Parallelism in Logic Programming: A Systematic Literature Review

Alessia Cerami
alessia.cerami@studio.unibo.it

Andrea Giordano
andrea.giordano5@studio.unibo.it

July 2021

1 Introduction

A logic program is expressed as a set of logical formulas precisely specifying *what* is true. This approach to programming is different from the others, where the concern is *how* a specific problem should be solved. Here, the specification how a problem is to be solved is expressed in the form of a precise sequence of steps executed by an engine.

One of the most attractive features of logic programming is the clean separation of logic and control. The efficiency of the solution to a problem can be improved without any change in the solution itself. The programming language Prolog utilizing one of many possible evaluation (control) strategies, is an example of a successful logic programming language.

The aim of this work is to understand how a logic program can be parallelized and what are the techniques studied in the last 30 years with an analysis of the problems arised.

A Systematic Literature Review process has been carried out following a methodological approach, with the claim of becoming a starting point for defining new research areas and future projects.

The work is structured as follows: section 2 describes and documents the research method carried out, section 3 shows the research results divided by category, section 4 summarizes the results in a table, section 5 answers the research questions and, finally, the conclusions are drawn in section 6.

2 Method

This is a Systematic Literature Review (SLR) that investigates the possibility to parallelize concurrent logic languages, like Prolog.

A SLR is a powerful technique that allows you to collect and summarize the information located in the scientific literature about a specific subject, using a rigorous and reproducible methodology. This SLR was carried out by defining a three-step review process: planning, conducting and reporting the review. This section presents the steps followed in order to perform the review, that are:

1. Identify the research questions.
2. Identify the sources.
3. Define a searching strategy.
4. Define a study selection criteria.
5. Assess the quality of study selection.

2.1 Research Questions

The research questions that have given rise to the need for this SLR are so defined:

1. Is it possible to parallelize Prolog? (G)
2. What are the different techniques of parallelization in logic programming? (Q1)
3. What are the problems of implicit parallelization? (Q2)
4. What are the algorithms to implicitly parallelize logic programming? (Q3)

2.2 Source

The search process was conducted on some of the most relevant digital libraries, which are described by Kitchenham and Charters [42]:

- Google Scholar: https://scholar.google.com
- IEEE Xplore: https://ieeexplore.ieee.org/Xplore/home.jsp
- Springer Link: https://link.springer.com/
- ACM Digital Library: https://dl.acm.org

In order to define the keywords, we adopterd the following approach:

1. Main terms are extrapolated from the research questions.

2. Synonyms and verbal forms are evaluated.

The resulting keywords are:

- **LP**: Logic program, Logic programming
- **Parallel***
- **Technique**
- **Algorithm**

The sources were interrogated using a predefined query, obtained from the previously identified keywords:

- *(Or\-Parallel* OR And\-Parallel* OR "Or Parallel" OR "And Parallel") AND ("Logic Program" OR "Logic Programming")*

2.3 Search Strategy

The documents taken into consideration are those in English published since 1989, exception for five articles [15], which was published in 1981, [43], which was published in 1986, [83], which was published in 1987, [30] and [88], which were published in 1988. When the search engine allowed it, those with restricted access were filtered.

The amount of articles obtained by submitting the query to the libraries is very abundant. Specifically, 17144 documents were found at the time of the research. All the papers collected from the query is shown in the table 1.

After a brief research conducted on Google Scholar, we noticed that, after the first 100 results, ordered by relevance, the correlation with the topic decreases. So, with this in mind, only the first 100 results were chosen from each library.

	Google Scholar	IEEE Xplore	Springer Link	ACM DL	Total
Query1	3710	231	12691	512	17144
Query 1 date range 1989-2021	3150	215	12236	422	16023

Table 1: Results of the search strategy.

2.4 Study Selection

In order to obtain the primary studies from those identified, which provide direct evidence about the research question, we applied a filtering process, wchich is carried out by means of constraints applied in the form of exclusion and inclusion criteria, reducing to the most relevant papers. The criteria that have been applied are defined as follows:

- Inclusion Criteria:
  - Full paper (not extended abstract).
  - Articles published in peer-reviewed journals or conference proceedings.
  - Sources describing in depth the topic of this research.
  - Sources answering the research questions.
  - In some particular cases, chapters.
- Exclusion Criteria:
  - Documents with limited access.
  - Papers not discussing the subject of this SLR: the main check was carried out on the abstract and, sometimes, on the introduction.
  - Papers not answering the research questions: the check is carried out on the full paper.
  - Duplicate reports of the same study.
  - Whole books and thesis.

In order to efficiently select the primary studies, we followed the subsequent points:

1. Duplicates have been searched for and deleted.
2. Titles and abstracts were used as the first filter to check the correlation with the topic to be discussed.
3. The complete papers have been reviewed in depth.

At the end of this selection process, the documents amount to 227.

2.5 Quality assessment

From the 227 remaining papers, the content was checked in depth and some documents were removed. In the end, five final papers have been added to the collection ([15], [30], [43], [83] and [88]), so, the amount of primary studies is 92. The added papers were obtained using the same query but with a wider range and, thus, those are prior to 1989.

3 Results

In the following section, the information extracted and used to write this SLR are reported. In order to provide a clear view and a better explanation, this section is divided into categories and subcategories according to a specific aspect of parallel logic programming.

3.1 Types of parallelism in Logic Programming

This section presents the macro categories of logic programming parallelism. In order to give to the reader a complete view of the argument, all the different types of parallelism are presented but the focus of this SLR is only on the implicit one.

Before presenting the macro types is important to briefly talk about nondeterminism. There are two type of them in logic programming as explained in [69] and [39] that are listed below.

- *Don't-Know nondeterminism*: It occurs when the program have more than one clause to choose from, it picks one nondeterministically and create a choice point used to backtrack in case of failure of the clause. This means that all the possible alternatives are tried and all the solutions can be found. In this type of nondeterminism only successful computations are considered as results while the failings are not observed.
- *Don't-Care nondeterminism*: It occurs when only a clause is choosen: no choice point are created and no backtracking is performed even if the clause fails. This means that only one alternative is explored so that only one result is obtained. In this type of nondeterminism any computation is observed (hence failure is a valid result) so there is only one output that is logically correct.

Following, based mainly on [16], [37], [22] and [35], are shown the macro types of parallelism in logic programming with a brief explanation.

3.1.1 Explicit parallelism

This type of parallelism involves the extention of logic programming languages with explicit constructs that enable the control of parallelism. This means that in this type of parallel programming, the programmer has to control parallelism manually through the code.
The explicit control of parallelism need a higher skilled programmer who can takes advantage of the extending constructs to produce very efficient code, but it can create difficulties in debugging and testing so it is not suited for everyone. There are three subtype of explicit parallel logic languages:

- Those that add *explicit message passing* primitives to Prolog.
- Those that add *blackboard* primitives to Prolog used by multiple processes running in parallel to communicate with each other.

- Those based on *guards*, *committed choice*, and *dataflow synchronization*.

In order to keep this SLR focused, we consider these approaches only marginally or in those cases where they introduce execution mechanisms that are applicable also in the case of implicit exploitation of parallelism.

3.1.2 Implicit parallelism

This type of parallelism involves the capability of the logic program engine to perform parallel operations autonomously, extracting parallelism from logic programs without any programmer intervention. Inn this type of parallelism, in contrast to the explicit one, there are no extentions to the logic language so that the programmer need not to worry about parallelism that, instead, is achieved automatically running multiple processes that works together to compute the results.
The aim is to speed up existing and new logic programs without any change to the code. There are three forms of implicit parallelism:

- *AND-parallelism*: The parallelization lays in the selection of the next literal to be solved. This allows to solve multiple literals concurrently. This form of parallelism, as mentioned into [90] and [93], can again be divided into:
  - *Independent AND-Parallelism*: literals of a clause are independent from each other, thus they do not share variables.
  - *Dependent AND-Parallelism*: literals of a clause are dependent from each other, thus they share variables.
- *OR-parallelism*: The parallelization resides in the selection of the clause to be used in the computation of the resolvent. This allows to try multiple clauses in parallel.
- *Unification-parallelism*: The parallelization is at the level of the unification process when complex terms are present.

Mixing these types of implicit parallelism together is possible, but involves the increase of problems that need to be handled.

3.1.3 Hybrid parallelism

One last form of parallelization is the hybrid solution which takes advantages of both implicit and explicit parallelism. It is used a user-accessible concurrent language, that typically is an extension of Prolog, and allows quite detailed manual parallelization.
An example of hybrid parallel model is ACE [56] in which the programmer can either use standard Prolog and let the compiler take care of detecting implicit parallelism or use explicit language constructs to express parallelism.

3.2 Implicit types of parallelism in Logic Programming

The aim of this SLR is to collect the knowledge about implicit parallelism in logic programming. To keep clarity and avoid confusion into the reader, this section is divided in subcategories based on the subtypes of parallelism. As already mentioned before, all the parallelism types can be mixed together, but it is an implementation matter because there are no conceptual differences, so, the combination of types is discussed in section 3.5.4. Following, a detailed description is given for each form of implicit parallelism.

3.2.1 OR-Parallelism

OR-parallelism resides in the selection of the clause to be used in the computation of the resolvent. It means that, this parallelism, can be exploited every time a subgoal unifies with more than one rule head: the bodies of each rule can be executed in parallel. A clear explanation can be found in [22].
This parallelism is the way of seraching for all the solutions to the query by exploring the whole search space generated by the multiple clause applicable at each resolution step.
Or-parallelism not only allow the parallel execution of multiple clause, but also enable the concurrent search into the whole search space generated for different alternatives applicable to the selected subgoal.
This type of parallelism is more frequent in applications that explore a large search space via backtracking.
Even if, at first sight, this parallelization is simple, in section 3.3 will be shown that many problems arise.

3.2.2 Independent AND-Parallelism

As mentioned in [90] and [93], Independent AND-Parallelism (IAP) provides a means for performing concurrent evaluation of subgoals in a clause. In this form of parallelism, a clause is seen as a problem that can be divided into subproblems which can then be solved simultaneously. Each of this subproblems will produce its own solution. At the end, the final solution is obtained by putting together the results of the subproblems.
From a logic programming point of view, this means that each literal appearing in the selected clause body can be executed in parallel. In IAP the parallel computation exploited is only that between independent subgoals and, therefore, is crucial to identify the independence.
Literals are considered *independent* when they do not share any unbound variable at run-time. That condition guarantees that the execution of one clause will not affect the other's execution. In this way, there is no need of introducing any form of synchronization during parallel execution.
The main focus in IAP is the detection of independence since only independent subgoals are allowed for AND-parallel execution. This can be done exclusively at run-time or exclusively at compile-time or by marking at compile-time selected literals and, when independence cannot be determined statically, generating a reduced set of efficient parallelization tests, to be checked at run-time, as shown in [5]. It exists a narrower form of

IAP, called restricted independent AND-Parallelism, that implies stronger independence constraints.

3.2.3 Dependent AND-Parallelism

The Dependent AND-Parallelism (DAP), [93], is similar to IAP with the difference that literals share dependencies from each other. It occurs when and-parallelism is allowed between subgoals which share unbound variables at the time of invocation of the goals. Thus, processes compete in binding a shared variable or cooperate if the goals share the task of creating the binding for the common variable. In both cases some form of synchronization is needed when subgoals are executed concurrently because their computation is affected by others.
This type of parallelism is harder to be exploited for Prolog, in fact, often, it is handled converting the language to a committed choice language or adopting other changes to operational semantics.

3.2.4 Unification parallelism

This parallelization, as explained in [35], arises during the unification of the arguments of a goal with the arguments of a clause head. The different argument terms can be unified in parallel as can the different subterms in a term. Thus, unification of two complex terms is broken down in pairwise unification of the different arguments. In this way, the sequence of unification can be performed in parellel.
This type of parallelism has a big disadvantage: due to its fine granularity, it requires specialized architectures in order to achieve results of any interest or, otherwise, it will incur in performance degradation due to the overhead. For this reason it is not studied in more detail in this SLR.

3.3 Problems of implicit parallelism

Parallelism has always been a challenging field and logic programming, that is intrinsically parallelizable, wants to be parallelized but many problems arise that need to be handled. Following, basing on [35] and [22], are explained some of the biggest issues common to every type of implicit parallelism:

- **Granularity control**: Knowing the size of a task, in term of execution time and space, is really important and it become almost a must if the parallelization is made on a distributed system where comunication is a bottleneck. In the reality, it is not possible to know the exact size of the task but only an estimation can be made. Many techniques and algorithms have been proposed to assess the size of tasks but, obviously, those implies some overhead that have to be taken into consideration and which can slow down the system if not properly managed.
- **Scheduler**: Many factors influence when a piece of work have to be executed and to which worker have to be assigned. This duty is in the hand of the scheduler, therefore,

lot of effort goes into its design and implementation. For example, Aurora has about five different schedulers and this shows the importance and the impact it has. This core worker have to deal with work balance, speculative work, preservation of Prolog semantic and other scheduling related issues. It is probably the main concern when implementing a parallel Prolog system.

- **Preservation of Prolog semantic**: Supporting full Prolog semantic is not a simple task because it contains side effects that have to be executed in a deterministic order. Techniques to give this support have been studied but they always implies a lost on parallelism. Furthermore, the cut operator introduce the problem of speculative work.

- **Process-based versus Processor-based**: Is it better to spawn a process for each goal encountered (process-based) or to spawn multiple thread ahead of time and assign them parts of the computation (processor-based)? The answer to this question is based on many aspects, hence, both of them are correct under specific circumstances. Many systems adopted the processor-based because it is easier to implement and, often, gives better performances. The choice is based on heuristics and no rules have been found to pick the most suited to the problem that need to be tackled.

- **Architectural Influence**: This problem is about how the architecture of the system is structured because it can significantly impact the performances. Many architectures have been studied and each one tackles problems in different ways obtaining different result but, actually, there are no evidences that a particular architecure is the best and research is needed.

- **Parallel Execution Visualization**: Even if not strictly related to the implementation of implicit parallelism, a note goes to the tools that shows a graphical representation of parallel execution. These tools, that help the user debugging code and system implementor for fine-tuning scheduler, are still under research and need more study and, in fact, no one of the available one handle all the type of parallelism together.

3.3.1 OR-Parallel problems

Besides the common problems, OR-Parallelism have only a main issue to tackle that is known as **Multiple environment representation**, as mentioned into [79].
First of all, when implementing this type of parallelism, you have to visualize it and the easiest way is through the or-parallel search tree. At this point it appear obvious that the main problem in implementing this form of parallelism is the management of multiple bindings. In fact, the problem arises in the moment of representing and accessing the conditional bindings that are variables that can be assigned by more process.
Different approaches have been proposed to handle multiple bindings and will be presented in section 3.4.2.

3.3.2 Independent AND-Parallel problems

Following, are reported the main problems that arise with Independent AND-parallelism that can be encotered into three different phases.

- **Ordering Phase**: During this phase, the main problem consist of the **detection of data dependencies** among subgoals. The condition of independence require that there are no shared variable at the run-time execution. There are various approaches for detectiong data-dependency and they will be described into section 3.4.3.
- **Forward Execution Phase**: During this phase, the only major problem is the efficient **determination of the goals** that are ready for independent and-parallel execution.
- **Backward Execution Phase**: When the failure of a subgoal occurs or more solutions are requested, the backward execution phase is entered. Backward execution, or backtracking, turns out to be difficult because subgoals are distributed in different processors.
  During this phase, the main problem is to **determine the subgoal to which** execution should **backtrack**, which means how to support intelligent backtracking. This is due to the mismatch of the order of subgoals in the stack with their backtracking order. Furthermore, backtracking may need to continue to the preceding subgoal, which may still be executing at the time backtracking takes place. These problems are complicated by the fact that independent and-parallel subgoals may have nested independent and-parallel subgoals currently executing which have to be terminated or backtracked over. Different approaches to backtracking in this kind of framework have been proposed as the one cited into [54] and [55].

3.3.3 Dependent AND-Parallel problems

Following, are reported the main problems that arise with dependent AND-parallelism as expressed in [84].

- **Detection of parallelism**: This issue consists of determining which subgoals need to be taken into consideration for DAP execution.
- **Management of DAP goals**: This issue consists of activation and management of parallel subgoals. The resolution of this issue is the same as in independent And-parallelism.
- **Management of shared variables**: This issue consist of validation and control of shared variables;
- **Backtracking**: This issue is due to the complexity of dealing with distributed backtracking.

3.3.4 AND/OR-Parallel problems

The problems that arises form combining and-parallelism and or-parallelism are various. Not only they are the sum of both types of parallelism but also those regarding control of execution, representation of environment and memory management. Furtermore, AND/OR parallelism has some requirements that are in contrast. For example:

- OR-parallelism focuses on improving the separation between the parallel computations, by assigning separate environments to the individual computing agents.
- AND-parallelism relies on the ability of different computing agents to cooperate and share environments to construct a single solution to the problem.

Other problems the combined systems could have are: whether they should support sequential Prolog semantics, whether to recompute solutions of independent goals or to reuse them or, finally, how to deal with an additional level of scheduling, that is to determine whether an idle worker should perform or-parallel work or and-parallel work.

3.4 Algorithms and techniques of implicit parallelism

In this section some alorithms and techniques are showed to the reader. First are explained those that handle common problems and then a paragraph is made for each type of parallelism containing only techniques and algorithms aimed at that specific type.

3.4.1 Handling common parallelism problems

Not all the common parallelism problem have been dealt with appropriately while some of them have attracted a lot of attention. The following list contains some of the mechanism discovered:

- In [86] and [87] is presented a scheme for controlling the grain size of tasks (goals) of Prolog programs on a process-based parallel logic programming system running in a distributed manner. The proposed model concentrates mainly on how to use granularity information effectively, performing most of the analysis at compile-time using only a few run-time tests.
- A more complex model for granularity control is explained in [45]. GRANLOG takes a Prolog program as input and gives a granulated program as final result. It is composed by three submodule: Global Analyzer, Grain Analyzer and Complexity Analyzer.
- Another method to control granularity is presented in [44], that offers solutions for the many problems involved, for both the cases when upper and lower bound information regarding task granularity is available, and for a generic execution model. Such problems include, on one hand, estimating the cost of goals, of the overheads associated with their parallel execution, and of the granularity control technique

itself. On the other hand, there is, also, the problem of devising, given that information, efficient compile-time and run-time granularity control techniques. This method, differently from the other, takes into consideration the overhead that it brings into the execution.

- In [85] is reported a method for preserving Prolog semantics in which the execution of logic programming can be seen as the process of maintaining a dynamic tree. The operational semantics of the language determines what operations on the tree are of interest. That method is used in DAP.
  Preventive strategies enforce the correct order of variable bindings by assigning producer or consumer status to each subgoal that shares a given dependent variable. The strategy used is: the leftmost subgoal that has access to the variable is designated as the producer for that variable, while all the others are consumers.
  The problem of testing leftness has been studied in a dynamically growing tree and has been provided an efficient pure pointer machine algorithms for several variants of the problem.

- The articles [26] and [27] explain how to preserve Prolog semantic both pure OR-parallel system and in independent AND-parallel system. The former is the "left-most principle" that states that side-effects can be executed only when the node is the left-most in the tree. The latter is a technique that require that the side-effect subgoal need to be suspended until all the preceding subgoal (left to right precedence) are finished.

- Another method to preserve Prolog semantic is presented in [38]. The idea is that instead of recompute every subgoal, some of the results can be reused and an algorithm to handle this problem is presented. The choice, *recompute or reuse*, is based on side-effects and where they appear in the resolution tree, using the left-most principle.

- In [14] is reported an example of scheduling and handling of speculative work for or-parallelism . Its function is to match processors with work and, in doing so, the scheduler either looks for or-nodes with a non-empty list of clauses or for idle processors. In both cases the scheduler will traverse the execution tree. A scheduling strategy is the way in which the tree is traversed.

- A scheduling algorithm is presented in [74]. This scheduler is aimed at distributed parallel logic programming systems but it is also suitable for those with shared memory architecture and the difference lies only in the parallel grain. This algorithm schedules, under centralized control, all of the parallel subtasks which are created during the execution of a program and allows the processors choicely to give their priorities to some tasks so that the solution to a question can be acquired as quickly as possible and that the parallel system obtains a higher speedup. In this algorithm co-scheduling is introduced and load-balancing of processors is considered.

3.4.2 Techniques used in OR-Parallelism

The problem of multiple binding representation, mentioned in section 3.3.1, is solved by designing a mechanism where each branch has some private area used to store conditional bindings applicable to itself. In [83] and [31] are reported many ways of achieving this mechanism. Some of them are:

1. Each or-branch owns a **private memory space**, which can be an array or a hash table, for its conditional bindings from where the binding is accessed whenever it is needed. Unconditional bindings can be shared among their corresponding branches as sequential Prolog system.

2. Keeping a **separate copy of the environment** for each branch of the tree, so that every time branching occurs at a node the environment of the old branch is copied or recreated in each new branch;

3. All conditional bindings are recorded into a **global data-structure** and each conditional binding is associated with a unique identifier that explains which or-branch it belongs to.

To each approach is associated a cost that is noncostant and incurr at different times. In [31] are reported the criteria that an ideal or-parallel system should respect.
Based on the criteria selected, there are different techniques of implementing this form of parallelism. Following, based on [83], [30], [78], [31], [2], [1], [17], [19], a brief description of them is reported.

- **Directory Tree Method**: In this method, each branch of the OR-tree is associated to a process. The binding environment of each process consists of *contexts*, created for each clause that is invoked, and *directories*, used to index the contexts efficiently. When a new branch is created, the new child node gets a copy of the directory of its parent. If an entry in the directory points to a context that contains no unbound variables, this context is shared. Otherwise, if the entry points to non-ground context, the context is copied into the environment of the child.

- **Hashing Window Method**: In this method, each node in the OR-tree has its own hashing window that is a simple hash-table where the conditional bindings of a node are stored. The address of a variable is used to compute the hash-value of a variable. Unconditional variables are not included in the hashing window, but in a hashing windows of one of the ancestors of a node.

- **Binding Array**: In the binding array technique, each worker has an auxiliary data structure called the binding array. When a conditional variable gets bound, the binding is stored in the binding array of the worker. To ensure consistency, when a worker switches from one branch of the or-tree to another, it has to update its binding array by *deinstalling* bindings from the trail of the nodes that are in the previous branch and *installing* the correct bindings from the trail of the nodes in switched branch.

- **Time Stamping Method**: In this methos, all bindings, which are distinguished by time-stamps of creation and the process-id of the process that created them, are stored in a global area, visible to all processes. An unbound variable is represented by the fact that no bound variable is present with the correct time stamp and process-id.

- **The Argonne-SRI Model**: The OR-parallel tree is divided into *favored*, *private*, and *shared* sections. Bindings that are created in the favored section are stored in-place in the node while that made by the other sections are stored in a hash table. Only a favored branch can directly access a variable.
  This model modify the SRI Model by introducing a treatment of favoured variables. Each processor has a private binding array which only contains unfavoured bindings. Favoured bindings are implemented by assigning to the variable value cell with a special mark. furthermore, this model guarantees that all variable access, even unfavoured, are very fast, constant-time operations.

- **The Manchester-Argonne Model**: The Manchester-Argonne Model, differently from the Argonne Model, does not create the hash table for an arc unless, and until, another process wants to share that arc. The process that creates the arc inserts the bindings in a binding list. The second process, which wants to share the arc, creates the hash table it requires from the information in the binding list. Each process also needs a private binding array to store the unfavoured bindings it itself makes.

- **Kabu-Wake Model**: The Kabu-Wake is a model that exploits the copy on stack technique. It associates to each binding a logical date, such a date being managed by each worker is used to discard invalid bindings.

- **The MUSE Model**: In the Muse or-parallel Prolog system, the binding environment of each process is not shared with others. When a process switches to another node in the or-parallel tree, it constructs the whole stack from that node to the root node in its memory. The copy of the stack can be regarded as the private data structure for accessing the variable bindings.

- **The Sparse Binding Array (SBA)**: In the Sparse Binding Array (which derives from Binding Arrays), each worker has a private virtual address space that fully shadows the system's shared address space. In other words, every work has its own shadow of the whole shared stacks that is used to store shared variables. Thus, the execution data structures and unconditional bindings are stored in the shared address space, while only conditional bindings are stored in the shadow area.

Other techniques used in OR-parallelism are:

- **Complexity of OR-Parallelism**: In [61] is presented a method to compute the complexity of an OR-Parallel system in term of lower-bound and upper-bound. It is explained how to reduce the upper-bound complexity and, in addition, an algorithm that reduce the complexity of checking if a branch is the leftmost is shown. This method is based on abstraction of the operations needed and, as result, it achieves to reduce the upper-bound cost without downsides.

- **Last Alternative Optimization - LAO**: In [32] is reported an optimization technique that simplify the structure of the Prolog search tree allowing for more efficient traversal. It is based on the principle of flattening which affirms that reducing the number of levels of nesting of an execution tree, generally results in improved performance. LAO is also based on the *principle of duality* according to which given a technique that has been developed for and-parallelism, this technique dual can be successfully applied to an or-parallel system.

- **Hierarchical Pincers Attack Search**: In [40] is described a Prolog Or-parallel processing scheme which operates on a multiprocessor system. In that scheme the OR-tree is searched from right and left side of each created subtree. Its intent is to allow coarser task granularity and to reduce the frequency of the task assignment and data transfer between processes.

- **Improvement of the Conery's model**: In [88] is reported an improvement to the *Conery's model* whose aim is to increase the running efficiency.
  Conery's algorithm represents the generator-consumer relationship by using data dependency graph which is constructed when the AND process is created. This model, in some case executes vain work, especially during the backward execution. The improved algorithm can avoid such kind of vain and infinite work, and can improve the running efficiency.

3.4.3 Different approaches to exploit independent AND-Parallelism

In [90] is reported how early models for independent AND-parallelism attempt to identify producer/consumer relationships among goals. Some of these models use the reordering technique which avoids the performance loss in post-execution reconciliation but sacrifices some parallelism. The goals that share same variable are implicitly divided into a producer goal and one or more consumer goals of the variable at run-time, so, the producer goal generates the values for the shared variable, while the consumer goals consume it. Goals that not contain any shared variables are executed in parallel.
Other approaches are that that do not use any reordering technique.

The problem of data dependency is solved by several approaches. Below, a brief description of thte various approaches existing in literature is reported.

- **Input/Output Mode**s: As introduced in [75], one way to overcome the data dependency problem is to require the user to specify the *mode* of the variables, that is, whether an argument of a predicate is an input or output variable. The Prolog system can assume the subgoal with *input-mode* variables to be a consumer, and the subgoal with *output-mode* variables as a producer. Input variables of a subgoal will become bound before the subgoal starts and output variables will be bound by the subgoal during its execution.

- **Static Data Dependency Analysis**: In this technique, the goal and the program clauses are globally analyzed at compile time assuming a worst case for subgoal

dependencies. No checks are done at run-time. The advantage is that no overhead is incurred at run-time.

- **Run-Time Dependency Graphs**: In this technique, an auxiliary structure, the *dependency graph* that is created at run-time, is used to detect the dependency information and to examine bindings of relevant variables every time a subgoal finishes executing. The advantage of this scheme is that maximal independent and-parallelism could be potentially exploited. This technique can be used to generate the maximal parallelism, but the runtime checking has a large cost.

- **Restricted And-Parallelism (RAP)**: The restricted AND-parallelism (RAP) approach takes place between the static data dependency analysis and the run-time dependency graphs methods. Data obtained from a compile-time analysis of the dependencies is used to speed up a run-time analysis of the data. In [68] is reported a system for exploiting restricted-and-parallelism in logic programs on a distributed non-shared memory architecture. As mentioned in [90], this approach take the advantage of the second and third ones and encapsulates the dependency information in the code generated by the compiler (CGE - Conditional Graph Expressions) along with the addition of some extra conditions on the variables.
  In [89] is proposed a new scheme, the *Complete Graph Scheme*, exploiting RAP that is based on dataflow computation and implemented in a non-shared execution model. It uses a technique that deals with binding conflicts by combining compile-time analysis of the clauses involved, with simple checks on variables at run-time low cost, based on the results of variable instantiation.
  This model has been efficiently implemented in the &-Prolog and-parallel system 3.5.2.1.

- **The Conery's Model**: In this method, a dataflow graph is created during the ordering phase. In this way, the producer–consumer relationships between subgoals became explicit.

- **APEX Model**: As reported in [78] and [31], the APEX (And-Parallel EXecution) model determines independent AND-parallelism by means of a token passing system to implement forward execution. In the ordering phase , all first occurrences of variables are assigned a token for that variable. A subgoal becomes executable when it receives tokens for all the uninstantiated variables in its current binding environment. Parallelism is exploited automatically when there is more than one executable subgoal in a clause. The APEX model incorporates intelligent backtracking [54], which is performed at the clause level.

- **CAAP - Compiling Approach for exploiting AND-Parallelism**: In the article [37] is described the functioning of a parallel logic programming system, which, during the precompile phase, use an abstract interpretation and a particular scheme called CAAP.
  The former tecnique is used to infer entry modes of procedures and to analyze data-dependency globally in logic programs. The latter is a framework for exploiting the maximal independent AND-parallelism with minimal expense at compile-time,

especially if the data-dependence graph is incorporated.
In the article, the CAAP scheme is described in three phases: analysis of entry mode, derivation of exit mode, determination of Conditional Graph Expression.

- **The UOUDG and UUDG Algorithms**: In [7] are presented two concrete algorithms which generate code annotated for unrestricted independent and-parallelism, starting from sequential code. The proposed algorithms process one clause at a time by working on a directed acyclic dependency graph. The first one, UOUDG, parallelizes a clause preserving the order of the solutions by respecting the relative order of literals in the original clause. The other, is implemented by randomly selecting, among the sets of goals which can be joined at every moment, the one with the *lowest cardinality*.
  The idea behind the annotation algorithms is to publish goals for parallel execution as soon as possible and to delay issuing joins as much as possible but always respecting the dependencies in the graph.

- **Sketch of a Shared Memory Implementation**: In [9] is reported an implementation, for shared-memory multiprocessors, that distributes responsibilities among several layers. Here, agents wait for work to be available, and execute it if so. Every agent is created as a thread and coordination among them is performed by means of shared data structures.
  This implementation exploits and-parallelism in logic programs with the objectives of simpler machinery and more flexibility. The approach is based on raising the implementation of some components to the source language level by using more basic high-level primitives than the fork-join operator and keeping only some relatively simple operations at a lower level.

In order to solve the problem of the efficient determination of the goals that are ready to be executed have been adopted different approaches. One work was [70] where flexible related scheduling and memory management approaches are studied. Another methodology is that described in [51] which adapts scheduling mechanisms developed for or-parallel systems to the case of independent and-parallel systems. In doing so, in the same way in which an or-parallel system tries to schedule work that is more likely to succeed first, and-parallel systems will benefit from scheduling work that is more likely to fail first.
The advantage of applying this methodology comes from the fact that most IAP systems support intelligent forms of backtracking over and-parallel calls which allow to quickly propagate failure of a subgoal to the whole parallel call. Other techniques used in IAP are:

- **Last parallel call optimization - LPCO**: In [52] is reported an optimization technique, LPCO, that has been developed in the context of non-deterministic programming, whose intent is to merge, whenever possible, distinct parallel conjunctions.
  It is triggered when the last call in Prolog clause is itself a parallel conjunction.

- **Nested Parallel Call Optimization - NPCO**: Another optimization technique reported in [52] is NPCO. It is a generalization of the previous one and it is triggered when a Prolog clause has a nested parallel conjunction and all goals follows the

parallel conjunction. Its intent is to allows savings in memory consumption and to speed up the execution of parallel programs.

3.4.4 Techniques used in dependent AND-Parallelism

Dependent AND-Parallelism is a form of parallelization that generalizes independent and-parallelism by allowing the concurrent execution of subgoals accessing shared variables and competing or cooperating in the creation of a binding for unbound variables.

The resolution of the detection of parallelism problem is similar to the process of annotating ("parallelizing") a program using independent and-parallelism. This is because DAP is finer grain instance of the general principle of independence, applied to the level of variable bindings.
Automatic and semi-automatic detection of potential valid sources of unrestricted DAP in logic programs was used in [36] in which the authors identified and made explicit the variables that are shared between the goals in the parallel conjunction.
A detection technique of and-parallelism was developed in [72]. It exploits and-parallelism using the *producer-consumer* scheme in which two subgoals can be executed in parallel only when there is no data-dependence between them.
The approach consists of three phases:

1. Analysis of entry modes,
2. Derivation of exit modes,
3. Determination of execution graph expressions

The management of shared variables in a dependent and-parallel execution requires the need of guaranteeing mutual exclusion during the creation of a binding for a shared variable and guaranteeing that the outcome of the computation respects sequential Prolog semantics.
That problem has been discussed in various works. Following, some work are briefly reported:

- **Committed Choice Parallelism**: This approach has been implemented by many committed choice languages, such as Concurrent Prolog, Parlog, GHC and KL1, explained into [77], [22] and [75]. This languages support dependent and-parallel execution and handle shared variables through a scheme based on the notion of producer and consumers which are explicitly identified at the source level. they are declared explicitly through mode declarations, or implicitly through strict rules on binding of variables that are external to a clause.
  Furthermore, in that languages concurrency is achieved by reducing several goals in parallel. The issue of insuring consistent bindings of shared variables is addressed by only allowing a variable to be bound once.

- **Andorra-I** (described into 3.5.4.3): In this system, [18], terms that need to be matched with a compound term are locked and a special instruction is added by the compiler at the end of the term construction to release the lock.

- **DDAS - Dynamic Dependent AND-Parallel Scheme**: In [90] and [71] is reported a goal synchronisation method used in Stream-parallelism that exploits dependent AND-parallelism in Prolog programs using an implicit synchronisation mechanism at the level of unification. Based on the compile-time annotation scheme for IAP, Prolog programs are compiled to the Extended Conditional Graph Expressions where an additional annotation is used to indicate dependent variables.
  In the DDAS model, goals sharing unbound variables can be handled by a producer-consumer relationship: consumers execute in parallel with the producer until they are about to bind a variable. They are then suspended until the producer binds the variable. The system takes advantage of the information on dependencies of variables to extract the parallelism and to make both backward and forward executions more precise.

- **Detecting And-Parallelism without literal ordering**: In [13] is reported an AND-parallelism detection scheme which does not order literals or analyze data dependency. The simple concept of variable holding allows the system to use AND-parallel literals concurrently without causing binding conflicts, the main problem in AND-parallelism.
  In that proposal, the system does not need literal ordering. It only needs to understand if a literal is ready to be executed at a given time and, when a ready literal is executed, its unbound variables are held for possible binding. The literal, whose unbound variable depends on a held variable, is declared not ready. In this way, binding conflicts are impossible.

- **Stream-Parallelism**: As mentioned in [78], it exists an approach to solve the binding conflict problem through cooperations among processes that share variables. It exists two kinds of stream-parallelism: *pipelining* and *cooperative* parallelism. In the former type, clauses that share common variables are linked in a producer-consumer relashionship. If a producer clause binds a variable, the bindings are passed to the consumer clause for further treatment. The pipelining approach requires that the producer-consumer relations are known and these information are derived from data-dependencies graphs.
  The latter type, allows a two-way communication between clauses sharing variables. Differently from pipelining, this form of parallelism allows that clauses that share variables can be evaluated in parallel. This approach relies on mode-declaration, specifying which attributes are taken as input and which as output. This information is used to set up the appropriate communication channels.

- **Literal Dependence Net**: A method to automatically extract literal dependencies is presented in [92]. This techniques, starting from a logic program, gives a LDN as output. The resulting net is an *arc-classifed digraph* that represent all four types of primary program dependencies in a concurrent logic program. This graph is mainly used in some algorithms to achieve AND-Parallelism but also in some tool for the Parallel Execution Visualization.

- **Argument Dependence Net**: In [91] is presented a technique for computing slicing of a parallel logic program. In the article it is explained how to compute a

ADN that can be used to "split" a program in slices that, then, can be executed in parallel. The algorithm is composed by three parts that, as output, gives the ADN. It is claimed that slicing at argument level is more precise than that at literal level.

3.4.5 Techniques used in AND/OR-Parallelism

In this section are presented models and techniques which support more than one type of parallelism.

3.4.5.1 The Sync Model

In [43] is proposed a parallel execution model for logic programming that realizes AND-parallelism and OR-parallelism.
The former is implemented constructing, in a dynamic way, the data flow graph of the literals into the clause body. In order to avoid binding conflicts, only one AND process is designed as the producer of a shared variable. All the other AND processes are designed as consumer of such variable.
The latter is achieved by the use of synchronization signals to the stream of partial solutions of the various AND processes.

3.4.5.2 Reduce-Or Parallel Model - ROPM Model

In [58], [59], [60], [41] is reported an execution model which exploits both AND (independent AND-parallelism) and OR parallelism in logic programs. It is based on a modification of the and-or tree, called the Reduce-Or Tree.
There are two types of nodes in the reduce-or tree, the *reduce-nodes* and the *or-nodes*. Every node in the tree contains the bindings of all variables that are either present in the node or are reachable through this node.
The corresponding *reduce-process* and *or-process* are associated with each reduce-node and or-node respectively. An or-process is responsible for creating an or-parallel branch, while a reduce-process deals with executing the body of the clause, which is represented as a data-join graph.
ROPM model uses a *partial solution set* to avoid the binding conflicts of conditional variables, thus, each node is associated with a partial solution set, which contains the bindings of all variables that are either local variables or the variables reachable through this node. newline ROPM is capable of generating the maximum available parallelism in Prolog programs and it is also complete. That is, every solution to the problem is found.

3.4.5.3 Paged Binding Array-Based Model

This model divides the binding array into pages and maintains a *Page Table* with a binding array. All the available workers are divided into teams. Different teams work in or-parallel with each other, while different workers within a team work in independent and-parallel. Different and-parallel computations within an or-parallel computation share the same binding array. Each one of them will use a different page, requesting a new page when it

runs out of space in the current one.
The PBA-based model also employs recomputation of independent goals, and, therefore, can support Prolog semantics, as described into [29] and [27]. In [28] is proposed an extention of the PBA model used for implementing And/Or parallelism, the *Shared paged Binding Array*. Differently from the PBA, it require the use of a global and a shared binding array.

3.4.5.4 The AO-WAM Model

In [78] is reported another model that exploits multiple parallelism. Infact, the AO-WAM model exploits independent and-parallelism by the method of conditional graph expressions and or-parallelism by the extended binding arrays technique. A cross-product node is introduced to apply reuse scheme on And/Or parallelism.

3.4.5.5 The PEPSys Model

In [35] and [30] is reported a model that combines and-parallelism and or-parallelism using a combination of time-stamping and hashing windows for maintaining multiple environments. In PEPSys, each node in the execution-tree has a process associated with it and each process has its own hash window. All the bindings of conditional variables generated by a process are time-stamped and stored in the process's hash window.
In that model, Independent and-parallel goals have to be explicitly annotated by the programmer.
That model can handle only two and-parallel subgoals at a time otherwise the subgoals are nested in a right associative relationship. If or-parallelism is nested within and-parallelism, then and-parallel branches can generate multiple solutions.

3.4.5.6 The Copy-On-Write model

In the Copy-On-Write model, reported in [17], each worker maintains a separate environment. Whenever a worker wants to share work from a different worker, it copies all the execution stacks. The particularity is that although stacks will be logically copied, they will be physically copied only on demand. This is possible, thanks to the Copy-On-Write mechanism provided by most modern Operating Systems.

3.4.5.7 Distributed Last Call Optimization

In [57] is proposed an algotithm used in AND/OR parallel execution of logic programs. This optimization reduces the total amount of copies during unification.
The most significant advantage of this optimization is the reduction in the number of messages that serves to improve memory utilization.
Another important optimization that serves to reduce messages and improve execution time is grain size control, that is the average amount of computation per message created during the computation.

3.4.5.8 Parallel execution models

- In [49] is reported a parallel execution model based on MUSE 3.5.1.5. This model supports restricted AND-parallelism, stream AND-parallelism and also OR-parallelism. Furthermore, the AND process solves the subgoals by creating OR-processes for each one.
  There are two scheduling work algorithms proposed in that model. The first one is based on allocating the processing elements to the branches of the AND/OR tree by a method matched with the execution of the Prolog language. The second one is based on static allocation for the processing elements to the branches of the tree starting from the top-most.

- In [12] is reported another parallel execution model whose language is based on pure Horne Clause Logic with the addition of a set of annotations. That model exploits And-parallelism and also Or-parallelism.
  It employs differeent techniques to execute a logic programming:

  – Performs frame inheritance for newly created goals.
  – Create a data-dependency graph to represent relations among goals.
  – Create process structures that are based on the information into the data-dependency graph.

3.4.5.9 Logatithm resolution model

In [4] is reported *Logarithm*, an all-solution parallel logic resolution model, supporting AND parallelism and OR parallelism, based on an intelligent distributed backtracking algorithm. It is used to speed up programs performing searches in large domains.
Stream AND-parallelism is also supported: partially instantiated terms can be communicated incrementally by a process to processes sharing variables with it.

3.4.5.10 A Multisystem-based parallel interpreter

In [23] and [24] an approach to restricted And-parallelism combined with Or-parallelism is presented. The proposed parallel interpretation model is based on what it is called *multisystems*. This consist in compact representations of several systems that are often called a set of bindings or also, a binding environment of variables. The approach presents some good characteristics:

- All the known techniques to exploit Restricted And-parallelism can be used with the proposed approach.

- The number of search paths could be exponentially smaller than that needed by existing interpreters (using Or-parallelism).

- Regarding the total running time, the small number of search paths created is a crucial factor because it avoids their execution.

- The reduction in the required memory space in the proposed interpreter (even exponential in a best case) follows from the fact that a multisystem is equivalent to a set of multiple bindings but represented in a compact and hierarchical way.

3.4.5.11 Support to Cut and Side-effects

The articles [26] and [27] present a technique to support full Prolog semantic focussing on extra-logical predicates like cut and other side-effects. The support is studied for an or- and independent and-parallel system. The crucial point presented is that it is fundamental to *recompute* solutions intead of *reusing* them. The limit of this model is that it is based on or-parallelism that use Conditional Graph Expressions (CGE). The method used to preserve Prolog semantic is an extension of the left-most principle already shown, that add the concept of "locally left-most".

Another method to support full Prolog semantic is presented in [38]. Its aim is to handle side-effects while mixing *recomputation* and *reuse* of subgoals using a "Selective Recomputation" approach. The order preserving concept is based on a *side-effect execution permit* token that is passed in the OR-forest in accordance with the execution order in sequential Prolog. In this article it is also introduced the concept of *soft side-effect* that increase the parallelism exploitable.

3.5 Parallel implementations

In this section are illustrated the various implementantion that have been proposed for supporting OR-Parallelism, Independent AND-Parallelism, Dependent AND-Parallelism and systems that support more than one type of parallelism.

3.5.1 Implementations Supporting OR-Parallelism

Following, the most famous systems that handle OR-parallelism are reported with a brief explanation. Those which support more than one type of parallelism are presented in section 3.5.4

3.5.1.1 Aurora

In [46] and [6] is presented Aurora, an or-parallel implementation of Prolog implemented for shared memory multiprocessors. Its implementation uses the SRI model to represent different bindings of the same logical variable corresponding to different branches of the search space. In the SRI model, a group of workers cooperate to explore the Prolog search tree, which is defined implicitly by the program and needs to be constructed explicitly during the course of the exploration. In order to share or-parallel work, Aurora protects the choice points with locks avoiding that several workers steal the same piece of work.
Furthermore, Aurora system supports cut and standard Prolog built-in predicates including those which produce side-effects. Procedures to this predicates are required to suspend until they are in the leftmost branch of the tree.

3.5.1.2 The YapOr System

In or-parallelism, alternative branches of the search tree may be executed in parallel and this may produce binding conflicts for shared variables. The YapOr system, [79], [21], [62] is used to solve this problem by implementing the *environment copying model*. In that model, each process keeps a separate copy of its own environment, but in an identical address space. Thus, permits to freely store assignments to shared variables without conflicts. Every time a process shares work with another one, all the execution stacks are copied to ensure that the requesting worker has the same environment state down to choice point where the sharing occurs.
Synchronization is needed at work sharing operations to ensure that each untried alternative is only explored once.

3.5.1.3 The ThOr Model

In [21] is reported another model that supports OR-parallelism. As in the YapOr system, in the ThOr system the memory model is based on *stack copying*. It has been designed to take advantage of the existing YapOr code but, having each stack at the same virtual memory addresses, does not hold true in ThOr. That is, to share work we need to have several copies of the same choice-point at different virtual memory addresses. To solve this problem, in ThOr is used the *shifted copying*. That method consists of: copy the memory between the workers sharing work and, then, adjust pointers.
Although shifted copying adds a linear overhead to copying operations, it offers some important advantages:

- It allows using the thread infrastructure as is.
- It allows shifting between stacks with different sizes.

3.5.1.4 OPTYap System

In [63], [66] is proposed the OPTYap system (that builds on YapOr), that is the first available system that can exploit parallelism from tabled programs. (Tabling is about storing and reusing intermediate answers for goals).
The OPT (Or-Parallelism within Tabling) model considers tabling as the base component of the system. Each computational worker behaves as a full sequential tabling engine. The or-parallel component of the system is triggered to allow synchronized access to the shared part of the search space or to schedule work. The OPT model split the search tree into a public and several private regions, one per worker. When a worker runs out of alternatives to exploit, it enters in scheduling mode. The YapOr scheduler is used to search for busy workers with unexploited work.
Experimentation on achieving scalability in tabled logic programming have been made through that system and are reported in [64].
An optimization strategy for OPTYap is presented in [65] where the aim is to deliver tabled answers as soon as it is found that they are safe from being pruned.

3.5.1.5 MUSE

Muse, which is reportedd in [2] and [1], is based on multiple sequential Prolog engines. Both conditional and unconditional variables are not shared in the MUSE system. Each processor operates as a sequential Prolog engine with its local memory.
The MUSE or-parallel Prolog system assumes a number of extended Prolog engines called workers. Each worker consists of two components: the *engine* and the *scheduler*. The former works as a sequential Prolog engine, the latter maintains the work between engines. Furthermore, the MUSE scheduler supports the sequential semantics of Prolog and efficient scheduling of speculative work.

3.5.1.6 PALS

In [80] is proposed a new distributed Or-parallel execution model, PALS, that is oriented for completely distributed memory multiprocessors and efficiently exploits or-parallelism. It adopts a new strategy, the i*ncremental stack-splitting* that is a modification of the stack copying, used to implement or-parallel Prolog efficiently in the MUSE system. The new strategy allows to reduce the communication during stack copying and make its implementation efficient because, the PALS system, only copies the difference between the stacks of the sending processor and the receiving processor.

3.5.1.7 YapDss System

In [67] is proposed an or-parallel Prolog system that implements stack splitting to exploit or-parallelism in distributed memory platform. YapDss uses a *multi-sequential approach* to represent computation state and work sharing among the computational workers.
YapDss uses a *bottommost policy* to dispatch work for or-parallel execution. Work is shared through copying of the execution stacks and diagonal splitting is used to split the available work.
In this implementation there is no support for cuts or side-effects.

3.5.1.8 PloSys

The article [47] present a parallel distributed system focussing on side-effects. The objective of PloSys is to build a *Prolog-like* OR-parallel logic programming system for distributed memory parallel architectures. "Prolog-like" means that PloSys will implement Prolog cut and side-effects.
The PloSys computational model is multi-sequential: each worker importing work copies the state of the worker exporting work. Scheduling is centralised. The management of cut and side-effects aims at limiting the number of messages exchanged during the execution. Cuts are handled with speculative work, while side-effects are executed only when on the left-most node. Side-effects executed in speculative work should be *reversible* either by restoring the previous state of computation, or by performing them in some cache that will be cleared if the side effect is not to be validated.

3.5.2 Implementations supporting independent AND-Parallelism

Following, a description of systems that exploit Independent AND-parallelism is provided.

3.5.2.1 &-Prolog

In [22] is reported the &-Prolog system that exploits independent AND-Parallelism combining a compiler performing independent detection with an efficient implementation of AND-parallelism on shared-memory multiprocessors.
This system supports both automatic and user-expressed parallelisms. This can be expressed explicitily with the &-Prolog language.
Input code is processed by several compiler modules: The *parallelizer* performs a dependency analysis on the input code using a conditional graph-based approach. It also receives information from the *Side-Effect Analyzer* on whether each non built-in predicate and clause of the given program is pure, or contains a side-effect. This information adds dependencies to correctly sequence such side-effects. The parallelizer then encodes the resulting graph using the & operator producing an "annotated" (parallelized) &-Prolog program.
The parallelizer also receives information from the *granularity analyzer* regarding the size of the computation associated with a given goal, as can be seen in [44].
In order to improve its performance, a tool, written in Prolog, was implemented. IDRA (IDeal Resource Allocation) [25] collects traces from sequential executions and uses them to simulate an ideal parallel execution of the same program.

3.5.2.2 &-ACE

In [51], [50], [33] is reported the &-ACE system that exploits independent AND-Parallelism and that is based on the logic programming paradigm and the Prolog technology. An And-Parallel Prolog system works by executing a program that has been "annotated" (parallelized) with parallel conjunctions. These parallel conjunction annotations are either inserted by a parallelizing compiler or hand-coded by the programmer.
Execution of all goals in a parallel conjunction is started when control reaches that parallel conjunction.
&ACE is very effective in exploiting parallelism, even from rather fine-grained applications, as the one explained into in [34] which was simplified the computation And-Or tree in order to make its traversal cheaper and faster

3.5.2.3 Shared-memory implementation

In [8] is reported an a high-level implementation model for independent and-parallelism which fully supports non-determinism, through backtracking, and provides flexible solutions for some of the main problems found in previous and-parallel implementations. That implementation provide a solution that is only valid for the parallel execution of goals which have exactly one solution each. This model is based on the multi-sequential marker model introduced by &-Prolog to which share the concept of agents. In that model, agents

are created with a small stack (which can grow on demand) and they wait for some work to be available. They do not continuously search for new tasks to be performed, in order to avoid active waiting. That choice was made for improve speedup in logic programs.

3.5.2.4 Extended RAP Execution Model

In [11] is reported an extend RAP execution model that supports intelligent backtracking and side effects. It consist of:

- *Forward algorithm*: During the farward execution, an environment will be created on stack when a clause head is matched, to store permanent variables and information used to continue execution and backtracking. The system will place onto its system stack a choice point. When a clause in which subgoals can backtrack intelligently is encountered, a special choice point will be created on the stack.
- *Backtracking algorithm*: It is invoked any time a failure occurs. The algorithm starts by checking the most recently created item from the system stack. If a normal choice point is the most recent item, then backtracking remains the same, as that for sequential execution. If a special choice point is the most recent one, a check on its type is made and the corresponding type of intelligent backtracking is used.

3.5.3 Implementations supporting dependent AND-Parallelism

Following, a description of systems that exploit Dependent AND-parallelism is provided.

3.5.3.1 DASWAM system

In [71] is reported the DASWAM system in which shared variables are represented as a new type of tagged cell and each shared variable is uniquely represented. Thus, all workers access the same representation of the shared variable. Producer and consumer status is determined via a search operation, performed at the time of variable binding. In that system no locks or mutual exclusion mechanism are required.

3.5.3.2 ACE

The ACE system [31], [53], [33] supports all form of parallelism.
The dependent and-parallelism has been incorpored in ACE using the *Filtered Binding Model* that directly encode in the access path itself the information that allows a subgoal to discriminate between producer and consumer accesses. Thus, the filtered binding model exploits restricted DAP and performs all operations in constant-time. The restriction is that unbound shared variables are not allowed to be bound to each other.
Independent and-parallelism is also exploited via conditional graph expressions.

3.5.4 Implementations supporting multiple type of parallelism

Here are exposed the main systems that support multiple type of parallelism.

3.5.4.1 The competition model

In [48] is reported a parallel execution model for logic programs developed in the framework of the AND/OR process model and supports both AND-parallelism and OR-parallelism. This model, called *The Competition Model* contains two kind of processes:

- An *OR-process*: Solves a single lieteral goal. If there are n clauses resolvable with the goal, then it creates n AND-processes to solve these n resolvents.
- An *AND-process*: Solves a conjunction of literals. If the conjunctioon consists of n literls, it creates n OR-processes for these literals.

Furthermore, it also consist of two algorithms:

- The *forward execution*: Allows a higher degree of parallelism adopting the concept of competition.
- The *backward execution*: Involves three operations: select the backtrack literal, select the reset literals and freeze some literals.

3.5.4.2 FORDAP

In [93] is reported a full OR/AND parallel execution model which exploits all types of implicit parallelization. It consists of:

- *EGE* (Execution Graph Expression)
- *Parallel Execution Tree*
- *Decomposition Scheme*
- *Solution Collection Scheme*

A logic program, which is submitted to FORDAP, is firstly transferred into the EGE at compile time, the parallel execution tree is formed (according to the request and the EGEs) at runtime, the request is then decomposed into multiple parallel goals end, at the end, all the solutions of the goals are collected and the final solution is built.

3.5.4.3 Andorra-I

The idea behind this system, [18], [78], [73] is that if a goal has only one solution, it can be executed regardless of what other goals have been initiated. Such goals are called *determinate*. If no determinate goals can be found, a branch point for the leftmost goal is inserted. Each solution that is found for this branch point may spawn a different parallel branch (OR-parallelism using the binding array technique). This model leads to AND-parallel coroutines if two determinate goals are executed in parallel. In other words, dependent and-parallelism is obtained by having determinate goals execute in parallel. The different alternatives to a goal may be executed in or-parallel.

In Andorra-I, workers are arranged into teams that cooperate to exploit or-parallelism. Workers within a team cooperate to exploit and-parallelism. A team is composed of a master and some or no slaves. The configuration of workers into a team is decided by the user. In [20] is explained that, in doing so, the problem that arises is the selection of a work, and more generally, where to redeploy workers to. The Andorra-I component responsible for doing that is the *preconfigurer*.

3.5.4.4 The Extended Andorra Model - EAM

In [78], [83] is proposed another model that exploits both AND-parallelism and OR-parallelism. It extends the basic Andorra model [76] in that it lets all AND-goals execute prematurely.
If only one solution for a goal is found, it is treated like a determinate goal in the basic Andorra model. Otherwise, goals can freely be executed as long as they do not influence the computation of the other goals. In this case, it is suspended.
Once execution cannot proceed, the producers *publish* all solutions to the consumer goals. For each solution, a new instance of a consumer goal is started.
The EAM is a general model, more powerful than the Basic Model, since it can limit the search even further by local searching. It also exploits more parallelism since it exploits all major forms of parallelism present in logic programs, that is: or-parallelism, independent and-parallelism, and dependent and-parallelism.
An aspect of EAM is that is does not distinguish between independence and dependence of conjunctive goals: it tries to execute them in parallel whenever possible. Furthermore, the extended Andorra mode subsumes both the committed choice logic programming and non-deterministic logic programming.

3.5.4.5 OASys - Or/And SYStem

In [81] is reported a software implementation designed for AND/OR-parallel execution of logic programs.
In OASys, the search space is represented by a tree in which the nodes, called U-nodes, denote the unifications between the calling predicates and the heads of the corresponding procedures. The connection between two u-nodes create a link that is successfull if the unification performed produce a consistent binding.
OASys supports OR-parallelism by searching simultaneously different paths of the search tree. It also supports AND-parallelism by executing in parallel the U-nodes belonging to the same path. The execution terminates in three circumstances:

- All links are successful, thus, a solution is found,
- One link is unsuccessful, thus, there is a failure in conjunction,
- Unification in a U-node fails, thus, mismatch.

3.5.4.6 DAOS - Scalable And-Or Parallelism

In [10] is reported an and/or-parallel Prolog system, DAOS, that exploits parallelism based on the distributed shared memory architecture. Both shared-memory-based communication and message passing techniques are adopted in that system.
Essentially, DAOS extends the binding array techniques to Sparse Binding Array to manage shared variables in and/or-parallel environment and extends RAP-WAM to support independent and-parallelism.

3.5.4.7 A Hierarchical Parallel Execution Model

In [82] is reported a hierarchical parallel execution model for Prolog programs that is based on And/Or parallelism as coarse-grain parallelism (it exploits binding array technique for Or-parallelism and RAP for And-parallelism) and uses an And/Or tree, and parallel unification as fine-grain parallelism.
That execution model can be logically broken into two levels of hierarchy:

- The first level is the And/Or parallel execution level
- The second level is a parallel unification level

3.5.4.8 Petri Net resolution

Firstly proposed in [15] and then proposed again in [3] one procedure for automated reasoning in a logic program is through Petri net models that can handle all possible types of parallelisms.

4 Analysis

This section summarizes the data collected with two tables. Table 2 shows all the articles grouped by years and search engine. Note that articles prior to 1989 have been found using the same engines used for the SLR and they are reported too. Table 3 presents a macro-classification of the papers. This should be used to quickly find references aimed at something specific instead of reading this whole paper.

Analysis				
	Before 90	'90-'94	'95-'99	2000-Today
IEEE Xplore	[88] [13]	[48] [37] [11] [12] [68] [40] [41] [26] [4] [89] [23] [59] [60]	[82] [50] [56] [49] [24] [39] [20] [32] [52] [34] [86] [54] [81] [17]	[55] [91] [64] [3] [21] [45]
ACM	[15]	[93] [90]	[5]	[35] [79]
Scholar	[43] [83] [30] [69]	[2] [46] [57] [31] [29] [92] [22]	[51] [28] [36] [47] [25] [27] [44] [18] [33] [6] [78] [73]	[74] [76] [19] [66]
Springer	[16]	[77] [72] [1] [14] [58]	[75] [71] [70] [53] [38] [87] [10] [62] [61]	[63] [80] [67] [84] [65] [85] [7] [8] [9]

Table 2: References partition based on years and search engine

Analysis		
	Algorithms and Techniques	Implementation
Common problems	[14] [26] [27] [44] [38] [86] [87] [74] [45] [85]	[44]
OR-Parallelism	[83] [30] [88] [2] [1] [40] [31] [32] [78] [17] [61] [19]	[2] [1] [46] [47] [6] [62] [63] [80] [64] [67] [65] [66] [21] [79]
IAP	[37] [68] [89] [31] [90] [75] [51] [70] [52] [78] [54] [7] [9]	[11] [22] [51] [50] [44] [25] [33] [34] [8]
DAP	[13] [72] [77] [22] [90] [92] [75] [36] [71] [18] [78] [91]	[31] [71] [33] [53]
AND/OR-Parallelism	[43] [30] [41] [12] [57] [58] [26] [29] [4] [23] [59] [60] [28] [24] [49] [27] [38] [78] [17] [35]	[15] [83] [48] [93] [82] [20] [18] [78] [10] [73] [81] [76] [3]

Table 3: Summary of the references used in this SLR

5 Discussion

Following will be provided an answer to each research question based on the results collected and the analysis carried out.

5.1 Q1: What are the different techniques of parallelization in logic programming?

Up to now, it exists, in literature, three class of thought about parallelism into logic programming. They are divided into:

- Explicit parallelism
- Implicit parallelism
- Hybrid parallelism

The first form of parallelism, the explicit one, implies the explicit involvement of the programmer who, by inserting particular constructs at the code level, has manual control over parallelism.
The advantage of this type of parallelization is that execution can be more efficient and the programmer have full control on what happens. The downside is that it requires a skilled programmer.

The second form of parallelism, the implicit one that is also the one at which this Systematic Literature Review aims, implies the capability of the logic program engine to perform parallel operations autonomously without the programmer intervention.
This parallelism is in turn divided into three subcategories that are:

- And-parallelism: Subgoals in a conjunction are solved in parallel, and clauses with the same head are tried sequentially.
  And-parallelism can be further divided into two types according to the way shared variables are handled. That is to say:
  - Independent And-Parallelism: Given a conjunction of subgoals, if they do not share the same runtime bindings with each other, then they can be run in parallel giving rise to IAP. Static data dependency checking is not enough for IAP, because in some cases binding dependency arises at runtime.
  - Dependent And-Parallelism: Given a conjunction of subgoals, subgoals sharing variables or runtime binding are executed in parallel as long as dataflow dependencies are preserved. DAP is more difficult to handle than IAP because the dependent binding restricts the arbitrary exploitation of and-parallel execution.
- Or-parallelism: Given a goal that matches multiple clause heads, the bodies of different clauses are executed in parallel, and subgoals in a conjunction in each body are executed sequentially. Or-parallelism is an efficient way to explore alternative solutions in parallel.
- And/Or-parallelism: Subgoals in a conjunction are executed in parallel and clauses with the same head are tried in parallel too.

The last form of parallelism, the hybrid one, takes advantage of both implicit and explicit parallelism.

A deep explanation about the different types of parallelism can be found in sections 3.1 and 3.2.

5.2 Q2: What are the problems of implicit parallelization?

Implicit parallelism has many problems. Some of them are in common to every type while others are specific.
In section 3.3 are reported all the known problems arising from parallelization of logic programs, divided in paragraphs depending on the type of implicit parallelism they are related to. Some of these problems have been studied in more details and have been tackled with lot of techniques and algorithms. Others, due to their minor impact on performances, have been addressed only by few researchers.
Despite the amount of studies, some of the bigger problems have not been solved. Furthermore, the complexity and cost of manteinance brought to the dereliction of the systems.

5.3 Q3: What are the algorithms to implicitly parallelize logic programming?

As explained in details in 3.4 many algorithms and techniques exist to handle different problems. Some of these methods tackle problems common to each type of parallelism while others try to improve performances adding optimizations.
Many proposal were made for each single type of parallelism both algorithms/technique and whole models.
Instead, for And/Or-parallelism, only a few algorithms and techniques were studied while a lot of models were presented. This imbalance in favor of models is probably due to the vast number of already existing techniques and algorithms for each type of parallelism, which have to be merged. This union raise new problems that have to be handled with ad hoc models.

6 Conclusion

The Systematic Literature Review is a powerful technique that allows you to obtain a general and methodological view on a certain topic of interest. This work, that identifies and analizes the different forms of parallelization in logic programming and the different techniques and implementations developed up to now, followed the standard SLR methodology, making sure the whole process was documented and easily reproducible.

In a first moment, a total of 16023 potential articles were identified using automatic searching in four digital libraries (Google Scholar, IEEE Xplore, Springer Link, ACM Digital Library) by submitting them the found query. After screening the articles and performing a pilot search, we selected and analyzed 92 of them.

The results of the systematic review indicate that Parallel Logic programming is, or

at least was, a highly active area of research. As can be seen, most articles are prior to 2000, meaning that interest has decreased. From what we understood, the reason is the complexity of manteinance of Prolog parallel implementation. Nonetheless the answer to the main question we had is yes, it is possible to parallelize Prolog.

This work aims at summarizing the great amount of existing papers about implicit parallelism in logic programming. Three are the different types of parallelism available: explicit, implicit and hybrid. Implicit parallelism is again divided in three subtype: OR-Parallelism, AND-Parallelism (independent and dependent) and Unification parallelism. Different approach has been proposed to address each subtype both theoretically and practically.

This SLR should help future work on parallelism in logic programming giving a good knowledge base and supporting new research and new projects.

7 Attachments

A public folder containing all the references used in this SLR is available at: here

References

[1] Khayri A. M. Ali and Roland Karlsson. Full prolog and scheduling or-parallelism in muse. *Int. J. Parallel Program.*, 19(6):445–475, 1990.

[2] Khayri A. M. Ali and Roland Karlsson. The muse or-parallel prolog model and its performance. In Saumya K. Debray and Manuel V. Hermenegildo, editors, *Logic Programming, Proceedings of the 1990 North American Conference, Austin, Texas, USA, October 29 - November 1, 1990*, pages 757–776. MIT Press, 1990.

[3] Alakananda Bhattacharya, Amit Konar, and Ajit K. Mandal. Concurrent resolution in logic programming using petri net models. In *International Conference on Computational Intelligence and Multimedia Applications (ICCIMA 2007)*, volume 2, pages 43–47, 2007.

[4] Jean-Paul Bodeveix and Érick Bizouarn. A parallel prolog execution model theoretical approach and experimental results. In *The Seventh International Parallel Processing Symposium, Proceedings, Newport Beach, California, USA, April 13-16, 1993*, pages 7–15. IEEE Computer Society, 1993.

[5] Francisco Bueno, María García de la Banda, and Manuel Hermenegildo. Effectivness of abstract interpretation in automatic parallelization: A case study in logic programming. *ACM Trans. Program. Lang. Syst.*, 21(2):189–239, March 1999.

[6] Vanusa Menditi Calegario and IC Dutra. Parallel conventional systems versus parallel logic programming systems on distributed shared memory architectures. Technical report, Citeseer, 1998.

[7] Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. Annotation algorithms for unrestricted independent and-parallelism in logic programs. In Andy King, editor, *Logic-Based Program Synthesis and Transformation, 17th International Symposium, LOPSTR 2007, Kongens Lyngby, Denmark, August 23-24, 2007, Revised Selected Papers*, volume 4915 of *Lecture Notes in Computer Science*, pages 138–153. Springer, 2007.

[8] Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. A high-level implementation of non-deterministic, unrestricted, independent and-parallelism. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 651–666. Springer, 2008.

[9] Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. Towards a high-level implementation of execution primitives for unrestricted, independent and-parallelism. In Paul Hudak and David Scott Warren, editors, *Practical Aspects of Declarative Languages, 10th International Symposium, PADL 2008, San Francisco, CA, USA, January 7-8, 2008*, volume 4902 of *Lecture Notes in Computer Science*, pages 230–247. Springer, 2008.

[10] Luís Fernando Castro, Vítor Santos Costa, Cláudio F. R. Geyer, Fernando M. A. Silva, Patrícia Kayser Vargas, and Manuel Eduardo Correia. DAOS - scalable and-or parallelism. In *Euro-Par '99 Parallel Processing, 5th International Euro-Par Conference, Toulouse, France, August 31 - September 3, 1999, Proceedings*, volume 1685 of *Lecture Notes in Computer Science*, pages 899–908. Springer, 1999.

[11] Si-En Chang, Mark L. Manwaring, and Y. Paul Chiang. Extended restricted and-parallelism execution model. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing, SPDP 1990, Dallas, Texas, USA, December 9-13, 1990*, pages 471–474. IEEE Computer Society, 1990.

[12] Albert C. Chen and Chuan-lin Wu. A parallel execution model of logic programs. *IEEE Trans. Parallel Distributed Syst.*, 2(1):79–92, 1991.

[13] W. Chung and W. Day. Detecting and-parallelism in logic programs without literal ordering. In *1989 Eighth Annual International Phoenix Conference on Computers and Communications*, pages 315,316,317,318,319, Los Alamitos, CA, USA, mar 1989. IEEE Computer Society.

[14] Andrzej Ciepielewski. Scheduling in or-parallel prolog systems: Survey and open problems. *Int. J. Parallel Program.*, 20(6):421–451, 1991.

[15] Keith L. Clark and Steve Gregory. A relational language for parallel programming. In Arvind and Jack B. Dennis, editors, *Proceedings of the 1981 conference on Functional programming languages and computer architecture, FPCA 1981, Wentworth, New Hampshire, USA, October 1981*, pages 171–178. ACM, 1981.

[16] Jacques Cohen. Logic programming and parallelism. *Annales Des Télécommunications*, 44:274–282, 1989.

[17] Vítor Santos Costa. COWL: copy-on-write for logic programs. In *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99), 12-16 April 1999, San Juan, Puerto Rico, Proceedings*, pages 720–727. IEEE Computer Society, 1999.

[18] Vítor Santos Costa, Ricardo Bianchini, and Inês de Castro Dutra. Evaluating parallel logic programming systems on scalable multiprocessors. In Hoon Hong, Erich Kaltofen, and Markus A. Hitz, editors, *Proceedings of the 2nd International Workshop on Parallel Symbolic Computation, PASCO 1997, July 20-22, 1997, Kihei, Hawaii, USA*, pages 58–67. ACM, 1997.

[19] Vítor Santos Costa, Ricardo Rocha, and Fernando M. A. Silva. Novel models for or-parallel logic programs: A performance analysis. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings*, volume 1900 of *Lecture Notes in Computer Science*, pages 744–753. Springer, 2000.

[20] Inês de Castro Dutra. Distributing and-work and or-work in parallel logic programming systems. In *29th Annual Hawaii International Conference on System Sciences*

(HICSS-29), January 3-6, 1996, Maui, Hawaii, USA, pages 646–655. IEEE Computer Society, 1996.

[21] Inês de Castro Dutra, Ricardo Rocha, Vítor Santos Costa, Fernando M. A. Silva, and João Santos. Scheduling or-parallelism in yapor and thor on multi-core machines. In *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 1581–1590. IEEE Computer Society, 2012.

[22] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Comput. Surv.*, 26(3):295–336, 1994.

[23] G. Escalada-Imaz. A multisystem-based parallel-interpreter of logic programs with restricted and-parallelism and or-parallelism. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, volume 1, pages 402–407 vol.1, 1994.

[24] G. Escalada-Imaz and F. Manya Serres. Reducing time by avoiding process serialisations in or-parallel interpretation of logic programs. In *1995 IEEE International Conference on Systems, Man and Cybernetics. Intelligent Systems for the 21st Century*, volume 5, pages 4539–4544 vol.5, 1995.

[25] M. J. Fernández, Manuel Carro, and Manuel V. Hermenegildo. IDRA (ideal resource allocation): Computing ideal speedups in parallel logic programming. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume II*, volume 1124 of *Lecture Notes in Computer Science*, pages 724–733. Springer, 1996.

[26] Gopal Gupta and Vítor Santos Costa. Complete and efficient methods for supporting side-effects and cuts in and-or parallel prolog. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, SPDP 1992, Arlington, Texas, USA, December 1-4, 1992*, pages 288–295. IEEE Computer Society, 1992.

[27] Gopal Gupta and Vítor Santos Costa. Cuts and side-effects in and-or parallel prolog. *J. Log. Program.*, 27(1):45–71, 1996.

[28] Gopal Gupta, Vítor Santos Costa, and Enrico Pontelli. Shared paged binding array: A universal datastructure for parallel logic programming. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, page 824. MIT Press, 1995.

[29] Gopal Gupta, Manuel V. Hermenegildo, and Vítor Santos Costa. And-or parallel prolog: A recomputation based approach. *New Gener. Comput.*, 11(3):297–321, 1993.

[30] Gopal Gupta and Bharat Jayaraman. Combined and-or parallel execution of logic programs. Technical report, NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE, 1988.

[31] Gopal Gupta and Bharat Jayaraman. Analysis of or-parallel execution models. *ACM Trans. Program. Lang. Syst.*, 15(4):659–680, 1993.

[32] Gopal Gupta and Enrico Pontelli. Last alternative optimization. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing, SPDP 1996, New Orleans, Louisiana, USA, October 23-26, 1996*, pages 538–541. IEEE Computer Society, 1996.

[33] Gopal Gupta and Enrico Pontelli. High performance parallel logic programming: The ACE parallel prolog system. In Moreno Falaschi, Marisa Navarro, and Alberto Policriti, editors, *1997 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE'97, Grado, Italy, June 16-19, 1997*, pages 25–32, 1997.

[34] Gopal Gupta and Enrico Pontelli. Optimization schemas for parallel implementation of nondeterministic languages and systems. In *11th International Parallel Processing Symposium (IPPS '97), 1-5 April 1997, Geneva, Switzerland, Proceedings*, pages 428–435. IEEE Computer Society, 1997.

[35] Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, 2001.

[36] Manuel V. Hermenegildo, Daniel Cabeza Gras, and Manuel Carro. Using attributed variables in the implementation of concurrent and parallel logic programming systems. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, pages 631–645. MIT Press, 1995.

[37] Shouren Hu, Yaoqing Gao, Zhiyi Hwang, and Yungui Ci. Design and implementation of a parallel logic programming system. In *Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence, TAI 1990, Herndon, VA, USA, November 6-9, 1990*, pages 511–518. IEEE Computer Society, 1990.

[38] Zhiyi Huang, Chengzheng Sun, and Abdul Sattar. Selective recomputation for handling side-effects in parallel logic programs. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Trach on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings*, volume 1292 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 1997.

[39] Matthew M. Huntbach and Graem A. Ringwood. Programming in concurrent logic languages. *IEEE Softw.*, 12(6):71–81, 1995.

[40] M. Kai and H. Kasahara. An efficient or-parallel processing scheme of prolog: Hierarchical pincers attack search. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing. Conference Proceedings*, pages 677–680, 1991.

[41] Laxmikant V. Kalé and Balkrishna Ramkumar. Implementation of a parallel prolog interpreter on multiprocessors. In V. K. Prasanna Kumar, editor, *The Fifth International Parallel Processing Symposium, Proceedings, Anaheim, California, USA, April 30 - May 2, 1991*, pages 543–548. IEEE Computer Society, 1991.

[42] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. *EBSE Technical Report*, 2007.

[43] Peyyun Peggy Li and Alain J. Martin. The sync model: A parallel execution method for logic programming. In *Proceedings of the 1986 Symposium on Logic Programming, Salt Lake City, Utah, USA, September 22-25, 1986*, pages 223–234. IEEE-CS, 1986.

[44] P. Lopez, M. Hermenegildo, and S. Debray. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation*, 21(4):715–734, 1996.

[45] Jorge Luis, Patrícia Vargas Mangan, Inês Dutra, and Claudio Fernando Resin. Granlog: An integrated granularity analysis model for parallel logic programming. In *Proceedings of the Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Programming Languages*, pages 74–88, 10 2000.

[46] Ewing L. Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross A. Overbeek, Rick Stevens, David H. D. Warren, Alan Calderwood, Péter Szeredi, and Seif Haridi. The aurora or-parallel prolog system. *New Gener. Comput.*, 7(2-3):243–271, 1990.

[47] E. Morel, J. Briat, J. Chassin De Kergommeaux, and C. Geyer. Side-effects in plosys or-parallel prolog on distributed memory machines. In *In JICSLP'96 Post-Conference Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1996.

[48] K.W. NG. The competition model-a comparison. In *IEEE TENCON'90: 1990 IEEE Region 10 Conference on Computer and Communication Systems. Conference Proceedings*, pages 146–149 vol.1, 1990.

[49] Mohamed Nour and Nadia H. Hegazi. Exploiting the sources of parallelism in logic programs. In *Second International Symposium on Autonomous Decentralized Systems, ISADS 1995, Phoenix, Arizona, USA, April 25-27, 1995*, pages 48–53. IEEE Computer Society, 1995.

[50] Enrico Pontelli and Gopal Gupta. Data parallel logic programming in &ace. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing, SPDP 1995, San Antonio, Texas , USA, October 25-28, 1995*, pages 424–431. IEEE, 1995.

[51] Enrico Pontelli and Gopal Gupta. On the duality between or-parallelism and and-parallelism in logic programming. In Seif Haridi, Khayri A. M. Ali, and Peter Magnusson, editors, *Euro-Par '95 Parallel Processing, First International Euro-Par Conference, Stockholm, Sweden, August 29-31, 1995, Proceedings*, volume 966 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 1995.

[52] Enrico Pontelli and Gopal Gupta. Nested parallel call optimization. In *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium, April 15-19, 1996, Honolulu, Hawaii, USA*, pages 225–229. IEEE Computer Society, 1996.

[53] Enrico Pontelli and Gopal Gupta. Parallel symbolic computation in ACE. *Ann. Math. Artif. Intell.*, 21(2-4):359–395, 1997.

[54] Enrico Pontelli and Gopal Gupta. Efficient backtracking in and-parallel implementations of non-deterministic languages. In *1998 International Conference on Parallel Processing (ICPP '98), 10-14 August 1998, Minneapolis, Minnesota, USA, Proceedings*, pages 338–345. IEEE Computer Society, 1998.

[55] Enrico Pontelli and Gopal Gupta. Backtracking in independent and-parallel implementations of logic programming languages. *IEEE Trans. Parallel Distributed Syst.*, 12(11):1169–1189, 2001.

[56] Enrico Pontelli, Gopal Gupta, and Manuel V. Hermenegildo. &ace: a high-performance parallel prolog system. In *Proceedings of IPPS '95, The 9th International Parallel Processing Symposium, April 25-28, 1995, Santa Barbara, California, USA*, pages 564–571. IEEE Computer Society, 1995.

[57] Balkrishna Ramkumar. Distributed last call optimization for portable parallel logic programming. *LOPLAS*, 1(3):266–283, 1992.

[58] Balkrishna Ramkumar and Laxmikant V. Kalé. A join algorithm for combining AND parallel solutions in AND/OR parallel systems. *Int. J. Parallel Program.*, 21(1):67–107, 1992.

[59] Balkrishna Ramkumar and Laxmikant V. Kalé. Machine independent AND and OR parallel execution of logic programs: Part i-the binding environment. *IEEE Trans. Parallel Distributed Syst.*, 5(2):170–180, 1994.

[60] Balkrishna Ramkumar and Laxmikant V. Kalé. Machine independent AND and OR parallel execution of logic programs: Part ii-compiled execution. *IEEE Trans. Parallel Distributed Syst.*, 5(2):181–192, 1994.

[61] Desh Ranjan, Enrico Pontelli, and Gopal Gupta. On the complexity of or-parallelism. *New Gener. Comput.*, 17(3):285–307, 1999.

[62] Ricardo Rocha, Fernando M. A. Silva, and Vítor Santos Costa. Yapor: an or-parallel prolog system based on environment copying. In Pedro Barahona and José Júlio Alferes, editors, *Progress in Artificial Intelligence, 9th Portuguese Conference on Artificial Intelligence, EPIA '99, Évora, Portugal, September 21-24, 1999, Proceedings*, volume 1695 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 1999.

[63] Ricardo Rocha, Fernando M. A. Silva, and Vítor Santos Costa. On a tabling engine that can exploit or-parallelism. In Philippe Codognet, editor, *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2237 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2001.

[64] Ricardo Rocha, Fernando M. A. Silva, and Vítor Santos Costa. Achieving scalability in parallel tabled logic programs. In *16th International Parallel and Distributed Processing Symposium (IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, 2002.

[65] Ricardo Rocha, Fernando M. A. Silva, and Vítor Santos Costa. Speculative computations in or-parallel tabled logic programs. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2004.

[66] Ricardo Rocha, Fernando M. A. Silva, and Vítor Santos Costa. On applying or-parallelism and tabling to logic programs. *Theory Pract. Log. Program.*, 5(1-2):161–205, 2005.

[67] Ricardo Rocha, Fernando M. A. Silva, and Rolando Martins. Yapdss: An or-parallel prolog system for scalable beowulf clusters. In Fernando Moura-Pires and Salvador Abreu, editors, *Progress in Artificial Intelligence, 11th Protuguese Conference on Artificial Intelligence, EPIA 2003, Beja, Portugal, December 4-7, 2003, Proceedings*, volume 2902 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2003.

[68] J.J. Ruz, F. Saenz, and L. Araujo. A transputer-based architecture to exploit the restricted-and-parallelism of prolog. In *[1991 Proceedings] 6th Mediterranean Electrotechnical Conference*, pages 1081–1084, 1991.

[69] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989.

[70] Kish Shen and Manuel V. Hermenegildo. Flexible scheduling for non-deterministic, and-parallel execution of logic programs. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume II*, volume 1124 of *Lecture Notes in Computer Science*, pages 635–639. Springer, 1996.

[71] Kish Shen and Manuel V. Hermenegildo. High-level characteristics of or- and independent and-parallelism in prolog. *Int. J. Parallel Program.*, 24(5):433–478, 1996.

[72] Huang Zhiyi; Hu Shouren;. Detection of and-parallelism in logic programs. *Journal of Computer Science and Technology*, 5(4), 1990.

[73] Marcio G. Silva, Inês de Castro Dutra, Ricardo Bianchini, and Vítor Santos Costa. The influence of architectural parameters on the performance of parallel logic programming systems. In Gopal Gupta, editor, *Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings*, volume 1551 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 1999.

[74] Jie Tao and Jiubin Ju. A task scheduling algorithm for parallel logic programming systems. In *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, volume 1, pages 266–271 vol.1, 2000.

[75] Evan Tick. The deevolution of concurrent logic programming languages. *J. Log. Program.*, 23(2):89–123, 1995.

[76] Eneia Todoran and Nikolaos Papaspyrou. Continuations for parallel logic programming. In Maurizio Gabbrielli and Frank Pfenning, editors, *Proceedings of the 2nd international ACM SIGPLAN conference on on Principles and practice of declarative programming, Montreal, Canada, September 20-23, 2000*, pages 257–267. ACM, 2000.

[77] Rajiv Trehan and Paul F. Wilk. Issues of non-determinism in *PROLOG* and the committed choice non-deterministic logic languages. *Artif. Intell. Rev.*, 4(3):211–232, 1990.

[78] MP Van Lohuizen. Parallel logic programming techniques. Technical report, Citeseer, 1998.

[79] Rui Vieira, Ricardo Rocha, and Fernando M. A. Silva. Or-parallel prolog execution on multicores based on stack splitting. In Umut A. Acar and Vítor Santos Costa, editors, *Proceedings of the POPL 2012 Workshop on Declarative Aspects of Multicore Programming, DAMP 2012, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 1–10. ACM, 2012.

[80] Karen Villaverde, Enrico Pontelli, Hai-Feng Guo, and Gopal Gupta. PALS: an or-parallel implementation of prolog on beowulf architectures. In Philippe Codognet, editor, *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2237 of *Lecture Notes in Computer Science*, pages 27–42. Springer, 2001.

[81] Ioannis P. Vlahavas, Petros Kefalas, and Constantin Halatsis. Oasys: an AND/OR parallel logic programming system. *Parallel Comput.*, 25(3):321–336, 1999.

[82] Dong Wang, H. Kobayashi, and T. Nakamura. Design and performance measurements of an execution model for the parallel processing of prolog programs. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, volume 2, pages 650–658 vol.2, 1995.

[83] David H. D. Warren. Or-parallel execution models of prolog. In Hartmut Ehrig, Robert A. Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT'87: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Pisa, Italy, March 23-27, 1987, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Functional and Logic Programming and Specifications (CFLP)*, volume 250 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 1987.

[84] Yao Wu, Enrico Pontelli, and Desh Ranjan. On the complexity of dependent and-parallelism in logic programming. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 361–376. Springer, 2003.

[85] Yao Wu, Enrico Pontelli, and Desh Ranjan. Computational issues in exploiting dependent and-parallelism in logic programming: Leftness detection in dynamic search

trees. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2005.

[86] George Xirogiannis. Granularity control for distributed execution of logic programs. In *Proceedings of the 18th International Conference on Distributed Computing Systems, Amsterdam, The Netherlands, May 26-29, 1998*, pages 230–237. IEEE Computer Society, 1998.

[87] George Xirogiannis and Hamish Taylor. A dynamic task distribution and engine allocation strategy for distributed execution of logic programs. In Peter M. A. Sloot, Marian Bubak, and Louis O. Hertzberger, editors, *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1998, Amsterdam, The Netherlands, April 21-23, 1998, Proceedings*, volume 1401 of *Lecture Notes in Computer Science*, pages 294–304. Springer, 1998.

[88] Huo yan Chen. The improvement of the and/or process model for parallel interpretation of logic programs. In *Proceedings of the 1988 IEEE International Conference on Systems, Man, and Cybernetics*, volume 1, pages 159–162, 1988.

[89] Kang Zhang. Using dataflow principle to exploit restricted and-parallelism in logic programs. In *Proceedings of TENCON '93. IEEE Region 10 International Conference on Computers, Communications and Automation*, volume 1, pages 150–153 vol.1, 1993.

[90] Kang Zhang. A review of exploitation of and-parallelism and combined and/or-parallelism in logic programs. *ACM SIGPLAN Notices*, 29(2):25–32, 1994.

[91] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. Computing executable slices for concurrent logic programs. In *2nd Asia-Pacific Conference on Quality Software (APAQS 2001), 10-11 December 2001, Hong Kong, China, Proceedings*, pages 13–22. IEEE Computer Society, 2001.

[92] *Literal Dependence Net and Its Use in Concurrent Logic Programming Environment*, 1994.

[93] Yuhua Zheng, Honglei Tu, and Li Xie. And/or parallel execution of logic programs: Exploiting dependent and-parallelism. *ACM SIGPLAN Notices*, 28(5):19–28, 1993.