CONSIDERATIONS CONCERNING PARALLEL AND DISTRIBUTED ARCHITECTURE FOR INTELLIGENT SYSTEMS

[1] Delia Ungureanu, [2] Dominic Mircea Kristaly, [3] Adrian Virgil Craciun

[1, 2]Automatics Department, "Transilvania" University of Brasov, M.Viteazu Street, no.5, 500174, Brasov, Romania, phone/fax: +40 0268 418836, delia@deltanet.ro, kdominic@vision-systems.ro

[3]Department of Electronics and Computers, "Transilvania" University of Brasov, Bd. Eroilor nr.29, 500036 Braşov - Romania, phone/fax +40 268 474718, craciun@vega.unitbv.ro

An analysis of Artificial Intelligence (AI) programs reveals that there exists a scope of massive parallelism in various phases of reasoning and search. Many modules of the reasoning programs can be realized on a parallel and distributed architecture. Generally, the parallelism in a reasoning program can appear at one of the following three levels: the knowledge representation level, the compilation and control level and the execution level.

The paper provides a brief introduction to the architecture of intelligent machines with special reference to representation and execution level parallelism in heuristic search, production systems and logic programming. We briefly discuss about some features of AI machines: symbolic processing, nondeterministic computation, dynamic execution, massive scope of parallel and distributed computation, knowledge management, open architecture.

The discussions about parallelism at knowledge representational level comprise discussions about parallelism in production systems and parallelism in logic programs including four different kinds of parallelisms: AND parallelism, OR parallelism, stream parallelism and unification parallelism. Different types of parallelisms may co-exist in a logic program. A schematic logic architecture of the system will be developed from the algorithm mentioned in the paper. This way, we perform an analysis of the time estimated to be covered in order to compare the relative performance of the architecture of the system used.

Keywords: Artificial Intelligence, knowledge representation, algorithm.

1. INTRODUCTION

The issues of designing efficient AI machines can be broadly classified into the following three levels:

1. Representational level;
2. Control and Compilation level;
3. Execution / Processor level.

An AI machine, in general, should possess the following characteristics:

a) *Symbolic processing*: An AI machine should have the potential capability of handling symbols in the phase of acquisition of knowledge, pattern matching and execution of relational operation on symbols.

b) *Nondeterministic computation*: In an AI problem, the occurrence of a state at a given time is unpredictable. Such systems are usually called nondeterministic, and require special architecture for efficient and controlled search in an unknown space.

c) *Dynamic execution*: Because of nondeterministic nature of the AI computation, the size of data structures cannot be predicted before solving the problems. So it is necessary to use dynamic allocation of memory. Deadlocked tasks should also be dynamically allocated to different processors and communication topology should also be dynamically altered.

d) *Massive scope of parallel and distributed computation*: In parallel processing of deterministic algorithms, a set of necessary and independent tasks are identified and processed concurrently. This class of parallelism is called AND- parallelism. The large degree of nondeterminism in AI programs offers an additional source of parallel processing. Tasks at a nondeterministic decision point can be processed in parallel. The later class is called OR-parallelism. Besides parallelism, many of the AI problems that include search and reasoning can be realized on a distributed architecture. System reliability can be improved to a high extent by such a distributed realization of the AI tools and models. The throughput and the reliability of the system thus can be enhanced jointly by fragmenting the system on a parallel and distributed architecture.

e) *Knowledge management*: Knowledge is an important component in reducing the complexity of a given problem. The richer is the knowledge base, the lesser is the complexity in problem solving.

f) *Open architecture*: AI machines should be designed in a way, so that it can be readily expanded to support modification or extension of algorithms for the given problems.

2. PARALLELISM IN HEURISTIC SEARCH

We propose the A* algorithm and the IDA* algorithm for heuristic search on OR graphs. The A* algorithm selects nodes for expansion based on the measure of f = g+ h, where g and h denote the cost of generating a node (state) n and the predicted cost of reaching the goal from n respectively. The IDA* algorithm, on the other hand, selects a node n for expansion as long as the cost f at node n is within a pre-defined threshold. When no solution is found within the pre-defined threshold, it is enhanced to explore further search on the search space.

Because of non-determinism in the search process, there exists ample scope to divide the search task into possibly independent search spaces and each search sub-task may be allocated to one processor. Each processor could have its own local memory and a shared network for communication of messages with other processors. Usually there exist two common types of machines for intelligent search. These are *i) Single Instruction Multiple Data (SIMD) and ii) Multiple Instruction Multiple Data (MIMD) machines*. In a SIMD machine, a host processor (or control unit) generates a single instruction at a definite interval of time and the

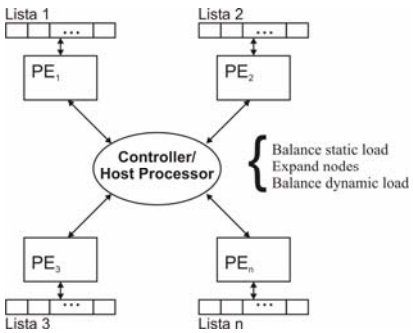

Fig. 1 A SIMD architecture for IDA*

processing elements work synchronously to execute that instruction. In MIMD machines, the processors, instructed by different controllers, work asynchronously.

We consider a SIMD architecture (fig.1) with n processors: P_1, P_2, …., P_n , each having a list for data storage. A host processor (controller) issues three basic types of commands to other processors (also called processing elements). These are:

i) Balance the static load (on processors),

ii) Expand nodes following guidelines and

iii) Balance the dynamic loads (on processors).

The static load balancing is required to provide each processor at least with one node. This is done by first expanding the search tree and then allocating the generated nodes to the processors, so that each get at least one node. Each processor can expand the sub-tree rooted at one of the supplied nodes. The expansion of the sub-trees is thus continued in parallel. The expansion process by the processors can be done by either of two ways: **i) Partial Expansion (PE)** and **ii) Full Expansion (FE)**. The algorithms for Partial Expansion with the corresponding traces are presented in fig. 2 and fig 3. During the phase of expansion, some processors will find many generated nodes, while some may have limited scope of expansion. Under this circumstance, the dynamic load balancing is required. The host processor identifies the individual processors as needy, wealthy and content based on their possession of the number of nodes in their respective lists. A wealthy processor, that has many nodes, can donate nodes to a needy processor, which has no or fewer nodes. The transfer of nodes from the lists of wealthy processors is generally done from the rear end. The readers may note the importance of DELETION operation (of Queue) at this step. A content processor has a moderate number of nodes and thus generally does not participate in the process of transfer of nodes.

Procedure Partial-Expansion
Begin
While the list is not empty **do**
Begin
Delete the front element n from the list *L*;
Generate a new child *c* of *n*;
If *n* has yet an ungenerated child
Then place *n* at the front of *L*;
If f(c) ≤ threshold
Then
If *c* is the goal
Then return with solution;
Else enter *c* the front of *L*;
End While;
End.

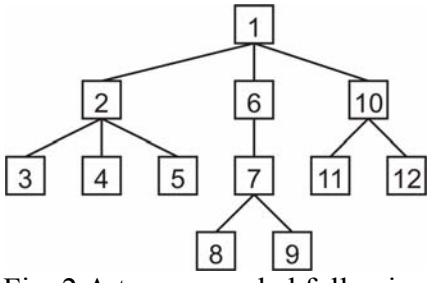

Fig. 2 A tree expanded following the ascending order of nodes using the PE algorithm

Step:						
	1	1(2, 6, 10)		7	7(8, 9)	1(10)
	2	2(3, 4, 5)	1(6, 10)	8	7(9)	1(10)
	3	2(4, 5)	1(6, 10)	9	1(10)	
	4		1(6, 10)	10	10(11, 12)	
	5	1(6, 10)		11	10(12)	
	6	6(7)				

Fig. 3 Trace of the procedure PE on the tree of fig. 2 where the elements of the list $p(c_1, c_2, \ldots c_n)$ represents a node p with its ungenerated children $c_1, c_2, \ldots c_n$.

3. PARALLELISM AT KNOWLEDGE REPRESENTATIONAL LEVEL

Distributed representation of knowledge is preferred for enhancing parallelism in the system. A Petri net, for example, is one of the structural models, where each of the antecedent and the consequent clauses are represented by places and the if-then relationship between the antecedent and consequent clauses are represented by transitions. With such representation, a clause denoted by a place may be shared by a number of rules. Distribution of fragments of a knowledge on physical units (here places and transitions) enhances the degree of fault tolerance of the system. Besides Petri nets, other connectionist approaches for knowledge representation and reasoning include neural nets, frames, semantic nets and many others.

3.1 Parallelism in Production Systems

A production system consists of a set of rules, one or more working memory and an inference engine to manipulate and control the firing sequence of the rules. The efficiency of a production system can be improved by firing a number of rules concurrently. However, two rules where the antecedents of the second rule and the consequents of the first rule have common entries are in pipeline and therefore should not be fired in parallel. A common question, which may be raised: is how to select the concurrently firable rules. A simple and intuitive scheme is to allow those rules in parallel, which under sequential control of firing yield the same inferences.

For efficient execution of a rule-based system, the elements in a set of compatible rules should be mapped onto different processing elements. If the mapping of the compatible rules onto different processing elements is not implemented, the resulting realization may cause a potential loss in parallelism. When two rules are input dependent or input-output dependent, they must be mapped to processing elements, which are geographically close to each other, thereby requiring less communication time.

3.2 Parallelism in Logic Programs

A Logic program, because of its inherent representational and reasoning

formalisms, includes four different kinds of parallelisms. These are AND parallelism, OR parallelism, Stream parallelism and Unification parallelism.

- **AND-Parallelism -** Consider a logic program, where the body of one clause consists of a number of Predicates, also called AND clauses, which may be unified with the head of other clauses during resolution. Generally, the resolution of the AND clauses is carried out sequentially. However, with sufficient computing resources, these resolutions can be executed concurrently. Such parallelism is usually referred to as AND parallelism. It is the parallel traversal of AND sub-trees in the execution tree.
- **OR-Parallelism -** In a sequential PROLOG program, each literal in the body of a clause is unified in order with the head of other clauses during the resolution steps.
- **Stream Parallelism -** Stream parallelism occurs in PROLOG, when the literals pass a stream of variable bindings to other literals, each of which is operated on concurrently. Literals producing the variable bindings are called producers, while the literals that use these bound values of variables are called consumers.
- **Unification Parallelism -** In a sequential PROLOG program, if a predicate in the body of a clause contains a number of arguments, then during unification of that predicate with the head of another clause, each argument is matched one by one. However, with adequate resources, it is possible to match the multiple arguments of the predicate concurrently with the corresponding positioned terms in a head clause. The parallelism of matching of variables of a predicate with appropriate arguments of other predicates is generally referred to as Unification Parallelism.

3.3 Parallel Architecture for Logic Programming

The different types of parallelisms may co-exist in a logic program. Petri nets can be employed to represent these parallelisms by a structural approach.

For an AND-parallelism example, we consider the following three clauses, where clause 1 can be resolved with clause 2 and 3 concurrently. From the previous section, we call such parallel resolution of AND clauses A(X) and B(X) the AND–parallelism. Such type of parallelism can also be represented and realized by Petri nets as shown in fig. 4.

$F(x) \leftarrow A(x) , B(X)$ (1)

$A(1) \leftarrow$ (2)

$B(1) \leftarrow$ (3)

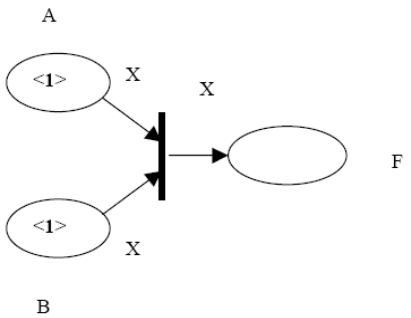

Fig. 4 A Petri net representing AND-Parallelism

To representing OR-Parallelism, we consider the following set of rules, where the predicate A(X) in clause 1 can be unified with the predicates A(1) and A(2) in clause 2 and 3. Since the resolution of the two OR-clauses (2) and (3) are done with clause 1 concurrently, we may refer to it as OR-parallelism. OR-parallelism can be realized

easily with Petri net models. For instance, the above program when represented with a Petri net takes the form of fig. 5.

F(x) ← A(X) , B(Y) (1)
A(1) ← (2)
A(2) ← (3)
B(1) ← (4)

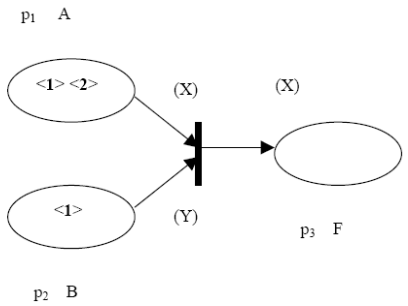

Fig. 5 A Petri net representing OR-Parallelism

The argument of predicates in clause (2) and (3) are placed together in place p_1, corresponding to the predicate A. Similarly the argument of the predicate in clause (4) is mapped at place p_2. The pre-conditions for the resolution process of clauses (2), (4) and (1) as well as clauses (3), (4) and (1) can be checked on the Petri net model concurrently. One important issue that needs mention here is the requirement of extra resources for maintaining this concurrency of the resolution process.

Stream-parallelism is often referred to as a special form of OR-parallelism. This parallelism, as already stated, has similarity with the pipelining concept. In pipelining, processes that depend on the data or instructions produced by other process are active concurrently. Typical logic programs have inherent pipelining in the process of their execution.

In unification parallelism, as already stated, the terms in the argument of a predicate are instantiated in parallel with the corresponding terms of another predicate.

4. CONCLUSIONS

The paper presents a SIMD architecture for parallel heuristic search and then explore the scope of parallelism in production systems and logic programming.

The architectural features of AI machines depend largely on the tools and techniques used in AI and on their applications in real world systems. The issues of designing efficient AI machines can be broadly classified into the following three levels: a. representational level, b. control and compilation level, c. execution/ processor level.

In those sections we try to elucidate various functional forms of parallelism in heuristic search and reasoning and illustrate the scope of their realization on physical computing resources. The distributed representation of knowledge is preferred for enhancing parallelism in the system.

5. REFERENCES

[1] Mahanti A., J.C. Daniels, *A SIMD approach to parallel heuristic search*, *Artificial Intelligence*, vol. 60, pp. 243-282, 1993.

[2] Konar A., *Artificial Intelligence and Soft Computing, Behavioral and Cognitive Modeling of the Human Brain*, CRC Press LLC, 2000, New York.