A Parallel Logic Programming Approach to Job Shop Scheduling Constraint Satisfaction Problems*

Jorge Puente, Ramiro Varela, Camino R. Vela, Cesar Alonso**
Centro de Inteligencia Artificial. Universidad de Oviedo en Gijón
Campus de Viesques. E-33271 Gijón. Spain.
Tel. +34-8-5182032. FAX +34-8-5182125.

Abstract

In this paper we put together a Parallel Logic Programming schema and a heuristic search strategy in order to solve constraint satisfaction problems. The idea is to introduce heuristic information in the construction of logic programs in order to improve the performance. The experimental results show that logic programs can be designed that exhibit parallelism, and that the use of heuristic information translates into speedup in obtaining answers.

Keywords: Parallel Logic Programming, Constraint Satisfaction Problems, Job Shop Scheduling, Heuristics.

1 Introduction

The aim of this paper is to propose a strategy for problem solving that combines parallelism and heuristic search. Parallelism will be exploited by means of the *RFD/RPS* model that we proposed [Var95b] for evaluating logic programs in parallel. As other models described in the literature [Con83, War90, Kal91, Pon95, She96], this model can exploit three of the most important sources of parallelism that the language of logic can express, namely *OR parallelism*, *independent AND parallelism (IAP)* and *producer/consumer parallelism*. On the other hand, we choose the *Job Shop Scheduling (JSS)* constraint satisfaction problem to deal with in this work. This is a NP-hard problem that has given rise to a high research effort, so a large number of solutions have been proposed. Among these solutions, we can found the application of almost every artificial intelligence technique, for instance constraint logic programming [Hen92], neural nets [Ado90], machine learning [Zwe92], genetic algorithms [Cor97] and heuristic search [Sad96]. Maybe, the last two ones being the most frequently used. Within the scope of this work, we consider the variable and value ordering heuristics proposed in [Sad96] for *JSS* problems. These heuristics will be used to guide the construction of logic programs for solving *JSS* problems, in order for these programs to be efficiently evaluated in parallel. As we will see, these programs exhibit all of the three types of parallelism above commented, and can be improved with the assistance of heuristics.

The remainder of the paper is organized as follows: in Section 2 we introduce the *RFD/RPS* process model for parallel evaluation of logic programs. Section 3 defines the *JSS* problem as it is considered in this paper. Section 4 describes the logic programming approach that we propose for *JSS* problems. Section 5 summarises the variable and value ordering heuristics proposed in [Sad96] and shows how they can be used in the construction of logic

* This work has been partially supported by the FICYT of the Principado de Asturias under Project PB-TIC-9703

** E-mail: {puente,ramiro,camino,calonso}@aic.uniovi.es

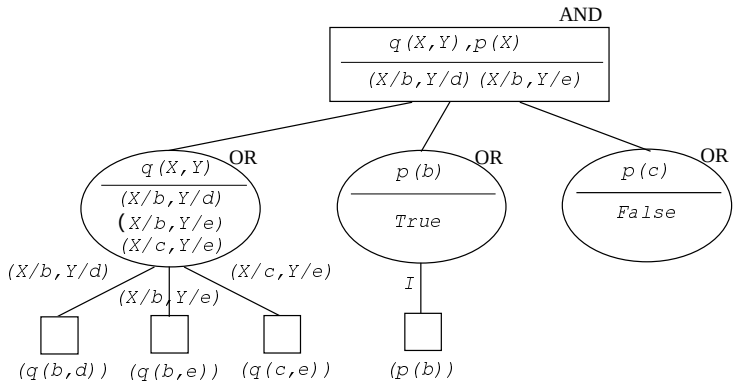


Figure 1

programs. Section 6 outlines the characteristics of the simulator tool that we have used for obtaining experimental results. Section 7 presents some experimental results that clarify the performance of our approach. Finally, in Section 8 we present the conclusions of the work.

2 The Process Model

We have proposed a new model for exploiting *IAP* with *OR* parallelism in which computations are represented by *AND/OR* trees. Figure 1 depicts the tree expanded by our model when the query `q(X,Y),p(X)` is evaluated with respect to the program given by the set of facts `p(a),p(b),q(c,e),q(b,e),q(b,d)`. The literals within a query are evaluated according to a partial ordering; we assume that for the former query the ordering is first `q(X,Y)` and then `p(X)`. Therefore, one *AND* process is created first for solving the query. This process generates one *OR* process for solving `q(X,Y)` which in its turn generates, in parallel, three *AND* processes associated respectively with the three clauses of the program that solve the literal. In this case, these processes produce the identity solution. This solution, previously composed with the label of the edge, gives a solution to the father *OR* node. This node sends the three solutions to the root *AND* node, which creates two parallel *OR* processes (one from the two solutions with the variable `X` instantiated to the constant `b`, and the other one from the third answer with the variable instantiated to the constant `c`) to compute solutions of `p(X)` compatible with those computed for `q(X,Y)`. That is, two processes are generated to solve instances of `p(X)`: `p(b)` and `p(c)`. The first one returns the solution `True`, whereas the second finishes with negative result. Finally at the root node we have two answers to the query.

The main feature of our proposed model is that the duplication of processes is avoided, in contrast to other models, such as those proposed in [Kal91, Gup92], which would generate

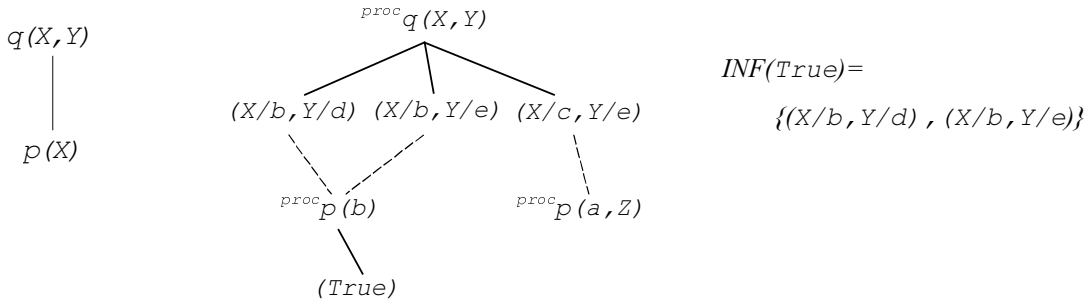


Figure 2

two processes for solving `p(b)` in the former example, one from each solution of the literal `q(X,Y)` with the variable `X` instantiated to the value `b`. In order to achieve that in an efficient way, we have developed a strategy for partial solutions management. In the example at hand, this strategy permits us to link two solutions of `q(X,Y)`, `(X/b,Y/e)` and `(X/b,Y/d)`, to the process generated for solving `p(b)`, thus avoiding its duplication. This is achieved by means of two data structures: a *Data Flow Lattice (DFL)* to codify the partial order among the literals of a query in order to exploit *IAP*, and a *Processes and Solutions Net (PSN)* to represent the partial solutions as well as the *OR* process identifiers that computed these partial solutions. Figure 2 shows the *DFL* (a) and the *PSN* (b) generated from the query at the root node of the *AND/OR* tree of Figure 1. Moreover, Figure 2c shows the application of the *inference function, INF*, to the node `True` of the *PSN*. This node indicates that the *OR* process labelled by $^{proc}p(b)$ finished with a positive answer. The function *INF* is in charge of computing solutions to the query by joining the partial solutions spread over the *PSN.*

Our model can exploit *producer/consumer (or consumer instance)* parallelism, as is shown in [Var96b]; this is a secondary source of parallelism that has to do with both *AND* and *OR* in the presence of non determinism. Given two literals with common variables, it consists of starting the evaluation of one instance of the second literal *(consumer)* to compute compatible solutions with one solution of the first *(producer)* as soon as this solution appears. In our example, if producer/consumer parallelism is exploited, as the literal `q(X,Y)` has multiple solutions, we can start exploring $p(x_i)$ as soon as we have a new value (X/x_i) from a solution to `q(X,Y)`. Therefore, as we can see in Figure 2b, the process $^{proc}p(b)$ is created as soon as the first solution to the literal `q(X,Y)` with the variable `X` instantiated to the value `b` is obtained. When the second one appears, it has only to be linked to the process identifier $^{proc}p(b)$. Producer/consumer parallelism is interesting in non deterministic programs. Furthermore, in the presence of infinite relations, it maintains the completeness of the system. As pointed out in [Kal91], exploitation of this parallelism was the point of departure for the *REDUCE-OR Process Model (RPM).* Exploitation of this class of parallelism was also one of our starting goals.

For further details about the process model, in particular about the *DFL* and the *PSN*, we refer to the interested reader to other works as [Var94, 95a, b, 96a, b].

3 The Job Shop Scheduling Constraint Satisfaction Problem

In this section we introduce the *JSS* problem we are considering along the paper. The job shop requires scheduling a set of jobs $\{J_1,...,J_n\}$ on a set of physical resources $\{R_1,...,R_q\}$. Each job *Ji* consists of a set of tasks $\{t_{i1},...,t_{imi}\}$ to be sequentially scheduled, and each task has a single resource requirement. We assume that there are a release date of all jobs and a due date between which all the tasks have to be performed. Each task has a fixed duration du_{ij} and a start time st_{ij} whose value has to be selected. The domain of possible start times of the tasks is initially constrained by the release and due dates.

Therefore, there are two non-unary constraints of the problem: *precedence constraints* and *capacity constraints*. Precedence constraints defined by the sequential routings of the tasks within a job translate into linear inequalities of the type: $st_{il} + du_{il} \leq st_{ik}$ (i.e. st_{il} before st_{ik}). Capacity constraints that restrict the use of each resource to only one task at a time translate into disjunctive constraints of the form: $st_{il} + du_{il} \leq st_{jk} \vee st_{jk} + du_{jk} \leq st_{il}$ (i.e. two tasks that use the same resource can not overlap).

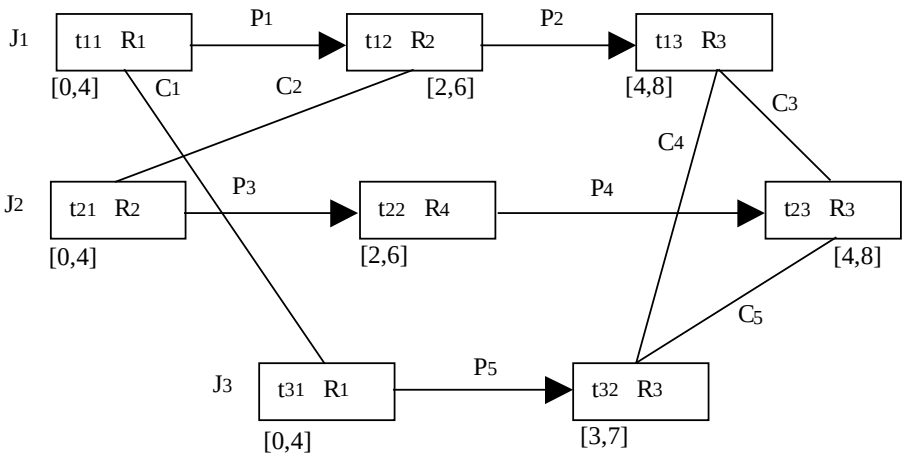


Figure 3

The objective is to come up with a feasible solution as fast as possible, a solution being a vector of start times, one for each task, such that starting at these times all the tasks end without exceeding the due date and all the constraints are satisfied.

None of the simplifying assumptions are required by the approach that will be discussed: jobs usually have different release and due dates, tasks within a job can have different duration, several resource requirements, and several alternatives for each of these requirements.

Figure 3 depicts an example with three jobs $\{J_1,J_2,J_3\}$ and four physical resources $\{R_1,R_2,R_3,R_4\}$. It is assumed that the tasks of the first two jobs have duration of two time units, whereas the tasks of the third one have duration of three time units. The release time is 0 and the due date is 10. Label *Pi* represents a precedence constraint and label *Cj* represents a capacity constraint. Start time values constrained by the release and due dates and the duration time of tasks are represented as intervals. For instance [0,4] represents all start times between time 0 and time 4, as allowed by the time granularity, namely {0,1,2,3,4}. Table 1 shows one of the solutions of the problem instance depicted if Figure 3.

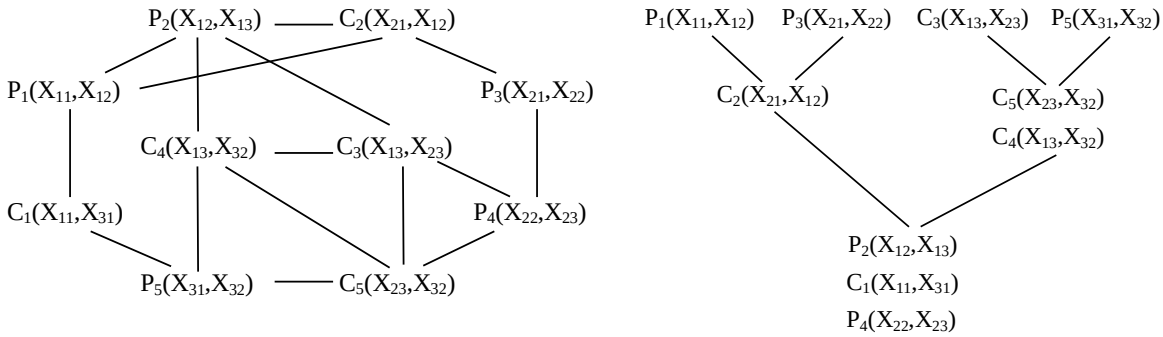
X_{11}	X_{12}	X_{13}	X_{21}	X_{22}	X_{23}	X_{31}	X_{32}
3	5	8	0	2	6	0	3

Table 1

4 The Logic Programming Approach to *JSS* Problems

In this section we show how the job shop scheduling problem can be represented in the language of logic by means of a set of clauses, and how these clauses can be determined in order to exploit parallelism. Firstly, for each task we define a single literal having a solution for each of the possible start times. So, for instance, the task t_{11} will be defined by the ground instances $t_{11}(0)$, $t_{11}(1)$, $t_{11}(2)$, $t_{11}(3)$, $t_{11}(4)$. Moreover, each one of the constraints will be defined by means of binary relations: one binary relation for each of the precedence constraints and two binary relations for each of the capacity constraints, for instance

$$P_1(X_{11},X_{12}) :- t_{11}(X_{11}),t_{12}(X_{12}),X_{12} \geq X_{11}+du_{11}.$$



a) b)

Figure 4

$C_1(X_{11},X_{31})$:- $t_{11}(X_{11}),t_{31}(X_{31}),X_{31} \geq X_{11}+du_{11}$.

$C_1(X_{11},X_{31})$:- $t_{11}(X_{11}),t_{31}(X_{31}),X_{11} \geq X_{31}+du_{31}$.

It is clear that an instantiation of the whole set of variables appearing in constraints making true each of them represents a solution of the problem. It is also clear that, in general, there are a big number of variables shared by two or more literals, and that every constraint literal has a lot of solutions, so it makes sense to organize the evaluation of the constraint literals under *IAP*. In order to do that, in this work we propose a strategy that consists of two steps: firstly, an *independent constraint tree* is computed from the *constraint dependency graph*; and then, from the independent constraint tree, a logic program that can be annotated for evaluation under *IAP* is determined. An independent constraint tree is a tree representing a partial ordering for the evaluation of the whole set of constraint literals under *IAP*; and the constraint dependency graph is an undirected graph representing the variable dependencies among the constraint literals. Figure 4a depicts the constraint dependency graph for the problem of Figure 3. The independent constraint tree is not unique for a given constraint dependency graph; Figure 4b shows an independent constraint tree determined from the graph of Figure 4a.

In order to compute an independent constraint tree from a constraint dependency graph, we

algorithm independent constraint tree (*G*: constraints dependency graph);

calculate *C={C1,...,Cn}* such that every *Ci* contains only one constraint of *G* that is independent of every constraint contained in the remainder *Cj*s; each of the *Ci*s of *C* is a leaf of the independent constraint tree;

calculate *CG* as the set of constraints of *G* not belonging to any *Ci*;

while *C* is not unitary **do**

for each *Ci* in *C* determine *Ci'* to contain the constraints of *CG* adjacent only to some of the constraints in *Ci*, if *Ci'* is not empty, update *CG* by removing the constraints in *Ci'*, update *C* also by removing *Ci* and inserting *Ci'*, *Ci'* is the label of the father node of *Ci*;

*{now in C there are not any constrain adjacent only to constraints of one of the Ci*s*}*

select *Ci* and *Cj* of *C* and determine *Ck* to contain the constraints of *CG* that are adjacent only to constraints in *Ci* and *Cj* and that are not adjacent to any constraint of the remainder *Cl*s of *C*; *{Ck might be empty}*

update *CG* by removing the constraints of *Ck*, remove *Ci* and *Cj* of *C*, insert *Ck* in the independent constraint tree as the father of the nodes *Ci* and *Cj*

endwhile

end.

Figure 5

propose the algorithm of Figure 5. This algorithm first tries to obtaining a set as big as possible of independent constraints, these constraints are the leaves of the tree. Then, a search for constraints dependent of only one or two computed nodes is repeated in order to determining the remaining nodes of the tree. As we can observe, there are several non deterministic actions that have to be solved by means of heuristics.

Now, from the independent constraints tree a logic program is determined. This program is not unique for a given tree, but distinct programs can be derived with different sizes in the clauses, so giving rise to different granularity levels of the processes generated during program evaluation. For instance, from the tree depicted in Figure 4b at least the following two programs can be determined (here the facts and the constraint relations are not represented).

Program_1

```
p1p3c2(X11,X12,X21,X22):-p1(X11,X12), p3(X21,X22), c2(X21,X12).
c3p5c5(X13,X23,X31,X32):-c3(X13,X23), p5(X31,X32), c5(X32,X23).
c3p5c5c4(X13,X23,X31,X32):– c3p5c5(X13,X23,X31,X32), c4(X13,X32).
p1p3c2c3p5c5c4p2(X11,X12,X21,X22,X13,X23,X31,X32):-
      p1p3c2(X11,X12,X21,X22), c3p5c5c4(X13,X23,X31,X32), p2(X12,X13).
p1p3c2c3p5c5c4p2c1(X11,X12,X21,X22,X13,X23,X31,X32):-
      p1p3c2c3p5c5c4p2(X11,X12,X21,X22,X13,X23,X31,X32), c1(X11,X31).
all(X11,X12,X13,X21,X22,X23,X31,X32):-
      p1p3c2c3p5c5c4p2c1(X11,X12,X21,X22,X13,X23,X31,X32), p4(X22,X23).
```

Program_2

```
p1p3c2(X11,X12,X21,X22):-p1(X11,X12), p3(X21,X22), c2(X21,X12).
c4c5(X23,X32,X13):-c4(X13,X32),c5(X32,X23).
c3p5c4c5(X13,X23,X31,X32):-c3(X13,X23), p5(X31,X32), c4c5(X23,X32,X13).
p2c1p4(X12,X13,X11,X31,X22,X23):-p2(X12,X13),c1(X11,X31), p4(X22,X23).
all(X11,X12,X13,X21,X22,X23,X31,X32):-
      p1p3c2(X11,X12,X21,X22), c3p5c4c5(X13,X23,X31,X32),
      p2c1p4(X12,X13,X11,X31,X22,X23).
```

5 Variable and Value Ordering Heuristics for the *JSS* Problem

As we have pointed out in the introduction, one of the original contributions of this work will be the utilization of heuristic information in the construction stage of logic programs for solving *JSS* problems. Our purpose is to incorporate the variable and value ordering heuristics proposed by Norman Sadeh and Mark S. Fox in [Sad96]. These heuristics are based on a probabilistic model of the search space. A probabilistic framework is introduced that accounts for the chance that a given value will be assigned to a variable and the chances that values assigned to different variables conflict with each other.

The heuristics are evaluated from the profile demands of the tasks for the resources. In particular the *individual demand* and the *aggregate demand* values are considered. The individual demand $D_{ij}(R_p,T)$ of a task t_{ij} for a resource R_p at time T is simply computed by adding the probabilities $\sigma_{ij}(\tau)$ of the resource R_p is demanded by the task t_{ij} at some time

Interv.	0	1	2	3	4	5	6	7	8	9	10
$D_{11}(R_1,T)$	0.2	0.4	0.4	0.4	0.4	0.2					
$D_{31}(R_1,T)$	0.2	0.4	0.6	0.6	0.4	0.2	0.2				
$D^{aggr}(R_1,T)$	0.4	0.8	1	1	0.8	0.4	0.2				
$D_{12}(R_2,T)$			0.2	0.4	0.4	0.4	0.4	0.2			
$D_{21}(R_2,T)$	0.2	0.4	0.4	0.4	0.4	0.2					
$D^{aggr}(R_2,T)$	0.2	0.4	0.6	0.8	0.8	0.6	0.4	0.2			
$D_{13}(R_3,T)$					0.2	0.4	0.4	0.4	0.4	0.2	
$D_{23}(R_3,T)$					0.2	0.4	0.4	0.4	0.4	0.2	
$D_{32}(R_3,T)$				0.2	0.4	0.6	0.6	0.6	0.4	0.2	
$D^{aggr}(R_3,T)$				0.2	0.8	1.4	1.4	1.4	1.2	0.6	
$D_{22}(R_4,T)$			0.2	0.4	0.4	0.4	0.2				
$D^{aggr}(R_4,T)$			0.2	0.4	0.4	0.4	0.2				

Table 2

within the interval $[T-du_{ij}+1,T]$. The individual demand is an estimation of the reliance of a task on the availability of a resource. Consider, for example, the initial search state depicted in Figure 3. As the task t_{12} has five possible start times or reservations, and assuming that there is no reason to believe that one reservation is more likely to be selected than another, each reservation is assigned an equal probability to be selected, in this case 1/5. Given that the task t_{12} has duration of 2 time units, this task will demand to the resource R_2 at time 4 if its start time is either 3 or 4. So, the individual demand of the task t_{12} for resource R_2 at time 4 is estimated as $D_{12}(R_2,4)=\sigma_{12}(3)+\sigma_{12}(4)=2/5$. On the other hand, the aggregate demand $D^{aggr}(R,\tau)$ for a resource is obtained by adding the individual demands of all tasks over the time. Table 2 shows the individual demands of all ten tasks of the problem, as well as the aggregate demands for all four resources.

From the aggregate demand of a resource a contention peak is identified. This is an interval of the aggregate demand of duration equal to the average duration of all the tasks with the highest demand. Table 2 shows the contention peaks of all the four resources. Then, the task with the largest contribution to the contention peak of a resource is determined as the most critical and therefore it is selected first for reservation. This is the heuristic of variable ordering referred in [Sad96] as *ORR (Operation Resource Reliance)*. This heuristic can be introduced in the construction of the independent constraint tree by inserting as leaves of the tree those constraints that involve tasks with large contribution to the corresponding contention peaks.

On the other hand, the value ordering heuristic proposed in [Sad96] is also computed from the profile demands for the resources. Given a task t_{ij} that demands the resource R_p, the heuristic consists of estimating the *survivability of the reservations.* The survivability of a reservation $\langle st_{ij}=T\rangle$ is the probability that the reservation will not conflict with the resource requirements of other tasks, that is, the probability that none of the other tasks require the resource during the interval $[T,T+du_{ij}-1]$. When the task demands are for only one resource, this probability can be estimated as [Sad96]

$$\left(1-\frac{\mathrm{AVG}\left(\mathrm{D}^{\mathrm{aggr}}\left(\mathrm{R}_{\mathrm{p}},\tau\right)-\mathrm{D}_{\mathrm{ij}}\left(\mathrm{R}_{\mathrm{p}},\tau\right)\right)}{\mathrm{AVG}\left(\mathrm{n}_{\mathrm{p}}(\tau)-1\right)}\right)^{\mathrm{AVG}\left(\mathrm{n}_{\mathrm{p}}(\tau)-1\right)*\mathrm{du}_{\mathrm{ij}}*\left(\mathrm{AVG}(\mathrm{du})^{-1}\right)},$$

where *du* stands for the average duration of the tasks, $n_p(\tau)$ is the number of tasks that can demand the resource R_p at time τ and $AVG(f(\tau))$ represents the average value of function $f(\tau)$ in the interval $[T,T+du_{ij}-1]$. Table 3 shows the survivability of all the reservations possible for all ten tasks of the problem.

As it looks clear, the value ordering heuristic consist of trying first the reservations with large values of its survivability. This heuristic can be easily introduced in the construction of the logic programs for the JSS problem by means of the ordering of declaration of ground literals that define each of the tasks: literals with large values are declared first. In particular, we will use the values of the survivability of the reservations in the initial state of the search process. So, during the program evaluation, reservations with high survivability will be used first under the assumption that they are more likely to be present within a solution of the whole problem.

For a depth study of these heuristics, as well as for further refinements, we refer to the interested reader to [Sad96].

6 Heuristic Programs and the Simulator Tool

In order to evaluate logic programs and to study their performance, we have developed a simulator of our interpretation model that emulates the evolution of the set of processes generated on an arbitrary number of processors. The scheduling policy of the processes is based on priorities that are proportional to the waiting time of the processes in the ready to run queue. After evaluation of a query with respect to a logic program, the simulator permits to lay out the process tree and the Gantt chart of the processes generated for solving the query. This information permits studying the amount of parallelism that is exploited during the evaluation of the programs, and so it allows us to evaluate the quality of the logic programs from the point of view of its parallel evaluation.

In order to simplify the switching context task, the atomic operations of the processes are assumed to be “not too small”, and the quantum of time assigned to a process for execution is the time of its next atomic operation. These atomic operations, as well as the structure of the processes, *AND* and *OR*, are represented in Figures 6a and 6b respectively. The procedure

```
AND Process (DFL)
 generate an OR process for each
 literal with no predecessors in
 the DFL;(*)
 while new answers can arrive from
 the descendants OR processes do
  wait-for-answer;
  process-answer;(*)
 endwhile;
 send to the parent OR process the
 answer end-of-process (*)
end.
```

a)

```
OR Process (literal)
 for each clause of the program which
 conclussion unifies with the literal
 generate an AND process to solve its
 body;(*)
 while new answers can arrive from the
 descendants AND processes do
  wait-for-answer;
  process-answer;(*)
 endwhile;
 send to the parent AND process the
 answer end-of-process (*)
end.
```

b)

Figure 6. (*) atomic operations

Interv.	0	1	2	3	4	5	6	7	8	9	10
t_{11}	0.73	0.54	0.44	0.54							
t_{12}			0.63	0.63	0.73	0.95	1				
t_{13}					0.41	0.3	0.3	0.35	0.53		
t_{21}	1	0.95	0.73	0.63	0.63						
t_{22}			1	1	1	1	1				
t_{23}					0.41	0.3	0.3	0.35	0.53		
t_{31}	0.59	0.51	0.51	0.59	0.73						
t_{32}				0.71	0.35	0.26	0.26	0.35			

Table 3

`process-answer` of the *AND* processes consist of taking an answer from its input queue and make all the work to process it. That is, joining the answer with the answers to the previous literals of the query, generating all the necessary processes for solving the successor literals of the query, and finally sending the new solutions to the query to the father *OR* process. In the case of the *OR* processes, the `process-answer` procedure is very simple, it only consists of sending each answer of its input queue to the father *AND* process. So we could consider the processing of several answers as an atomic operation, in order to assign a similar quantum of processor time to both, *AND* and *OR* processes.

The current release of the simulator is built on KappaPC 2.3 object oriented environment, and so it has some limitations mainly due to the limited number of active instances that the tool can manage at a given time. As a consequence, the simulator often spends an unacceptable amount of time when evaluates very big programs. So, in order to reduce the number of instances generated during a simulation session, we introduce the following transformation in logic programs for solving *JSS* problems. Instead of defining each of the tasks by means of a relation with as many ground instances as possible reservations, we define a relation for each of the constraints having one ground instance for each compatible join of solutions of the pair of tasks involved. Now, the ordering among the constraint ground instances is made in base to the product of survivability values of tasks ground instances involved. For example, as we had the clause $P_1(X_{11},X_{12})$:-$t_{11}(X_{11}),t_{12}(X_{12}),X_{12} \geq X_{11}+du_{11}$, and solutions $t_{11}(1)$ and $t_{21}(3)$ are compatibles with each other, having these reservations probabilities 0.73 and 0.63 respectively as shown in Table 3, we declare the ground instance $P_1(1,3)$ with a probability 0.74*0.63=0.46. Table 4 shows the corresponding ground instances to all ten constraints, declared in the order established by their probability values.

As we can see in Table 4, a number of ground instances contributing to the solution of Table 1 arise next to the beginning of the relation, but some other do not. So it would be interesting to design a strategy for determining the independent constraint trees that include in the leaves those literals whose good instances arise close to the beginning of the relation. We have pointed out in section 5 that the variable ordering heuristic could be incorporated in our strategy by inserting as leaves of the tree those constraints with a large contribution to the contention peak of the aggregate demand. In the example, the constraint involving tasks with the largest contribution to the corresponding contention peaks is $P_5(X_{31},X_{32})$ as we can see in Table 2 (tasks t_{31} and t_{32}). Then, following the heuristic, we have to put the literal $P_5(X_{31},X_{32})$ as a leaf of the tree, as done in the tree of Figure 4b. In this case, this heuristic seems to work

well because the ground instance $P_5(0,3)$ that contributes to the solution of Table 1 arises at the first position of the relation as shown in Table 4.

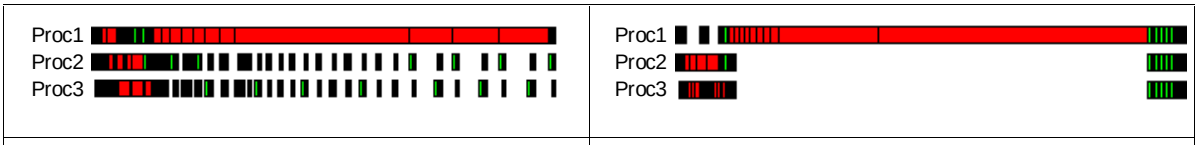
7 Experimental Results

In this section we include some experimental results showing, on one hand, the amount of parallelism that can be exploited in solving *JSS* problems; and on the other hand, the speedup that the variable and value ordering heuristics produce in obtaining answers.

As we can see from the logic programs determined by solving the problem of Figure 1 (for example the Program_2 of Section 4 including the ground instances of Table 4), they exhibit all types of parallelism that our model can exploit; namely, OR parallelism, independent AND parallelism and producer/consumer parallelism. The importance of the first two ones was widely proclaimed in the literature, so, we start showing an example in order to make clear the improvement introduced by the producer/consumer parallelism. We consider the Program_2 but only four ground instances of each of the ten constraints of Table 4, including those instances that contribute to the solution of Table 1, declared at the beginning of the relation. Figure 7 depicts the Gantt charts of the processes generated to solving the query `all(X11,X12,X13,X21,X22,X23,X31,X32)`, when the producer/consumer parallelism is exploited (Figure 7a) and when it is not exploited (Figure 7b). As we can observe, in the first case there is a large displacement of many CPU intervals of the processes towards starting execution time. As a consequence, the time of answer is lower in the first case than in the second one.

$P_1(X_{11},X_{12})$	$P_2(X_{12},X_{13})$	$P_3(X_{21},X_{22})$	$P_4(X_{22},X_{23})$	$P_5(X_{31},X_{32})$
p1(0,6). 0.73	p2(6,8). 0.53	p3(0,2). 1.00	p4(2,8). 0.53	p5(0,3). 0.42
p1(0,5). 0.69	p2(5,8). 0.50	p3(0,3). 1.00	p4(3,8). 0.53	p5(4,7). 0.26
p1(1,6). 0.54	p2(4,8). 0.39	p3(0,4). 1.00	p4(4,8). 0.53	p5(0,7). 0.21
p1(4,6). 0.54	p2(3,8). 0.33	p3(0,5). 1.00	p4(5,8). 0.53	p5(3,7). 0.21
p1(0,4). 0.53	p2(2,8). 0.33	p3(0,6). 1.00	p4(6,8). 0.53	p5(0,4). 0.21
p1(1,5). 0.51	p2(5,7). 0.33	p3(1,3). 0.95	p4(2,4). 0.41	p5(2,7). 0.18
p1(0,2). 0.46	p2(2,4). 0.26	p3(1,4). 0.95	p4(3,7). 0.35	p5(1,4). 0.18
p1(0,3). 0.46	p2(4,7). 0.26	p3(1,5). 0.95	p4(5,7). 0.35	p5(1,7). 0.18
p1(3,6). 0.44	p2(2,7). 0.22	p3(1,6). 0.95	p4(2,7). 0.35	p5(3,6). 0.15
p1(2,6). 0.44	p2(3,7). 0.22	p3(2,4). 0.73	p4(4,7). 0.35	p5(0,5). 0.15
p1(2,5). 0.42	p2(4,6). 0.22	p3(2,5). 0.73	p4(3,5). 0.30	p5(0,6). 0.15
p1(3,5). 0.42	p2(2,5). 0.19	p3(2,6). 0.73	p4(3,6). 0.30	p5(1,6). 0.13
p1(1,4). 0.39	p2(2,6). 0.19	p3(3,5). 0.63	p4(2,5). 0.30	p5(1,5). 0.13
p1(1,3). 0.34	p2(3,5). 0.19	p3(3,6). 0.63	p4(2,6). 0.30	p5(2,5). 0.13
p1(2,4). 0.32	p2(3,6). 0.19	p3(4,6). 0.63	p4(4,6). 0.30	p5(2,6). 0.13
$C_1(X_{11},X_{31})$	$C_2(X_{21},X_{12})$	$C_3(X_{13},X_{23})$	$C_4(X_{13},X_{32})$	$C_5(X_{32},X_{23})$
c1(0,4). 0.53	c2(0,6). 1.00	c3(4,8). 0.22	c4(8,3). 0.38	c5(3,8). 0.38
c1(0,3). 0.43	c2(0,5). 0.95	c3(8,4). 0.22	c4(7,3). 0.25	c5(3,7). 0.25
c1(1,4). 0.39	c2(1,6). 0.95	c3(6,8). 0.16	c4(6,3). 0.21	c5(3,6). 0.21
c1(0,2). 0.37	c2(1,5). 0.90	c3(5,8). 0.16	c4(8,4). 0.19	c5(4,8). 0.19
c1(2,4). 0.32	c2(0,4). 0.73	c3(8,5). 0.16	c4(4,7). 0.14	c5(7,4). 0.14
c1(1,3). 0.32	c2(2,6). 0.73	c3(8,6). 0.16	c4(8,5). 0.14	c5(5,8). 0.14
c1(4,0). 0.32	c2(2,5). 0.69	c3(7,4). 0.14	c4(7,4). 0.12	c5(4,7). 0.12
c1(4,1). 0.28	c2(1,4). 0.69	c3(4,7). 0.14	c4(4,6). 0.11	c5(6,4). 0.11
c1(3,0). 0.26	c2(0,2). 0.63	c3(6,4). 0.12	c4(5,7). 0.11	c5(7,5). 0.11
	c2(0,3). 0.63	c3(4,6). 0.12		
	c2(3,6). 0.63	c3(7,5). 0.11		
	c2(4,6). 0.63	c3(5,7). 0.11		
	c2(1,3). 0.60			
	c2(3,5). 0.60			
	c2(2,4). 0.53			
	c2(4,2). 0.40			

Table 4



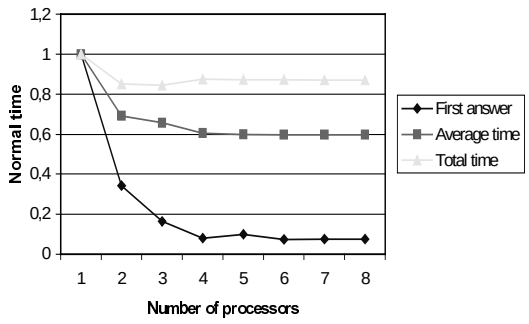
a) with producer/consumer parallelism. The time of answer is 855 units

b) without producer/consumer parallelism. The time of answer is 6244 units

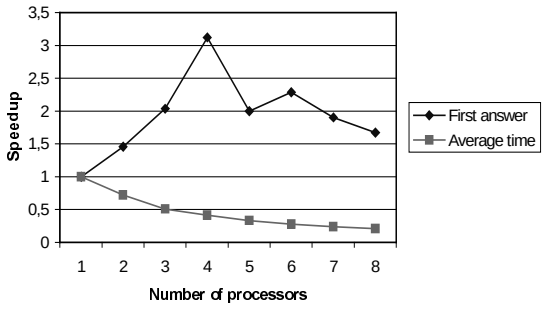
Figure 7. Gantt charts of two simulations for solving the same problem on three processors

Now, in order to clarify the importance of the ordering among the ground instances of the constraint literals, the same program is evaluated, but now moving the ground instances that contribute to the solution at the end of the respective relations. In this case, the time of the answer is 5519 units, which is larger than the time produced in the execution of Figure 7a, where the ground instances contributing to the solution were placed to the beginning of all ten relations. This result makes it clear the importance of a good ordering among the ground instances in order to obtain the answers as quick as possible. Now, we consider the whole program, that is, the Program_2 and all the ground instances of Table 4. In order to clarify the performance of the value ordering heuristic, we simulate the evaluation of the program, first keeping the ordering of Table 4 among the ground instances produced by the heuristic, and then with the inverse ordering. Table 5 shows the arrival time of all twelve answers, as well as the average time. As we can see, the ground instances ordering produced by the heuristics translates into an important speedup.

Finally, in Figure 8 we present some results showing the improvement on performance when the number of processors increases. Figure 8a shows the evolution of first answer time, average answer time and total execution time when the number of processors varies form 1 to 8. As we can observe, the time of the first answer, that is usually the most significant parameter, decreases quickly with the number of processors until a number of processors is reached, in this case 4, that exploits all the parallelism that the problem instance exhibit. On the other hand, Figure 8b shows the speedup obtained by increasing the number of processors. In any case, we consider the Program_2 with all ground instances ordered by the heuristic as shown in Table 4.



a)



b) speedup defined as $\left(\frac{time\_with\_1\_processor}{time\_with\_n\_processors}\right)/n$

Figure 8.

Answers to the query `all(X11,X12,X13,X21,X22,X23,X31,X32)`	Arrival time with the ordering produced by the heuristic	Arrival time with the inverse ordering
{ X11/4, X12/6, X13/8, X21/0, X22/2, X23/6, X31/0, X32/3 }	4542	279873
{ X11/4, X12/6, X13/8, X21/0, X22/3, X23/6, X31/0, X32/3 }	5358	232000
{ X11/3, X12/6, X13/8, X21/0, X22/2, X23/6, X31/0, X32/3 }	7998	263182
{ X11/3, X12/6, X13/8, X21/0, X22/3, X23/6, X31/0, X32/3 }	8775	218288
{ X11/3, X12/5, X13/8, X21/0, X22/2, X23/6, X31/0, X32/3 }	12922	253609
{ X11/3, X12/5, X13/8, X21/0, X22/3, X23/6, X31/0, X32/3 }	13871	210598
{ X11/4, X12/6, X13/8, X21/0, X22/4, X23/6, X31/0, X32/3 }	25272	195292
{ X11/3, X12/6, X13/8, X21/0, X22/4, X23/6, X31/0, X32/3 }	32384	184719
{ X11/3, X12/5, X13/8, X21/0, X22/4, X23/6, X31/0, X32/3 }	37106	179484
{ X11/4, X12/6, X13/8, X21/1, X22/3, X23/6, X31/0, X32/3 }	110452	121825
{ X11/3, X12/6, X13/8, X21/1, X22/3, X23/6, X31/0, X32/3 }	111531	112500
{ X11/3, X12/5, X13/8, X21/1, X22/3, X23/6, X31/0, X32/3 }	116724	105895
{ X11/4, X12/6, X13/8, X21/1, X22/4, X23/6, X31/0, X32/3 }	129241	92451
{ X11/3, X12/6, X13/8, X21/1, X22/4, X23/6, X31/0, X32/3 }	136279	86145
{ X11/3, X12/5, X13/8, X21/1, X22/4, X23/6, X31/0, X32/3 }	142002	81989
{ X11/4, X12/6, X13/8, X21/2, X22/4, X23/6, X31/0, X32/3 }	227164	29625
{ X11/3, X12/6, X13/8, X21/2, X22/4, X23/6, X31/0, X32/3 }	229245	29602
{ X11/3, X12/6, X13/8, X21/2, X22/4, X23/6, X31/0, X32/3 }	239543	29578
Average time	88356	150370

Table 5. Results of simulation of Program_2 on a number of 4 processors

8 Conclusions

In this work, a new strategy for problem solving that combines parallel logic programming and heuristics for guide the search is proposed. The experimental results show that logic programming can express the parallelism that constraint satisfaction problems exhibit, and that heuristics can help to design these programs in order to improve performance. Nevertheless, in order to obtain more reliable results, experimentation with bigger problem instances would be necessary, for example with the benchmarks proposed in [Sad96]. In order to do that, we are developing a more powerful tool capable of simulating bigger programs in a reasonable amount of time. We expect that simulation studies will allow us to improve the strategy of designing heuristic programs that can be further executed on a real parallel machine in order to compare the results with other approaches.

REFERENCES

[Ado90] H. M. Adorf and M. D. Johnston. *A discrete stochastic neural network algoritm for constraint satisfaction problems*. Proc. of the International Joint Conference on Neural Networks. San Diego. 1990.

[Con83] J. S. Conery, *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. Ph. D. Th. Dpto. Information and Computer Science. Univ. California. Irvine. 1983

[Cor97] D. Corne and P. Ross. *Practical Issues and Recent Advances in Job- and Open- Shop Scheduling.* Eds. D. Dasgupta and Z. Michalewicz. Springer-Verlag.

[Gup92] G. Gupta *Parallel Execution of Logic Programs on Shared Memory Multiprocessors.* Ph. D. Thesis Dept. Of Computer Science. Univ. North Carolina at Chapel Hill. 1992.

[Hen92] P. van Hentenryck, H. Simonis and M. Dincbas. *Constraint satisfaction using constraint logic programming*. Artificial Intelligence 58, pp. 113-159. 1992.

[Kal91] L. V. Kalé. *The REDUCE-OR Process Model for Parallel Interpretation of Logic Programs*. The Journal of Logic Programming. Vol 11, pp. 55-84. 1991

[Pon95] E. Pontelli, G. Gupta and M. Hermenegildo. *&-ACE: A High Performance Parallel Prolog System.* Proc. 9th International Parallel Processing Symposium, pp. 564-571. IEEE Press. 1995.

[Sad96] N. Sadeh and M. S. Fox. *Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem*. Artificial Intelligence 86, pp. 1-41. 1996.

[She96] K. Shen. *Initial Results from the Parallel Implementation DASWAM*. Proc. of the Joint International Conference and Symposium on Logic Programming. MIT Press. 1996.

[Var94] R. Varela. *El Modelo RPS para la Gestión del Paralelismo AND Independiente en Programas Lógicos*. Proccedings of the 1994 Joint Conference on Declarative Programming GULP_PRODE'94, pp. 251-265. 1994.

[Var95a] R. Varela, E. Sierra, L. Jiménez y C. R. Vela. *Combinación de Soluciones Parciales en Programación Lógica Paralela*. C-AEPIA'95. Alicante. 1995.

[Var95b] R. Varela. *Un Modelo para el Cálculo Paralelo de Deducciones en Lógica de Predicados*. Tesis Doctoral. Departamento de Matemáticas, Universidad de Oviedo. 1995.

[Var96a] R. Varela and C. R. Vela. *AND/OR Trees for Parallel Deductions*, ITHURS'96. León, Spain. July 1996.

[Var96b] R. Varela, C. R. Vela and J. Puente. *Efficient Producer/Consumer Parallelism in Logic Programming*. APPIA-GULP-PRODE'96. San Sebastian. July 1996.

[War90] D. H. D. Warren. *The Extended Andorra Model with Implicit Control*. ICLP'90 Parallel Logic Programming Workshop. 1990.

[Zwe92] M. Zweben, E. Davis, B. Daun, E. Drascher, M. Deale and M. Eskey. *Learning to improve constraint-based scheduling*. Artificial Intelligence 58, pp. 271-296. 1992.