High-level Characteristics of Or- and Independent And-parallelism in Prolog[‡]

Kish Shen[§]
Department of Computer Science
University of Manchester, U.K.
kish@cs.man.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, their 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 determine the intrinsic amount of parallelism, independently of implementation factors, thus facilitating this separation. We expect this study will be instrumental in better understanding and comparing results from actual implementations, as shown by some examples provided 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 obtained.

Keywords: Logic Programming, Simulation, Or-Parallelism, And-Parallelism, Combining Parallelism

1 Introduction

There has been considerable research interest in the implicit parallel execution of logic programs, resulting in the proposal of many execution models in the literature (for example, [18, 41, 52, 48, 6, 45, 29, 46, 15, 2, 3, 37, 34, 60, 56, 30, 8, 50, 26]). Many of these proposals have been implemented, with some associated results reported, and of these, some now approach sequential Prolog in stability and usability, while also providing good performance improvements.

[‡]This paper is an extended and updated version of [63]. The descriptions in various parts of the paper have been extended; new descriptions, such as the discussion of speculative scheduling have been included; finally, new results are presented: some more programs are studied, and results for Aurora and Muse are provided, along with new results for &-Prolog.

[§]Most of the work reported in this paper was carried out while this author was at: Computer Laboratory, Cambridge University, UK

The main reason for the above mentioned interest is that with implicit parallelism, logic programming can maintain its advantages (in ease of programming, *etc.*), while gaining the performance benefits of parallelism. In this way, the addition of parallelism does not add significantly to the complexity of programming, as logic programming separates the specification of the problem (the "logic") from (at least the lower level) details of the control of execution, which can be relatively transparent to the programmer.

The different proposals for parallel execution of Prolog essentially specify different methods of parallelising the control. They differ in the type of parallelism they exploit, and also to the way the exploitation of such parallelism is implemented. They can lead to very different execution schemes, with varying effectiveness in extracting and exploiting parallelism in programs.

Performance results have been presented to date for a good number of proposals. However, it is generally difficult to interpret such results. Firstly, the studies have understandably tended to concentrate on programs that are reasonably suited to the type of parallelism being exploited. It seems important to have a broader view of the nature and availability of the parallelism across a more representative set of programs. Secondly, and perhaps more importantly, most of the published results reflect the combined effects of at least two factors: 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. In fact, most performance studies have concentrated on studying, analysing, or optimising the low level factors. Comparatively little effort has been devoted to the equally important task of determining the impact of the higher level factors. We believe that the lack of understanding of such factors can easily lead to misleading conclusions when interpreting the results from actual implementations. Thirdly, and finally, most of the results published to date are relatively specific to the various systems proposed, and provide either few comparisons or comparisons with only other very similar systems, which make it difficult to abstract the results away from the particular system under study.

In this paper, we present a high-level simulation study of the amount, characteristics, and inter-relationship of the two most common forms of parallelism exploited in many of the approaches, or- and (independent) and-parallelism, in a wide selection of Prolog programs, from simple benchmarks to medium-sized applications. Prolog is chosen as it is by far the most popular logic programming language, has recently been standardised, and is also the most popular candidate for implicit parallelisation.

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. In addition, the results are potentially applicable across a wide range of approaches. Thus, in addition to allowing better understanding of each approach, the results can be also used to compare different approaches which would otherwise be difficult to compare.

Before discussing our study further, we first present a brief introduction to the forms of parallelism available in a Prolog program, followed by a discussion of related work. A discussion of the model of parallelism simulated in the simulator follows. We then expand on our stated objective of studying the nature of the parallelism, and describe the simulation tools. A description of the experiments performed and results obtained is followed by a discussion of these results. Comparisons of the simulated results with some existing systems are then given, and, finally, future work is discussed. Throughout this discussion we assume some familiarity with Prolog and logic programming — it is beyond the scope of this paper to give a detailed introduction to Prolog except to give the definitions of the terms used in this paper. The readers are referred to textbooks such as [16, 68] for a good introduction to the language.

1.1 Summary of Prolog

The section gives a very brief overview of some of the logic programming related terminology that will be used in this paper. The definitions are not designed to be detailed, formal or precise. Rather, they serve only to introduce the terms and give some idea of how they are used.

In Prolog, the execution of a program can be regarded as a process of finding zero, one or more proofs to a supplied **query** with respect to the program. A Prolog program consists of **predicates**, each of which consists of one or more **clause**. A clause consists of a **head** and a **body**. The head consists of a goal (referred to as a **head goal**), and the body consists of zero or more goals (each of which are referred to as a **body goal**. A **goal** is the basic unit from which Prolog program is composed. A query also consists of one or more goals.

The basic unit of computation in Prolog is the **resolution**, which consists of the **unification** of a query goal with a head goal of a clause from the program, and if the unification is successful, the addition of the body goals of the clause to the query. Unification consists of trying to match the query goal to a head goal, with possible bindings (or **instantiation** of variables in the goal). If the goals match, the unification succeeds, otherwise it fails. In general, zero, one or more clause heads from the program can successfully unify with a query goal. Thus, Prolog execution can be thought of as the exploration of a **search-space** (sometimes referred to as a **search-tree**). The search-space consists of the different paths which can be followed in order to find one or more proofs for the query. Each path consists of a series of unifications. More than one path exist because there can be more than one successful candidate for each unification. If there is no successful candidate to a particular unification, the unification is said to **fail**, and the path leads to no proof. A path leads to a proof when all the query goals have been successfully unified.

In sequential Prolog, the exploration of this search-space proceeds in a depth-first, left-to-right manner: that is, when there is more than one candidate for unification with a goal, the leftmost (textually in the program) is tried first, and, if it is successful, then the body goals of the successful clause are added as the next query goals to try, again in a textually left to right manner. If failure occurs, then the system **backtracks** to the last unification where there are still pending alternative candidates, where the next alternative will be tried.

Prolog is considered to be a declarative language, and has predicate calculus as its mathematical basis. However, in order to make the language practical, some goals can perform actions which are outside of the scope of the calculus: these include goals which perform **side-effects** as part of their executions: e.g. I/O functions and those that reduce (**prune**) the search-space of the program. "Cut" (`!`) is the standard sequential pruning operator in Prolog. It is a side-effecting predicate which removes parts of the search-space relative to the point where the cut appears. Those parts will then not be explored during the execution of the program.

1.2 Parallelism in Prolog

In order to make the paper self-contained, we include a brief discussion of parallelism in Prolog. The reader is referred to [13] for references and a more detailed overview of the field.

Prolog provides many opportunities for parallel execution. The parallelism can be classified in many ways, but all classifications contain at least two forms of parallelism, which are the most generally accepted terms: and-parallelism, and or-parallelism [17]. In their most general definition, they include most proposed ways of exploiting parallelism in logic programming. Many classification schemes recognise other forms of parallelism which are really subclasses of the above two forms. Here, we will use the following definition, and subdivide and-parallelism into two subclasses:

Or-parallelism: Or-parallelism arises if potentially more than one path of finding the proof(s) can be tried at the same time. In sequential Prolog, the different paths are tried sequentially in textual (i.e. left-to-right) order.

And-parallelism: And-parallelism arises if parts of an attempt at a proof (subproofs) can themselves be potentially executed at the same time. And-parallelism can be further classified into two types:

- *independent and-parallelism*: only subproofs which are known to be "independent" from (i.e. their executions are not affected by) all other subproofs that are in the process of being executed are allowed to be executed in parallel.
- *dependent and-parallelism*: subproofs are allowed to execute in parallel even when it cannot be known in advance that they will not affect each other. Note that this does *not* mean that they will affect each other: for example, they may turn out to be independent. In fact, it is possible to devise schemes which ensure that even those subproofs

that are really dependent will not affect each other (for example, [56, 62]). The important difference with respect to independent and-parallelism is that the execution of such subproofs *can* overlap in time. Thus, from this point of view, independent and-parallelism can be seen as a special case of dependent and-parallelism.

In this study, we analysed what are currently the two most well-established and frequently exploited forms of parallelism: or-parallelism, and independent and-parallelism at the goal level (i.e. where the sub-proofs being executed in parallel correspond to whole Prolog goals). Goals are considered to be independent if they do not share unbound variables, because instantiation of unbound shared variables is the only way one goal can affect another. In addition, we also consider "non-strict" independent and-parallelism [35, 76, 36], where some goals which share variables are also considered independent because they meet some conditions which ensure that they do not affect each other's execution.

Other forms of and-parallelism are possible, but most are either very dependent on the approach being taken and thus difficult to generalise (e.g. dependent and-parallelism), (interesting) sub-cases of the previously mentioned forms of and-parallelism (e.g. data-parallelism [8, 50, 32]), or they exploit parallelism at a lower level (e.g. unification parallelism, pipeline parallelism), which in general can only give a relatively limited speedup to programs [67, 5].

Although Prolog provides many opportunities for parallelism, actual exploitation of such parallelism presents many practical problems, including the problem of smoothly integrating the exploitation of the different forms of parallelism. Hence there are many different proposed approaches.

In our study, we wish to define some high-level model which many schemes can be abstracted to, to allow the simulation results to be as generally applicable as possible. 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 alternative paths in parallel. 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 [22]. Moreover, there is an even more open issue as to how or-parallelism and and-parallelism are to be combined. These issues will be discussed in Section 3.

2 Related Work

There are relatively few high-level studies of parallel Prolog. The following are known to us: [57, 14, 64, 39, 57]. Of these, only Sehr and Kalé [57] studied both or- and independent and-parallelism. However, the results they presented provided less information than our study, as it only estimated the maximum ideal speedup (which they call the ‘critical path times’), without the number of processors needed to achieve this. This information is provided by our system, but in addition we provide the variation of speedups with number of processors, which is more important in gauging the performance of real systems, where resources are always limited. All the other high-level studies were of or-parallelism only, and in general studied a smaller set of programs than our study.

Some lower level studies of specific parallel Prolog systems can also provide some insights into the more general higher level issues. The one that is perhaps most closely related to our work is the study by Fagin and Despain [23] of their or- and independent and-parallel Prolog model, PPP. This work provided one of the earliest studies of the properties of combining the two forms of parallelism. However, this study was quite specific to the PPP model, in which or-parallelism is quite severely limited under and-parallelism, and thus the study is not very applicable to more recent and less restrictive schemes for combining the parallelism. Furthermore, we feel that it is important to examine a greater number of more realistic programs than was considered in this study. More recently, one of us [24] approached the problem of obtaining ideal speedups by obtaining the timing information from actual executions (&-Prolog running sequentially, although other Prologs can be used), and then using this timing information to obtain speedups. Both or- and independent and-parallelism were studied separately, although not combining the two as in this study. This approach (called IDRA) can produce quite accurate predictions of speedups for many programs, but to some extent this accuracy is specific to using the same implementation for which the speedups are being predicted when obtaining the timings. In addition, its objectives are much more restrictive: it was not designed as a general study of characteristics of parallelism.

3 Model of parallelism simulated

We employ interchangeably the widely used terms of **worker** and **agent** to refer to the entities that perform the computation, or **work**. Parallelism is achieved by allowing several workers/agents to simultaneously explore the search-space of a program. Each worker explores the search-space 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-space, and thus the search-space can be thought of as being divided into “chunks” of sub-tree, with each sub-tree being executed sequentially. 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 which are not working can come and “steal” part of the sub-tree by splitting it. Conceptually, the search-space can be thought of as being annotated with **sources** of parallelism, which generate

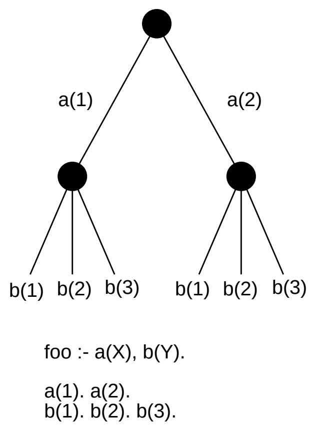


Figure 1: Example execution

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, allowing the exploration of more than one possible proof to the goal. Available and-nodes allow sibling-goals within a clause (**and-goals**) to be run in and-parallel with each other, cooperating to find a particular proof.

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 this study we selected the restricted and-parallelism (RAP) rule, first proposed by DeGroot [21] and refined (by proposing backward semantics and improved graph expressions for controlling parallelism) by one of us [37]. 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 ready availability of an automatic annotator for this type of parallelism [75, 51, 11] and of an actual implementation (&-Prolog) with which to contrast the results of the simulations.

3.1 Avoiding re-computation

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. Consider the example execution in Figure 1: the program and the search-space explored by a sequential execution is shown. As the goals `a(X)` and `b(Y)` are independent, they can be executed in and-parallel. Note that sequentially, `b(Y)` is executed twice, one for each branch of `a(X)`. The execution of `b(Y)` is independent of that of `a(X)` and therefore the two executions are identical. Thus, if `b(Y)` is computed once only, and then "reused" in the two branches of `a(X)`, then the amount of computation performed overall can be reduced. The amount of computation saved in such a scheme can (in theory) be very significant. Reusing such computations is possible even for sequential systems (e.g. through use of some form of memo function [49] and, in particular in Prolog, using all solution predicates), but this would mean keeping the state of computation of the goal to be reused around, thus consuming memory and, in addition, the condition needed for reuse — independence of the goals — also needs to be detected. When exploiting independent and-parallelism, the independence is detected already as part of the parallelisation process, although memory would still need to be set aside for preserving the state of goals that might be reused. In the case of a system that combines or- and and-parallelism, there is in principle no memory disadvantage with preserving the state of the goals, and there is the possibility that the amount of computation might be reduced drastically. Thus, reuse of computation may seem particularly attractive in systems that combine both and- and or-parallelism, and most early proposals for combining the two forms of parallelism include reuse of computation (e.g. [42, 6, 29]). These systems differ in how and- and or-parallelism are exploited and combined, but for the reusage of computation, essentially each and-goal is computed once (in the example of Figure 1, `a(X)` and `b(Y)` are each computed once) in and-parallel, and their results combined to form all the possible combinations of the solutions that are computed in a sequential system (6 solutions in the example of Figure 1). However, one drawback of such a method for combining independent and-parallelism and or-parallelism is that it is complicated both for an actual implementation and a simulation to actually combine the various solutions from the and-goals, and the process of combining the solutions would add overheads to any implementation that supports it. In addition, it is more difficult to support full Prolog because it is harder to handle side-effects correctly with respect to sequential Prolog: the reusage of a goal must ensure that any side-effects of the goal be performed each time the goal is reused. This again increases the complexity. Therefore, it appears interesting to study how much search-space reduction goal reusage would obtain in programs.

3.2 Combining the parallelism

In addition to the implementation complexity of reusing and-goals, the *unrestricted* combination of independent and- and or-parallelism adds both to the conceptual and implementation complexity. An alternative is to restrict parallelism in some way when both types of parallelism are combined. This leads to simpler schemes, but obviously at the expense of some parallelism. Examples of such approaches are those of Conery [17], Fagin and Despain [23], and Biswas *et al.* [9]. 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 with goal reusage”, or simply “no or-under-and”, or-parallelism is not allowed within and-parallelism. Reusage of goals as described in the previous section is allowed, which can reduce the amount of computation performed. The restriction on parallelism makes it much simpler to implement goal reusage, at the expense of parallelism. In the second scheme, which we called "or-under-and with no goal reusage", or simply "or-under-and", or-parallelism is allowed within and-parallelism, but with no reusage of goals — goals on separate paths are computed separated as in sequential Prolog.

no or-under-and, reusage	or-under-and, no reusage
‖ a(1) b(1)	‖ a(1) a(2) b(1) b(2) b(3)
b(2)	‖ b(1) b(2) b(3)
b(3)	
a(2) (b(1) b(2) b(3) reused)	

Figure 2: Parallelism in the two schemes

To illustrate the two methods of combining the parallelism, consider the example program of Figure 1 again. Figure 2 shows how the parallelism would be arranged for the two methods. For "no or-under-and with goal reusage", as `a(X)` and `b(Y)` are to be executed in and-parallel, they cannot be executed in or-parallel because of the restriction on combining parallelism. Only one solution (the leftmost: `a(1)` and `b(1)`) is initially executed in parallel (as shown in the first row of the left side of Figure 2). Backtracking is then used to produce the other solutions of `b(Y)`, one at a time: thus, first, the second alternative for `b(Y)`, `b(2)`, is found, producing the solution: `a(1), b(2)`; followed by the next solution `a(1), b(3)` in the same way.

As this scheme also has goal reusage, then as each solution for `b(Y)` is produced, it is also stored for later reuse, so when all three alternatives of `b(Y)` have been produced, and the system backtracks again, this time to execute the second alternative of `a(X)` (`a(2)`). Once `a(2)` is produced, then as the solutions for `b(Y)` are already available through reusage, the three remaining solutions (`a(2), b(1)`; `a(2), b(2)`; `a(2), b(3)`) are available without having to compute `b(Y)` again.

In the case of "or-under-and with no goal reusage", `a(X)` and `b(Y)` can be run in and-parallel together and, in addition, each goal can run in or-parallel. Thus, the alternatives `a(1)` and `a(2)` would be generated in parallel. With no goal reusage, the alternatives for `b(Y)` can only be used for one of the alternatives of `a(1)`: following Prolog, they are combined with the leftmost alternative, `a(1)`. Alternatives of `b(Y)` for `a(2)` have to be computed separately. The second set of alternatives for `b(Y)` are computed in or-parallel as soon as `a(2)` succeeds. The reason this is not done earlier is because, in general, it cannot be known in advance how many alternatives there would be for the left goal (`a(X)` in this case)[1], and thus how many sets of the right goal (`b(Y)` in this case) have to be computed separately.

It is important to point out that the issues of a) re-computation and b) of restriction in the way the two forms of parallelism can be combined, are orthogonal. This allows us to infer conclusions about other models implementing other combinations from only the two simulation schemes proposed. For example, since the goals which are reused do not change because of

[1] Although it is clear that there are two alternatives which would succeed here, in real programs the situation would generally be less clear.

changes in the nesting restrictions, it is possible to see how useful reusage would be in general, from data obtained with one of the parallelism nesting schemes.

4 Issues for investigation

The following issues were investigated in the study:

- Which (if any) programs are suitable for the two types of parallelism simulated? What types of programs are suitable for each of the two schemes of combining the types of parallelism?
- Do the two types of parallelism generate tasks at different places in the search-tree and are the sizes of tasks different? That is, is the nature of the tasks generated different?
- How does the speedup of various types of parallelism and ways of combining them vary with the number of workers?
- How do overheads affect the speedups for the various types of parallelism?
- How do the two methods of combining and- and or-parallelism compare? Are they effective ways of combining and- and or-parallelism? How much work is saved by reusing goals? How do and- and or-parallelism interact?

In addition, although less central to the objective of the paper, we would also like to estimate the overhead involved in evaluating the tests in the CGEs. Finally, while, as mentioned in the introduction, the main aim of the paper is to study issues related to the characteristics of parallelism in Prolog programs in the abstract, rather than in the context of specific systems, it is also our aim to illustrate how the simulation results and the simulator itself can be used as a tool for studying specific parallel systems. To this end, we also explore the following issue:

- Are the results obtained from the simulator meaningful for real systems? Can we use the simulated results to aid in the analysis of real systems?

5 The simulator

The simulator is a greatly modified version of the or-parallel simulator described in [58], with support for independent and-parallelism (in the form of RAP) added. 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.

The simulator is written in Prolog and divided into two parts: a “static” simulator (basically a Prolog engine which generates a graph representing the search-tree explored by the Prolog program being simulated) and a “dynamic” simulator (which simulates the processing of this tree by a number of workers).

5.1 Assumptions

In order to make the results applicable to as wide a range of models and implementations as possible, as few assumptions as possible were made about the underlying hardware and execution model. The speedups can thus be regarded in some way as the "ideal speedup" attainable with the program given the annotation. Many actual schemes can be idealised directly to the simulator's model of or-parallelism, e.g. [53, 72, 74, 1, 6, 23]. Furthermore, results are also meaningful for schemes that are not based on the same conceptual model used in the simulator. With respect to the various schemes related to Conery's AND/OR process model (for example, [17, 77, 41, 10]), our model can be thought of as the case where there is no cost in creating a new or-process (however, note that Conery's model restricts or-parallelism in some other ways). As another example, the simulated model can be regarded as an idealised realisation of Clocksin and Alshawi's Delphi model [15], where the oracles have perfect knowledge of the search-tree, and can send workers to the right paths perfectly.[2] Similarly, the simulated model can be regarded as an idealised version of the randomised parallel backtracking method [40], where the "correct" alternatives are always selected. Also, as the task switching can be cost-less, the results are meaningful even for Lin's self-organising task scheduler [46], which transforms the program to obtain a better size distribution for the tasks. This approach does not increase the inherent parallelism as its purpose is to make task switching less costly. The major assumptions made in the simulator are:

- The basic time interval used is one unification, i.e. all (successful) unifications take the same amount of work to perform and hence take the same amount of time to execute. If a goal fails totally (i.e. no match at all could be found), it is also assumed to take the same amount of time as one unification.

  It seems reasonable to assume that unifications take roughly the same amount of time to perform as in a sequential system. Indeed, a "logical inference" is basically a unification, and is commonly used to measure the performance of a Prolog system. There are variations on the work done (and thus the time taken to do it) per unification, but to a first approximation, the *average* time taken for each unification can be assumed to be constant over the execution of the program. This approximation is all that is needed for an abstract, high level simulator. The extra work and overhead required for a parallel Prolog system may have some effect on the validity of this assumption. However, in an idealised situation, the overhead of a real system can be ignored, and the extra work needed assumed to be zero. Certainly it has been shown (for both or-parallelism and IAP) that the overhead can be kept low [69, 38], so the assumption that all unifications are equal should still be a reasonable approximation.

[2]This describes the case where resources (workers) are unlimited. With limited resources, the results can still be regarded as an idealised version of Delphi where all re-computation besides the initial computation needed to get a worker to a unexplored node are cost-less.

- Perfect indexing for clauses is assumed. That is, only unifications which will succeed are tried. Therefore, unification failures only occur when there are no candidates for unification at all.

  No real Prolog system can have perfect indexing, though some form of indexing does exist in most Prolog systems to filter out some of the failures. The ideal case is perfect indexing, where all such failures are filtered out. Again, the ideal case is assumed.

- When the simulator encounters a parallel node, all the work (or-parallel alternatives or sibling and-goals) is made available at that time unit, and can be picked up by other workers at the same time.

  In a real system, parallel work cannot be made available to other workers immediately, as it takes some time to spawn parallel work. Furthermore, the parallel work available at a node probably cannot all be taken at the same time. For example, for or-parallelism, the alternative clauses probably have to be selected one at a time. This is difficult to model, and again the simplifying assumption is to idealise the situation.

- Overheads are modelled by allocating a fixed amount of delay to the start and termination of a task. This can be regarded as overhead for task-switching itself and also overhead within a task. The amount of delay can be varied between simulations.

  This way of modelling overheads has the advantage of being simple. It assumes a constant overhead for each task, which is probably reasonably accurate for some overheads in real systems: some would indeed be constant and some would average out to be constant, although it can vary from task to task. However, it is also possible to have overheads which have some sort of dependencies on the size of the task, and such overheads are not modelled accurately. However, the main aim is again not to model any individual system accurately, but to provide some more general and high-level information. The simple modelling is sufficient to show how sensitive a particular program's speedups (under some form of parallelism) might be to perturbation by overheads: the greater the perturbation (i.e. where a small overhead have a large effect on the simulated speedups), the less likely the indicated speedup will be achieved in a real system: this shall be demonstrated clearly by the comparison with real systems in Section 7.

  Note that some systems would be expected to have higher overheads: e.g. systems which run on distributed machines. This does not affect the validity of the speedups produced by the simulator; the ideal case is still the figures with 0 overheads, but the realistic speedups would be expected to be better modelled using higher overheads.

- "Cut" is dealt with as in sequential Prolog, so or-branches in the search tree to the right of the "cut" are not tried. This behaviour is chosen as, firstly, this allows programs that would generate an infinite search-tree without the "cut" to be simulated. Secondly, it seems that this new behaviour is closer to the “ideal” situation where a system could predict which branches would be cut away and therefore not try them. In practice, it is impossible to make such predictions in all cases, and thus some work to the right of a cut would be performed in real systems, but the amount of work can be reduced by scheduling, up to the idealised limit of not performing any such work at all.

- Backtracking in a CGE behaves essentially like Prolog, so no intelligent backtracking is done. If an and-goal fails, and-goals to the right of it are not executed, whereas those to its left are tried fully. If the and-goal immediately to the left of the failed and-goal has not completed yet, then backtracking takes place when it has finished. Backtracking in a real and-parallel system (with and without or-parallelism) is a complex issue, with many possible schemes that would give correct behaviour with varying degrees of sophistication and complexity of implementation. We avoid all this complexity in our simulation.

  The search-space explored by the simulator is therefore exactly that explored by a sequential Prolog system, and not what an independent and-parallel system might explore: in a real and-parallel system, the actual search-space explored would be the same as the sequential one in the absence of failure, but with failure, the exact search-space explored is non-determinate, with the sequential search-space being one possibility. For the simulator, it is very difficult to deal with failures within a CGE in a different way from the sequential case due to the method of simulation. It is also unclear that assuming the search-space to be something other than the sequential case would bring any more accuracy, due to the many possible schemes of backtracking in actual systems. Therefore, again, the simplest case is assumed in the simulator, which, once more, corresponds to the ideal case.

- Backtracking takes no time.

  In a parallel system, especially an and-parallel system, backtracking may be more expensive than in a sequential system for various implementational reasons. In addition, the time taken for backtracking is very highly implementation dependent and is therefore unsuitable for modelling by a high-level simulator. Again, the ideal situation is that backtracking takes no time.

- And- and or-parallelism are combined in one of two ways. Only one way can be used for a particular simulation. These two ways are “no or-under-and” and “or-under-and” as described in more detail in Section 3.2.

- The CGE tests for groundness and independence were assumed to be handled by the underlying implementation, instead of being done as Prolog source level code since this reduces the cost of these tests considerably. However, assuming that the tests are handled at a low level makes modelling their cost on our simulator more difficult. In a compiled system, the tests would each be compiled into one abstract machine instruction of the Prolog engine. A study by Tick of the sequential WAM (the most commonly used Prolog abstract machine) suggests that about 15 WAM abstract machine instructions are executed per procedure invocation (see table 3-3 of [71]). Successive procedure calls correspond to a unification. Thus, this implies that on average, an abstract machine instruction is $1/15$ the cost of a unification, the basic time-interval used in our simulator. Naturally, the actual cost of each abstract machine instruction does vary, especially if it needs to handle potentially large structures such as performing the CGE tests on large structures, where the whole structure needs to be traversed, where the cost would depend on the size of the structure. We would thus like to model this variation in the cost as well. We thus 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 herein, the tests exhaustively traverse the terms.

At this approximate level, it is reasonable to model the extra cost at each recursion level as the cost of an extra abstract machine instruction. In the majority of cases, an abstract machine instruction does not operate on large structures, and thus the cost of $1/15$ reflects more the cost of operating on relatively small structures. Of course, this modelling is probably not very accurate, but nevertheless it should be useful in giving us some idea of how expensive or otherwise CGE tests are.

- At the end of a task, a worker is free to switch task to any available node. If the worker was an and-task, 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.

  A real system may (or may not) have restrictions on which goals can be taken by an idle worker, and task selection and switching may be relatively expensive. Again, this is idealised by assuming it to be cost-less.

- With everything else being equal, available and-nodes are favoured over available or-nodes in selecting an available node.

  There is no strong reason for doing this. Some form of scheduling must be assumed, and it seems that and-goals may have some advantage over or-goals because and-parallelism may have some advantage in memory usage over or-parallelism. This has no effect with the simulator, and given the many idealised assumptions, the impact of a particular scheduler on the actual speedups is not high in any case.

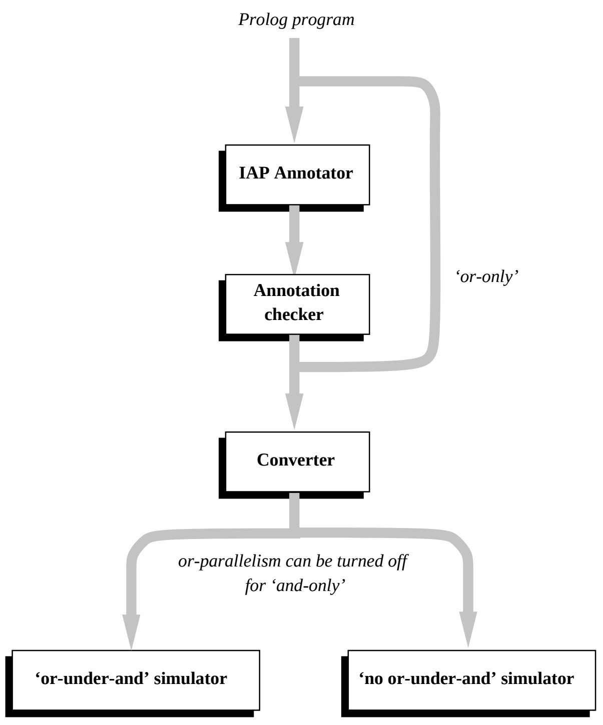


Figure 3: Result Generation Process

5.2 Generation of results

As shown in Figure 3, 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 is the "mel" annotator introduced in [75] and described in [51]. 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] This annotated program was then converted to the format used by the simulator and simulated.

6 Summary and discussion of results

6.1 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 and can thus be easily analysed: in most cases, the annotations which would lead to maximum theoretical independent and-parallelism are known, and the programs are

[3]This step was quite instrumental in flagging a number of bugs in early implementations of the "mel" annotator. It became unnecessary at the later stages of this study as the annotators matured.

thus annotated. The second category includes existing applications, running relatively simple or small input to get the execution time to a level suitable for simulation. These programs are more representative of realistic programs. The advantage of such application programs is that they allow us to study more realistic examples, but, at the same time, we cannot be as certain that the annotations used were the optimal ones for and-parallel execution.

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.

Brief descriptions of the programs are given in appendix A.

6.2 Overall 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 performed assuming no overhead. The simulation was then also done assuming 8 units of overhead per task (4 units each at the start and finish of a task). This amount of overhead was used simply to give us some idea about the stability of the predicted speedups: a constant amount of overheads for all the programs has to be used in order to allow the effect on different programs to be compared against one another: 8 units of overhead was chosen because it was not so small that its effect would not show up in most cases, and not so large that it would come to dominate the behaviour.

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 4, 5 and 6. The first table records the "static" data obtained from the static part of the simulator, and the other two record the "dynamic" data. To reduce the amount of data presented, Figures 5 and 6 only show the results for one example when results are available for execution of the program with different data. Tables 7 and 8 present the results for changing the data the program is being run with. 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.

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

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

uncon. CGE 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 figures are marked by a "*".

6.3 Discussion of the tables

The results 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. The exact amount of parallelism of course often depends on the size of

[4]Note that "ratio" was called "over." in [64]. We feel that "ratio" is a more accurate name.

name	Σ res.	sol.	par. CGE	seq. CGE	uncon. CGE	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%)
cluster	35370	1	100	0	100	0 (0%)	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(cp1)	105465	1	23	0	23	0 (0%)	0 (0%)
compiler(cp2)	193494	1	10	0	10	0 (0%)	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%)
floorplan	43296	4	787	88	785	90 (0.208%)	0 (0%)

Figure 4: Summary of Static data from simulations

the input data – the influence of this important issue will be studied in the following section. Both types of parallelism have obvious areas of application where significant speedups can be achieved. Or-parallelism is more abundant in programs that require substantial searching, such as the *warplan* programs. Independent and-parallelism is more abundant in algorithms which follow the pattern of "divide and conquer" algorithms – such as *compiler(cp3)* and *annotator*. Overall, independent and-parallelism and or-parallelism gave essentially comparable speedups over the range of programs studied. Not many programs that contain significant amounts of both independent and- and or-parallelism were found. *cluster* and *compiler(cp3)* were the only programs which approached having significant amounts of both types of parallelism. In addition, the or-parallelism in *cluster* is very fine grained as all the tasks are very small.[5] Nevertheless, it

[5]In fact, in systems such as Aurora and Muse, there is no or-parallelism for this program due to some low-level optimisations of the Prolog system: the low granularity or-parallelism is due to quick failures on some branches after performing simple "guard" type tests. The

name	And only			Or only		
	max. perf.	half perf.	ratio	max. perf.	half perf.	ratio
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	80.0%*
hanoi	52.3×@427	26.1×@53	61.5%	1×@1	1×@1	100%
cluster	32.04×@54	15.9×@20	90.5%	3.16×@4	1.93×@2	48.6%
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_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(cq3)	1.03×@3	0.96×@1	94.1%*	2.18×@27	1.49×@2	84.2%
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%
floorplan	1.02×@2	1.00×@1	93.3%*	41.08×@233	20.5×@26	91.5%

Figure 5: Summary of dynamic data for and- & or- parallelism from simulations

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". The gain of reusing goals by "no or-under-and" is insufficient to compensate for the loss of parallelism. This shows that banning or-parallelism inside and-parallelism may be too drastic a restriction, although this depends of course on the price in overhead incurred by allowing such 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 re-compute the reused goals. Note that some of this loss can be regained by the extra amount of parallelism, and that in any case no extra cost is involved with respect to the sequential computation, which also performs such re-computation. In addition, as mentioned before, the re-computation can be avoided by programming the formation of the cross product explicitly

optimisations were able to avoid such tests altogether.

name	Or-under-and			No or-under-and		
	max. perf.	half perf.	ratio	max. perf.	half perf.	ratio
	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	56.5×@∼475	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%
cluster	55.62×@326	27.87×@41	57.0%	32.04×@54	15.9×@20	90.5%
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_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(cq3)	2.32×@27	1.51×@2	85.6%	1.93×@10	0.96×@1	81.6%*
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%
floorplan	42.41×@256	21.3×@27	89.2%	41.92×@256	21.2×@27	89.3%

Figure 6: Summary of dynamic data for combined and/or parallelism from simulations

at the Prolog level (an example is given in [61]). This can be made easier for the programmer by providing predicates that perform the cross product of several goals (along the lines of meta-logical predicates such as `set_of/3`). Furthermore, program analysis may be able to transform some programs into a form that tries to avoid re-computation. The important point is that since reusage of goals appears to be not common in general, the cases where it is useful can be dealt with by specialised means instead of providing a general and complex mechanism.

Another point in favour of a re-computation scheme over a reusage scheme is that it is much easer to deal with side-effects. In fact, “or-under-and” has, since the first publication of this study [63], formed the basis for some proposed implementation schemes for combining independent and- and or-parallelism [27].

The speedup obtained from all forms of parallelism simulated is far from linear as the number of workers is increased towards the maximum demand. This is shown by the number of workers required to achieve half the maximum performance being in general much less than half that required to achieve maximum performance. This will be discussed in more detail later.

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%*
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(cp1)	2.09×@3	1×@1	100.0%*	1×@1	1×@1	100%
compiler(cp2)	4.78×@6	2.94×@3	100.0%	1×@1	1×@1	100%
compiler(cp3)	7.48×@15	3.84×@4	98.2%	2.49×@8	1.66×@2	63.3%
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%
orsim(sp1)	1.14×@2	1×@1	99.9%*	1.29×@5	1×@1	87.5%*
orsim(sp2)	8.32×@∼20	4.44×@6	99.5%	1.47×@5	1×@1	87.9%*

Figure 7: Scalability of 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%
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(cp1)	2.09×@3	1×@1	100.0%*	2.09×@3	1×@1	100.0%*
compiler(cp2)	4.78×@6	2.94×@3	100.0%	4.78×@6	2.94×@3	100.0%
compiler(cp3)	16.9×@∼60	8.88×@10	78.3%	7.48×@15	3.84×@4	98.2%
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%*
orsim(sp1)	1.44×@6	1×@1	87.0%*	1.15×@5	1×@1	99.1%*
orsim(sp2)	10.7×@∼43	5.13×@6	94.8%	8.59×@18	4.51×@6	97.9%

Figure 8: Scalability for combined and/or parallelism from simulations

6.4 More detailed look at the results

The summary tables are not sufficient to show some of the observations made during the study. This is partly because these observations depended on details not shown in the summary, and partly because some involved extra experiments. Some of these observations will be presented in this section.

- 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" to some extent 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 quite fine, and sometimes occurs in small "bursts", with intervening sequential areas, thus resulting in a classical instance of the "Amdahl effect".

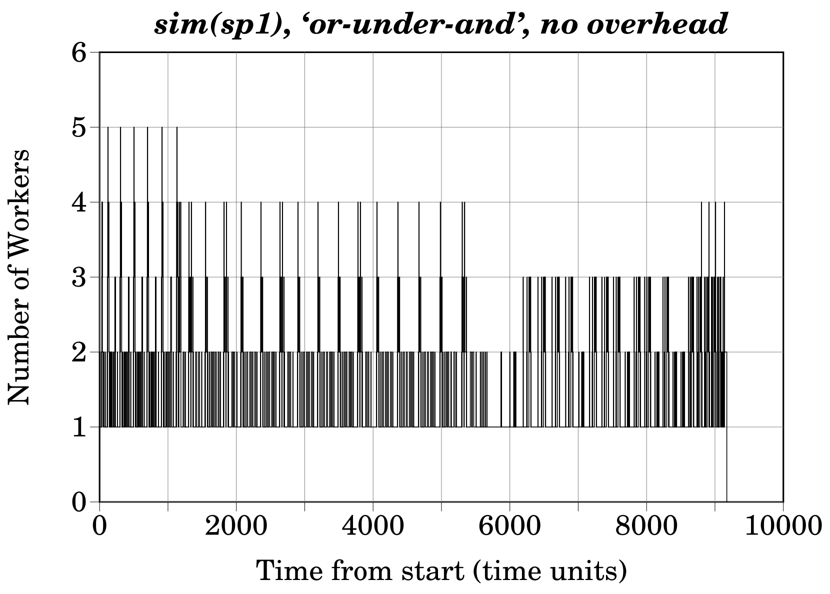


Figure 9: Execution profile for *sim(sp1)*, “or-under-and”, no overhead

Figure 9 shows such an example. It should be possible to improve the performance by trying to reduce the scheduling of tasks with very fine grain. A simple runtime control mechanism, where a new task does not immediately allow available nodes to be taken by other workers until the task has performed some amount of work, thus reducing the number of tasks with very fine granularity, was proposed by one of us [64]. This method was incorporated and tested in the Aurora or-parallel Prolog system. It did result in coarser granularity and improvement in performance for some programs, but also a decrease in performance for other programs, because the mechanism affects all tasks, and thus delays the start of tasks which lead to significant parallelism (because they have large granularity) as well as limiting tasks with small granularity [69]. A better approach may be to perform some form of granularity analysis (e.g. [20, 43, 79, 47, 19]) at compile time to obtain information of the likely granularity of tasks, thus allowing granularity to become more coarse without limiting the parallelism in tasks which are not fine grained.

- The speedups for “or-under-and” show that combining and- and or-parallelism can lead to a significant increase in performance if both types of parallelism are present in the program. Examples of the speedups with the various types of parallelism are shown in Figures 10 and 11. They show 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

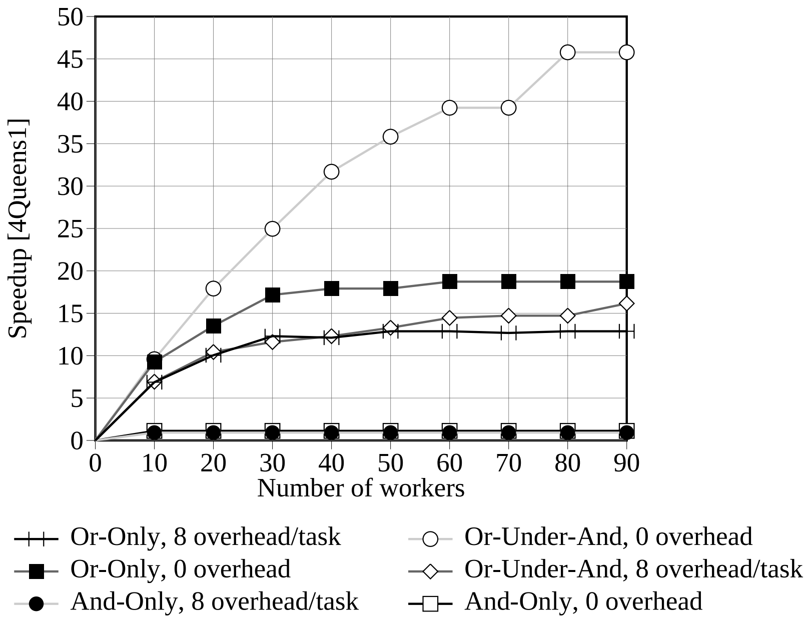


Figure 10: Speedups for *4Queens1*

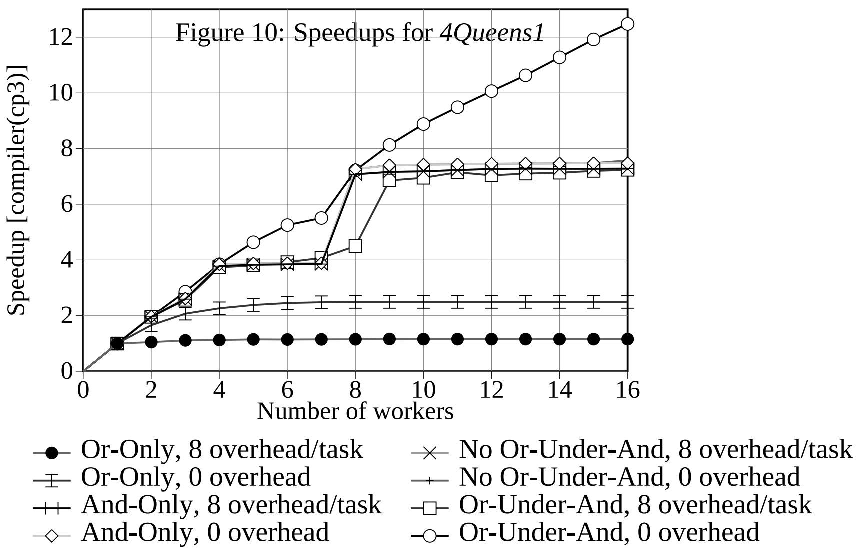


Figure 11: Speedups for *compiler(cp3)*

speedup of about 1.2 only.

Figure 12 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.

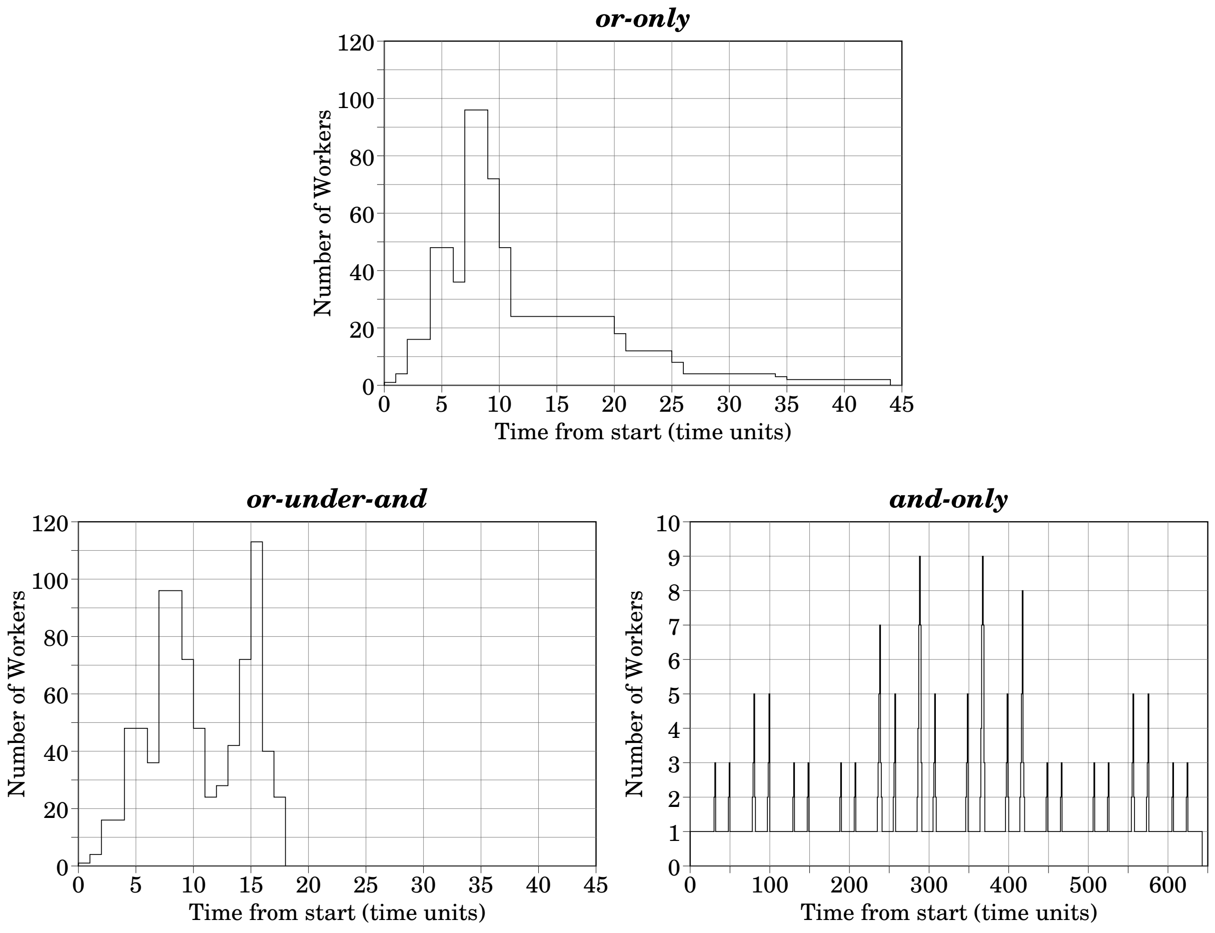
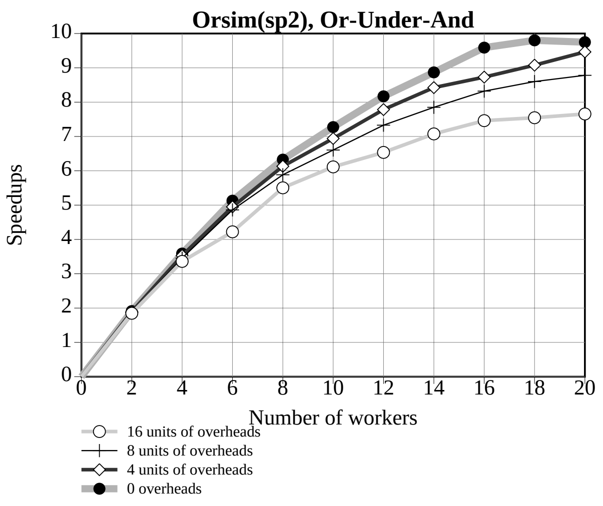


Figure 12: Execution profiles for *4Queens1* under various parallel schemes

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 the and-parallelism in the case of *4Queens1*, is very fine grained and is thus 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 utilising both forms of parallelism. In addition, and perhaps more importantly, it allows us to parallelise the execution of a much wider range of programs than either parallelism alone would.

- As expected, the greater the delay overhead, the slower the performance of a program. The variations are shown in Figure 13 for two programs: *orsim(sp2)* and *tak*. Although *orsim(sp2)* has lower speedups than *tak*, the impact of overheads on speedups is smaller: the reason is that the task sizes are greater in the case of *orsim(sp2)*. Figures 5 and 6 show that for *orsim(sp)*, the effect of overheads is not the same for or- and and-parallelism: the



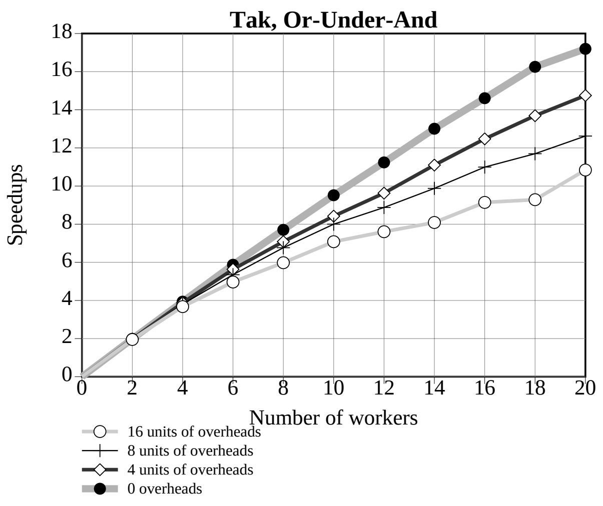


Figure 13: Variations of Speedups with Overheads

impact is greater for the or-tasks than for the and-tasks, implying that the and-tasks are of larger granularity. In fact, most of the slowdown seen in Figure 13 for the program is due to the effect of overheads on the or-tasks. Note also that even with such a simple modelling of the overheads, the expected behaviour of the differences between the speedups increasing with increasing number of workers is observed.

- In many cases in all the four parallel schemes, no further speedups are obtained beyond a critical number of workers (the "demand"), even though more workers continue to be used by the system, up to the maximum parallelism of the program. Thus, the maximum performance is not necessarily reached only at the maximum parallelism exploited by a program: sometimes it is reached much earlier. The reason for this is that in many cases, before the maximum parallelism is reached, there are enough idle workers around to pick up the work that would be taken up by the extra workers, i.e. the parallelism is merely redistributed with extra workers after the "demand" is reached, leading to lower utilisation of individual workers, but no speedups.

- An important question is how much or-parallelism exists in programs where there is only one solution; or in programs which have more than one solution, but where only the first solution is needed. The results for the *warplan* program suggest that or-parallelism can be useful in some cases even if the program has only one solution. However, the results presented do not apply for the case where the return of the first solution is enforced by a cut. A further study was conducted to evaluate such cases. The results are summarised in Figure 14. The programs studied are taken from the benchmark set used by Szeredi [69] for the study of Aurora, which has been used to study several or-parallel Prolog systems.

  The results suggest that there can be substantial amounts of or-parallelism, even if only the first solution is required. It can therefore be profitable to exploit or-parallelism in such

name	Σ res.	max. perf.	half perf.
farmer	136	2.7×@5	1.7×@2
house	1089	20.2×@52	10.4×@13
parse1	455	4.9×@17	2.7×@3
parse2	1114	7.6×@28	3.6×@4
parse3	383	5.0×@20	2.7×@3
parse4	2879	8.4×@74	4.3×@5
parse5	7354	8.7×@74	4.3×@5
db4	4627	112.9×@514	56.4×@80
db5	6630	88.4×@614	43.9×@59
8-queens1	10567	67.7×@291	33.9×@42
8-queens2	25650	102.2×@∼320	50.9×@60

Figure 14: Speedups for First Solution Only

cases.

Furthermore, this has significant implications for scheduling strategies used in existing or-parallel Prolog systems. With many of the early schedulers used in systems such as Aurora and Muse, almost no speedups were obtained for programs which used the cut to force the return of the first solution only. The reason is that these schedulers took no account of the cut, and scheduled workers to work on work that would be later discarded by the cut. More recent schedulers in Aurora and Muse (e.g. [7, 65, 4, 66]) tackle this problem by allowing a worker to suspend the work it is doing and switch to more profitable work if it discovers that the current piece of work is *speculative*, i.e. that it might be discarded. Much better speedups than the older schedulers have been obtained for these programs. However, the speedups are still generally significantly less than those attainable in theory, as given by the simulator, which, as mentioned before, assumes the ideal case where no discarded work is performed: for example, a speedup of 3.25 is reported in [65] for 8-queens1 for 10 workers, compared to 9.59 for 10 workers for the simulator. This disagreement is considerably more than the differences for the same systems in programs which have little or no discarded work, as reported in Section 7. The most likely reason is that much wasteful work is still performed by executing work that will be discarded, and thus there is still much room for possible improvements to these scheduling strategies.

- 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 *boyer_si(2)* running the same data with strict independent and-parallelism. *tp* was also found to contain non-strict independent and-parallelism – what little independent and-parallelism that exists is almost all non-strict. This suggests that this type of parallelism deserves further study and we have started developing compiler technology to exploit it. A technique for detecting this type of parallelism using global analysis information available with well known analysis domains is reported in [12].

- The amount of parallelism obtained can depend greatly on the particular problem being solved, as shown in Figures 7 and 8. In some cases, the amount of parallelism depends on the size of the problem (*annotator* and *cluster* are good examples 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.

  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 (generated using a fairly simple annotator) 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 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) was 2.09$\times$ only, versus 7.48$\times$ for *compiler(cp3).*

- The simulator was originally written as a sequential Prolog application, without any notion of making it parallelisable. 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 parallelisable by very slight modifications of the program, resulting in a program with significant amounts of independent and-parallelism. This suggests that there are programs from which it may be difficult to automatically extract parallelism, but are nevertheless parallelisable with only a little effort in modifying them. Of the application programs that exhibited little initial parallelism, we were most familiar with the two versions of the simulator, and of these, we were able to easily parallelise the or-simulator, but not the and/or simulator. The number of examples is far too small to generalise, but it is encouraging.

  An important point with the parallelised version of the or-simulator is that there is not much overhead in the parallel version. The parallelised version 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 should be avoidable.

7 Comparisons with real systems

At the time of making our comparisons, there were several mature or-parallel systems, and at least one reasonably mature independent and-parallel system which allow comparison with the results from the simulator. Here we present comparisons between the simulator and results from Aurora and Muse, two or-parallel Prolog systems, and &-Prolog, an independent and-parallel Prolog system. Note that this comparison between the systems are not meant to be an exhaustive study of these systems; rather, it is intended to show how the simulator can be used to aid any study of these systems.

The simulator has been used previously to aid the performance evaluation of Aurora [69], and we feel that it could also be used to evaluate the performance of other parallel Prolog systems. Two of the programs from that study, 8queens1, which solves the 8-queens puzzle, and *parse5*, which is the natural language parsing part of Chat-80, finding all possible parses for a sentence,[6] plus *orsim2(sp2)* from this simulation study, were compared to the results from the simulator. For &-Prolog, *boyer_nsi(2)*, *orsim(sp1)* and *orsim(sp2)* from this simulation study were selected for comparison.

	8queens1 >867×@1300				parse5 58.46×@256				orsim(sp2) 1.47×@5			
#	Time	Act.	Pre.(0)	Pre.	Time	Act.	Pre.(0)	match	Time	Act.	Pre.(0)	match
1	8.02	1×	1×	1× (0)	4.00	1×	1×	1× (0)	2.12	1×	1×	1× (0)
2	4.03	1.99×	2.00× (0.05)	1.99× (78)	2.01	1.99×	2.00× (1.6)	2.00× (0)	2.18	0.97×	1.27× (8.1)	0.97× (32)
3	2.70	2.97×	3.00× (0.2)	2.97× (72)	1.36	2.94×	3.00× (2.9)	2.93× (6)	2.27	0.93×	1.44× (10.1)	0.92× (46)
4	2.03	3.96×	4.00× (0.2)	3.96× (66)	1.07	3.74×	4.00× (3.8)	3.74× (14)	2.28	0.93×	1.45× (10.6)	0.94× (44)
5	1.66	4.84×	5.00× (0.4)	4.84× (70)	0.87	4.60×	5.00× (6.5)	4.58× (12)	2.33	0.91×	1.47× (13.8)	0.91× (36)
6	1.39	5.78×	5.99× (0.3)	5.78× (130)	0.74	5.41×	6.00× (10.0)	5.45× (8)	2.26	0.94×	1.47× (13.8)	0.95× (32)
7	1.21	6.64×	6.99× (0.5)	6.62× (98)	0.66	6.07×	7.00× (12.8)	6.04× (10)	2.30	0.92×	1.47× (13.8)	0.93× (34)
8	1.06	7.57×	7.99× (0.5)	7.54× (100)	0.62	6.45×	7.99× (14.6)	6.38× (16)	2.31	0.92×	1.47× (13.8)	0.93× (34)
9	0.94	8.55×	8.99× (0.5)	8.56× (74)	0.55	7.27×	8.99× (17.0)	7.23× (12)	2.29	0.93×	1.47× (13.8)	0.93× (34)
10	0.86	9.34×	9.98× (0.8)	9.32× (90)	0.53	7.56×	9.98× (19.3)	7.65× (14)	2.26	0.94×	1.47× (13.8)	0.95× (32)
11	0.79	10.18×	10.98× (0.9)	10.18× (74)	0.49	8.16×	10.98× (22.2)	8.12× (14)	2.28	0.93×	1.47× (13.8)	0.93× (34)

Figure 15: Comparison of actual and predicted speed-up for Muse (version 14.gamma,#1)

Figures 15 — 17 show the comparison of the simulator's results with those of the three parallel Prolog systems. The results for Aurora and Muse were gathered from the a Sequent Symmetry with 12 80386 processors at 16MHz, and those for &-Prolog from a Sequent Symmetry with 10 80386 processors at 20MHz.

In the tables, the maximum performance of each program, as measured by the simulator, is given under their names[7], and the columns have the following meaning:

Number of workers

[6]The sentence was "Which European countries that contain a city the population of which is more than 1 million and that border a country in Asia containing a city the population of which is more than 3 million border a country in Western Europe containing a city the population of which is more than 1 million?"

[7]We were not able to obtain the maximum speedup for *8queens1*; it was still producing reasonable speedups with 1300 workers. The maximum performance is well beyond this.

#	8queens1 >867×@1300				parse5 58.46×@256				orsim(sp2) 1.47×@5			
	Time	Act.	Pre.(0)	match	Time	Act.	Pre.(0)	match	Time	Act.	Pre.(0)	match
1	8.14	1×	1×	1× (0)	5.29	1×	1×	1× (0)	2.32	1×	1×	1× (0)
2	4.06	2.00×	2.00×	2.00× (0)	2.84	1.86×	2.00×	1.86× (36)	2.37	0.98×	1.27×	0.98× (30)
3	2.78	2.93×	3.00×	2.93× (184)	1.84	2.87×	3.00×	2.88× (12)	2.35	0.99×	1.44×	0.99× (38)
4	2.11	3.86×	4.00×	3.85× (296)	1.50	3.53×	4.00×	3.54× (26)	2.33	1.00×	1.45×	1.00× (38)
5	1.65	4.94×	4.99×	4.94× (28)	1.17	4.52×	5.00×	4.54× (14)	2.40	0.97×	1.47×	0.97× (30)
6	1.38	5.90×	5.99×	5.89× (50)	1.03	5.14×	6.00×	5.14× (16)	2.36	0.98×	1.47×	0.99× (28)
7	1.21	6.73×	6.98×	6.73× (72)	0.89	5.95×	7.00×	5.91× (12)	2.35	0.99×	1.47×	0.99× (28)
8	1.05	7.76×	7.98×	7.77× (58)	0.82	6.45×	7.99×	6.39× (16)	2.35	0.99×	1.47×	0.99× (28)
9	0.93	8.75×	8.99×	8.73× (50)	0.76	6.97×	8.98×	6.90× (16)	2.36	0.98×	1.47×	0.99× (28)
10	0.85	9.58×	9.96×	9.58× (52)	0.73	7.25×	9.98×	7.65× (14)	2.37	0.98×	1.47×	0.99× (28)
11	0.78	10.45×	10.96×	10.48× (48)	0.69	7.68×	10.98×	8.12× (14)	2.34	0.99×	1.47×	0.99× (28)

Figure 16: Comparison of actual and predicted speed-up for Aurora (version 0.6/Foxtrot #8)

#	boyer_nsi(2) 12.77×@∼74				orsim(sp1) 1.14×@2				orsim(sp2) 8.32×@∼20			
	Time	Act.	Pre.(0)	match	Time	Act.	Pre.(0)	match	Time	Act.	Pre.(0)	match
1	1.239	1×	1×	1× (0)	0.49	1×	1×	1× (0)	1.799	1×	1×	1× (0)
2	0.670	1.85×	1.97× (1.5)	1.85× (44)	0.43	1.14×	1.14× (0.1)	1.14× (18)	1.1	1.65×	1.87× (0.3)	1.65× (378)
3	0.45	2.75×	2.91× (4.8)	2.77× (8)	0.43	1.14×	1.14× (0.1)	1.14× (18)	0.67	2.69×	2.68× (0.6)	2.68× (0)
4	0.35	3.54×	3.76× (8.1)	3.55× (6)	0.43	1.14×	1.14× (0.1)	1.04× (18)	0.53	3.39×	3.37× (1.4)	3.37× (0)
5	0.299	4.14×	4.53× (15.0)	4.19× (4)	0.43	1.14×	1.14× (0.1)	1.14× (18)	0.439	4.10×	3.75× (1.0)	3.75× (0)
6	0.259	4.78×	5.28× (18.0)	4.81× (4)	0.43	1.14×	1.14× (0.1)	1.14× (18)	0.389	4.63×	4.43× (0.5)	4.43× (0)
7	0.240	5.16×	5.91× (21.0)	5.08× (6)	0.429	1.14×	1.14× (0.1)	1.14× (18)	0.350	5.14×	5.34× (3.8)	5.13× (10)
8	0.230	5.39×	6.53× (21.1)	5.40× (8)	0.429	1.14×	1.14× (0.1)	1.14× (18)	0.329	5.47×	5.84× (3.0)	5.47× (32)
9	0.219	5.66×	7.11× (22.2)	5.67× (10)	0.429	1.14×	1.14× (0.1)	1.14× (18)	0.309	5.82×	6.15× (1.1)	5.82× (34)

Figure 17: Comparison of actual and predicted speedup for &-Prolog (version 0.2.2.2/C1.2)

Time Time in seconds to execute the program on the three systems. This is the fastest of many timings – the fastest time instead of the average time is chosen because the fastest time corresponds closer to the ideal speedup.

Act. The actual speedup of the systems over the execution time on 1 worker.

Pre.(0) Speedup predicted by simulator, assuming 0 units of overhead. The number in bracket is the percentage slowdown of this speedup if an overhead of 8 units is assumed. This gives some idea of how sensitive the speedup is to overheads.

match The amount of overheads needed for the speedup from the simulator to most closely match that of the actual speedup. The predicted speedups obtained with the overheads is shown in brackets.

There is general agreement between the actual speed-ups and the simulated speed-ups. Programs predicted to have high speed-ups have high speed-ups, programs predicted to have low speed-ups have low speed-ups. In addition, the numerical agreement is generally better if the speedup is predicted to be less sensitive to overheads (e.g.

parse5, *boyer_nsi(2)*). In general, the agreement for &-Prolog seems better than for the other two systems; although the speedups for 2 agents seem to be poorer and should be investigated. This behaviour (probably due to scheduler) would probably not have been detected if the simulator were not used.

The results illustrate how the simulator can help in interpreting actual performance data. For example, the reason that such low speedups are achieved by *orsim(sp1)* for &-Prolog and *orsim(sp2)* for Muse and Aurora are due almost entirely to a lack of parallelism, instead of some other implementation factor. Also, the fact that the speedups for *orsim(sp2)* under &-Prolog is not 1-to-1 with number of agents is again due to the amount of available parallelism: this is certainly something that would be very difficult to decide without the simulator's result. However, using the simple overhead model in the simulator, we can obtain even more information: e.g. the results show that both *parse5* (under or-parallelism) and *boyer_nsi(2)* (under independent and-parallelism) have speedups which are very sensitive to overheads, and thus the relatively poor numerical agreement between the actual speedups and the predicted speedups in these cases are probably not due to some very significant overheads; in fact, the agreement for it parse5 for both Muse and Aurora is probably better than their results for *8queens1*, although the numerical agreement seems better.

A useful feature of the simulator's results is that they allow the selection of programs that can stress the actual system with a small number of workers. An example of this is programs with low maximum speed-up and/or fine granularity. The programs used for the &-Prolog comparison, and *orsim(sp2)* from the two or-parallel Prolog comparisons, are examples of such programs. Other examples appeared in Szeredi's study, which he called the "low speed-up group" ("Group L").

Another useful feature of the results is that they allow comparison between different Prolog systems, even if they exploit different forms of parallelism, which makes meaningful direct correlation of results difficult. With the simulator, more meaningful comparison can be made by dividing simulated programs into groups based on their maximum speed-ups and granularity. The idea is to select different programs with similar parallel characteristics for different parallel Prolog systems, and then use them to compare such systems. In this case, *orsim(sp1)* running on &-Prolog can be compared to *orsim(sp2)* on Muse and Aurora, and *boyer_nsi(2)* on &-Prolog can probably be compared to *parse5* (because of their granularity). The results suggest that the agreement between actual and predicted speedups is about the same or perhaps slightly better for &-Prolog. We believe that this is at least partly because scheduling in &-Prolog is much less costly than in either Aurora or Muse.

8 Conclusion and Future Work

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 much greater range of programs than exploiting 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 involved in this combination. We are actively researching incorporating this method into actual implementations in the ACE [28, 54] and the DASWAM/Prometheus systems [59, 61]. 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, even when exploiting both forms of parallelism, there are still programs that cannot be sped up. Some of these may have have *dependent* and-parallelism, which is not exploited by the models studied. We are also examining ways to exploit full dependent and-parallelism within the framework of Prolog [60, 62]. Initial results show that more and-parallelism can be exploited in some realistic programs. Another interesting alternative is to exploit only deterministic and-parallelism, as in the Andorra model [55, 56], for which interesting results have been shown. This leaves out some independent and-parallelism (for example, and-parallelism of independent goals which contain choice points) but provides deterministic dependent and-parallelism at potentially less cost than a full dependent and-parallel system.

The simulator has provided us with valuable information on the nature of both independent and- and or-parallelism, and has allowed us to better understand the results from actual implementations. For example, it allowed us to see how close systems like Aurora, Muse, and &-Prolog come to achieving the ideal speedups predicted by the simulator. The information obtained has already been used to refine implementations, as done by Szeredi [69], or to develop new compilation technology, as done by us in the context of non-strict independence [12]. Furthermore, it allowed us to sensibly compare the results obtained from &-Prolog to those obtained from Aurora and Muse, running different benchmarks. We expect that the simulator's results can also be applied to better understand other implementations.

Finally, we are also currently using the simulator to study the quality of the automatic annotation technology that has been developed for &-Prolog. We also plan to use the simulator to help us better evaluate some of the new parallel Prolog systems we are developing, such as ACE [28, 54], DASWAM [62], and CIAO [33, 31].

9 Acknowledgements

We thank Vítor Santos Costa, David Warren and other members from the PEPMA and ParForCE ESPRIT Projects, the Gigalips group, the CLIP group at UPM, and the anonymous referees for their help and discussions on this work. We also gratefully acknowledge the invaluable help of Mats Carlsson with SICStus Prolog, without which this work would not have been possible. We also thank Bristol for the use of their Symmetry. The Sequent Symmetry at UPM was made possible by a grant from British Telecom. The original work described in this paper was funded in part by ESPRIT project PEPMA, while the latest work was funded in part by ESPRIT project ParForCE and CICYT project IPL-D.

A Programs simulated

The benchmark-type programs are the following:[8]

qsort(20) : This is a version of the quick-sort algorithm. A list of 20 numbers, generated randomly, is sorted in this example.

qsort(100) : A longer list of 100 random numbers is sorted.

serialise : This program takes an input list and converts each item to a number. The number is the order of that item in the sorted list. The list for this simulation consists of 25 characters.

numbers : This is a solution to a simple numeric puzzle. It is an example of a simple 'generate and test' program.

4Queens1 : This is a solution to the 4-Queens problem. This is a sequentially inefficient version of the solution.

4Queens2 : This is another solution to the 4-Queens problem. It is sequentially more efficient than *4Queens1*.

map1 : This is a program to solve the map colouring problem, i.e. colouring a map such that no 2 neighbouring countries have the same colour. All the solutions to the problem are returned. The data consists of 5 countries, 4 colours, with the colour for one of the countries pre-set.

atlas : This program searches a database consisting of populations and areas of countries, and finds pairs of countries with population densities that are within 5% of each other. The database consists of 25 countries.

deriv : This program does symbolic differentiation by specifying the differentiation rules in Prolog.

vmatrix(10) : This program multiplies a 10 by 10 matrix and a 10 by 1 matrix. The matrix is represented as a list of lists.

matrix(10) : This program multiplies a 10 by 10 matrix and another 10 by 10 matrix. The matrix is represented as a list of lists.

tak : This is a translation of the standard Takeuchi Lisp benchmark [25] by Evan Tick [70].

hanoi : This program solves the Towers of Hanoi problem. The example is for 9 discs.

cluster : This is an implementation of the core part of a network clustering algorithm used by British Telecom Research Labs. This program was written by A. Beaumont to exploit and-parallelism, based on an original British Telecom program. It is used as a benchmark for Andorra-I [78]. Here, the clustering is performed on 100 elements, instead of the 500 used in Andorra-I.

The application-type programs are the following:

[8]The text of the programs cannot be included for space reasons. They are available however by ftp by contacting the authors. The simulator itself is also available in order to be able to generate ideal parallelism numbers for other benchmarks.

warplan(wq1) : This is Warren's Warplan planning program. A plan is generated for moving a robot to a particular point in a "strips" world.

warplan(wq2) : Warplan generates a plan for moving a robot to a certain location in a "blocks world."

compiler(cp1) : This is a slightly modified version of the public domain Prolog compiler by Van Roy [73], compiling a version of the atlas benchmark that has a smaller database.

This version contained no or-parallelism because of the limitations of the simulator at the time this test was done. And-parallelism was annotated by hand at the top-level only.

compiler(cp2) : This is Van Roy's compiler compiling a version of the deriv benchmark with only the differentiation rules.

compiler(cp3) : This is Van Roy's compiler, with further annotations for and-parallelism obtained by using the annotator, and with or-parallelism. The compiled code is a small subset – 8 clauses – of the database predicates of atlas.

boyer_nsi(2) : This is the version of the Prolog Boyer theorem prover benchmark, translated by Evan Tick [70] from the one in the Gabriel Lisp benchmarks [25]. The theory used is a simple tautology:

$$((x \rightarrow y) \wedge (y \rightarrow z)) \Rightarrow (x \rightarrow z)$$

where

$$x = f((a+b)+(c+0))$$

$$y = f((a \times b) \times (c+d))$$

$$z = lessp(remainder(a,b), member(a, length(b)))$$

tp : This is a version of a propositional theorem prover by Ross Overbeek. It has been modified by Mats Carlsson and Carl Kesselman for more efficient sequential execution. Here, one of the supplied example theorems (ct.3) was used.

chatp(cq1) : This program is the natural language analysis part of the Chat-80 system, starting from the list of input words to the generation of the final query (i.e. after rearrangement of goals by query planning). [9] The parsed question was "Where is China?".

chatp(cq2) : Same program as *chatp(cq1)*, with the question "Is London in United Kingdom?".

chatp(cq3) : Same program as *chatp(cq1)*, with the question "Which countries are European?".

sim(sp2) : This is the first part of the and/or simulator itself (slightly modified so that it can simulate itself). The and-parallelism was produced by using the annotator. The program simulated by the simulated simulator is the atlas program with a database of 6 countries.

orsim(sp2) : This is an older version of the first part of the simulator which simulates or-parallelism only. It was modified from the original or-parallel simulator so that it is basically the same as the or-parallel part of the and-or simulator. The program was modified slightly and hand annotated with independent and-parallelism. The same small atlas program as in *sim(sp2)* is simulated.

orsim(sp1) : This is the or simulator simulating a 5 element naïve reverse.

annotator : This is the annotator used for generating and-parallelism, running on a small set of clauses.

floorplan : This is a floor plan design program by L. B. Kovács [44]. It generates valid partitions of an area into various rooms, given a set of constraints. The query used has 6 rooms, constrained to be within certain sizes, and six additional requirements such as placement of windows.

[9] The results presented here are significantly different from those presented in [64] because more of the Chat system is simulated here, and also because of the change of behaviour of "cut" in the simulator.

References

[1] K. A. M. Ali. OR-Parallel Execution of Prolog on BC-Machine. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 2*, pages 1531–1545. The MIT Press, 1988.

[2] K. A. M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445–475, 1990.

[3] K. A. M. Ali and R. Karlsson. The Muse Approach to Or-Parallel Prolog. Technical Report SICS/R-90/R9009, Swedish Institute of Computer Science, 1990.

[4] K. A. M. Ali and R. Karlsson. Scheduling Speculative Work in Muse and Performance Results. *International Journal of Parallel Programming*, 21(6), December 1992. Published in Sept. 1993.

[5] J. Barklund. *Parallel Unification*. PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1990.

[6] U. C. 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 *Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Volume 3*, pages 841–850, 1988.

[7] A. J. Beaumont and D. H. D. Warren. Scheduling Speculative Work on Or-parallel Prolog Systems. In *Logic Programming: Proceedings of the Tenth International Conference*, pages 135–149. The MIT Press, 1993.

[8] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: the Language and its Implementation. In D. S. Warren, editor, *Proc. 10th Intl. Conf. Logic Programming*, pages 283–298, Cambridge, Mass., 1993. MIT Press.

[9] P. Biswas, S. Su, and D. Yun. A Scalable Abstract Machine Model to Support Limited-OR/Restricted AND Parallelism in Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 1160–1179, Seattle,Washington, 1988.

[10] P. Biswas, S.-C. Su, and D. Y. Y. Yun. A Scalable Abstract Machine Model to Support Limited-OR (LOR)/Restricted-AND Parallelism (RAP) in Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 2*, pages 1160–1179. The MIT Press, 1988.

[11] 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.

[12] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.

[13] J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

[14] 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*, Manchester, U.K., 1985.

[15] W. F. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5(4):361–376, 1988.

[16] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, second edition, 1984.

[17] 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.

[18] J. S. Conery and D. F. Kibler. Parallel interpretation of logic programs. In *Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture.*, pages 163–170, October 1981.

[19] S. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[20] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task granularity analysis in logic programs. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 174–188, June 1990.

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

[22] A. L. Delcher and S. Kasif. Some Results on the Complexity of Exploiting Dependency in Parallel Logic Programs. *The Journal of Logic Programming*, 6(3):229–241, May 1989.

[23] B. S. Fagin and A. M. Despain. The Performance of Parallel Prolog Programs. *IEEE Transactions on Computers*, 39(12):1434–1445, Dec. 1990.

[24] M. J. Fernández, M. Carro, and M. Hermenegildo. IDRA (IDeal Resource Allocation): A Tool for Computing Ideal Speedups. In *ICLP Workshop on Parallel and Data Parallel Execution of Logic Programs*, 1994.

[25] R. P. Gabriel. *Performance and Evaluation of Lisp Systems*. Computer Systems Series. The MIT Press, 1985.

[26] 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.

[27] G. Gupta and M. V. Hermenegildo. Recomputation based Implementation of And-Or Parallel Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992, Volume 2*, pages 770–782. Institute for New Generation Computing, June 1992.

[28] G. Gupta, M. V. Hermenegildo, E. Pontelli, and V. S. Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–110. MIT Press, June 1994.

[29] G. Gupta and B. Jayaraman. Combined And-Or Parallelism on Shared Memory Multiprocessors. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference, 1989, Volume 1*, pages 332–349. The MIT Press, 1989.

[30] G. Gupta, V. Santos Costa, R. Yang, and M. V. Hermenegildo. IDIOM: A Model Integrating Dependent-, Independent-, and Or-parallelism. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of 1991 International Symposium*, pages 152–166. The MIT Press, 1991.

[31] M. Hermenegildo, F. Bueno, M. G. de la Banda, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995. Available from `http://www.cliplab.org/`.

[32] M. Hermenegildo and M. Carro. Relating Data–Parallelism and And–Parallelism in Logic Programs. In *Proceedings of EURO–PAR'95*, number 966 in LNCS, pages 27–42. Springer-Verlag, August 1995.

[33] M. Hermenegildo and T. CLIP Group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.

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

[35] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[36] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[37] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, The University of Texas At Austin, 1986.

[38] M. V. Hermenegildo and K. J. Green. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In D. H. D. Warren and P. Szeredi, editors, *Logic Programming: Proceedings of the Seventh International Conference*, pages 253–268. The MIT Press, 1990.

[39] L. Hirschman, W. C. Hopkins, and R. C. Smith. Or-Parallel Speed-Up in Natural Language Processing: A Case Study. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 263–279. The MIT Press, 1988.

[40] V. K. Janakiram, D. P. Agrawal, and R. Mehrotra. A Randomised Parallel Backtracking Algorithm. *IEEE Transactions on Computers*, 37(12):1665–1676, Dec. 1988.

[41] L. V. Kalé. *Parallel Architectures for Problem Solving*. PhD thesis, State University of New York at Stony Brook, 1985.

[42] L. V. Kalé. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming*, pages 125–133. IEEE, 1987.

[43] A. King and P. Soper. Granularity analysis of concurrent logic programs. In *The Fifth International Symposium on Computer and Information Sciences*, Nevsehir, Cappadocia, Turkey, October (1990).

[44] L. B. Kovács. An Incremental Prolog Systems Development for Floor Plan Design by Dissecting. In *Proceedings of The Practical Application of Prolog, volume 2*, Apr. 1992.

[45] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. MIT Press, August 1988.

[46] Z. Lin. Self-organizing Task Scheduling for Parallel Execution of Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 859–868. ICOT, 1992.

[47] P. López-García, M. Hermenegildo, and S. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In H. Hong, editor, *Proc. of First International Sym-*

posium on Parallel Symbolic Computation, PASCO'94, pages 133–144. World Scientific, September 1994.

[48] E. L. Lusk, R. Butler, T. Disz, R. Olson, R. A. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Vol. 3*, pages 819–830. Institute for New Generation Computer Technology, 1988.

[49] D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218:19–22, Apr. 1968.

[50] H. Millroth. Reforming Compilation of Logic Programs. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of 1991 International Symposium*, pages 485–499. The MIT Press, 1991.

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

[52] L. Naish. Parallelizing NU-Prolog. In R. Kowalski and K. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. University of Washington, MIT Press, August 1988.

[53] R. A. Overbeek, J. Gabriel, T. Lindholm, and E. L. Lusk. Prolog on Multiprocessors. Internal report, Argonne National Laboratory, Argonne, IL 60439, 1985.

[54] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*, pages 564–572. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.

[55] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *Proceedings of the Eighth International Conference on Logic Programming*, 1991.

[56] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I compilation. *New Generation Computing*, 14(1), Jan. 1996.

[57] D. C. Sehr and L. V. Kalé. Estimating the Inherent Parallelism in Prolog Programs. In *Proceedings of the International Conference on Fifth Generation Computer System*, pages 783–790. ICOT, 1992.

[58] K. Shen. An Investigation of the Argonne Model of Or-Parallel Prolog. Master's thesis, University of Manchester, 1986. Available as University of Manchester Computer Science Technical Report UMCS-87-1-1.

[59] K. Shen. Prometheus: An And/Or Parallel Prolog — A High-level View. Technical Report TR-91-39, Computer Science Department, University of Bristol, 1991.

[60] K. Shen. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming,* pages 717–731. The MIT Press, 1992.

[61] K. Shen. *Studies of And/Or Parallelism in Prolog.* PhD thesis, Computer Laboratory, University of Cambridge, 1992.

[62] K. Shen. Implementing Dynamic Dependent And-parallelism. In *Logic Programming: Proceedings of the Tenth International Conference,* pages 167–183. The MIT Press, 1993.

[63] K. Shen and M. V. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of 1991 International Symposium,* pages 135–151. The MIT Press, 1991.

[64] K. Shen and D. H. D. Warren. A Simulation Study of the Argonne Model for Or-Parallel Execution of Prolog. In *Proceedings of the Fourth Symposium on Logic Programming.* Computer Society Press of the IEEE, Sept. 1987.

[65] R. Y. Sindaha. The Dharma scheduler – Definitive scheduling in Aurora on Multiprocessors Architecture. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing,* pages 296–303. IEEE Computer Society Press, Dec. 1992.

[66] R. Y. Sindaha. Branch-level Scheduling in Aurora: The Dharma Scheduler. In D. Miller, editor, *Logic Programming: Proceedings of the 1993 International Symposium,* pages 403–419. The MIT Press, 1993.

[67] A. Singhal and Y. N. Patt. Unification Parallelism: How much can we exploit? In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of North American Conference 1989, Volume 2,* pages 1135–1147. The MIT Press, 1989.

[68] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[69] P. Szeredi. Performance Analysis of the Aurora Or-Parallel System. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference, 1989, Volume 2,* pages 713–732. The MIT Press, Oct. 1989.

[70] E. Tick. Memory Performance of Lisp and Prolog Programs. In *Third International Conference on Logic Programming,* number 225 in Lecture Notes in Computer Science, pages 642–649. Imperial College, Springer-Verlag, July 1986. Also available as Stanford University Technical Report CSL-TR-86-291.

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

[72] P. Tinker and G. Linstrom. A Performance-Oriented Design for Or-Parallel Logic Programming. In J.-L. Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference, Volume 2,* pages 601–615. The MIT Press, 1987.

[73] P. Van Roy. A Prolog Compiler for the PLM. Master's thesis, University of California at Berkeley, August 1984. Also available as Technical Report UCB/CSD 84/203.

[74] D. H. D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87,* Lecture Notes in Computer Science. Springer-Verlag, March 1987.

[75] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming,* pages 684–699. MIT Press, August 1988.

[76] W. Winsborough and A. Wærn. Transparent And-Parallelism in the Presence of Shared Free Variables. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1,* pages 700–710. The MIT Press, 1988.

[77] M. J. Wise. *Prolog Multiprocessors*. Prentice-Hall, 1986.

[78] R. Yang, A. J. Beaumont, I. Dutra, V. Santos Costa, and D. H. D. Warren. Performance of the Compiler-based Andorra-I System. In *Logic Programming: Proceedings of the Tenth International Conference,* pages 150–166. The MIT Press, 1993.

[79] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A. V. S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992, Volume 2,* pages 809–816. Institute for New Generation Computing, June 1992.