Proceedings of the 1993 International Conference on Logic Programming

Some Paradigms for Visualizing Parallel Execution of Logic Programs[1]

M. Carro
L. Gómez
M. Hermenegildo
{mcarro, lgomez, herme}@fi.upm.es
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
Boadilla del Monte, Madrid 28660—Spain

Abstract

This paper addresses the design of visual paradigms for studying the parallel execution of logic programs. First, an intuitive method is proposed for arriving at the design of a paradigm and its implementation as a tool for a given model of parallelism. This method is based on stepwise refinement starting from the definition of basic notions such as events and observables and of some precedence relationships among events which hold for the given model of parallelism. The method is then applied to several types of parallel execution models for logic programs (Or–parallelism, Restricted And–parallelism, Determinate Dependent And parallelism) for which visualization paradigms are designed. Finally, VisAndOr, a tool which implements all of these paradigms is presented, together with a discussion of its usefulness through examples.

1 Introduction

Writing programs for parallel hardware has traditionally been considered a difficult task both because of the intrinsic difficulty of having to coordinate several execution threads and because of the need for considering the particular characteristics of the target machine which may also arise. On the other hand, languages which are essentially declarative, and logic languages in particular, offer great opportunities for transparently exploiting parallelism. Their well understood semantics makes them more amenable to automatic parallelization than traditional imperative languages, thus freeing the programmer from the error–prone task of data dependency analysis. One remaining problem, however, is that much of the complexity is transferred to the compiler or the program evaluation system, whose implementation then becomes quite a challenge. Furthermore, in practice, although programmers are certainly freed from worrying about low level issues, their view may be so separated from the real execution that it may be difficult for them to realize how their program is behaving. It is our belief, and the thesis of this paper, that a clear and intuitive graphical presentation of the actual parallel execution structure at a suitable level of abstraction can greatly help both the implementor of logic programming systems and the user of such systems to achieve better results in their tasks.

In this paper we will present an approach to the study of the run–time behavior of parallel logic systems based on devising *visualization paradigms* which represent

[1]The research presented in this paper has been supported in part by ESPRIT projects number 6707 "PARFORCE" and number 2471 "PEPMA" .

the execution of such systems. The gap between the general characteristics of program execution in the parallel system under study and the final visualization paradigm used to represent it will be bridged using a methodology based on stepwise refinement, which is sketched in Section 2. In Section 3 different visualization paradigms will be devised in a natural fashion for a number of models of parallelism in logic programs, based on the structure of the dependencies that hold for the execution represented. In Section 4 VisAndOr, a tool implementing such paradigms, will be presented. Its features and usefulness will be illustrated through examples. Section 5 presents some details of the implementation of VisAndOr. Section 6 provides an overview of related work. Section 7 suggests additional uses beyond visualization of the approach presented. Finally, Section 8 presents our conclusions and suggestions for future work.

2 Basic Notions and Methodology

In order to derive homogeneous visualization paradigms starting from the basic properties of different execution models we propose the aforementioned use of a methodology loosely based on stepwise refinement. We briefly introduce three basic notions for this purpose (since these concepts are quite primitive, we will rely somewhat on the reader's intuition to avoid verbosity): an **observable** is any generic characteristic of the system under study whose variation we want to track. An **event** is a uniquely distinguishable instantiation of an observable in an execution. A **trace** is a collection of events which corresponds to a particular execution. Observables abstract out details of concrete operations, and allow concentrating on characteristics predefined as interesting for study. Events usually include their type, which names the corresponding observable, and some additional information, which distinguishes a particular event from other events of the same type. Such information may include for example time stamps, goal and agent[2] identifiers, references to related events, etc. In general it is required that no two identical events can ever appear in the same execution. Also, we assume that we are pursuing a static visualization of the whole execution (a dynamic visualization can be seen as an incremental construction of a static one).

Once an execution model is chosen the first step is to decide at what abstraction level the model is to be visualized. This is done by defining the observables and the information which will be encoded in the events. Also, a notion of dependency among events is defined. A graph structure results from this dependency relation which is then used as the basis for developing the visualization. Finally, common characteristics of the possible graph structures generated are identified and unifying principles found in order to devise a concrete visualization paradigm. Such a paradigm should be as simple, flexible, accurate, and intuitive as possible, reflect the structure of the graph, and, hopefully, be homogeneous across execution models and types of parallelism. Of course, the final result can certainly only be satisfactory if the resulting visualizations prove to be useful enough in practice for program development or system debugging, and this may require several iterations through the design cycle. Finally, some models can be seen as a combination of several individual models. A visualization paradigm can often be derived in such cases by combining the corresponding individual visualization paradigms in a way that mimics the combined execution model. Sometimes this is not possible (prop-

[2]Throughout the paper we use the term "agent" (or worker) to refer to the process, normally mapped on a processor, that is working on a task.

Observable	Comment
START_EXECUTION	Start of the whole execution
END_EXECUTION	End of the whole execution
START_GOAL	The task (corresponding to a goal) starts
FINISH_GOAL	The task (corresponding to a goal) ends
SUSPEND	A task is suspended
RESTART	A task is restarted
FORK	Execution splits in two branches
JOIN	Different branches join

Table 1: Some common observables for parallel execution of logic programs

erties of independence do not hold, or the resulting representation is not intuitive or elegant). In these cases the problem may have to be tackled from scratch as a whole.

3 Visualization Paradigms

Our objective is to apply the methodology sketched in the preceding section and develop paradigms for the visualization of parallel execution models for logic programs. We will focus on the visualization of Or–parallelism [Lus88, CSW88, CA88, AK90, Kar92], Restricted And–parallelism (RAP) [DeG84, HG90], and Determinate Dependent And–parallelism (DDAP) as in [Nai88, SCWY90] and committed choice languages. We are also interested in visualizing combinations of these models.

3.1 Common Design Concepts

Examples of common observables, aimed at studying the systems of interest under the viewpoint of tasks rather than processes, are shown in Table 1. Given that our main interest is the visualization of *parallel* execution, the set of observables has been chosen to abstract away detail in the sequential parts. As additional characteristics to be visualized for particular execution models are identified, new observables will be defined. For brevity, the presentation will be somewhat simplified w.r.t. what is really present in the implementation (described in Section 4) both in terms of the number of events and the information content of the events.

With respect to the information contained in the events, since time is perceived to be an important issue in parallel execution, all events carry a time stamp corresponding to the time when the event occurred. This induces a natural precedence relation among events. Other types of (causal) dependencies are also present. For example, a goal can only start after its parent forks it. The conceptual graph for each execution is naturally constructed using both the time precedence and the other dependencies among events.

The visualization paradigms aim at effectively displaying the structure of the graph, as well as some information associated with each event. Temporal precedence will be assigned to spatial precedence in the vertical axis, so that the later an event is generated, the farther from the top it will be placed.[3] Despite the importance of time, it is not the only vertical coordinate basis that we will use, as will be shown

[3] This orientation was chosen instead of, for example, left–to–right orientation for similarity with the usual drawings of the resolution trees.

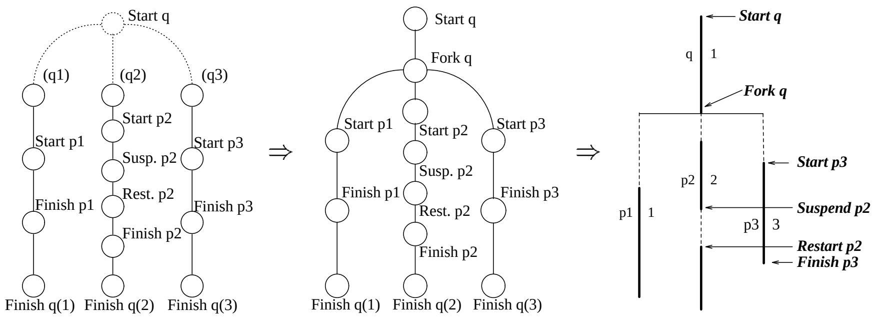

Figure 1: Some dependency graphs for Or–parallelism and a visualization

in Section 3.6. The information attached to the events will be depicted in a number of ways: for example, a different color can be assigned to each agent (or a pattern or label attached to the proper place in a monochrome display—e.g this paper).

3.2 Or–parallelism

Or–parallelism corresponds to the parallel execution of different alternatives of a given predicate. Since each alternative belongs conceptually to a different "universe" there are (in principle) no dependencies among alternatives. However, dependencies do arise in real systems due to the particular way in which common parts of alternatives are shared. Consider for example the following program which has three alternatives for predicate `p`:

```
q:- p.          p1:- ...
                p2:- ...
                p3:- ...
```

A possible dependency graph is the one depicted in Figure 1 left: different alternatives are represented by different *universes* (this might correspond, for example, to an execution on the Deplhi system [CA88]). Note that $\mathtt{p}_2$ is suspended at some point and then restarted. In fact, this suspension is probably caused by $\mathtt{p}_1$, in the sense that $\mathtt{p}_2$ is waiting for some built–in to be executed in $\mathtt{p}_1$. In this first design we have chosen to abstract these other types of dependencies away. Many practical models share computation up to the branch point (or copy what was done before at that point). This situation is depicted in Figure 1, center, where a FORK has been introduced, which explicitly shows the point where execution branches (this might correspond for example to a Muse [AK90, Kar92] or Aurora [Lus88, Car90] execution). One common important feature of the dependency graphs of Or–parallel executions is that branches do not join. In terms of dependencies among observables, FORKs do not need to be balanced by JOINs. The resulting graph is thus a tree.[4] We are assuming that `p`'s alternatives are sequential. Otherwise a similar representation would be recursively applied.

A visualization paradigm is shown in Figure 1, right. The nodes of the graph have been replaced by segment starts and endings, marked with arrows in the figure.[5] Edges of the graph are represented by vertical and horizontal segments. As mentioned before actual time is represented by the vertical axis. The point

[4] Although all–solutions predicates can be depicted using this paradigm, the resulting representation is not natural. A visualization closer to the user's intuition for these predicates needs structures similar to those of Restricted And–parallelism.

[5] These arrows have been added for clarification, and are not part of the visualization paradigm.

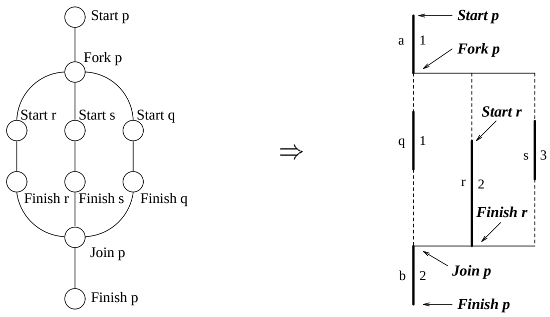

Figure 2: A Dependency graph for Restricted And–parallelism and its visualization

where the FORK happens is marked with a horizontal thin line, whereas parallel tasks are represented as vertical lines. Each vertical thick segment represents an agent working, whereas a vertical dotted line represents a task on which no agent is working. Information associated to the events not explicitly shown on the graph is added as labels (and, if on a color display, as colors). In this case, these labels are intended to mean the clause which created the branch and the agent working on it. These basic ideas will be retained throughout the paper.

Actual parallelism achieved at each point in time can be derived simply by looking at the number of vertical thick lines present at each vertical coordinate—which represents a point in time in the execution—whereas the potential parallelism can be deduced from the total amount of vertical lines. Potential parallelism not being exploited can also be detected. Task suspension is represented by (dashed) interruptions in the vertical thick lines.

3.3 Restricted And–parallelism

Restricted And–parallelism (RAP) refers to the execution of independent goals in the body of a clause using a fork and join paradigm.[6] In this case data dependencies among the goals before and after the parallel execution and the goals executed in parallel can exist. Consider the &–Prolog [HG90] program below, where the "&" operator, in place of the comma operator, stands for And–parallel execution (`a`, `q`, `r`, `s`, and `b` are assumed to be sequential):

```
p:- a, q & r & s, b.
```

A (simplified) dependency graph for this program is depicted in Figure 2, left. In the RAP model there is a JOIN corresponding to each FORK (failures are not seen at this level of abstraction), and FORKS are followed by START_GOALs of the tasks originated. In turn, JOINs are preceded by FINISH_GOALS. In the case of nested FORKs, the corresponding JOINs will appear in reverse order to that of the FORKs. The START_GOAL and FINISH_GOAL events (note that finish can also be caused by ultimate goal failure) must appear balanced by pairs. Under these conditions, a RAP execution can be depicted by a directed acyclic planar graph, where And–parallel executions appear nested.

A possible visualization paradigm for RAP is shown in Figure 2, right. JOIN and FORK events are depicted as horizontal thin lines. The rest of the information is common with the paradigm for Or–parallelism.

[6] Non–restricted Independent And–parallelism allows execution structures which cannot be described by FORK–JOIN events. Such structures are generated, for example, by Conery's or Lin and Kumar's models [Con83, LK88] and by &–Prolog when `wait` is used.

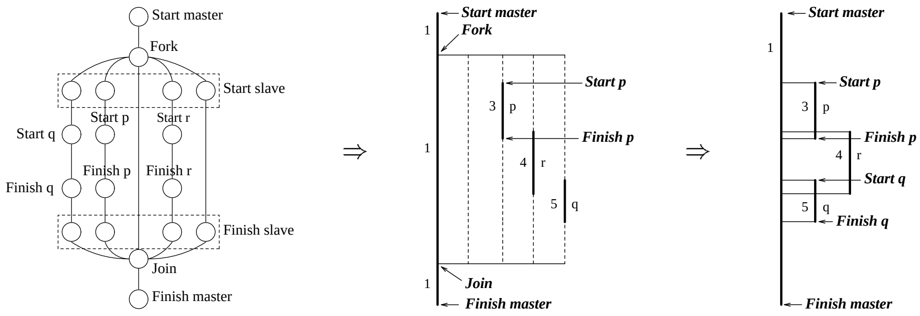

Figure 3: Determinate Dependent And–parallelism and two possible visualizations

3.4 Determinate Dependent And–parallelism

Determinate Dependent And–parallelism (DDAP) performs parallel execution when goals are detected to be determinate at run–time (the parallelism in "committed choice languages" is also of this type). Let us consider, for example, the Andorra–I system [SCWY90]: non–deterministic work is performed during the *non–deterministic phase* by a special agent, called the *master*. When deterministic work becomes available the *deterministic phase* is entered and a number of other agents, the *slaves*, join the master to perform this work in parallel. START_TASK/FINISH_TASK events can be used to reflect the state of each slave. The dependencies are different from those found in RAP: the graph is not always nested. Two new observables can also be defined in order to notify the start and finish of a slave: START_SLAVE and FINISH_SLAVE. A dependency graph corresponding to a possible execution is given in Figure 3, left.

Quite a number of visualization paradigms can be chosen for DDAP. A possible one is that illustrated in Figure 3, middle, in which slaves fork from the master and wait (dashed lines) for determinate work to be available. As soon as a slave starts working on a task, a solid thick line is started. When the slave becomes idle again the line returns to a dashed pattern. This has the advantage of showing every agent even if no work is being performed by it (in fact, this could also be done for RAP). This process–oriented representation is very similar to traditional processor occupation charts. An alternative, task–oriented representation, which appears to be more useful in practice, is to depict the master as a thick vertical line and make slaves appear to split from it when determinate reductions are performed. Apart from producing a less crowded picture an additional reason for this choice will become clear in the following section since it is related to the visualization of the combination of DDAP with Or–parallelism.

3.5 Combinations of the Previous Types of Parallelism

Combinations of the previous schemes are possible. We will mention two of them. Or–parallelism can be combined with DDAP as in Andorra–I [SCWY90], where Or–parallelism is not allowed under DDAP. Several master–slave teams are formed which independently work on different branches. The resulting graph is simply a tree of DDAP graphs, each of its branches being a separate universe. A similar scheme to that of Section 3.2 can be used for the Or–parallel part. Since various masters exist, the FORK and JOIN events must include information about their identity.

If visualization were performed by combining the fork approach for Or–parallelism (Figure 1, middle) and the global fork approach (Figure 3, middle) a tree of agent occupation charts would be obtained. But this visualization scheme is weak if slaves are allowed to migrate from a team to another team, because all slaves would, in principle, belong to every team. A good compromise would be to show only the slaves which are effectively working in a team—and this is what Figure 3, right, shows. Thus, we propose a combination of Figure 3, right and Figure 1, middle.

Combining RAP with Or–parallelism [Kal87, GH92] is somewhat more tricky. Allowing And–parallelism under Or–parallelism is not (conceptually) a problem, since each Or branch represents a separate universe in which And–parallelism evolves independently. The converse situation is more complicated: allowing Or–parallelism inside And–parallelism means that multiple Or branches belonging to different And–parallel goals have to be joined. This leads, in general, to a lattice structure which is not easy to visualize in an intuitive manner. However this lattice structure can be transformed to a tree by taking a "recomputation" view of execution, as presented in [GH92].

3.6 Events *vs.* Time

The paradigms in the previous sections can be assumed to use the vertical axis to represent time. This is useful for many purposes. However, we have also found it very useful to enumerate sequentially the events, respecting their precedence in time, and use this number as the vertical coordinate. This gives a different, interesting view, which is very helpful in the cases in which the structure of the execution is more important than its duration, because in this view fragments of the execution which have high activity in a short time are given more relevance that long periods of sequential execution.

4 From the Paradigm to the Tool: Examples

In this section we present VisAndOr, a tool which implements (an extension of) the visualization paradigms of the preceding sections.

4.1 VisAndOr General Features

VisAndOr shows statically the whole parallel execution of a logic program in a single window. Most of the paradigms presented are supported through a uniform interface. The VisAndOr general layout is the same for all the visualization paradigms. VisAndOr also incorporates a good number of additional features beyond the simple paradigms proposed which are of much help in practice. VisAndOr reads event files which have previously been dumped by the system under study.[7] Figure 14, left, shows a window dump of the current version of VisAndOr and can be used for reference throughout this section.[8]

The topmost area of VisAndOr holds the menus, the messages, and the dialog boxes. The bottommost area shows the type of parallelism being depicted and the name of the current trace file. Visandor offers two simultaneous views of the

[7] In general the events are generated at a low level, so that the programs to be traced do not need to be rewritten in any way. However, the design of VisAndOr poses no restrictions on how the traces are to be generated—i.e. they could be also generated using for example a meta–interpreter or a simulator.

[8] The horizontal thick line in the middle can be ignored since its meaning is totally local to the topic addressed in that picture.

execution: a small window at the top right corner always shows the whole execution (the *summary view*). The central part of VisAndOr shows (a selected part of) the execution in more detail (the *main view*). In both of these views either time or events can be chosen as vertical measurement units. When a trace is loaded VisAndOr scales the execution to exactly fit all of the main view and all of the summary view. In the case of complex executions "condensation" (reduction of detail) is thus automatically performed in both views simply by reduction to screen resolution. The automatic scaling mentioned above can be disabled so that the time scale active before loading the trace persists. This can be used to compare different traces on the same time scale.

The point in time (or in the event scale) to which any point of the picture belongs can be measured accurately with the help of the mouse. In fact, two points can be studied simultaneously simply by clicking and dragging to select a rectangle. The instant corresponding to the uppermost and bottommost sides of the rectangle, as well as the difference between them (measured in actual time or number of events), is then shown over the menus. This allows measuring time differentials and lengths of segments (execution times, wait times, scheduling times, etc.). This rectangle also always appears in the small window. The area selected by the rectangle can can optionally be zoomed out to fit the whole main view in order to perform detailed analysis of the execution, as shown in Figure 11. In that case the rectangle seen in the summary view corresponds to the area being depicted in the main view. Also, when such a zoom is performed, slide bars appear surrounding the central window which allow navigation through the whole picture. This can also be done by dragging the rectangle in the summary view: the main view responds showing the corresponding part of the execution at the current zoom level.

As suggested before (Section 3.1) when running in a color (or grey-scale) display VisAndOr presents additional information contained in the events through colors: for example, each agent is identified by a different color, and the thick lines corresponding to task execution are drawn in the color of the agent executing them at each point in time. This helps greatly in visualizing how scheduling has been performed: for example, when scheduling is being successful at exploiting locality in the task tree, the trace tends to have a clustered distribution of colors, usually a sign of good scheduling. Colors randomly spread all over the execution imply the opposite situation.[9] In a simple black and white display line patterns or numeric identifiers are used. The stack set used for the execution of a task can also be labelled through identifiers.[10] As an additional help in scheduling analysis, a single agent's life–line can be highlighted (this works also on a b/w display). Another way in which VisAndOr presents additional information is by placing icons at the points at which those events occur (as shown in Figure 11, discussed later).

Finally, the VisAndOr window can be dumped in PostScript either to a file or to a printer from within the tool. VisAndOr can also generate idraw–compliant PostScript (idraw was used for example to add the arrows to the figures in this paper).

4.2 VisAndOr by Example

VisAndOr is currently interfaced with the Or–parallel systems Aurora and Muse, with the Independent And–parallel system &–Prolog and with the Determinate De-

[9] This is generally referred to as "tutti-frutti" scheduling, in VisAndOr lingo.

[10] Allowing the separate study of stack sets and agents is very useful in the study of memory management in execution models where they are differentiated.

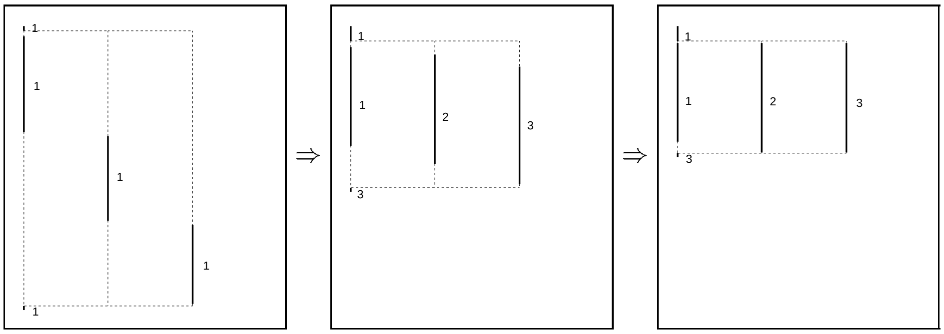

Figure 4: Restricted And–parallelism: sequential and parallel execution

pendent And+Or–parallel system Andorra–I. All these systems can generate traces which VisAndOr is able to understand. We will use a set of such traces to illustrate the usefulness of VisAndOr through examples. Unfortunately, space limitations do not allow presenting the figures at the ideal size for visualization –the size at which they appear on the computer screen– since each figure would occupy a whole page. The reader is kindly asked to bear with the small size of the pictures and encouraged to get a copy of VisAndOr and experiment with it along the lines described here.

Figure 4 visualizes traces from the execution of the following clause in &–Prolog:

```
p:- q & r & s, t.
```

where `q`, `r`, and `s` are sequential. The leftmost picture represents execution by one agent. Time is being used as measurement unit (vertical axis). It is easy to see that there is no actual parallelism and only one task is active at a time: there is only one thick line present at each vertical coordinate and only one color (pattern or label— as in this case—for black and white) for thick lines. On the other hand it is also easy to see the amount of parallelism implicit in the parallel program, and the dependencies present (thick lines can slide up dashed lines through parallel execution, but not cross horizontal lines or other thick lines). It is also easy to get an idea of the amount of work in each task (length of the thick lines). The figure in the middle corresponds to the same program, but executed by three agents. The time scale of the leftmost picture has been retained in order to appreciate the speedup. Delays in the start of some branches can also be observed and can be easily measured with the mouse (measuring the length of the dotted vertical segments right below the FORK segment). Tasks are slightly longer due to parallelism overheads. The amount of parallelism actually exploited at each point in time (number of thick lines at each vertical coordinate) is easy to observe (again, in a color display each task would have a different color, corresponding to the agent working on it, as there are labels in the figure). The rightmost figure represents an ideal execution of this program, with no scheduling delays.[11]

Figure 5 visualizes an execution of the well known matrix multiplication benchmark for two 4x4 matrices, by one agent (leftmost picture) and four agents (middle and rightmost pictures). Figure 6 shows the middle and rightmost pictures of Figure 5 scaled up by the same factor and with agent numbers attached to the tasks. This example illustrates how program transformations which improve parallel execution can be studied with VisAndOr. The initial parallelized clauses (as generated by an old version of the parallelizing compiler or perhaps by a naive user) are the following:

[11] This trace was obtained automatically using the IDRA tool – see Section 7.

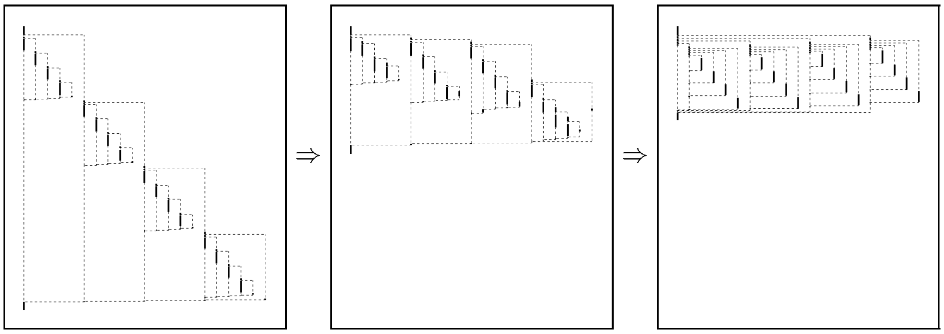

Figure 5: Restricted And–parallelism: nested structure and different scheduling policies

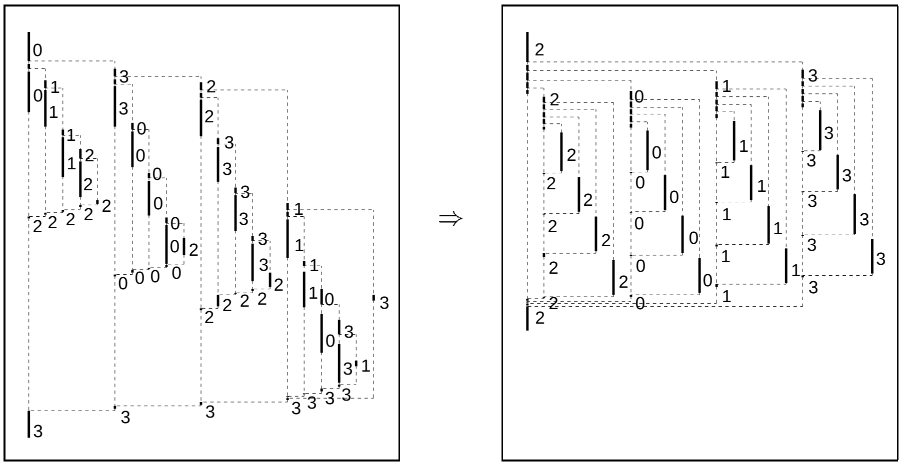

Figure 6: Agent-task mapping

```
matrixmatrix([Vector1|Matrix1], Matrix2, [Vector3|Matrix3]):-
         matrixvector(Matrix2, Vector1, Vector3) &
         matrixmatrix(Matrix1, Matrix2, Matrix3).

matrixvector([Vector1|Matrix1], Vector2, [Scalar|Vector3]):-
         vectorvector(Vector1, Vector2, Scalar) &
         matrixvector(Matrix1, Vector2, Vector3).
```

The vector by vector scalar multiplication is performed sequentially for granularity control reasons. The leftmost picture, although obtained from a sequential execution, clearly shows the parallel task structure. It has a *fractal* nature, as many of the pictures presented by VisAndOr—it is this fractal nature that allows understanding larger executions at a glance by looking at their repetitive structure. The picture in the middle shows that actual scheduling of tasks is only fair—labels, as shown in Figure 6, left, are not very well clustered, and tasks are being started quite slowly. The latter is due to the order of the goals in the parallel conjunctions: in &–Prolog the first goal in a conjunction is normally kept by the agent working on the conjunction while the other one is scheduled for parallel execution. This can be seen in Figure 6 by analyzing the labels: the recursive `matrixvector/3` goals are being picked up by an agent different from the one which created them and the `matrixvector/3` parallel goals are created slowly from left to right, as the

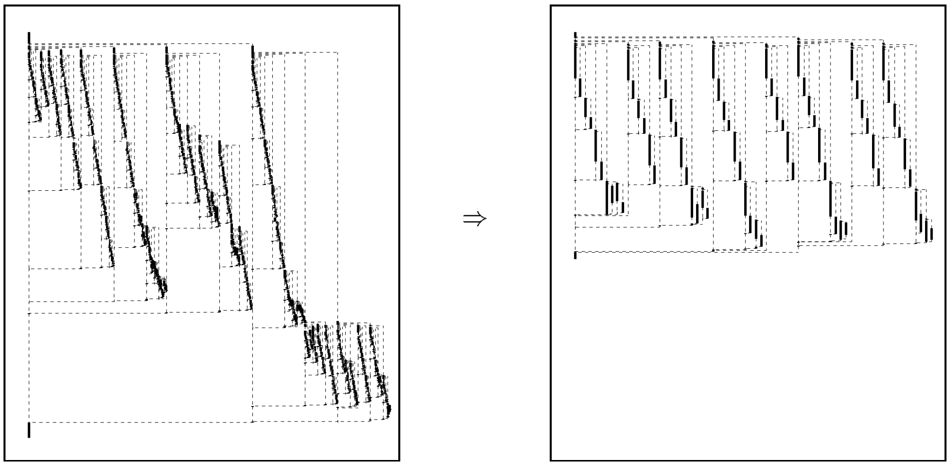

Figure 7: Restricted And–parallelism visualization: granularity control

`matrixvector/3` recursive goals are picked up. Things can be improved by reversing the the order of the goals in the parallel conjunctions so that the agent which is executing a clause picks up the recursive goal, as shown in the rightmost picture (this is actually done automatically by the parallelizer), which also shows very good scheduling. These improvements can be seen to result in better speedups.

Figure 7 shows executions of the the well–known *fibonacci* program in two different situations (from left to right): by eight agents without granularity control, and by eight agents with granularity control. The impact of granularity control on speedup is illustrated. The leftmost figure figure shows an abundance of very small tasks. Such situations can be improved by means of granularity analysis, which tries to find out when parallel execution is *not* desirable, because scheduling costs would be higher than the performance gained. The fewer, larger tasks which appear in the figure on the right are due to such control and can be seen to result in improved speedup. In general, we have found visualization to be of much help in understanding the relative weights of different parts of programs in the whole execution.

Figure 8 shows two executions of the *quicksort* program: on one agent (left) and on four agents (right). The vertical segments before the forks correspond to partition, which in this example is executed sequentially before splitting the two quicksort goals. This happens recursively – the fractal structure is quite evident in this example. The speedup obtained by parallel execution is also clear, as well as the fact that the execution of partition is a sequential bottleneck: in the parallel execution it becomes a larger portion of the total execution time – a graphical example of Amdahl's law. It is interesting to compare the *fibonacci* and *quicksort* executions. Both have repetitive patterns, but the source is somewhat different. *fibonacci* executions show mainly the structure of the algorithm, which is similar to *quicksort* in that a given problem is reduced to two simpler problems. But *quicksort* is completely *data–driven*, and its trace really reflects the distribution of the data in the lists obtained through the lengths of the vertical segments (which, as mentioned before, represent the execution time of each call to partition).

Figure 9 shows a simple Muse trace (of the five queens problem executed on eight agents). Again, time is being used as measurement unit. The structure of the (parallel) search tree is clearly seen. Again, it is also easy to get an idea of the

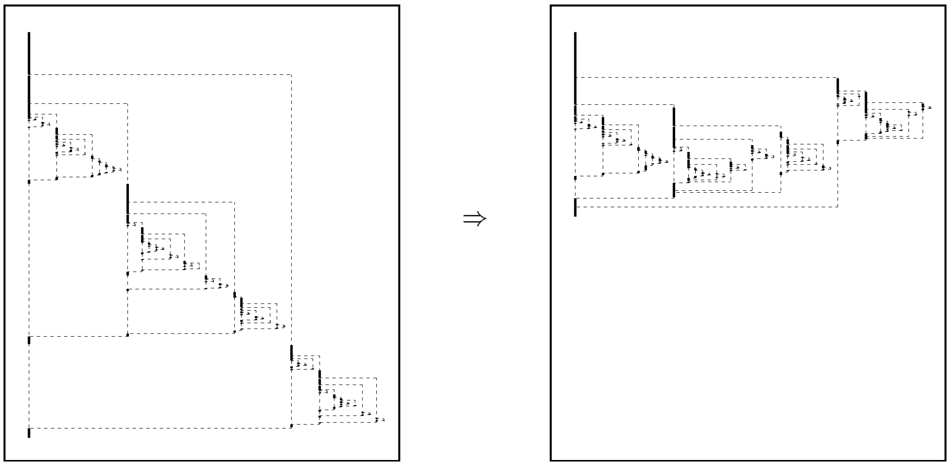

Figure 8: Quicksort, on 1 and 4 agents

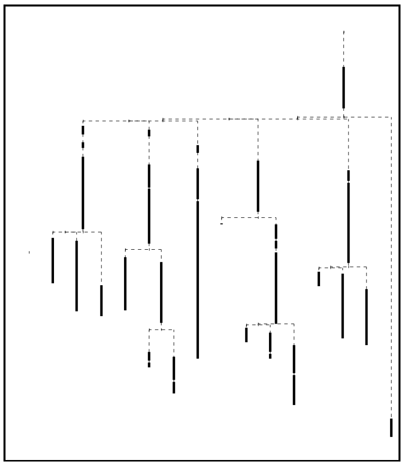

Figure 9: A simple Muse trace

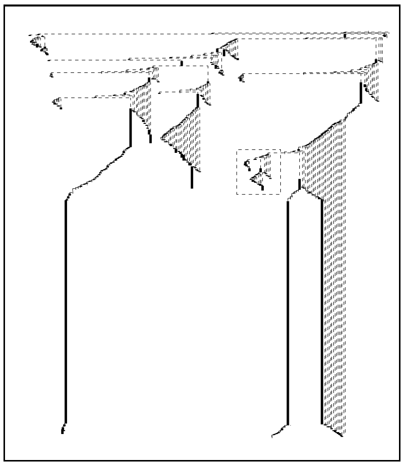

Figure 10: Aurora trace

amount of work in each branch (length of the thick lines), the amount of parallelism made available by the scheduler (number of branches), and the amount of parallelism actually exploited at each point in time (number of thick lines at each vertical coordinate). Delays in the start of some branches can also be observed. Some of these delays are due to scheduler shortcomings while others (as the suspensions and resumptions that appear as dotted vertical lines interrupting thick vertical lines, as stated in the paradigm design) can be due for example to waiting on side-effects. This can be visualized by activating the icons.

Figure 10 shows a somewhat intricate Aurora execution. Still, it is straightforward to identify three main Or–parallel branches which dominate the execution. In this figure, the mouse has been used to select a part of the execution, which is marked by a small dashed rectangle. This part has been zoomed out (Figure 11, where the slide bars also appear). Icons mark points where goals are made public for parallel execution, start, and finish with success.

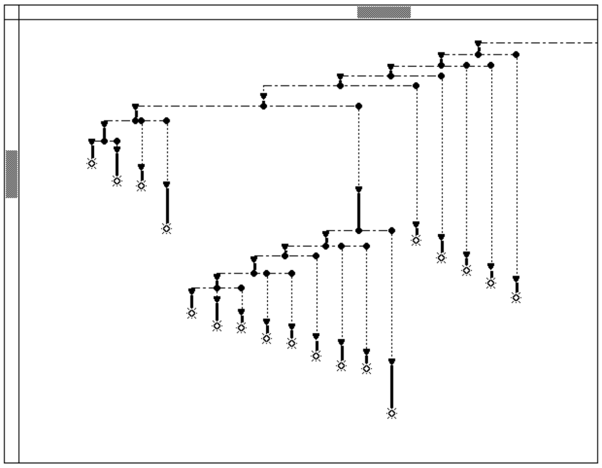

Figure 11: Zoom of Aurora trace

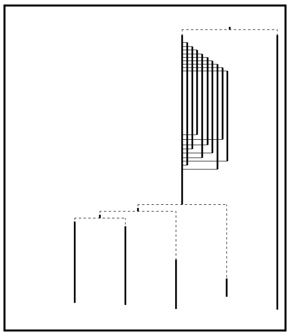

Figure 12: Andorra–I trace

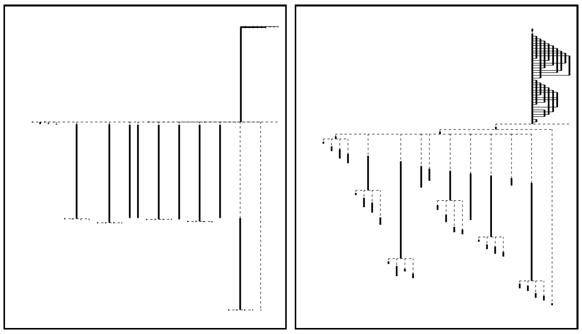

Figure 13: Time versus events in Andorra–I

Figure 12 shows a simple Andorra–I trace. At the top the execution splits into two Or–parallel branches. One of them starts Determinate Dependent And work and more Or–parallelism after this work is finished. The other branch corresponds to a sequential execution. Or–parallel task creation and waiting times are also visible.

Figure 13 shows two views of another Andorra–I execution. The leftmost one corresponds to using time as units in the vertical axis whereas the rightmost one corresponds to using number of events as units. The former shows the real weight of tasks in the execution and the sequential bottlenecks. The latter reveals interesting details about the structure of the execution which were previously hidden since short executions with high parallel activity are now given more relevance than before. This allows detecting problems which otherwise would be very difficult to to perceive.

4.3 ViMust

In joint work at SICS and UPM VisAndOr has been interfaced with another tool based on the notion of event: the Muse Trace Tool (Must) [SS90]. Figure 14 is a snapshot of the resulting tool, which has been given the name of ViMust (the

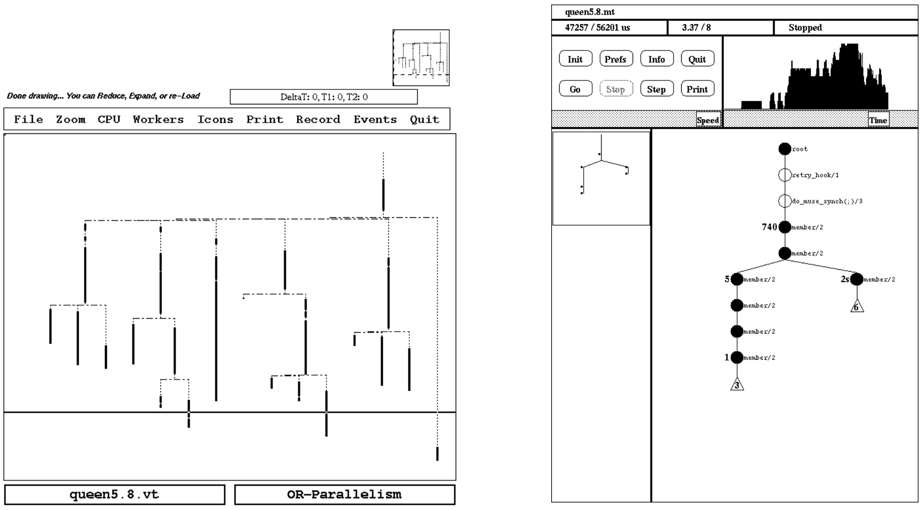

Figure 14: ViMust: Must and VisAndOr working together

problem represented is "queens5" running on eight agents, as in Figure 9). The advantage in combining these tools is that they are quite complementary. Must is constructed along the lines of the original WAMTrace tool [DL87] developed at Argonne National Labs in the context of Aurora. It shows snapshots and animations of Or–parallel executions by presenting at each instant the subtree corresponding to the actual paths being explored in parallel at that point in time, labelled with information about the state of each worker. Thus it shows subgraphs of the whole graph represented by VisAndOr. In the VisAndOr view execution times are clearly represented. Must is strong for example at presenting the movement of agents (workers) in the tree through animation. In ViMust VisAndOr and Must work together by exchanging messages asynchronously. VisAndOr indicates the point in time displayed by Must with a horizontal time line which moves as time advances in the Must animation. Conversely, the line in VisAndOr can be moved with the mouse, and Must presents the snapshot of the execution at that time. In Figure 14 the two branches depicted by Must correspond to the paths that lead to the two thick lines (active branches) intersected by the time-line in visandor. This is an example of the generality and uniformity of event traces as an interface: even though the Must trace format was developed independently it can be trivially translated to VisAndOr trace format. More examples are given in Section 7.

5 Implementation Details and Related Issues

VisAndOr is currently written in C for the X–Window system. The design is quite modular. The actual calls to X drawing routines are kept isolated in a window-system dependent module. This allowed at some point having for example also a "suntools" implementation. The current implementation uses the Xt library and Athena Widgets. They are useful, but somewhat limited and low–level. Taking advantage of the modularity of the design of the tool a richer, "Interviews"–based interface is now being completed.

The graphical objects which form the picture are internally stored in a table. Their coordinates are in a virtual, unlimited space which is mapped to the real screen when a change of scale or representation units is requested. An interesting issue is how zooming is performed. One idea is to redraw the entire picture in the new coordinates and store it in memory. This is what is done automatically by the Athena widgets. This allows fast movement around the picture but requires large amounts of (server) memory and can finally be very slow due to paging. An alternative is to recompute only the part that is being visualized of the zoomed picture. This saves memory but implies slow movement through the picture since this movement implies recalculation.

A problem which was identified in previous versions of VisAndOr was that the simple algorithm to assign the space for the different branches, based on first partitioning the space available evenly at each fork and then reassigning space according to total the number of children, exhausted the numerical accuracy of the computer. This happened in branches that were already indistinguishable in the screen. But errors could in some cases be carried up to higher levels, giving a wrong appearance to the whole picture. A possible solution could have been to give up computing when the branches were too near to be distinguishable on the screen. However, such a solution would still fail if a zoom were requested, and we wanted the algorithm to work in a virtual, unlimited space, independent of the screen's resolution. A solution based on infinite precision arithmetic was discarded as too computationally expensive. A simple algorithm which only needs integer arithmetic was devised which is based (informally) on doing a bottom-up traversal of the tree of events, gathering the number of children below at each step. This has allowed visualizing quite large and complex traces.

Adapting emulators and schedulers to produce VisAndOr traces has proven to be relatively simple in all the cases tried. It is important of course that the gathering of events impact execution as little as possible. To this end events are stored in a table in memory and only dumped to a file or pipe when measurement has finished. Low impacts (close to 1%) can be achieved, although it depends of course on the density of events traced. One problem on some operating systems is accurate and fast measurement of time, which is needed for the timestamps of each event. This is not a problem for example on Sequent machines since a clock with μ-second resolution, mapped to a memory position and which can thus be read in one bus cycle is provided. On some Sun machines this is not the case: a system call needs to be performed which takes a considerable amount of time. If traces from a single agent execution are being generated this effect can be balanced by always subtracting the time spent in the call from the actual time. Another interesting issue is that time stamps (and most timing information given by the parallel logic programming system, as is the case in all the systems used herein) always has to refer to "wall clock" time, since process time is not a useful measure for many purposes in a parallel machine.

6 VisAndOr and Some Other Related Visualization Tools

The Transparent Prolog Machine [EB88] is an interpreter which displays a running trace of a sequential execution of a program, with a pedagogical orientation. It thus not oriented towards visualization of parallelism.

ParaGraph by Aikawa *et al.* [AKK$^+$92] is aimed at tuning the Parallel Inference Machine (PIM). It performs low–level (processor–oriented) and high–level (goal–oriented) profiling, highly tailored to the KL1 language and the PIM.

VISTA [Tic92] is also oriented towards giving visual feedback for committed-choice logic programs. Information about events is displayed radially with time growing outwards from the center. Explicit condensation is performed (if needed). VISTA pictures have shapes reminiscent of a shell, due to the mapping of an event structure into a polar coordinate system. This system is, in some ways, similar to VisAndOr's forerunner, VisiPAL [HN90].

Another tool named Paragraph has been developed by Heath and Etheridge [HE91]. It is aimed at visualizing events in distributed systems. The views it proposes are quite orthogonal to those of VisAndOr and can be seen as complementary to it.

Finally, WAMTrace [DL87] is a visualization tool for Or–parallel Prolog, originally written for ANLWAM, and later adapted for Aurora. Since Must was based on it the description of Must given in previous sections applies also to WAMTrace.

7 Beyond visualization

Visualization is not the only application of VisAndOr's event traces. As an example, these traces are also used by IDRA [FCH92], another tool developed at UPM. Given a sequential or parallel trace IDRA finds out optimal agent allocation and the corresponding speedup for a given number of agents. A new trace corresponding to that scheduling can be generated, which can in turn be visualized with VisAndOr (see Figure 4). IDRA can, for example, compute very accurate ideal speedups from traces of parallelized programs executed on uniprocessors. The speedup obtained with IDRA is a maximum value for the program annotation performed which can be used to evaluate such annotation and to gauge the optimality of an actual parallel execution.

In fact, VisAndOr and IDRA are two examples of a more general approach to the design of tools and interfaces. The part of the a system which is to be studied can be seen as a *partial machine*, its behavior being modeled by *primitives* which would correspond to the observables we have presented as the basis for our visualization paradigm. Giving different semantics to these events, different whole abstract machines are generated which give us different views of the same partial machine.

8 Conclusions and Future Work

We have reported on a tool, VisAndOr, aimed at the visualization of the parallel execution of logic programs. Several visualization paradigms have been developed based on stepwise refinement starting from the definition of basic notions such as events and observables and some precedence relationships among events which hold for the given model of parallelism. The top–down approach followed in the design of the visualization paradigms gives a homogeneous feel to visualizations of quite different systems. The resulting tool has been interfaced with several actual implementations of parallel logic systems, currently &–Prolog, Aurora, Muse, and Andorra–I. We have shown the usefulness of the paradigms and the tool in several areas by means of examples. The tool, and its combination with Must, ViMust, are being used actively for debugging and tuning—with reportedly favorable results—by the developers and programmers of Aurora, Muse, and Andorra–I at SICS and Bristol University. For us VisAndOr has proved an invaluable aid in the development, debugging, and tuning of parallel programs for the &–Prolog system and of the system's parallelizing compiler and run-time system.

It should be pointed out that VisAndOr can also be used to visualize parallel systems quite unrelated to logic programming: any execution model which can be abstracted by means of observables whose dependency structure is understood by VisAndOr can be visualized. One example is the classical cobegin–coend model of parallelism, as used in Parallel Pascal.

As future work we plan to follow two different but complementary directions: one is to augment the paradigm-independent features of the tool, including some ideas present in ParaGraph [HE91]. This may be done by combination as with Must. Another direction is to improve the existing paradigms and design new ones for other forms of parallelism and other combinations. In particular, a 3–D representation for Independent And+Or–parallelism is being currently developed. Also, we are planning on improving the visualization of Dependent And–parallel models in order to more clearly represent the producer–consumer and suspension–resumption relations.

9 Acknowledgments

The authors would like to thank Tony Beaumont, Inês Dutra, and David Warren from Bristol University and Roland Carlsson from SICS for useful discussions and much feedback on VisAndOr. The interfacing of MuseTrace and VisAndOr was done in collaboration with Roland Carlsson of SICS. Very special thanks to Roger Nasr for his VisiPAL implementation to which our current tools owe much. Thanks also to María José Fernández for the implementation of IDRA which has augmented the usefulness of VisAndOr. Also to the members of the CLIP group at UPM for their comments on earlier drafts of this article and their help in the implementation of the tools presented.

References

[AK90] K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*. MIT Press, October 1990.

[AKK+92] Seiichi Aikawa, Mayumi Kamiko, Hideyuki Kubo, Fumiko Matsuzawa, and Takashi Chikayama. Paragraph: A Graphical Tuning Tool for Multiprocessor Systems. In *Proceedings of the Fifth Generation Computer Systems*, pages 286–293. Tokio, ICOT, June 1992.

[CA88] William F. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5(4):361–376, 1988.

[Car90] M. Carlsson. *Design an Implementation of an OR–Parallel Prolog Engine*. PhD thesis, SICS and the Royal Institute of Technology, S-164 28 Kista, Sweden, March 1990.

[Con83] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[CSW88] J. Chassin, J. Syre, and H. Westphal. Implementation of a Parallel Prolog System on a Commercial Multiprocessor. In *Proceedings of Ecai*, pages 278–283, August 1988.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[DL87] T. Disz and E. Lusk. A Graphical Tool for Observing the Behavior of Parallel Logic Programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 46–53, 1987.

[EB88] M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4), 1988.

[FCH92] M. J. Fernández, M. Carro, and M. Hermenegildo. IDeal Resource Allocation (IDRA): A Technique for Computing Accurate Ideal Speedups in Parallel Logic Languages. Technical Report TR Number FIM26.3/AI/92, Computer Science Faculty, Technical University of Madrid, September 1992.

[GH92] G. Gupta and M. Hermenegildo. Recomputation based Implementation of And-Or Parallel Prolog. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 770–782. Institute for New Generation Computer Technology (ICOT), June 1992.

[HE91] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, September 1991.

[HG90] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[HN90] M. Hermenegildo and R. Nasr. A Tool for Visualizing Independent And-parallelism in Logic Programs. Technical report, U. of Madrid (UPM), 1990. Presented at NACLP-90 Workshop on Parallel Logic Programming, Austin, TX.

[Kal87] L. Kale. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Fourth International Conference on Logic Programming*, pages 616–632. Melbourne, Australia, May 1987.

[Kar92] R. Karlsson. *A High Performance OR-Parallel Prolog System.* PhD thesis, SICS and the Royal Institute of Technology, S-164 28 Kista, Sweden, March 1992.

[LK88] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. University of Washington, MIT Press, August 1988.

[Lus88] E. Lusk et. al. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems.* Tokyo, November 1988.

[Nai88] L. Naish. Parallelizing NU-Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. University of Washington, MIT Press, August 1988.

[SCWY90] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, April 1990.

[SS90] J. Sundberg and C. Svensson. MUSE TRACE: A Graphic Tracer for OR–Parallel Prolog. Technical Report T90003, SICS, 1990.

[Tic92] Evan Tick. Visualizing Parallel Logic Programming with VISTA. In *International Conference on Fifth Generation Computer Systems*, pages 934–942. Tokio, ICOT, June 1992.