A High-Level Implementation of Non-deterministic, Unrestricted, Independent And-Parallelism

Amadeo Casas , Manuel Carro , and Manuel V. Hermenegildo

Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, USA
School of Comp. Science, Univ. Politécnica de Madrid, Spain and IMDEA-Software

Abstract. The growing popularity of multicore architectures has renewed interest in language-based approaches to the exploitation of parallelism. Logic programming has proved an interesting framework to this end, and there are parallel implementations which have achieved significant speedups, but at the cost of a quite sophisticated low-level machinery. This machinery has been found challenging to code and, specially, to maintain and expand. In this paper, we follow a different approach which adopts a higher level view by raising some of the core components of the implementation to the level of the source language. We briefly present an implementation model for independent and-parallelism which fully supports non-determinism through backtracking and provides flexible solutions for some of the main problems found in previous and-parallel implementations. Our proposal is able to optimize the execution for the case of deterministic programs and to exploit *unrestricted* and-parallelism, which allows exposing more parallelism among clause literals than fork-join-based proposals. We present performance results for an implementation, including data for benchmarks where and-parallelism is exploited in non-deterministic programs.

Keywords: And-Parallelism, High-level Implementation, Prolog.

1 Introduction

New multicore technology is challenging developers to create applications that take full advantage of the power provided by these processors. The path of single-core microprocessors following Moore's Law has reached a point where very high levels of power (and, as a result, heat dissipation) are required to raise clock speeds. Multicore systems seem to be the main architectural solution path taken

by manufacturers for offering potential increases in performance without running into these problems. However, applications that are not *parallelized*, will show little or no improvement in performance as new generations with more processors are developed. Thus, much effort is currently being put and progress being made towards alleviating the hard task of producing parallel programs. This includes the design of new languages that provide better support for the exploitation of parallelism, libraries that offer improved support for parallel execution, and parallelizing compilers, capable of helping in the parallelization process.

In particular, declarative languages (and logic programming languages among them), have been traditionally considered an interesting target for exploiting parallelism. Their high-level nature allows a coding style closer to the problem which preserves more of the original parallelism. Their separation between control and the declarative meaning, together with relatively simple semantics, makes logic programming a formally simpler framework which, however, allows studying and addressing most of the challenges present in the parallelization of imperative languages

There are two main forms of parallelism in logic programming *Or-parallelism* refers to the execution of different branches in parallel, while *And-parallelism* executes simultaneously some goals in the resolvent. The latter can be exploited independently of whether there is implicit search or not. Two main forms of and-parallelism have been studied. Independent and-parallelism (IAP) arises between two goals when the execution of one of them does not influence the execution of the other. For pure goals a sufficient (and a-priori) condition for this is the absence of variable sharing at run-time among these goals. "Dependent" and-parallelism (DAP) is found when the literals executed in parallel share variables at run-time, and they compete to bind them. In this paper we will focus on independent and-parallelism.

Systems like &-Prolog DDAS and others have exploited and-parallelism, while certain combinations of both and- and or-parallelism have been exploited by e.g. &ACE AKL , and Andorra-I Many of these systems adopted similar implementation ideas. This often included a parallelizing compiler to automatically transform the original program into a semantically-equivalent parallel version of it and a run-time system to exploit the potential increase in performance provided by the uncovered parallelism. These systems have been shown very effective at exploiting parallelism efficiently and obtaining significant speedups However, most of them are based on quite complex, low-level machinery (which included an extension of the WAM instructions, and new data structures and stack frames in the stack set of each agent), which makes implementation and maintenance inherently hard.

we proposed a high-level implementation that raised some of the main components of the implementation to the source level, and was able to exploit the flexibility provided by unrestricted and-parallelism (i.e., not limited to fork-join operations). However, provided a solution which is only valid for the parallel execution of goals which have exactly one solution each, thus avoiding some of the hardest implementation problems. While it can be argued that a

large part of application execution is indeed single-solution, on one hand this cannot always be determined a priori, and on the other there are also cases of parallelism among non-deterministic goals, and thus a system must offer a complete implementation, capable of coping with parallel non-deterministic goals, in order to be realistic. Other recent related work includes which proposes a set of high-level multithreading primitives. This work focuses more on providing a flexible multithreading interface, rather than on performance.

In this paper, we present a high-level implementation that is able to exploit unrestricted IAP over non-deterministic parallel goals, while maintaining the optimizations of previous solutions for non-failing deterministic parallel goals. Our proposal provides solutions for the trapped-goal and garbage-slot problems, and is able to cancel the execution of a parallel goal when needed.

2 Decomposing And-Parallelism

Independent and-parallelism has traditionally been expressed using the (restricted, i.e., fork-join) `&/2` operator as the lowest-level construct to express parallelism between goals. However, our intention is to support *unrestricted* and-parallelism, which has been shown capable of exploiting more of the parallelism intrinsic in programs To this end, we will use more flexible primitives :

- `G &> H` schedules the goal `G` for parallel execution and continues with the code after `G &> H`. `H` is a *handler* which contains (or *points to*) the state of `G`, and will be used for communicating the executing state between agents.
- `H <&` waits for the goal associated with `H` to finish. After `H <&` succeeds, all the bindings that `G` could possibly generate are ready. Note also that, assuming goal independence between `G` and the calls performed while `G` was being executed, no binding conflicts will arise.

With the previous definitions, the `&/2`[1] operator can be expressed as:

```
A & B :- A &> H, call(B), H <&.        (1)
```

The particular order of literals is for performance, since when running the common tail-recursive case `p:-q&p`, `p` should spawn parallel `q`'s with no delay.

Also, note that `&>/2` and `<&/1` are not intended to replace `&/2` at the language level, due to its expressiveness and conciseness, in case no extra parallelism can be exploited with them (i.e., we leave the door open to more optimized implementations of `&/2` than what the definition above suggests). The `&>/2` and `<&/1` primitives are not dependent on any particular architecture, and were in fact first implemented in a distributed-memory setting However, as the implementation we propose now addresses shared-memory multiprocessors, the bindings made by `G` while executing will be immediately visible, and goal independence makes it possible to work out a solution with the no-slowdown property.

`G &> H` ideally takes a negligible amount of time to execute, although the precise moment in which `G` actually starts depends on the availability of resources

(primarily, free agents or processors). On the other hand, `H <&` suspends until the associated goal finitely fails or returns an answer. Actual backtracking is performed at `H <&`, and the memory reserved by the handler is released when `G &> H` is reached on backtracking. If `G &> H` is reached on backtracking but `H <&` was not reached on forward execution, this means that some of the goals between these two points has failed without a solution, and the execution of goal `G` (whatever its state) is to be cancelled. Section 3 explains further the design and implementation of these operators.

3 Shared-Memory Implementation

Our shared-memory implementation for unrestricted IAP is based on the *multi-sequential, marker model* introduced by &-Prolog and adopted by many and-parallel systems, both for IAP and DAP It has some general similarities with that model, such as the concept of agent, which corresponds to a thread associated to a particular stack set, mostly a Warren Abstract Machine and the ring of stack sets which interconnects all the agents. For simplicity, each thread will be always associated to the same stack set.

However, there exist significant differences between our proposal and the &-Prolog run-time model, which we will present in the following sections.

3.1 Goal Stacks vs. Goal Lists

In our model, each agent is extended with a goal list, implemented as a doubly-linked list in C, whose functionality is similar to that of the goal stack in the &-Prolog run-time model. The goal list entries store pointers to those goals which have been prepared for parallel execution, and thus agents that are idle can search for parallel goals to execute by consulting the goal lists of the rest of the agents. A list is used instead of the traditional stack due to the greater flexibility needed in order to deal with the *unrestricted* nature of the `&>/2` and `<&/1` operators (instead of, or in addition to `&/2`): goals can be joined in any order —not necessarily the inverse to the order in which they were published— and, in the case of goal cancellation, arbitrary goal entries inside the list may have to be removed. For instance, the conjunction $(g_1\texttt{\&}g_2\texttt{\&}\ldots\texttt{\&}g_n)$ can be executed as

$$(g_1\texttt{\&>}H_1, g_2\texttt{\&>}H_2, \ldots, g_n, \ldots, H_2\texttt{<\&}, H_1\texttt{<\&})$$

as per Equation (1), but in fact any order for the joins would be equally correct.

3.2 Parcall Frames vs. Handlers

Parcall frames in the &-Prolog run-time model are additional (environment) stack frames used for the coordination and synchronization of the parallel execution. In &-Prolog a parcall frame is created as soon as a parallel call is made, and it has a slot for each of the literals $g_1, g_2 \ldots g_n$ in the parallel call $g_1\texttt{\&}g_2\texttt{\&}\ldots\texttt{\&}g_n$, in order to keep track of the execution of each of these goals.

In most WAM implementations the handling of environments is relatively brittle and introducing different elements in the environment stack complicates

things. As an alternative to parcall stack frames, our proposal makes use of heap structures, created by and accessible from source-level code that we call *handlers*, as already mentioned in Section 2. Each handler is associated to a particular parallel goal and is used to synchronize the publishing agent and the agent which picks up the goal. Handlers store information such as, e.g., pointers to the parallel goal and its location in the goal list (to remove it from there in case the goal is not taken by any other agent), a field to mark the goal as deterministic or not, the state of the execution, and pointers to both the publishing and the executing agents to release their execution when so needed.

3.3 Markers vs. (Prolog) Choice Points

Markers are used in the &-Prolog run-time model to set boundaries between different sections in the stack, each of them corresponding to the *segment* of execution of a parallel goal. This separation of segments in the stack is used to provide a solution to the *trapped goal* problem Markers are also used in &-Prolog to implement storage recovery mechanisms during backtracking of parallel goals, in order to solve the *garbage slot* problem

Our proposal to avoid the use of new stack frames to implement markers is the creation of normal choice points, and in a simple way by creating alternatives (through predicates with more than one clause) directly in the source-level code of the scheduler (see Section 3.4). This is done whenever a parallel goal is to be executed (see Figure 1(e)). In addition to that, pointers to the choice points that mark the beginning and end of the goal execution will be stored in the handler associated to that goal, in order to delimit the segment of execution and make them accessible during backwards execution. This is also done in part at the source level. Section 3.4 provides further explanation of how backwards execution over parallel goals is performed using these choice points.

3.4 Implementation

Figure 1 presents a sketch of our high-level implementation of the scheduler for unrestricted IAP. The implementation divides the responsibilities between different layers. The user-level parallelism primitives `&>/2` and `<&/1` (and thus `&/2`) are at the top of the Prolog level. The algorithms for goal publishing, goal searching, and forward and backwards execution are implemented in Prolog, with some support from low-level primitives designed to provide, e.g., locking, untrailing, and management of segments of executions. Primitives related to forward execution of parallel goals were already presented

In our implementation, agents are created with a small stack (which can grow on demand) and they wait for some work to be available. They do not continuously search for new tasks to be performed, in order to avoid active waiting.[3] Several high-level primitives are provided for the creation of a particular number

```
Goal &> Handler :-
   add_goal(Goal,nondet,Handler),
   undo(cancellation(Handler)),
   release_some_suspended_thread.
```

(a) Non-deterministic goal publishing.

```
Handler <& :-
   enter_mutex_self,
   (
      goal_available(Handler) ->
      exit_mutex_self,
      retrieve_goal(Handler,Goal),
      call(Goal)
   ;
      check_if_finished_or_failed(Handler)
   ).
Handler <& :-
   add_goal(Handler),
   release_some_suspended_thread,
   fail.
```

(b) Goal join and speculation.

```
check_if_finished_or_failed(Handler) :-
   (
      goal_finished(Handler) ->
      exit_mutex_self,
      sending_event(Handler)
   ;
      (
         goal_failed(Handler) ->
         exit_mutex_self,
         fail
      ;
         suspend,
         check_if_finished_or_failed(Handler)
      )
   ).
```

(c) Checking status of goal execution.

```
sending_event(_).
sending_event(Handler) :-
   enter_mutex_self,
   enter_mutex_remote(Handler),
   set_goal_tobacktrack(Handler),
   add_event(Handler),
   release_remote(Handler),
   exit_mutex_remote(Handler),
   check_if_finished_or_failed(Handler).
```

(d) Sending event to executing agent.

```
call_handler(Handler) :-
   retrieve_goal(Handler,Goal),
   save_init_execution(Handler),
   call(Goal),
   save_end_execution(Handler),
   enter_mutex(Handler),
   set_goal_finished(Handler),
   release(Handler),
   exit_mutex(Handler).
call_handler(Handler) :-
   enter_mutex(Handler),
   set_goal_failed(Handler),
   release(Handler),
   metacut_garbage_slots(Handler),
   exit_mutex(Handler),
   fail.
```

(e) High-level markers definition.

```
agent :-
   enter_mutex_self,
   work,
   agent.
agent :- agent.

work :-
   (
      read_event(Handler) ->
      (
         more_solutions(Handler) ->
         move_execution_top(Handler)
      ;
         move_pointers_down(Handler)
      ),
      exit_mutex_self,
      fail
   ;
      (
         find_goal(H) ->
         exit_mutex_self,
         call_handler(H)
      ;
         suspend,
         work
      )
   ).
```

(f) Agent code.

Fig. 1. High-level solution for unrestricted IAP

of agents. When an agent is created, it executes the code shown in Figure 1(f), and during normal execution it will start working on the execution of some goal, or will sleep because there is no task to perform. An agent searches for parallel goals by using a work-stealing scheduling algorithm based on those

Figure 1(a) presents the code for `&>/2`, which publishes a goal for parallel execution. A pointer to the parallel goal is added to the goal list,and a signal is sent to one of the agents that are currently waiting for some task to do. This agent will resume its execution, pick up the goal, and execute it. In addition, when

`&>/2` is reached in backwards execution, the memory reserved by the handler is released. Also, if the goal was taken by another agent and the goal execution was not finished yet, `cancellation/1` (which raises a per-agent flag which is periodically polled by every agent) asks the executing agent to abort the execution of the goal. This increases the overall performance of the system by avoiding unnecessary work, as we will show in Section 4. Moreover, in order to be able to execute this operation in the presence of cuts in the code of the clause, it is invoked via the `undo/1` predicate.

Figure 1(b) presents the implementation of `<&/1`. First, the publishing agent needs to check whether the goal was picked up by some other agent or not. If it was not taken then the publishing agent will remove it from the goal list and execute it locally (using `call/1`), and then it will continue executing scheduler code. If the goal was taken by some other agent then its status will be checked (i.e., to know whether the goal execution has already finished or failed) as shown in Figure 1(c). If the goal execution fails then the parallel goal will be added to the goal list of the publishing agent, so it can be reexecuted by some other agent. This is a form of speculative execution, since the reexecution of that literal may not be needed for the actual computation. However, it increases the actual parallelism in the system. It should be noted that the goal execution will be canceled if the corresponding `&>/2` is reached on backtracking.

If the goal execution succeeds and `<&/1` is reached on backtracking, then backwards execution needs to be performed. If the goal was not taken by some other agent then backwards execution is trivially performed. If it was picked up by some other agent then the publishing agent sends a signal to the executing agent with a request for a new solution for that goal. The executing agent will serve the signal as soon as it is able. In order to enable this communication, each agent has an *event queue* from which the agent pops events consisting of pointers to handlers associated to the goals to be backtracked over. The primitives which perform this communication are `add_event/1`, which pushes a new pointer to a handler in the event queue of the agent which executed the associated goal, and `read_event/1`, which either removes the item in the event queue to perform backwards execution over the parallel goal associated to it, or fails if the event queue is empty. Figure 1(d) presents the source code to push the corresponding event to the executing agent, releasing its execution if it was suspended.

When an agent pops an event (Figure 1(f)), backwards execution over a parallel goal needs to be performed. If the segment of execution is at the top of its stack, then the agent will invoke `fail/0` and a new solution will be obtained. However, it might be the case that the segment of execution of the parallel goal is *trapped*, i.e., it is currently not at the top of the stack. In this case, there are two possible scenarios. If the goal is known not to have additional solutions,[4] then the segment where the goal lies does not need to be expanded and the pointers to the top of the segment in the handler are simply made to point to the beginning of the segment. The trail section corresponding to that segment

?- a(X) &> Ha, b(Y) &> Hb, c(Z), Hb <&, Ha <&, fail.

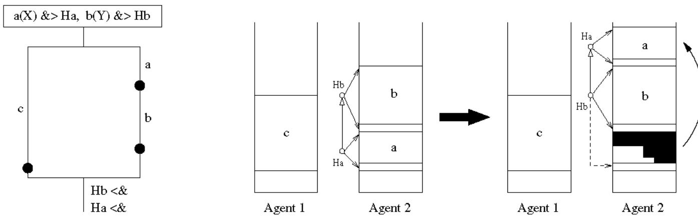

Fig. 2. Copying trapped goal onto the top of the stack

is used to undo the bindings. After this, the stack and trail pointers are restored to their previous values —i.e., they point to the top of the corresponding stacks.

If there may be more solutions for that goal, then a mechanism is needed to untrap its segment of execution. Several solutions have been proposed to solve this problem. A first approach consists of avoiding it altogether by carefully selecting goals to be executed so that they cannot cause trapped goals (which would dramatically reduce the amount of exploited parallelism). Another solution is to create a new, independent stack set for every goal taken, which would probably be memory-inefficient or impose an extra overhead in memory management. Our proposal is a variant of the solution adopted by several parallel systems (e.g., &-Prolog, ACE, DASWAM, ...), which essentially try to continue the goal execution on top of the stack. However, in our case, and for simplicity, when a trapped goal is to be backtracked over, its execution segment is *copied* on top of the stack, where it can expand freely. The garbage slot created is marked as such, and can be recovered when everything between this garbage slot and the top of the stack turns into garbage (or on backtracking). Most implementations of garbage collectors do not recover dead choice points, and thus the garbage collection algorithm needs to be changed to work with parallel execution and cross-agent pointers. Improved garbage collectors could use the pointers to boundaries of every live segment stored in the handlers.

Figure 1(e) shows how the limits of the segment of execution of the parallel goal are stored in the handler, so their values can be accessed in backwards execution, via the `save_init_execution/1` and `save_end_execution/1` primitives, which actually have similar behavior to that of the input markers and end markers in the &-Prolog model. Note that the choice point created by the predicate `call_handler/1` is in fact the input marker of the parallel execution, but again defined in the source language. Finally, when the goal execution fails, the `metacut_garbage_slots/1` primitive will pop from the stack those discarded segments of the stack that are right underneath the segment of execution.

Figure 2 shows an example of this solution for the trapped goal and garbage slot problems. We assume that variables `X`, `Y`, and `Z` are independent. When the literals `a/1` and `b/1` are taken and executed by the second agent, the pointers that define the actual segment of execution of both literals are stored in the corresponding handler. Thus, when `Ha <&` is reached in backtracking, the segment of execution

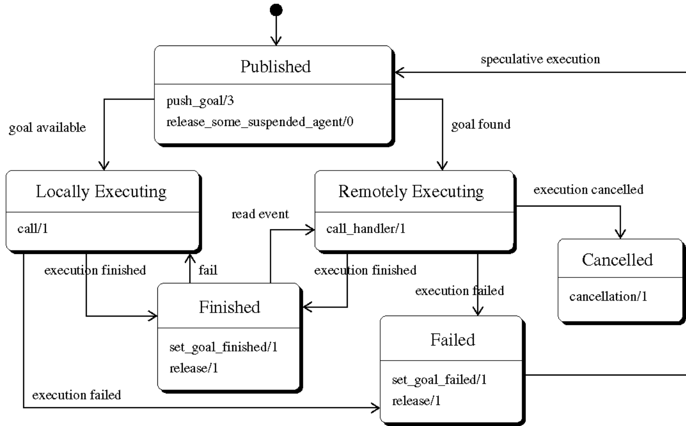

Fig. 3. State diagram of a parallel goal

of literal `a/1` is trapped, and it is copied on top of the stack in order to have enough space to expand and obtain a new solution for the goal `a/1`. The handler associated to the literal `b/1` will in addition mark the garbage slot left by the literal `a/1`, which will be freed when the execution of the literal `b/1` fails.

Figure 3 presents a diagram which shows the different states in which a parallel goal can be according to the code in Figure 1. First, a goal is published to be executed in parallel by adding a pointer to it in the goal list and releasing the execution of an agent that is currently idle. When performing the goal join, if the goal is still available it will be executed locally. If the goal was picked up by some other agent, it will be executed remotely. A goal execution can be cancelled if the outcome of the execution is not needed for the actual computation. If the goal execution is not cancelled and succeeds, it may be backtracked over with the communication between agents performed via pushing and popping events. If it fails, the goal will be published again for parallel execution.

4 Performance Evaluation

We will now present some performance results obtained with our implementation for a selection of both deterministic and non-deterministic benchmarks (see Table 1), parallelized with unrestricted independent and-parallelism. Our proposal has been implemented on the Ciao multiparadigm system All the benchmarks were automatically parallelized using CiaoPP and starting from their sequential code. The performance results were obtained by averaging ten runs on a state-of-the-art multiprocessor, a Sun Fire T2000 with 8 cores (4 threads each) and 8 Gb of memory running in 32-bit compatibility mode.

Table 2 presents the speedups obtained for some deterministic benchmarks parallelized using unrestricted IAP. The speedups were obtained with respect to the execution time of the sequential version of the benchmarks. Thus, the

Table 1. Benchmarks executed with unrestricted IAP

AIAKL	Simplified *AKL* abstract interpreter.	**MMatrix**	Matrix multip. (50×50).
Ann	Annotator for and-parallelism.	**Numbers**	Obtains a number from a list of others.
Boyer	Simplified version of *Boyer-Moore* theorem prover.	**Palindrome**	Generates a palindrome of 2^{14} elements.
Chat-80	Question parser of *Chat-80.*	**Progeom**	Constructs a perfect difference set of order n.
Deriv	Symbolic derivation.	**Queens**	The *n-queens* problem.
FFT	Fast Fourier Transform.	**QueensT**	Solves the *n-queens* problem T times.
Fibonacci	Doubly recursive *Fibonacci.*	**QuickSort**	Sorts a 10,000 element list.
Hamming	Calculates *Hamming* numbers.	**Takeuchi**	Computes *Takeuchi.*
Hanoi	Solves *Hanoi* puzzle.		

columns tagged *1* measure the *slowdown* coming from executing a parallel program in a single processor. Rows tagged with the '`&!`' symbol measure the execution of the benchmarks with some optimizations for the case of deterministic parallel goals, on our previous, determinism-only model and implementation [8]. Rows tagged with the '`&`' symbol measure the speedups obtained with all the mechanisms required by the implementation presented in Section 3. The difference in speedups between both parallel versions is of little significance in most cases, and only in very few cases (for example, `Boyer` and `Fibonacci`) the difference is relevant. Note that determinism can either be annotated by hand or, in many cases, automatically detected [4,16]. In any case, reasonably good speedups are obtained, despite the fact that the proposal suffers from the overhead added by the source-level coded scheduler etc., but which, in return, offers other advantages such as significantly reduced development (and maintenance) time, more flexibility, simpler and faster experimentation, etc.

Table 3 presents the speedups obtained for some non-deterministic benchmarks. Some of them do not obtain any speedup when executed in parallel due to the very fine granularity of the parallel goals and the high-level nature of our implementation. However, super-linear speedups can be achieved in other benchmarks (e.g., Chat-80), thanks to the implementation of goal cancellation.

A fact that limits the system performance is the expansion of the agent stack sets when running out of space. Stack sets are initially created small and they dynamically grow as needed. This fits the behavior of a naive user who lets the system run and adjust itself; a more seasoned user could create the stack sets with a size which appropriate for a particular application. Due to the work-stealing strategy adopted and the shared-memory nature of our implementation, there may be cross-agent pointers. The approach we have taken to ensure a correct stack set expansion is to suspend the execution of all the agents. The stack set which is short on space is then expanded, the pointers pointing to that stack set (from any agent) are updated, and the execution of the agents finally resumes.[5]

Table 2. Speedups obtained for deterministic unrestricted IAP benchmarks

Benchmark	Op.	Number of agents								
		Seq.	1	2	3	4	5	6	7	8
AIAKL	&!	1.00	0.99	1.82	1.82	1.82	1.83	1.83	1.83	1.82
	&	1.00	0.93	1.70	1.71	1.72	1.74	1.75	1.72	1.72
Ann	&!	1.00	0.96	1.84	2.72	3.56	4.38	5.16	5.88	6.64
	&	1.00	0.96	1.85	2.72	3.57	4.35	5.14	5.87	6.61
Boyer	&!	1.00	0.92	1.76	2.58	3.16	3.39	4.01	4.31	4.55
	&	1.00	0.90	1.21	1.83	2.06	2.26	2.30	2.39	2.56
Deriv	&!	1.00	0.83	1.59	2.38	3.07	3.78	4.49	4.98	5.49
	&	1.00	0.84	1.60	2.34	2.99	3.73	4.43	4.56	4.85
FFT	&!	1.00	0.98	1.73	2.06	2.67	2.78	2.95	2.96	3.11
	&	1.00	0.98	1.72	1.97	2.65	2.67	2.75	2.93	2.97
Fibonacci	&!	1.00	0.98	1.91	2.84	3.73	4.62	5.51	6.41	7.35
	&	1.00	0.98	1.58	2.04	2.53	3.28	4.06	4.61	5.46
Hamming	&!	1.00	0.92	1.04	1.43	1.65	1.65	1.65	1.65	1.65
	&	1.00	0.92	1.02	1.41	1.63	1.62	1.62	1.62	1.62
Hanoi	&!	1.00	0.95	1.76	2.47	3.09	3.39	3.65	3.87	4.10
	&	1.00	0.96	1.77	1.91	2.84	3.13	3.54	3.76	4.02
HanoiDL	&!	1.00	0.73	1.44	2.08	2.77	3.37	4.04	4.58	5.19
	&	1.00	0.74	1.43	1.89	1.87	2.73	3.07	3.59	3.87
MMatrix	&!	1.00	0.77	1.51	2.31	3.02	3.76	4.52	5.21	5.72
	&	1.00	0.77	1.48	2.16	2.88	3.51	4.05	4.57	4.96
Palindrome	&!	1.00	0.95	1.77	2.36	2.95	3.33	3.62	3.94	4.15
	&	1.00	0.96	1.78	2.14	2.56	3.11	3.30	3.74	3.90
QuickSort	&!	1.00	0.97	1.74	2.26	2.91	3.16	3.39	3.49	3.54
	&	1.00	0.97	1.71	2.17	2.43	2.60	2.93	3.06	3.19
QuickSortDL	&!	1.00	0.95	1.69	2.30	2.81	3.10	3.25	3.47	3.60
	&	1.00	0.95	1.68	2.14	2.39	2.56	2.92	2.94	3.19
Takeuchi	&!	1.00	0.86	1.17	2.24	2.97	3.29	3.75	4.28	5.69
	&	1.00	0.86	0.89	1.69	2.23	3.00	3.34	3.36	4.29

Table 3. Speedups obtained for non-deterministic unrestricted IAP benchmarks

Benchmark	Number of agents								
	Seq.	1	2	3	4	5	6	7	8
Chat-80	1.00	2.31	4.49	5.42	6.91	9.79	9.95	11.10	17.29
Numbers	1.00	1.84	1.79	1.79	1.79	1.79	1.79	1.78	1.78
Progeom	1.00	0.99	0.96	0.97	0.98	0.98	0.98	0.98	0.98
Queens	1.00	0.99	0.94	0.94	0.94	0.94	0.94	0.94	0.94
QueensT	1.00	0.99	1.90	2.41	3.18	4.71	4.61	4.58	4.57

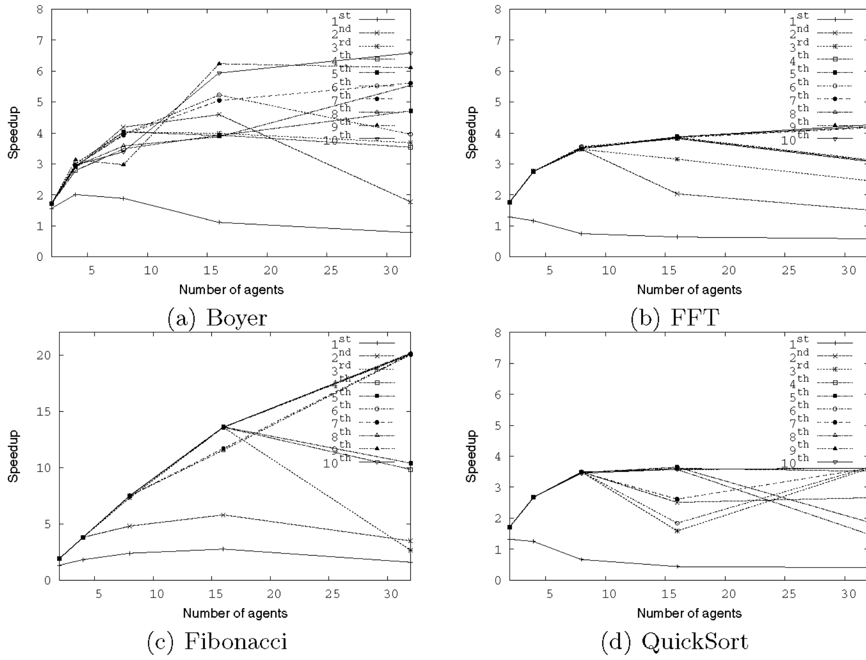

Fig. 4. Speedups for some selected benchmarks with stack set expansion

Table 4. Behavior of **Queens(8)** with different numbers of agents

		Benchmark											
		Queens, 2 agents				Queens, 4 agents				Queens, 8 agents			
		No		Gr		No		Gr		No		Gr	
		1	N	1	N	1	N	1	N	1	N	1	N
`G &> H`		11,810	171,858	9	290	11,810	171,858	9	290	11,810	171,858	9	290
Taken	$\overline{x}$	6,649	97,798	9	290	6,860	99,373	9	290	6,476	96,056	9	290
	σ	9.35	45.04	0.00	0.00	16.15	65.02	0.00	0.00	13.49	59.04	0.00	0.00
LBack	$\overline{x}$	858	14,319	0.00	0.00	618	10,905	0.00	0.00	755	12,786	0.00	0.00
	σ	1.03	1.25	0.00	0.00	14.93	99.89	0.00	0.00	5.79	23.59	0.00	0.00
RBack Top	$\overline{x}$	1,838	29,725	2	234	2,345	38,420	2	234	2,208	36,261	2	234
	σ	0.46	2.14	0.00	0.00	15.14	98.66	0.00	0.00	6.34	26.53	0.00	0.00
RBack Tp	$\overline{x}$	0	0	0	0	0	0	0	0	0	0	0	0
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

That scheme indeed affects the performance of the execution. Figure 4 presents the speedups obtained by executing ten times some selected benchmarks with 2, 4, 8, 16 and 32 agents. By joining together the points corresponding to the n-th execution with a given number of processors, we can construct a profile of how the speedup evolves as the system executes several times the same program. The first executions suffer from stack expansions but, after some runs, the stack set of each agent reaches an appropriate size, the number of expansions diminishes, and thus the performance results stabilize. Note also that, for the case of more

Table 5. Behavior of **Progeom(5)** with different numbers of agents

		Benchmark											
		Progeom, 2 agents				Progeom, 4 agents				Progeom, 8 agents			
		No		Gr		No		Gr		No		Gr	
		1	N	1	N	1	N	1	N	1	N	1	N
`G &> H`		215	154,260	1	60	215	154,260	1	60	215	154,260	1	60
Taken	$\overline{x}$	100	72,375	0	1	91	65,643	0	1	55	75,113	0	1
	σ	1.85	248.69	0.00	0.80	1.36	414.68	0.00	0.70	3.49	192.25	0.00	0.78
LBack	$\overline{x}$	1	738	0	29	3	2,131	0	29	9	364	0	29
	σ	0.46	52.03	0.00	0.80	1.10	83.78	0.00	0.70	0.80	26.82	0.00	0.78
RBack – Top	$\overline{x}$	10	6,530	0	1	8	5,131	0	1	2	6,907	0	1
	σ	0.57	52.08	0.00	0.80	1.10	84.26	0.00	0.70	0.80	27.02	0.00	0.78
RBack – Tp	$\overline{x}$	0	0	0	0	0	0	0	0	0	0	0	0
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 6. Behavior of **Fibonacci(25)** with different numbers of agents

		Benchmark											
		Fibonacci, 2 agents				Fibonacci, 4 agents				Fibonacci, 8 agents			
		No		Gr		No		Gr		No		Gr	
		1	N	1	N	1	N	1	N	1	N	1	N
`G &> H`		121,392	121,392	1,596	1,596	121,392	121,392	1,596	1,596	121,392	121,392	1,596	1,596
Taken	$\overline{x}$	1	1	1	1	5	5	5	5	37	37	31	31
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.97	3.97	2.39	2.39
LBack	$\overline{x}$	121,391	121,391	1,595	1,595	121,387	121,387	1,591	1,591	121,355	121,355	1,565	1,565
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.97	3.97	2.39	2.39
RBack – Top	$\overline{x}$	1	1	1	1	5	5	5	5	18	18	16	16
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.40	2.40	0.98	0.98
RBack – Tp	$\overline{x}$	0	0	0	0	0	0	0	0	19	19	15	15
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.86	2.86	1.68	1.68

than 8 agents, the limitations in the hardware of the multiprocessor machine[6] used also affect the actual performance of the execution.

Tables 4 to 6 present data from the execution of some of the non-deterministic, and-parallel benchmarks. They present data from executions with 2, 4, and 8 agents, using or not granularity control (resp., `Gr` and `No`), and in cases where only one solution (`1`) or all solutions (`N`) are requested. The first row in the table (`G &> H`) contains the number of parallel goals. The second row (*Taken*) presents the number of parallel goals picked up by some other agent ($\overline{x}$ stands for the average and σ for the standard deviation in ten runs). The third row (*LBack*) represents the number of times that backtracking over parallel goals took place locally because the goal was not picked up by some other agent.[7] The fourth row (*RBack*) shows the number of times a parallel goal was backtracked over remotely. *Top* and *Tp* count, respectively, how many times remote backtracking was performed at the top of the stack and on a trapped goal. A conclusion from these results is that, while the amount of remote backtracking is quite high, the number of trapped goals is low. Therefore the overhead of copying trapped

segments to the top of the stack should not be very high in comparison with the rest of the execution.

We expect to see a similar behavior in most non-deterministic parallel programs where parallel goals are of fine granularity or very likely to fail: these two behaviors make the piling up of segments corresponding to the execution of loosely related parallel goals in the same stack relatively uncommon, which indeed reduces the chances to suffer from trapped goal and garbage slot problems.

5 Conclusions

We have presented a high-level implementation of unrestricted, independent and-parallelism that can execute both deterministic and non-deterministic programs in parallel. The approach helps taming the implementation complexity of previous solutions by raising many of the main implementation components to the source level. This makes the system easier to code, maintain, and expand. Our evaluation of actual parallel executions shows that quite useful speedups can be obtained with the approach, including for benchmarks which perform backtracking over non-deterministic parallel goals In several cases, super-linear speedups were obtained thanks to the backtracking model implemented.

We believe that the results obtainable with this approach will improve further as the speed of the source language continues to increase. Recent compilation technology and implementation advances provide hope that it will eventually be possible to recover most of the efficiency lost due to expressing the parallel machinery using the high-level language. In the meantime, performance can also be improved by, once the components of the system are stabilized, selectively lowering again the implementation of those flagged as bottlenecks, if the benefits surpass the added complexity and reduced flexibility. Performance can also be improved, e.g., by exploiting the fact that smarter schedulers are, in principle, easier to write than with other approaches.

References

Ait-Kaci, H.: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press, Cambridge (1991)

Ali, K.A.M., Karlsson, R.: The Muse Or-Parallel Prolog Model and its Performance. In: 1990 North American Conference on Logic Programming, pp. 757–776. MIT Press, Cambridge (1990)

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G. (eds.): The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School, UPM (2006), `http://www.ciaohome.org`

Bueno, F., López-García, P., Puebla, G., Hermenegildo, M.: A Tutorial on Program Development and Optimization using the Ciao Preprocessor. Technical Report CLIP2/06, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain (January 2006)

Cabeza, D., Hermenegildo, M.: Implementing Distributed Concurrent Constraint Execution in the CIAO System. In: Proc. of the AGP 1996 Joint Conference on Declarative Programming, pp. 67–78 (July 1996)

Carro, M., Hermenegildo, M.: Concurrency in Prolog Using Threads and a Shared Database. In: 1999 International Conference on Logic Programming, pp. 320–334. MIT Press, Cambridge (November 1999)
Casas, A., Carro, M., Hermenegildo, M.: Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 138–153. Springer, Heidelberg (2008)
Casas, A., Carro, M., Hermenegildo, M.: Towards a High-Level Implementation of Execution Primitives for Non-restricted, Independent And-parallelism. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 230–247. Springer, Heidelberg (2008)
Conery, J.S.: The And/Or Process Model for Parallel Interpretation of Logic Programs. Ph.D thesis, The University of California At Irvine, Technical Report 204 (1983)
Gupta, G., Pontelli, E., Ali, K., Carlsson, M., Hermenegildo, M.: Parallel Execution of Prolog Programs: a Survey. ACM Transactions on Programming Languages and Systems 23(4), 472–602 (2001)
Hermenegildo, M.: An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In: Shapiro, E. (ed.) ICLP 1986. LNCS, vol. 225, pp. 25–40. Springer, Heidelberg (1986)
Hermenegildo, M.: Parallelizing Irregular and Pointer-Based Computations Automatically: Perspectives from Logic and Constraint Programming. Parallel Computing 26(13–14), 1685–1708 (2000)
Hermenegildo, M., Carro, M.: Relating Data–Parallelism and (And–) Parallelism in Logic Programs. The Computer Languages Journal 22(2/3), 143–163 (1996)
Hermenegildo, M., Greene, K.: The &-Prolog System: Exploiting Independent And-Parallelism. New Generation Computing 9(3,4), 233–257 (1991)
Hermenegildo, M., Nasr, R.I.: Efficient Management of Backtracking in AND-parallelism. In: Shapiro, E. (ed.) ICLP 1986. LNCS, vol. 225, pp. 40–55. Springer, Heidelberg (1986)
Hermenegildo, M., Puebla, G., Bueno, F., López García, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming 58(1–2), 115–140 (2005)
Janson, S.: AKL. A Multiparadigm Programming Language. Ph.D thesis, Uppsala University (1994)
López-García, P., Hermenegildo, M., Debray, S.K.: A Methodology for Granularity Based Control of Parallelism in Logic Programs. J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation 21, 715–734 (1996)
Lusk, E., Butler, R., Disz, T., Olson, R., Stevens, R., Warren, D.H.D., Calderwood, A., Szeredi, P., Brand, P., Carlsson, M., Ciepielewski, A., Hausman, B., Haridi, S.: The Aurora Or-parallel Prolog System. New Generation Computing 7(2/3), 243–271 (1988)
Moura, P., Crocker, P., Nunes, P.: High-level multi-threading programming in logtalk. In: Warren, D.S., Hudak, P. (eds.) PADL 2008. LNCS, vol. 4902, pp. 265–281. Springer, Heidelberg (2008)
Muthukumar, K., Bueno, F., García de la Banda, M., Hermenegildo, M.: Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. Journal of Logic Programming 38(2), 165–218 (1999)

Pontelli, E., Gupta, G.: Efficient Backtracking in And-Parallel Implementations of Non-Deterministic Languages. In: Lai, T. (ed.) Proc. of the International Conference on Parallel Processing, pp. 338–345. IEEE Computer Society, Los Alamitos (1998)
Pontelli, E., Gupta, G., Hermenegildo, M.: &ACE: A High-Performance Parallel Prolog System. In: International Parallel Processing Symposium, pp. 564–572. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society (April 1995)
de Morais Santos-Costa, V.M.: Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I. Ph.D thesis, University of Bristol (August. 1993)
Shen, K.: Overview of DASWAM: Exploitation of Dependent And-parallelism. Journal of Logic Programming 29(1–3), 245–293 (1996)
Warren, D.H.D.: An Abstract Prolog Instruction Set. TR 309, SRI International (1983)