

A Sequential View of AND-parallelism Through Partial AND-Processes

Bernd Schend

BASF AG

ZXT-Informatik-Technologie

6700 Ludwigshafen, West Germany

Abstract

Most implementations of AND-parallelism tackle the shared variable problem by running literals in parallel only if they have no variables in common and thus are independent from each other. But even in its independent form, AND-parallel clause execution remains an intricate matter with numerous problems to be solved. First to mention, a carefully worked out synchronization mechanism is required to control process activities like dependency checking, literal ordering, message passing and intelligent backtracking. Due to the complex nature of these problems, implementation of AND-parallelism often results in a rather opaque interplay of processes. Therefore, based on a new kind of process, this paper presents a sequential - Prolog like - view of AND-parallelism for making it more understandable on one hand and easier to implement on the other. Processes of the new type will be called *partial AND-processes*.

1 Introduction

In process based systems AND-parallelism is usually realized by introducing producer/consumer relationships among clause literals and their corresponding OR-processes. Therewith independent literals act as parallel producers of variable bindings, which will be consumed by dependent literals in subsequent steps. Dividing clause literals into producers and consumers shows the advantage of avoiding binding conflicts since shared variables can have only one producer at a time. Realizing the producer/consumer approach to AND-parallelism, however, requires a sophisticated synchronization scheme to guarantee correct process interaction. Especially in the backward phase of clause evaluation [Chang, 1985, Conery, 1987], a lot of activities have to be controlled, ranging over message passing, reordering literals, cancellation and creation

of processes. Moreover, to complicate things, the whole synchronization job is settled on a single AND-process in most systems. But loading a process with so much work may seriously affect its message passing capacity and thus turn it into a communication bottleneck. With all these drawbacks in mind one might question, whether producer/consumer parallelism can actually be realized in a better arranged manner. Obviously it would help a system designer if he could view parallel clause execution in terms of a well known and easy to understand sequential system. Offering such a view through *partial AND-processes*, a new kind of process, will be the subject of this paper.

The paper starts by introducing some notational framework. After that, the principles of process based program execution will be illustrated by the hand of AND/OR-trees. Two fundamental search strategies for such trees will be discussed then, before the special strategy giving AND-parallelism a sequential touch is explained in section four. Finally we pick up the task of producing output substitutions through intelligent backtracking in order to demonstrate how partial AND-processes solve a typical implementation problem of AND-parallelism.

2 Notation

For the rest of the paper the reader is assumed to be familiar with the basic terminology of logic programming as introduced by Lloyd [1984], for example. Some supplementary notation used subsequently is given below.

Variables will be named by letters x, y, z, u , and v . Letters a, b, c , and d denote constants, and f, g, h function symbols. An *expression* can be a term or literal. If e represents an expression, $\text{Var}(e)$ is the set of all variables occurring in e . Two expressions e_1 and e_2 are called *dependent* if $\text{Var}(e_1) \cap \text{Var}(e_2) \neq \emptyset$. Substitutions are named by Greek letters like θ, σ , and γ ; a substitution $\theta = \{x_1/t_1, \dots, x_n/t_n\}$, $n \geq 0$, *binds* variable x_i to

the term t_i , $1 \leq i \leq n$. *Composition* $\theta_1 \circ \theta_2$ of two substitutions is defined as $\theta_1 \circ \theta_2(e) = \theta_1(\theta_2(e))$ for every expression e . Finally, the number of elements of a finite set M will be denoted by $|M|$.

3 Program Execution by AND/OR-processes

Every logic program - started with some goal - implicitly defines an AND/OR-tree [Kowalski 1979], which makes program execution a matter of searching all the answers to the user's query. In this connection an execution strategy for logic programs naturally turns into a *search strategy*, which tells a system how to traverse the tree under consideration. To illustrate the effects of different search strategies, we now describe the behaviour of some fictitious process systems when executing the first clause from figure 1.

- | | | |
|--|-----------------------|-----------------------|
| (1) $p(x,y) \leftarrow q(x), r(y), s(x)$ | | |
| (2) $q(a) \leftarrow$ | (4) $r(c) \leftarrow$ | (6) $s(a) \leftarrow$ |
| (3) $q(b) \leftarrow$ | (5) $r(d) \leftarrow$ | (7) $s(b) \leftarrow$ |

Figure 1 Example program

Based on AND/OR-trees, processes created during program execution constitute an *AND/OR-process tree*. By the way, viewing processes as nodes of a tree allows us, for instance, to talk of the AND-father of a process, just saying the father is an AND-process. Conventions for drawing process trees are shown in figure 2.

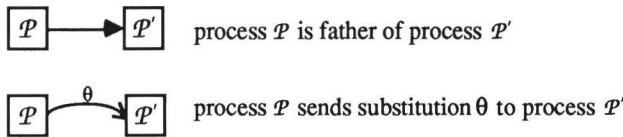


Figure 2 Parts of a process tree

As the special strategy coming up with partial AND-processes will be both, sequential and parallel in nature, the single components of this mixture will be considered first.

Sequential Search Strategy

The sequential component within our mixed strategy is a left to right one comparable to that used in standard Prolog systems. Figure 3 mirrors its effect on execution of clause $p(x,y) \leftarrow q(x), r(y), s(x)$, which - for the sake of simplicity - is assumed to be called with the empty input $\theta_0 = \emptyset$.

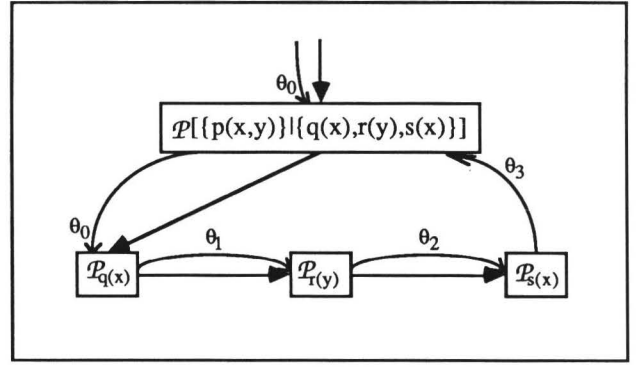


Figure 3 Sequential execution of $p(x,y) \leftarrow q(x), r(y), s(x)$

The process tree resulting from our left to right strategy shows an AND-process, named $P[\{p(x,y)\} | \{q(x), r(y), s(x)\}]$, which is created to execute our example clause. In order to solve this task, the AND-process initially starts a descendant OR-process $P_{q(x)}$ to compute solutions for $\theta_0 q(x) = q(x)$. Having found the answer $\gamma_1 = \{x/a\}$, process $P_{q(x)}$ produces its output substitution $\theta_1 = \gamma_1 \circ \theta_0 = \{x/a\}$ and then creates a descendant OR-process $P_{r(y)}$ to solve literal $\theta_1 r(y) = r(y)$. On the basis of our example program, $P_{r(y)}$ subsequently finds the answer $\gamma_2 = \{y/c\}$ and computes the output $\theta_2 = \gamma_2 \circ \theta_1 = \{x/a, y/c\}$. Next, OR-process $P_{s(x)}$, created for $\theta_2 s(x) = s(x)$, will find $\gamma_3 = \emptyset$ as a solution and send substitution $\theta_3 = \gamma_3 \circ \theta_2 = \{x/a, y/c\}$ to the AND-process above. Finally, θ_3 represents a solution for the whole program clause.

Parallel Search Strategy

In contrast with a sequential strategy, using a parallel one causes some body literals to be executed in parallel. Figure 4 shows the effect of a somehow extremely parallel strategy since it solves all body literals simultaneously.

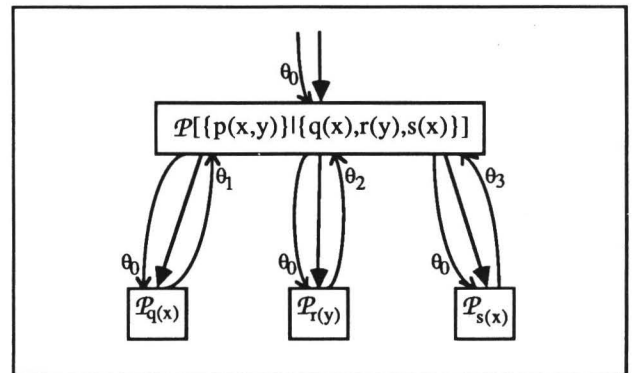


Figure 4 Parallel execution of $p(x,y) \leftarrow q(x), r(y), s(x)$

According to the parallel strategy, all body literals are executed independently by corresponding OR-processes. Process $\mathcal{P}_{q(x)}$, for instance, works on $\theta_0 q(x) = q(x)$ and produces the output substitution $\theta_1 = \{x/a\}$. In the same way, processes $\mathcal{P}_{r(y)}$ and $\mathcal{P}_{s(x)}$ compute answers for $\theta_0 r(y)$ and $\theta_0 s(x)$, which we assume to be $\theta_2 = \{y/c\}$ and $\theta_3 = \{x/b\}$, respectively. Yet, from the point of the AND-process above, θ_1 , θ_2 and θ_3 represent only partial solutions for its own problem, which is to find answers for whole program clause. Therefore, having received the single solutions, an AND-process has to join them to a complete one, provided values of shared variables are consistent with each other. Consistency of variable bindings, however, is not guaranteed. Just look at processes $\mathcal{P}_{q(x)}$ and $\mathcal{P}_{s(x)}$ from above, which bind variable x to non unifiable values.

To get around the consistency problem, most systems [Chang, 1985, Conery, 1987, DeGroot, 1984] synchronize OR-processes by making them producers or consumers of variable bindings. Since shared variables can have only one producer at a time, no conflicting bindings will arise. But as mentioned before, producer/consumer parallelism often leads to a rather intricate interplay of processes. The intention of this paper, therefore, is to give a more perspicuous view of AND-parallelism and - by doing so - convey new ideas of how typical implementation problems can be solved in a more understandable and efficient manner. A new type of process, called partial AND-process, will play a central role thereby.

4 Partial AND-processes

As their name already indicates, partial AND-processes bear some likeness to *normal* AND-processes, but differ in the scope of the problem to deal with. Whereas an AND-process has to solve all the literals of a clause body, a partial AND-process works on a subset of body literals only. Such a subset just corresponds to a group of independent literals selected for parallel execution. Whereas literals within a group will be solved in parallel, group execution is purely sequential in nature. Consequently, we get a Prolog like clause execution on the level of p(artial)AND-processes and their corresponding literal groups (see upper half of figure 5).

In regard of a more detailed description of what happens in figure 5, we first of all claim that special routines for dependency checking and literal grouping are available to AND- and pAND-processes. Due to lack of space, we simply view these routines as black boxes and describe their behaviour only superficially. Again, for the execution of our program

clause we assume its input substitution θ_0 being empty.

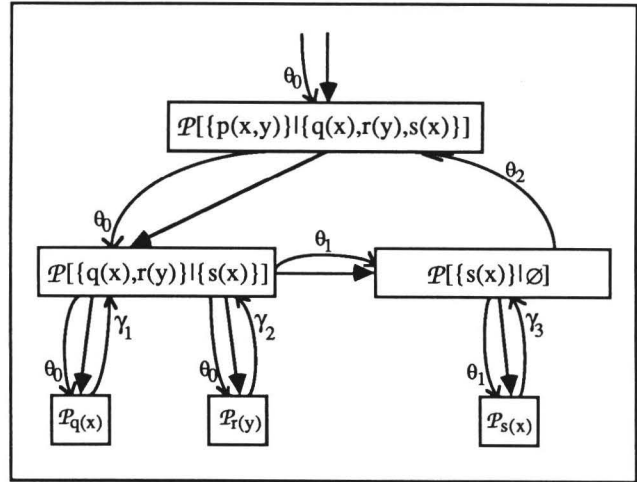


Figure 5 pAND-parallel execution of $p(x,y) \leftarrow q(x), r(y), s(x)$

The AND-process $\mathcal{P}[\{p(x,y)\} | \{q(x), r(y), s(x)\}]$ starts clause execution by determining literal dependencies within the environment θ_0 . This practically means testing, whether $\text{Var}(\theta_0 L) \cap \text{Var}(\theta_0 L') \neq \emptyset$ for all pairs (L, L') of distinct body literals. As a result we get two alternatives for grouping independent literals, namely $G_1 = \{q(x), r(y)\}$ and $G_2 = \{s(x), r(y)\}$. Next, the AND-process decides to group $q(x)$ and $r(y)$ together, and upon that, has them executed by a pAND-process $\mathcal{P}[\{q(x), r(y)\} | \{s(x)\}]$. This one starts OR-processes $\mathcal{P}_{q(x)}$ and $\mathcal{P}_{r(y)}$ to solve literals $\theta_0 q(x)$ and $\theta_0 r(y)$ in parallel. On the basis of our example program from figure 1 the answers $\gamma_1 = \{x/a\}$ and $\gamma_2 = \{y/c\}$ will be computed and send to $\mathcal{P}[\{q(x), r(y)\} | \{s(x)\}]$. Having received both substitutions, the pAND-process produces the output substitution $\theta_1 = (\gamma_1 \cup \gamma_2) \circ \theta_0 = \{x/a, y/c\}$ and immediately creates its son $\mathcal{P}[\{s(x)\} | \emptyset]$ with input substitution θ_1 . In order to solve $\theta_1 s(x) = s(a)$ the pAND-son starts an OR-process $\mathcal{P}_{s(x)}$, from which it receives the answer substitution $\gamma_3 = \emptyset$. Finally, $\mathcal{P}[\{s(x)\} | \emptyset]$ computes its output $\theta_2 = \gamma_3 \circ \theta_1 = \{x/a, y/c\}$ and transfers it via message to the AND-process above.

The last example showed partial AND-processes acting as producers and consumers of variable bindings. Yet, unlike Conery's model [Conery, 1987], bindings are not produced for single literals, but for whole groups of literals. To identify the literal group a pAND-process is executing we have used the notation $\mathcal{P}[\Gamma | \Pi]$ with Γ and Π being sets of clause literals. Separation by a stroke characterizes Γ as the literal group actually executed by the process, whereas literals in Π have to

be solved by other pAND-processes in subsequent steps. Due to this separation, literals in Γ get an active and literals in Π a passive character from the point of $\mathcal{P}[\Gamma\Pi]$.

Definition 4.1

Suppose $\mathcal{P}$ is an AND- or pAND-process named $\mathcal{P}[\Gamma\Pi]$. Then $\mathcal{AL}(\mathcal{P}) = \Gamma$ is the set of *active* and $\mathcal{PL}(\mathcal{P}) = \Pi$ the set of *passive* literals of $\mathcal{P}$.

Active literals and their corresponding OR-processes can be regarded as producers of variable bindings, which are gathered by a pAND-process to build its own output substitutions. However, that there are several OR-processes contributing to an output, is completely hidden to the pAND-son, which consumes it as an input substitution. From the point of a consuming son, its pAND-father together with all its active literals simply appears as a single module just producing output substitutions.

5 Producing Output Substitutions

Solutions produced by pAND-processes will be sent as output substitutions to the consuming process, which itself receives them as input substitutions. In general, sending of solutions is demand driven, since new output/input substitutions are only produced when explicitly requested by the consumer. Output substitutions generated by pAND-processes are characterized below.

Definition 5.1

Let $\mathcal{P}$ be a pAND-process with input substitution θ_{in} and its active literal set $\mathcal{AL}(\mathcal{P}) = \{L_1, \dots, L_n\}$, $n \geq 1$. Further, let $S_1, \dots, S_n$ be the sets of answer substitutions computed by descendant OR-processes, i.e. S_j contains all answers found for $\theta_{in}L_j$, $1 \leq j \leq n$.

Then $OS(\mathcal{P}, \theta_{in}) = \{\text{Join}(\gamma_1, \dots, \gamma_n) \cdot \theta_{in} \mid \gamma_j \in S_j, 1 \leq j \leq n\}$ is the set of *output substitutions* of $\mathcal{P}$, where $\text{Join}(\gamma_1, \dots, \gamma_n) = \gamma_1 \cup \dots \cup \gamma_n$.

The definition above points out that every output substitution produced by a pAND-process consists of a constant and variable part. The first one corresponds to the input substitution θ_{in} and the latter to a tuple $(\gamma_1, \dots, \gamma_n)$ of answers computed by descendant OR-processes. Therefore, given a fixed input substitution θ_{in} , each output substitution $\theta_{out} = \text{Join}(\gamma_1, \dots, \gamma_n) \cdot \theta_{in}$ is practically determined by the tuple $(\gamma_1, \dots, \gamma_n)$. Moreover every such tuple in its turn may be identified with a tuple of natural numbers. Simply realize that the elements of a set S_j can be numbered in some way, for

instance, in the order they are computed by the corresponding OR-process. Conversely, to get an output substitution from a tuple $(m_1, \dots, m_n)$ of element numbers, just take the m_1 -th element from S_1 , the m_2 -th element from $S_2, \dots$, the m_n -th element from S_n , join them together and produce a θ_{out} through composition with θ_{in} .

This strong relationship between number tuples and output substitutions allows us to make production of output substitutions more perspicuous by reducing it to a problem of generating tuples of natural numbers. For solving the generation problem we simply adopt the nested loop model introduced by Conery [1987]. A pAND-process will always start tuple generation with the initial tuple $(1, \dots, 1)$, subsequently changing its components as if they were variables of nested for-loops, i.e. the rightmost components run faster than the ones to the left. Note, if component i , $1 \leq i \leq n$, is increased, all components j , $i < j \leq n$, are reset to 1. As number tuples of the kind mentioned above directly determine output substitutions, they are called *output tuples* from now on.

As said before, production of output tuples and substitutions is demand driven, i.e. a pAND-process produces a new output only when explicitly requested by its pAND-son. Such a request always goes along with a *backtracking between partial AND-processes*. What's behind this type of backtracking will be demonstrated below with a slightly modified version of our example program.

- (1) $p(x, y) \leftarrow q(x), r(y), s(x), t(y)$
- (2) $q(a) \leftarrow$ (4) $r(c) \leftarrow$ (6) $s(b) \leftarrow$ (7) $t(u) \leftarrow$
- (3) $q(b) \leftarrow$ (5) $r(d) \leftarrow$

To execute clause (1) with input substitution $\theta_0 = \emptyset$, an AND-process, named $\mathcal{P}[(p(x, y)) \mid (q(x), r(y), s(x), t(y))]$, will be started initially. Let us assume $q(x)$ and $r(y)$ to be grouped together after dependency checking and executed by a pAND-process $\mathcal{P}[(q(x), r(y)) \mid (s(x), t(y))]$. For solving $\theta_0 q(x)$ and $\theta_0 r(y)$, OR-processes $\mathcal{P}_{q(x)}$ and $\mathcal{P}_{r(y)}$ will be created subsequently, which compute the answers $\gamma_{11} = \{x/a\}$, $\gamma_{12} = \{x/b\}$ for the first, and $\gamma_{21} = \{y/c\}$, $\gamma_{22} = \{y/d\}$ for the second literal. Suppose that the pAND-process receives the substitutions in the order γ_{11} , γ_{12} and γ_{21} , γ_{22} , respectively. According to the nested loop model, the initial tuple $(1, 1)$ is used to produce the first output substitution $\theta_{(1,1)} = \text{Join}(\gamma_{11}, \gamma_{21}) \cdot \theta_{in} = \{x/a, y/c\}$. Next, the process creates its pAND-son $\mathcal{P}[(s(x), t(y)) \mid \emptyset]$ with input $\theta_{(1,1)}$. This son in return starts the OR-processes $\mathcal{P}_{s(x)}$ and $\mathcal{P}_{t(y)}$, the first of which detects that there are no matching clauses for its literal $\theta_{(1,1)}s(x) = s(a)$. Figure 6 shows a snapshot of the process tree

at this moment.

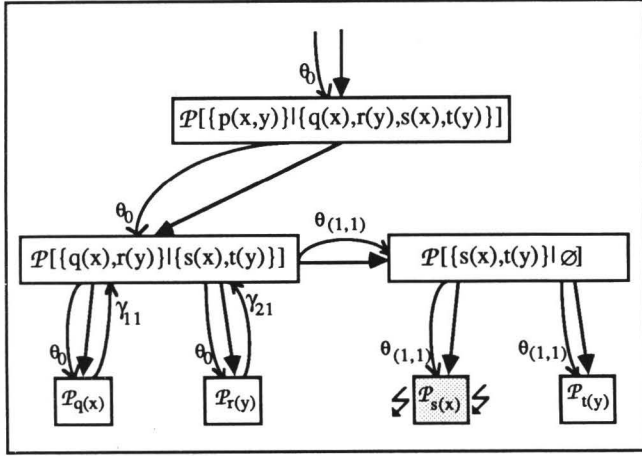


Figure 6 Backtracking situation between pAND-processes

Process $P_{s(x)}$ reports its failure to $P[{s(x),t(y)}|\emptyset]$, which immediately reacts by requesting a new input substitution from its pAND-father $P[{q(x),r(y)}|{s(x),t(y)}]$. Based on the nested loop model, the father generates the next output tuple (1,2) and sends the corresponding substitution $\theta_{(1,2)} = \text{Join}(\gamma_{11}, \gamma_{22}) \cdot \theta_{in} = \{x/a, y/d\}$ to the requesting pAND-son. However, $\theta_{(1,2)}$ will obviously lead to a failure, too, because the value of variable x remains unchanged by the new substitution. So, our pAND-process generated just another *failure tuple*.

Definition 5.2

Let $\mathcal{P}$ be a pAND-process with output tuple t and corresponding output substitution θ_t . Tuple t is called a *failure tuple* if $\theta_t L$ is unsolvable for some active literal of the pAND-son of $\mathcal{P}$. Else it will be called a *success tuple*.

To avoid repeated generation of failure tuples - like (1,1) and (1,2) in the example above - a pAND-process must incorporate some kind of intelligent backtracking [Conery, 1987, Woo and Choe, 1986], or equivalently, use some form of *intelligent tuple generation*. This should, for instance, allow process $P[{q(x),r(y)}|{s(x),t(y)}]$ of figure 6 to generate tuple (2,1) directly without regarding (1,2). Tuple (2,1) and its corresponding output substitution $\theta_{(2,1)} = \text{Join}(\gamma_{12}, \gamma_{21}) \cdot \theta_{in} = \{x/b, y/c\}$ bind x to a different value, with which clause execution succeeds finally.

The question now is, how tuple generation can be made intelligent. To achieve this aim, a pAND-process $\mathcal{P}$ obviously needs some kind of information about what was wrong with its actual output substitution. Within the parallel execution scheme presented here, this information will be supplied by the

pAND-son of $\mathcal{P}$ by sending all its *failure literals*.

Definition 5.3

Let $\mathcal{P}$ be a pAND-process named $\mathcal{P}[\Gamma|\Pi]$. An active literal $L \in \Gamma$ is called a *failure literal* if its corresponding OR-process fails to compute any answer substitution. The set of all failure literals of $\mathcal{P}$ is denoted by $\mathcal{FL}(\mathcal{P})$. For an AND-process $\mathcal{P}$, we set $\mathcal{FL}(\mathcal{P}) = \Gamma$ by definition.

Our example process $P[{s(x),t(y)}|\emptyset]$ from figure 6, for instance, would report $s(x)$ as a failure literal to its pAND-father $P[{q(x),r(y)}|{s(x),t(y)}]$. Now the father process can conclude that it must produce a new binding for variable x , for changing the value of y would definitely not repair the failure and thus lead to another backtracking. As a new value for x can only be achieved by changing the first component of the actual tuple (1,1), we automatically know (1,2) to be another failure tuple. This is confirmed by looking at its corresponding output substitution $\theta_{(1,2)} = \{x/a, y/d\}$, which changes the value of y , but not that of x .

In order to determine all its failure literals a pAND-process proceeds as follows: After descendant OR-processes have been created for every active literal, it enters a kind of *test state*. There the process waits until all OR-sons have acknowledged by sending a success or failure message. The literal of a failing OR-process will be recorded as a failure literal thereby. Note, a pAND-process will wait forever if one of its OR-sons is caught in an infinite loop. By the way, waiting for all OR-processes to acknowledge allows us to handle so called multiple failures [Conery, 1987, Woo and Choe, 1986] in a rather simple manner. Further, collecting all failure literals before backtracking results in a more intelligent version since information about failures gets more comprehensive. This information will be used to repair the failure by generating a new output tuple, the corresponding substitution of which binds at least one variable in each failure literal to another value. Such tuples and substitutions will be called *failure relevant* from now on.

In terms of our nested loop model the whole problem of generating tuples intelligently may now be stated as: What loop variable, or equivalently, what tuple component should be changed in order to make its corresponding output substitution relevant to the failure? To answer this question we first of all introduce a successor relation on AND- and pAND-processes. Definition of the relation is based on the process tree of figure 7, which shows an AND-process $\mathcal{P}[\Gamma_0|\Pi_0]$ with descendant pAND-processes $\mathcal{P}[\Gamma_1|\Pi_1], \dots, \mathcal{P}[\Gamma_n|\Pi_n]$. Note, $\Pi_n = \emptyset$.

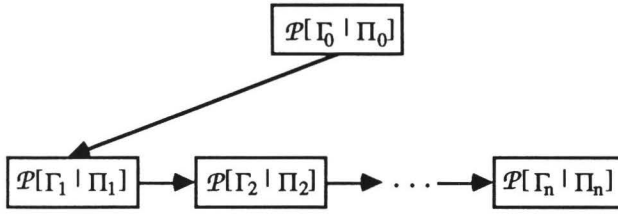


Figure 7 AND-process $\mathcal{P}[\Gamma_0 | \Pi_0]$ with descendant pAND-processes

Definition 5.4

Let $\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_n, n \geq 1$, be the processes of figure 7 with $\mathcal{P}_i = \mathcal{P}[\Gamma_i | \Pi_i]$ and $0 \leq i \leq n$. The *successor* of $\mathcal{P}_i$ is defined by $\text{succ}(\mathcal{P}_i) = \mathcal{P}_j$ with $j = i+1 \bmod n+1$. Similarly we define the *predecessor* of $\mathcal{P}_j$ by $\text{pred}(\mathcal{P}_j) = \mathcal{P}_i$ iff $\text{succ}(\mathcal{P}_i) = \mathcal{P}_j$. Instead of $\text{succ}(\mathcal{P})$ and $\text{pred}(\mathcal{P})$, we also use the notations $\mathcal{P}_{\text{succ}}$ and $\mathcal{P}_{\text{pred}}$, respectively.

Next, we relate active literals of a pAND-process to active literals of its successor. Literals of the first will be treated as producers, literals of the second as consumers of variable bindings. In addition, a special function is introduced for mapping active literals to tuple components and vice versa.

Definition 5.5

Let $\mathcal{P}$ be a pAND-process with input substitution θ , $L_p \in \mathcal{AL}(\mathcal{P})$ and $L_c \in \mathcal{AL}(\mathcal{P}_{\text{succ}})$.

- a) $\text{consumer}(L_p) = \{L' \in \mathcal{AL}(\mathcal{P}_{\text{succ}}) \mid \text{Var}(\theta L_p) \cap \text{Var}(\theta L') \neq \emptyset\}$.
- b) $\text{producer}(L_c) = \{L' \in \mathcal{AL}(\mathcal{P}) \mid \text{Var}(\theta L_c) \cap \text{Var}(\theta L') \neq \emptyset\}$.
- c) $\#$ is a bijective mapping from $\mathcal{AL}(\mathcal{P})$ to $\{1, \dots, n\}$,
 $n = |\mathcal{AL}(\mathcal{P})|$. For a set Γ of literals we define
 $\#\Gamma = \{\#L \mid L \in \Gamma\}$.

To get an idea of intelligent tuple generation, imagine a pAND-process $\mathcal{P}$ with input substitution $\theta_0 = \emptyset$ and $\mathcal{AL}(\mathcal{P}) = \{p(x), q(y), r(z)\}$, where $\#p(x) = 1$, $\#q(y) = 2$ and $\#r(z) = 3$. In addition, let $(2,3,3)$ be the actual failure tuple of $\mathcal{P}$ and $\text{FL} = \{s(x,y), u(z)\}$ the set of failure literals reported by its successor. First of all $\mathcal{P}$ computes $\text{producer}(L)$ for every $L \in \text{FL}$ resulting in $\text{producer}(s(x,y)) = \{p(x), q(y)\}$ and $\text{producer}(u(z)) = \{r(z)\}$. Mapping producer literals to tuple indices by the hand of function $\#$, we get $\#\text{producer}(s(x,y)) = \{1,2\}$ and $\#\text{producer}(u(z)) = \{3\}$. Now, set $\{1,2\}$ expects us to change the first or second component of $(2,3,3)$, in order to produce an output substitution which binds a variable in $s(x,y)$ to another value. Accordingly, set $\{3\}$ demands the third component to be changed in order to get a new binding for

$u(z)$. Altogether, both requirements can be satisfied by increasing the second tuple component, because this will automatically reset the third component to 1. Hence, the resulting tuple $(2,4,1)$ causes producer literals $q(y)$ and $r(z)$ to contribute to the new output substitution, and all in all, modified variable bindings will be brought to their consumers $s(x,y)$ and $u(z)$.

In general, the simple algorithm below will be used by pAND-processes to determine the tuple component to be changed after backtracking.

Algorithm for Changing Tuple Components

Let $\mathcal{P}$ be a pAND-process $\mathcal{P}[\Gamma | \Pi]$ and FL the set of failure literals reported by its successor. Further, for a literal $L \in \text{FL}$ let $\max_L = \max\{\#L' \mid L' \in \text{producer}(L)\}$.

```

if  $\text{producer}(L) \neq \emptyset$  for every  $L \in \text{FL}$ 
then  $i := \min\{\max_L \mid L \in \text{FL}\};$ 
      "change the i-th tuple component"
else "stop tuple generation".

```

In the example above we had $\text{FL} = \{s(x,y), u(z)\}$. Computing $\max_L$ for every failure literal L results in $\max_{s(x,y)} = \max\{1,2\} = 2$ and $\max_{u(z)} = \max\{3\} = 3$. Using the algorithm for changing tuples, we get the 2-nd component, $2 = \min\{2,3\}$, as the one to be changed.

In general, changing the i-th component of a tuple usually means increasing its value by one. However, this may be impossible sometimes because all members of the answer set S_i have already been considered, i.e. the value of the i-th component corresponds to the number of elements in S_i . Should this happen, a pAND-process reacts by looking for the first component to the left of i which can be changed. If there is such a component, its value will be increased therewith resetting all components to its right to 1. After having produced the corresponding output substitution, the process enters a kind of forward state, in which dependency checking and literal grouping is done again based upon the new substitution. If the resulting literal group is different from that of the actual pAND-son, the son process will be cancelled and a new one created. Otherwise no cancelling is necessary and the substitution is directly sent to the actual pAND-son. Upon that, all consumer literals receiving a new input substitution will be resolved, i.e. their corresponding OR-processes are initialized and restarted with the new bindings. Finally, having restarted its OR-sons, a pAND-process enters the test state mentioned before, waiting again for success or failure messages.

Over against this, if a pAND-process should fail to find an increasable tuple component, no more output tuples can be generated and tuple generation is finished. A pAND-process, which is unable to produce new output substitutions, initiates backtracking to its own predecessor. In this manner backtracking eventually arrives at the AND-process above, telling it that clause execution is done.

At the end of this section some aspects of our backtracking scheme should be discussed:

The nested loop model for tuple generation shows the advantage of being easy to implement. But its simplicity must be paid for in some sense as tuple generation may become incomplete if OR-processes compute infinite answer sets. One might expect this a serious drawback since many tuples will never be considered. Yet, generating all of infinitely many tuples is impossible, anyway. The only disadvantage is that successful output substitutions are probably not produced if a pAND-process is stuck in an infinite answer set. Fortunately loops of this kind may be stopped through intelligent backtracking.

Requesting a new output/input substitution through backtracking is generally not restricted to pAND-processes, but also used by an AND-process to obtain all solutions for its clause. This is simply done by backtracking to its own predecessor immediately after having received an input substitution from it. The head literal of the clause will be reported as the failure literal thereby.

6 Conclusions

Implementing AND-parallelism in the producer/consumer style requires a sophisticated synchronization scheme for controlling OR-processes, which work on body literals independently. Especially handling failures through intelligent backtracking makes clause execution a rather complicated problem, the solution of which often results in a confusing interplay of processes in practice. To make AND-parallel clause execution more perspicuous and therefore easier to implement, this paper conveyed a sequential view of AND-parallelism. Based on a heterogeneous execution strategy, which is sequential and parallel in nature, independent literals are grouped together and treated as a single module. Literal groups are solved sequentially by partial AND-processes, which in their turn start descendant OR-processes for executing group literals in parallel. The main advantage resulting from the sequential part of the execution strategy is that clause execution may be tackled in a simple Prolog like manner. As an example, the

backtracking scheme presented here bears some similarity to the naive right to left method of Prolog, except that it is made intelligent on the basis of producer/consumer relations. Multiple failures of OR-processes can be handled efficiently by partial AND-processes, all the more since information about failing literals is gathered to achieve a more intelligent backtracking version. Apart from this, the load on AND-processes will be eased significantly, allowing them to react to incoming messages promptly.

References

- [Chang, 1985] J.H. Chang. *High Performance of Prolog Programs Based on a Static Data Dependency Analysis*. Ph.D. Thesis, 1985.
- [Conery, 1987] John S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, 1987.
- [DeGroot, 1984] Doug DeGroot. Restricted And-Parallelism. In *Proceedings of the International Conference On Fifth Generation Computer Systems*, Tokyo, 1984, pp. 471-478.
- [Kowalski, 1979] Robert A. Kowalski. *Logic for Problem Solving*, Artificial Intelligence Series, Vol.7, Elsevier-North Holland, New York, 1979.
- [Lloyd, 1984] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1984.
- [Woo and Choe, 1986] N.S. Woo and K.-M. Choe. Selecting the Backtrack Literal in the AND/OR-Process Model. In *IEEE 1986 Symposium on Logic Programming*, Salt Lake City, Utah, pp. 200-210.