Report Number: WUCS-88-24

1988-10-01

A Survey of Three Applications of Parallelism in AI

Rosanne M. Fulcomer and William E. Ball

Once the BEMAS [13] system was completed and recorded in Common Lisp, research efforts were channeled toward three primary areas. This report will present a briefly review of some research in these areas, which are: parallelizing truth maintenance systems, parallelizing production systems, and parallel search. The area of parallel search has been studied by many over the past years and we will only present current research that has been accomplished. This review represents the beginning research into the development of a parallel inference model.

A SURVEY OF THREE APPLICATIONS OF PARALLELISM IN AI

Rosanne M. Fulcomer and William E. Ball

WUCS-88-24

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

This work was supported by McDonnell Aircraft Company under contract number Z71021.

Contents

1 Introduction

Once the BEMAS [13] system was completed and recoded in Common Lisp, research efforts were channeled toward three primary areas. This report will present a brief review of some research in these areas, which are: parallelizing truth maintenance systems, parallelizing production systems, and parallel search. The area of parallel search has been studied by many over the past years and we will only present current research that has been accomplished. This review represents the beginning research into the development of a parallel inference model.

2 Parallel Truth Maintenance Systems

A major problem facing AI techniques today is that computation time for inferencing and related activities is combinatorially explosive, when based on sequential reasoning. This means that large knowledge bases become almost impossible to use in practical applications that require a fast or predictable response time (such as in any real-time system). However, by designing and using efficient parallel algorithms, we hope to generate a significant increase in speed with a guaranteed upper bound on computation time. This approach will make it possible to apply these powerful logical techniques to larger and more complex information systems, thus producing more accurate, reliable, and complete answers for many applications. Since we have done research in the area of truth maintenance, we will begin by discussing how parallelism can aid this type of computation.

2.1 Parallel JTMS

The paper "A Diffusing Computation for Truth Maintenance" by Charles Petrie [23] was published in the 1986 IEEE International Conference on Parallel Processing. What is described in the paper is a parallel algorithm for Doyle's Truth Maintenance System [7] (termed JTMS for this paper) using the technique of diffusing computation given by Dijkstra and Sholten [5].

To take advantage of the proofs supplied by Dijkstra using diffusing computation, Petrie proposes a mapping of each node in a TMS to be a separate processor. Justifications are then represented as directed arcs from

antecedents to the consequent; only a single consequent is computed by each rule in this representation. The purpose of using diffusing computation is to give a status assignment of IN or OUT to each node corresponding to the labeling that is performed in Doyle's algorithm.

Following is a review of the basic TMS as proposed by Doyle. A TMS is used along with an inference engine (IE) to maintain a consistent set of beliefs and inferences. The IE computes its inferences using knowledge. These inferences are then passed to TMS, which creates a node for each belief and maintains the dependencies among these beliefs. Queries to the IE can be directed to the TMS, which can give a truth value for the belief queried and reasons why this truth value is applicable. Doyle's JTMS (Justification TMS) uses the closed world assumption along with the basic nonmonotonic reasoning facilities. A justification corresponds to the support list of a node in Doyle's original work. Since Petrie does not make use of conditional proofs, also discussed in the original JTMS, we will not discuss them here. But we wish to point out that usually an implementation of Doyle's work is not necessarily complete without the implementation of conditional proofs, or something that works like a conditional proof.

Each node in the TMS is to be assigned a status of IN or OUT, where an IN label means the node is believed to be true, and an OUT label means either it is not known what truth value the node has or the node is not believed to be true. Associated with each TMS node is a set of justifications, in which each justification contains an INSET and an OUTSET. An INSET contains a reference to those nodes which must be believed in order for the node to be labeled IN. An OUTSET contains a reference to those nodes which must not be believed in order for the given node to be labeled IN. A justification in the justification set of a given node is valid if every node referenced in its INSET is labeled IN and every node referenced in its OUTSET is labeled OUT. If one justification in the justification set is valid then the given node is labeled IN, otherwise the node is labeled OUT. If the INSET and OUTSET of a justification are empty, then the given node is considered to be a premise, or fact. If the set of justifications is empty, then there is no support possible for the given node, so it will always be labeled OUT. The last definitions to be given is that of consistency and well-foundedness. A TMS network is consistent when each node is assigned a status of IN if and only if it has at least one valid justification and OUT otherwise. A TMS network is well-founded if no node is in its own believed repercussions, where the set of

believed repercussions is the transitive closure of the IN consequences which the node supports.

Status assignments are accomplished by the use of diffusive computation. Petrie admits that his computation is incomplete is the same sense that Doyle's is incomplete, which is with respect to unsatisfiable circularities being introduced into the network by a new justification. This in turn would create a graph for which no consistent assignment of statuses can be found. Petrie assures that his algorithm will terminate even if a consistent labeling is not found. It will also assign a consistent labeling if possible even if the labeling is not well-founded.

In this approach, each processor stores the set of justifications for that node. Messages are sent to consequences and signals are sent to antecedents in the justification set. When all nodes are in a neutral state, a node becomes the environment, or root node, when its status changes from OUT to IN by the entrance of a new justification (passed from the IE). One must assume that all nodes are in their neutral state, though this is not stated anywhere in Petrie's paper. If this assumption is not made, then it would allow recursive diffusing computation with multiple environments. The proof that this computation will terminate is not addressed in the Dijkstra and Sholten paper.

The algorithm begins by the root node issuing a "NIL sweep", which sets the status of the transitive closure of consequences to NIL. This is done by the use of diffusing computation and presents no problems in itself. Once the "NIL sweep" is performed, diffusing computation begins for the assignment of IN/OUT status assignments.

The definition of the engager corresponds to Dijkstra's in [5], in which the sender of the first message to a successor is called the engager of the successor and this sender will be the last to receive a signal from the successor before the successor becomes neutral, i.e. neither sending messages nor signals. Once a node receives its first message from its engager, it checks to see if its status will change. NIL status will change to IN or OUT, so all statuses will change at least once. If the status is unchanged, a signal is sent back to the predecessor and no messages are sent to the consequents. If the status of the node is changed, it sends messages to its consequents. A processor also replies to any sender "immediately" if it has already received another message to which it has not replied, i.e. it is engaged, if the message does not change the node's current status assignment. A node replies to the engager (under

normal circumstances) when it receives replies from all of its consequences or successors.

To detect unsatisfiable circularities, Petrie proposes the following approach. Along with a status message, the sending node sends a list of the transitive closure of its antecedents or predecessors to its consequent node. Then if the consequent node appears in the list it is sent by an antecedent that changes its status, it signals a "trouble reply" to its engager.

In [12] we present the problems with this algorithm and a solution.

2.2 Parallel ATMS

Efforts have been made to parallelize the other major TMS systems. Dixon and de Kleer [6] recently presented their algorithms to parallelize the Assumption Based Truth Maintenance System, or ATMS [2] [3] [4], and Vilain [29] also presented his paper on parallelizing the McAllester's Logic Based Truth Maintenance System, or LTMS [21]. The next two section describe both attempts.

de Kleer's ATMS is a propositional inference engine designed to simplify the construction of problem solvers that search complex search spaces efficiently. ATMS provides a general mechanism for controlling problem solvers by explicitly representing the structure of the search space and the dependencies of the reasoning steps. However the ATMS achieves problem solver efficiency by propositional reasoning about problem solver steps, and for large problems these operations comprise a significant amount of computation themselves. ATMS is the basis for the Multiple World System used in KEE^{TM}.

Massively parallel computers provide orders of magnitude more computational power that serial machines by connecting thousands or millions of processors with some form of communication network. The processors are kept very simple, operating from a shared instruction stream and providing a limited instruction set. The difficulty with such massive parallelism is making use of such a machine, i.e. distributing the tasks among the processors so that the limited computational power and communication available at each processor are well matched to the operations that need to be performed. Dixon and de Kleer's paper shows how the propositional reasoning performed by the ATMS is well suited to massively parallel hardware. This is shown by implementing the ATMS on a Connection Machine, in which

the implementation provides a superset of the functionality of the original sequential implementation.

When ATMS is used, problem solving becomes a cooperative process. First the problem solver determines the choices to be made and their immediate consequences, and transmits these to the ATMS. Then the ATMS determines which combinations of choices are consistent and which conclusions they lead to. With these results, the problem solver explores additional consequences of those conclusions, possibly introducing new choices. The cycling between the problem solver and the ATMS ends when a set of choices is found to satisfy the goal or all combinations are proven contradictory. Some definitions are as follows. An *assumption* is a problem state representing primitive binary choices. A *node* is a problem state corresponding to propositions whose truth is dependent on the truth of the assumptions. A *justification* is a dependency relationship among assumptions and nodes as determined by the problem solver and presented to the ATMS.

To the problem solver, nodes and assumptions represent propositions in the problem domain. Their structure is used by domain-specific inference rules and the results of inference are recorded as justifications. To the ATMS, assumptions and nodes are atomic. The only relations among them are the justifications the problem solver has reported to the ATMS so far in the computation. Thus, all the relevant domain knowledge and structure can bee represented with justifications.

Other important definitions utilized by the parallel ATMS are as follows. An *assumption space* is a boolean n-space defined by the set of all assumptions, in which each point corresponds to some total assignment of truth values to assumptions. A *support* is a point in the assumption space that supports a node if the truth values of assumptions at that point together with the justifications logically entail the node's truth. *Consistency* means a point is consistent if the truth values of assumptions at that point are consistent with the justifications. Otherwise, if a contradiction is entailed, the point is inconsistent. The *extension of a node* is the subset of the assumption space that supports that node, excluding inconsistent points. *IN or OUT* means that a node is labeled IN if it is supported by at least one consistent point in the assumption space, otherwise it is OUT.

The ATMS performs four basic operations for the problem solver. They are to create a new assumption, create a new node, record a justification, and return a node's extension. In addition, the ATMS maintains an efficient

representation of each node's current extension, and the set of points discovered to be inconsistent. Quickly updating these representations after each operation is the key to any ATMS implementation. Creating a node and returning an extension require no changes. Creating an assumption doubles the assumptions space and doubles the extensions of each node. Adding a justification can change node extensions in complex ways. The extension of the consequent of a node must include the intersection of the extensions of its antecedents. If there is no circularity of justifications, then the extension of each node is the union over all its justifications. If the justifications are circular the ATMS must find the set of minimal extensions that satisfy the constraints made by the additional justifications. To compute the new extensions when a justification is added, ATMS uses a form of constraint relaxation.

The operations on extensions are as follows. The first is to compute the intersection of the antecedents' extensions. Then it should be determined whether the result is subsumed by the current extension of the consequent. If it is not subsumed, compute the new extension of the consequent from the union of the old extension with the intersection of the antecedents' extensions. Lastly, remove a set of points that has been discovered to be inconsistent from each node and double the extensions of every node when a new assumption is added.

The Connection Machine [16] was chosen to be the architecture on which to implement the ATMS. This architecture consists of from 16K to 64K processors, each with 4K to 64K bits of memory. Each processor can emulate several processors, allowing, for example, 256K virtual processors on a 64K machine, each with one quarter the memory of a real processor. The processors execute from a single instruction stream produced by a host computer. The basic operations of the Connection Machine are as follows: a general bit-field combine operation, a very low overhead bit move operation between adjacent processors, a higher-overhead general bit move operation from each processor to any other processor, implemented by special purpose routing hardware, and an operation that ORs together one bit from each processor. Not all processors need to execute every instruction. Processors may be individually deactivated and reactivated as determined by computations that are performed.

In this paper, Dixon and de Kleer give two representations for extensions on the Connection Machine and sketch the necessary algorithms. These al-

gorithms are far from complete. The first algorithm associates one processor with each consistent point in the assumption space. Node extensions are represented as a subset of the consistent point, by assigning an additional bit per processor for each node to record whether this point supports the node. Intersections and unions can then be computed by using single bit operations that can be performed in parallel. The extension of a node can be returned to the host machine by retrieving the truth value assignments from each active processor that has the appropriate bit set. When a new assumption is created, a forking operation is used to double the number of active processors. This operation along with the initialization of the assumption can be done in parallel.

The second representation for extensions delays the forking from a new assumptions as long as possible, on a per processor basis. This is done so that contradictions may be discovered elsewhere and reduce the number of activated processors. It is possible that the assumption will cause a contradiction that has been found already in a processor, eliminating the need to fork at all. To do this they allow each active processor to represent a subspace of the assumption space, by an assignment to each assumption of true, false, or both. The processor subspaces are disjoint, and together include all consistent points. Node extensions are represented as a union of these subspaces, again with one bit per processor. This makes computing intersections more complicated, as it is no longer a one-bit parallel operation.

One can see that these algorithms may need an exponential number of processors in the worst case. They remedy this problem by utilizing a chronological backtracking algorithm. This algorithm aids in cutting the number of processors by not allowing the earlier algorithm to create a processor for each point in the assumption space or fork.

They implemented a parallel ATMS which can broadcast justifications with forking inhibited, so that those processors that would deactivate or force an assumption's truth value do so, while those that would fork do nothing. It is only necessary to record that some processors were blocked from forking and that the justification should be rebroadcast at some later point. All justifications must be processed before computing a node's extension. This algorithm was run on the thirteen queens problem, on a thirteen by thirteen chess board. It ran 70 times faster on a 16K Connection Machine than the fastest sequential implementation on a Symbolics Lisp Machine. The actual parameters were 60 seconds vs. 4235 seconds to find 73,712 solutions.

2.3 Parallel LTMS

Vilain, in a recent paper [29], describes a parallel implementation of McAllester's RUP TMS (also termed Logic Truth Maintenance System, or LTMS), which is a single-context (as compared to a multiple-context ATMS), monotonic (as compared to a non-monotonic JTMS) truth maintenance system. His target hardware is the BBN Butterfly multiprocessor, which is an MIMD machine that supports up to 256 independent processors with fully shared memory.

In McAllester's model of a TMS, beliefs are ground propositions and the dependencies between beliefs are expressed as a database of logical clauses, each of which is a disjunct of literals. Propositions can take on one of three truth values, TRUE, FALSE, or UNKNOWN. Truths are either assigned to propositions by user assertions, or as a result of truth maintenance which is simple deduction, using Waltz-style [31] constraint propagation. A proposition is a node in a constraint graph and each clause is a constraint link. Constraints are propagated when the truths of the propositions in all but one literal in a clause become known. If the literals are all unsatisfied, the clause is then used to deduce the truth of the remaining literal.

McAllester notes that his TMS is essentially performing unit clause resolution, which is P complete. Problems in this class are strongly conjectured not to have a parallel solution that is guaranteed to terminate in less than polynomial time. So the worst-case performance of any parallel algorithm for McAllester's TMS can not be expected to improve on that of the best serial solution. What Vilain aims to do is to find parallel TMS strategies that exploit sources of concurrency that are independent of any inherent TMS sequentiality.

The parallel algorithm that Vilain presents is implemented in Butterfly LISP. The TMS constraint graph can be distributed across all the processors, providing a global data structure for the graph, to which each processor has independent access with the same average time. Within the constraint propagation algorithm, constraints attached to a node are checked for potential deductions when a truth value is determined for that node. The concurrency is introduced by checking the clauses in parallel. Synchronization is only a problem when two processes attempt to set the truth of a node simultaneously. This could lead to redundant propagation or undetected contradictions if the simultaneous attempts disagree in truth values for the given node. Thus, Vilain makes the testing for an unknown truth value of

a node an atomic operation, and as implemented on the Butterfly, only one competing process can set the truth value of the node. Vilain presents his algorithm in Butterfly LISP code and proves its correctness when the TMS database is restricted to Horn and NoGood clauses.

Vilain presents a worst-case example of the inherent sequentiality of McAllester's TMS. He notes that these pathological cases seldomly occur. The cases where they occur in circuit simulation were those examples with a moderate constraint fanout with a predominance of constraints that had a low probability of leading to an actual deduction when checked. He presents a way to transform the constraint graph to a new compiled graph, but notes that this still doesn't eliminate the inherent sequentially of the problem, but can increase the degree of run-time concurrency if the $M(n^2)$ processor requirement can be satisfied. The interested reader can refer to the paper for the exact translation algorithm of the original graph to the compiled one.

Then what are the ways in which concurrency can be increased even though this problem exists? Asserting multiple premises simultaneously will increase the degree of useful parallelism. This type of assertion occurs when multiple hypotheses are to be checked. Multiple retraction of assertions is another strategy to increase concurrency. But assertions and retractions cannot be performed in parallel. Also, since there are concurrent strategies for construction of the constraint graph, they can be coupled with the constraint propagation method to occur simultaneously.

When these strategies are utilized, it is necessary to change the TMS model to mix problem solving with truth maintenance, i.e. instead of the problem solver finishing before passing its results to the TMS and the TMS finishing before passing its results back.

Vilain proposes a way of handling the mixing with the use of noticers or demons that concurrently inform the problem solver of the changing truth values of beliefs. Issues such as database consistency must then be dealt with, which Vilain leaves as an open question. Again, this is a question that the reviewing author is also attempting to answer. His implementation does not explicitly address this issue, but deals with only those contradictions and inconsistencies which are detected.

As far as the speed-up in performance achieved by such an implementation, Vilain had problems using an uncompiled version of Butterfly LISP, and is in the process of reevaluating his parallel TMS on a new compiled version of LISP.

3 Parallel Production Systems

Production systems (or rule-based systems) are widely used in AI for modeling intelligent behavior and building expert systems. The problem with production systems is the same problem seen in the previously discussed AI techniques. For reasonably sized domains, production systems quickly become too computationally expensive and very slow. Thus, research in parallelizing production systems has emerged in order to speed up problem execution time.

To discuss this research, it is necessary to present some background on the OPS5 system [10] and general productions systems, though we will assume reasonable familiarity with the material. Because this work is very detailed, it will be necessary to gloss over several important aspects in order to get the full picture.

A production system has a database of rules or productions, called the production memory, and a database of working memory elements (WME's) or facts, called the working memory. Each rule consists of a conjunction of condition elements called the left hand side (LHS) and a set of actions called the right hand side (RHS). The LHS identifies when a rule can be executed by matching it against the working memory. The RHS specifies which WME's are to be added or deleted when the rule is executed.

The production-system interpreter is the underlying mechanism that determines the set of satisfied productions and controls the execution of the production-system program. The interpreter executes a production-system program by performing the following recognize-act cycle. (1) Match step: matches the LHS of all productions against the contents of working memory. (2) Conflict-Resolution step: one of the instantiated productions is chosen for execution. (3) Act step: the chosen production is executed.

3.1 Parallelizing OPS5

In Gupta's [15] research on parallelizing OPS5, he discusses two general types of search used in production systems. The first is *knowledge search*, finding information relevant to solving the problem. The second is *problem-space search*, search for the goal state. Problem space search is basically a combinatorial AND/OR search. But, if the problem space search is done sequentially, then if knowledge search is done intelligently, problem space search

can be pruned. But the more intelligent the knowledge search, the more knowledge is needed to guide the search, thus possible making the search combinatorially explosive. Knowledge search is performed during the match phase of the recognize-act cycle, and this is the step in which Gupta places most of his efforts of parallelism.

For this research, the Rete [11] match algorithm in OPS5 and Soar system is chosen to be parallelized. Soar [20] is a new production system formalism to provide expert systems with general reasoning power and the ability to learn. More detail on Soar will be presented in the next section. The Rete algorithm is very detailed, and we will refer the reader to a quite lengthy description in Gupta's book. For our purposes we will concentrate on the general ideas behind parallelizing the Rete match algorithm which extends to other match algorithms.

The data-flow like organization of the Rete network is the key feature that permits exploitation of parallelism at a relatively fine grain. It is possible to evaluate the activations of different nodes in the Rete network in parallel and to evaluate multiple activations of the same node in parallel and to process multiple changes to working memory in parallel. The parallel evaluation of node activations in the Rete network also corresponds to higher-level, more intuitive forms of parallelism in production systems. For example, evaluating different node activations in parallel corresponds to (1) performing match for different productions in parallel (also called production-level parallelism) and (2) performing match for different condition elements within the same production in parallel (also called condition-level parallelism). Rete is a state-saving model. That is, the algorithms store the results of the execution match from previous recognize-act cycles, so that only the changes made to the working memory by the most recent production firing need be processed every cycle. For this model the state corresponding to only fixed combinations of condition elements in the left-hand side is stored. This imposes some sequentiality on the evaluation of nodes, but the tradeoffs are worthwhile.

Gupta describes the characteristics of six OPS5 and Soar application production systems with which he performs essentially complete evaluations of his algorithms in order to find where bottlenecks were and how to improve the algorithm to its maximum efficiency. The systems have different numbers of productions and operate differently with the constraints of the algorithm, giving a good overall measure of the performance of the algorithm.

Overall parallelism can be used while performing each of the three steps

of the production system execution; match, conflict-resolution, and act. It is also possible to overlap the parallel processing performed in the match step with the conflict resolution step in the same cycle, and to overlap the act step of one cycle with the match step of the next cycle. All processors must synchronize after the match and conflict-resolution steps of a cycle before proceeding to the act step of that same cycle. But parallelism can be introduced within that act step.

Gupta describes (and implements a simulator) for the architecture of the production-system machine (PSM). The major characteristics of this architecture are: (1) Shared-memory multiprocessor with about 32-64 processors. (2) The individual processors should be high performance computers, each with a small amount of private memory and a cache. (3) The processors should be connected to the shared memory via one or more shared buses. (4) The multiprocessor should support a hardware task scheduler to help enqueue node activations that need to be evaluated on the task queue and to help assign pending node activations to idle processors.

The results of this parallelization were not what was expected, yet they were still satisfactory and encouraging. Initial expectations were of 100-fold to 1000-fold speed-up when parallelism was introduced. But the actual speed-up was about 10-fold.

3.2 Parallelism in a Learning Production System

In this section, Soar will be presented along with other results found when parallelism was introduced into the production system. Soar is a system developed to be capable of general intelligence. It is a system which uses a learning mechanism called *chunking* to create new productions which are summaries of results and adds these productions to the system. When the chunks eventually fire, the provide a learning-transfer mechanism. In this paper by Tambe, et al [28], the Soar/PSM-E implementation is presented. PSM-E is a C-based implementation of OPS5 on the Encore Multimax.

As with all production systems, execution is slow for large systems built in Soar, due to the matching procedure. Because chunking adds new productions, matching becomes even more of a bottleneck. Also, the additions of such chunks present various other computation problems that do not arise in non-learning production system. This paper concentrates only on the optimization of the matching procedure.

PSM-E is a highly optimized C-based parallel implementation of OPS5 on the Encore Multimax. This implementation consists of a control process that selects and fires an instantiation, and one or more match processes that actually perform the RETE match, as has been discussed in the previous section in the work by Gupta [15].

Soar consists of three modules: decide, chunking, and a production system. The production system in Soar is similar to that of OPS5. The modifications to OPS5 are that RETE must support the addition of productions at run-time, conjunctive negations. Also, all productions may be fired in parallel and only add working memory elements.

The decide module is responsible for the creation and deletion of all of the system's goals, as well as the selection of problem spaces, states and operators. The elaboration phase of this module matches the productions to determine the conflict set. Then all of the instantiations of the conflict set are fired in parallel. The phase cycles until no new instantiations are generated. The decision phase of this module attempts to make a decision about the problem space, state, or operation to be used to continue toward the goal. If a decision cannot be reached a subgoal is created on which it applies its problem solving capabilities. The chunking module is the learning mechanism. It creates new productions by generalizing and caching the results of problem-solving. The addition of chunks improves the Soar's performance when viewed in terms of subproblems required and the number of decisions within the subproblem.

Soar/PSM-E operates in a mode where Soar uses PSM-E as a matching engine. For this work, both Soar and PSM-E keep separate copies of working memory, since only the matching facility is being optimized. When productions are fired, both Soar and PSM-E update their working memories. Chunks, which are formed in Soar are passed to PSM-E as they are added in Soar.

The productions added due to chunking must be compiled into machine code like the rest of the system. Two mechanisms are used to make the compilation faster and provide sharing; a tree data structure for the RETE network and a jumptable to integrate the new code. As stated in the previous section, node sharing in RETE is necessary for speedup in parallelism.

Three tasks were run, one of which was the 8-puzzle. Without chunking, all speedups were fairly low due to contention for shared memory objects. Two shared memory objects in this system are the memory nodes of RETE

and the single shared task queue. Hashing was used to decrease contention of the memory nodes, but the main source of contention was the task queue. This contention was reduced by the use of multiple task queues, one for each process, but the speedups are still much less that ideal speedups.

Other problems with speedup are the presence of *long chains* which are long chains of dependent node activations caused by productions with large number of condition elements. Even given more processors, the system cannot get through a long chain any faster. They plan to introduce a constrained bilinear network organization to reduce the length of the long chains, but this was not done for this paper.

High speedup was obtained in the update phase when chunking was used. High degrees of parallelism can be achieved since the entire set of wmes is matched and the chunks are matched while updating them. In conclusion, the authors believe that such a learning production system is not bounded by the 10-20 fold empirical bound that applies to non-learning production systems.

3.3 Parallel Asynchronous Rule System

Schmolze [27] proposes a new strategy for executing production systems in parallel, called PARS (Parallel Asynchronous Rule System), which has been implemented on a hypercube machine and has promised high levels of speedup as shown by recent preliminary tests. His approach is somewhat like that of Ishida and Stolfo [17], in which many rules can be executed simultaneously, with each rule execution on a separate processor. The problem with Ishida and Stolfo's system is that all processors must complete the match step before beginning the conflict-resolution step, and then again before the act step. Thus, there is no overlapping of any of the steps in execution cycle, nor is there overlapping of one execution cycle with a following execution cycle. Schmolze's proposal intends a more asynchronous system and better performance that Ishida and Stolfo. A consequence of this is that his system departs from the overall semantics of an OPS5-like system, with the drawback being that OPS5 is almost an accepted standard in building production systems. Another major drawback in this system is the amount of programmer involvement necessary to parallelize the system.

In Schmolze's system, each of the processors is essentially a separate production system with local production and working memories. The program-

mer, along with possibly having to modify some of the initial rules, must specify an addressing scheme that determines which processors are to receive each possible WME (called the address of the WME) and which processors are to receive each rule (called the address of the rule). There is no restriction on the processors having disjoint rule addresses or WME addresses. The reader can see that this is quite a lot of work for the programmer to perform.

Each processor maintains a queue of actions and executes a similar basic loop to the OPS5 system, with the conflict-resolution step called the select step. The match step is performed by each processor, matching the LHS of its local rule set to its local working memory. The select step selects one of the matched rules for execution per processor. An additional step is added, called the "send step". This step sends the actions of the rule to the appropriate processors. The local act step then removes actions of the processor's queue, and updates its own working memory, in addition to possibly responding to commands sent from an overall controlling process. Since the choice of a rule to execute is a local decision, this basic loop can be executed asynchronously among all processors. Though this eliminates the problems with synchronization in previous systems, the addressing scheme for WME's and rules must be extremely effective.

Thus the programmer must discover the manner in which effective parallelism can be achieved and program the production system accordingly. The addressing scheme must be complete in the sense that no rule firings are missed. Because of the extra work necessary for the programmer, Schmolze proposes the development of an "assistant" which will perform some of addressing and aid the programmer. This assistant will have dynamic addressing available, and also distribute the rules across the processors using the algorithm given by Ishida and Stolfo. Programmers must also write a complete rule set, so that the same result is achieved no matter the order in which the rules are chosen or the type of conflict-resolution used, since this step is local to each processor. One final problem given to the programmer is to write a serializable rule set, in which case the final results produced by the parallel system can also be produced by the serial system. An algorithm to automate this activity is currently being developed by Schmolze.

4 Parallel Search

Heuristic search is a technique used to reduced the computational complexity of exhaustive search. Such strategies as alpha-beta pruning have been used to prune search trees for two player games. When nodes in such strategies are examined in parallel, it is likely that branches will be searched that would otherwise be pruned in sequential alpha-beta.

4.1 Advancements in Parallel Heuristic Search

Ferguson and Korf [9] put the different approaches to parallelizing search algorithms into three categories. One is to parallelize the processing of individual nodes, such as move generation and heuristic evaluation. Another approach, called parallel window search, was first developed by Baudet [1]. In this approach, different processors are assigned non-overlapping ranges for alpha and beta, with one processor having the true minimax value within its window, and finding it faster by virtue of starting with narrow bounds. The third, which they feel is the most promising approach, is tree decomposition, in which different processors are assigned different parts of the tree to search. The most recent experimental work on this has been done by Kumar, et al [25], to parallelize an algorithm called IDA* [18]. We will be discussing this algorithm in a later section. In IDA*, the total amount of work done is independent of the order in which the tree is searched.

The Ferguson and Korf have begun the development of a distributed tree search (DTS) strategy which can apply to those cases where alpha-beta search prunes and parallel search does not. Their algorithm is based on the idea that different processors will be assigned to search different parts of the tree. It is a generic algorithm that uses an arbitrary number of processors without shared memory or centralized control and is independent of the type of tree search to be performed.

There have been algorithms which perform parallel branch and bound searches with the best for alpha-beta pruning by Vornberger [30], as specifically applied to chess. But DTS is general and can apply to alpha-beta, but can also apply to other types of tree searches which gives this algorithm more generality than other more specific algorithms.

DTS works as follows. Given a tree with non-uniform branching factor and depth, the problem is to search it in parallel with an arbitrary number

of processors as fast as possible. DTS is begun by creating a root process (corresponding to the root of the tree to be searched) and assigning to this root node all available processors. Then expand the node and divide the processors among the children. The parent process then blocks awaiting the return of the processes the children have used. If some child processes are returned before others the parent node reassigns these available processes to "help out" the other children. Results of the search are also returned with the processes. These results may indicate success, failure, alpha, or beta values, etc., depending on the application. This process allocation results in efficient load balancing among processors which is a general concern when implementing parallel algorithms.

Once all child processes have completed, the parent node returns its results and processes to its parent. When only one processor is available to a node for searching, a depth-first search would take place. If DTS is given n processors where n is the number of tree nodes, search is done breadth-first.

When applied to alpha-beta search, the problem of communication-overhead arises, since an alpha or beta cutoff value for one branch of the tree may be determined at another distinct part of the tree. Also, during parallel search, the algorithm cannot take advantage of all the heuristic information the sequential search uses, so wasted work may be performed. This is termed search overhead.

Because of these problems, Ferguson and Korf developed a Branch-and-Bound processor allocation strategy for performing parallel alpha-beta search. They introduce a cutoff bound that corresponds to alpha and beta cutoffs so that the same pruning performed by serial alpha-beta is performed by parallel alpha-beta. If no cutoff bound exists at a node, then the processors are assigned depth first. A cutoff bound can then be established quickly. If a cutoff bound is initially passed to a node, or has been established by the first child of the node, then the processors are assigned in the usual breadth-first manner. The idea is to establish useful bounds before searching the children in parallel, in order to avoid evaluating extra nodes that would be pruned by the serial alpha-beta.

A proof that in the case of perfect node ordering, the Branch-and-Bound allocation strategy will evaluate the same nodes as serial alpha-beta is given in the paper. Also, a discussion of the algorithm as applied to the game Othello is presented. More evaluation of performance is needed before solid results can be given.

4.2 Parallel IDA*

Iterative-deepening A* is an important admissible algorithm for state-space search which has been shown to be optimal both in time and space for a wide variety of state-space search problems. In contrast, A* [22] requires exponential storage for most practical problems, and is less easily parallelizable that IDA* in terms of simplicity and overhead. Iterative deepening consists of repeated bounded depth-first search over the search space. In each iteration, IDA* performs a cost-bounded depth-first search by cutting off a branch when its total cost exceeds a given threshold. For the first iteration, the threshold is set to the cost of the initial state. Then for each successive iteration, the threshold used is the minimum of all node costs that exceeded the previous threshold in the preceding iteration. The algorithm continues until a goal is expanded. If the cost function is admissible, the IDA* is guaranteed to find an optimal solution. IDA* expands asymptotically the same number of nodes as A*, and its storage requirement is linear with respect to the depth of the solution.

Rao, et al, have developed a parallel version of IDA* that does not appear to be limited in the amount of parallelism. They tested its effectiveness by implementing their algorithm for the 15-puzzle on a Sequent Balance 21000 parallel processor and have been able to obtain almost linear speedup using up to 30 processors that are available on the machine.

Rao has parallelized IDA* by sharing the work done in each iteration among a number of processors. Each processor searches a disjoint part of the search space depth-first. When a processor has finished, it attempts to search a branch that another processor has yet to search. Once the space has been completely searched, termination is detected for the iteration, and a new threshold is computed. All processors quit when a solution is found.

Since the cost bounds for each iteration of PIDA* are identical to that of IDA*, the first solution found by any processor in PIDA* is an optimal solution, so computation ends when the first solution is found. It is possible for PIDA* to expand fewer nodes on the last iteration than that of IDA*, which leads to the observation of speedup greater than N using N processors (this is called acceleration anomaly).

Each processor maintains a local stack of the nodes to be searched. When it empties its stack it can take work off of the stack of another processor. Three basic procedures are executed by each processor. The first is to perform

a bounded depth-first search as long as work is available on the stack. When no work is available, try to get work from other processors. When no work can be obtained, attempt to detect termination. The objective of this work is to minimize communication overhead between processors, keep the work exchange infrequent, and detect termination quickly.

The actual results that were achieved showed superlinear speedup, i.e. greater than N for N processors. They then modified PIDA* to find all optimal solutions and compared it against IDA* finding all optimal solutions, producing speedup close to N for N processors.

4.3 Parallel Window Search

Powley [24] presents another view of the use of IDA*. This research is based on attempts to improve sequential IDA* and to use it in parallel search. His paper combines node ordering and parallel window search with IDA*. In A* and IDA*, for each node n, g(n) is the cost of reaching the node so far, h(n) is the estimated cost of a path from the node to a goal state. A* will expand the node with the minimum f(n) = g(n) + h(n). IDA* uses this to compute its thresholds.

The idea of node ordering in pure depth-first search is to expand the children of a node in an order that minimizes the time to find a goal. The ordering is performed for an iteration by using the heuristic information from the last iteration. Thus, the heuristics are used to order the nodes and after the ordering, pure depth-first search is used to find the next iteration of nodes. In IDA* with node ordering the nodes with the smallest h values or largest g values are expanded first, since the nodes with the smallest h values are likely to be closest to a goal, and the nodes with the largest g values require less search to find that a node does not lead to a goal. The reason is that every move increases g and the total cost g + h is limited by the threshold for the iteration, making the maximum depth that can be reached below a node n in that iteration be t - g(n). Thus the nodes with the largest values of g generate the shallowest subtrees below them. Ideally, what is needed is to generate the final iteration using an ordering of all the nodes from the next-to-last iteration, but this requires storing the complete next-to-last iteration, requiring exponential memory. A solution to this is to save only a fixed number of paths. But in IDA* good node ordering really only effects the time spent in the last iteration. It has no effect on the previous

iterations and there are serious disadvantages to only saving certain paths.

Parallel window search is the use of different processes to search to different thresholds or windows simultaneously, in the hope that one of them will find a solution. Window search originated in two-player game search [1] [19], but has not yet been applied to single-agent search, which is the focus of this research. The thresholds are those determined by IDA*. Though the first solution found may not be optimal, optimality can still be guaranteed by completing all shallower thresholds and comparing their solutions to that of the best goal found. Only limited speedup is achieved by this approach, because the search is dominated by the time to perform the last iteration, even if the others are performed in parallel. Since parallel window search speeds up the time to perform every iteration but the last and node ordering only speeds up the last iteration, an algorithm that used a combination of the two approaches was developed and tested.

This algorithm works as follows. N search processors are initially assigned the first N thresholds to search, with each process searching a different threshold. All processes use the ordering scheme previous discussed. Then when a process finishes searching to its current threshold, it re-orders its node set and broadcasts this ordering information to all other processes. Next, the process jumps over the other processes to the next threshold to be searched. The search terminates when the first goal is found.

The problem with the speedup results achieved by this algorithm is the comparisons the authors made. They compared their algorithm to the serial IDA* algorithm running the same problem instances. But their algorithm uses serial IDA* and embellishes it with node ordering and parallelizes it with window search. It seems that comparisons should only be made running the embellished algorithm on sequential machine and comparing that to the parallel implementation. Since the serial IDA* will have less overhead, speedup results will be lower. This is exactly what the authors experienced, a peak in the speedup and then a sharp decrease as the number of processes (and overhead) grew. Their best result, using the 15-puzzle problem, was on the most difficult problem instances, in which the elapsed real time was on average 1820 times faster that serial IDA* in terms of nodes generated.

5 Conclusion

We have presented a synopsis of papers which were reviewed as part of the beginning research for developing a parallel inference model. Current work is being done to improve Petrie's algorithm and to prove the correctness of the new algorithm [12].

References

[1] G. Baudet. The design and analysis of algorithms for asynchronous multiprocessors. Technical Report Ph.D. dissertation, Dept. of C.S., Carnegie Mellon University, April 1978.

[2] Johan deKleer. An assumption-based tms. *AI*, 28(2):127–162, mar 1986.

[3] Johan deKleer. Extending the atms. *AI*, 28(2):163–196, mar 1986.

[4] Johan deKleer. Problem solving with the atms. *AI*, 28(2):197–224, mar 1986.

[5] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), 1980.

[6] M. Dixon and J. de Kleer. Massively parallel assumption-based truth maintenance. In *Proceedings of AAAI*, 1988.

[7] Jon Doyle. A truth maintenance system. *AI*, 12(3):231–272, nov 1979.

[8] Jon Doyle and Philip London. A selected descriptor-indexed bibliography to the literature on belief revision. *SIGART Newsletter*, (71), apr 1980.

[9] C. Ferguson and R.E. Korf. Distributed tree search and its application to alpha-beta pruning. In *Proceedings of AAAI*, 1988.

[10] C.L. Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[11] C.L. Forgy. Rete: A fast algorithm for the man pattern/many object pattern match problem. *Aritificial Intelligence*, Sept. 1982.

[12] R. Fulcomer and W.E. Ball. Parallel algorithm for justification truth maintenance. Technical Report WUCS-88-26, Washington University, 1988. submitted for publication.

[13] R. Fulcomer, J.A. Fingerhut, and W.E. Ball. Bemas user's manual, 2nd edition. Technical Report WUCS-88-22, Washington University, 1988.

[14] James W. Goodwin. Watson: A dependency directed inference system. In *Proceedings of Non-Monotonic Reasoning Workshop*, pages 103–114. American Association for AI, oct 1984.

[15] A. Gupta. *Parallelism in Production Systems.* Pitman Publishing, London, 1987.

[16] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, Massachusetts, 1985.

[17] T. Ishida and S.J. Stolfo. Towards the parallel execution of rules in production system programs. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1985.

[18] R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[19] V. Kumar and L.N. Kanal. Parallel branch-and-bound formulations for and/or tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):768–778, Nov. 1984.

[20] J.E. Laird. Soar user's manual. Technical Report 4th Edition, Xerox PARC, 1986.

[21] D.A. McAllester. An outlook on truth maintenance. Technical Report AI Memo 551, MIT AI Lab.

[22] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Press, 1980.

[23] C. J. Petrie. A diffusing computation for truth maintenance. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages 691–695, 1986.

[24] C. Powley and R.E. Korf. Single-agent parallel window search with node ordering. In *AAAI Workshop on Parallel Algorithms*, 1988.

[25] V.N. Rao, V. Kumar, and K. Ramesh. A parallel inplementation of iterative-deepening a*. In *Proceedings of AAAI Conference*, pages 133–138, 1987.

[26] Michael Reinfrank. An introduction to non-monotonic reasoning. Technical Report MEMO-SEKI 85-02, jun 1985.

[27] J.G. Schmolze. An asynchronous parallel production system with distributed facts and rules. In *AAAI Workshop on Parallel Algorithms*, 1988.

[28] M. Tambe, D. Kalp, A. Gupt, C. Forgy, B. Milnes, and A. Newell. Soar/psm-e: Investigating match parallelism in a learning production system. In *ACM/SIGPLAN PPEALS*, pages 146–160, Sept. 1988.

[29] Marc Vilain. A parallel truth maintenance system. In *AAAI Workshop on Parallel Algorithms*, 1988.

[30] O. Vornberger. Parallel alpha-beta versus parallel sss*. In *Proceedings of the IFIP Conference on Distributed Processing*, Oct. 1987.

[31] D. Waltz. Understanding line drawings of scenes with shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–92. McGraw-Hill, New York, 1975.