Non-Strict Independent And-Parallelism

Manuel Hermenegildo[2]
Universidad Politécnica de Madrid (UPM)
Facultad de Informática
28660-Boadilla del Monte, Madrid - Spain
`herme@fi.upm.es` *or* `herme@cs.utexas.edu`

Francesca Rossi
University of Pisa
Computer Science Department
Corso Italia 40
56100 Pisa, Italy
`rossi@dipisa.di.unipi.it`

Abstract

This paper presents and develops a generalized concept of Non-Strict Independent And Parallelism (NSIAP). NSIAP extends the applicability of Independent And-Parallelism (IAP) by enlarging the class of goals which are eligible for parallel execution. At the same time it maintains IAP's ability to run non-deterministic goals in parallel and to preserve the computational complexity expected in the execution of the program by the programmer. First, a parallel execution framework is defined and some fundamental correctness results, in the sense of equivalence of solutions with the sequential model, are discussed for this framework. The issue of efficiency is then considered. Two new definitions of NSI are given for the cases of pure and impure goals respectively and efficiency results are provided for programs parallelized under these definitions which include treatment of the case of goal failure: not only is reduction of execution time guaranteed (modulo run-time overheads) in the absence of failure but it is also shown that in the worst case of failure no speed-down will occur. In addition to applying to NSI, these results carry over and complete previous results shown in the context of IAP which did not deal with the case of goal failure. Finally, some practical examples of the application of the NSIAP concept to the parallelization of a set of programs are presented and performance results, showing the advantage of using NSI, are given.

1 Introduction

A number of proposed parallel execution models for logic programs include exploitation of "independent and-parallelism" (IAP) (e.g. see [7], [14], [6], [3], [5], [12], [15], [19], [20] and their references). One reason for this is that, as shown in [10], IAP-based models have the highly desirable properties of allowing parallel execution of non-deterministic goals and offering performance improvements through the use of parallelism while at the same time preserving the conventional "don't know" semantics of logic programs.[3] In addition, other properties were shown in [10] regarding the computational complexity of IAP execution when compared with that of sequential execution, i.e. with the complexity which is expected by the programmer when writing the program. However, only results for the case when no goal failure occurs were presented. Also in [10], the conventional notion of goal independence (Strict Independence –SI) is extended by defining a notion of Non-Strict Independence (NSI) which allowed the execution of more goals in parallel than SI. However the notion of NSI given is somewhat restricted. Furthermore, no practical examples of the benefits attainable through NSI were given.

This paper pursues further the subjects of extending the applicability of IAP and studying its properties. New, separate definitions of NSI are proposed for the cases where the goals involved are pure or non-pure. Both of these definitions represent relaxations of the one presented in [10]. In addition, the treatment of the efficiency of the parallel execution of SI and NSI goals is completed by providing results for the different cases when goal failure occurs. Without loss of generality, and for reasons of conciseness, only the case of NSI is treated in this paper: the results are directly applicable to SI, since NSI subsumes SI. Finally, some practical examples of the application of the NSIAP concept to the parallelization of a set of programs are presented and performance results from the execution of the NSI-annotated programs on an actual parallel system are given which illustrate the advantages of using NSI. A particular case of NSI was hinted at by DeGroot [7] and Chang [5] in the "qsort" example. The MA3 system, presented in [18], incorporated an earlier concept of NSI. Some of the problems that our definition of NSI tries to characterize and avoid have also been informally discussed by Winsborough and Waern in [20].

This paper is written in the spirit of an extended abstract, with the objective of providing more intuition into the proposed concepts. Formal descriptions and complete proofs are given in [11]. The rest of the paper proceeds as follows: Section 2 illustrates the correctness and complexity issues involved in running independently in parallel goals which share variables. Based on these considerations, Section 3 gives the definition of non-strict independence for pure goals, while Section 4 gives the equivalent definition

[3]This property also holds trivially in OR-parallel models – see [17] and its references–, and in the Andorra model [4]. In the Andorra model described in [4], though, only deterministic goals are allowed to execute in (dependent) and-parallel. The concepts presented in this paper are of direct applicability to the inclusion of generalized forms of IAP into an Andorra-like framework (such as in "Kernel Andorra Prolog").

for any combination of pure and impure goals. Section 5 then compares the parallel and sequential proof length for the class of goals we consider. Finally, Section 6 presents some results from a practical application of non-strict independence to the parallelization of a logic version of a subset of the B-M theorem prover showing actual performance data on the &-Prolog parallel system, and Section 7 summarizes the contributions of the paper.

2 Independent Parallel Execution of Goals

We now introduce a new execution framework, which is similar to the usual sequential one in logic programming (i.e. that one of SLD-resolution), but where some of the goals can be *independently* run in parallel. Although such a framework is not the most general one for describing and-parallel execution of goals, since only independent executions are handled, it is sufficient for our purposes: in in this paper we are mostly interested in describing properties of models which run goals in parallel independently.[4] Throughout the paper, the notation used follows that of Lloyd [16] and Apt [2, 1].

The presentation in this section will be rather informal: its purpose is to provide intuition for the reading of the following, more formal, sections. In this spirit, we describe the intuitive idea in the model considered as follows: partition the given resolvent so as to obtain some new parallel resolvents and a remaining ("continuation") part, execute the parallel resolvents, and then embed the information gathered from such execution into the remaining part. There are two main changes with respect to the sequential framework:

- the usual sequential SLD-resolution proof procedure at each step selects only one goal in the current resolvent. Obviously, if we want to run some of the goals in this resolvent in parallel we have to allow the selection of more than one goal.
- while in the sequential framework the result of the execution of one of the goals is made visible to the other goals by the usual notion of composition of substitutions, such a notion is not sufficient to express the combination of the results of the parallel execution of two or more goals. Thus we need to treat the case of parallel composition of substitution specially.

Let us explain more precisely the two frameworks. Assume $G = (g_1, \ldots, g_n)\theta$ is the current resolvent. The sequential SLD-resolution proof procedure with left-to-right selection rule would

- execute $g_1\theta$ obtaining the answer substitution θ_1,
- execute $g_2\theta\theta_1$, obtaining θ_2,

[4]Other execution frameworks, which allow parallel executions to affect each other and more flexible synchronization of goals, can be considered as well, and their combination with the one described here leads to a most general framework. However, this subject is beyond the scope of this paper.

- execute $g_3\theta\theta_1\theta_2$, obtaining θ_3,

and so on until the execution of all the goals in G.

In this framework the composition of substitutions is formally defined as follows (see [1]): consider two substitutions $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ and $\eta = \{y_1/s_1, \ldots, y_m/s_m\}$. Their composition $\theta\eta$ is obtained by

1. constructing the set $\{x_1/t_1\eta, \ldots, x_n/t_n\eta, y_1/s_1, \ldots, y_m/s_m\}$,
2. deleting the pairs x_i/t_i such that $x_i = t_i$, and
3. deleting the pairs y_i/s_i such that $y_i \in \{x_1, \ldots, x_n\}$.

If, on the contrary, we want to run some of the goals in parallel, say g_i and g_j (the extension to more than two goals is straightforward), one possible execution scheme for G could be the following:

- partition G into the resolvents
  - $G_1 = (g_i)\theta$,
  - $G_2 = (g_j)\theta$,
  - $G_3 = (g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_{j-1}, g_{j+1}, \ldots, g_n)\theta$,
- execute G_1 and G_2 in parallel obtaining the answer substitutions θ_1 and θ_2,
- apply the "composition" of θ_1 and θ_2 to G_3,
- execute G_3.

Our goal is, within the execution model presented, to run in parallel as many goals as possible while maintaining correctness and efficiency. In other words, we assume that the semantics described by using a sequential execution (with a left-to-right selection rule) corresponds to the intended model of the program. Thus we would like to preserve such semantics. This means that given a goal G, of which we know the result of its sequential execution, and the time complexity to obtain it, we would like to execute some of the goals in G in parallel obtaining the same answer as before possibly in a shorter time. Note, however, that using the described standard composition of substitutions in the last step above can lead to incorrect results, as shown by the following example. Consider the resolvent
$:-p(x), q(x).$
and the following definition of p and q:
$p(a).$
$q(b).$

In this case, the sequential execution framework first executes $p(x)$, returning $\{x/a\}$ and the new resolvent $:-q(x)\{x/a\}$, i.e. $:-q(a)$, whose execution fails, thus making the whole given resolvent fail. On the other hand, the parallel execution framework executes in parallel $p(x)$ and $q(x)$, returning $\{x/a\}$ and $\{x/b\}$ respectively. Note now that the composition of

$\{x/a\}$ and $\{x/b\}$ is, according to the above definition, the substitution $\{x/a\}$ (the pair x/b being deleted by means of step 3). Thus we obtain a different answer.

In this simple example it is easy to realize that the problem is due to the sharing of the variable x which both the p and q goals try to instantiate. However, incorrect answers can be obtained even when there is no conflicting binding for the shared variables. Let us consider the following example, where we have the resolvent
$: -p(x, y), q(y)$
and the following definitions of p and q:
$p(z, z)$.
$q(a)$.
If we run $p(x, y)$ and $q(y)$ sequentially, we first execute $p(x, y)$ returning $\theta_p = \{x/z, y/z\}$, and then we execute $q(y)\theta_p$, i.e $q(z)$, returning $\theta'_q = \{z/a\}$. Thus, in the end, we obtain the substitution $\theta = \theta_p\theta'_q = \{x/a, y/a, z/a\}$. If we now execute $p(x, y)$ and $q(y)$ in parallel, we obtain θ_p from the execution of $p(x, y)$, and $\theta_q = \{y/a\}$ from the execution of $q(y)$, thus ending with their composition $\theta_p\theta_q = \{x/z, y/z\}$ or $\theta_q\theta_p = \{y/a, x/z\}$ as final substitution, that is obviously different from the θ obtained from the sequential execution, and thus again an incorrect result.

One possible way to avoid the possibility of the parallel execution returning different answers from the sequential one might be to change the definition of composition of substitutions to be used when composing the results of parallel execution. More precisely, the new definition could be recursively defined as follows:

1. construct the set $\{x_1/t_1\eta, \ldots, x_n/t_n\eta, y_1/s_1, \ldots, y_m/s_m\}$,

2. delete the pairs x_i/t_i such that $x_i = t_i$, and

3. delete the pairs y_j/s_j such that $y_j = x_i$ and compose with the mgu of t_i and s_j.

In the first of the above examples, this new definition would fail in computing the composition of $\{x/a\}$ and $\{x/b\}$, because there is no mgu for a and b. Thus the results of the sequential and the parallel execution would be the same.

In the second example, the composition of $\{x/z, y/z\}$ and $\{y/a\}$ would be computed, according to the new definition, to be $(\{x/z, y/z, y/a\} - \{y/a\})$ composed with the mgu of z and a, which is $\{z/a\}$, thus obtaining $\{x/z, y/z\}$ composed with $\{z/a\}$, which is $\{x/a, y/a, z/a\}$. Thus again the answer substitutions of the sequential and the parallel execution coincide.

It is interesting to note that this new definition of composition reduces to the usual one when the goals in the resolvent to be run in parallel do not share any variables. In fact, in this case step 3 of the definition (which is the only different one) will never be applied.

The new definition of composition of substitutions described appears to be useful and appropriate in the context of the parallel execution of goals. However, the adoption of a new definition of composition in principle requires a revision of well known results in logic programming, which rely on the standard definition. This is beyond the scope of this paper, and we will instead adopt a different approach herein: we will transform any set of goals to be executed in parallel into one where no variables are shared, in such a way that correctness (w.r.t. the given resolvent) is preserved, generality is not sacrificed, and the usual definition of composition, with all the well-known results which follow from it, can be used. The transformation has the added advantage that it allows other simplifications at the implementation level since the standard variable representation mechanism can be used.

The transformation that we consider involves eliminating any shared variables in goals which are to be executed in parallel by renaming all their occurrences so that no two occurrences in different goals have the same name, and adding some unification goals to reestablish the lost links. Since the sharing of variables is eliminated, now the standard definition of composition of substitutions can be used. As an example, given the resolvent
$: -p(x), q(x).$
we would obtain the new resolvent
$: -p(x), q(x'), x = x'.$
The idea is that we want different occurrences of x in different goals to have different names. These goals of the form $x' = x$ are called "back-binding" goals, and are related to the back-unification goals defined in [14], and the closed environment concept of [6]. Now p and q do not share any variables in the new resolvent. Thus we can compose their answer substitution by using the usual notion of composition. Also, note that the new resolvent is trivially equivalent (in terms of possible answers) to the given one since the unification goals simply explicitate some bindings.

Let us consider again the two examples above. In the first one, after the transformation we would have
$: -p(x), q(x'), x = x'.$
$p(a).$
$q(b).$

Thus the parallel execution of p and q produces $\{x/a\}$ and $\{x'/b\}$, whose composition is (usual definition) $\{x/a, x'/b\}$. After that, we are left with the resolvent $: -(x = x')\{x/a, x'/b\}$, i.e. $: -a = b$, which fails, as in the sequential execution.

For the other example, we have:
$: -p(x, y), q(y'), y = y'$
$p(z, z).$
$q(a).$

Here the the parallel execution of p and q produces $\{x/z, y/z\}$ and $\{y'/a\}$, whose composition is (usual definition) $\{x/z, y/z, y'/a\}$. After that, we execute the resolvent $: -(y = y')\{x/z, y/z, y'/a\}$, i.e. $: -(z = a)$, which returns $\{z/a\}$. Thus the final answer substitution is the composition of $\{x/z, y/z, y'/a\}$ and $\{z/a\}$, i.e. $\{x/a, y/a, y'/a, z/a\}$, which coincides with the answer substitution obtained by the sequential execution (if projected on the same variables).

Thus, for the rest of this paper, we will use the above described renaming technique whenever we want to execute in parallel goals which share variables. The parallel execution framework proposed at the beginning of this section is now transformed as follows: assume $G = (g_1, \ldots, g_n)$, suppose that we want to execute in parallel g_i and g_j, that the renamed version of these two goals is g'_i and g'_j respectively, and that the new collection of goals generated by the renaming is R. Thus:

- consider the resolvents
  - $G_1 = (g'_i)\theta$,
  - $G_2 = (g'_j)\theta$,
  - $G_3 = R\theta$,
  - $G_4 = (g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_{j-1}, g_{j+1}, \ldots, g_n)\theta$,
- and the steps:
  - execute G_1 and G_2 in parallel obtaining the answer substitutions θ_1 and θ_2,
  - execute $G_3\theta_1\theta_2$ obtaining θ_3,
  - execute $G_4\theta_3$.

Such new parallel scheme is obviously correct, for pure goals, w.r.t. the sequential one, due to the equivalence of the resolvent after the renaming, and to the use of standard composition of substitutions. However, programs in practice often contain extra-logical predicates and this causes additional problems. Two extra-logical predicates of interest are $var/1$ and ! (cut). Consider the following example:
$: -p(x), q(x).$
$p(a).$
$q(y) : -var(y), !, y = b.$
$q(a).$
The proposed renaming of the resolvent would result in
$: -p(x), q(x'), x = x'.$
and the parallel execution would fail while the sequential execution would succeed with $\{x/a\}$. This is a consequence of the fact that the switching lemma ([16]) doesn't hold for goals which are impure. The implication of this in our context is that if there are impure goals in the program not all goals can be parallelized while maintaining correctness in terms of equivalence of solutions with sequential execution. Thus, in this case some conditions on the classes of goals which can be parallelized need to be enforced. Sufficient conditions are presented in section 4.

Having dealt with the issue of correctness of results a remaining issue is the convenience of parallel execution in terms of efficiency. In fact, the parallel execution (according to our model) of any set of goals could (although being correct) result in an increase of the execution time, thus defeating the very aim of parallel computation.

Consider the following example:
$: -p(x), q(x).$
$p(a).$
$q(b) : -proc.$
where *proc* is very costly to execute.
The renamed resolvent is:
$: -p(x), q(x'), x = x'$. Both the sequential and the parallel execution fail. However, the sequential execution executes $p(x)$ returning $\{x/a\}$, and then fails in trying to match $q(x)\{x/a\}$, i.e. $q(a)$, to any rule head, thus never going into the execution of *proc*. On the contrary, the parallel execution executes in parallel $p(x)$ and $q(x')$ returning $\{x/a\}$ and $\{x'/b\}$ (but only after executing *proc* too), and then fails in the execution of the goal $x = x'\{x/a\}\{x'/b\}$, i.e $a = b$. Thus the sequential execution is obviously much more efficient.

The cause of such a problem in this example, and also in general, is that the execution of p affects the execution of q in the sequential framework, thus restricting the search space of q. As mentioned before, our parallel framework does not allow parallel executions to affect each other, so the search space of q is bigger. A solution to this efficiency problem is to parallelize only goals which do not restrict each other's search space in the sequential execution.

In this example, the restriction of the search space of q is due to the instantiation of a shared variable (x) by p. However, this is not the only way to restrict a search space. In fact, aliasing of shared variables that are going to be instantiated by more than one goal can achieve the same effect. Consider the following example:
$: -p(x, y), q(x), r(y).$
$p(z, z).$
$q(a).$
$r(b) : -proc.$
where again *proc* is very costly to execute. Similarly, the sequential execution first executes $p(x, y)$ returning $\{x/z, y/z\}$, then $q(z)$ returning $\{z/a\}$ and then fails in trying to match $r(a)$. The parallel execution on the contrary first executes in parallel $p(x, y)$, $q(x')$ and $r(y')$. Thus $r(y')$ can match with $r(b)$ and this leads to the execution of *proc*. This means that, if we want to be at least as efficient as the sequential execution, we have to eliminate this last situation as well.

Thus a general solution to solve both the above described problems (concerning the restriction of the search space) is to run in our parallel framework only those sets of goals where

- either there are no shared variables, or
- no shared variable is instantiated by more than one goal.

In [10], the first situation is named "strict independence", while the second one is a special case of the concept of "non-strict independence" (restricted to pure goals).

In the following section, we will formalize the considerations of this section into the concept of "non-strict independence" both for pure and impure goals, proving the sufficiency of such concepts for the correctness and efficiency of the parallel execution of non-strictly independent goals.

3 Non-Strict Goal Independence for Pure Goals

As mentioned before our goal is, within the execution model under consideration, to run in parallel as many goals as possible while maintaining correctness and efficiency. However, given a general goal, the sequential and parallel execution frameworks described in the previous section do not necessarily return the same answer, nor are they proved to have the same time complexity. Thus our aim is to define a restricted class of goals for which such properties (correctness and efficiency) can be proved to be preserved in the case of parallel execution. This is the purpose of the following definitions, which describe a class of pure goals whose parallel execution is always correct and efficient.

Definition 1 (v- and nv-binding) : *A binding x/t is called a v-binding if t is a variable, otherwise it is called an nv-binding.*∎

Definition 2 (non-strict independence for pure goals) : *Consider a collection of pure goals $g_1, \ldots, g_n$ and a given substitution θ. Consider also the set of shared variables $SH = \{v \mid \exists i, j, 1 \leq i, j \leq n, i \neq j, v \in (var(g_i\theta) \cap var(g_j\theta))\}$ and the set of goals containing each shared variable $G(v) = \{g_i \mid v \in var(g_i\theta), v \in SH\}$. Let θ_i be the answer substitution for $g_i\theta$. The given collection of pure goals is non-strictly independent for θ if the following two conditions are satisfied:*

- *for all $v \in SH$, at most one $g \in G(v)$ nv-binds v;*
- *$\forall i = 1, \ldots, n$, if $var(g_i\theta)$ contains more than one shared variable, say $x_1, \ldots, x_k$, then $x_1\theta_i, \ldots, x_k\theta_i$ are strictly independent.*∎

The first condition of the above definition requires that at most one goal further instantiates a shared variable, while the second one takes care of the aliasing problem discussed in the previous section.

Note that the situation where the given set of goals does not contain any shared variable (called "strict independence" in [10]) is a special case of the above definition for pure goals. Note also that the above definition of non-strict independence for pure goals is needed only for efficiency purposes. In fact, the parallel execution of any set of pure goals in our framework is always correct, as mentioned in the previous section.

In our framework, the resolvent is not renamed if the pure goals do not share any variable. Otherwise (if the pure goals share variables but they are non-strictly independent), the renaming technique described in the previous section is applied. In both cases, we end up with the parallel execution of a set of pure goals without shared variables, followed by the execution of all the goals added by the renaming, which always succeed. In fact, they are goals of the form $x = x'$ where only one of x and x' can be instantiated, due to the given definition.

We will later discuss the complexity of the parallel execution of a set of pure goals without shared variables w.r.t. to their sequential one. But first we need to consider the situations where pure as well as impure goals can occur.

4 Non-Strict Goal Independence for General Goals

Definition 3 (non-strict independence) *: Consider any collection of pure and/or impure goals $g_1, \ldots, g_n$ and a given substitution θ. Consider also the set of shared variables $SH = \{v \mid \exists i, j, 1 \leq i, j \leq n, i \neq j, v \in (var(g_i\theta) \cap var(g_j\theta))\}$ and the set of goals containing each shared variable $G(v) = \{g_i \mid v \in var(g_i\theta), v \in SH\}$. Let θ_i be the answer substitution for $g_i\theta$. The given collection of goals is non-strictly independent for θ if the following two conditions are satisfied:*

- *for all $v \in SH$, at most the rightmost impure $g \in G(v)$, say g_i, or only one pure $g \in G(v)$ to the right of g_i nv-binds v;*
- *$\forall i = 1, \ldots, n$, if $var(g_i\theta)$ contains more than one shared variable, say $x_1, \ldots, x_k$, then $x_1\theta_i, \ldots, x_k\theta_i$ are strictly independent.*■

The difference between this definition of non-strict independence and that one for only pure goals is that here not only is only one goal allowed to instantiate a shared variable, but this goal must be the rightmost impure goal, or a goal to the right of that. The choice of the rightmost impure goal is not arbitrary, but it is necessary to be able to obtain the same semantics as the sequential execution framework, i.e. of SLD-resolution with left-to-right selection rule. In particular, such selection rule is the one that plays the most important role in this choice. In fact, the execution of an impure goal could obtain different answers if the goal is executed with less instantiated arguments. Thus, if there is any goal to the left of an impure goal which affects its execution in the sequential framework (by instantiating a variable), we should reflect this behaviour in the parallel execution as well. But because our parallel framework does not allow communication between different goals executed in parallel, we have to avoid this case by sequentializing such goals. The proof of the correctness of executing in parallel goals satisfying the above definition can be found in [10].

Again, given a set of (pure and possibly impure) goals, either they do not share variables, or they do but we rename them so that we obtain a new set of goals without shared variables (to be executed in parallel), plus a set of new goals describing the renaming. For the same reason as before, such goals always succeed and collectively involve only one resolution step, so we can consider them irrelevant to the study of the complexity of the parallel execution w.r.t. the sequential one. It is important to point out that, although the renaming has been described as done at run-time on the resolvent, it is possible in many cases to do this renaming and part of the detection of NSI at compile-time. Some techniques for doing this can be found in [10].

5 Complexity results: parallel vs. sequential execution

As noted above, we only need to study the execution of a set of goals without shared variables. We will here consider only two goals, g_1 and g_2, the generalization to any number of goals being straightforward.

Let us call W_s (resp. W_p) the work involved in the sequential (resp. parallel) execution, and T_s (resp. T_p) the time of such execution. In the following we will express them always as a function of the number of execution steps in the goals, i.e. k_1 and k_2.

First we consider the case where no goal fails. For the sequential execution, we have:

- $W_s = k_1 + k_2$ and
- $T_s = k_1 + k_2$.

For the parallel execution we have:

- $W_p = k_1 + k_2$, and
- $T_p = max(k_1, k_2)$.

Thus $W_p = W_s$ and $T_p < T_s$, i.e. the amount of work to be done is the same, but the execution time is always less.

Let us consider now the situation where one of the two goals fails. Here there are two different cases, the first one being the failure of the rightmost goal (i.e. g_2) and the other one being the failure of the leftmost one (i.e. g_1).

In the first case, the sequential execution with the left-to-right selection rule entirely (and unnecessarily) executes g_1 and then g_2 until its failure. On the contrary, assuming the possibility of communication of failure between processors ([13]), the parallel execution executes g_1 and g_2 contemporarely until the failure of g_2. More precisely, we have:

- $W_s = k_1 + k_2$,
- $T_s = k_1 + k_2$,
- $W_p = k_1 + k_2$ iff $k_1 \leq k_2$, and $W_p = 2k_2$ iff $k_2 \leq k_1$, and
- $T_p = k_2$.

Note that in this case $2k_2 \leq k_1 + k_2$, thus $W_p \leq W_s$ and $T_p < T_s$, which is an even better situation w.r.t. the non-failure case, because not only is the time shorter, but the work can be less as well.

The final case concerns the failure of the leftmost goal. Here the sequential execution, because of the left-to-right selection rule, is able to stop with the failure of g_1 without executing g_2, while the parallel one executes both goals until the failure of g_1, thus doing part (or all) of the useless execution of g_2. More precisely, we have:

- $W_s = k_1$,
- $T_s = k_1$,
- $W_p = k_1 + k_2$ iff $k_2 \leq k_1$, and $W_p = 2k_1$ iff $k_1 \leq k_2$), and
- $T_p = k_1$.

Thus $W_p \geq W_s$ and $T_p = T_s$, i.e. the work can be more but the execution time is always the same.

In a practical implementation this is only true, of course, if the parallelism overhead is sufficiently low. However, low overhead appears to be attainable in most cases as demonstrated by systems such as &-Prolog/RAP-WAM [12, 8] and APEX [15]. It is also important to note that the results presented assume a "left biased" scheduling strategy. i.e. one which guarantees that the goals to the left in syntactic order are selected for execution first (this is the case, for example, in the RAP-WAM).

In summary, in absence of failure, or failure of the rightmost goal, the parallel execution shows great advantage. Even in the worst case of failure of the leftmost goal, the execution execution time can never be longer, although some more work could be done. Thus, depending on resource availability, it may be preferable to run in parallel goals that do not fall in the latter situation. This can often be determined at compile time if global analysis of the program is allowed.

It is important to note that the above results hold for finite failures only. In case of infinite failure of the sequential execution, it is possible for the parallel execution to finitely fail. Thus the finite failure set of the parallel execution model could be larger than the one for the sequential model. As an example, let us consider the goal $: -p(x), q(y)$.
and the program
$p(x) : -p(x)$.
In this case, the sequential execution would infinitely fail in the execution of $p(x)$, while the parallel execution would finitely fail due to the finite failure of $q(y)$.

However, it is even more important to note that finite failures of the sequential execution can never be transformed into infinite failures in the parallel framework. Thus the infinite failure set is never enlarged by executing in parallel some of the goals.

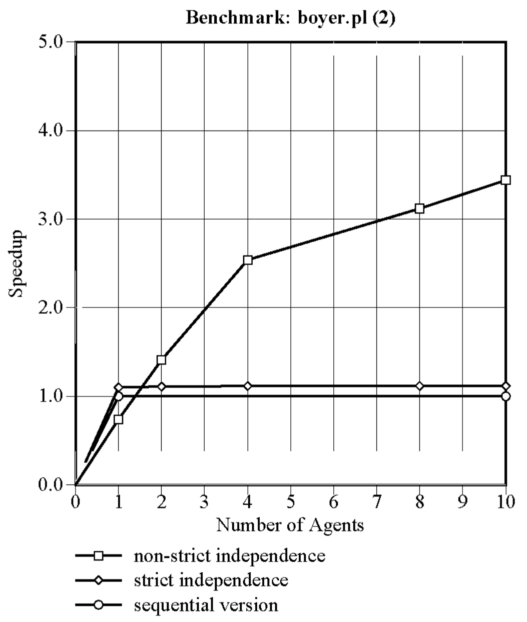


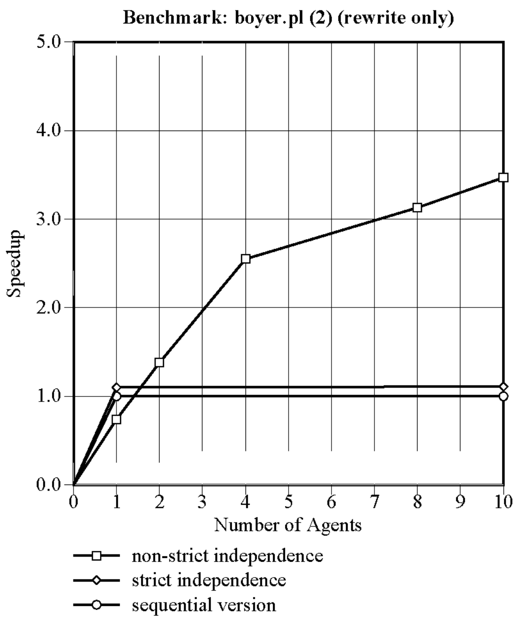
Figure 1: Speedup for `boyer(2)` - strict vs. non-strict independence

Nproc	seq	si	nsi
1	6338 (6179 + 159)	6338 (6179 + 159)	8479 (8320 + 159)
2	6339 (6179 + 160)	6269 (6179 + 90)	4479 (4389 + 90)
4	6339 (6179 + 160)	6238 (6169 + 69)	2488 (2419 + 69)
8	6339 (6179 + 160)	6228 (6169 + 59)	2029 (1970 + 59)
10	6339 (6179 + 160)	6228 (6169 + 59)	1838 (1779 + 59)

Table 1: Execution Time, `boyer.pl` (`rewrite` + `prove`) on Sequent Balance: 1-10 Processors, sequential vs. strict indep. vs. non-strict indep.

6 Using Non-Strict Independence in Practice

In this section we present actual run-times from the result of parallelizing a relatively large benchmark (`boyer`, a reduced version of the B-M theorem prover written by Evan Tick) using both strict- and non-strict independence. This benchmark proves theorems in basically two steps: a rewriting step ("rewrite," which comprises most of the computation) and a tautology checking step ("prove"). Table 1 gives execution times for the benchmark running on the unoptimized version of the &-Prolog system [9], using 1-10 Sequent Balance processors, for the original, sequential program and for the cases in which the program has been parallelized using either strict- or non-strict independence. The results for the whole benchmark are represented in speedup form in figure 1. It can be observed that while no useful speedup



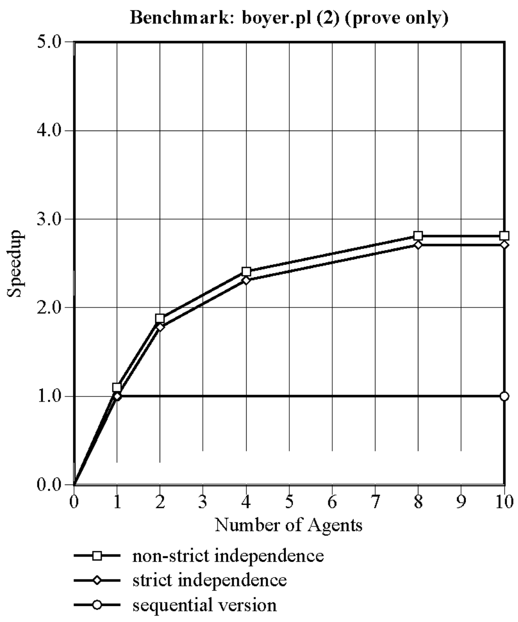


Figure 2: Speedup for "rewrite" and "prove" of `boyer(2)` - strict vs. non-strict independence

can be obtained by using strict independence, reasonable speedups[5] can be obtained using the non-strict independence notion. It is interesting to observe, as shown in figure 2, that strict-independence is relatively successful at parallelizing the "prove" part of the algorithm. On the other hand it is unsuccessful at parallelizing the "rewrite" part, while non-strict independence parallelizes both. The fact that, as can be seen in table 1, "rewrite" represents the bulk of the computation, explains why, despite parallelizing the "prove" part correctly, no significant speedup is observed for non-strict independence in the whole benchmark.

7 Conclusions

Having presented a framework for the independent and-parallel execution of goals, we discussed the problems concerning the correctness and the efficiency of running goals using such framework. In particular, we showed that the traditional composition of substitutions cannot be used in such a framework, and proposed two solutions: using a new definition of composition, which we obtained as an extension of the old one, or performing a renaming of the goals and using the traditional composition. Based on such discussion, we proposed two restrictions to the classes of goals that can be executed in parallel while preserving correctness and the efficiency of the sequential execution for the cases of only pure goals or a mixture of pure and impure goals.

[5]This speedup can be made arbitrarily large by using appropriate data. In this case a theorem requiring a relatively small proof was used.

Also, efficiency results were provided for the parallel execution of goals in the proposed classes, which included treatment of the case of goal failure. We showed that not only is reduction of execution time guaranteed (modulo run-time overheads) in the absence of failure but also that in the worst case of failure no speed-down occurs. Finally, we showed actual performance results from the parallel execution of the non-strictly independent goals in a logic program version of the Boyer-Moore theorem prover. These results show that significant additional speedup can be obtained through the use of non-strict independence.

References

[1] K. Apt. Introduction to Logic Programming. Technical Report TR-87-35, Dept. of Computer Science, The University of Texas at Austin, July 1988.

[2] K. Apt and M. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–863, July 1982.

[3] 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, 1988.

[4] P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog—The Language and Application in Distributed Simulation. In *International Conference on Fifth Generation Computer Systems.* Tokyo, November 1988.

[5] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.

[6] J. S. Conery. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *Symp. on Logic Prog.*, pages 457–467, August 1987.

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

[8] M. V. Hermenegildo et al. An overview of the pal project. Technical Report ACT-ST-234-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, 1989.

[9] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. Technical Report ACA-ST-536-89, MCC, Austin, TX 78759, November 1989.

[10] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[11] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. Technical Report ACA-ST-537-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, November 1989.

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

[13] M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 40–55. Imperial College, Springer-Verlag, July 1986.

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

[15] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[16] J. W. Lloyd. *Logic Programming.* Springer-Verlag, 1984.

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

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

[19] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.

[20] W. Winsborough and A. Waern. Transparent and- parallelism in the presence of shared free variables. In *Fifth International Conference and Symposium on Logic Programming*, pages 749–764, 1988.