A Parallel Expert System for the Control of a Robotic Air Vehicle*

Donald J. Shakley
Avionics Laboratory
Air Force Wright Aeronautical Laboratories
Wright-Patterson AFB, OH 45433

Gary B. Lamont
Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

Abstract

Expert systems can be used to govern the intelligent control of vehicles, for example the Robotic Air Vehicle (RAV) which is currently a research project at the Air Force Avionics Laboratory. Due to the nature of the RAV system the associated expert system needs to perform in a demanding real-time environment. The use of a parallel processing capability to support the associated expert system's computational requirement is critical in this application. Thus, algorithms for parallel real-time expert systems must be designed, analyzed and synthesized. The design process incorporates a consideration of the rule-set/face-set size along with representation issues. These issues are looked at in reference to information movement and various inference mechanisms. Also, examined is the process involved with transporting the RAV expert system functions from the TI Explorer, where they are implemented in the Automated Reasoning Tool (ART), to the iPSC Hypercube, where the system is synthesized using Concurrent Common LISP (CCLISP). The transformation process for the ART to CCLISP conversion is described. The performance characteristics of the parallel implementation of these expert systems on the iPSC Hypercube are compared to the TI Explorer implementation.

Introduction

Artificial Intelligence (AI) is concerned with the designing of computer systems that exhibit intelligent characteristics of human behavior. These methods are used when other direct approaches start to deteriorate due to a lack of generality of solution. Examples of such behavior include language understanding, reasoning, and problem solving (Barr and Feigenbaum, 1981). These problems are studied in AI by using a computational model. Many computational models exist for AI problems. A computational model is a formalism used to describe a method of solution. These models present different ways to represent the problem domain. Examples of these models include production systems, semantic networks, frames, and logic (Fischler and Firschein, 1987).

A specializaed area of AI called expert systems development has had considerable success with a multitude of applications. Many applications use production systems or rule-based system structures employing commercial expert system shells. These structures apply heuristics to solving the problem along with algorithmic methods. The success of these expert systems is directly related to the quantity and quality of the associated knowledge base (rules and facts).

Real-time applications exist that involve "hard" problems that currently defy generic algorithmic approaches. Thus, problem solving paradigms from Artificial Intelligence (AI) are being applied to these applications using expert system structures. Due to the computational complexity, however, these approaches have poor computer performance characteristics (Gupta, 1986). Parallel processing seems to offer a possibility to improve expert system computational performance for hard real-time problems.

Parallel processing is the use of more than one processing element to compute the solution to a problem. By using more processing elements, it is hoped that the time to solve the problem is reduced over the time to solve the problem on a single processor. There are several ways to achieve performance improvements in computer systems besides parallel architectures: faster hardware technology, improved serial architectures, better algorithms and code optimization. There are several reasons, however, for looking toward parallel architectures. First, parallel architectures can evolve as fast as hardware technologies become available. Second, many problems associated with AI are computationally "hard" (exponential time-order) or NP-complete. If a problem is NP-complete, this implies that time-order improvements in solution algorithms are unlikely due to many years of computational studies (Aho, Hopcroft, and Ullman, 1974). It should be noted that parallelism can not produce polynomial time solutions to exponential time problems (Norman, 1985). But, it is possible to improve the constant term of the time complexity. Also, the exponential time bound is often times worst case. In AI problems, the use of heuristics can reduce the time complexity of state space searches in specific applications. Third, some problems seem to lend themselves to parallel solutions because the problems decompose easily into independent, computationally equivalent pieces. Production systems, for example, seem to fall into this category because of the large number of rules that must be matched during each production cycle.

This paper discusses real-time processing with application to the Robotic Air Vehicle (RAV) expert systems. Consideration is given to parallel search algorithms and associated knowledge-based structures in the design and implementation of a parallel processing expert system. Experimental and theoretical results are presented.

Real-Time Processing

There are several important issues in the analysis and design of real-time computer applications. One of

* Research supported by the Air Force Wright Aeronautical Laboratories and the Strategic Defense Initiative.

C - 3

the important characteristics is the critical nature of the system execution speed in reference to external events. This can be viewed in terms of the response time of the system to a particular input. For a real-time system, "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" (Norman, 1985). This can vary with the system, but the time is generally relatively small. Relatively small is definitely less than a second and often in the milliseconds or less (Ward and Mellor, 1985).

Another critical characteristic is limited memory capacity. Real-time software typically needs to run in an environment where the size of the program can become a problem. A third consideration is the correctness and integrity of real-time software. The system needs to run correctly and without failure a high percentage of the time (Ward and Mellor, 1985). These represent the most critical issues dealing with real-time systems. In addition, real-time expert systems must also focus on efficient memory interfacing, integration with specific application software processes, efficient inferencing mechanisms, and external temporal commands and events.

The problem with a real-time system on a serial architecture is that the execution time and space requirements are relatively fixed for a given operation. A desirable feature of a real-time system would be a variable time and space performance based on the need. With parallel architectures this could be possible. If a problem needed a faster solution based on the time requirement, then more processors could be added to produce the appropriate speedup. This could only be done if the speedup were predictable.

The need for production systems within real-time systems is growing. With parallel processing of production systems, the execution speed is increasing. For real-time systems this speedup needs to be predictable, so that at any given moment more processors can be brought to bear on a problem to decrease the coefficient of the time complexity of the solution.

An example of a real-time application, which is a current research project at the Air Force Avionics Laboratory, is the Robotic Air Vehicle (RAV). It is an air vehicle with the capability of autonomous flight operation. This vehicle needs the capability for the "intelligent" control of an air vehicle, the capability to plan and replan missions, and the capability to access flight data on various geographical locations (airbases, airports, cities, etc). By "intelligent" it is meant that the system can react to conditions rather than fly on a rigid preprogrammed flight path. A diagram of the system can be seen in Figure 1. This system is an example of hierarchical control which may permit real-time performance due to its decompositional structure. This is also known as meta-level expert system organization. Thus, the course granularity of the multple expert system framework provides yet another source of parallelism.

This study focuses on the "intelligent" flight control components of the system. This component was selected for the feasibility study due to its reliance on production systems and its maturity in relation to the entire research project (Shakley, 1987a). The control of the vehicle can be thought of as a search through a finite state-space over a time period of the vehicle's operation. The problem of intelligent control of a robot is a control-type NP-complete problem that is best suited to be solved by a production system in real-time. Therefore, this system makes an excellent tool for the study of parallel AI search techniques for real-time applications.

The RAV system is an example of an intelligent

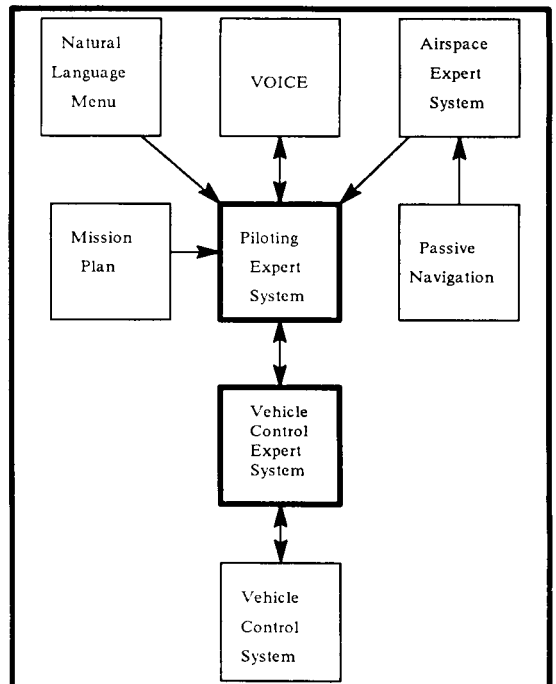

Figure 1. RAV System Diagram. (McNulty, 1987: 1327).

real-time robotic control system implemented using an expert or production system (McNulty, 1987). The purpose of this investigation is to try to increase the performance of the expert system by reanalysis, redesign and reimplementation of the system on appropriate parallel architectures. The hypothesis of this study is that the performance of the RAV expert system can be improved in a predictable and linear manner.

This research is also intended to be a feasibility study of the various issues involved with implementing a parallel expert system. These include implementing an expert system written in Automated Reasoning Tool (ART) on a TI Explorer and on the iPSC Hypercube using Concurrent Common LISP (CCLISP). ART is a knowledge engineering language used in the development of expert systems. CCLISP is a dialect of Gold's Common LISP that has been enhanced to allow for message passing on the iPSC Hypercube. LISP was chosen since it was available on both the TI Explorer and iPSC Hypercube making the transportation of the code from one machine to the other easier. This study is most interested in examining the execution speed of real-time systems that use production system structures. Achieving execution "speedup" rests on the use of parallel algorithms. The results are not intended to specify final real-time execution times, but rather present an analysis of speedup possibility due to parallel processing of production systems.

The current knowledge base for the RAV has been obtained from TI through the Air Force Avionics Laboratory. This includes a basic demonstration routine. Portions of this demonstration are used to exercise the system. The control for the expert system is developed using the basic principles of production system control for an inference engine. The current RAV software uses the Automated Reasoning Tool (ART) as the inference engine (McNulty, 1987). ART can not be used with the parallel environment since it is not available for the iPSC Hypercube. The inference engine is implemented on the TI Explorer Lisp machines where it can be tested against the

knowledge base and the rule execution timing results can be compared to the ART inference engine. The parallel expert system is implemented on the Intel iPSC Hypercube with up to 32 processing elements (PEs) to explore larger degrees of parallelism.

Parallel Search

The advent of parallel computer architectures have addressed the possibility of faster execution of many computer applications. Parallel architectures have brought about new problems as well as the old in terms of software analysis and design. For an application to be implemented on parallel architecture, a way must be found to decompose the problem into component parts. Several important issues are concerned with this decomposition. First, the work must be *distributed* as evenly as possible for an equitable load balancing. Second, the *communication* between the pieces needs to be kept to a minimum. This is to reduce the communication overhead associated with the various processors communicating with each other. However, when this communication occurs, the processors need to be *synchronized* with respect to each other. This is to prevent problems with updating shared variables that can produce erroneous or unpredictable results. Proper synchronization also prevents the occurrence of deadlock between processors (Ishida and Stolfo, 1985).

Speedup is the most common performance measurement (metric) in parallel computing. This is the ratio of the run time of the concurrent software running on n nodes over the the run time of the best serial solution. An application is said to be "perfectly parallel" or have a linear speedup if this ratio is n. Often the speedup approaches the linear speedup, but does not reach it due to the communications overhead between the processing elements (Gupta, 1986). Although rare, speedups have been observed greater than n. This is called *super linear speedup.* At first this seems to be absurd, but upon further study it does seem reasonable. Super linear speedup usually occurs when the application is so large on a serial system that certain overheads are incurred, but when placed on many processors none of the pieces is large enough to incur the same overhead. Thus super linear speedup is observed(Kornfeld, 1981). In addition, there is usually a point at which the addition of more processors does not improve the speedup (Gupta, 1986).

Communications overhead is a large concern is parallel computing. Communication takes several forms. The first is the time to set the job up on the parallel system or the time to distribute the work. The second involves the time needed to collect the results of the job. The third is the communication needed between the processors during the running of the job. An important measurement is the time a processor is communicating versus processing. This measurement along with the setup time and cleanup time gives a good indication of the overhead associated with the parallel process.

Load balancing is another important criteria for parallel computing. This is the percentage of the total processor power that is used during the job. A perfect load balance would be one in which all the processors are busy all the time. This perfect balance is impossible due to two factors. First, depending on the connection network for the processors, setup and cleanup processes provide for times when not all the processors are busy. Second, there is usually some fraction of the job that is inherently serial. This part of the job has to be performed on one processor while the other processors are idle. These two factors are innate to the problem. Poor load balancing can also be designed into a problem due to a improper decomposition. This balancing can occur in two ways, either static or dynamic. Dynamic load balancing is adapting the load to the current state of processing. This is a difficult task due to the addition overhead incurred and the meta-level control needed.

Several other performance measurements are needed to baseline a production system. These include 1) the number of productions or rules, 2) the number of working memory elements or facts, 3) the composition of the rules which includes the number of clauses in the LHS and RHS of the rule, and 4) the average number of rules eligible to be selected on a given cycle. These are but a few basic components, other characteristics depend on the system and inference engine being examined.

Since production systems are a type of search, parallel decomposition techniques for search problems can be applied to production systems. So, production systems can be decomposed along the control functions like a branch-and-bound or it can be decomposed by its data. In the case of a production system the data can be thought of as two parts (Figure 2). The first part is the facts or the working memory (WM). Although the working memory can have several meanings in this context it refers to the initial facts and axioms as well as the facts added due to the firing of rules. The second part is the rules or the production memory (PM). These two parts are not always distinct, but can overlap. For example, the result of a rule could be the addition of a new rule. The reasons for making the distinction in the types of data is that in some cases it is much easier to decompose the rules than it is to decompose the facts. The latter requires data dependencies to be worked out while the former requires less restrictive decomposition considerations.

As described earlier the concurrency available in decomposing the functions is limited. This is particular true for production systems where over 90% of the time is spent in the match function (Gupta, 1986). So the main emphasis is placed on the decomposition of the data. The methods for implementing a production system tend to center around ways to decompose the rules (PM) and the facts (WM). This has lead to several algorithms to accomplish this decomposition and their placement on separate processors. Examples include a Full Distribution of Rules, the Original DADO, Miranker's TREAT, and Fine Grain RETE (Shakley, 1987a). These algorithms are generally at the level where the underlying inference engine structure is unimportant. The methods are more concerned with the dependencies of the rules on each other and the facts (WM). The initial prototype knowledge structure consisted of a frame-based data structure of facts and a list structure of rules. The impact of this selection is analyzed in a subsequent section.

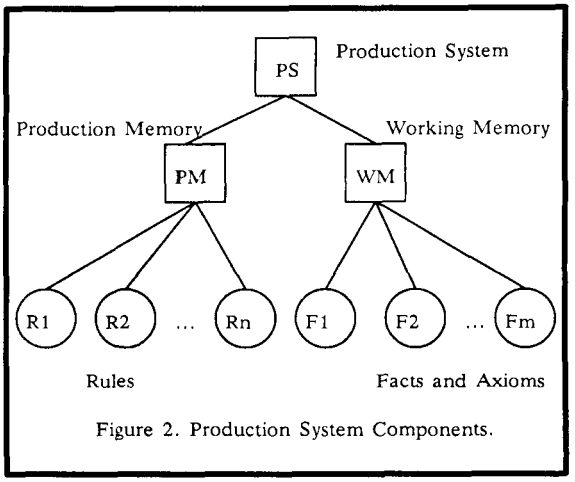

Figure 2. Production System Components.

Analysis and Design of Inference Engine

The main structure of the piloting control is a layered series of two expert systems. Each expert system has several components organized by functionality (Figure 3). These components provide a source of data independence of rules and working memory. The system contains an "average size" production and working memory. The system contains over 350 rules. The working memory consists of schemata which are frame-like structures. Each frame contains slots that hold the individual facts. There are approximately 160 schemata. The average number of slots per frame is approximately ten, therefore the total number of facts is about five times the number of rules.

The requirement for an inference engine is to perform the basic production system cycle: match, select, and act. [This cycle is embodies the resolution process (Nilsson, 1980).] This inference engine software should be able to match the rules of the RAV expert system with the facts in working memory. It should select one of these rules and add the results of the RHS of the selected rule to the working memory.

The current RAV system implemented on the TI Explorer uses the Automated Reasoning Tool (ART) for this process. The lack of availability of ART on the iPSC Hypercube requires that another control process implementation be developed. However, the new implementation should be compatible with the ART rules and working memory structure.

ART is a very complex and extensive tool. To try to rebuild the generic ART system on the hypercube would require a prohibitive development time. Therefore, simplicity of design is a critical component. The new control process should only provide the functionality of ART that the RAV requires.

The previous designed inference engine was modified for parallel implementation. The algorithm for the parallel design is presented in Figure 4. The actual change in the design to the serial inference engine is small. The changes occur in the select and act phase. Each processing element (PE) or node of the parallel design needs to report the rule selection to the system. The PE then has to coordinate with the system in order to act to update the working memory. Three alternatives seem appropriate: a star, a binary tree, or a spanning tree. With the star, one node acts as the central point with all other nodes communicating with that node. This would require longer than nearest neighbor communication or one node hops. The binary tree can be implemented with nearest neighbor communication, but only on higher dimension cubes. The spanning tree (Figure 5) offers the appropriate functionality with nearest neighbor communication. A rule selected on node 14 would be sent to node 6. At node 6, this received rule would be added to the agenda and node 6 would select a rule. This continues up the tree until node 0 receives all the selected rules from its children. It then selects an overall rule and passes it down the tree to all its children. Each child then passes the selected rule to its children until all nodes receive the selected rule for firing. The communication network for the flow of information is designed as a spanning tree. The reason for this type of tree is because it preserves nearest neighbor connections and the height of the tree is the logarithm base 2 of p where p is the number of PEs. A node only communicates with other nodes a distance of one away and the length of a path from the bottom of the tree to the top is the dimension of the cube.

The only design aspect remaining is the methodology of placing rules of the RAV production system components on the parallel system's PEs. The first design to place the rules on the PEs, decomposed the rules by component on the different PEs. This was unsatisfactory since this produced an uneven load balance. Therefore, a more appropriate decomposition of the rules was to equally distribute the rules to the various PEs (Figure 6) .

RAV Components

PES	VCES
Autopilot	Airwork
Commnav	Airwork-New
Departures	Autopilot
Hold-Arc	Elevation
Intercepts	Heading
Intercepts-New	Speedbrake
Landings	Throttle
Mission	
Recover	
Takeoffs	
Targets	
Target-All	

Figure 3. RAV Components.

1. Initialize: Place a copy of the simple inference engine on each PE. Place a copy of WM on each PE. Place a subset of the PM on each PE.

2. Repeat until done;

3. Match and select on each PE.

4. Report selection up tree.

5. Overall selection made at root node.

6. Broadcast WM change to all PEs.

7. end repeat;

Figure 4. Parallel Inference Engine Algorithm.

Low-Level Design & Implementation

The parallel inference engine was implemented in CCLISP on the iPSC Hypercube. Since the serial inference engine was implemented on the TI Explorer using Common Lisp the changes needed due to language differences were minimal. CCLISP (Broekhuysen, 1987b) was not as extensive as Common Lisp on the TI Explorer (Explorer, 1985). For example, CCLISP did not support CADDDR, but this was easily changed. The language issues simply did not provide a major obstacle. There was, however, major effort involved in implementing a parallel inference engine. This centered around the communication between nodes.

The parallel design of the inference engine required that the selected rule from all the nodes be collected at one node for the final selection, and then that selection needs to be passed to all the other nodes. This is accomplished with a spanning tree.

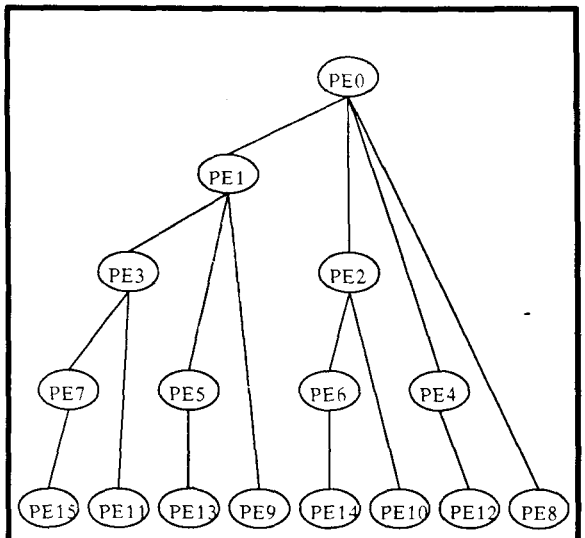

Figure 5. Spanning Tree Connections.
(Brandenburg and Scott, 1986).

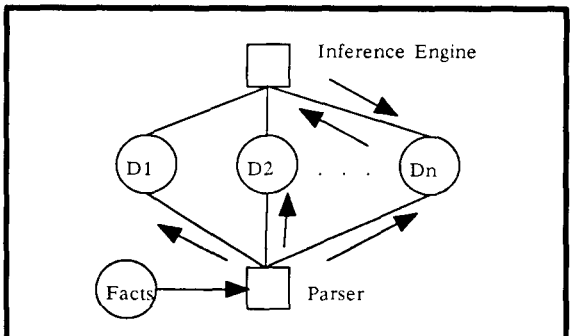

Figure 6. Rule Distribution.

The algorithms for determining the parent and children nodes of the tree dependencies uses a logical "or" of the binary node numbers. This was difficult to implement in LISP, so a table look-up was used. This proved to be very simply and efficient, but somewhat inflexible since only node zero can be used as the root node.

The message passing within CCLiSP presented some problems. There were several ways to pass messages. They ranged from low level message passing to high level fast loading (FASL) node streams. The low level message passing required that the length of the message length be known. This proved to be a major limitation, given that the messages to be passed would be variable length rules. Therefore, the high level FASL node streams were selected for their abstraction. These streams did provide a problem. There was no defined way to do a receive-wait. This allows a process to enter receive mode until a message is received. This process is very convenient for synchronizing nodes. This function had to be built using a loop doing repeated receives until a message was received from another node. One other note concerning this process. The documented function 'listen' was not implemented (Broekhuysen, 1987b). This would have provided a method to test the message buffer for an incoming message without actually doing a read.

The changes to the actual serial inference engine were small and confined to a small number of modules. The first module had to be changed to provide the proper termination test. This is important to insure that the individual nodes terminated only when no overall rule was available, not just when the node found no matches. The other module had to be changed to incorporate the communications with the other nodes. Several other routines were needed to assist this latter module to make the communication.

The RAV system consisted of the original components of the RAV expert system designed by TI. In its original form it consisted of a series of plans, needs, and schemata which was a higher level abstraction than the ART rules (McNulty, 1987). The plans and needs were then "compiled" into ART rules for execution using software developed by TI (Lystad, 1987). The only way to get the schemata and rules from the RAV Plans and Needs was to compile them into files rather than into the ART system. From there, the rules and schemata are then transformed into a format that the serial and eventually the parallel inference engine could accept. This transformation was partially automated with a routine and further transformed by hand to come up with the final format compatible with the implemented inference engine. The total translation was not done programmatically due to the complexity of the software involved to parse and recognize the various ART syntax forms.

Experimental Analysis of Results

The only test suite available was a demonstration developed by TI midway through the development of the system. In fact, the expert system used in this study was not complete and was only a demonstration prototype (Graham, 1987). This demonstration was considerably lengthy and required the perfect execution of all the rules and implementation of all associated ART functionality. The alternative was to develop small prearranged sets of facts that would trigger a subset of rules. This was the preferable choice since the inference engine could not deal with all the rule formats in their entirety. The complete demonstration was not used, therefore a full comparison of the systems could not be done.

The code and expert system for the parallel RAV system was transported to the iPSC Hypercube from the TI Explorer to a microVAX to a VAX across the Defense Data Network (DDN) to the AFIT VAX and finally to the iPSC Hypercube (Shakley, 1987a). This was perhaps the most "trying" of the problems associated with this whole implementation. This was because of the many machines that had to be traversed to get the code from the TI Explorer to the iPSC. This was only done after the tape-to-tape transfer failed due to a mismatch in tape-formats.

The code was analyzed for its effectiveness and correctness by comparing its execution with that of the ART system. The code performs slowly compared to systems like ART which runs at between 2-30 rules per second (Gupta, 1986). On the Explorer, the inference engine plods along at about one cycle every 30 seconds or 2 rules per minute. This is with the smaller rule base. With the larger rule base, the system runs one cycle every 113 seconds. On the iPSC Hypercube, the serial system runs at one cycle in about 11 seconds with the smaller database and in about 79 seconds with the larger database. This situation involves two phenomena that need explanation. First, why does 60 extra rules slow the process down so much? The reason for this is in the format of these rules. They are rules that have a variable binding on the schema name. This means they must go through the list of schema names looking for a match. These rules can not take ad-

vantage of the indexing created by the frames. This is a process not handled well by the inference engine. Second, why does the TI Explorer go slower than the iPSC Hypercube? The surface appearance is that the Common Lisp on the Explorer is much more extensive than that on the iPSC Hypercube. These would create more overhead on the Explorer and allow the hypercube to process faster.

The speedups are far less than linear (Figure 7).

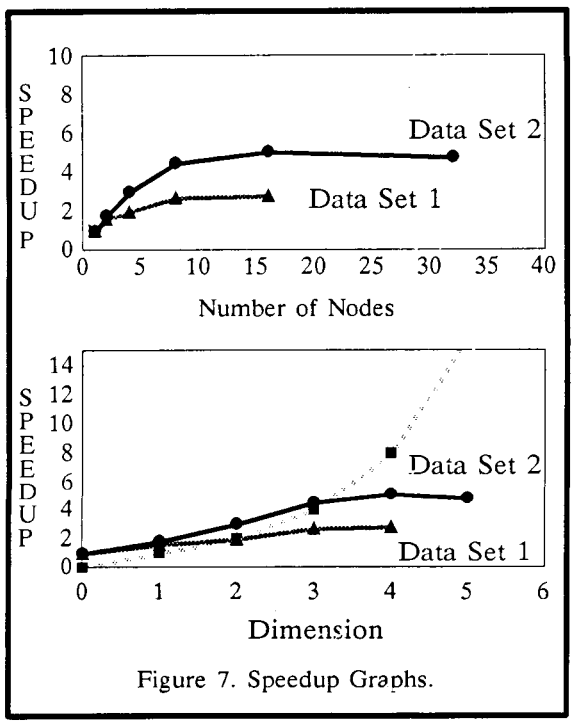

Figure 7. Speedup Graphs.

The speedups taper off with the dimension of the cube. The select time on some of the nodes exceeds the match time on the node. In some instances, the select time far exceeds the match time. There are two explanations for this phenomena. The first is that the problem size is too small for the higher dimension cube. This can be seen from the first data set. On the higher dimension cube, the match time is less than the select time for one series of communication. The result is a much longer cycle than what would be expected from a linear speedup. The second cause for this slowdown is poor load balancing. Each node has an equal number of rules and each rule gets checked for a match on each cycle. The reason the load is imbalanced is due to the composition of the individual rules. The rules have different numbers of clauses that causes each rule to have a variable length match time. Also, the order of the clauses creates different match times. Even though a rule may have a long list of clauses to match, if the first one fails, then the rule matches quickly. Finally, the different types of matches take varying lengths of time depending on the format of the clause. A match involving a schema variable match takes much longer than a simple slot match.

Theoretical Analysis (Shakley, 1987b)

The computational complexity of a production system can be divided into several components. These correspond to the match, select, and act phases. The last component's (act) computational complexity is easiest to analyze. During the act phase, one rule has been selected and the clauses of the consequent are either added to or retracted from the fact database. Although, it depends on the data structure of the facts, it is a polynomial operation. In most cases, the complexity is either constant or linear. This would occur with any type of indexing or linked list data structure of facts. The computational complexity of the other two components is more difficult to analyze and in fact depends on the types of clauses used within the rules.

A set of parameters is needed to discuss the computational complexity of the production system. These parameters are found in Figure 8.

nR - Number of Rules
nF - Number of Facts
lC - Maximum length of a rule clause
nA - number of antecedant clauses
nC - number of consequent clauses
nV -number of variables

Figure 8. Production System Farameters.

The first type of production system considered is one with only constant terms in the rule clauses. The initial observation about the system is that the fact database is unordered. Therefore, the entire database must be searched for each clause of each rule. The worst case time complexity is then O(nR*nA*nF) for the match phase. This complexity grows as the number and complexity of rules grows and as the number of facts grows. In a monotonic system, one in which only facts are added, not retracted, this complexity grows as the system operates adding more facts.

The select phase can take two forms. In the first, the select phase stops whenever the first rule is matched successfully. The second form waits until all the rules have been matched, and then selects one rule from all the successfully matched rules. In the worst case, the matcher matches every rule's clauses to every fact. Therefore, the two forms of the select only affect the complexity of the select. In the first form, the complexity is constant and in the other it is O(nR).

The first form has some disadvantages that make it unattractive. By taking the first rule that matches successfully, the production system is creating an implied order of the rules. This creates preconditions to the subsequent rules. For a rule to be selected, not only does the condition of the rule have to be meet, but also the conditions of the previous rules can not be meet. Therefore, the second select form creates a certain sense of "randomness" to select process. Therefore the overall complexity of a production system with constant clauses is O(nR*nA*nF) for one match, select, act cycle. It should be noted that the entire production system is NP-Complete and if retractions of facts are allowed the production system is not even guaranteed to terminate.

The second type of rule clause considered is one in which variables are allowed. This type of clause has the potential to match with many different facts. The match complexity of these types of clauses are the same as in the previous case: O(nR*nA*nF). However, the select phase is more complicated. There is a successfully matched rule for each instantiation of each variable. Therefore, the select phase has a worst case time complexity of O(nR*nV*nF). Where the number of variables equal the

number of atomic elements within the clause then nV equals nA times the lC. This produces a complexity of O(nR*nA*nF*lC). Both the preceding cases assume that there is no order to facts. Also, there have been no simplifying assumptions about the structure of the database.

The first step in simplifying the fact data structure is to provide a frame structure for the facts, an Object-Attribute-Value (OAV) structure. This provides a way of ordering the facts. In this way, the value can be accessed by indexing into the fact data structure using the object and the attribute. A clause in a rule can now be matched in constant time, O(1). Therefore, the time to match is reduced to O(nR*nA). The time to select is still O(nR) where no variables are associated with the clauses. If a variable is associated with the value within a clause. then the time to match is the same. However, the time to select is O(nR*nV) or O(nR*nA) since there is only one variable per clause. The problem arises if variables are introduced into either the Object or Attribute fields of the clause. If both all of the OAV are variables then the system degenerates into the case with no ordering of the facts. The time complexity for the system is O(nR*nA*nF*lC). The only case left is if two of the three are variables. The match time O(nR*nF*nA) in the worst case. Each rule must be matched with each fact for each clause in the rule. In conclusion, this simplifying feature alone only save time when used with only one variable item per clause and that item must be the value of the OAV. It should be noted that if the variable is in another field, then the indexing scheme could be changed to account for the change in the clause structure.

From the previous example, it can be seen that better performance can be achieved if the facts can be organized into frame structures that allow for the indexing of facts. This benefit, however, only allows for a variable in the value field. Also, on each cycle this entire match, select, act cycle has to be reaccomplished. A state saving feature within the production system could reduce the time for all but the first match, select, act cycle. This data structure stores from cycle to cycle the rules that had previously matched. Then only those rules that were affected by the selected rule would need be considered on each pass. Therefore, each fact needs an associated list of affected rules. Then, when this fact was changed via a rule only the affected rules would need to be considered for matching. In the worst case, every rule affects every other rule creating the situations in the previous cases with the appropriate time complexities. In this case, this simplification has no improvement. However, rare is the system where every rule affects every other rule. The reduction is actually within the coefficient of the equation. The time complexity for the OAV case is O(nR*nA). The coefficient of this complexity is one. With the above simplification, each subsequent cycle is (1/aR)*nR*nA where aR is the number of affected rules. This method also reduces the cycle time for the multiple variable case of OAV. In the original case, the time complexity was O(nR*nA*nF) for each cycle of the production system. This still holds for the first cycle, but every cycle afterwards is considerably less. In fact, each cycle is O((1/aR) *nR*nA). This is true even for the rules with multiple variable clauses, since a rule can only affect a limited number of instantiations of a multiple variable clause rule. Only when a rule affects all the instantiations of a multiple variable clause rule is the complexity of that following cycle increased to O(nR*nA*nF). The state saving technique does increase the space complexity of the system. Each fact has to have a list of pointers to each rule that is affected by the fact

What does this all mean in picking an inference data structure and strategy? The answer lies in the problem domain. It revolves around the characteristics of the system: the number and complexity (number of antecedents) of the rules, and the number and format of the facts. If the format of the facts is "random" in that no ordering can occur, then the system is doomed to poor performance. If on the other hand the facts can be organized into frames or some form of OAV where the value is the only variable field. Then an order of magnitude improvement can be obtained. If variability is allowed in other fields of the frame, then poor performance is reintroduced. If a state saving feature is introduced, then in the average case the performance can be improved on all, but the first cycle of the production system. This state saving feature, however, does require additional space. Therefore, a mixture of inferencing techniques could be the "best" solution.

A study consisted of analyzing the various types of inferencing with the different types of clauses in the Robotic Air Vehicle Expert System. The expert consists of over 300 rules and over 160 frames with approximately 10 slots per frame. The system was originally implemented with an unordered list of facts. The system performed "well" with a subset of rules and facts that did not exceed approximately 10 rules and 50 facts. That is rules that contained only one variable item per clause. One cycle took approximately 1-2 minutes. The system was then upgraded to a frame based data structure [In the first case, each slot was transformed into a single fact and treated as unordered]. This system performed "well" with all rules and frames. The system completed a cycle in approximately 30 seconds. However, this system had problems with clauses that had multiple variable items. These rules took approximately 1 minute per rule. The cycle time was never calculated due to the extremely long time per rule. It should be noted that the rules of this type were limited in numbers and were less than 50. The next step was to create a state saving system. This system only matched and considered the rules that were affected by the previously selected rule. This required that the rules be compiled into the data structure of the facts, so that each fact could point to the rules that were affected by the modification of that fact. The first cycle of this system still took approximately 30 seconds to complete without the multiple variable items. Then each subsequent cycle took approximately 5-10 seconds thereafter. The sequence of different inferencing techniques confirmed to the prediction of the computational complexity analysis. Further, work is still desirable to provide an even broader base of sample data. This would help to confirm even further the time complexity analysis.

Conclusions

This research study investigated the feasibility of parallel architectures to improve the performance of potential real-time software. In particular, the feasibility of parallel architectures to improve the NP-complete problem of state space search in the form of a production system. The RAV expert system is an example of such a system.

The speedup results from the inference engine were disappointing, however, it did show that speedups were possible. The speedups suffered from a combination of two factors. The first was a relatively small problem compared to the communications overhead. It was observed that for a system with greater than eight nodes, the time to perform the match cycle on a node was less than the time to communicate with the other nodes. Also, the speedup suffered due to a load imbalance. The method for decomposing the rules proved to be unsatisfactory. The method did not take into account the variability among rules. Real-time performance was not achieved as was anticipated. With improvements in the inference engine and

load balance, significant improvements could be possible.

The performance of the iPSC Hypercube in comparison to the TI Explorer was fairly positively . The iPSC Hypercube performed about twice as fast as the TI Explorer on the inference engine. However, this seems to be due to the simplicity of the LISP on the iPSC Hypercube.

The inference engine developed in this study performed adequately. The inference engine fired a rule about once every 30 seconds or at a rate of just under 2 a minute on the TI Explorer. The engine fired a rule one every 18 seconds on a single node of the iPSC Hypercube or just over 3 rules a minute.

Recommendations

This study probably raises more questions than it answers. Beginning with the serial inference engine. An area of study would be the performance of inference engines. The characteristics of inference engines and their performance would have been invaluable to this research. More work could be done to improve the inference engine in this study. The inference engine in this study could be redone using the Rete algorithm. The benchmarking of inference engines and inference engine techniques would be valuable. Also, with regard to inference engines, this study started to automate and simulate the functionality of ART on the iPSC Hypercube. The further development of the process could provide a valuable tool for expert system development. The expert system could be developed on the TI Explorer using ART and transferred to the iPSC Hypercube for performance studies.

There needs to be better ways to characterize the work needed to match a rule so that more effective iPSC Hypercube load balancing can be performed (Bailor and Seward, 1988). This would depend on the structure of the inference engine, the structure of the rules and the structure of the facts for efficient partitioning and ditribution of data. An automated means should be developed to handle this load balancing. This automated method could be used for adaptive time-based load balancing. In addition, temporal dependency mechanisms could be used to achieve this adaptive load balance.

The current study used a complete set of facts on each node and only one rule was fired at a time across the entire network. The RAV software showed promise for further levels of concurrency. The structure of the RAV expert system shows great promise in firing several rules in one cycle of the inference engine. This could provide a significant time and space savings. The RAV expert system is to operate in a real-time environment. This means a varying time requirement for operations. The implementation of an automatic way to dynamically increase the speed of a computation through a meta-level of knowledge and control would be valuable to the design of real-time software.

Bibliography

Aho, Alfred V., John E Hpcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley Publishing Company, 1974

Bailor, Paul D. and Walter D. Seward. "A Generalized Strategy for Partitioning and Distributing Data in Data Parallel Algorithms," Unpublished report. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, June 1988.

Barr, Avron and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence, Volume I.* Standford, CA: HeurisTech Press, 1981.

Brandenburg, Joseph E. and David S. Scott. *Embeddings of Communication Trees and Grids into Hypercubes.* iPSC Technical Report No 1. Intel Scientific Computers, Beaverton, OR, 1986.

Broekhuysen, Martin, editor. *Concurrent Common Lisp User's Guide Version 1.1.* Cambridge, MA: Gold Hill Computers, 1987a.

-----. *Concurrent Common Lisp Reference Manual Version 1.1.* Cambridge, MA: Gold Hill Computers, 1987b.

Explorer. Lisp Reference (2243201-0001). Texas Instruments Incorporated, Austin, TX, June 1985.

Fischler, Martin A. and Oscar Firschein. *Intelligence: The Eye, the Brain, and the Computer.* Reading, MA: Addison-Wesley Publishing Company, 1987.

Gupta, Anoop. *Parallelism in Production Systems.* PhD dissertation. Carnegie-Mellon University, Pittsburgh, PA, March 1986.

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

Kornfeld, William A. "The Use of Parallelism to Implement a Heuristic Search," *Proceedings of the 7th International Joint Conference on Artificial Intelligence* 575-580 (1981).

Lystad, Garr S. "The TI Dallas Inference Engine (TIDIE) Knowledge Representation System," *Proceeding of the IEEE National Aerospace and Electronics Conference* 1348-1351 (May 1987).

McNulty, Christa. "Knowledge Engineering for Piloting Expert System," *Proceedings of the IEEE National Aerospace and Electronics Conference* 1326-1330 (May 1987).

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

Norman, Capt Douglas 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 AFIT/GCS/ENG/85D-12. School of Engineering, Air Force Institute of Technology (AU). Wright-Patterson AFB, OH, December 1985.

Shakley, Donald J. *Parallel Artificial Intelligence Search Techniques for Real-Time Applications,* MS Thesis AFIT/GCS/ENG/87D-24. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1987a.

-----. *Order-of Analysis of Production Systems.* Unpublished report. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1987b.

Ward, Paul T. and Stephen J. Mellor. *Structured Development for Real-Time Systems Volume 1: Introduction and Tools.* New York, NY: Yourdon Press, 1985.