A Simulation Study of Or- and Independent And-parallelism

Kish Shen[§]
Computer Laboratory
Cambridge University, UK
ksh@cl.cam.ac.uk

Manuel V. Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid, Spain
herme@fi.upm.es / herme@cs.utexas.edu

Abstract

Although studies of a number of parallel implementations of logic programming languages are now available, the results are difficult to interpret due to the multiplicity of factors involved, the effect of each of which is difficult to separate. In this paper we present the results of a high-level simulation study of or- and independent and-parallelism with a wide selection of Prolog programs that aims to facilitate this separation. We hope this study will be instrumental in better understanding and comparing results from actual implementations, as shown by an example in the paper. In addition, the paper examines some of the issues and tradeoffs associated with the combination of and- and or-parallelism and proposes reasonable solutions based on the simulation data.

Keywords: Parallelism, Logic Programming, Performance Analysis, Or-parallelism, Independent And-parallelism, Simulation, Compile-time Parallelization.

1 Introduction

Recent interest in implicit parallel execution of logic programs has lead to the development of many execution models, with research concentrating on two promising approaches (or-parallelism and and-parallelism) or some combination thereof. Results associated with various implementations have been published (for example, [2, 1, 20, 11, 14, 16]). However, these results are generally difficult to interpret: Firstly, these studies have understandably concentrated on programs that are reasonably suited to the type of parallelism being exploited. However, it also seems important to have a broader view of the nature and availability of the parallelism across a more representative set of programs. Secondly, in the results themselves the effects of at least two factors are combined: the *inherent* amount of parallelism in the benchmarks used with respect to the idealised model of parallelism under consideration, and the (lower level) impact of the implementation itself. Ideally these two factors should be separated. Most performance studies have concentrated on the low level factors. Comparatively little effort has been devoted to the equally important higher level factors, the subject of this paper. We present a high-level

[§]Address from Oct. 1991: Department of Computer Science, University of Bristol, Bristol, U.K.

simulation study of the amount and characteristics of the or- and (independent) and-parallelism in a wide selection of Prolog programs, from simple benchmarks to medium-sized applications. The simulation approach provides a measure of the ideal or inherent amount of parallelism which is largely independent of implementation effects. Furthermore, a simulation study is more flexible than studies associated with real implementations as a simulation is not constrained by the available hardware (*e.g.* the number of processors in a parallel system), and unlike a real implementation, results are not perturbed by making measurements on it.

A high-level model of or-parallelism is relatively simple to define, as the ideal or inherent amount of or-parallelism can be defined as running all alternatives in parallel.[1] For independent and-parallelism, however, the situation is more complex. There are two basic issues: what is understood by "goal independence", and how and when such independence is detected and the corresponding goals scheduled for parallel execution. Naïve approaches to solving each of these problems are inefficient or even intractable in practice [7]. Moreover, there is an even more open issue as to how or-parallelism and and-parallelism are to be combined. Our choices regarding these issues are the subject of the first sections of the paper, after a brief overview of related work. We then describe the simulation tools. A brief description of the experiments performed and results obtained is followed by a discussion of these results. A reasonable familiarity with Prolog, logic programming, and and- and or-parallelism is assumed throughout the paper.

2 Related Work

Other high-level studies of parallel Prolog known to us include [4, 19, 13]. These are all studies of or-parallelism only. Very little high-level information is available for independent and-parallelism, and even less for independent and-parallelism combined with or-parallelism. The closest to a high-level study of both types of parallelism is the work by Fagin [8]. However, on balance, his study is relatively low level and quite specific to his PPP model. This is especially so because or-parallelism is quite severely limited under and-parallelism in his scheme. Furthermore, we feel that it is important to examine a greater number of more realistic programs than was in his study.

3 Model of parallelism simulated

As stated in the introduction, it is assumed that the reader is reasonably familiar with the general issues in parallelism and Prolog, so only brief descriptions of the parallelism simulated will be given here.

First, we employ interchangeably the terms **worker** and **agent** (as used in Gigalips project [23] and the &-Prolog [11] project, respectively) to refer to the entities that perform the computation, or **work**. Parallelism is achieved by allowing several workers/agents to simultaneously explore the search-tree of a program. Each worker explores the search-tree in much the same way as a sequential Prolog engine: depth-first, left-to-right. Generally, each worker will be assigned to a different part of the search-tree, and thus the search-tree can be thought of as being divided into "chunks" of sub-tree, with each sub-tree being executed sequentially.[2] Each such sub-tree is referred to as a **task**. When a worker finishes exploring its sub-tree, it may then start on an unexplored part of the sub-tree. This process is referred to as **task switching**.

As a worker works on a task, opportunities for parallelism are generated – *i.e.* other workers can come and "steal" part of the sub-tree by splitting it. If not, the work would (eventually) be

[1]For full Prolog, those alternatives which will eventually be pruned should not be executed at all in the ideal case.

[2]Of course, each sub-tree may be only one resolution long, and so it does not matter if it is sequential or not in this case.

done by the worker itself. Conceptually, the search-tree can be thought of as being annotated with **sources** of parallelism, which generate **available nodes**, *i.e.* points where parallelism is possible. Available nodes may be of two types: **available or-nodes**, and **available and-nodes**. Available or-nodes allow goals (**or-goals**) to be run in or-parallel, and available and-nodes allow sibling-goals within a clause (**and-goals**) to be run in and-parallel with each other.

As stated in the introduction, it is difficult to model independent and-parallelism without making assumptions about the detection and selection for scheduling of the available and-goals. For our study we selected the restricted and-parallelism (RAP) rule, first proposed by DeGroot [6] and refined (backward semantics) by one of us [10]. Parallelism is specified by generalised "Conditional Graph Expressions" (CGEs) where conditional tests are used to determine whether the goals are to be executed in parallel or not. The choice of RAP was influenced by the availability of an automatic annotator for this type of parallelism [24] and of an actual implementation (&-Prolog) with which to contrast the results of the simulations.

3.1 Avoiding recomputation

With independent and-parallelism, the opportunity arises to perform less work than in a sequential system, as each independent and-task need be performed once only. That is, or-branches inside a CGE can reuse independent and-tasks generated in "earlier" or-branches. Thus, the most general way to combine and- and or-tasks is to compute all the or solutions to each and-task once, and then form all the possible combinations of joining these tasks to produce all the solutions that are produced in a sequential system. This is essentially what was proposed for example in systems such as PEPSys [2] and the AO-WAM [9]. However, such a method for combining independent and-parallelism and or-parallelism is complicated both for an actual implementation and a simulation.

3.2 Combining the parallelism

In addition to the implementation complexity of reusing and-tasks, the *unrestricted* combination of independent and- and or-parallelism adds both to the conceptual and implementation complexity. An alternative is to restrict the use of parallelism in some way when they are combined. This leads to simpler schemes, but, obviously, at the expense of some parallelism. Examples of such approaches are those of Conery [5], Fagin [8], or Biswas *et al.* [3]. Many other restriction schemes are possible.

In this study, the effects of restricted and unrestricted combinations of and- and or-parallelism were studied using two schemes: in the first scheme, which we called "no or-under-and", or-parallelism is not allowed within and-parallelism. This prevents the combinatorial explosion in the usage of resources, but limits parallelism. In the second scheme, which we called "or-under-and", or-parallelism is allowed within and-parallelism. When an or-alternative within an and-goal is completed successfully, it starts executing the and-goals that follow it in the clause in and-parallel. In effect, the and-parallelism within each or-branch is handled in exactly the same way as the "point backtracking" method in the RAP-WAM [10].

A further difference between the two schemes mentioned above is that in the "no or-under-and" scheme there is reusage of independent and-goals. In the "or-under-and" scheme the and-goals are not reused. Instead such goals are recomputed as in sequential Prolog. It is important to point out that the issue of recomputation and that of restriction in the way parallelism can be combined are orthogonal. This allows us to infer conclusions about other models from only two simulation schemes. These two schemes were chosen to allow us to make inferences about a wide variety of schemes.

4 The simulator

The simulator is a greatly modified version of the or-parallel simulator described in [17] so that it can deal with independent and-parallelism (in the form of RAP) as well as or-parallelism. The actual model used for the simulator is an idealised version of the RAP-WAM for independent and-parallelism, with or-parallelism also being idealised.

In this study, we have been able to simulate programs that are more than 10 times larger than the largest examples in [19], so a much wider range of programs has been examined.

4.1 Assumptions

Like in the original or-parallel simulator, as few assumptions about the underlying hardware as possible are made, in order to make the results applicable to as wide a range of models as possible. The speedups can thus be regarded in some way as the "ideal speedup" attainable with the program given the annotation. The basic time unit used for the simulation is a resolution, which is assumed to be the same for all resolutions. This and other assumptions made in the original simulator are discussed in detail in [17], and due to lack of space will not be repeated here. Additional assumptions associated with and-parallelism and the combination with or-parallelism are:

- The CGE tests for groundness and independence were assumed to be compiled into the underlying abstract machine instructions. Tick's study of the sequential WAM suggests that about 15 WAM abstract machine instructions are executed per procedure invocation (see [21, table 3-3]). We assume the basic cost of each CGE test is 1/15 the cost of a unification, plus extra cost (again at 1/15 of a unification) per level of recursion needed to traverse the structures being tested. This is then rounded up to the next larger integer to arrive at a cost in terms of unifications. For the work reported here, the tests exhaustively traverse the terms.
- At the end of a task, a worker is free to switch task to any available node. If the worker was an and-task (*i.e.* picked an available and-node), it first tries to pick any sibling and-node that is still available to the left of the and-task it just completed. If no such node exists, it is free to choose any available node.

4.2 Generation of results

Prolog programs were first run through a CGE annotator program, generating Prolog code which was annotated with CGEs (&-Prolog). The annotator used for this simulation was the "mel" annotator introduced in [24] and described in [15]. This was done only with the more complex programs, since the simpler programs could be easily annotated by hand. Then a checker program was used to check the validity of the CGEs.[3] The annotated program was then converted to the format used by the simulator and simulated.

5 Summary and discussion of results

5.1 Programs simulated

Two broad categories of programs were simulated. The first category includes simple benchmark-type programs. Most of these were used to benchmark sequential systems, and were not specifically designed for exhibiting parallelism. These programs are useful as they are relatively simple

[3]This step was quite instrumental in flagging a number of bugs in early implementations of the "mel" annotator.

and can thus be easily analyzed. The second category includes existing applications, running relatively simple input to get the execution time to a level suitable for simulation. These programs are more representative of realistic programs.

For some of the programs simulations were performed for different sets of input data in order to observe the sensitivity of the simulation results to the size and nature of the input. In other cases a few similar programs solving the same problem but using different algorithms were studied. In the latter case we distinguish programs in the tables by slight variations of the program name. In the former, by providing a different label in brackets.

Detailed descriptions of the programs will not be given here. Most are relatively self-explanatory. See [18, chapter4] for more details. The examples from *qsort(20)* to *hanoi* can be considered as the benchmark-type programs. Of the application-type programs, the *compiler(cp3)* example is based on the Prolog compiler by Van Roy [22]; *boyer_nsi(2)* is the "boyer" theorem proving program which exploits non-strict independence as defined in [12], tp is based on a propositional theorem prover by Ross Overbeek. The two *orsim* examples are a version of the simulator used for the original or-parallel study, modified for exploiting independent and-parallelism. The two *sim* examples are the program used for this simulation study.

5.2 The tables of results

Each program was simulated using the two ways of combining and- and or-parallelism, and with and-parallelism only (with no reusage of goals) and or-parallelism only. For each of these, the program was simulated for a range of workers, generally from one worker up to the maximum number of workers the program could take advantage of with that form of parallelism. The simulation was first done assuming no overhead. The simulation was then also done assuming 4 units of overhead at the start and finish of a task (giving a total of 8 units of overhead per task). Also, various tests to examine various other aspects of the parallelism and annotations were performed. Some of these results are summarised in the tables in figures 1, 2 and 3. The first table records the "static" data obtained from the static part of the simulator, and the other two record the "dynamic" data. The meaning of the columns is as follows:

name Name of program simulated.

Σ **res.** Total number of resolutions (successes and failures) in program when executed by Prolog, *i.e.* assuming no reuse of computation or CGE test overhead.

sol. Number of solutions given by the program.

pCGE The number of run-time invocations of CGEs whose test succeeded. This includes the unconditional CGEs.

sCGE The number of run-time invocations of sequential CGEs, *i.e.* those CGEs whose test failed.

uCGE The number of run-time invocations of unconditional CGEs.

Σ **cost** The cost, in number of resolutions, of the CGE tests. The number in brackets is the percentage cost with respect to Σ res.

Σ **reused** The size, in number of resolutions, of the reused resolutions. The number in brackets is the percentage size with respect to Σ res.

max. perf. "Maximum performance" in terms of the maximum speedup, and the minimum number of workers at which this is achieved (the "demand"), for the particular form of parallelism being studied. The format is: <speedup> $\times$@<number of agents>

The number given assumes that all available or-nodes in the search tree are allowed to run in or-parallel, and also assumes that there is no overhead. It is also *with respect to* *the sequential execution with no CGE annotations* (*i.e.* "actual speedups"). $\sim$ is used to indicate a value that has been estimated by interpolation between two actually simulated values.

half perf. The number of workers that are needed to achieve approximately half the numeric value of maximum speedup for the particular form of parallelism. The same format as "max. perf." is used.

ratio This is $\frac{t_h^0}{t_h^8} \times 100$ where t_h^0 is the time for executing the program with 0 overhead and t_h^8 the time for 8 units of overhead (per task). Both time measurements are made with the number of workers in "half perf."[4] This is a measure of sensitivity of the program to overhead. The closer the ratio is to 100%, the less sensitive it is. "half perf." number of workers were used as it was considered to be a representative figure for the program. However, if the program has insignificant amounts of parallelism, such that "half perf." occurs at 1 worker, measuring the overhead at 1 worker would give a ratio very close to 100%. For these programs, the "ratio" figure is computed using the speedups at "max. perf." number of workers. Such data are marked by a "*".

name	Σ res.	sol.	pCGE	sCGE	uCGE	Σ cost	Σ reused
qsort(20)	307	1	20	0	20	0 (0%)	0 (0%)
qsort(100)	2490	1	100	0	100	0 (0%)	0 (0%)
serialise	504	1	9	0	9	0 (0%)	0 (0%)
numbers	898	16	0	0	0	0 (0%)	0 (0%)
4Queens1	824	2	144	0	144	0 (0%)	0 (0%)
4Queens2	377	2	48	14	29	58 (15.4%)	42 (11.1%)
map1	3662	18	505	28	409	344 (9.4%)	152 (4.2%)
atlas	2678	4	1	0	1	0 (0%)	1679 (62.7%)
deriv	2874	1	483	0	483	0 (0%)	0 (0%)
vmatrix(10)	326	1	110	0	110	0 (0%)	0 (0%)
tak	21356	1	1186	0	1186	0 (0%)	0 (0%)
hanoi	2560	1	511	0	0	1013 (39.6%)	0 (0%)
warplan(wq1)	2039	1	81	17	16	160 (7.85%)	0 (0%)
warplan(wq2)	3131	1	74	56	17	179 (5.72%)	0 (0%)
compiler(cp3)	13374	1	56	0	16	80 (0.598%)	0 (0%)
boyer_si(1)	2749	1	2	168	2	168 (6.11%)	0 (0%)
boyer_si(2)	28056	1	5	2180	5	2180 (7.77%)	0 (0%)
boyer_nsi(2)	30486	1	2436	0	2436	0 (0%)	0 (0%)
tp	10273	1	158	59	158	59 (0.574%)	0 (0%)
chatp(cq1)	1204	1	50	27	35	55 (4.57%)	0 (0%)
chatp(cq2)	1067	1	52	25	40	42 (3.94%)	0 (0%)
chatp(cq3)	1356	1	77	26	65	50 (3.69%)	0 (0%)
sim(sp1)	9465	1	234	0	234	0 (0%)	0 (0%)
orsim(sp1)	9197	1	21	0	21	0 (0%)	0 (0%)
sim(sp2)	35346	1	877	0	877	0 (0%)	0 (0%)
orsim(sp2)	34117	1	66	0	66	0 (0%)	0 (0%)
annotator	14481	1	15	0	15	0 (0%)	0 (0%)

Figure 1: Summary of Static data from simulations

5.3 Discussion of the tables

The tables show that many of the programs do exhibit speedups with either or-parallelism or independent and-parallelism. However, neither are ubiquitous; indeed, a few of the "real" applications do not have much of either parallelism in it. Both independent and-parallelism and or-parallelism seem to be present in most programs, though in some cases they can be in insignificant amounts. Both types of parallelism have obvious areas of application where their

[4]Note that "ratio" was called "over." in [19]. We feel that "ratio" is a more accurate name.

name	And only			Or only		
	max. perf.	half perf.	ratio	max. perf.	half perf.	ratio
qsort(20)	1.56×@3	1×@1	85.7%*	1.25×@2	1×@1	65.0%*
qsort(100)	2.8×@8	1.7×@2	98.8%	1.34×@2	1×@1	59.4%*
serialise	1.08×@4	1×@1	94.9%*	2.0×@6	1.0×@1	63.4%*
numbers	1×@1	1×@1	99.1%	1×@1	1×@1	99.1%
4Queens1	1.27×@5	1×@1	77.8%*	18.7×@52	9.1×@10	75.8%
4Queens2	0.97×@3	0.87×@1	76.1%*	7.25×@15	3.5×@4	82.9%
map1	1.22×@6	0.91×@1	75.0%*	41.1×@59	20.6×@26	78.4%
atlas	1.00×@2	1×@1	99.7%	243×@576	116×@185	41.5%
deriv	84.5×@∼248	42.3×@60	50.4%	1×@1	1×@1	100%
vmatrix(10)	9.06×@18	4.66×@6	45.2%	1×@1	1×@1	100%
tak	45.6×@∼396	22.9×@30	76.9%	1.13×@2	1.0×@1	100.0%
hanoi	52.3×@427	26.1×@53	61.5%	1×@1	1×@1	100%
warplan(wq1)	1.46×@10	0.93×@1	88.3%*	8.90×@∼19	4.71×@5	72.0%
warplan(wq2)	1.09×@4	0.95×@1	94.5%*	12.8×@∼30	6.75×@7	77.9%
compiler(cp3)	7.48×@15	3.84×@4	98.2%	2.49×@8	1.66×@2	63.3%
boyer_si(1)	0.98×@3	0.97×@1	99.7%*	1.18×@5	1×@1	76.9%*
boyer_si(2)	0.94×@4	0.94×@1	99.6%*	1.23×@5	1×@1	72.4%*
boyer_nsi(2)	12.77×@∼74	6.54×@8	82.5%	1.20×@3	1×@1	75.3%*
tp	1.14×@4	0.99×@1	97.6%*	1.17×@5	1×@1	96.7%*
chatp(cq1)	1.01×@3	0.96×@1	94.4%*	1.67×@20	1×@1	72.7%*
chatp(cq2)	1.01×@3	0.96×@1	92.8%*	1.84×@17	1×@1	72.0%*
chatp(cq3)	1.03×@3	0.96×@1	94.1%*	2.18×@27	1.49×@2	84.2%
sim(sp1)	1.21×@4	1×@1	94.2%*	1.29×@4	1×@1	85.4%*
orsim(sp1)	1.14×@2	1×@1	99.9%*	1.29×@5	1×@1	87.5%*
sim(sp2)	1.12×@5	1×@1	97.0%*	1.47×@4	1×@1	83.4%*
orsim(sp2)	8.32×@∼20	4.44×@6	99.5%	1.47×@5	1×@1	87.9%*
annotator	10.0×@∼16	4.88×@6	98.8%	1.28×@5	1×@1	99.9%

Figure 2: Summary of dynamic data for and- & or- parallelism from simulations

name	Or-under-and			No or-under-and		
	max. perf.	half perf.	ratio	max. perf.	half perf.	ratio
qsort(20)	2.0×@5	1×@1	56.7%*	1.8×@3	1×@1	68.1%*
qsort(100)	3.7×@14	1.8×@2	83%	3.1×@8	1.8×@2	88.7%
serialise	2.2×@9	1×@1	59.1%*	1.3×@5	1×@1	82.0%*
numbers	34.5×@81	17.3×@21	48.6%	34.5×@81	17.3×@21	48.6%
4Queens1	45.8×@76	22.9×@27	48.6%	45.8×@76	22.9×@27	48.6%
4Queens2	6.9×@25	3.7×@5	70.1%	1.50×@4	0.96×@1	71.2%*
map1	63.1×@202	31.6×@42	58.9%	5.15×@26	2.46×@3	91.7%
atlas	243×@552	122×@177	37.3%	12.6×@24	5.89×@3	56.9%
deriv	84.5×@∼248	42.3×@60	50.4%	84.5×@∼248	42.3×@60	50.4%
vmatrix(10)	9.06×@18	4.66×@6	45.2%	9.06×@18	4.66×@6	45.2%
tak	≥54.5×@300	≥29.5×@40	78.2%	46.0×@∼396	27.7×@∼40	73.2%
hanoi	52.3×@427	26.1×@53	61.5%	52.3×@427	26.1×@53	61.5%
warplan(wq1)	7.5×@∼20	3.46×@4	75.9%	1.66×@6	0.93×@1	83.1%*
warplan(wq2)	11.7×@∼30	5.53×@6	71.8%	1.98×@∼19	0.95×@1	89.1%*
compiler(cp3)	16.9×@∼60	8.88×@10	78.3%	7.48×@15	3.84×@4	98.2%
boyer_si(1)	1.16×@7	0.94×@1	77.9%*	1.14×@3	0.94×@1	79.2%*
boyer_si(2)	1.18×@12	0.93×@1	73.4%*	1.17×@4	0.86×@1	73.6%*
boyer_nsi(2)	16.54×@∼74	8.50×@10	66.3%	12.87×@∼74	6.56×@8	82.1%
tp	1.38×@7	0.99×@1	93.6%*	1.38×@7	0.99×@1	93.6%*
chatp(cq1)	1.75×@18	0.96×@1	70.2%*	1.51×@12	0.96×@1	79.5%*
chatp(cq2)	1.87×@20	0.96×@1	69.1%*	1.58×@11	0.96×@1	78.0%*
chatp(cq3)	2.32×@27	1.51×@2	85.6%	1.93×@10	0.96×@1	81.6%*
sim(sp1)	1.50×@5	1×@1	82.1%*	1.37×@4	1×@1	86.0%*
orsim(sp1)	1.44×@6	1×@1	87.0%*	1.15×@5	1×@1	99.1%*
sim(sp2)	1.67×@8	1×@1	79.4%*	1.52×@5	1×@1	86.4%*
orsim(sp2)	10.7×@∼43	5.13×@6	94.8%	8.59×@18	4.51×@6	97.9%
annotator	12.5×@25	6.49×@8	86.4%	10.0×@∼16	4.88×@6	98.8%

Figure 3: Summary of dynamic data for combined and/or parallelism from simulations

exploitation results in a significant amount of speedup. Or-parallelism is present in programs that require substantial searching, such as the *warplan* programs. Independent and-parallelism is present in algorithms which are "divide and conquer" in nature – such as *compiler(cp3)* and *annotator*. Not many programs that contain significant amounts of both independent and- and or-parallelism were found. *compiler(cp3)* was the only program which approached having significant amounts of both types of parallelism. Nevertheless, it is interesting to see how the two types of parallelism interacted.

In all programs studied, the "or-under-and" method of combining and- and or-parallelism gave better or equal speedups as "no or-under-and". This shows that banning or-parallelism inside and-parallelism is too drastic a restriction. The gain of reusing goals by "no or-under-and" is insufficient to compensate for the loss of parallelism. In fact, of the programs tested, only a few were able to benefit from reusing goals, with *atlas* being the only one to gain significantly (63.5% of all resolutions). This suggests that, in general, although there is or-parallelism inside and-parallelism (otherwise "or-under-and" should be no worse than "no or-under-and"), not much of it leads to success (as otherwise there should be more reused goals).

It seems that "or-under-and" is quite a good compromise method for combining and- and or-parallelism – it avoids the complexities of allowing unrestricted or-parallelism under and-parallelism with full reusage of goals, with hopefully small loss in speed by needing to recompute the reused goals – note that some of this loss can be regained by the extra amount of parallelism.

In programs that contained both types of parallelism, we observed that in some cases the speedup of "or-under-and" is greater than the product of the speedups of "and only" and "or only". This is especially apparent in *4Queens1*. To our knowledge, this "supermultiplicative" effect was first reported by Fagin [8], and is attributed to the presence of various speedup techniques in a program affecting independent parts of the program. In his case, it was due mainly to the presence of intelligent backtracking. In our case, the effect is due to or-parallelism and independent and-parallelism only. Furthermore, the extent of independence between the two types of parallelism can be seen by observing the difference between the speedups obtained from "or-under-and" and "no or-under-and" – this is because "no or-under-and" restricts or-parallelism within and-parallelism, thus reducing the amount of available parallelism and speedup, so the more independent the two forms of parallelism, the less the reduction in speedup of "no or-under-and", and the closer it is to the "or-under-and" speedup.[5] We observe that in programs with two forms of parallelism, the smaller the difference between the two methods of combining the parallelism, the more pronounced the supermultiplicative speedups are. This suggests that the supermultiplicative speedup is indeed due to the forms of parallelism being independent of each other.

5.4 More detailed look at the results

The summary tables are not sufficient to show some of the observations made during the study. Some of these observations will be presented in this section, observations made in [19] will not be repeated here.

- For both or-parallelism and independent and-parallelism, and also for the two methods of combining them, the speedup diverges from the ideal 1-to-1 speedup relatively quickly, especially if overhead is considered (although this effect can of course be "pushed forward" by increasing the sizes of the programs). We think this is due at least partly to the fact that in many cases, especially for the larger, more realistic programs, the granularity of the parallelism is very fine, and occurs in small "bursts". Figure 4 shows such an example.

- The speedups for "or-under-and" show that combining and- and or-parallelism can lead to significant increase in performance if both types of parallelism are present in the program.

[5]This holds if there is little or no reusage of goals in the "no or-under-and" method. This is indeed the case as already discussed.

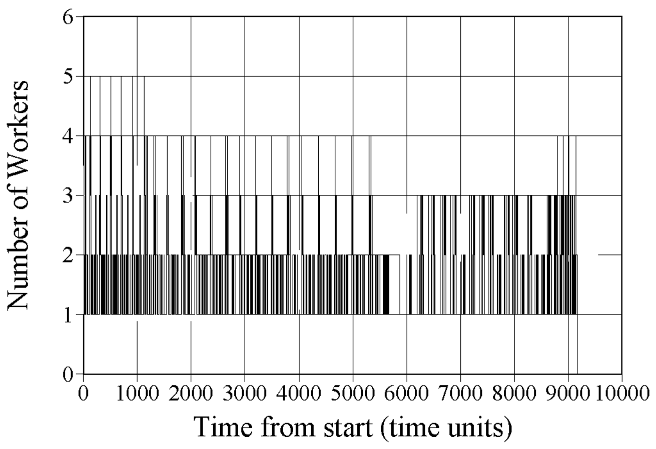


Figure 4: Execution profile for *sim(sp1)*, "or-under-and", no overhead

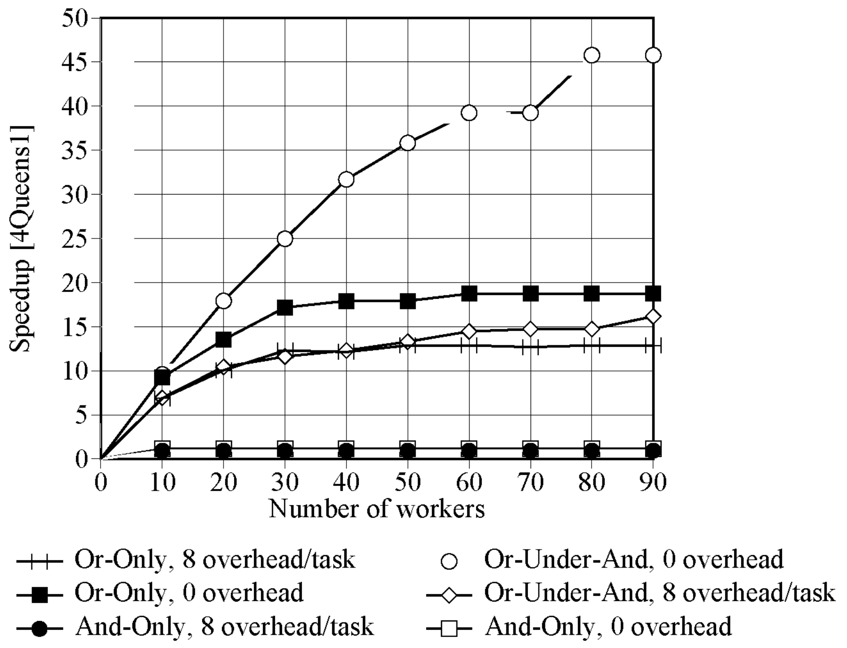
Examples of the speedups with the various types of parallelism are shown in figure 5. This shows the speedups for *4Queens1* and *compiler(cp3)*. With no overheads, it can be seen that the speedup from "or-under-and" is significantly higher than that obtained by other means. The graphs show that the speedups with "or-under-and" continue to increase after the other methods have flattened out; and this is most striking in *4Queens1*, where the or-parallelism does not overlap with the and-parallelism at all. The difference is all the more remarkable as and-parallelism on its own gives a maximum speedup of about 1.2 only.

Figure 6 shows the execution profiles for *4Queens1* under "or-under-and", "and only", and "or only". This clearly shows that the and-parallelism occurs after the or-parallelism, and in fact the main effect is to "fold" the or-parallel branches together, thus greatly increasing the effectiveness of both forms of parallelism.

However, it should be noted that with overheads, the advantage of "or-under-and" decreases significantly, and in the case of *compiler(cp3)*, and-parallelism on its own gave better speedups beyond about 7 workers. The reason for this is that the or-parallelism in this case, and and-parallelism in the case of *4Queens1*, is very fine grained and is strongly affected by the overheads. In general, though, we can say that combining or- and and-parallelism does offer us the opportunity to increase speedups of programs significantly by more effectively utilizing both forms of parallelism.

- Significant amounts of non-strict independent and-parallelism seem to exist in some programs. *boyer_nsi(2)*, exploiting non-strict independent and-parallelism, gave much better speedups than the *boyer_si(2)* running the same data with strict independent and-parallelism. This simulation result was obtained before &-Prolog was operational, and gave us confidence in the possibility of non-strict independent and-parallelism, confirmed by actual &-Prolog speedups reported in [12]. *tp* was also found to contain non-strict independent and-parallelism – what little independent and-parallelism that exists is almost all non-strict.

- The amount of parallelism obtained can depend greatly on the particular problem being solved. In some cases, the amount of parallelism depends on the size of the problem (*annotator* is probably a good example of this). Some other programs are not very sensitive to the problem size (*e.g. sim* – the and/or simulator). However, many programs have more complex dependencies on the problem being solved.



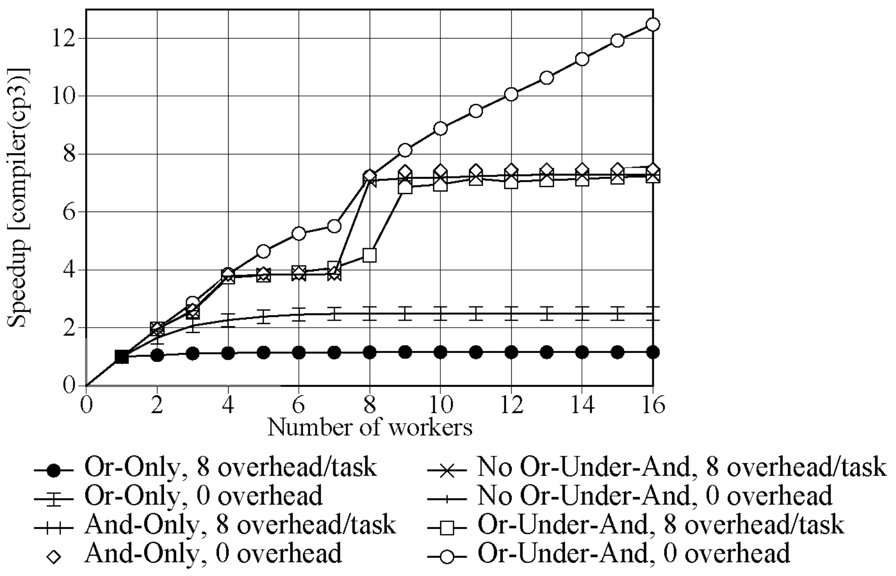


Figure 5: Speedups for *4Queens1* and *compiler(cp3)*

For example, for *orsim*, or-parallelism in the simulated program can be mapped to real independent and-parallelism in the simulator. Thus, *orsim(sp1)*, which is simulating naïve reverse, a program with no or-parallelism, has very little speedup, whereas *orsim(sp2)*, which is simulating a small version of the highly or-parallel *atlas* program, gave good speedups. As another example, the amount of computation needed to compile the clauses in *compiler* is very heavily dependent on the size of the clause. Parallelism arises from clauses being compiled in parallel, so the best results are achieved with clauses of equal sizes, as in *compiler(cp3)*. When the compiler was run on other programs with greater

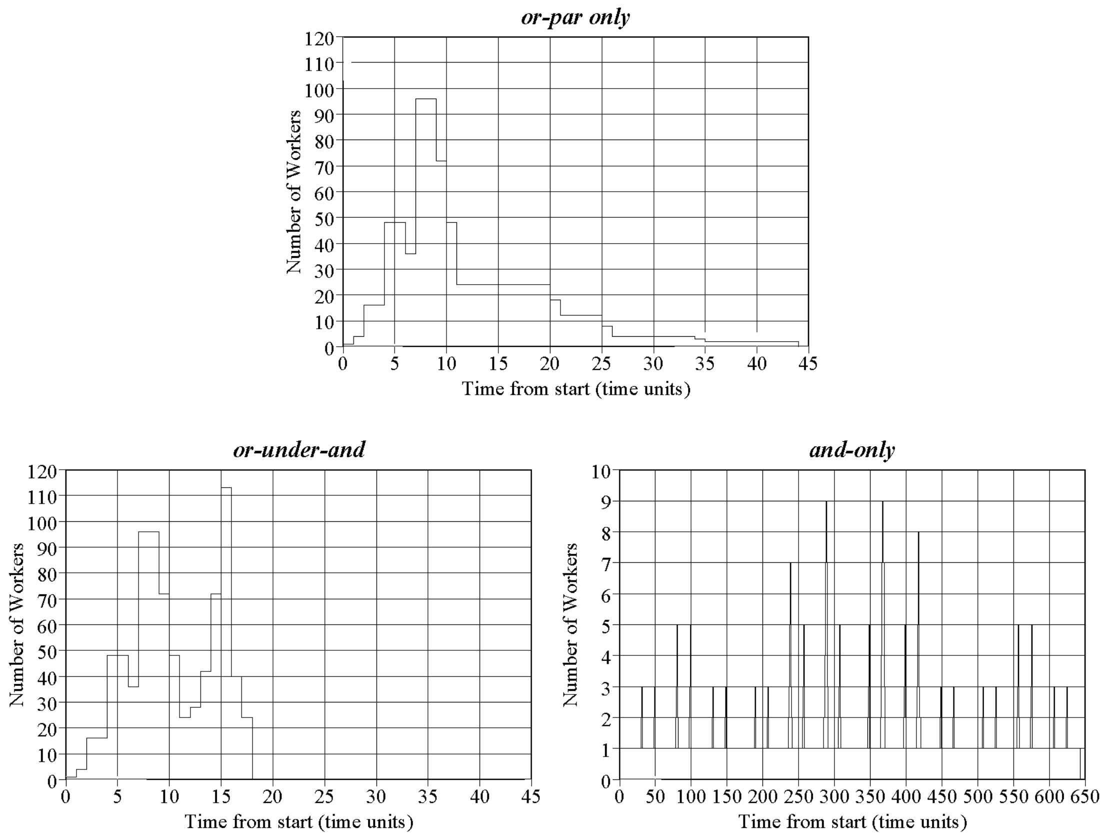


Figure 6: Execution profiles for *4Queens1* under various parallel schemes

differences between clause sizes, the speedup was correspondingly lower: for example, compiling a version of the *atlas* benchmark with a small database took 105465 resolutions, nearly 8 times longer than *compiler(cp3)*, but the maximum speedup (for and-parallelism only) is 2.09× only, versus 7.48× for *compiler(cp3)*.

- It is encouraging that a version of the simulator itself was easily parallelizable. The simulator was originally written as a sequential Prolog application, without any notion of making it parallelizable. Indeed, the simulator originally contained very little parallelism, and automatic annotation was not able to extract much parallelism. However, the simulator that simulated or-parallelism was easily parallelizable by very slight modifications of the program, resulting in a program with significant amounts of independent and-parallelism. The and/or version of the simulator, however, is not so easily parallelizable. We were familiar with the simulator, which allowed us to see where it could be parallelised. Thus, it seems that some programs, which may initially have little parallelism, can be parallelised easily. It is still an open question as to how common this is.

  The overhead in parallelizing the or-simulator is low – it contains just over 1% more unifications than the original (33775 resolutions for the original or-parallel simulator, versus 34117 for *orsim(sp2)*, simulating the same program), and part of this cost is due to the way independent and-parallelism has to be expressed in the system we used, and this should be avoidable.

6 Comparisons with real systems

Here we present the results from comparing the simulator with &-Prolog only, as some comparison of data from the simulator with an or-parallel system has already been published in Szeredi's study of Aurora [20, table 2].

#	boyer_nsi(2)			orsim(sp1)			orsim(sp2)		
	Act.	Pre.(0)	Pre.(8)	Act.	Pre.(0)	Pre.(8)	Act.	Pre.(0)	Pre.(8)
2	1.88×	1.97×	1.94×	1.09×	1.14×	1.14×	1.82×	1.87×	1.87×
3	2.71×	2.91×	2.77×	1.09×	1.14×	1.14×	2.60×	2.69×	2.67×
4	3.40×	3.76×	3.49×	1.08×	1.14×	1.14×	3.11×	3.38×	3.33×
5	4.50×	4.54×	3.94×	1.08×	1.14×	1.14×	3.84×	3.76×	3.72×

Figure 7: Comparison of actual and predicted speedup for a Sequent Balance

#	boyer_nsi(2)			orsim(sp1)			orsim(sp2)		
	Act.	Pre.(0)	Pre.(8)	Act.	Pre.(0)	Pre.(8)	Act.	Pre.(0)	Pre.(8)
3	1.73×	2.91×	2.77×	1.03×	1.14×	1.14×	2.43×	2.69×	2.67×
5	3.19×	4.54×	3.94×	1.04×	1.14×	1.14×	2.60×	3.76×	3.72×
7	3.78×	5.91×	4.88×	1.01×	1.14×	1.14×	3.84×	5.36×	5.16×

Figure 8: Comparison of actual and predicted speedup for a Sequent Symmetry

Figures 7 and 8 show the comparison of results from the simulator with those of &-Prolog running on a Sequent Balance and a Sequent Symmetry respectively. The benchmarks selected for comparison have relatively low maximum ideal speedups (between 1.14 and 12.77), as this would allow divergences from linear speedups to show up clearly with the limited number of processors used in the comparison. The columns in the table have the following meaning:

# Number of agents

Act. The actual speedup of &-Prolog over the execution time on 1 agent. The fastest of 5 timing runs of the program is used to compute this speedup. The fastest time instead of the average time is chosen because we are interested in comparing what &-Prolog is capable of with the ideal speedup.

Pre.(0) Speedup predicted by simulator, assuming 0 units of overhead.

Pre.(8) Speedup predicted by simulator, assuming 8 units of overhead per task (4 each at start and end of task).

The agreement between the simulator's result and actual speedups on a Balance is excellent. The speedup for a Symmetry is consistently worse than that for a Balance. We believe that the main reason for this is that the &-Prolog system was originally developed on a Balance and has not been specially converted to run on a Symmetry.[6]

The relatively poor agreement between the results of the Symmetry and the simulator should be seen in perspective: Using the criterion to classify the benchmarks in the study of Aurora [20], then all the programs used in the comparison fall in "Group L", the "low speedup" group of benchmarks. The agreement of Aurora with their Group L benchmarks is similar to the agreement of &-Prolog on the Symmetry (the actual speed is between 79-91% of the predicted speed (with overhead) for Aurora, and between 84-91% for &-Prolog). Considering that Aurora is a more mature and tuned system than &-Prolog, we believe it is reasonable to expect &-Prolog to be capable of extracting more of the inherent parallelism from programs (*i.e.* how close to

[6] &-Prolog does achieve quite good speedups on the Symmetry for problems that have more inherent parallelism than those used for this comparison [11].

the "ideal" figures of the simulator) than Aurora. We feel that a major reason for this is that the scheduler for available work, a major source of overhead for Aurora, can be kept extremely simple in &-Prolog.

7 Conclusion

We have studied the nature of or- and independent and-parallelism in Prolog programs. We find that not many programs contain both forms of parallelism. Rather, programs tend to exhibit one form of parallelism or the other. Thus, a system which exploits both forms of parallelism can be expected to provide speedups for a quite greater range of programs than one which exploits either form of parallelism on its own. We believe that the "or-under-and" method of combining the two forms of parallelism is a good solution to the problems of combining the parallelism. We are actively researching incorporating this method into a real implementation. From our examples, and extrapolating the results we have for running realistic programs on small example data to larger data, it seems reasonable to expect 10 to 100 fold speedups for realistic programs running on realistic data.

However, with both forms of parallelism, there are still some programs that cannot be speeded up. Notably, programs with dependent and-parallelism only, which is not considered in this study. We are also examining ways to exploit dependent and-parallelism within the framework of Prolog.

The simulator has provided us with valuable information on the nature of both independent and- and or-parallelism, and it has allowed us to better understand the results from actual implementations such as &-Prolog. For example, it allowed us to sensibly compare the results obtained from &-Prolog to those from Aurora, running different benchmarks. We expect that the simulator's results can also be applied to better understand other implementations.

References

[1] H. Alshawi and D. B. Moran. The Delphi Model and Some Preliminary Experiments. In *Proc. Fifth ICLP/SLP*, 1988.

[2] U. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.-C. Syre, and H. Westphal. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Proc. FGCS'88*, 1988.

[3] P. Biswas, S. Su, and D. Yun. A Scalable Abstract Machine Model to Support Limited-OR Restricted AND parallelism in Logic Programs. In *Proc. Fifth ICLP/SLP*, 1988.

[4] A. Ciepielewski, S. Haridi, and B. Hausman. Initial Evaluation of a Virtual Machine for Or-Parallel Execution of Logic Programs. In *IFIP-TC10 Working Conference on Fifth Generation Computer Architecture*, 1985.

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

[6] D. DeGroot. Restricted AND-Parallelism. In *Proc. FGCS'84*, 1984.

[7] A. Delcher and S. Kasif. Some Results on the Complexity of Exploiting Dependency in Parallel Logic Programs. *J. of Logic Programming*, 6(3), 1989.

[8] B. S. Fagin. *A Parallel Execution Model for Prolog.* PhD thesis, U. of California at Berkeley, Nov. 1987.

[9] G. Gupta and B. Jayaraman. Combined And-Or Parallelism on Shared Memory Multiprocessors. In *Proc. NACLP*, 1989.

[10] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, U. of Texas at Austin, August 1986.

[11] M. V. Hermenegildo and K. J. Green. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proc. Seventh ICLP*, 1990.

[12] M. V. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *Proc. Seventh ICLP*, 1990.

[13] L. Hirschman, W. C. Hopkins, and R. C. Smith. Or-Parallel Speed-Up in Natural Language Processing: A Case Study. In *Proc. Fifth ICLP/SLP*, 1988.

[14] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Proc. Fifth ICLP*, 1988.

[15] K. Muthukumar and M. V. Hermenegildo. The CDG, UDG, and MEL methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Proc. Seventh ICLP*, 1990.

[16] B. Ramkumar and L. V. Kalé. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *Proc. NACLP*, 1989.

[17] K. Shen. An Investigation of the Argonne Model of Or-Parallel Prolog. Master's thesis, U. of Manchester, 1986. TR UMCS-87-1-1.

[18] K. Shen. Studies of And/Or Parallelism in Prolog. PhD thesis in preparation, 1991.

[19] K. Shen and D. H. D. Warren. A Simulation Study of the Argonne Model for Or-Parallel Execution of Prolog. In *Proc. Fourth SLP*, 1987.

[20] P. Szeredi. Performance Analysis of the Aurora Or-Parallel System. In *Proc. NACLP*, 1989.

[21] E. Tick. *Studies In Prolog Architectures.* PhD thesis, Stanford University, Stanford, CA 94305, June 1987.

[22] P. Van Roy. A Prolog Compiler for the PLM. Master's thesis, U. of California at Berkeley, 1984. Technical Report UCB/CSD 84/203.

[23] D. H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog – Abstract Design and Implementation Issues. In *Proc. Fourth SLP*, 1987.

[24] R. Warren, M. V. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Proc. Fifth ICLP/SLP*, 1988.