IDRA (IDeal Resource Allocation): A Tool for Computing Ideal Speedups

M. J. Fernández **M. Carro** **M. Hermenegildo**

{mjf, mcarro, herme}@dia.fi.upm.es

Facultad de Informática
Universidad Politécnica de Madrid (UPM)
Boadilla del Monte, Madrid 28660—Spain

Abstract

Performance studies of actual parallel systems usually tend to concentrate on the effectiveness of a given implementation. This is often done in the absolute, without quantitave reference to the potential parallelism contained in the programs from the point of view of the execution paradigm. We feel that studying the parallelism inherent to the programs is interesting, as it gives information about the best possible behavior of any implementation and thus allows contrasting the results obtained.

We propose a method for obtaining ideal speedups for programs through a combination of sequential or parallel execution and simulation, and the algorithms that allow implementing the method. Our approach is novel and, we argue, more accurate than previously proposed methods, in that a crucial part of the data – the execution times of tasks – is obtained from actual executions, while speedup is computed by simulation. This allows obtaining speedup (and other) data under controlled and ideal assumptions regarding issues such as number of processor, scheduling algorithm and overheads, etc. The results obtained can be used for example to evaluate the ideal parallelism that a program contains for a given model of execution and to compare such "perfect" parallelism to that obtained by a given implementation of that model.

We also present a tool, *IDRA*, which implements the proposed method, and results obtained with *IDRA* for benchmark programs, which are then compared with those obtained in actual executions on real parallel systems.

1 Introduction

In recent years a number of parallel implementations of logic programming languages, and, in particular, of Prolog, have been proposed (some examples are [HG90, AK90, SCWY90, She92, Lus90]). Relatively extensive studies have been performed regarding the performance of these systems. However, these studies generally report only the absolute data obtained in the experiments including at most a comparison with other actual systems implementing the same paradigm. This is understandable and appropriate in that usually what these studies try to asses is the effectiveness of a given implementation against state–of–the–art sequential Prolog implementations or against similar parallel systems.

In this paper, and in line with [SH91], we pose and try to answer a different question: given a (parallel) execution paradigm, how large is the maximum benefit that can be obtained from executing a program in parallel in a system designed according to that paradigm? What are the resources (for example, processors) needed to exploit all parallelism available in a program? (we will refer to this as "maximum parallelism") How much parallelism can be ideally exploited for a given set of resources (e.g. a given number of processors)? (we will refer to this as "ideal parallelism"). The answers to these questions can be very useful in order to evaluate actual implementations, or even parts of them, such as, for example, parallelizing compilers. However, it is clear that such answers cannot be obtained from actual implementations, either because of limitations of the implementation itself or because of limitations of the underlying machinery, such as, for example, the number of processors or the available memory. It appears that any approach for obtaining such an answer has to resort to a greater or lesser extent to simulations.

There has been some previous work in the area of ideal parallel performance determination through simulation, in particular, the work of Shen [SH91] and Sehr [SK92]. These approaches are similar in spirit and objective to ours, but differ in the approach (and the results).

In [SH91] a method is proposed for the evaluation of potential parallelism. The program is first executed by a high–level meta-interpreter/simulator which computes ideal speedups for independent and–parallelism, or–parallelism, and combinations thereof. Such speedups can be obtained for different numbers of processors.

This work is interesting, firstly in that it proposed the idea of obtaining ideal performance data through simulations in order to be able to evaluate the performance of actual systems by contrasting them with this ideal and, second, because it provides ideal speedup data for a good number of programs. However, the simulator proposed does suffer from some drawbacks. The first one is that all calculations are performed using as time unit a resolution step – i.e. all resolution steps are approximated as taking the same amount of time. This approximation makes the simulation either conservative or optimistic in programs with (respectively) small or large head unifications. To somewhat compensate for this, and to simulate actual overheads in the machine, extra time can be added at the start and end of each task. The second drawback is that the meta-interpretive method used for running the programs limits the size of the executions which can be studied due to the time and memory consumption implied.

In [SK92] a different approach was used, in order to overcome the limitations of the method presented above. The Prolog program is instrumented to count the number of WAM instructions executed at each point, assuming a constant cost for each WAM instruction. Only "maximal" speedup is provided. Or–parallel execution is simulated by detecting the critical (longest) path and comparing the length of this path with the sequential execution length. Independent and-parallel execution is handled in a similar way by explicitly taking care of the dependencies in the program. Although this method can be more accurate than that of [SH91] it also has some drawbacks. One is the fact mentioned above that only maximal speedups are computed, although this could presumably be solved with a back-end implementing scheduling algorithms such as the ones that we will present. Other is that the type of instrumentation performed on the source code does not allow taking control instructions into account. Also, a good knowledge of the particular compiler being used is needed in order to mimic its encoding of clauses. Furthermore, many WAM instructions take different amounts of time depending on the actual variable bindings appearing at run-time, and this would be costly and complicated to take into account. Finally, the problem of being able to simulate large problems is only solved in part by this approach, since running the transformed programs involves non-trivial overheads over the original ones.

The approach that we propose tries to overcome the limitations of previous approaches by using precise timing information, rather than approximations, and allowing gathering information for much larger executions. We do that by placing the splitting point between actual execution and simulation at a different location: sequential tasks are not simulated or transformed but rather executed directly in real systems. Timing data is gathered, only at the minimal number of points, by a modified Prolog implementation. The part that is simulated regards the possible (alternative) schedulings of those sequential tasks (while respecting the precedences among the tasks). We argue that this allows obtaining more precise data when compared to previous methods. The modification to the Prolog implementation is minimal and also data gathering has negligible effects on execution time.

The paper is structured as follows: Section 2 describes more in depth our approach and the techniques used in its implementation. Sections 3 and 4 show how the maximum and ideal parallelism are calculated. In Section 5 an overview of *IDRA*, the actual tool, is given. Section 6 contains examples of simulations made using *IDRA* and comparisons of actual implementations with the results of the simulation.

2 Parallelism and Trace Files

As we said before, we want to simulate alternative schedulings of parallel executions. To do this, we use a description of the execution which contains the relationships and dependencies which hold among the tasks (used to simulate new correct schedulings, i.e., executions where the precedence relationships are met), and the length (in time) of each task. This description can be produced by executions in actual implementations (not necessarily parallel ones: only the description of the concurrency in the execution and each task's length must appear, the parallelism among tasks being introduced by means of

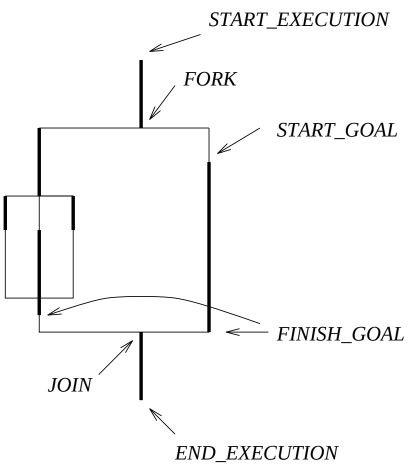

Figure 1: And–parallel execution

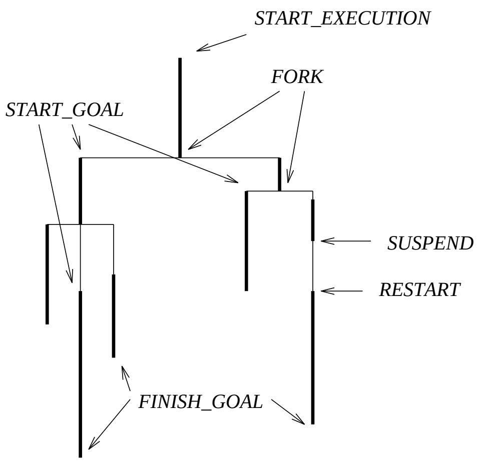

Figure 2: Or–parallel execution

Node	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
FORK	Execution splits in several branches
JOIN	Different branches join
SUSPEND	A task is suspended
RESTART	A task is restarted

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

the simulation) or even using other high–level simulators able to produce information about dependencies in the program and an estimation of the (relative) cost of executing each sequential task.

Among the information we can extract from these descriptions, the following may be of interest:

- Maximum parallelism: this corresponds to the parallelism obtained with an unbound number of processors, assuming no scheduling overheads.
- Ideal parallelism: this corresponds to the speedup ideally attainable with a fixed number of processors. The tasks–processors mapping here decides the actual speedups attained. Optimal scheduling algorithms and currently implemented algorithms are clear candidates to be studied.

Maximum parallelism is useful in order to find out the absolute maximum performance of a program. This would serve to compare different programs: for example, different parallelizations/sequentializations of a given program (i.e., when different annotators [MH90] for parallelism are being used) or different parallel algorithms proposed for a given problem. Ideal parallelism is useful in order to compare a given implementation against its ideal behavior for a given number of processors. This will allow, for example, checking how the performance evolves with an increasing number of processors.

In the following sections we will give a small review of or– and restricted independent and–parallelism, before we focus on the structure of the execution description and how it is used to create an execution graph which describes the execution in a more tractable manner.

2.1 The Description of the Execution

The descriptions of the executions are stored in the form of *traces*, which are series of *events*. These events are gathered at run–time by the system under study. The events reflect *observables* (interesting points in the execution), and allow the reconstruction of a skeleton of the parallel execution. The types of events used, along with a brief description, are shown in Table 1. Each event has enough information to establish the dependencies with other events from the same execution and to know details of the sequential tasks in the computation. Figures 1 and 2 show, respectively, a representation of an and–parallel and an or–parallel execution, with some events marked at point where they occur.

2.2 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.[1] In this case dependencies exist among the goals before and after the parallel execution and the goals executed in parallel. Consider the *&–Prolog* [HG90] program below, where the "&" operator, in place of the comma operator, stands for and–parallel execution (`a` ... `g` are assumed to be sequential):

```
main:- a, c & b, g.
c:- d & e & f.
```

A (simplified) dependency graph for this program is depicted in Figure 1. 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.

2.3 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, each alternative does depend on the fork that creates it. In fact, additional dependencies arise in real systems due to the particular way in which common parts of alternatives are shared and due to side–effects. Consider for example the following program which has three alternatives for predicates `p` and `q`:

```
main:- p.
main:- q.          q:- ...
p:- ...            q:- ...
p:- ...            q:- ...
p:- ...
```

A possible graph depicting an execution of this predicate is the one shown in Figure 2. Note that the rightmost branch in the execution is suspended at some point and then restarted. In fact, this suspension is probably caused by its sibling, because a side–effect predicate or a cut would impose a serialization of the execution. One common important feature of the or–parallel execution is that branches do not join. In terms of dependencies among events, FORKs do not need to be balanced by JOINs. The resulting graph is thus a tree.[2] We are assuming that `p` and `q`'s alternatives are sequential. Otherwise a similar representation would be recursively applied.

[1] 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.

[2] Although all–solutions predicates can be depicted using this paradigm, the resulting representation is not natural. A

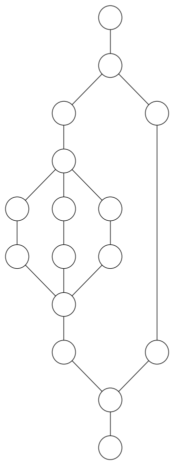

Figure 3: Execution graph, and–parallelism

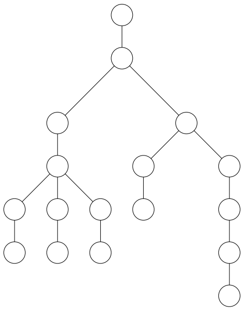

Figure 4: Execution graph, or–parallelism

2.4 The Execution Graph

Traces are converted into *execution graphs*, which are used by the simulator as its first structure. An execution graph is a directed weighted graph $G(X, U, T)$ where:

$X = \{x_0, x_1, \ldots, x_{n-1}\}$ is a set of nodes

$U = \{u_{i,j}, 0 \leq i < j < n\}$ is the set of edges connecting node x_i to node x_j.

$T = \{t_{i,j}, 0 \leq i < j < n\}$ is the set of weights corresponding to each $u_{i,j}$.

In the execution graph each node corresponds to an event, and each edge to a dependency between the events the nodes represent. Each node $x \in X$ has an associated type $type\_of(x)$ and the point in time in which the corresponding event has occured, $time(x)$. The weight in each edge represents the time elapsed between the events represented by the nodes that edge connects. Among these edges we distinguish two types: those which represent sequential execution and those which represent delays introduced by scheduling. The edges fall, thus, in one of the following two categories:

Scheduling: FORK to START_GOAL, FINISH_GOAL to JOIN.

Execution: START_GOAL to FINISH_GOAL, START_GOAL to FORK, JOIN to FINISH_GOAL, JOIN to FORK.

The events SUSPEND and RESTART do not appear in the lists above; their treatment will be discussed later, as they are handled in a special way. The JOIN event, and its associated node, only appears in and–parallel executions. In Figures 3 and 4 the execution graphs corresponding to the traces depicted in Figures 1 and 2 are shown.

In the next section we will see how the execution graph can be used to find out the maximum parallelism inherent in an execution.

3 Maximum Parallelism

As we said in Section 2, to calculate maximum parallelism we assume a null scheduling time and an infinite number of processors, so that newly generated parallel tasks can be started without any delay at all. Two interesting results we can obtain from a simulation with these characteristics are the maximum speedup attainable and the minimum number of processors needed to achieve it.

visualization closer to the user's intuition for these predicates needs structures similar to those of Restricted and–parallelism. Furthermore, depiction of dependencies due to side–effects leads to arbitrary graphs. This is also the case for and–parallelism.

It is clear that these figures are theoretical limits, only possible to obtain through simulation, but they can serve as reference to compare alternative parallelizations of a program, without the possible biases and limitations that actual executions can impose. Data about speedup and number of processors can be obtained by building a new graph in which the labels of the edges are modified as follows:

- The time of each FORK–START_GOAL edge is set to zero, to eliminate scheduling times.
- The time of each FINISH_GOAL–JOIN edge (for and–parallelism) is set to zero for the longest task among a set of siblings, and the time for the FINISH–JOIN edge of its siblings is changed so that all of them perform the JOIN at once.

Whit this rewriting, the length of every path from START_EXECUTION to END_EXECUTION will give the shortest execution time. Let us assume that the node corresponding to the START_EXECUTION is x_0, and that the node corresponding to the END_EXECUTION is x_{n-1}. The rewriting process is as follows:

Step 1 $\forall x_i, x_j \in X$ s.t. $t_{i,j} \in T$ and $type\_of(x_i) =$ FORK and $type\_of(x_j) =$ START_GOAL, set $t_{i,j} = 0$.

Step 2 (calculate labels):

Step 2.1 Set $t_0 = 0$.

Step 2.2
$\forall x_i \in X$ s.t. $i > 1$ and $type\_of(x_i) \neq$ JOIN, set $time(x_i) = time(x_j) + t_{j,i}$, where $t_{j,i} \in T$
$\forall x_i \in X$ s.t. $i > 1$ and $type\_of(x_i) =$ JOIN, set $time(x_i) = \max_{x_j \in X, u_{j,i} \in U}(time(x_j))$.

Step 3 $\forall x_i, x_j \in X$ s.t. $u_{i,j} \in U, type\_of(x_i) =$ FINISH_GOAL and $type\_of(x_j) =$ JOIN, set $t_{i,j} = time(x_j) - time(x_i)$.

The minimum time in which the program could be executed is $time(x_{n-1})$. The minimum number of processors needed to achieve this minimum execution time is the maximum number of tasks $N(t)$ simultaneously actives at a given time t, i.e., $N(t)$ is the number of nodes $x_i \in X$ such that $type\_of(x_i) =$ START_GOAL or $type\_of(x_i) =$ JOIN and $\forall x_j \in X$ such that $\exists u_{i,j} \in U$, $time(x_i) \leq t \leq time(x_j)$. The minimum number of processors needed to execute without delays is the maximum of $N(t)$, $\forall t$ such that $time(x_0) \leq t \leq time(x_{n-1})$.

This algorithm is suitable both for or– and and–parallel execution graphs; for or–parallel execution graphs an additional minor step has to be done, to consider the END_EXECUTION event as a global JOIN, where the total time of the execution is stored.

The above tells us how much parallelism there is in a program and how many processors would be necessary to exploit it. High speedups do not mean that the program is necessarily a good candidate for parallel execution: it depends on the number of processors at which the maximum parallelism is achieved. A high number of processors in a small problem usually indicates that the execution consists of a large number of small tasks, thus requiring some sort of granularity control to obtain the best results in real executions.

The SUSPEND and RESTART events can be generated when a sequential task is temporarily suspended and restarted afterwards. This happens, for example, when or–parallel systems wait for a branch to be leftmost in order to execute side–effect predicates, and can also be generated by and–parallel systems if dependencies appear among parallel branches which are being executed in parallel. A complete simulation would take these events into account, but we decided not to do so for a single reason: in practical systems the generation of these events depends completely on the actual execution, and may or may not be present in a given execution, depending on how the scheduling has been performed. Thus, for a complete simulation, all the possible dependencies in any possible correct scheduling would have to be provided. The actual approach is to consider the SUSPEND and RESTART events as non existent, so effectively incorporating the time taken by them into the task execution time. This is also done for the ideal (Section 4) parallelism as well.

4 Ideal Parallelism

When determining ideal parallelism, the possible differences come from the scheduling algorithm utilized, since we only take into account the relationships among the tasks and their length. Scheduling algorithms can be classified depending on whether they are deterministic (used when all data pertaining the execution is available [MC69, LL74, Hu61]) or non deterministic (in which random variables with known characteristic function are used to model non available data [HB88]). Our case is the first one.

From a high level point of view, the ideal parallelism simulation takes:

- an execution graph $G(X, U, T)$,
- a scheduling algorithm P, and
- a number of processors N,

and returns the maximum speedup attainable using that algorithm with the execution description as input data. The execution graph is internally transformed into a job graph, because job graphs are used to formulate most of the existing scheduling algorithms. This is done in order to facilitate the addition of other algorithms to the implementation. In the job graph each node represents a sequential task, and the edges between them represent the dependencies in the execution.

4.1 The Job Graph

A job graph $G(X, U)$ consists of a set of nodes $X = \{x_0, \ldots, x_{n-1}\}$ and a set of edges $U = \{u_{i,j} : 0 \leq i < j < n\}$, where each $u_{i,j}$ represents an edge from node x_i to x_j. The graph contains a node for each task in the execution and an edge for each dependency between tasks. Each node has a unique identifier (an integer from 0 to $n-1$) as well as information related to the task it represents, such as its length. There is a partial ordering $\prec$ among the tasks in X given by the the dependencies present in the execution. We will say that $x_i \prec x_j$ iff $u_{i,j} \in U$. The ideal parallelism problem for a fixed number of processors can be stated as finding the starting time of each task, i.e., a function $\sigma : X \longrightarrow \mathcal{Z}^+$ such that:

(a) No more than m tasks are active at a time:

$$\forall u \geq 0 \ |\{x \in X \text{ s.t. } \sigma(x) \leq u \leq \sigma(x) + \mathit{length}(x)\}| < m$$

(b) No task starts before its predecessors have finished:

$$\forall x_1, x_2 \in X : x_1 \prec x_2 \longrightarrow \sigma(x_1) + \mathit{length}(x_1) \leq \sigma(x_2)$$

(c) σ finds the minimum overall time: let $L = \max_{x \in X}(\sigma'(x) + \mathit{length}(x))$ for a given σ'. Then σ is such that L is the minimum for all possible functions σ' that meet **(a)** and **(b)**.

Such σ gives the starting time for each task. From it, a processor–task mapping is straightforward, since it is required that no more than m processors be active at a time. Each time a processor is freed, the task with the nearest starting time can be assigned to it.[3]

The construction of the job graph is slightly different for and– and or–parallelism, because of the non existence of JOIN events in or–parallelism. Figures 5 and 6 show the two job graphs for the examples we have been using throughout the paper.

4.2 Scheduling Algorithms

It is interesting to find out absolute upper bounds for speedups achievable with a perfect scheduling and a given number of processors. Unfortunately, obtaining an optimal task/processor allocation is, in general, an NP complete problem [GJ79]. Since we want to deal with sizeable, non trivial, programs, this option is too computationally expensive to be used. Instead, we will employ a scheduling algorithm which does

[3] Under the implicit assumption that any processor is able to execute any task.

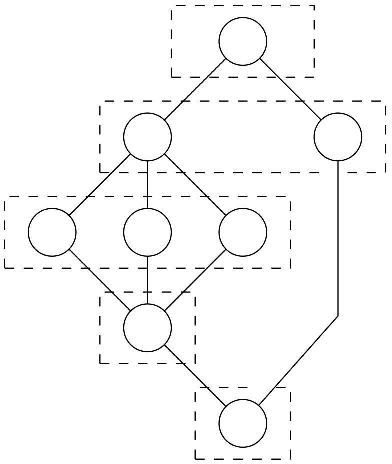

Figure 5: Job graph for and–parallelism

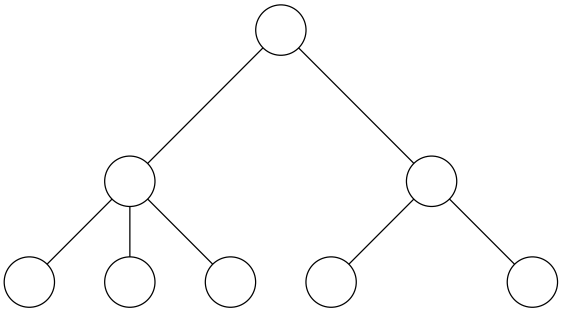

Figure 6: Job graph for or–parallelism

not always find out the best task/processor allocation, but which is much more amenable to run and which gives an *adequate* (able to compute a reasonable answer for a typical input), but not *appropriate* (every processor is attached to a sequential task until this task is finished) scheduling. The algorithm we implemented to find out quasi–optimal schedulings is the so–called *subsets* algorithm. This algorithm in fact gives optimal results under certain conditions (which are, however, not met in our more general case). The reader is referred to [HB88] for more information on this issue.

Although a (quasi–)optimal scheduling gives an estimation of the maximum speedup for a given execution and number of processors, this scheduling is not likely to be found in a real system. That is why we also implemented an approximate version of the scheduling scheme found in the *&–Prolog* system [HG91, Her87]. We expect the comparison of the actual *&–Prolog* system speedups and the results obtained from *IDRA* to serve as an assessment of the accuracy of our technique, whereas the comparison among a (quasi–)optimal scheduling and a real one would serve to estimate the performance of the actual system.

The variation of the inherent parallelism with the size of the problem is also a topic of interest. Frequently one wants more performance not only to solve existing problems faster, but also to be able to tackle larger problems in a reasonable amount of time. In simple problems the number of parallel tasks and the expected attainable speedups can be calculated, but in non–trivial examples it may not be so easy to estimate that. A problem in which the number of parallel tasks generated does not increase accordingly with the size of the problem would not benefit from a larger machine. In Section 6 runs using real traces to find out maximum performances are given.

4.2.1 The Subsets Algorithm

The **subsets** algorithm avoids performing a global scheduling by splitting the nodes in the job graph into disjoint subsets. The nodes in each subset are independent among them, and so they are candidates for parallel execution. The initial subset S_0 is the starting node, and for each S_i, S_{i+1} is the set of nodes which can start once all the nodes in S_i have finished. Once the graph is split, each subset is scheduled separately. In Figure 5 the subsets in an and–parallel job graph are shown inside dashed rectangles.

Once the graph has been partitioned into p subsets $S_0, \ldots, S_{p-1}$, each subset is scheduled almost independently. If the tasks in S_{i+1} started after the last task in S_i finish, the subsets could have been scheduled independently. Since a given task in S_{i+1} may depend only on some of the tasks in S_i, we set the starting time of each task in S_{i+1} to be the time in which all their predecessor tasks in S_i have finished. In each subset $S_i = \{t_1, \ldots, t_k\}$, the scheduling algorithm assigns one t_j to one processor from $P = \{T_0, \ldots, T_{p-1}\}$. Each processor j is modeled as a number T_j which represents the moment from which it is free to execute new work. The first subset is $S_0 = \{x_0\}$, and for each subset $S \neq S_0$, the algorithm performs as follows:

For each task $t_j \in S$ do:

Step 1 Let $Time_j = \max_{x \in X, x \prec t_j}(x)$. This is the earliest time in which t_j can start.

Step 2 If there is any processor $T_p \in P$ such that $T_p \leq Time_j$, assign processor p to task t_j and set $T_p = T_p + length(t_j)$.

Step 3 Otherwise, find $T_q = \min_{T_i \in P}(T_i)$. Assign task t_j to processor q and set $T_q = T_q + length(t_j)$.

Tasks are assigned to free processors. If no free processor exists at a given moment, the first processor to become idle is chosen. The non–determinism in Step 2 is one of the sources of the non optimality of the algorithm, since it is possible that non optimal schedulings will be performed in a subset. In Step 3, T_q is chosen using a heuristic that tries to increase the occupation time of the processors.

4.2.2 The Andp Algorithm

The **andp** scheduling algorithm tries to mimic the behavior of a *&–Prolog* scheduler. For each processor, *&–Prolog* has the notion of *local* and *non local* work: local work is the work generated by a given processor, and it is preferably assigned to it. The dependencies among tasks are used to find out which work is to be considered as local by a processor. To keep track of the local work of each processor, the definition of a processor is augmented to be the 2–tuple $\langle T_p, L_p \rangle$ where T_p is as before, and L_p is the list of tasks generated by processor p. Roughly speaking, the scheduling algorithm tries first to execute tasks locally; if this is not possible, a task is stolen from another processor's stack.

The andp scheduling algorithm can be split into two different parts: the first one takes care of obtaining work available in the system, and the second one generates new work and stores it in the processor's local stack. The part of the scheduling algorithm that is in charge of getting work is as follows:

Step 1 Set $L_0 = \{t_0\}$ and assign x_0 to processor 0.

Step 2 If $\forall \langle T_i, L_i \rangle \in P,\ L_i = \emptyset$, finish. Otherwise select the processor p such that $T_p = \min_{\langle T_i, L_i \rangle \in P}(T_i)$

Step 3 If $L_p \neq \emptyset$ assign the first task $x \in L_p$ to processor p and go to Step 2.

Step 4 If $L_p = \emptyset$, let $N = \{\langle T_i, L_i \rangle \text{ s.t. } \langle T_i, L_i \rangle \in P,\ L_i \neq \emptyset\}$ and find the processor q such that $T_q = \min_{\langle T_i, L_i \rangle \in N}(T_i)$. Assign the first task $x \in L_q$ to processor p and go to Step 2.

The generation of new work, after task x_i from the list of tasks L_q is assigned to processor p, is the following:

Step 1 Set $L_q = L_q - \{x_i\}$.

Step 2 Set $T_p = T_p + length(x_i)$.

Step 3 Set $L_p = L_p \cup \{x_j \in X \text{ s.t. } x_i \prec x_j\}$.

5 Overview of the Tool

A tool, named *IDRA* (IDeal Resource Allocation) has been implemented using the ideas and algorithms shown before. The traces used by *IDRA* are the same as those used by *VisAndOr* [CGH93], a tool to visualize parallel execution of logic programs, and thus it can be used to calculate ideal and maximum speedups for the systems *VisAndOr* can visualize (namely, the independent and–parallel system *&–Prolog* and the or–parallel systems Muse and Aurora; the deterministic dependent and–parallel system Andorra–I is not supported yet).

The tool itself has been completely implemented in Prolog. In addition to the generation of maximum and ideal speedups, *IDRA* can generate a new trace file for ideal parallelism, which can be visualized using *VisAndOr* and compared with the original one. *IDRA* can also be instructed to generate automatically speedup data for a range of processors. This data is dumped in a format suitable for a tool like xgraph to read.

The traces used with *IDRA*, as those used with *VisAndOr*, need not be generated by a real parallel system. Instead, it is possible to generate them with a sequential system augmented to dump information about parallelism. The only requirement is that the dependencies among tasks be properly reflected, and that the timings be accurate.

Accuracy in the timings has not been straightforward to obtain. Usual UNIX environments have a vague notion of what an accurate timing is. We found that calls to standard OS routines to find out the current time either were not accurate enough for our purposes, or the time employed in such calls largely exceeded the total execution time of the benchmark, thus leading to incorrect results (sequential tasks being traced were much longer than without tracing). To obtain accurate timings we used the microsecond resolution clock available in some Sequent multiprocessors [Seq87]. This clock is not only very precise, but also memory mapped and can thus be accessed in the time corresponding to one memory access, with negligible effect on performance. We have also developed a technique for dealing with clocks with high but predictable access times, by subtracting the accumulated clock access time from the timings.

6 Using *IDRA*

In this section we will show examples of the use of *IDRA* on real execution traces. The traces we will use have been generated by the *&–Prolog* system, both for or– and and–parallelism. The ones corresponding to and–parallelism were generated by *&–Prolog* running programs parallelized for independent and–parallel execution. The generation of the traces corresponding to or–parallelism needed of a slight modification of *&–Prolog* to make it issue an event each time a choice-point is created. The reason to generate or–parallel traces using *&–Prolog* was that or–parallel schedulers usually make work available to parallel execution when they find it worth, and not in every choice-point. This, in our approach, would not allow us to find out the maximum or ideal parallelism, since opportunities for performing work in parallel would be lost.

The results of the simulations have been compared with actual executions in *&–Prolog* and MUSE, to assess the accuracy and stability of our simulation.

6.1 Description of the Programs

In this section we briefly describe the programs used to test the tool. This is included to help in understanding their behavior both in simulation and in execution.

- Programs with and–parallelism

  pderiv performs symbolic derivation.

  occur counts occurrences in lists.

  tak computes the Takeuchi function.

  boyer is an adaptation of the Boyer–Moore theorem prover.

  matrix performs square matrix multiplications.

  quicksort is the standard quicksort program, here using *append/3* instead of difference lists.

  bpebpf calculates the number e, using the series $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots$ A divide–and–conquer scheme is used both for the series and for each of the factorial calculations. This causes the generation of a very large number of tasks.

  bpesf is similar to above, but each factorial is computed sequentially. The number of tasks is much smaller than above.

  pesf also calculates e using the same series, but here each factor is computed in parallel with the rest of the series, from left to right.

- Programs with or–parallelism:

  domino calculates all the legal sequences of 7 dominoes.

  jugs calculates all the solutions of 8 movements for the water jugs problem.

queens computes all the solutions to the 5 queens problem.

witt is a conceptual clustering program.

lanford1 this program finds out some elements needed to complete a Lanford sequence.

lanford2 this program is similar to **lanford1**, but the data structures are completely different.

Program	Speedup	Processors	Performance
deriv	100.97	378	0.26
occur	31.65	49	0.64
tak	44.16	315	0.14
boyer	3.49	11	0.31
matrix (10)	26.86	80	0.33
matrix (15)	58.70	170	0.34
matrix (20)	101.91	286	0.35
matrix (25)	161.68	462	0.34
quicksort (400)	3.93	15	0.26
quicksort (600)	4.07	17	0.23
quicksort (750)	4.28	19	0.22
bpebpf (30)	23.21	260	0.08
bpesf (30)	10.11	31	0.32
pesf (30)	2.59	25	0.10

Table 2: Maximum and–parallelism

Program	Speedup	Processors	Performance
domino	32.01	59	0.54
jugs	1.95	8	0.24
queens	18.14	40	0.45
witt	1.12	25	0.04
lanford1	19.72	44	0.44
lanford2	114.87	475	0.24

Table 3: Maximum or–parallelism

6.2 Maximum Parallelism Performance

The maximum parallelism performance for the programs above mentioned appears in Tables 2 and 3. They show, for each of the benchmarks already referred to, the maximum speedup attainable according to the simulation, the number of processors at which this speedup is achieved, and the relative performance with respect to a linear speedup , i.e., performance = $\frac{\text{speedup}}{\text{processors}}$.

The numbers that appear next to some of the benchmark names correspond to the size of the input data: for **matrix**, the number of rows and columns of the matrix to be multiplied; for **quicksort**, the length of the list to be sorted, and for **bpebpf**, **bpesf** and **pesf**, the number of factors in the series.

Programs which require a large number of processors despite the problem to be solved not being very big are those where tasks are small. This would suggest that a parallel system would need of some sort of granularity control to execute them efficiently. This turns out not to be always the case for real executions on shared memory multiprocessors with a small number of processors, as we will see in Section 6.3 and Table 4, but will certainly be an issue in larger or distributed memory machines.

In programs with a regular structure, such as **matrix**, potential speedups grow accordingly with the size of the problem, which in turn determines the number of tasks available. However, in programs where

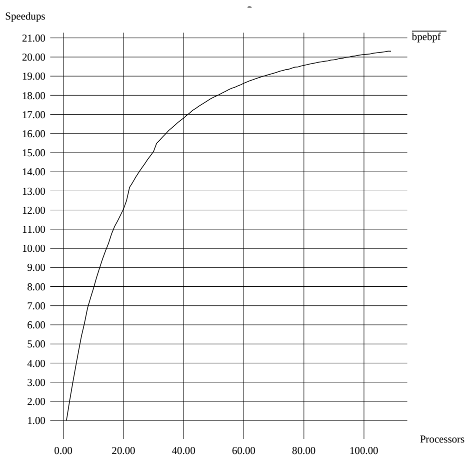

Figure 7: Computation of e

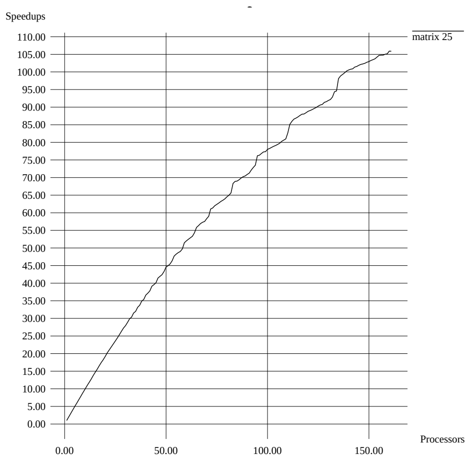

Figure 8: 25×25 matrix multiplication

the length of the tasks is variable and the structure of the execution is not homogeneous, the maximum speedup achievable grows very slowly with the size of the problem. This is the case of, for example, **quicksort**, in which the sequential parts caused by the partitioning and the appending of the list to be sorted finally dominate the whole execution, preventing further speedups and confirming once again Amhdal's law.

6.3 Ideal Parallelism Performance

For each and–parallel and or–parallel benchmark we have found the ideal parallelism and the actual speedups on one to nine processors. The results are shown in Tables 4 and 5. In each of those tables there are three rows for each benchmark: the uppermost one is the predicted speedup obtained using the *subsets* algorithm, the middle one is the speedup obtained using the *andp* algorithm, and the lower-most one has been obtained using *&–Prolog* and Muse, i.e., the speedups there are actual ones.

The data obtained with *&–Prolog* was gathered using a version of the scheduler with reduced capabilities (for example, no parallel backtracking was supported) and a very low overhead, so that the andp simulation and the actual execution be as close as possible. In general the results from the simulation are very close to those obtained from the actual execution, which seems to imply that the simulations results are quite accurate and useful. Usually, the results with the subsets scheduling algorithm are slightly better, but due to its non optimality, it is surpassed sometimes by the andp algorithm and by *&–Prolog* itself (see, for example, the row corresponding to the quicksort benchmark). With respect to the relationship between the speedups obtained by the andp algorithm and the actual *&–Prolog* speedups, sometimes the actual speedups are slightly better than the simulation and sometimes they are not, but in general they are quite close. This is understandable, given the heuristic nature of these algorithms.

Benchmarks that show good performance in Table 3 have good speedups here also. But the inverse is not true: benchmarks with low performance in maximum parallelism can perform very well in actual executions (see, for example, the data for **bpebpf**). Figure 7 shows the simulated speedups for the benchmark **bpebpf**; Figure 8 shows a similar figure for **matrix** multiplication. The speedup in the first one, although showing a logarithmic behavior, is quite good for a reduced number of processors. The second one has a larger granularity and shows almost linear speedups with respect to the number of processors. When the number of processors increases beyond a limit, the expected sawtooth effect appears due to the regularity of the tasks and their more or less homogeneous distribution among the available processors.

Concerning the data for or–parallelism, Muse performs slightly worse than the prediction given by the simulation. This is not surprising, since Muse has an overhead associated with task switching (due to copying) that *&–Prolog* does not have. However, there is one case where Muse performs better than *IDRA* prediction: the water jugs benchmark. In fact, in this case Muse beats even the maximum parallelism

Program	**Time (ms)**	**Scheduling**	**Processors**								
		Algorithm	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
deriv	240	subsets	1.00	1.99	2.99	3.97	4.95	5.93	6.90	7.86	8.82
		andp	1.00	1.99	2.97	3.94	4.86	5.77	6.79	7.56	8.40
		real	1.00	2.00	3.00	4.00	4.80	4.80	6.00	8.00	8.00
occur	1750	subsets	1.00	1.99	2.97	3.97	4.49	5.14	5.96	7.10	8.73
		andp	1.00	1.99	2.55	3.28	3.97	4.45	5.12	5.92	7.08
		real	1.00	1.96	2.96	3.97	4.48	5.83	5.83	7.00	8.75
tak	610	subsets	1.00	1.99	2.97	3.93	4.86	5.77	6.65	7.51	8.33
		andp	1.00	1.97	2.95	3.91	5.48	5.76	6.57	7.54	8.30
		real	1.00	1.90	2.65	3.58	4.35	5.08	5.54	6.09	6.77
boyer	110	subsets	1.00	1.78	2.34	2.65	2.84	2.94	3.05	3.09	3.13
		andp	1.00	1.79	2.37	2.76	3.02	3.15	3.25	3.30	3.31
		real	1.00	1.57	1.83	2.20	2.20	2.20	2.20	2.20	2.20
matrix (10)	170	subsets	1.00	1.98	2.91	3.86	4.74	5.57	6.41	7.26	8.02
		andp	1.00	1.97	2.70	3.59	4.59	5.21	6.09	6.86	7.54
		real	1.00	1.88	2.83	3.39	4.25	5.66	5.66	8.50	8.50
matrix (15)	550	subsets	1.00	1.99	2.96	3.94	4.91	5.84	6.76	7.71	8.62
		andp	1.00	1.97	2.85	3.51	4.40	5.36	6.37	7.15	7.84
		real	1.00	1.96	2.89	3.92	4.58	5.50	6.87	7.85	7.85
matrix (20)	1270	subsets	1.00	1.99	2.98	3.97	4.94	5.92	6.88	7.85	8.80
		andp	1.00	1.99	2.78	3.56	4.36	5.23	6.07	6.95	8.01
		real	1.00	1.95	2.95	3.84	4.88	5.77	6.68	7.47	8.46
matrix (25)	2460	subsets	1.00	1.99	2.98	3.98	4.97	5.94	6.92	7.91	8.88
		andp	1.00	1.97	2.73	3.51	4.44	5.54	6.41	7.34	7.98
		real	1.00	1.98	2.96	3.96	4.91	5.85	6.83	7.93	8.78
quicksort (400)	590	subsets	1.00	1.76	2.32	2.69	2.95	3.15	3.28	3.35	3.40
		andp	1.00	1.76	2.26	2.66	3.00	3.23	3.68	3.60	3.60
		real	1.00	1.73	2.26	2.68	3.10	3.27	3.47	3.47	3.47
quicksort (600)	1070	subsets	1.00	1.80	2.41	2.84	3.15	3.38	3.53	3.64	3.71
		andp	1.00	1.75	2.25	2.75	3.20	3.34	3.79	3.97	4.00
		real	1.00	1.72	2.37	2.74	3.14	3.45	3.68	3.82	3.96
quicksort (750)	1500	subsets	1.00	1.78	2.36	2.75	3.04	3.25	3.38	3.47	3.53
		andp	1.00	1.71	2.42	2.60	3.13	3.55	3.66	3.75	3.67
		real	1.00	1.82	2.41	2.88	3.40	3.65	3.94	4.05	4.16
bpebpf (30)	220	subsets	1.00	1.96	2.88	3.74	4.60	5.41	5.41	5.41	5.41
		andp	1.00	1.93	2.81	3.69	4.30	5.16	5.60	6.32	6.98
		real	1.00	1.83	2.44	3.66	4.40	4.40	5.50	5.50	7.33
bpesf (30)	180	subsets	1.00	1.96	2.88	3.75	4.53	5.18	5.99	6.33	6.75
		andp	1.00	1.88	2.59	3.27	3.67	4.23	4.56	5.08	5.12
		real	1.00	1.80	2.57	3.60	4.50	4.50	4.50	6.00	6.00
pesf (30)	200	subsets	1.00	1.47	1.74	1.92	2.05	2.14	2.20	2.26	2.31
		andp	1.00	1.41	1.65	1.83	1.95	2.02	2.10	2.18	2.26
		real	1.00	1.33	1.66	1.81	1.81	1.81	2.00	2.00	2.22

Table 4: Ideal and–parallelism

prediction in Table 3, which would imply that something is wrong. We have traced the reason for that behavior to an erroneous trace. This is because of the way in which *&–Prolog* dumps traces for or–parallelism imposes a small overhead for each possible branch in the execution. This overhead is small and thus it was not felt that it should be compensated for. However, the water jugs benchmark has a large amount of very small tasks, so that the overhead associated with the creation of an event is a sizeable part of the time reported for a task. This results, from the point of view of the simulation, in larger tasks than in the actual implementation, and this produces misleading results. The timings could have been adjusted easily (by subtracting the time taken in creating the event as mentioned previously) and a correct trace generated. However, we felt that it was interesting to see how a small error in computing

Program	Time (ms)	Scheduling Algorithm	Processors								
			1	2	3	4	5	6	7	8	9
domino	130	subsets	1.00	1.98	2.94	3.86	4.75	5.61	6.42	7.20	7.97
		andp	1.00	1.98	2.92	3.86	4.78	5.61	6.54	7.32	8.26
		real	1.00	1.62	2.16	2.60	3.25	3.25	3.25	3.25	4.33
jugs	220	subsets	1.00	1.47	1.60	1.67	1.72	1.74	1.76	1.78	1.78
		andp	1.00	1.43	1.61	1.71	1.70	1.70	1.70	1.70	1.70
		real	1.00	2.00	2.75	3.66	3.66	4.40	5.50	5.50	5.50
queens	70	subsets	1.00	1.97	2.92	3.82	4.70	5.48	6.22	6.93	7.55
		andp	1.00	1.95	2.77	3.77	4.72	5.33	5.89	6.30	6.48
		real	1.00	1.75	2.33	2.33	3.50	3.50	3.50	3.50	3.50
witt	5090	subsets	1.00	1.05	1.07	1.08	1.09	1.09	1.09	1.09	1.09
		andp	1.00	1.05	1.07	1.08	1.09	1.09	1.09	1.09	1.09
		real	1.00	1.05	1.07	1.09	1.10	1.10	1.10	1.11	1.11
lanford1	160	subsets	1.00	1.98	2.91	3.79	4.59	5.34	6.04	6.67	7.45
		andp	1.00	1.97	2.92	3.82	4.73	5.53	6.27	7.29	8.09
		real	1.00	1.77	2.28	3.20	4.00	4.00	4.00	4.00	5.33
lanford2	2090	subsets	1.00	1.99	2.99	3.98	4.97	5.95	6.92	7.88	8.85
		andp	1.00	1.99	2.98	3.97	4.96	5.91	6.88	7.87	8.85
		real	1.00	1.97	2.86	3.66	4.54	5.35	6.33	6.96	7.74

Table 5: Ideal or–parallelism

the size of tasks can result in a large error in the computed speedups. We believe this further supports one of the assumptions in our approach, that of performing simulations but based on *accurate* estimates of task execution times.

7 Conclusions and Future Work

We have reported on a technique and a tool to compute ideal speedups using simulations which have as input data information about executions gathered using real systems. We have applied it to or– and independent and–parallel benchmarks, and compared the results with those from actual executions. The results show that the simulation is quite reliable and corresponds well with the results obtained from actual systems. In particular, results are very close to those obtained from a the *&–Prolog* system. This corresponds with expectations, since the particular version of the *&–Prolog* systems used has very little overhead associated with parallel execution. The results for or–parallelism and Muse also offer a strong correspondence between simulation and actual system, being somewhat better in the simulation, which is understandable when considering the slight task creation overhead incurred due to copying. The technique can be extended for other classes of systems and execution models, provided that the data which models the executions can be gathered with enough accuracy.

As far as the sizes of the executions that can be simulated, this is not limited by the trace generation phase, which uses resources comparable to those of the actual execution, but rather by the simulation phase. Our Prolog implementation of this phase is rather naive. In the future, optimizations would be necessary in order to allow larger traces to be processed in a reasonable amount of time. We also plan to modify the simulator in order to support other execution paradigms, such as Andorra–I [SCWY90], ACE [GHPC94], AKL [JH91], IDIOM [GSCYH91] etc. and study other scheduling algorithms. Finally, we believe the same approach can be used to study issues other than ideal speedup, such as memory consumption, copying overhead, etc.

8 Acknowledgments

We would like to thank S. Debray and the members of the CLIP group at the CS department of the TU Madrid for trusting *IDRA* and providing very valuable feedback on its use.

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.

[CGH93] M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.

[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.

[GHPC94] G. Gupta, M. Hermenegildo, Enrico Pontelli, and Vítor Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*. MIT Press, June 1994. to appear.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, 1979.

[GSCYH91] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: Integrating Dependent and-, Independent and-, and Or-parallelism. In *1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

[HB88] Kai Hwang and Fayé Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1988.

[Her87] M. V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*, pages 556–575. University of Melbourne, MIT Press, May 1987.

[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.

[Hu61] T.C. Hu. Parallel sequencing and assembly line problems. *Operating Research*, 9(6):841–848, November 1961.

[JH91] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[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.

[LL74] J.W. Liu and C. L. Liu. Bounds on scheduling algorithms for the heterogeneous computing systems. In *1974 Proceedings IFIP Congress*, pages 349–353, 1974.

[Lus90] E. Lusk et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

[MC69] R.R. Muntz and E.G. Coffman. Optimal preemptive scheduling on two processor systems. *IEEE Transactions on Computers*, pages 1014–1020, November 1969.

[MH90] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[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.

[Seq87] Sequent Computer Systems, Inc. *Sequent Guide to Parallel Programming*, 1987.

[SH91] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *1991 International Logic Programming Symposium*. MIT Press, October 1991.

[She92] K. Shen. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In *Proc. Joint Int'l. Conf. and Symp. on Logic Prog.* MIT Press, 1992.

[SK92] D.C. Sehr and L.V. Kalé. Estimating the Inherent Parallelism in Logic Programs. In *Proceedings of the Fifth Generation Computer Systems*, pages 783–790. Tokio, ICOT, June 1992.