." In *Proceedings of the National Conference on Artificial Intelligence*, pages 36–41, 1987.

26. Lamanna, Cpt Catherine A. *A Performance Study of the Hypercube Architecture*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio. June 1988.

27. Luger, George F. and William A. Stubblefield. *Artificial Intelligence and the Design of Expert Systems*. Redwood City, California: The Benjamin/Cummings Publishing Company, Inc., 1989.

28. McNulty, Christa. "Knowledge Engineering for Piloting Expert System." In *Proceedings of the IEEE National Aerospace and Electronics Conference*, pages 1326–1330. IEEE. May 1987.

29. Milnes, Brian. The Production System Machine Project Member. Telephone interview. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 20 July 1989.

30. Nilsson, N. J. *Principles of Artificial Intelligence*. Palo Alto, California: Tioga Publishing Company, 1980.

31. Norman, Capt Douglass O. *Reasoning in Real-Time for the Pilot Associate: An Examination of Model Based Approach to Reasoning in Real-Time for Artificial Intelligence Systems using a Distributed Architecture*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1985.

32. Oflazer, Kemal. "Partitioning in Parallel Processing of Production Systems," *IEEE*, pages 92–100 (1984)

33. Popolizio, John J. "CLIPS: NASA's Cosmic Shell," *Artificial Intelligence Research*, 1:743–747 (August 1988).

34. Quinn, Michael J. *Designing Efficient Algorithms for Parallel Computers*. New York: McGraw-Hill, Inc., 1987.

35. Riley, Gary. *Implementation of an Expert System Shell on a Parallel Computer*. Technical Report, NASA/Johnson Space Center, 1988. Houston, Texas.

36. Schildt, Herbert. *Artificial Intelligence Using C*. Berkley, California: McGraw-Hill, Inc., 1987.

37. Shakley, Capt Donald J. *Parallel Artificial Intelligence Search Techniques for Real-Time Applications*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1987.

38. Winston, Patrick Henry and Berthold Klaus Paul Horn. *LISP*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1984.

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

a REPORT SECURITY CLASSIFICATION Unclassified		1b RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			
4 PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/89D-6		5 MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b OFFICE SYMBOL (If applicable) AFIT/ENA	7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433		7b ADDRESS (City, State, and ZIP Code)	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AF Wright Aeronautics Labs	8b OFFICE SYMBOL (If applicable) AFWAL/AFAL	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code) Air Force Wright Aeronautics Labs Wright-Patterson AFB, Ohio 45433		10 SOURCE OF FUNDING NUMBERS	

10 SOURCE OF FUNDING NUMBERS			
PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11. TITLE (include Security Classification)

Hypercube Expert System Shell - Applying Production Parallelism   UNCLASSIFIED

12. PERSONAL AUTHOR(S)

William A. Harding, Capt, USAF

13a. TYPE OF REPORT MS Thesis	13b TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1989 December	15. PAGE COUNT 131

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	Expert System	Rete Algorithm
09	02		Parallel Processing	Hypercube
			Pattern Matching	Production Parallelism

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This research investigation proposes a hypercube design which supports efficient symbolic computing to permit real-time control of an air vehicle by an expert system. Design efforts are aimed at alleviating common expert system bottlenecks, such as the inefficiency of symbolic programming languages like Lisp and the disproportionate amount of computation time commonly spent in the match phase of the expert system match-select-act cycle. Faster processing of Robotic Air Vehicle (RAV) expert system software is approached through 1) fast production matching using the state-saving Rete match algorithm, 2) efficient shell implementation using the C-Programming Language and 3) parallel processing of the RAV using multiple copies of a serial expert system shell. In this investigation, the serial C-Lanuage Integrated Production System (CLIPS) shell is modified to execute in parallel on the iPSC/2 Hypercube. Speedups achieved using this architecture are quantified through theoretical timing analysis, and comparison with serial architecture performance results, with earlier designs' performance results, with best case results and with goal performance.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a NAME OF RESPONSIBLE INDIVIDUAL Dr. Gary L. Lamont	22b TELEPHONE (Include Area Code) 513-255-3450	22c OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86   Previous editions are obsolete.   SECURITY CLASSIFICATION OF THIS PAGE

END

FILMED

2-90

DTIC