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 observing 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 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, Determinate Dependent And parallelism, Restricted 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 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 filled 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 use-

[1]The research presented in this paper has been supported in part by ESPRIT projects number 7195 "ACCLAIM" and number 6707 "ParForce".

fulness will be illustrated through examples. Section 5 presents some details of the implementation of VisAndOr and Section 6 compares VisAndOr with other visualizations tools. Section 7 presents 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, 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 paradigm. Finally, common characteristics of the possible graph structures generated should be identified and unifying principles found in order to devise a 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 paradigm 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 (properties of independence do not hold, or the graphical representation is not intuitive or elegant). In these cases the problem has to be tackled from scratch as a whole, and perhaps even different observables will have to be defined in order to arrive at a satisfactory representation.

3 Visualization Paradigms

Our objective is to apply the methodology sketched in the preceding section and develop paradigms for the visualization of parallel models for execution of logic programs. We will focus on the visualization of Restricted And–parallelism (RAP) [DeG84, Her86], Or–parallelism [AK90b, Lus88, CSW88, CA88] and Determinate Dependent And–parallelism (DDAP) [BHW88]. We are also interested in visualizing combinations of these models.

[2]Throughout the paper we use the term "agent" (or worker) to refer to the process, normally mapped on an agent, 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: Common observables for parallel execution of logic programs

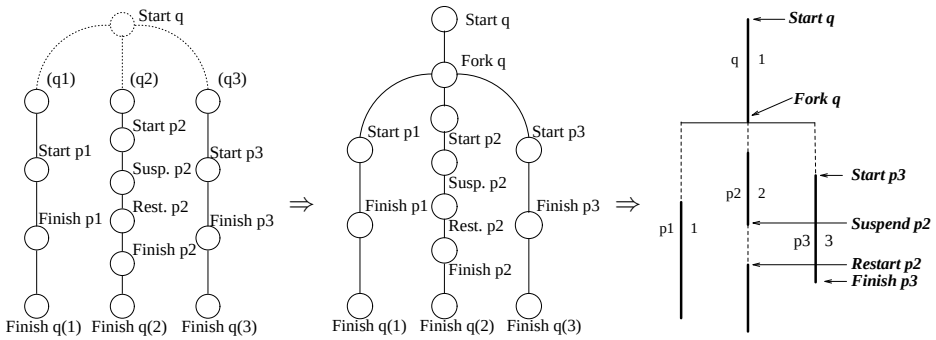

Figure 1: Dependency graphs for Or–parallelism and visualization

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 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 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 time being the main source of precedence, it is not the only coordinate basis that we will use, as will be shown 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 label attached to the proper place in a monochrome display—e.g this paper).

3.2 Or–parallelism

Or–parallel execution corresponds to the parallel execution of different alternatives of a given predicate. It is exploited, for example, in Muse [Kar92, AK90a], Aurora [Lus88, Car90] and the Delphi system [CA88]. 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*. Note that p_2 is suspended at some point and then restarted. In fact, this suspension is probably caused by p_1, in the sense that p_2 is waiting for some built–in to be executed in 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. 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]

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 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 in 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 being resolved in the branch and the agent working on it. These basic ideas will be retained throughout the paper.

Real parallelism achieved can be seen 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), as implemented for example by &–Prolog [HG90, HG91], 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 program below, where the "&" operator, in place of the comma operator, stands for And–parallel execution:

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

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

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

[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.

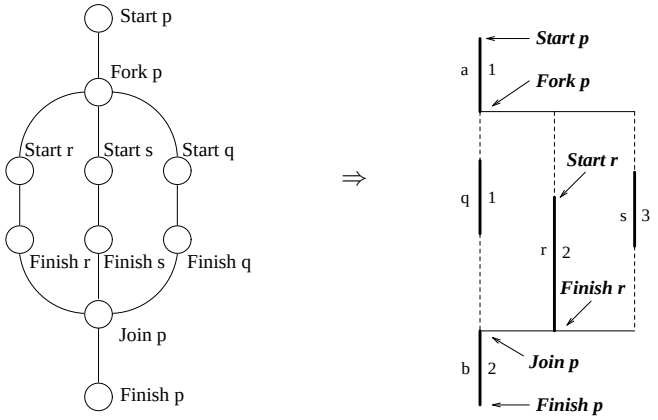

Figure 2: Dependency graph for Restricted And–parallelism and its visualization

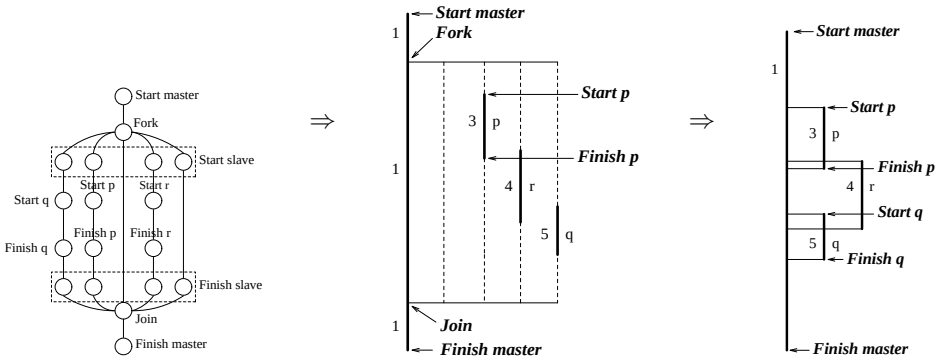

Figure 3: Determinate Dependent And–parallelism and two possible graphical representations

```
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 must 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, the 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.

3.4 Determinate Dependent And–parallelism

Determinate Dependent And–parallelism (DDAP) performs parallel execution when goals are detected to be determinate at run–time. There is a special agent, called the *master*, which is in charge of performing non–deterministic work. When deterministic work becomes available a number of other agents, the *slaves*, perform it in parallel. Two new observables are defined in order to notify the start and finish of a slave: START_SLAVE and FINISH_SLAVE.

The dependencies are different from those found in RAP: only one global fork is done, splitting from the master, regardless of whether there is And–parallel work available or not. In the determinate phase of the execution the master works with the slaves performing determinate reductions, and in the nondeterminate phase the master (alone) does nondeterminate reductions. START_GOAL/FINISH_GOAL events, issued by the slaves, reflect the state of each 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, it becomes a solid thick line. When the task is finished, the slave becomes idle again. This has the advantage of showing every agent even if no work is being performed by it. This process–oriented representation is very similar to the traditional agent occupation charts, but it is . 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 Section 3.5 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 [SCWY91], where Or–parallelism is not allowed under DDAP. Several master–slave teams are formed which independently work on different branches. The resulting graph is merely 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 JOIN and FORK 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 processor 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 is somewhat more tricky. AND_FORKS need to be distinguished from OR_FORKS. 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

In the previous sections we have often assumed 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 simultaneously 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 9, 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. A small window at the right always shows the whole execution. The bottommost area shows the type of parallelism being depicted and the name of the current trace file. The central part of VisAndOr shows the (selected part of the) execution. Time or events can be chosen as vertical measurement units. When a trace is loaded VisAndOr scales the execution to exactly fit the central part of the screen. In the case of complex executions, condensation is thus automatically performed by the screen resolution limitations — in fact, this is what happens, for example, in the small window at the top right corner. The scaling mentioned above can be disabled so that the time scale active before loading the trace remains active. This can be used to compare different executions.

Time or events can be measured accurately with the help of the mouse, simply by clicking and dragging to select a rectangle. The instant corresponding to the uppermost and bottommost edges of the rectangle, as well as the difference between them (measured in actual time or number of events), is shown over the menus. This rectangle, which also appears in the small window, can optionally be zoomed out to perform detailed analysis of the execution (Figure 5). When such a zoom is actually performed, slide bars appear surrounding the central window. Then, navigation through the picture can be accomplished using the slide bars or, alternatively, dragging the rectangle in the small window. The central window immediately responds, showing the corresponding part of the execution at the current zoom level.

VisAndOr can place icons at interesting points in the execution to give additional information about events (Figure 5): for example, success or failure of a branch in an Or–parallel execution. In a color display each agent is depicted in a different color. This helps to appreciate how scheduling has been performed: when scheduling favors locality in the search tree, the trace tends to have unevenly distributed colors; this is usually a better scheduling policy. Colors uniformly spread all around the execution mean the opposite situation. As suggested before (Section 3.1), to perform a more detailed analysis, stack set and agent identifiers can be attached to each sequential task, so that it is possible to follow their history throughout the execution[9]. As an additional help to perform scheduling analysis, a single agent's life–line can be highlighted.

The window can be dumped in PostScript format within VisAndOr either to a file or to a printer. VisAndOr can also generate idraw–compliant PostScript for the pictures being drawn.

VisAndOr is currently interfaced with the Or–parallel systems Aurora and Muse, with the Independent And–parallel system &-Prolog and with the Determinate Dependent And+Or–parallel system Andorra–I. All these systems can generate traces which VisAndOr is able to understand.

[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.

[9] Allowing the separate study of stack sets versus agents is mandatory in execution models where they are not intimately related.

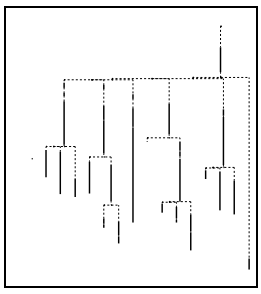

Figure 4: A simple Muse trace

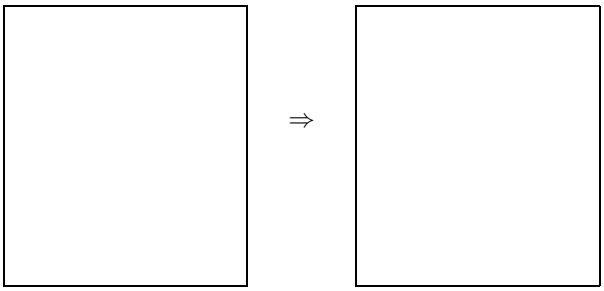

Figure 5: Aurora trace and zoom

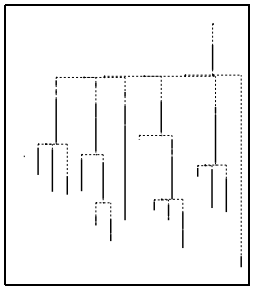

Figure 6: A simple Muse trace

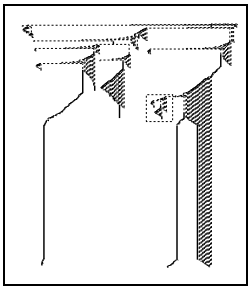

Figure 7: Aurora trace

4.2 VisAndOr by Example

Figure 6 shows a simple Muse trace. Time is being used as measurement unit. Branches suspension and resuming are shown as dotted vertical lines interrupting thick vertical lines, as stated in the paradigm design. There are also delays in the starting of some branches, which can be attributed (since no more information is available) to scheduling duties or perhaps to the need of waiting for built–ins in branches at the left. Another possibility is that no resources (processors) are available at this moment. This is not the case, quite obviously, but in more complicated executions realizing this is difficult. Assigning a different color to each processor greatly helps detecting such phenomena.

Figure 7 shows the visual representation of a somewhat intricate Aurora execution. Even without the actual program code, it is straightforward to realize that there are three Or–parallel branches which dominate the execution. In this figure, a dashed rectangle selects a part of the execution. This part is blown up in Figure 8. Slide bars, which can be used to navigate through the execution, appear surrounding the execution. Icons mark points where goals are made public for parallel execution, start and finish with success[10].

VisAndOr has been successfully interfaced with another tool based on the notion of event: the Muse Trace Tool (MUST) [SS90]. MUST has been constructed along the lines of the original WAM–Trace tool [DL87] developed at Argonne National Labs in the context of Aurora. MUST shows snapshots of Muse executions as well as animations of such executions. The tree shown by MUST corresponds to the actual path being explored in parallel (thus it shows subgraphs of the whole graph represented by VisAndOr), and contains information about the state of each worker.

[10]In fact, only success icons appear in Muse and Aurora traces, since the events are issued by the scheduler, which is unaware of whether a given goal failed or succeeded.

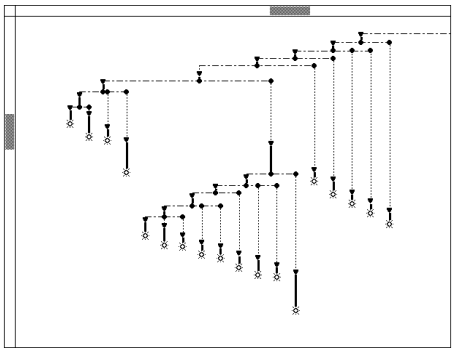

Figure 8: Zoom of Aurora trace

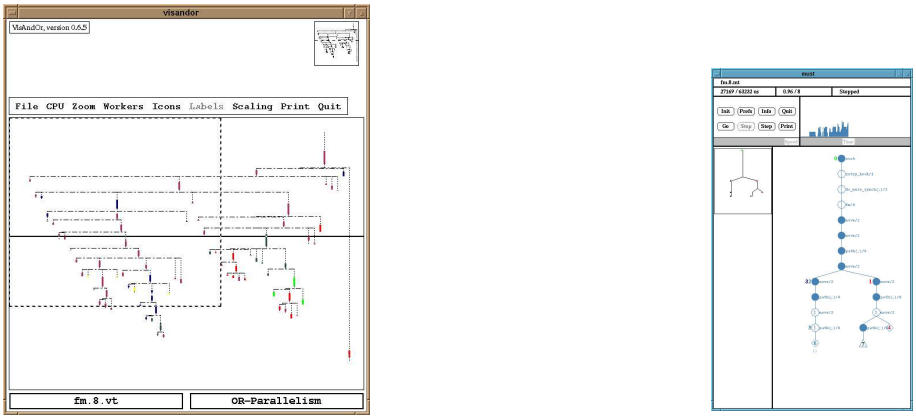

Figure 9: ViMust: Must and VisAndOr working together

VisAndOr and MUST can work together through a "standard input–output" based protocol which allows each one to send messages to the other asynchronously. VisAndOr indicates the point displayed by MUST with a horizontal line and answers to the messages sent by MUST to move the line. Conversely, the bar can be moved from VisAndOr with the mouse, and MUST receives the appropriate message to show a snapshot of the execution as required. Figure 9 is a snapshot of the resulting tool which has shown to be of great use at SICS. The resulting system has been given the name of ViMust. This is an example of how the use of events as an interface effectively helps integration of tools and the study of remote systems: Must traces were completely different from VisAndOr traces, and were translated to the desired format by a simple program.

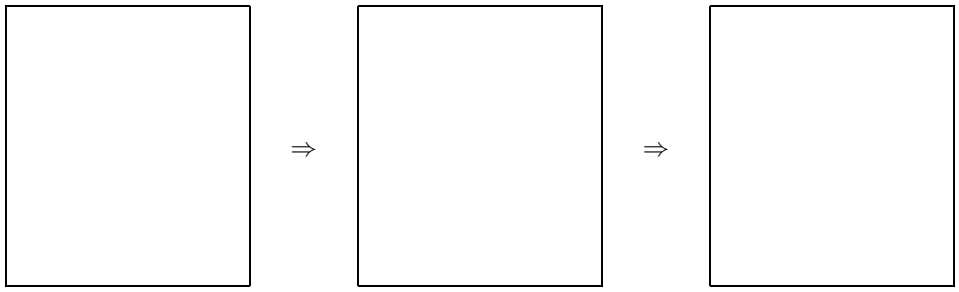

Figure 10: Restricted And–parallelism: sequential and parallel execution

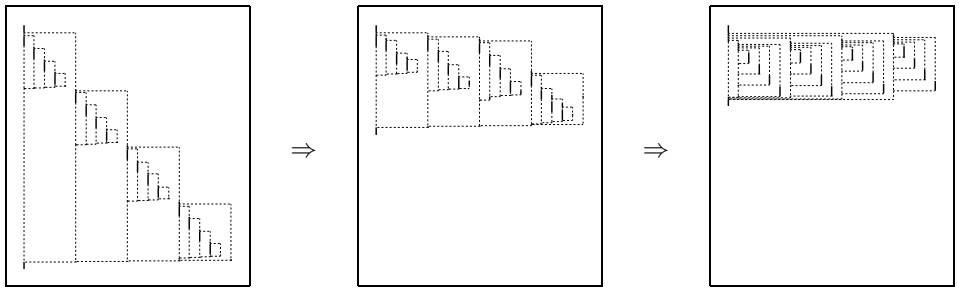

Figure 11: Restricted And–parallelism: nested structure and different scheduling policies

In Figure 10 simple traces of a predicate with a three–branched And FORK are shown. The leftmost picture represents the predicate executed in one agent, but scheduled for parallel execution. Only one task is active at a time: there is only one thick line at each vertical coordinate. The figure in the middle corresponds to the same program, but executed on three agents; the time scale of the leftmost picture has been retained, so that the benefits of parallel execution in terms of time can be easily seen. Each task has a different scheduling time, as shown by the different length of the dotted vertical segments right below the FORK segment. The rightmost figure represents an ideal execution of this program, where scheduling delays have been dropped away to zero[11].

[FCH92].

Figure 11 shows a 4x4 matrix multiplication, in one agent (leftmost picture) and four agents (middle and rightmost pictures). The recursive clauses with And–parallelism of the actual &–Prolog code look like this:

```
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 has been performed sequentially—a sort of granularity control. This example will show how scheduling can be studied with VisAndOr. The leftmost picture has been executed in only one agent; the structure is clearly visible. The picture in the middle shows the same program, executed in four agents. Since the `matrix_vector/3` goals are

[11] This trace was automatically obtained using the IDRA tool.

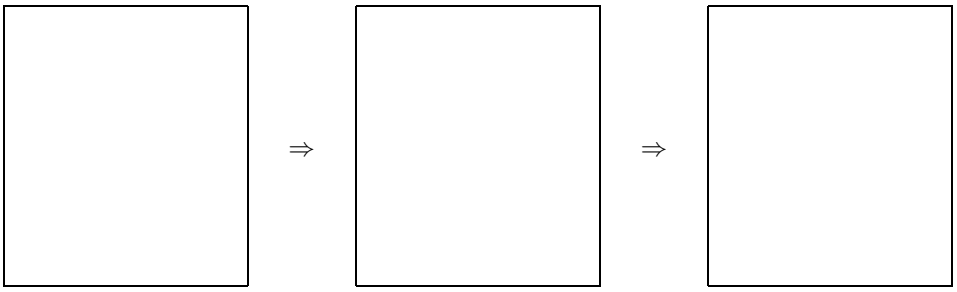

Figure 12: Restricted And–parallelism visualization: granularity control

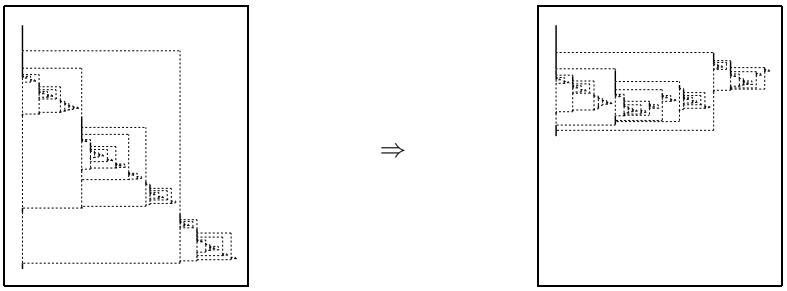

Figure 13: Quicksort, on 1 and 4 processors

executed in a stack set different than the one they were created in (`matrix_vector/3` goals are created from left to right, as recursive goals are picked up), we can infer that recursion steps are, in this example, executed by different agents. A better scheduling, in which the agent which is executing a clause also picks up the recursive goal, is shown in the rightmost picture. This last execution runs about three times as faster as the sequential one. An utopian execution would achieve a speedup of four, but there are clearly visible sequential delays, imposed by scheduling and recursion steps, which impede this performance. A programmer debugging and tuning a scheduler would greatly appreciate this kind of feedback, which shows a high–level intuitive view of the dynamic nature of scheduling, abstracting the concrete algorithm to concentrate on its actual behavior.

Figure 12 shows executions of the the well–known *fibonacci* program in three different situations. From left to right, in only one agent, on eight agents without granularity control, and in eight agents with granularity control. This figure shows the tremendous impact of (a) parallel execution, and (b) granularity control. Whereas in the leftmost figure there is only one sequential execution thread, the figure in the middle shows various (up to eight) parallel tasks, but the visualization is somewhat confuse, tending to fractal[12].

This sort of executions can be cleaned up by means of granularity analysis, which tries to find out when parallel execution is *not* desirable, because scheduling costs would be bigger than the performance gained. Granularity analysis' target is to find the point where sequential execution is cheaper. By adding granularity control, so that small tasks are not scheduled for parallel execution, a remarkable speedup is obtained in Figure 12, right, with respect to the *naïve* parallel execution. The structure of the granularity–controlled program is much more clear than the previous one, and its execution is about twice as fast. This is a very easy example. In more complicated examples, it is difficult to perceive the impact of granularity parameters, and visualization is of much help to understand the interrelations of the different parts of the programs and their actual relative

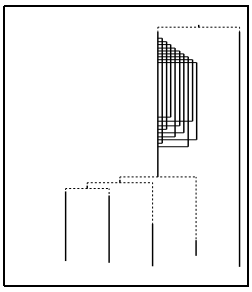

Figure 14: Andorra–I trace

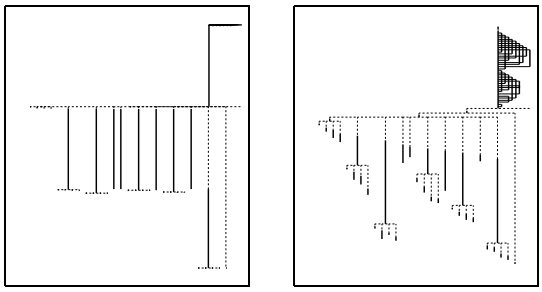

Figure 15: Time versus events in Andorra–I

weight in the whole execution.

Figure 13 shows two executions of the *quicksort* program: on one processor, at left, and on four processors, at right. Apart from the speedup obtained by parallel execution, the fractal layout is evident in this example. It is interesting to compare the *fibonacci* and *quicksort* executions. Both two 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 initial data distribution[13].

Figure 14, shows a simple Andorra–I trace. At the top the execution split in two Or–parallel branches; one of them has, in turn, Determinate Dependent And work and more Or–parallelism after this work is finished. The other branch corresponds to a sequential execution. Or–parallel tasks' birth is easy to appreciate, as well as waiting times before actual work. As mentioned before, two–level scheduling in Andorra–I can be visualized with VisAndOr by the colors mechanism and slaves visualization paradigm. The VisAndOr visualization paradigm can be used to understand intuitively the impact of checking determinacy conditions at run–time (as Andorra–I does): events could be associated with the start and end of this checking and signaled, for example, with icons. Its relative importance (in terms of execution time) versus the parallelism achieved can be evaluated by simply having a look at the pictures.

Figure 15 shows two views of the same execution. The leftmost one corresponds to the vision in the *time space*, whereas the rightmost one corresponds to the same execution in the *events space*. Valuable details about the structure of the execution which were previously hidden appear now: short executions with high parallel activity are now given more relevance than before, so allowing potential scheduling/correctness problems which otherwise would be very difficult to appreciate to be perceived.

5 Implementation Details and Related Problems

VisAndOr is written in C and runs under the X–Window environment. It has been constructed using the Xt library and Athena Widgets. They have been found to be useful, but sometimes the lack of flexibility when defining graphical objects became a problem. The inner structure is quite modular. Each feature is accessed through a call back routine activated by the corresponding button or menu item. The execution events are internally stored in a virtual space which is mapped to the real screen when a change of scale or representation units is requested. This can lead to problems when zooming small regions, due to the lack of virtual memory in some X–Window servers.

[12]The fractal layout is a characteristic of many RAP programs, due to the recursive character of Prolog execution. Considerations of space do not allow us to show more elaborated examples.

[13]Of course, modulo scheduling—we can assume that an unbound number of processors are available.

One problem which was identified in previous versions of VisAndOr was that the algorithm to assign the space for the different branches exhausted the numerical accuracy of the computer. Of course, 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 separated in the screen. This was not a good solution, since, on one hand, we wanted the algorithm to work in a virtual, unlimited space, unaware of the screen's resolution, and, on the other hand, this would finally fail if a zoom were requested. A solution based on infinite precision arithmetic was discarded as too computationally expensive. Fortunately, a quick algorithm which did not employ at all floating point arithmetic was devised and implemented, which allow us to study traces much more complex than it had been possible previously, so assessing the effectiveness of the tool in real cases.

The difficulty of adapting the emulators and schedulers to dump traces is not, in general, very high, although in some systems, accurate measurements of time became a problem. In particular the &–Prolog implementation on Sun Sparc posed a problem which can also happen in other architectures. Time has to be consulted when an event is to be recorded. Unlike machines like the Sequent Balance, in which the time is stored in a memory position, so that consulting it was very quick, Sun OS needs a system call to be performed. This system call could take a sizable proportion of the total process time, thus seriously impacting time stamps accuracy. This effect was balanced by taking into account how long each system call lasted and subtracting it from the actual time. Time stamps always refer to "wall clock" time, since process time cannot be used to establish a precedence among events generated by different parallel processes.

6 VisAndOr and 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. No program transformation is needed, and it has many good details, such as being able to show coarse–grained or fine–grained views, and having a good treatment of meta–predicates. This, together with a careful graphic design, makes it a very good tool to understand and study the sequential Prolog execution model.

The ParaGraph tool[14] by Aikawa *et al.* [AKK+92] is aimed at tuning the Parallel Inference Machine (PIM) [GSN+88]. It is aimed to perform low–level (processor–oriented) and high–level (goal–oriented) profiling. ParaGraph gathers data during program execution using primitives of the KL1 [UC91] language. It is not a general purpose tool, but rather a highly machine and language dependent tool.

The WAMTrace tool [DL87] is a visualization tool for Or parallel full Prolog, originally written for ANLWAM, and later used for Aurora. This tool shows an animation of the parallel search tree, with different icons being extensively used to reflect different workers' states and node types. The main difference with VisAndOris that VisAndOroffers a static and much more schematic view, conveying the whole execution. A similar viewpoint is offered by MUST. The extensively use of animated icons provide a dynamic stream information; this approach could be of interest to represent suspension in DAP and in Constraint Logic Languages.

Finally, the VISTA tool [Tic92] intends to give effective visual feedback to a programmer tuning concurrent logics programs. The program's logical procedure–invocation is displayed radially from the root with explicit condensation (if this is needed). The drawings obtained with VISTA have the peculiar shape of a snail shell, due to the mapping of the (parallel) search tree into a polar coordinate system. This system, which represents Deterministic Dependent And–parallelism, is, in some ways, similar to VisAndOr's forerunner, VisiPal[HN90].

[14]There are two different visualization tools with the same name: the one we are currently referring to and the ParaGraph tool by Heath and Etheridge, described in [HE91], which are very different and must not be confused.

7 Beyond visualization

Visualization is not the only topic in which the event driven scheme is useful. Dumping data tailored to different needs is a flexible way of interfacing tools with engines, each tool possibly assigning a different semantics to the same set of events.

The interface between the engine being traced and the tool, dictated by the format of the events, allows event files to be generated remotely or transformed to simulate special execution conditions.

As an example, the events currently recorded for VisAndOr can be directly used for purposes other than visualization. IDRA, a tool already developed in our working group uses as input the same traces that VisAndOr does, but with a different purpose. IDRA finds out the optimal agent allocation and the corresponding speedup for a given parallel execution and a given number of agents. A new trace corresponding to that scheduling can be generated, which can in turn be visualized with VisAndOr. The speedup obtained with this trace, compared with the one obtained in a real parallel system, is a valuable indication of the quality of the actual scheduling algorithm.

VisAndOr and IDRA are two examples of a more general approach to the design of tools and interfaces, in which it is decided which part of a system is interesting to study. This part can be seen as a *partial machine*, being its behavior modeled with *primitives* which would correspond to the observables we have presented as the basis for our visualization paradigm. Giving different semantics to these events, different 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 depicting the parallel execution of logic programs. The tool's visualization paradigm has been developed following a methodology which starts by determining at what level we want to visualize the model, what the basic elements are at this level, and which dependencies hold among them. The tool has been interfaced with a few implementations of parallel logic processing paradigms, currently &–Prolog, Aurora, Muse and Andorra–I, and it is being actively used at Bristol, SICS and Madrid. It has been reported its usefulness as debugging and tuning tool for parallel logic systems. The top–down approach followed in the design of the visualization paradigm makes the tool homogeneous in the representation and in the user interface. The interface with the execution platform is formally defined up to the point of being comparable to an abstract machine's language, so that different semantics can be used to highlight different characteristics or to transform the execution skeleton to simulate different environments. This is important, because it allows to easily extend and adapt the tool to visualize different paradigms and to easily interface it with other different tools designed under similar principles. This has been done, and the resulting tools improved their usefulness.

Our future work will follow two different but complementary directions: on one hand, the implementation of the tool is to be improved with new additional features which do not require designing new paradigms, in the style of the ParaGraph tool [HE91] (processor utilization, real parallelism achieved versus potential parallelism present...). On the other hand, new conceptual representations will be designed to support several other forms of parallelism and their combinations. A 3–D representation for Independent And+Or–parallelism is under study. Dependent And–parallel execution models need the producer–consumer relation and the suspension of goals to be visualized in a clear manner; this is an issue in which much can still be done.

9 Acknowledgments

The authors would like to thank Tony Beaumont, Inês Dutra, and David Warren from U. of Bristol and Roland Carlsson from SICS, for useful discussions on visualization and feedback on our tools. The connection 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. The implementation of IDRA was done by María José Fernández, which was of much help when comparing real executions with ideal ones using VisAndOr. Also to the members of the CLIP group at U.P.M. for their help on earlier drafts of this article and during the implementation of the tools herein presented.

References

[AK90a] K. A. M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1990. Vol. 19, No. 6, pp. 445–475.

[AK90b] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. 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 Intl. Conf. on Fifth Generation Computer Systems*, pages 286–293. Tokio, ICOT, June 1992.

[BHW88] P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog—The Language and Application in Distributed Simulation. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[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. IEEE Computer Society Press, 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. Fernández, M. Carro, and M. Hermenegildo. IDRA (IDeal Resource Allocation): A Tool for Computing Ideal Speedups. Technical Report FIM26.3/AI/92, School of Computer Science, Technical University of Madrid, September 1992. Presented at the ICLP'94 Post Conference Workshop on Parallel and Data Parallel Execution of Logic Programs.

[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.

[GSN+88] A. Goto, M. Sato, N. Nakajima, K. Taki, and A. Matsumoto. Overview of the Parallel Inference Machine (PIM). In *International Conference on Fifth Generation Computer Systems*. ICOT, 1988.

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

[Her86] M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[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.

[HG91] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HN90] M. Hermenegildo and R. I. Nasr. A Tool for Visualizing Independent And-parallelism in Logic Programs. Technical Report CLIP1/90.0, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, 1990. Presented at the NACLP-90 Workshop on Parallel Logic Programming, Austin, TX.

[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. 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.

[SCWY91] 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 PPOPP*, pages 83–93. ACM, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[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.

[UC91] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, December 1991.