IDRA (IDeal Resource Allocation): Computing Ideal Speedups in Parallel Logic Programming

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

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

Departamento de Inteligencia Artificial
Facultad de Informática
Universidad Politécnica de Madrid
Boadilla del Monte – 28660 Madrid – Spain
Fax number: +34–1–352–4819
Contact author: M. Carro

Abstract

We present a technique to estimate accurate speedups for parallel logic programs with relative independence from characteristics of a given implementation or underlying parallel hardware. The proposed technique is based on gathering accurate data describing one execution at run–time, which is fed to a simulator. Alternative schedulings are then simulated and estimates computed for the the corresponding speedups. Such speedups can be used to compare different parallelizations of a program or to evaluate the performance of parallel systems. A tool implementing the aforementioned techniques is presented, and its predictions are compared to the performance of real systems, showing good correlation.

Keywords: Parallel Logic Programming; Simulation; Parallelism and Concurrency; Performance Evaluation.

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 [HG91, 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 try to find techniques to answer a different question: given a (parallel) execution paradigm, what is the maximum benefit that can be obtained from executing a program in parallel in a system designed according to that paradigm? (we will refer to this as "maximum parallelism"). What are the resources (for example, processors) needed to exploit all parallelism available in a program? How much parallelism can be ideally exploited for a given set of resources (e.g. a fixed 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, such answers cannot be obtained from an actual implementation, either because of limitations of the implementation itself or because of limitations of the underlying machinery, such as the number of processors or the available memory. It appears that any approach for obtaining such answers 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 logic programs, 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 executed by a high–level meta-interpreter/simulator which computes ideal speedups for independent and–parallelism, or–parallelism, and combinations thereof (see [Con83] and Section 3 for a description of different types of parallelism in logic programs). 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 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 [War83, AK91] 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. Another is that the type of instrumentation performed on the 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 precission and execution size limitations of previous approaches by using precise timing information. Also, it allows 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 executed directly in real systems.

Although the techniques we present have been designed within the area of parallel logic programming, we believe that the core idea can be applied to any execution paradigm, and that the techniques (and tools) developed can be applied directly to those paradigms conforming to the initial assumptions.

The paper is structured as follows: Section 2 sketches our objectives. Section 3 describes more in depth our approach and the techniques used in its implementation. Section 4 relates the traces obtained at run–time with the graphs used to simulate alternative schedulings. Sections 5 and 6, respectively, show how the maximum and ideal parallelism are calculated. In Section 7 an overview of *IDRA*, the actual tool, is given. Section 8 contains examples of simulations made using *IDRA* and comparisons of actual implementations with the results of the simulation.

2 Objectives

Our objective is to perform speedup analysis of executions of parallel logic programs, in a relatively independent way from the characteristics (such as number of processors, absolute speed, etc.) of the platform in which they have been executed. Given a (parallel) program and a number (which may be unbound) of processors, different schedulings can (and do) greatly affect the total execution time.[1]

Among the information we can extract from alternative schedulings, the following may be of interest:

[1] Of course, faster processors will affect the absolute execution time as well, but since ideally this speed scales to the whole execution, the speedup obtained with respect to sequential executions should not change. This, in practice, is not true, since, for example, bus bandwidth limits the attainable speedup in memory–intensive applications. This is, precisely, one example of the limitations we want to overcome.

- 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 determine the absolute maximum performance of a program, i.e., the minimum time in which it could have been executed while respecting the dependencies among tasks. This is used, for example, for comparing different parallelizations/sequentializations of a given program (e.g., if different domains or annotators for parallelism are being evaluated, see for example [BGH94a, BGH94b]) or different parallel algorithms proposed for a given problem (e.g. [DJ94]). In the simulation we know that the speedup obtained has not been limited by the machine itself (e.g., number of processors, bus contention, etc.)

Ideal parallelism can be used to test the absolute performance of a given scheduling algorithm in a fixed number of processors, by comparing the speedup obtained in the machine with the maximum speedup attainable using that number of processors. The efficiency of an implementation can also be studied by testing the actual speedups against those predicted by the simulator using the same scheduling algorithm as the implementation. Also, how the performance of a program evolves for a number of processors as large as desired can be studied; this gives interesting information about the potential parallelism in a program.

We want our simulation to be useful for medium size applications, and the results to be as accurate as possible. That is why the simulation takes place at the scheduling level, the sequential task timing being (preferably) obtained using real executions.

3 Parallelism and Trace Files

To simulate an alternative scheduling of a parallel execution we need a certain description of that execution. This description must contain, at least, 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. Such a description can be produced by executing programs 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 the simulation) augmented to generate execution logs, 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. This considerably widens the applicability of the developed tool because it allows studying the (expected) performance of parallel programs and scheduling algorithms without the need of an actual parallel machine or in non–realistic conditions (for example, unbound number of processors).

Node	**Comment**
START_EXECUTION	Start of the whole execution
END_EXECUTION	End of the whole execution
START_GOAL	A sequential task starts
FINISH_GOAL	A sequential task ends
FORK	Execution splits
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 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. Some types of events that we have found useful, 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 the relevant details of the sequential tasks in the computation. Figures 1 and 2 represent two parallel executions, in which some events have been marked at the point where they occur for the types of parallelism we will discuss in the following sections. The length of the vertical segments is intended to reflect the actual time taken by the sequential tasks and the scheduling delays.

In our case timing data is gathered by a modified Prolog implementation (but it can be also generated by other means), which ensures that the timing information is realistic. The part that is simulated regards the possible (alternative) schedulings of those sequential tasks, while respecting the precedences among the tasks.

It should be noted that, in parallel dialects of Prolog, collecting traces is easy from the user point of view. The structure of the Prolog language and its

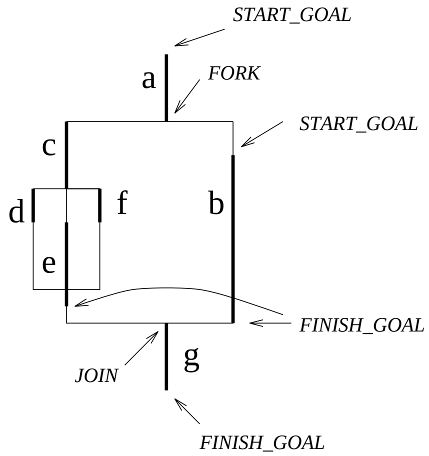

Figure 1: And–parallel execution

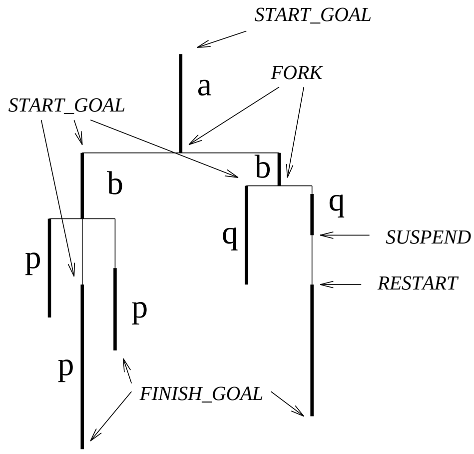

Figure 2: Or–parallel execution

implicit control helps to automatically identify the "interesting places" (for example, where a sequential task starts or finishes) in the execution.

The parallel execution models which we will deal with in this paper stem naturally from the view of logic programming as a process–oriented computation. The two main types of parallelism available in a logic program are **and–parallelism** and **or–parallelism**. We will briefly review some related concepts in the following sections.

3.1 Restricted And–parallelism

Restricted and–parallelism [DeG84, Her86] refers to the execution of independent goals in the body of a clause using a fork and join paradigm. Independent goals are those that meet some "independence conditions" (for example, they do not share variables at run time, thus avoiding all possible Read–Write and Write–Write conflict). Run–time tests may be placed in the program source, if a static analysis (either automatic or by hand) could not determine if independence conditions always hold or not.[2] The only dependencies existing in RAP appear among the goals before and after the parallel execution and the goals executed in parallel. Consider the *&–Prolog* [HG91] program below, where the "&" operator, in place of the comma operator, stands for and–parallel execution (the predicates not defined here 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 closing 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.

3.2 Or–parallelism

Or–parallelism corresponds to the parallel execution of different alternatives of a given predicate. Since each alternative belongs conceptually to a different "universe" there are (in principle) no dependencies among alternatives. However, 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 and extra–logical constructs that can affect the execution of other branches — from a dependency graph point of view, much as `wait` introduces dependencies among and–parallel branches. For the sake of simplicity we will not address those cases in depth here.

As an example, consider the following program, which has alternatives for predicates `b`, `p` and `q`:

[2] 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 model [Con83] and by *&–Prolog* [HG91] when `wait`, which can suspend a task until a certain condition is met, is used.

```
main:- a, b.
b:- p.                 p:- ...
b:- q.                 p:- ...
q:- ...                p:- ...
q:- ...
```

Assuming that p and q have no or–parallelism inside, a possible graph depicting an execution of this predicate is shown in Figure 2. Note that the rightmost branch in the execution is suspended at some point and then restarted. This suspension is probably caused by its sibling, because a side–effect predicate or a cut would impose a serialization of the execution. In terms of dependencies among events, FORKs are not balanced by JOINs. The resulting graph is thus a tree. If p or q had parallelism, inside a similar representation would be recursively applied.

The END_EXECUTION event, which does not appear in Figure 2, can be supposed to be issued immediately after the last FINISH_GOAL by the system executing the program, or added afterwards by some tool.

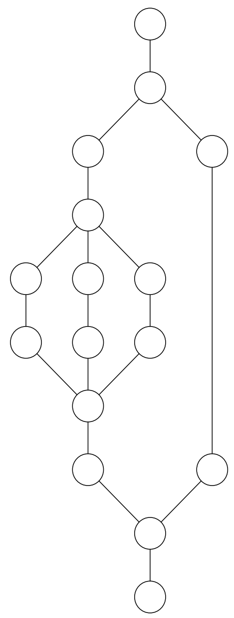

Figure 3: Execution graph, and–parallelism

4 From Traces to Graphs

From a practical point of view, the format of the traces may depend on the system that created them: traces may have information that is not necessary, or be structured in an undesirable way, perhaps because they may serve to other purposes as well.[3] On the other hand, scheduling algorithms are usually formulated in terms of the well–known *job graphs* (see, e.g., [MC69, LL74, Hu61, HB88]). However, in job graphs only tasks and relationships are reflected: scheduling delays do not appear—or are assumed to be a part of the tasks themselves. To be able to change the delays introduced by the scheduling algorithms, and to somewhat separate the traces from the internal structures, we will use *execution graphs* as an intermediate object that abstracts the trace containing only the information needed to simulate new schedulings.

4.1 The Execution Graph

An execution graph translates the idea of events and their dependencies into a mathematical object. An execution graph is a directed acyclic 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, and

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

In the execution graph each node corresponds to an event, and has associated a type (the same of the event—see Table 1) and the point in time in which the corresponding event has occured. Each edge reflects a dependency between events, and its associated weight represents the time elapsed between them. We distinguish two types of edges: those which represent the sequential execution of a task and those which represent delays introduced by scheduling. The edges fall, thus, in one of the following two categories:

Scheduling Edges: FORK to START_GOAL, FINISH_GOAL to JOIN.

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

Figures 3 and 4 show, respectively, the structure of the execution graphs corresponding to the traces depicted in Figures 1 and 2 (the weights have been omitted for simplicity). The execution graphs is a formal, intermediate representation for event traces. This representation is transformed into a job graph, wich is in turn used to simulate the schedulings.

4.2 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 $U = \{u_{i,j}, 0 \leq i <$

[3]This is the case for the actual parallel systems that we study—see Section 7—where traces originally designed for visualization are used by our simulation.

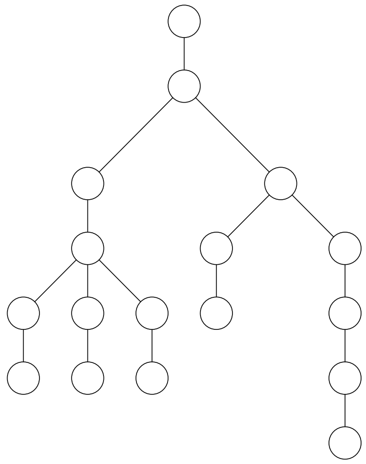

Figure 4: Execution graph, or–parallelism

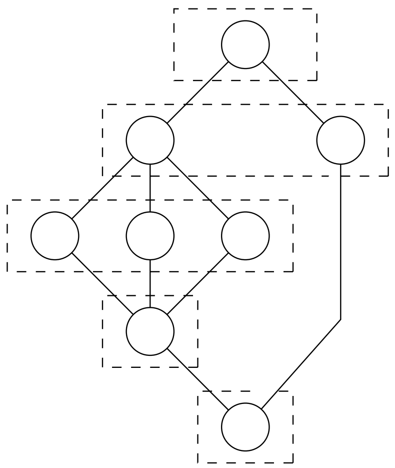

Figure 5: Job graph for and–parallelism

$j < n\}$, where each $u_{i,j}$ represents an edge from node x_i to x_j. The graph contains a node for each sequential task in the execution and an edge for each dependency between tasks. Each node x_i has information related to the task it represents, such as its length $l(x_i)$ and its starting time $t(x_i)$. There is a partial ordering $\prec$ among the tasks in X given by the dependencies present in the execution. We will say that $x_i \prec x_j$ iff $u_{i,j} \in U$. Figures 5 and 6 show job graphs for the and– and or–parallel examples we have been using throughout the paper.

Job graphs are obtained from execution graphs by eliminating the scheduling times (represented by scheduling edges) and transforming the execution edges (which represent actual sequential tasks) into nodes. The dependencies in the job graph and the length of each task are inherited from the execution graph. This transformation can, of course, be parameterized to take into account actual or minimal

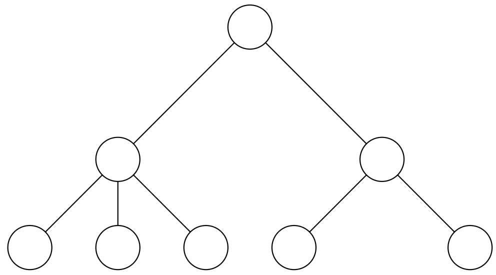

Figure 6: Job graph for or–parallelism

scheduling delays, incrementing the usefulness of the tool.

4.3 Scheduling in Job Graphs

A scheduling for a given execution $G(X, U)$ can be formally viewed as a function $\sigma : X \rightarrow \mathcal{Z}^+$ that assigns a starting time to each task, the task's length remaining unchanged. In order for σ to represent a correct scheduling, no task can start before all its predecessors have finished:

$$\forall x_i, x_j \in X : x_i \prec x_j \rightarrow \sigma(x_i) + l(x_i) \leq \sigma(x_j) \quad (1)$$

A scheduling σ that minimizes the time spent in the execution has to meet the following condition:

$$\text{Let } L_{\sigma'} = \max_{x \in X}(\sigma'(x) + l(x)) \text{ for a } \sigma' \in \{X \rightarrow \mathcal{Z}^+\}.$$

$$\text{Then } \sigma \text{ is such that } L_\sigma = \min_{\sigma' \in \{X \rightarrow \mathcal{Z}^+\}} (L_{\sigma'}) \quad (2)$$

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 functions are used to model non available data [HB88]). Since we are doing "post–mortem" scheduling simulations, our case is the former.

5 Maximum Parallelism

As mentioned in Section 2, maximum parallelism assumes a null scheduling time and an infinite number of processors, so that newly generated tasks can be started without any delay at all. A scheduling with these conditions can be modeled as a function σ, as described in Section 4.3 and which meets conditions 1 and 2.

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. Obtaining both these numbers is an NP–complete problem [GJ79]; however, the exact maximum speedup and an upper bound on the number of processors is easy to obtain. This is still useful, because it gives an estimation of the best performance that can be expected from the program(s) under study. It can serve to compare alternative parallelizations of a program, without the possible biases and limitations that actual executions impose, but still retaining the accuracy in the timing of the tasks.

We can recalculate the starting time $t(x)$ assigned to each node $x \in X$, starting at 0 for the first node in the execution, so that the starting time in each task corresponds to the maximum of the ending time of its predecessors. Then, assuming that x_{n-1} is the node corresponding to the last task in the execution, the minimum time that the execution can take is $t(x_{n-1}) + l(x_{n-1})$. From this, speedups with respect to sequential executions are straightforward to obtain, the sequential execution time being the sum of the lengths of all the tasks.

The maximum number of tasks simultaneously active is an upper bound on the minimum number of processors needed to achieve this execution time. Let $N(t)$ be defined as

$$N(t) = |\{x \in X | t(x) \leq t \leq t(x) + l(x)\}| \qquad (3)$$

i.e., $N(T)$ is the number of tasks active at time t. The minimum number p of processors needed to execute without delays is

$$p = \max_{0 \leq t < t(x_{n-1}) + l(x_{n-1})} N(t)$$

assuming again that the program execution starts at time 0.

Note that high speedups do not necessarily mean that the program is a good candidate for parallel execution: this depends, of course, on the number of processors at which this maximum parallelism is achieved. We will see examples illustrating this in Section 8.2.

6 Ideal Parallelism

By ideal parallelism we refer to the situation in which, for a given number m of processors, a perfect scheduling has been performed, in the sense that the minimum execution time possible (with that number of processors) was achieved. A scheduling algorithm that performs ideal parallelism can be modeled by a mapping σ as defined in 4.3, to which the following restriction has been added:

$$\forall t, 0 \leq t < t(x_{n-1}) + l(x_{n-1}), N(t) \leq m \qquad (4)$$

where $N(t)$ is as defined in equation 3, i.e., the number of tasks simultaneously active is less than or equal to m.

Such σ gives the optimum starting time for each task. From it, a processor–task mapping is straightforward, since it is required that no more than m tasks be active at a time. When a task is finished, the processor that executed it can be assigned to the task with the nearest starting time.[4]

It would be interesting to find out the speedups achievable using a perfect scheduling. 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 non optimal scheduling algorithms which give 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.

From a high level point of view, the ideal parallelism simulation takes a description of the execution, a scheduling algorithm A, and a number of processors N, and returns the maximum speedup attainable in the form of a function $t : X \to \mathcal{Z}^+$ that reflects the calculated starting time for each task.

The algorithm we implemented to find out quasi–optimal schedulings is the so–called *subsets* [HB88] algorithm, which in fact gives optimal results under certain conditions (that are however not always met in our more general case).

Testing the quality of an existing scheduler against an idealized one is also interesting, because that comparison would give an idea of how good is the implementation of the scheduling algorithm. Following that idea, we also implemented a 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 problem size 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 es-

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

timate that. Problems in which available parallelism does not increase with the size of the problem would not benefit from a larger machine. In Section 8 examples illustrating this are given.

In the next two sections we will describe the two scheduling algorithms currently implemented in the simulation tool.

6.1 The *Subsets* Scheduling Algorithm

The **subsets** [HB88] algorithm avoids performing a global scheduling by splitting the nodes (tasks) in the job graph into disjoint subsets (those inside dashed rectangles in Figure 5). The nodes in each subset represent tasks that are independent among them, and so they are candidates for parallel execution.

Each processor $j, 0 \leq j < p$, is modeled as a number T_j which represents the moment from which it is free to execute new work. The set $P = \{T_0, \ldots, T_{p-1}\}$ contains the availability times of the processors in the system. At any given time, no task can be scheduled before $\min_{T \in P}(T)$.

The initial subset is a singleton containing only the first task: $S_0 = \{x_0\}$, and for each subset S_i, S_{i+1} is the set of nodes which can start once all the nodes in S_i have finished.

If all the tasks in subset 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, the starting time of each task in S_{i+1} is set to the time in which all their predecessor tasks in S_i have finished. In each subset $S_i = \{t_1, \ldots\}$, the scheduling algorithm assigns one task t_j to one processor from P. For each subset $S \neq S_0$, the algorithm performs as follows:

For each task $t_j \in S$ do:

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

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

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

Tasks are assigned to free processors. If no free processor exists at a given moment, the first processor to become idle is chosen. The need to make a choice in the non–deterministic Step 2 is one of the sources of the non optimality of the algorithm. In Step 3, T_q is chosen using a heuristic that tries to increase the occupation time of the processors.

6.2 The *Andp* Scheduling Algorithm

The **andp** scheduling algorithm [Her87] mimics the behavior of one of the *&–Prolog* schedulers. 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. To keep track of the local work, each processor is modeled as a couple $\langle T, L \rangle$ where T is as before, and L is the list of tasks generated by the processor. Roughly speaking, the scheduling algorithm tries first to execute tasks locally; if this is not possible, a task is stolen from another processor's list.

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.

Processor 0 is selected as having the initial task; thus at the beginning $L_0 = \{x_0\}$. The rest of the processors have empty stacks: $L_i = \emptyset, 0 < i < p$, and all of them are free: $T_i = 0, 0 \leq i < p$. The part of the scheduling algorithm that is in charge of getting work is as follows:

Step 1 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 2 If $L_p \neq \emptyset$ assign the first task $x \in L_p$ to processor p and go to Step 1.

Step 3 If $L_p = \emptyset$, find the processor q such that $T_q = \min_{\langle T_i, L_i \rangle \in N, L_i \neq \emptyset}(T_i)$. Assign the first task $x \in L_q$ to processor p and go to Step 2.

The generation of new work, after task x from the list of tasks L_q is assigned to processor p, is as follows:

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

Step 2 Set $T_p = T_p + l(x)$.

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

7 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 the visualization tool *VisAndOr* [CGH93]. Thus, *IDRA* can calculate 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 — as well as others which implement parallelism of a similar structure) using directly the trace files that *VisAndOr* accepts, without the need of any further processing.

The tool itself has been completely implemented in Prolog. Besides the computation of maximum and ideal speedups, *IDRA* can generate new trace files for ideal parallelism, which can in turn be visualized using *VisAndOr* and compared to 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* (and with *VisAndOr*), need not be generated by a real parallel system. This is a very interesting feature, in that it is possible to generate them with a sequential system augmented to dump information about concurrency. The only requirement is that the dependencies among tasks be properly reflected, and that the timings be accurate.

In some platforms accuracy in the timings has not been straightforward to obtain. Somo usual UNIX environments do not provide good access to the system clock: 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 were a significant portion of the total execution time of the benchmark, thus leading to incorrect results (sequential tasks being traced were noticeably longer than without tracing). To obtain accurate timings we used the microsecond resolution clock available in some Sequent multiprocessors [Seq87], which is not only very precise, but also memory mapped and can thus be read in the time corresponding to one memory access, with negligible effect on performance. For platforms in which the clock has a high but predictable access time, we had to develop a technique based on subtracting the accumulated clock access time from the timings.

The overhead of gathering the traces depends ultimately on the system executing the program being traced. For the *&–Prolog*/Muse systems, it typically falls in the range 0% – 30% — usually less than 20% — of the total execution time.

The time that a simulation takes depends, of course, on the trace being inspected. It can be substantially larger than the execution itself if the program executes many small tasks, and can be shorter than executing the actual program in the opposite case: few, large tasks.

8 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 for and–parallelism, and by *&–Prolog* and Muse for or–parallelism. 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 (and that of Muse in particular) usually make work available to parallel execution only for some choicepoints. This, in our approach, would not allow us to find out the maximum or ideal parallelism hidden in the program, since opportunities for performing work in parallel would be lost. This is why *&–Prolog*-generated or–parallel traces achieve better speedups than the corresponding ones generated by Muse: more tasks can be scheduled for parallel execution. On the other hand, the reason why the Muse scheduler does not schedule all possible tasks for parallel execution is that the added overhead would possibly result in poorer speedups.

8.1 Description of the Programs

We include a brief description of the programs used to test the tool, in order to help in understanding their behavior, both in simulation and in execution. The sequential execution time and the number of tasks generated by each benchmark program are shown in Table 2, as an indication of the program size. The figures that appear next to some of the benchmark names represent 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 with and–parallelism

  deriv performs symbolic derivation.

  occur counts occurrences in lists.

  tak computes the Takeuchi function.

  boyer adaptation of the Boyer–Moore theorem prover.

  matrix square matrix multiplications (the vector times vector multiplications are sequential tasks).

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

Program	Time (ms)	Number of tasks generated	
deriv	240	2109	
occur	1750	126	
tak	610	4744	
boyer	110	747	
matrix–10	170	321	
matrix–15	550	726	
matrix–20	1270	1270	
matrix–25	2460	2047	
quicksort–400	590	1230	
quicksort–600	1070	1500	
quicksort–750	1500	1700	
bpebpf–30	220	1395	
bpesf–30	180	90	
pesf–30	200	93	
		&–Prolog or traces	Muse traces
domino	130	1002	340
queens	70	458	176
witt	5090	1878	230
lanford1	160	458	130
lanford2	2090	2047	832

Table 2: Some information about each benchmark program

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.

queens computes all the solutions to the 5 queens problem.

witt is a conceptual clustering program.

lanford1 determines some elements needed to complete a Lanford sequence.

lanford2 similar to **lanford1**, but with different data structures.

8.2 Maximum Parallelism Performance

Tables 3 and 4 show the maximum speedup attainable according to the simulation, the number of processors at which this speedup is achieved, and the relative efficiency with respect to a linear speedup, i.e., efficiency $= \frac{\text{speedup}}{\text{processors}}$ for the programs mentioned above.

Programs which require a large number of processors despite the problem to be solved not being very big are usually those where tasks are small. This would suggest that a parallel system would need 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,[5] as we will see in Section 8.3 and Table 5, 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 the length of the tasks is variable and the execution structure is not homogeneous (i.e., **quick-**

[5]In addition, *&–Prolog* concept of local work allows to speed up programs with small granularity, since stealing local tasks is much cheaper than stealing foreign tasks

Program	Speedup	Processors	Efficiency
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 3: Estimated maximum and–parallelism

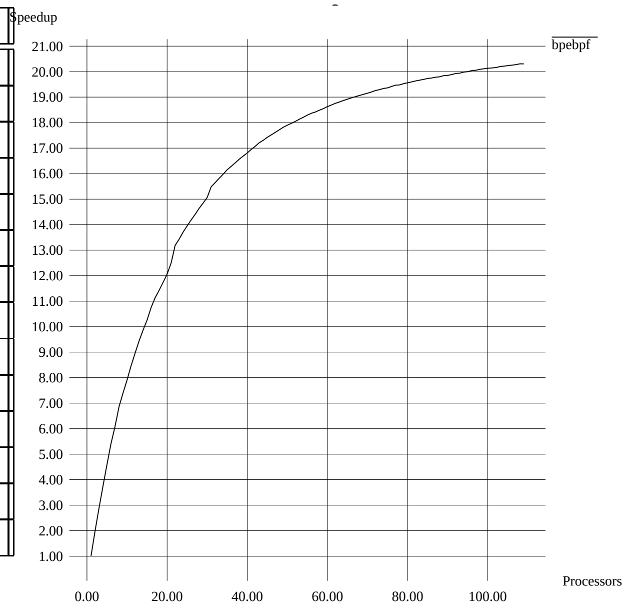

Figure 7: Computation of e

Program	Speedup	Processors	Efficiency
domino	32.01	59	0.54
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 4: Estimated maximum or–parallelism

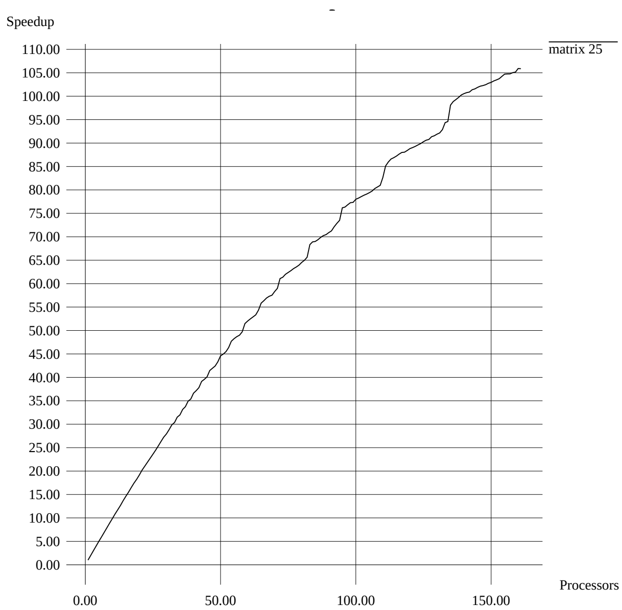

Figure 8: 25×25 matrix multiplication

sort), the expected maximum speedup achievable grows very slowly with the size of the problem. In the case of **quicksort**, the sequential parts caused by the partitioning and appending of the list to be sorted finally dominate the whole execution, preventing further speedups and giving an example that confirms once again Amhdal's law.

8.3 Ideal Parallelism Performance

For each benchmark we have determined the ideal parallelism and the actual speedups on one to nine processors (Table 5 and 6). For each benchmark, the rows marked *real* correspond to actual executions in real systems (*&–Prolog* for the and–parallel benchmarks, and Muse for the or–parallel ones). The rows marked *subsets* and *andp* correspond to simulations performed using those algorithms. There are two additonal subdivisions for each benchmark in the or–parallel case, under the column "Tracing System", which reflect in which system were gathered the traces.

The data obtained for and–parallelism 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 remarkably close to those obtained from the actual execution, which seems to imply that the simulation results are quite accurate

Program	**Scheduling Algorithm**	**Processors**								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
deriv	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	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	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	4.85	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	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	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	6.80	8.50
matrix–15	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	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	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	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	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	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	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	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	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 5: Ideal and–parallelism

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–750 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 Tables 3 and 4 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 somewhat worse than the prediction given by the simulation when *&–Prolog or* traces are used. This is not surprising, since Muse has an overhead associated with task switching (due to copying) that is not reflected in the traces. Moreover, the traces correspond to the case in which all *or* branches are available for parallel execution, whereas the traces generated by Muse only contain the branches that the Muse scheduler considered worthwhile for parallel execution. Thus, in the case of the *or* traces generated by *&–Prolog*, more (and smaller) parallel tasks (and potential parallelism) exist – thus the higher speedups predicted by the tool, which largely surpass those obtained from real executions. In the case of simulations using Muse traces, the predictions are more accurate (but then, they do not reflect the parallelism available in the benchmark, but rather that exploited by Muse). In general, the results show the simulation to be highly accurate and reliable. In fact, the system has been used successfully in several studies of parallelizing transformations [DJ94] and parallelizing compilers [BGH94b].

9 Conclusions and Future Work

We have reported on a technique and a tool to compute ideal speedups using simulations which uses 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 with those obtained from 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 technique can be extended for other classes of systems and execution models (also beyond logic programming), provided that the data which models the executions can be gathered with enough accuracy.

We 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 to 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.

10 Acknowledgments

We would like to thank S. Debray of U. of Arizona 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. We also would like to thank British Telecom, for a grant which provided the Sequent Symmetry used to carry out the experiments shown in this paper.

The research presented in this paper has been supported in part by ESPRIT project 6707 "PARFORCE" and CICYT project TIC93–0976–CE.

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*, pages 757–776. MIT Press, October 1990.

[AK91] Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT Press, 1991.

[BGH94a] F. Bueno, M. García de la Banda, and M. Hermenegildo. A Compara-

Program	Tracing System	Scheduling Algorithm	Processors								
			1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
domino	Muse	subsets	1.00	1.95	2.88	3.75	3.92	3.92	3.92	3.92	3.92
		andp	1.00	1.89	2.74	3.56	3.92	3.92	3.92	3.92	3.92
	&–Prolog	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
queens	Muse	subsets	1.00	1.91	2.72	3.41	3.41	3.41	3.41	3.41	3.41
		andp	1.00	1.87	2.54	3.41	3.41	3.41	3.41	3.41	3.41
	&–Prolog	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	Muse	subsets	1.00	1.12	1.17	1.19	1.19	1.19	1.19	1.19	1.19
		andp	1.00	1.08	1.12	1.12	1.12	1.12	1.12	1.12	1.12
	&–Prolog	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	Muse	subsets	1.00	1.95	2.83	3.62	4.20	4.62	4.62	4.62	4.62
		andp	1.00	1.89	2.67	3.37	4.32	4.62	4.62	4.62	4.62
	&–Prolog	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	Muse	subsets	1.00	1.85	2.55	3.15	3.73	4.30	4.77	5.12	5.50
		andp	1.00	1.91	2.50	2.94	4.02	4.51	5.51	5.85	5.88
	&–Prolog	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 6: Ideal or–parallelism

tive Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *Parallel Symbolic Computation*, pages 63–73. World Scientific Publishing Company, September 1994.

[BGH94b] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[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.

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

[DJ94] S. K. Debray and M. Jain. A Simple Program Transformation for Parallelism. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[GHPC94] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Program-*

ming, pages 93–110. MIT Press, June 1994.

[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.

[Her86] M. V. 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.

[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.

[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.

[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.

[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.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.