Loredana MOCEAN, PhD
Monica CIACA, PhD
Business Information Systems Department
"Babes – Bolyai" University, Cluj – Napoca
Halim Khelalfa, PhD
University of Wollongong , Dubai

A FORMAL MODEL FOR IMPLEMENTATION OF OR PARALLELISM

Abstract. *OR parallelism can be exploited when a relationship is defined by more than a clause and a calling subgoal may be unified by more than a procedure header. In such a case, the bodies of the clauses involved may be executed in parallel, giving raise to an OR parallel execution. So OR parallelism becomes an efficient method for exploiting alternative solutions in parallel.*
*In section **1** we concentrated the main problems and limitations which appear in OR parallelism implementation, presenting also the most important result (due to Gupta and Jayaraman) obtained until now with respect to this aspect: the impossibility of simultaneously fulfilling the three criteria which define the implementation of an ideal OR parallel system.*
*In section **2** we presented the main memory management mechanisms involved in a classical sequential implementation of the Prolog language.*
*The analysis and characterization of the OR parallel execution models from section **3** are mainly performed relatively to the types of binding environments (centralized or distributed). We present and analyze also models based on multi-agents systems and methods based on stack operations.*
*Section **4** represents the main original contribution of this paper, building a formal model for OR parallelism implementation and making an analysis of its complexity. The results obtained here formally validate the limitation reported by Gupta and Jayaraman, being also of a significant practical utility regarding possible improvements for OR parallel implementations.*
*In section **5** we present a classification based on the three criteria, characterizing a set of implementations proposed in the literature or being in use at this time.*
Keywords. *Models, Implementation, OR, Parallelism, Lemma, Theorem.*

JEL Classification:

1. Introduction

Problems related with OR parallelism implementation.

A major problem in exploiting OR parallelism is that it does not expose a *constant time complexity.* It depends on variable access time, node creation time and on the time needed by a processor for starting the execution of a new branch. For building

an ideal OR parallel system, it is necessary to accomplish all of the following three criteria [13]:

- ◊ Constant time *environment creation*;
- ◊ Constant time *variable access and binding*;
- ◊ Constant time *context switch*.

Gupta and Jayaraman [13] showed that these three criteria cannot be simultaneously fulfilled. In other words, non constant time costs can not be avoided when managing binding environments. However, they can be reduced by designing an efficient scheduler.

2. Memory management in sequential Prolog.

Memory management in the case of OR parallelism is approached here comparatively with the classical methodology from sequential languages. For this reason we analyzed first the memory management solutions for economic processes, provided by the sequential Prolog language.

The (sequential) Prolog interpreter uses the following stacks for representing the current state of the resolution of economic process (see figure 1):

1. **the Environment stack** for managing the current state of the resolution of economic process;
2. **the Variables stack** for keeping the variables bindings;
3. **the Trail stack** which allows backtracking by managing variables unbinding.

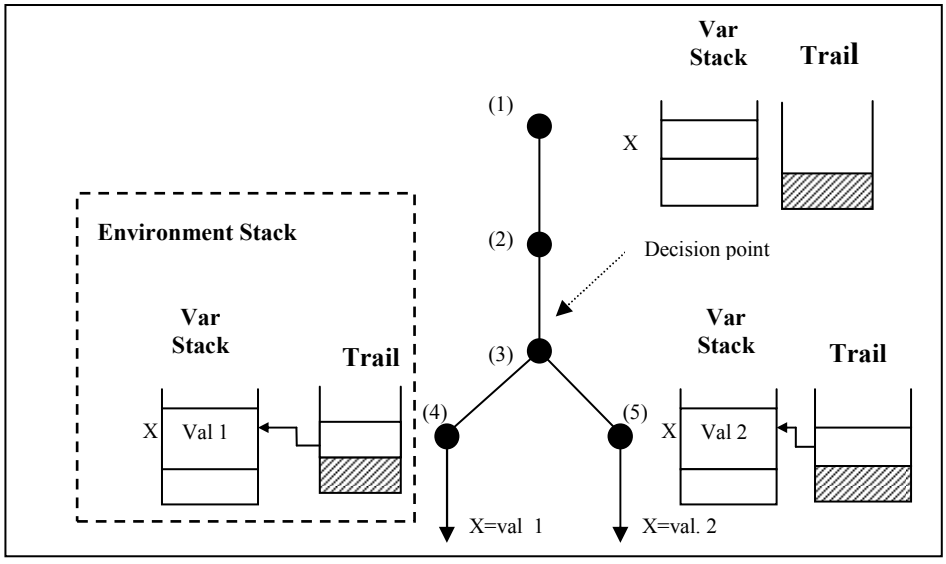


Figure 1. The management of stacks in economic processes, provided by sequential Prolog

3. OR parallel models.

3.1. OR parallel models based on centralized binding environments.

The basic idea in such an approach is to build a virtual stack for every process which will share as much information as possible with the other stacks and to duplicate the values only for the variables which uniquely binds in every process. The binding schemes for this type of models obey the following rules:

- variables bindings are kept locally in individual clauses;
- goal unification with the header of a clause often requires access to previously bound variables;
- unification between two free variables is accomplished by binding one variable to the reference of the other.

In the next paragraph we present some important models.

*a.) The **directory tree** model.*

This scheme was developed by Ciepielewski and Haridi [6]. In this model every branch of the OR tree has an associated process. The process binding environment is composed by *contexts*. At every clause invocation a new context is created. Every process has a separate binding environment, but some of the contexts may be used also by some other branches (see figure 2).

For efficiently accessing its environment, a process uses *directories*. A ***directory** of a process is a vector of contexts references. The environment of a process consists though from the contexts to which points its directory*. The *n*-th location from the directory contains a pointer to the *n*-th context of the process.

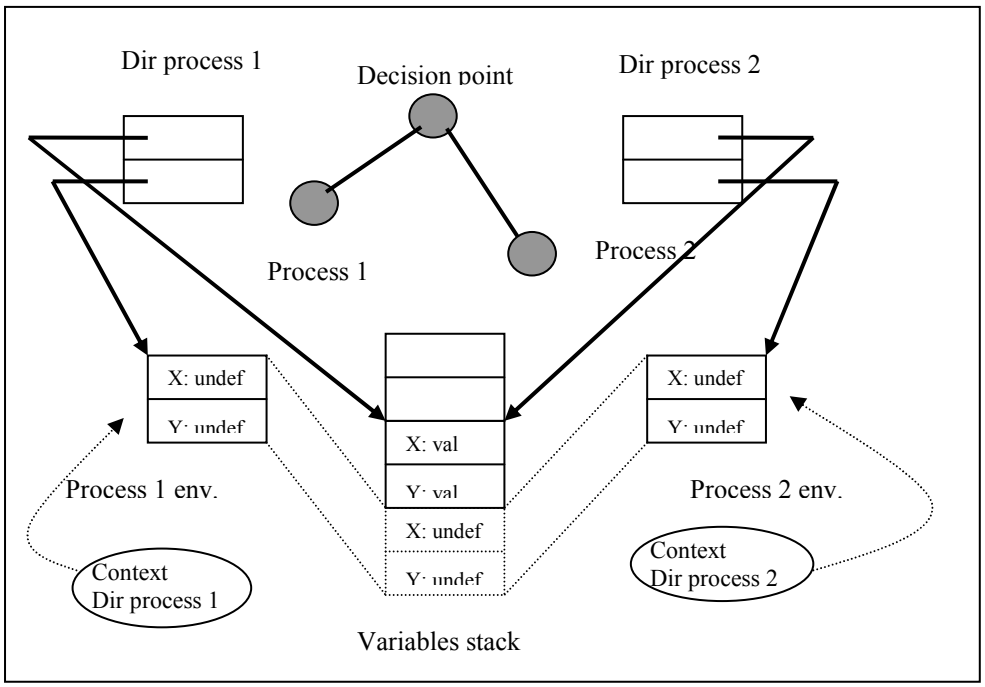


Figure 2. The directory tree model

Data structures involved in this model are shown in figure 3.

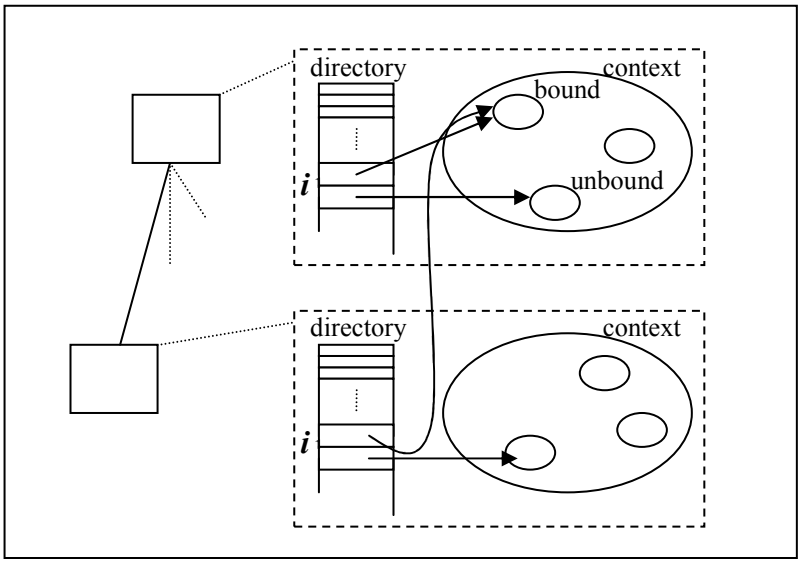


Figure 3. Data structures involved in the directory tree model

b). The ***hashing windows*** model.
This scheme was proposed by Borgwardt [5]. In this model separate binding environments are managed by means of certain *hash windows*. Every node from the OR tree has its own hash window where its conditional bindings are stored. A hash function is applied for every variable address for determining the address of the beginning of the list (cathegory) in which that conditional binding will be stored. Unconditional bindings will not be stored in hash windows, but directly into the tree nodes.

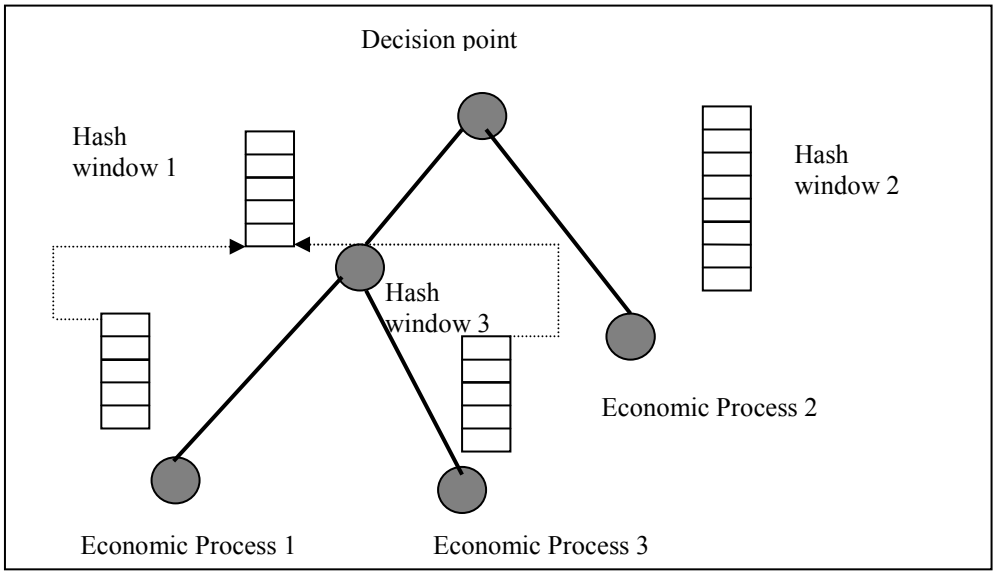


Figure 4. **Hashing windows technique**

c). The ***time stamping*** model.
This temporal scheme was developed by Tinker and Lindstrom [24]. It uses time stamps to identify the correct bindings for an environment.

3.2. OR parallel models based on distributed environments.

In OR parallel models based on distributed environments, the number of the binding environments visible for a process is limited to one or two, which makes dereferencing operations much simpler than in the case of centralized environments. However, the independence of the binding environments is obtained with the supplementary cost of some binding and copying operations. Distributed models differ from one another in the way the independence of the binding environments is achieved.
The existing models from this category are:
a). the **EPILOG** model
b). the ***data-driven*** **OR** parallel system [15]
c). the ***variable importation*** scheme [20]
d). the ***closed environments*** scheme
e). the **DIALOG** model
f). the ***forward stack*** model and the ***binding arrays*** model.

3.3. Models based on multi-agent systems.

The previous paragraphs describe OR model based on centralized and distributed

binding environment. Some models represents a compromise between two of them, they divide the notion of execution stage in a local stage and global stage, which contain details about task scheduling and blobal binding environments, information shared by all the processors.
Two important models of this category are:
a.) The **SRI** model.
B). The **MUSE** (***MUltiple SEquential Prolog engines***) model.

4. Methods based on stacks operations.

This paragraph presents techinuque of using of the stacks in PR model paralellism.
a). The ***copying of stacks*** method.
This scheme assumes that every agent has its own instances (copies) of the stacks and it does not access the stacks of the other agents. Sharing of activities is obtained by copying the corresponding parts of the stack from one agent's memory location to another (see figure 5).
b). ***Sharing of stacks*.**
c). ***Recomputing of stacks*.**

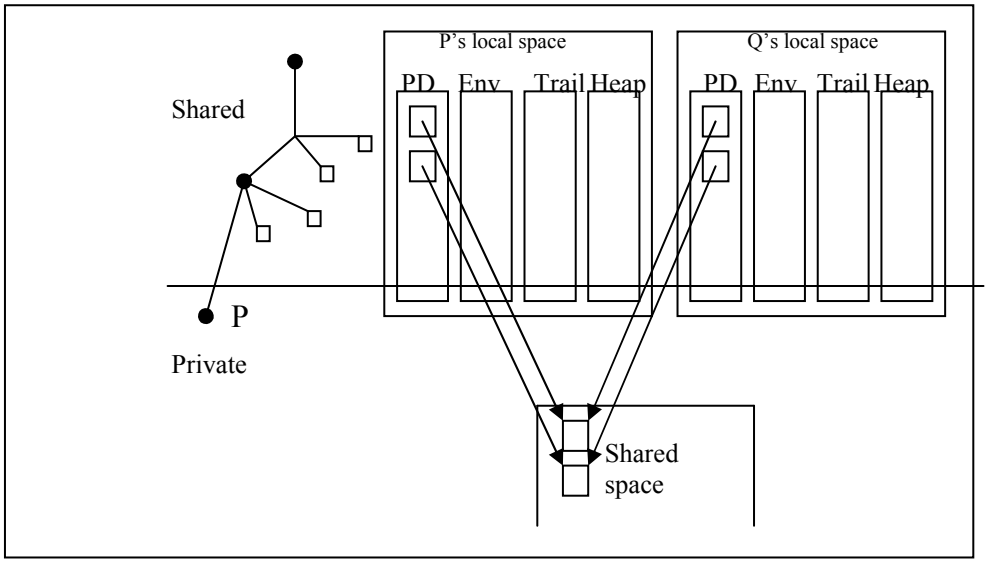


Figure 5. The result of the stack copying operation

5. A formal model for the implementation of OR parallelism.
The only ones who tried until now to develop a formal model for OR parallelism were Gupta and Jayaraman [12].The authors do not approach also a complexity analysis for the defined operations, even if in our opinion their model is adequate for such a analysis.

Starting from the model and the results mentioned above, in section 5.1. ***we develop our own formal model for the OR parallelism implementation***. Inspired by the Gupta-Jayaraman model, our model is substantially modified for keeping only the necessary elements which allow a complexity analysis.
Using this formal model, we obtained an important result presented in our theorem and which extends the results of [12]. The obtained complexity limit of $\Omega(\log N)$ formally proves the impossibility of obtaining an OR parallel implementation with constant time per operation.
The practical importance of this result is that independently of the operational semantics of the logic programming language and independently of the optimizations tried or the chosen implementation scheme, we can not avoid the limitations imposed by the obtained complexity value. By means of our result, we can give to the parallel logic system developers clear directions regarding the possible improvements of the performances of the implemented systems.

5.1. The formal model. Notations and terminology.

Definition. A ***nondeterministic program*** is a set of procedural definitions, each of the form ***header; body***, where *header* has the form ***id(pars)***, the same ***id*** procedure being able to appear in the header of many different procedural definitions. The syntactic category ***pars*** represents a (possible empty) list of some formal parameters.

- ***N*** denotes the set of the nodes in the OR search tree;
- ***V*** denotes the set of all variables;
- ***T*** denotes the set (domains) of terms or values;
- ***P*** denotes the set of processors;
- ***M*** denotes the set of memory locations in the multiprocessor system;
- $\mathcal{P}(S)$ is the powerset of S and $|S|$ is the cardinal of S.

Definition. An ***OR search tree*** for a given nondeterministic program and for a given query is a tree, every node having an associated continuation and a ***local environment***, such that:

1) for the root node the query is its associated continuation and the set of variables appearing in the query form its local environment;
2) every node $n \neq$ root is created after choosing a different procedural definition for executing the first call from the body of n's parent node and:
   a) the continuation of node n consists from the statements which compose the body of the chosen procedure definition followed by the statements after the first call from n's parent (the statements are assumed to be interpreted in the framework corresponding to n).
   b) the local environment, $l(n)$, corresponding to n is the set of all variables present in the procedure definition, where $l:N \rightarrow \mathcal{P}(V)$.

Definition. ***The partial relationship "<=".*** For two nodes n_1 and n_2 of an OR search tree, we write $n_1 <= n_2$ iff n_1 and n_2 are on the same path starting from the root and either $n_1 = n_2$, either n_1 is closer to the root than n_2. In the latter case we write $n_1<n_2$.

Definition. ***The global environment***, *g(n),* of a node *n* from an OR search tree is the union of the variable sets from all the local environments encountered on the path from the root node to the *n* node. That is $g(n) = \{v| (\exists x)\ x \leq n$ and $v \in l(x)\}$.

Definition. ***Access node, binding node.*** For every variable *v* from the local environment of a node *n*:

1) It exists a subset N of the tree nodes, subset called ***the set of the access nodes*** for *v*, in which every access node *m* has the property that $m \geq n$ (all the nodes below the current node are access nodes for variable *v*).
2) It exists a subset of access nodes called ***the set of the binding nodes*** for *v*, which may be described by using a partial function $bind:V \rightarrow N$, with the property that $bind(X)=u$ if the binding operation for variable *X* accesible at the current node happened at node *u*. Regarding the single assignment property, function *b* obbeys the requirement: if exists a node *m* such that $b(v)=m$, then $b(v)$ is undefined for all nodes $y<m$ and $b(v)=m$ for every $y \geq m$.

The set of binding nodes for *v* is the set $NL = \{m| b(v) = m\}$.

Lemma. ***Uniqueness property of the node bindings***. If a variable *v* has two distinct binding nodes n_1 and n_2, then $n_1 \not< n_2$ and $n_2 \not> n1$, that is n_1 and n_2 are not on the same path starting from the root.

Definition. A ***complete OR search tree*** is a tree in which:

1. for every leaf node *l*, the continuation of *l* is either empty (terminal node ending with success), either the first call from its continuation can not be processed due to a nondefined procedure (leaf node with failure);
2. for every non-leaf node *n*, we have in the OR search tree a child node for every procedure definition which may be used for running the first call from *n'*s continuation.

Definition. ***Variable access.*** A variable access operation implies determining the eventual binding node of that variable. We define for this purpose the function $access: V\ x\ N \rightarrow N$, with $access(X,u) = v$ iff $v = bind(X)$ and $v \leq u$.

5.2. A complexity analysis for OR parallelism implementation.

By the generic term of *variable management* we subsumed two distinct operations:

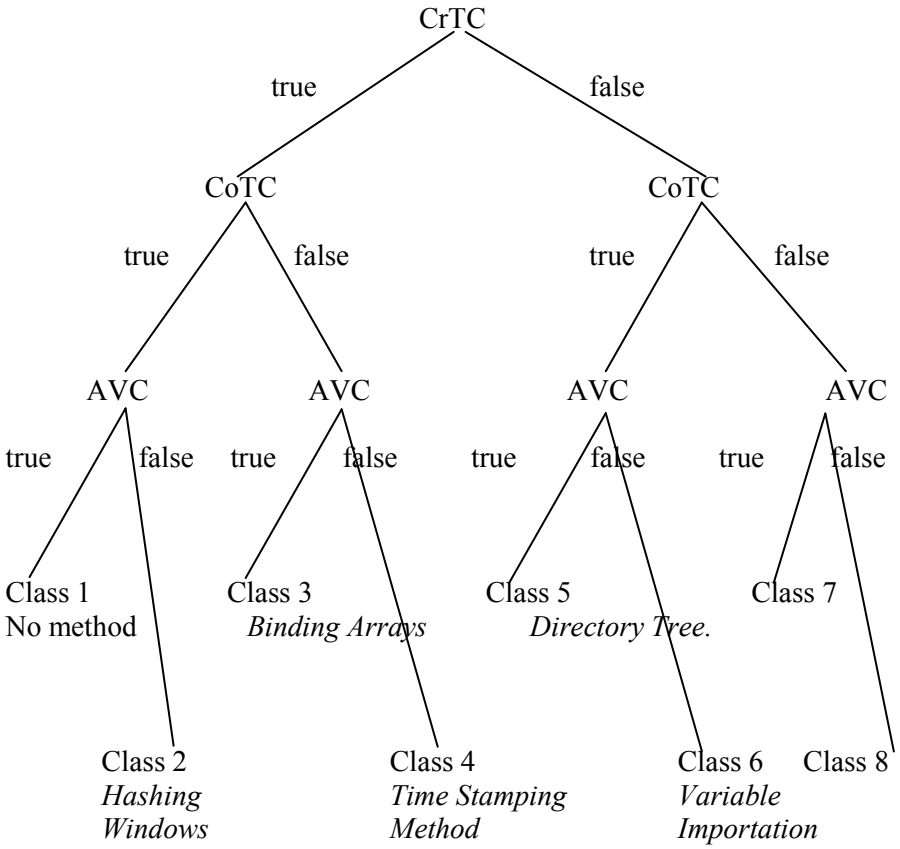
variable *binding* and variable *accesssing*. The **bind** operation does not imply special complexity overhead, because it performs an association operation at the level *of the current node*, being though a constant time operation and its complexity may be considered O(1). The main subject of our analysis will be the **access** operation. In its definition two nodes of the OR parallel tree are involved: the current node and the eventual binding node that has to be identified. Intuitively, it becomes evident the fact that this searching process can not be independent of the size of the search tree. In the proof of our theorem we present also the formal arguments of such an assertion.

Theorem. Variables management in an OR parallel implementation is a problem of complexity Ω(log N).
Demonstration.
The apel *Access(X,u)* ask from the level of *u* node the determination eventually of binding node of variable X (for take from there the asking value for X variable).
In a sequential system must go through the current branch from node to node until the evenly meet of binding node. As [12] it is established a complexity *O(n)* for this process (the problem of finding an element in a set). The question asked is if in a parallel system this disability cannot be overtake and *access* operation is taking course in constant time.
In a parallel process exists the possibility of simultaneous access of the nodes. In particularly, the number of these nodes depends on the architectural model considered. For keep the degree of generality of our analysis we will not fix a architectural particular model, but we will consider that in our model exists a finite and enough number of processors.
The reason of this statement is that we want that the limitation we will identify not become from the absence of architectural model, but be inherent of mechanism of implementation of OR parallelism.
At the level of précised architectural model, *access* operation is taking course at worst in *f(N)* time units (steps) where N is the number of nodes of the tree and function f is the one that we propose to determine as result of derivation of a limit of complexity (in ideal case we wish to be constant).
For every program, the factor of ramification of the OR parallel tree, is superior limited by a constant. The considering only of the binary trees in the next sentences don't affects the degree of generality of our demonstration.
The number of the nodes of complete deep binary tree *k* is $N=2^{k+1}-1$. Suppose that the maximum parallelization degree is reached (what we already proposed in our model – sufficient processors, but in finite numbers – so that at each apparition of a new task not to wait the release of computing resources).
In this conditions, the accessing capacity in a binary tree for an access operation is $CA= 2^{f(N)+1}-1$ nodes in those *f(N)* steps.
Observation. In the particularly case in which at each step we advance exactly a level of a tree, we have *CA=N*, so, capacity of accessing coincides the number of

the nodes of the tree, and those *f(N)* steps coincides with the deep of the tree, so *f(N) = k*.
The case f(N) < *k* expresses the situation in which the sequential processing it will be more performing that parallel one, so are of interest for our analysis remain the case of f(N) ≥ *k*.

6. A classification of OR parallel execution models.

We used the three Gupta-Jayaraman criteria as a basis for classifying the different OR parallel execution models. In figure 6. we can identify the criteria satisfied by the different methods present in the literature.



CrTC = Constant time task creation
CoTC = Costant time task switching
AVC = Constant time variable access

Figure 6. A classification of the OR parallel execution models

7. Conclusions

In this chapter we performed a presentation, a classification and an analysis of the most important OR parallel execution models.

The results obtained by Gupta and Jayaraman were used as a starting point for developing our own formal model for the problem of OR parallelism implementation and for a classification of the OR parallel execution models. In the literature, their presentation is made almost exclusively in a descriptive manner. The efficiency of the models is also explained or implied more on an intuitive basis than on strong mathematical reasoning. That is why we considered of a capital importance the proposal of a mathematical model to theoretically validate the informal and experimental results reported until now.

We tried that by means of our original contributions of this chapter – the formal model proposed in 5.1 and the complexity analysis performed in section 5.2 – to initiate a coherent framework for the development, presentation and analysis of the parallel logic systems, focused to help in obtaining more sound results and with a higher degree of generality.

REFERENCES

[1]**F.E.Allen (1969),** ***Program optimization*****, in** ***Annual Review in Automatic Programming 5****, International Tracts in Computer Science and Technology and their Applications*, vol.13, Pergamon Press, Oxford, England, pp.239-307;

[2]**R. Bahgat (1993),*****Pandora: Non-Deterministic Parallel Logic Programming,*** PhD Thesis, Department of Computing, Imperial College of Science and Technology, Feb. 1991, Published by World Scientific Publishing Co.;

[3]**J. Barklund (1990),** ***Parallel Unification****,* PhD Thesis, Uppsala University;

[4]**U. Baron, J.C. de Kergommeaux et al. (1988),** ***The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation of Results****,* in *Proceedings of the International Conf. on Fifth Generation Computer Systems,* Tokyo, pp. 841-850;

[5]**P. Borgwardt (1984),** ***Parallel Prolog using Stack Segments on Shared Memory Multiprocessors****, Proceedings of the International Symposium on Logic Programming,* Atlantic City, NJ, pp 2-11;

[6]**A. Ciepielewski, S. Haridi (1983),*****A Formal Model for OR-Parallel Execution of Logic Programs****, IFIP 83,* North Holland, P.C. Mason (ed.), pp.299-305;

[7]**J.S. Conery (1983),*****The AND/OR Process Model for Parallel Interpretation of Logic Programs****,* PhD. Dissertation, Univ. California, Irvine;

[8]**J.S. Conery (1987),** ***Parallel Execution of Logic Programs****,*Kluwer, Dordrecht;

[9]**V. Santos Costa, David H.D. Warren and Rong Yang (1991),*****Andorra-I: A Parallel Prolog System that Transparently Exploits both and- and or-Parallelism,*** in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* ACM Press, April 1991, pp. 83-93;

[10]**I. Foster and S. Tuecke (1993),** ***Parallel Programming with PCN*****.** Argonne National Laboratory, January;

[11]**D. Gelernter (1985),*****Generative communication in Linda*****.** *ACM Transactions*

on Programming Language Systems, 7(1), 1985, pp.80-112;
[12]**G. Gupta, B. Jayaraman (1993),** ***Analysis of OR-Parallel Execution Models,*** *ACM Transactions on Programming Languages,* 15(4) pp.659-680, Sept.;
[13]**G. Gupta (1993), M. Hermenegildo and V. Santos Costa –** ***And-Or Parallel Prolog: A Recomputation Based Approach,*** *New Generation Computing,* 11 (3-4) pp.297-323;
[14]**G. Gupta (1994),** ***Multiprocessor Execution of Logic Porgrams****,* Kluwer Academic Publishers, Norwell;
[15]***Z.* Halim (1986),** ***A Data-driven Machine for OR-Parallel Evaluation of Logic Programs****, New Generation Comput. ,* pp.5-33;
[16]**L. V. Kale (1987) ,** ***Parallel Execution of Logic Programs: the REDUCE-OR Process Model,*** in *Fourth International Conference on Logic Programming*, pages 616-632. Melbourne, Australia, May;
[17]**P.H.J.Kelly (1989),** ***Functional Programming for Loosely-coupled Multiprocessors*** MIT Press;
[18]**R.M.Karp, R.E.Miller and S.Winograd (1967),** ***The Organization of Computations for Uniform Recurrence Equation****s*, in *Journal of the ACM*, 14(3), pp.563-590, July;
[19]**L. Lamport (1974),** ***The Parallel Execution of DO Loops***, in *Communications of the ACM*, 17(2), pp. 83-93;
[20]**G. Lindstrom (1984),** ***OR-Parallelism on Applicative Architecture****, Proc. 2nd International Logic Programming Conf,* pp.159-170, July;
[21]**Y.Muraoka (1971),** ***Parallelism Exposure and Exploitation in Programs*****,** Ph.D. Thesis, Tech.Rep. 71-424, University of Illinois at Urbana-Champaign;
[22]**V.S. Sunderam (1990),** ***PVM: A Framework for Parallel Distributed Computing****,* in *Concurrency: Practice & Experience*, 2(4), pp.315-339;
[23]**Sverker Janson ; Seif Haridi (1990),** ***Programming Paradigms of the Andorra Kernel Language****,* Technical Report PEPMA Project, Sweden, Nov.;
[24]**P. Tinker (1988),*****Performance of an OR-parallel Logic Programming System****, International Journal of Parallel Programming,* 17, pp.59-92;
[25] **P.C. Treleaven (1990),** ***Parallel Computers: Object-oriented, Functional, Logic***. J. Wiley & Sons;
[26]**Alexandru Vancea(1999) ,** ***Paralelizarea automată a programelor***, Teză de doctorat, "Babeş-Bolyai" University Cluj-Napoca;
[27]**Monica (Ciaca) Vancea, Alexandru Vancea (2001),** ***An Analysis of Models for Parallel Logic Programming,*** in *Research Seminars, Preprint No. 1*, pp. 21-32;
[28]Monica Vancea(2004), *Tehnici de implementare în limbaje de programare logică paralelă*, Presa Universitară Clujeană;
[29]**D. H. D. Warren (1983),** ***An Abstract Prolog Instruction Set.*** Technical Report 309, Artificial Intelligence Center, SRI International, 1983.