Parallelism in AI Programs

*Dennis F. Kibler**
Irvine Computational Intelligence Project
Information and Computer Science Department
University of California at Irvine

John Conery
Department of Computer and Information Science
University of Oregon

ABSTRACT

A folk theorem is developing which suggests that parallel solution of AI programs will not afford a speedup of more than one order of magnitude. We critically review this folk theorem by analyzing some of the problems used to "prove" it, and then cite work that provide examples of better than one order of magnitude improvement for these problems. We examine two representative AI algorithms where parallelism would achieve speedups of two orders of magnitude with a reasonable number of processors.

I. Introduction

A few years ago the combination of rapidly decreasing prices for microprocessors along with their increasing computational power spawned the belief that massive amounts of parallelism would greatly enhance the power of computational models of intelligence. Early experiences with multiprocessor architectures dampened this belief. In particular Lenat found that up to four processors could be efficiently used to speedup the execution of his Eurisko program, but beyond that additional processors were not useful [I]. Fennell and Lesser's multiprocessor implementation of Hearsay II gave a speedup of 3.8 with 4 processors, 4.2 with 16 processors, and an estimated 14.7 with an infinite number of processors [2]. An empirical analysis of a forward chaining rule-based expert systems by Forgy suggested an upper limit of a speedup of 35 by parallel rule execution [3]. A generalization of these experiences has led to the emergence of a folk theorem which states that the amount of achievable or effective parallelism in AI programs will be limited to a factor of ten.

In this paper we show the flaws in arguments supporting the above-mentioned folk theorem. We do this by providing examples from the literature of parallel algorithms for rule-based systems and intelligent search. These algorithms are capable of leading to better than one order of magnitude speedup through the use of a "reasonable" number of processors. By "reasonable" we mean a speedup of a factor of *N* should be obtained from approximately *N* log N processors, not *N* * *N* processors.

The folk theorem addresses itself to problems that are dominated by global search, and our critique is based on parallel algorithms for just these sorts of tasks. For other types of AI programs, for example perceptual tasks such as scene analysis and speech recognition, effective parallelism on the order of four orders of magnitude can be achieved, and are essential (see Duff [4]).

Finally, we should point out that the folk theorem puts an upper bound on the asymptotic speedup of AI programs. Our response is in the same vein, showing where asymptotic increases in execution time *may* be obtained through different algorithms. We do not address the engineering problems of communication costs and load balancing that must be taken into account when implementing the algorithms on a multiprocessor.

II. Folk Theorem

The most expensive portion of most AI programs occurs during some type of search. Search is apparently parallel; a branch in the search space is a natural place to apply multiple processors. When tried, however, the speedup was not proportional to the number of added processors, but leveled off after a speedup of a factor of 4-6 times, regardless of the number of additional processors.

The argument for this lack of speedup follows: unlike a data base program which (perhaps) must follow all paths of a search, most AI programs involve heuristic search where Borne paths are not explored fully. The output of heuristic search is usually neither the optimal solution nor all solutions, but some "satisfactory" solution. In such cases, additional processors, after the first few, tend to increase the size of the search space explored, but fruitlessly. Resources are devoted to unimportant paths, increasing the total number of computations performed, but not decreasing, in proportion, the time to find a solution. The argument is that a clever sequential algorithm which selects the most promising paths will perform nearly as well (or better) than a naive parallel algorithm that explores all paths.

Although practical experience supports this argument, both the argument and the experience are seriously misleading. The argument makes implementation assumptions about the way in which parallelism ought to

* Partial support *for this research was provided by Hughes AI Research Center, Calabaaas, CA.*

be achieved, and these assumptions are mirrored in the actual implementation. Consequently the flawed argument is merely repeated in the implementation. A correct implementation of parallelism would execute the same operations as the sequential algorithm, only with some computations carried on simultaneously. For a correct analysis of the limits of the potential speedup due to parallelism we need to introduce some concepts.

Let a computation C be specified by a sequence of primitive operations. The nodes of the computation graph associated with C are the primitive operations. A directed arc is drawn between operation f1 and operation f2 if f1 enables f2, e.g. it produces some information needed by f 2. To make this analysis automatic it is useful to have a functional language. For logic programs the creation of this computation graph can be done using methods developed by Conery and Kibler [5,6],

We can now make a number of simple inferences about computation graphs. The computation graph is always acyclic. The depth of the graph gives the minimum completion time for the computation. The size of the graph, as measured by the number of nodes, gives the completion time for a uniprocessor (which effectively does a depth first traversal, never visiting a node until all of its ancestors have been visited). The maximum effective parallelism, i.e. the maximum number of processors that can be simultaneously working, is a function of the breadth of the graph. The maximum speedup is the ratio of the size to the depth.

The difficulty for achieving a large amount of parallelism in AI programs, without incurring exponentially growing costs, is that the particular computation graph is hidden within a much larger, broader computation graph of all potential operations. In other words, a "trace" of a sequential execution gives one possible graph. Consequently it is very difficult to distribute the computation amongst the various processors so that a speedup proportional to the number of processors is achieved, since additional processors are visiting nodes that were not even part of the graph in the sequential computation. Indeed very surprising effects of parallelism have been noted: using *k* processors, one may achieve a speedup of greater than *k, or* conversely a speedup of less than 1, i.e. a slow-down over uniprocessor speeds. Li and Wah analyzed necessary conditions for these anomalies [7].

In light of these unexpected results one can understand why the first approaches to achieving parallelism were inconclusive. Indeed, a simple parallel search (called OR parallelism) allows for no parallelism within deterministic subcomputations. Research on functional programming languages has shown substantial speedups can be achieved for such tasks as sorting, matrix multiplication, summing a vector, etc. Moreover these speedups can be realized without the programmer specifying when and where the parallelism should be instigated [5,6]. One of our points is that even though the time complexity of an AI program is dominated by search time, there may be substantial speedup from executing deterministic subcomputations in parallel. The execution of these computations in parallel is referred to as AND parallelism.

Relying on both AND and OR parallelism, we examine two representative AI algorithms and show that the amount of speedup is orders of magnitude beyond that suggested by the folk theorem.

III. Alpha-Beta

Consider the standard alpha-beta algorithm that is the basis of most chess programs. Note that this algorithm lies within the purview of the argument developed in the folk theorem, so one might be led to expect a maximum speedup of one order of magnitude. Finkel and Fishburn developed a special parallel algorithm for alpha-beta search which achieves a speedup of *k* with k processors [8]. Although this approach gives unbounded speedup, it takes 10,000 processors to yield a speedup of 100. We shall demonstrate a different approach to gain a factor of 100 which uses roughly 500 processors. For concreteness, let us suppose that the terminal evaluation function is the sum of 16 features and that the branching factor is exactly 32. The effective branching factor, due to alpha-beta search, is about 10.

Following a naive approach to achieve parallelism, we would take 32 processors and give each of them one branch of the tree to search. Since the effective branching factor is 10, most of the computations performed by this parallel search would not be done by a sequential processor and, in fact, are unimportant to the final result. The overall net gain would be a factor of 10, as predicted by the folk theorem. Carrying this process three levels into the tree, we see that 32,768 processors are used, with only 1000 searching useful paths; the effective speedup (as a percentage of the number of processors used) soon levels off.

Now let us achieve the parallelism in another way. We will not allocate processors until the next-to-last expansion in the tree, while growing the tree with the usual sequential algorithm, pruning as we go. When we reach the next to last expansion, we spread 32 processors out amongst the terminal nodes. For each terminal node we allocate an additional 16 processors, one processor for each feature in the evaluation function. We have now allocated a total of 512 processors to the task. The amount of speedup depends on the total number of useful computations that this allocation scheme provides. Again the fact that the effective branching factor is 10 implies that only 10*16 of the processors are doing useful work. In addition the initial tree generation is done sequentially, but once the search has depth more than 2, this has little effect on computation time, as can easily be computed. (We will

examine the effect of inherently sequential subcomputation in a later section.) Thus we have achieved a speedup of more than two orders of magnitude with a comparable number of processors.

Another parallel alpha-beta algorithm is the "key node" method of Lindstrom [9]. No asymptotic performance bounds are given, but simulation results (which take into account message passing and memory contention effects) show a speedup of better than 10 with only 20 processors.

IV. Rule Interpreter

Let us suppose we have a standard forward chaining rule-based interpreter with 2,000 rules. The predicates or conditions for these rules come out of a language with 100 predicates. We further suppose that rules have an average of 6-8 conditions and 2-4 actions. A standard rule-interpretation cycle is:

1. Find all rules that can fire
2. Choose "best" rule
3. Apply selected rule

Now one could achieve a speedup of nearly a factor of 150 over this simple rule interpreter by dedicating a processor to each rule, as step 1 is the time-dominating portion of the computation. Roughly, the argument is: Step 1 requires about 15,000 operations (2000 rules with 6-8 conditions); Step 2 requires about 90 operations (scoring each of about 30 rules and finding the "best" one); and Step 3 requires about 10 operations (doing 2-4 actions). The last two steps take 100/15,000% of the time, ergo the speedup of about 150. However, it is not necessary to dedicate one processor to each rule in order to achieve a substantial speedup. The Rete algorithm, developed by Forgy, uses changes to the working memory to determine possible changes to the conflict set [10]. His measurements indicate that each modification to the working memory enabled or disabled about 35 rules. With an average of 3 modifications to memory per rule execution, we should have to check the lefthand sides of roughly 100 rules, not all 2000. Furthermore, these figures (3 changes, 35 rules) appear to be independent of the number of rules in the system.

These observations have led some to believe that there is minimal effective parallelism in production systems. Since not all rules have to be checked on each cycle, adding processors to check rules will not help. Beyond the speedup in the execution of Rete (at most an order of magnitude) there is no more possible parallelism in production systems.

However, as Ofiazer points out, the analysis is based on production systems written in the OPS5 language, and his conclusions about minimal parallelism should be limited to systems written in that language [11]. Thus taking a set of rules designed for a sequential production system interpreter, and then looking for parallelism in that set, will lead one to erroneous conclusions about production systems in general. A similar line of thinking for numerical applications would lead one to conclude that since there is minimal parallelism in any given FORTRAN program, the problem solved by that program cannot be solved in parallel.

As an example of a rule interpreter that is not organized along the lines of the test-select-execute cycle, consider the "suspension interpreter" defined in STAMMER [12]. This interpreter associates a process with each partially instantiated rule. The conditions of each rule are regarded as a set of tasks to be completed. Moreover, associated with each condition is the list of processes (partial rule instantiations) that involve the condition. When new facts are added to working memory, all of the relevant rules (processes) can try to complete their execution. Those processes that complete their tasks belong to the conflict resolution set. (Processes are also returned to earlier points in their computation if elements of working memory are deleted, but this is a rarer occurrence). Depending on the number of processors available, one can distribute suspended processes to processors. The overhead for this activity can be very high, but a speedup for two-orders of magnitude is not out of the question.

The selection of the best rule from the conflict resolution set involves either finding the maximum or sorting. It is, perhaps, surprising that both finding the maximum or sorting n items can be done in $0(\lg(n))$ time by n/2 processors. This is the second most costly operation in the rule interpreter and, with parallelism, can be effectively eliminated. The overall increase in efficiency is still dominated by the two-order of magnitude speedup in computing the conflict resolution set.

Another approach towards achieving parallelism for rule interpreters is that suggested by Stolfo and Shaw, which uses the tree-machine architecture of the DADO machine [13,14]. The Rete algorithm was invented to efficiently execute production systems on a standard von Neumann machine. Two of the limitations assumed by the Rete algorithm, that working memory not contain variables and that working memory change slowly, are avoided by the DADO architecture. The achievable speedup of the architecture proposed by Slolfo and Shaw is not yet known [14]. More recently Shaw has proposed the NON-VON architecture [15], which mixes simple and complex processors. Simulations of NON-VON on productions systems have yielded speedups of a factor of 100, based on the same examples as examined by Forgy and Gupta.

Lastly, we should mention that sequential assumptions and constraints have already been built into our abstract formulation of a rule-interpreter. If we have a pure inference system where there is no need to retract deductions, then why should we bother to decide which inference is the most fruitful? If a rule (process) becomes fireable, let

it fire. There is no need to segment execution into finding fireable rules, selecting a rule, and then executing it. With a pure inference system rule instantiation can be tested and fired in parallel.

The point of these discussions is that there are many ways to implement parallelism and the effectiveness of parallel processing for AI has not yet been determined. In the next section we examine the speedup limitation due to inherently sequential subcomputations.

V. Amdahl's Law

A deep concern for unexcelled computational speed led Amdahl to examine the potential speedups arising from parallel computation. Amdahl supposed that he had *P* processors to allocate to a computation which contained an inherently sequential subcomputation. He supposed that the sequential portion of the computation required a fraction / of the total computation. Under these assumptions, the upper limit on the amount of speedup *S* possible is given by the formula:

$$S \leq \frac{1}{f + (1 - f)/P}$$

In particular we see that even with an unlimited number of processors this computation cannot be spedup by more than a factor of 1 / /. For example, if *1%* of a computation is inherently sequential, then the maximum speedup is a factor of 100. Thus, even if only a marginal portion of the computation is sequential, the maximum achievable speedup is severely limited.

We must be careful in applying Amdahl's law, as demonstrated by the preceding section. It is not enough to show that a particular algorithm has an inherently sequential subcomputation. We must ensure that the algorithm has not already been conceptualized in a sequential manner.

VI. Conclusions

This papers argues that the apparently small gains due to parallelism are an attribute of a naive approach towards parallelism. Several examples were developed which demonstrate that both the conclusion and the argument for the folk theorem are wrong. A means for estimating the effects of sequentiality on parallelism was given. An accurate method for evaluating the speedup and the number of processors required, based on the concept of a computation tree, was developed. To achieve this parallelism is not simple, but holds promise for AI computations. Let us avoid AI's history of jumping to conclusions and turning first impressions into strongly held beliefs/prejudices. We are still gathering evidence on the effectiveness of parallel processing for AI.

Acknowledgements

Some of the arguments presented here were fine-tuned in conversations with Gary Lindstrom, Chip Maguire and David Shaw.

References

[I] Lenat, D.B., "Computer Software for Intelligent Systems'', *Scientific American,* Sept. 1984, 204-213.

[2] Fennell, R.D. and V.R. Lesser, "Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay II", *IEEE Trans. Computers,* Vol. C-26, No. 2, Feb. 1977,98-111.

[3] Forgy, C, A. Gupta, A. Newell, and R. Wedig, "Initial Assessment of Architectures for Production Systems", AAAI-84, 116-120.

[4] Duff, M.J.B, "Review of the CLIP image processing system", *Proc. National Computer Conference,* 1978, 1055-1060.

[5] Conery, J.S. and D.F. Kibler, "Parallel Interpretation of Logic Programs", *Functional Languages and Computer Architecture Proceedings,* March 1981, 163-174.

[6] Conery, J.S. and D.F. Kibler, "AND Parallelism in Logic Programs", *IJCAI-83,* 539-543.

[7) Li, G. and B.W. Wah, "How to Cope with Anomalies in Parallel Approximate Branch-and-Bound Algorithms", *AAAI-84,* 212-215.

(8) Finkel, R.A. and J.P. Fishburn, "Parallelism in Alpha-Beta Search", *Artificial Intelligence Journal,* volume 19, number 1, September, 1982, 89-106.

[9] Lindstrom, G. "The Key Node Method: A Highly Parallel Alpha-Beta Algorithm", Technical Report UUCS 83-101, University of Utah.

[10] Forgy, C.L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", *Artificial Intelligence Journal,* volume 19, number 1, September, 1982, 17-37.

[11] Oflazer, K. "Partitioning in Parallel Processing of Production Systems", 1984.

[12] Bechtel, R., D. Kibler, and P. Morris, "Incremental Deduction in a Real-Time Environment", *CSCSI/III,* May 1980, 26-33.

[13] Stolfo, S.J. and D.E. Shaw, "DADO: A Tree-Structured Machine Architecture for Production Systems", *AAAI-82,* 242-246.

[14] Stolfo, S.J., "Five Parallel Algorithms for Production System Execution on the Dado Machine", *AAAI-84,* 300-307.

[15] Shaw, David, "NON-VON Architecture", colloquium at University of California, Irvine, April, 1985.