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

School of Computer Science
Technical University of Madrid

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 corresponding speedups. 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; 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,SCWY91,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 we try to find techniques to answer different questions, and in a relatively architecture-independent way: given a (parallel) execution paradigm, what is the maximum benefit that can be obtained from executing a program in parallel following that paradigm? 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)? 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. It appears that any approach for obtaining such answers has to resort to a greater or lesser extent to simulations.

* The research presented in this paper has been supported in part by ESPRIT project 6707 "PARFORCE" and CICYT project "IPL-D."

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

In [SH91] programs are executed by a high–level meta-interpreter/simulator which computes ideal speedups for different numbers of processors. This work is interesting in that it proposed the idea of comparing the ideal performance obtained simulations with that of actual systems. However, the simulator proposed does suffer from some drawbacks. Resolution steps are used as time units, thus causing some lack of accuracy in certain benchmarks (extra time can be added to the start and end of tasks, to somewhat compensate for that, and to allow simulating machine overheads). Also, the size of the executions is limited by the time and memory consumption of the interpretive method.

In [SK92] Prolog programs are instrumented to count the number of WAM [AK91] instructions executed, assuming a constant cost for each WAM instruction. Speedups are calculated by comparing the critical path for the parallel execution with the sequential execution length. Although this method can be more accurate than that of [SH91], it also has some drawbacks: only maximum speedups are computed, the type of instrumentation performed on the code does not allow taking control instructions into account, and the different amount of time that many WAM instructions may need at run–time is not taken into account. Finally, the problem of simulating large executions is only partially solved by this approach.

Our approach tries to overcome the precision and execution size limitations of previous approaches. We achieve both goals by placing the splitting point between execution and simulation at a different point: programs are executed directly in (instrumented) real systems. Simplified execution traces which contain accurate task timing and dependency information are generated during the program execution (even on only one processor).

Space limitations only allow a concise and rather informal presentation. A full version of this paper can be obtained from `http://www.clip.dia.fi.upm.es/`.

2 Parallelism in Logic Programming

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** [Con83]. We will briefly review some related concepts in the following sections.

Restricted And–parallelism: Restricted and–parallelism (RAP) [DeG84,HG91] refers to the execution of independent goals using a fork and join paradigm. Independent goals are those that meet some "independence conditions" at run–time (for example, variables are not shared, thus avoiding all possible Read–Write and Write–Write conflict). The only dependencies existing in RAP appear among the conjunction of goals executed in parallel and the goals before and after the parallel execution. Consider the *&–Prolog* [HG91] program below, where the

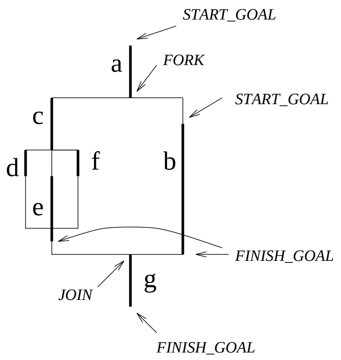

Fig. 1. And–parallel execution

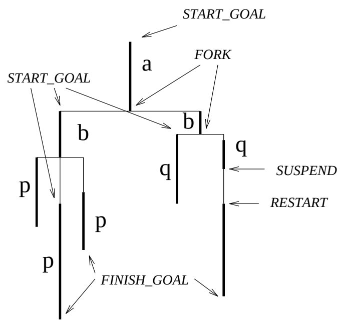

Fig. 2. Or–parallel execution

"&" operator, in place of the comma operator, stands for and–parallel execution (d, e and f 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 JOINs close FORKs in reverse order, and tasks are started either by START_GOAL or by JOIN ad finished either by FINISH_GOAL or by FORK. Under these conditions, a RAP execution can be depicted by a directed acyclic planar graph, where and–parallel executions appear nested.

Or–parallelism: Or–parallelism corresponds to the parallel execution of different alternatives of a given predicate. Since each branch 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. As an example, consider the following program, which has alternatives for predicates b, p and q:

```
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 a side–effect predicate or a *cut*, which would impose a serialization of the execution (such suspensions also appear in and-parallel traces). In terms of dependencies among events, FORKs are not balanced by JOINs. The resulting graph is thus a tree.[1]

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

3 Replaying the Essential Parallel Execution

To simulate alternative schedulings of a parallel execution we need a description of that execution, which must contain, at least, the length of each task and the relationships and dependencies which hold among the tasks. Such a description can be produced by executing programs in actual implementations instrumented to generate execution logs, or, with less accuracy, even using high–level simulators.

The descriptions of the executions are stored in the form of *traces*, which are series of *events*, gathered at run–time by the system under study, and carrying the necessary information with them. The events reflect *observables* (interesting points in the execution), and allow the reconstruction of a skeleton of the parallel execution. Figures 1 and 2 represent two parallel executions, in which some events have been marked at the point where they occur. The length of the vertical segments is intended to reflect the actual time taken by the sequential tasks and the scheduling delays.

3.1 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 other purposes as well.[2] This, and the fact that scheduling algorithms are usually formulated in terms of *job graphs* (see, e.g., [Hu61,HB88]), in which only tasks and their relationships are reflected (scheduling delays do not appear—or are assumed to be a part of the tasks themselves), makes it desirable to separate the simulation from the actual traces. Job graphs are obtained from traces in our system by using an intermediate representation (*execution graphs*) which allows making such transformations easily, and independently from the initial trace format. This translation can be parameterized to take into account actual or minimum scheduling delays, incrementing the usefulness of the tool.

3.2 Maximum Parallelism

The term *maximum parallelism* denotes the parallelism obtained with an unbound number of processors assuming no scheduling overheads, so that newly generated tasks can be started without any delay. Maximum parallelism is useful in order to determine the minimum time in which a program could have been executed while respecting the dependencies among tasks. Alternative parallelizations/sequentializations of a given program can thus be studied [DJ94,BdlBH94], as well as different algorithms for a given task, independently of machine limitations.

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. Unfortunately, obtaining both these numbers

[2] This is the case for the actual parallel systems that we study—see Section 4—where the traces used by our simulation were originally designed for visualization.

is an NP–complete problem [GJ79]. However, we can find out the maximum speedup simply by removing all scheduling times and "flattening" the trace. An upper bound on the minimum number of processors can also bee obtained from the maximum number of tasks active at a time. This 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.

3.3 Ideal Parallelism

Ideal parallelism corresponds to the speedup ideally attainable with a fixed number of processors. The task to processor mapping determines the actual speedups attained. Ideal parallelism can be used to test the absolute performance of a given scheduling algorithm for a fixed number of processors, and also to test the efficiency of an implementation, by comparing the actual speedups with those predicted by the simulator using the same scheduling algorithm as the implementation. Studying how the performance of a program evolves for a number of processors as large as desired gives also interesting information about the potential parallelism in a program. Another interesting issue which can be studied is the variation of inherent parallelism with problem size: frequently one wants to solve existing problems faster, but also to be able to tackle larger problems in a reasonable amount of time. In non–trivial examples the number of parallel tasks and the expected attainable speedups may not be easy to estimate, and problems in which the available parallelism does not increase with the problem size would not benefit from larger machines. Section 4 has illustrating examples.

As in 3.2, obtaining an optimal task to processor allocation is, in general, an NP–complete problem [GJ79]. To be able to deal with non–trivial executions, we will resort to 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.

We have implemented and tested two scheduling algorithms: the **subsets** algorithm [HB88], which groups the tasks into disjoint subsets which are scheduled (almost) independently, and the **andp** algorithm, which mimics the behavior of one of the *&–Prolog* schedulers. This scheduler tries to favor locality by assigning to a given processor the work which was created by itself. It also tries to increase the speed at which parallel tasks are created in recursive clauses [HC96].

4 *IDRA*: An Implementation and Its Use

A tool, named *IDRA* (IDeal Resource Allocation), has been implemented using the ideas sketched before. The traces used by *IDRA* are the same as those used by the visualization tool *VisAndOr* [CGH93]. The tool itself has been completely implemented in Prolog. Besides computing 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 ones.

The traces used with *IDRA* (and with *VisAndOr*) need not be generated by a real parallel system. It is possible to generate them with a sequential system augmented to dump information about concurrency (or even with a high–level

Table 1. Maximum and–parallelism

Program	Speedup	Procs.	Eff.
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–25	161.68	462	0.34
qsort–400	3.93	15	0.26
qsort–750	4.28	19	0.22
bpebpf–30	23.21	260	0.08

Table 2. Maximum or–parallelism

Program	Speedup	Procs.	Eff.
domino	32.01	59	0.54
queens	18.14	40	0.45
lanford1	19.72	44	0.44
lanford2	114.87	475	0.24

simulation of the execution paradigm under study). The only requirement is that the dependencies among tasks be properly reflected, and that the timings be accurate.

In our case, timing data is gathered by a modified Prolog implementation which ensures that the timing information is realistic. The implicit control of Prolog makes identifying the "interesting places" in the execution, and generating the corresponding events, automatic. 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.

In the following sections we will show examples of the use of *IDRA* on real execution traces. These traces have been generated by the *&–Prolog* system for and–parallelism, and by Muse and a slightly modified version of *&–Prolog* for or–parallelism. This modification was needed in order to obtain or–parallel traces with all possible or–parallel tasks: the Muse scheduler does not make all possible or–parallel tasks available for granularity reasons,[3] thus disallowing the correct simulation of ideal or maximum speedups. The *&–Prolog* modified version dumps traces which contain all or–parallel tasks available; therefore, *&–Prolog* or traces contain many more, smaller tasks than Muse traces.

4.1 Maximum Parallelism Performance

Tables 1 and 2 show the maximum speedup attainable according to the simulation, an upper bound on the number of processors required to achieve this speedup, and the relative efficiency (**Eff.**) with respect to a linear speedup, i.e., $\mathbf{Eff} = \frac{\mathbf{speedup}}{\mathbf{processors}}$.

Clearly, short executions which require a large number of processors usually have small tasks. This suggests that a parallel system would need some sort of granularity control to execute them efficiently (see, e.g. [LGHD94]). This turns out not to be always the case for real executions on shared memory multiprocessors with a small number of processors,[4] as we will see in Section 4.2 and Table 3, but would certainly be an issue in larger or distributed memory machines.

[3] This is common in schedulers for or–parallelism.

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

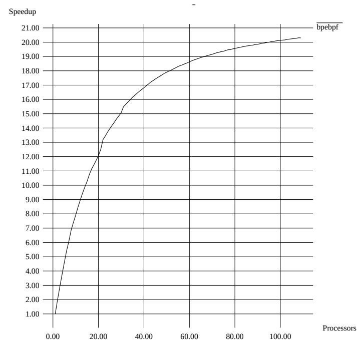

Fig. 3. Computation of e

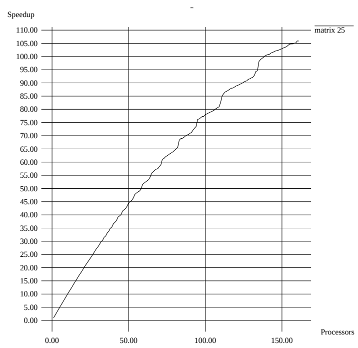

Fig. 4. 25×25 matrix multiplication

In programs with a regular structure (e.g., **matrix** — square matrix multiplications), potential speedups grow accordingly with the size (and the number of tasks) of the problem: the efficiency remains approximately constant. However, in programs with a non–homogeneous execution structure (i.e., **qsort** — Hoare's *QuickSort* algorithm), the expected maximum speedup achievable grows slowly with the size of the problem, and the efficiency tends to decrease with the problem size. In the case of **qsort**, the sequential parts of the program would finally dominate the whole execution, preventing further speedups (Amdhal's law).

4.2 Ideal Parallelism Performance

For each benchmark we have determined the ideal parallelism and the actual speedups on one to nine processors (Tables 3 and 4). The rows marked *real* correspond to actual executions in *&–Prolog* (for the and–parallel benchmarks) and Muse (for the or–parallel ones). Two additional subdivisions for each benchmark in the or–parallel case, under the column "Tracing System" reflect in which system the traces were gathered.

&–Prolog actual traces were gathered with a low–overhead version of the *&–Prolog* scheduler with reduced capabilities, so that the **andp** simulation and the actual execution be as close as possible. The remarkable similarity of the simulated and the actual speedups supports our thesis that the simulation results are accurate. The **subsets** scheduling algorithm performs slightly better, but due to its non optimality, it is surpassed sometimes by the **andp** algorithm and by *&–Prolog* itself (e.g., in the **qsort–750** benchmark). Sometimes the actual *&–Prolog* speedups are slightly better than the **andp** simulation: this is understandable, given the heuristic nature of these algorithms.

Benchmarks with good performance in Tables 1 and 2 show good speedups here also. But the inverse is not true: benchmarks with low efficiency in maximum parallelism can perform well in actual executions: for example the simulated speedups for the benchmark **bpebpf** (Figure 3), are quite good for a reduced number of processors (see Table 3). As expected, more regular benchmarks display a good, predictable behavior; for example, **matrix–25** has a larger granularity and shows almost linear speedups with respect to the number of pro-

Table 3. Ideal and–parallelism

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–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
qsort–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
qsort–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

cessors (Figure 4). 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, given the already mentioned differences between Muse traces and *&–Prolog* or traces. Simulations which use Muse traces show more accurate predictions, but they reflect the parallelism exploited by Muse instead of the parallelism available in the benchmark.

5 Conclusions and Future Work

We have reported on a technique and a tool to compute ideal speedups using simulations which use as input data information about executions gathered using real systems, or even high–level simulations. We have applied it to or– and independent and–parallel benchmarks, and compared the results with those from actual executions. In general, the results show the simulation to be highly accurate and reliable, and its results match quite well those obtained from actual

Table 4. Ideal or–parallelism

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
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

systems (in particular, those obtained from the *&–Prolog* system). In fact, the system has been used successfully in several studies of parallelizing transformations [DJ94] and parallelizing compilers [BdlBH94].

We believe that both the core idea and the actual tool developed can be applied to any parallel execution paradigm (not only logic programming) whose task structure conforms to any of those in our initial assumptions, provided that the data which models the execution can be gathered accurately enough.

We plan to modify the simulator in order to support other execution paradigms with more complex task structures (e.g., Andorra–I [SCWY91], ACE [GHPSC94], AKL [JH91], 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 and copying overhead.

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.

[BdlBH94] 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.

[GHPSC94] 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 Programming*, pages 93–110. MIT Press, June 1994.

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

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

[HC96] M. Hermenegildo and M. Carro. Relating Data–Parallelism and (And–) Parallelism in Logic Programs. *The Computer Languages Journal*, 22(2/3):143–163, July 1996.

[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.

[LGHD94] P. López-García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In Hoon Hong, editor, *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.

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

[SCWY91] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–93. ACM, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[SH91] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *International Logic Programming Symposium*, pages 135–151. 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.