Towards a High-Level Implementation of Execution Primitives for Unrestricted, Independent And-Parallelism

Amadeo Casas[1] Manuel Carro[2] Manuel V. Hermenegildo[1,2]
{amadeo, herme}@cs.unm.edu {mcarro, herme}@fi.upm.es

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

Abstract. Most efficient implementations of parallel logic programming rely on complex low-level machinery which is arguably difficult to implement and modify. We explore an alternative approach aimed at taming that complexity by raising core parts of the implementation to the source language level for the particular case of and-parallelism. We handle a significant portion of the parallel implementation at the Prolog level with the help of a comparatively small number of concurrency-related primitives which take care of lower-level tasks such as locking, thread management, stack set management, etc. The approach does not eliminate altogether modifications to the abstract machine, but it does greatly simplify them and it also facilitates experimenting with different alternatives. We show how this approach allows implementing both restricted and unrestricted (i.e., non fork-join) parallelism. Preliminary experiments show that the performance sacrificed is reasonable, although granularity control is required in some cases. Also, we observe that the availability of unrestricted parallelism contributes to better observed speedups.

Keywords: Parallelism, Virtual Machines, High-level Implementation.

1 Introduction

The wide availability of multicore processors is finally making parallel computers mainstream, thus bringing a renewed interest in languages and tools to simplify the task of writing parallel programs. The use of declarative paradigms and, among them, logic programming, is considered an interesting approach for obtaining increased performance through parallel execution on multicore architectures, including multicore embedded systems. The high-level nature of these languages allows coding in a style that is closer to the application and thus preserves more of the original parallelism for automatic parallelizers to uncover. Their amenability to semantics-preserving automatic parallelization is also due, in addition to this high level of abstraction, to their relatively simple semantics, and the separation between the control component and the declarative specification. This makes it possible for the evaluator to execute some operations in any order (including in parallel), without affecting the meaning of the program. In addition, logic variables can be assigned a value at most once, and

thus it is not necessary to check for some types of flow dependencies or to perform single statement assignment (SSA) transformations, as done with imperative languages. At the same time, the presence of dynamic data structures with "declarative pointers" (logical variables), irregular computations, or complex control makes the parallelization of logic programs a particularly interesting case that allows tackling complex parallelization-related challenges in a formally simple and well-understood context [14].

Parallel execution of logic programs has received considerable attention and very significant progress has been made in the area (see, e.g., [11] and its references). Two main forms of parallelism have been exploited: *Or-parallelism* (Aurora [22] and MUSE [2]) parallelizes the execution of different clauses of a predicate (and their continuations) and is naturally applicable to programs which perform search. *And-parallelism* refers to the parallel execution of different goals in the resolvent. It arises naturally in different kinds of applications (independently of whether there is implicit search or not), such as, e.g., divide-and-conquer algorithms. Systems like &-Prolog [16], DDAS [27] and others have exploited and-parallelism, while certain combinations of both and- and or-parallelism have been exploited by e.g. &ACE [24], AKL [20], and Andorra [26].

The basic ideas of the &-Prolog model have been adopted by many other systems (e.g., &ACE and DDAS). It consists of two components: a parallelizing compiler which detects the possible runtime dependencies between goals in clause bodies and annotates the clauses with expressions to decide whether parallel execution can be allowed at runtime, and a run-time system that exploits that parallelism. The run-time system is based on an extension of the original WAM architecture and instruction set, and was originally implemented, as most of the other systems mentioned, on shared-memory multiprocessors, although distributed implementations were also taken into account. We will follow the same overall architecture and assumptions herein, and concentrate as well on (modern) shared-memory, multicore processors.

These models and their implementations 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 makes implementing and maintaining these systems inherently hard. In this paper we explore an alternative approach that is based on raising some components to the source language level and keeping at low level only selected operations related to, e.g., thread handling and locking. We expect of course a performance impact, but hope that this division of concerns will make it possible to more easily explore variations on the execution schemes. While doing this, another objective of our proposal is to be able to easily exploit unrestricted and-parallelism, i.e., parallelism that is not restricted to fork-join operations.

2 Classical Approaches to And-Parallelism

In goal-level and-parallelism, a key issue is which goals to select for parallel execution in order to avoid situations which lead to incorrect execution or slowdown [19, 14]. Not only errors but also significant inefficiency can arise from the

simultaneous execution of computations which depend on each other since, for example, this may trigger more backtracking than in the sequential case. Thus, goals are said to be independent if their parallel execution will not perform additional search and will not produce incorrect results. Very general notions of independence have been developed, based on constraint theory [10]. However for simplicity we discuss only those based on variable sharing.

In *Dependent and-parallelism (DAP)* goals are executed in parallel even if they share variables, and the competition to bind them has to be dynamically dealt with using notions such as sequencing bindings from producers to consumers. Unfortunately this usually implies substantial execution overhead. In *Strict Independent and-parallelism (SIAP)* goals are allowed to execute in parallel only when they do not share variables, which guarantees the correctness and no-slowdown. *Non-strict independent and-parallelism (NSIAP)* is a significant extension, also guaranteeing the no-slowdown property, in which goals are parallelized even if they share variables, provided that at most one goal binds a shared variable or the goals agree in the possible bindings for shared variables. Compile-time tools have been devised and implemented to statically detect cases where this holds, thus making the runtime machinery lighter and faster. Undetermined cases can, if deemed advantageous, be checked at runtime.

Another issue is whether any restrictions are posed on the patterns of parallelization. For example, *Restricted and-parallelism (RAP)* constrains parallelism to (nested) fork-join operations. In the &-Prolog implementation of this model conjunctions which are to be executed in parallel are often marked by replacing the sequential comma (`,/2`) with a parallelism operator (`&/2`).

In this paper we will focus on the implementation of IAP and NSIAP parallelism, as both have practically identical implementation requirements. Our objective is to exploit both restricted and unrestricted, goal-level and-parallelism.

Once a method has been devised for selecting goals for parallel execution, an obviously relevant issue is how to actually implement such parallel execution. One usual implementation approach used in many and-parallel systems (both for IAP [16, 24] and for DAP [27]) is the *multi-sequential, marker model* introduced by &-Prolog [13]. In this model parallel goals are executed in different *abstract machines* which run in parallel. In order to preserve sequential speed, these abstract machines are extensions of the sequential model, usually the Warren Abstract Machine (WAM) [29, 1], which is the basis of most efficient sequential implementations. Herein we assume for simplicity that each (P)WAM has a parallel thread (an "agent") attached and that we have as many threads as processors. Thus, we can refer interchangeably to WAMs, agents, or processors. Within each WAM, sequential fragments appear in contiguous stack sections exactly as in the sequential execution.[3] The new data areas are [16]:

Goal List: A shared area onto which goals that are ready to execute in parallel are pushed. WAMs can pick up goals from other WAMs' (or their own) goal lists. Goal list entries include a pointer to the environment where the

[3] This can actually be relaxed: *continuation markers* [28] allow sequential execution to spread over non-contiguous sections. We will not deal with this issue here.

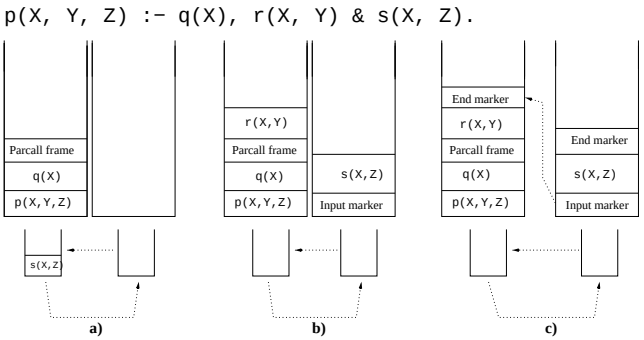


Fig. 1. Sketch of data structures layout using the marker model.

goal was generated and to the code starting the goal execution, plus some additional control information.

Parcall Frames: They are created for each parallel conjunction and hold the necessary data for coordinating and synchronizing the parallel execution of the goals in the parallel conjunction.

Markers: They separate stack sections corresponding to different parallel goals. When a goal is picked up by an agent, an *input marker* is pushed onto the choicepoint stack. Likewise, an *end marker* is pushed when a goal execution ends. These are linked to ensure that backtracking will happen following a logical (i.e., not physical) order.

Figure 1 sketches a possible stack layout for a program such as:

```
p(X, Y, Z) :- q(X), r(X, Y) & s(X, Z).
```

with query `p(X, Y, Z)`. We assume that `X` will be ground after calling `q/1`. Different snapshots of the stack configurations are shown from left to right. Note that in the figure we are intermixing parcall frames and markers in the same stack. Some actual implementations have chosen to place them in different parts of the available data areas.[4]

When the first WAM executes the parallel conjunction `r(X, Y) & s(X, Z)`, it pushes a parcall frame onto its stack and a goal descriptor onto its goal stack for the goal `s(X, Z)` (i.e., a pointer to the WAM code that will construct this call in the argument registers and another pointer to the appropriate environment), and it immediately starts executing `r(X, Y)`. A second WAM, which is looking for jobs, picks `s(X, Z)` up, pushes an input marker into its stack (which references the parcall frame, where data common to all the goals is stored, to be used in case of *internal failure*) and constructs and starts executing the goal. An end marker is pushed upon completion. When the last WAM finishes, it will link the markers (so as to proceed adequately on backtracking and unwinding), and execution will proceed with the continuation of `p/3`.

Classical implementations using the marker model handle the `&/2` operator at the abstract machine level: the compiler issues specific WAM instructions for `&/2`,

[4] For example, in &ACE parcall frames are pushed onto a separate stack and their slots are allocated in the heap, to simplify memory management.

which are executed by a modified WAM implementation. These modifications are far from trivial, although relatively isolated (e.g., unification instructions are usually not changed, or changed in a generic, uniform way).

As mentioned in the introduction, one of our objectives is to explore an alternative implementation approach based on raising components to the source language level and keeping at low level only selected operations. Also, we would like to avoid modifications to the low-level compiler. At the same time, we want to be able to easily exploit unrestricted and-parallelism, i.e., parallelism that is not restricted to fork-join operations. These two objectives are actually related in our approach because, as we will see in the following section, we will start by decomposing the parallelism operators into lower-level components which will also allow supporting unrestricted and-parallelism.

3 Decomposing And-Parallelism

It has already been reported [6, 5] that it is possible to construct the and-parallel operator `&/2` using more basic yet meaningful components. In particular, it is possible to implement the semantics of `&/2` using two end-user operators, `&>/2` and `<&/1`, defined as follows:[5]

- `G &> H` schedules 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 goal `G`.
- `H <&` waits for the goal associated with `H` (`G`, in the previous item) to finish. At that point all bindings `G` could possibly generate are ready, since `G` has reached a solution. Assuming goal independence between `G` and the calls performed while `G` was being executed, no binding conflicts will arise.

`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/processors). On the other hand, `H <&` suspends until the associated goal finitely fails or returns an answer. It is interesting to note that the approach shares some similarities with the concept of *futures* in parallel functional languages. A future is meant to hold the return value of a function so that a consumer can wait for its complete evaluation. However, the notions of "return value" and "complete evaluation" do not make sense when logic variables are present. Instead, `H <&` waits for the moment when the producer goal has completed execution, and the "received values" (a tuple, really) will be whatever (possibly partial) instantiations have been produced by such goal.

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

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

[5] We concentrate on forward execution here. See Section 4.5 for backtracking behavior. Also, although exception handling is beyond our current scope, exceptions uncaught by a parallel goal surface at the corresponding `<&/1`, where they can be captured.

(Actual implementations will of course expand `A & B` at compile time using the above definition in order not to pay the price of an additional call and the meta-call. The same can be applied to `&>` and `<&`.) However, these two new operators can additionally be used to exploit more and-parallelism than is possible with `&/2` alone [9]. We will just provide some intuition by means of a simple example (an experimental performance evaluation is included in Section 5.)[6]
Consider predicate `p/3` defined as follows:

```
p(X,Y,Z) :- a(X,Z), b(X), c(Y), d(Y,Z).
```

whose (strict) dependencies (assuming that `X,Y,Z` are free and do not share on entry) are shown in Figure 2. A classical fork-join parallelization is shown in Figure 3, while an alternative (non fork-join) parallelization using the new operators is shown in Figure 4. We assume here that solution order is not relevant.

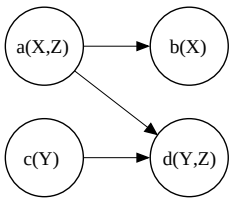


Fig. 2. Dep. graph for `p/3`.

It is obvious that it is always possible to parallelize programs using `&>/2` and `<&/1` and obtain the same parallelism as with `&/2` (since `&/2` can be defined in terms of `&>/2` and `<&/1`). The converse is not true. Furthermore, there are cases (as in Figure 4) where the parallelizations allowed by `&>/2` and `<&/1` can be expected to result in shorter execution times, for certain goal execution times [9]. In our example, the annotation in Figure 3 misses the possible parallelism between the subgoals `c/1` and `b/1`, which the code in Figure 4 allows: `c/1` is scheduled at the beginning of the execution, and it is waited for in `Hc <&`, just after `b/1` has been scheduled for parallel execution.

In addition to `&>/2` and `<&/1`, we propose specialized versions in order to obtain additional functionality or more efficiency. In particular, `&!>/2` and `<&!/1` are intended to be equivalent to `&>/2` and `<&/1`, respectively, but only for single-solution, non-failing goals, where there is no need to anticipate backtracking during forward execution. These primitives allow the parallelizer to flag goals that analysis has detected to be deterministic and non-failing (see [18]), and this can result in important simplifications in the implementation.

4 Sketch of a Shared Memory Implementation

Our proposed implementation divides responsibilities among several layers. User-level parallelism and concurrency primitives intended for the programmer and parallelizers are at the top and written in Prolog. Below, goal publishing, searching for available goals, and goal scheduling are written at the Prolog level, relying on some low-level support primitives for, e.g., locking or low-level goal management, with a Prolog interface but written in C.

[6] Note that the `&>/2` and `<&/1` operators do not replace the fork-join operator `&/2` at the language level due to its conciseness in cases in which no extra parallelism can be exploited with `&>/2` and `<&/1`.

```
p(X, Y, Z):-
        a(X, Z) & c(Y),
        b(X) & d(Y, Z).
```

Fig. 3. Nested fork-join annotation.

```
p(X, Y, Z) :-
        c(Y) &> Hc,
        a(X, Z),
        b(X) &> Hb,
        Hc <&,
        d(Y, Z),
        Hb <&.
```

Fig. 4. Using the new operators.

In our current implementation for shared-memory multiprocessors, and similarly to [16], agents wait for work to be available, and execute it if so. Every agent is created as a thread attached to an (extended) WAM stack set. Sequential execution proceeds as usual, and coordination with the rest of the agents is performed by means of shared data structures. Agents make new work available to other agents (and also to itself) through a *goal list* which is associated with every stack set and which can be consulted by all the agents. This is an instance of the general class of *work-stealing* scheduling algorithms, which date back at least to Burton and Sleep's [4] research on parallel execution of functional programs and Halstead's [12] implementation of Multilisp, and the original &-Prolog abstract machine [13, 16], for logic programs.

In the following subsections we will introduce the library with the (deterministic) low-level parallelism primitives and we will present the design (and a sketch of the actual code, simplified for space reasons) of the main source-level algorithms used to run deterministic, non-failing goals in parallel. We will conclude with some comments on the execution of nondeterministic goals in parallel.

4.1 Low-Level Parallelism Primitives

The low-level layer has been implemented as a Ciao library ("`apll`") written in C which provides basic mechanisms to start threads, wait for their completion, push goals, search for goals, access to O.S. locks, etc. Most of these primitives need to refer to an explicit goal and need to use some information related to its state (whether it has been taken, finished, etc.). Hence the need to pass them a `Handler` data structure which abstracts information related to the goal at hand.

The current (simplified) list of primitives follows. Note that this is not intended to be a general-purpose concurrency library (such as those available in Ciao and other Prolog systems —in fact, very little of what should appear in such a generic library is here), but a list of primitives suitable for efficiently implementing at a higher-level different approaches to exploiting independent and-parallelism. We are, for clarity, adding explicitly the library qualification.

`apll:push_goal(+Goal,+Det,-Handler)` atomically creates a unique *handler* (an opaque structure) associated to `Goal` and publishes `Goal` in the goal list for any agent to pick it up. `Handler` will henceforth be used in any operation related to `Goal`. `Det` describes whether `Goal` is deterministic or not.

`apll:find_goal(-Handler)` searches for a goal published in some goal list. If one exists, `Handler` is unified with a handler for it; the call fails otherwise, and

it will succeed at most once per call. Goal lists are accessed atomically so as to avoid races when updating them.[7]

`apll:goal_available(+Handler)` succeeds if the goal associated to `Handler` has not been picked up yet, and fails otherwise.

`apll:retrieve_goal(+Handler,-Goal)` unifies `Goal` and the goal initially associated to `Handler`.

`apll:goal_finished(+Handler)` succeeds if the execution state of the goal associated to `Handler` is finished, and fails otherwise.

`apll:set_goal_finished(+Handler)` sets to finished the execution state of the goal associated to `Handler`.

`apll:waiting(+Handler)` succeeds when the execution state of the agent which published the goal associated to `Handler` is suspended and fails otherwise.

Additionally, a set of locking primitives is provided to perform synchronization and to obtain mutual exclusion at the Prolog level. Agents are synchronized by using two different locks:[8] one which is used to ensure mutual exclusion when dealing with shared data structures (i.e., when adding new goals to the list), and another one which is used to synchronize the agent waking up when `<&/1` is waiting for either more work to be available, or the execution of a goal picked up by some other agent to finish. Both can be accessed with specific (`*_self`) predicates to specify the ones belonging to the calling agent. Otherwise, they are accessed through a goal `Handler`, and then the locks accessed are those belonging to the agent which created the goal that `Handler` refers to (i.e., its *creator*).

`apll:suspend` suspends the execution of the calling thread.

`apll:release(+Handler)` releases the agent which created `Handler` (which could have suspended itself with the above described predicate).

`apll:release_some_suspended_thread` selects one out of any suspended threads and resumes its execution.

`apll:enter_mutex(+Handler)` attempts to enter mutual exclusion by using the lock of the agent associated to `Handler`, in order to access its shared variables.

`apll:enter_mutex_self` same as above, with the agent's own mutex.

`apll:exit_mutex(+Handler)` *signals* the lock in the realm of the agent associated to `Handler` in order to exit mutual exclusion.

`apll:exit_mutex_self` same as above with the calling thread.

The following sections will clarify how these primitives are intended to be used.

4.2 High-level Goal Publishing

Based on the previous low-level primitives, we will develop the user-level ones. We will describe a particular strategy (which is the one used in our experiments) in which idle agents are suspended and resumed depending on the availability of work, instead of continuously looking for tasks to perform.

[7] Different versions exist of this primitive which can be used while implementing different goal scheduling strategies.

[8] Note that both locks are local to the thread, i.e., they are not global locks.

```
H <&! :-
    apll:enter_mutex_self,
    (
        apll:goal_available(H) ->
        apll:retrieve_goal(H,Goal),
        apll:exit_mutex_self,
        call(Goal)
    ;
        apll:exit_mutex_self,
        perform_other_work(H)
    ).
```

```
perform_other_work(H) :-
    apll:enter_mutex_self,
    (
        apll:goal_finished(H) ->
        apll:exit_mutex_self
    ;
        find_goal_and_execute,
        perform_other_work(H)
    ).
```

Fig. 6. Goal join with continuation.

A call to `&!>/2` (or `&>/2` if the goal is nondeterministic) publishes the goal in the goal list managed by the agent, which makes it available to other agents. Figure 5 shows the (simplified) Prolog code implementing this functionality (again, the code shown can be expanded in line but is shown as a meta-call for clarity). First, a pointer to the goal generated is inserted in the goal list, and then a signal is broadcast to let suspended agents know that new work is available. As we will see later, the agent receiving the signal will resume its execution, pick up the new parallel goal, and start its execution.

```
Goal &!> Handler :-
    apll:push_goal(Goal,det,Handler),
    apll:release_some_suspended_thread.
```

Fig. 5. Publishing a (deterministic) parallel goal.

After executing `Goal &!> H`, `H` will hold the state of `Goal`, which can be inspected both by the thread which publishes `Goal` and by any thread which picks up `Goal` to execute it. Therefore, in some sense, `H` takes the role of the *parcall frame* in [16], but it goes to the heap instead of being placed in the environment. Threads can communicate and synchronize through `H` in order to consult and update the state of `Goal`. This is especially important when executing `H <&!`.

4.3 Performing Goal Joins

Figure 6 provides code implementing `<&!/1` (the deterministic version of `<&/1`). First, the thread needs to check whether the goal has been picked up by some other agent, using `apll:goal_available/1`. If this is not the case, then the publishing agent executes it locally, and `<&!/1` succeeds trivially. Note that mutual exclusion is requested with `apll:enter_mutex_self/0` in order to avoid incorrect concurrent accesses to (shared) data structures related to goal management.

If the goal has been picked up by another agent and its execution has finished, then `<&!/1` will automatically succeed (note that mutual exclusion is entered again in order to safely check the goal status). In that case, the bindings made during goal execution are, naturally, available, since we are dealing with a shared-memory implementation. If the goal execution has not finished yet then the thread will search for more work in order to keep itself busy, and it will only suspend if there is definitely no work to perform at the moment. This ensures that overall efficiency is kept at a reasonable level, as we will see in Section 5.

```
find_goal_and_execute :-
    apll:find_goal(Handler),
    apll:exit_mutex_self,
    apll:retrieve_goal(Handler,Goal),
    call(Goal),
    apll:enter_mutex(Handler),
    apll:set_goal_finished(Handler),
    (
        apll:waiting(Handler) ->
        apll:release(Handler)
    ;
        true
    ),
    apll:exit_mutex(Handler).
find_goal_and_execute :-
    apll:exit_mutex_self,
    apll:suspend.
```

Fig. 7. Finding a parallel goal and executing it.

```
create_agents(0) :- !.
create_agents(N) :-
    N > 0,
    conc:start_thread(agent),
    N1 is N - 1,
    create_agents(N1).



agent :-
    apll:enter_mutex_self,
    find_goal_and_execute,
    agent.
```

Fig. 8. Creating parallel agents.

We want to note, again, that this process is protected from races when accessing shared variables by using locks for mutual exclusion and synchronization.

Figure 7 shows the code for `find_goal_and_execute/0`, which searches for work in the system. If a goal is found, the executing thread will retrieve and execute it, ensure mutual exclusion on the publishing agent data structures (where the handler associated to the goal resides), mark the goal execution as finished and resume the execution of the publishing agent, if it was suspended. In that case, the publishing agent (suspended in `eng_suspend/0`) will check which situation applies after resumption and act accordingly after recursively invoking the predicate `perform_other_work/1`. If no goal was available for execution, `find_goal_and_execute/0` will suspend waiting for more work to be created.

4.4 Agent Creation

Agents are generated using the `create_agents/1` predicate (Figure 8) which launches a number of O.S. threads using the `start_thread/0` predicate imported from a generic concurrency library (thus the `conc` prefix used, again, for clarity). Each of these threads executes the `agent/0` code, which continuously either executes work in the system and looks for more work when finished, or sleeps when there is nothing to execute. We assume for simplicity that agent creation is performed at system startup or just before starting a parallel execution. Higher-level predicates are however provided in order to manage threads in a more flexible way. For instance, `ensure_agents/1` makes sure that a given number of executing agents is available. In fact, agents can be created lazily, and added or deleted dynamically as needed, depending on machine load. However, this interesting issue of thread throttling is beyond the scope of this paper.

4.5 Towards Non-determinism

For simplicity we have left out of the discussion and also of the code the support for backtracking, which clearly complicates things. We have made significant progress in our implementation towards supporting backtracking so that, for example, the failure-driven top level is used unchanged and memory is recovered orderly at the end of parallel executions. However, completing the implementation of backtracking is still the matter of current work.

There are interesting issues both at the design level and also at the implementation level. An interesting point at the design level is for example deciding whether backtracking happens when going over `&>/2` or `<&/1` during backward execution. Previous work [6, 5] leaned towards the latter, which is also probably easier to implement; however, there are also reasons to believe that the former may in the end be more appropriate. For example, in parallelized loops such as:

```
p([X|Xs]):- b(X) &> Hb, p(Xs), Hb <&.
```

spawning `b(X)` and keeping the recursion local and not the other way around is important because task creation is the real bottleneck. However, the solution order is not preserved if backtracking occurs at `<&/1`, but it is if backtracking occurs at `&>/2`. Note that in such loops the loss of last call optimization (LCO) is only of relative importance, since if there are several solutions to either `b/1` or `p/1`, LCO could not be applied anyway, and a simple program transformation (to store handlers in an accumulating parameter) can recover it if necessary.

At the implementation level, avoiding the "trapped goal" and "garbage slots" problems [17] is an issue to solve. One approach under consideration to this end is to move trapped stack segments (sequential sections of execution) to the top of the stack set in case backtracking is needed from a trapped section. Sections which become empty can be later compacted to avoid garbage slots. In order to express this at the Prolog level, we foresee the need of additional primitives, still the subject of further work, to manage stack segments as first-class citizens.

Another fundamental idea in the approach that we are exploring is not to create markers explicitly, but use instead, for the same purpose, standard choice points built by creating alternatives (using alternative clauses) directly in the control code (in Prolog) that implements backtracking.

5 Experimental Results

We now present performance results obtained after executing a selection of well-known benchmarks with independent and-parallelism. As mentioned before, we have implemented the proposed approach in Ciao [3], an efficient system designed with extension capabilities in mind. All results were obtained by averaging ten runs on a state-of-the-art multiprocessor, a Sun Fire T2000 with 8 cores and 8 Gb of memory. While each core is capable of running 4 threads in parallel, and in theory up to 32 threads could run simultaneously on this machine, we only show speedups up to 8 agents. Our experiments (see the later comments related to Figure 10) show that speedups with more than 8 threads stop being linear even

AIAKL	Simplified *AKL* abstract interpreter.	**Hamming**	Calculates *Hamming* numbers.
Ann	Annotator for and-parallelism.	**Hanoi**	Solves *Hanoi* puzzle.
Boyer	Simplified version of the *Boyer-Moore* theorem prover.	**MergeSort**	Sorts a 10000 element list.
		MMatrix	Multiplies two 50×50 matrices.
Deriv	Symbolic derivation.	**Palindrome**	Generates a palindrome of 2^{14} elements.
FFT	Fast Fourier transform.	**QuickSort**	Sorts a 10000 element list.
Fibonacci	Doubly recursive *Fibonacci*.	**Takeuchi**	Computes *Takeuchi*.
FibFun	Functional *Fibonacci*.	**WMS2**	A work scheduling program.

Table 1. Benchmarks for restricted and unrestricted IAP.

for completely independent computations (i.e., 32 totally independent threads do not really speed up as if 32 independent processors were available), as threads in the same core compete for shared resources such as integer pipelines. Thus, beyond 8 agents, it is hard to know whether reduced speedups are due to our parallelization and implementation or to limitations of the machine.

Although most of the benchmarks we use are quite well-known, Table 1 provides a brief description. Speedups appear in Tables 2 (which contains only programs parallelized using restricted [N]SIAP, as in Figure 3) and 3 (which additionally contains unrestricted IAP programs, as in Figure 4). The speedups are with respect to the sequential speed on one processor of the original, unparallelized benchmark. Therefore, the columns tagged *1* correspond to the slowdown coming from executing a parallel program in a single processor. Benchmarks with a *GC* suffix were executed with granularity control with a suitably chosen threshold and benchmarks with a *DL* suffix use difference lists and require NSIAP for parallelization. All the benchmarks in the tables were automatically parallelized using CiaoPP [18] and the annotation algorithms described in [9] (*TakeuchiGC* needed however some unfolding in order to uncover and allow exploiting more parallelism using the new operators, as discussed later).

It can be deduced from the results that in several benchmarks the *natural* parallelizations produce small granularity. This, understandably, impacts our implementation since a sizable part of it is written in Prolog, which implies additional overhead in the preparation and execution of parallel goals. Thus, it is not possible to perform a fair comparison of the speedups obtained with respect to previous (lower-level) and-parallel systems. The overhead implied by the proposed approach produces comparatively low performance on a single processor and in some cases with very fine granularity, such as Boyer and Takeuchi, speedups are shallow (below 2×) even over 8 processors. In these examples execution is dominated by the sequential code of the scheduler and agent management in Prolog. However, even in these cases, setting a granularity threshold based on a measure of the input argument size [21] much better results can be obtained. Figure 11 depicts graphically the impact of granularity control in some benchmarks. Annotating the parallelized program to take into account granularity measures based on the size of the input arguments, and finding out the optimal threshold for a given platform, can be done automatically in many cases [21, 23].

Benchmark	Number of processors								
	Seq.	1	2	3	4	5	6	7	8
AIAKL	1.00	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
Ann	1.00	0.98	1.86	2.65	3.37	4.07	4.65	5.22	5.90
Boyer	1.00	0.32	0.64	0.95	1.21	1.32	1.47	1.57	1.64
BoyerGC	1.00	0.90	1.74	2.57	3.15	3.85	4.39	4.78	5.20
Deriv	1.00	0.32	0.61	0.86	1.09	1.15	1.30	1.55	1.75
DerivGC	1.00	0.91	1.63	2.37	3.05	3.69	4.21	4.79	5.39
FFT	1.00	0.61	1.08	1.30	1.63	1.65	1.67	1.68	1.70
FFTGC	1.00	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
Fibonacci	1.00	0.30	0.60	0.94	1.25	1.58	1.86	2.22	2.50
FibonacciGC	1.00	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
Hamming	1.00	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
Hanoi	1.00	0.67	1.31	1.82	2.32	2.75	3.20	3.70	4.07
HanoiDL	1.00	0.47	0.98	1.51	2.19	2.62	3.06	3.54	3.95
HanoiGC	1.00	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
MergeSort	1.00	0.79	1.47	2.12	2.71	3.01	3.30	3.56	3.71
MergeSortGC	1.00	0.83	1.52	2.23	2.79	3.10	3.43	3.67	3.95
MMatrix	1.00	0.91	1.74	2.55	3.32	4.18	4.83	5.55	6.28
Palindrome	1.00	0.44	0.77	1.09	1.40	1.61	1.82	2.10	2.23
PalindromeGC	1.00	0.94	1.75	2.37	2.97	3.30	3.62	4.13	4.46
QuickSort	1.00	0.75	1.42	1.98	2.44	2.84	3.07	3.37	3.55
QuickSortDL	1.00	0.71	1.36	1.95	2.26	2.76	2.96	3.18	3.32
QuickSortGC	1.00	0.94	1.78	2.31	2.87	3.19	3.46	3.67	3.75
Takeuchi	1.00	0.23	0.46	0.68	0.91	1.12	1.32	1.49	1.72
TakeuchiGC	1.00	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63

Table 2. Speedups for restricted IAP.

Benchmark	Parallelism	Number of processors								
		Seq.	1	2	3	4	5	6	7	8
FFTGC	Restricted	1.00	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	Unrestricted	1.00	0.98	1.82	2.31	3.01	3.12	3.26	3.39	3.63
FibFunGC	Restricted	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Unrestricted	1.00	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
Hamming	Restricted	1.00	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	Unrestricted	1.00	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
TakeuchiGC	Restricted	1.00	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	Unrestricted	1.00	0.88	1.62	2.39	3.33	4.04	4.47	5.19	5.72
WMS2	Restricted	1.00	0.99	1.01	1.01	1.01	1.01	1.01	1.01	1.01
	Unrestricted	1.00	0.99	1.10	1.10	1.10	1.10	1.10	1.10	1.10

Table 3. Speedups for both restricted and unrestricted IAP.

Table 3 shows a different comparison: some programs have traditionally been executed under IAP using the restricted (nested fork-join) annotations, and can be annotated for parallelism using the more flexible `&>/2` and `<&/1` operators, as in Figures 3 and 4. In some cases those programs obtain little additional

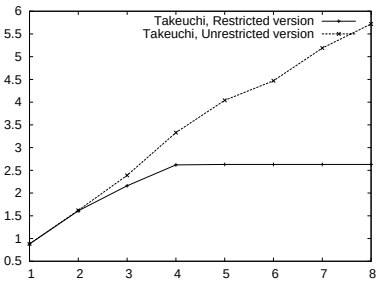


Fig. 9. Restricted and unrestricted IAP versions of *Takeuchi*.

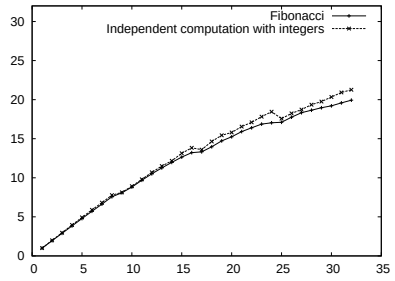


Fig. 10. *Fibonacci* with gran. control vs. maximum speedup in real machine.

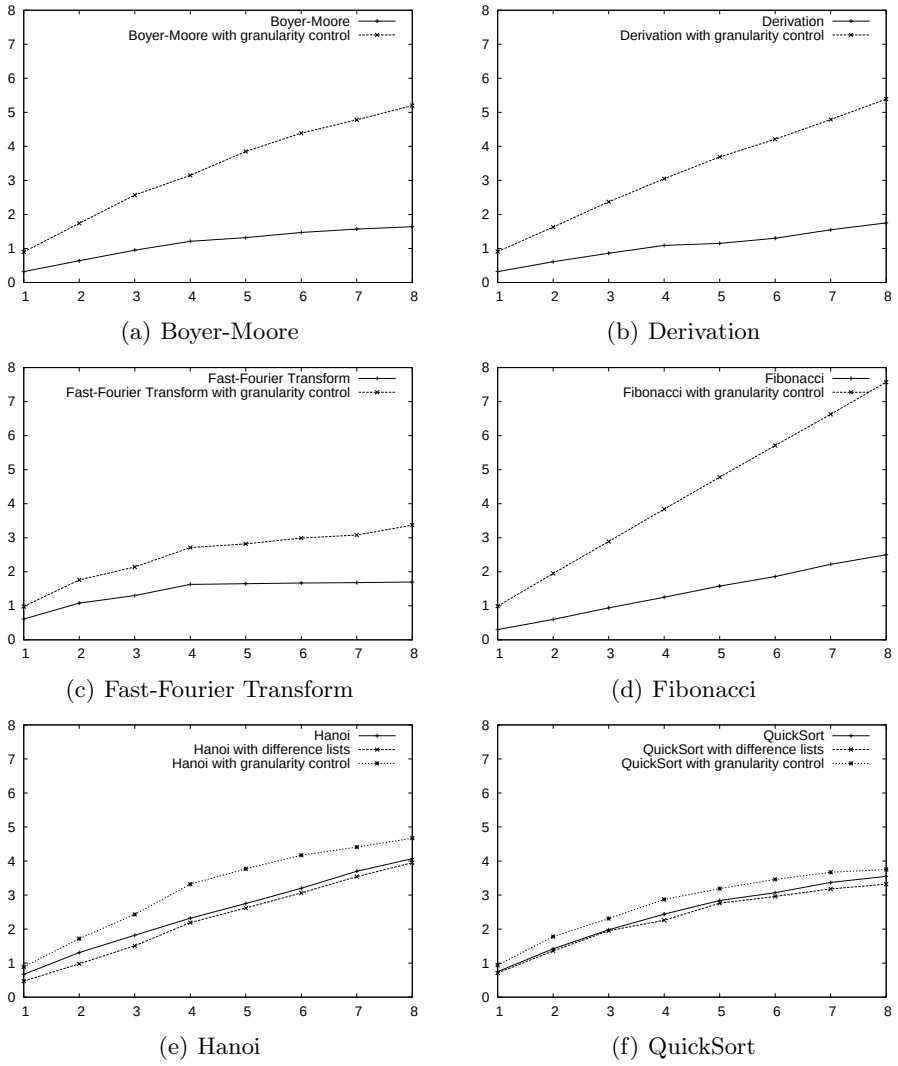
speedup, but, interestingly, in other cases the gains are very relevant. An interesting example is the *Takeuchi* function which underwent a manual (but mechanical) transformation involving an *unfolding* step, which produced a clause where non-nested fork-join [15] can be taken advantage of, producing a much better speedup. This can be clearly seen in Figure 9. Note that the speedup curve did not seem to stabilize even when the 8 processor mark was reached.

The *FibFun* benchmark is also an interesting case. A definition of Fibonacci was written in Ciao using the functional package [8] which implements a rich functional syntactic layer via compilation to the logic programming kernel. The automatic translation into predicates does not produce however the same Fibonacci program that programmers usually write (input parameters are calculated right before making the recursive calls), and it turns out that it cannot be directly parallelized using existing order-preserving annotators and restricted IAP. On the other hand it can be automatically parallelized (including the translation from functional to logic programming notation) using the unrestricted operators.

Despite our observation that the T2000 cannot produce linear speedups beyond 8 processors even for independent computations, we wanted to try at least a Prolog example using as many threads as natively available in the machine, and compare its speedup with that of a C program generating completely independent computations. Such a C program provides us with a practical upper bound on the attainable speedups. The results are depicted in Figure 10 which shows both the ideally parallel C program and a parallelized Fibonacci running on our implementation. Interestingly, the speedup obtained is only marginally worse than the best possible one. In both curves it is possible to observe a sawtooth shape, presumably caused by tasks filling in a row of units in all cores and starting to use up additional thread units in other cores, which happens at 1×8, 2×8, and 3×8 threads.

6 Conclusions

We have presented a new implementation approach for exploiting and-parallelism in logic programs with the objectives of simpler machinery and more flexibility. The approach is based on raising the implementation of some components to the



(a) Boyer-Moore (b) Derivation

(c) Fast-Fourier Transform (d) Fibonacci

(e) Hanoi (f) QuickSort

Fig. 11. Speedups for some selected benchmarks with and without granularity control.

source language level by using more basic high-level primitives than the fork-join operator and keeping only some relatively simple operations at a lower level. Our preliminary experimental results show that reasonable speedups are achievable with this approach, although the additional overhead, at least in the current implementation, makes it necessary to use granularity control in many cases in order to obtain good results. In addition, recent compilation technology and implementation advances [7, 25] provide hope that it will eventually be possible to recover a significant part of the efficiency lost due to the level at which parallel execution is expressed. Finally, we have observed that the availability of unrestricted parallelism contributes in practice to better observed speedups. We are

currently working on improving the implementation both in terms of efficiency and of improved support for backtracking. We have also developed simultaneously specific parallelizers for this approach, which can take advantage of the unrestricted nature of the parallelism which it can support [9].

Acknowledgments: this work was funded in part by the IST program of the European Commission, FP6 FET project IST-15905 *MOBIUS*, by the Ministry of Education and Science (MEC) project TIN2005-09207-C03 *MERIT-COMVERS* and by the Madrid Regional Government CAM project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo and Amadeo Casas were also funded in part by the Prince of Asturias Chair in Information Science and Technology at UNM.

References

1. Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT Press, 1991.
2. K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.
3. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.
4. F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Functional Programming Languages and Computer Architecture*, pages 187–195, October 1981.
5. D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System.* PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.
6. D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint Conference on Declarative Programming*, pages 67–78, July 1996.
7. M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo. High-Level Languages for Small Devices: A Case Study. In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.
8. A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *FLOPS'06*, Fuji Susono (Japan), April 2006.
9. A. Casas, M. Carro, and M. Hermenegildo. Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, number 4915 in LNCS, pages 138–153, The Technical University of Denmark, August 2007. Springer-Verlag.
10. M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems*, 22(2):269–339, March 2000.
11. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

12. R. H. Halstead. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS*, 7(4):501–538, October 1985.
13. M. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.
14. M. Hermenegildo. Parallelizing Irregular and Pointer-Based Computations Automatically: Perspectives from Logic and Constraint Programming. *Parallel Computing*, 26(13–14):1685–1708, December 2000.
15. M. Hermenegildo and M. Carro. Relating Data–Parallelism and (And–) Parallelism in Logic Programs. *The Computer Languages Journal*, 22(2/3):143–163, July 1996.
16. M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
17. M. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 40–55. Imperial College, Springer-Verlag, July 1986.
18. M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
19. 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.
20. Sverker Janson. *AKL. A Multiparadigm Programming Language*. PhD thesis, Uppsala University, 1994.
21. P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.
22. E. Lusk et al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.
23. E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.
24. 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.
25. Vítor Santos-Costa. Optimising Bytecode Emulation for Prolog. In *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *LNCS*, pages 261–277. Springer-Verlag, 1999.
26. Vítor Manuel de Morais Santos-Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I*. PhD thesis, University of Bristol, August 1993.
27. K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, November 1996.
28. K. Shen and M. Hermenegildo. Flexible Scheduling for Non-Deterministic, And-parallel Execution of Logic Programs. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 635–640. Springer-Verlag, August 1996.
29. D.H.D. Warren. An Abstract Prolog Instruction Set. TR 309, SRI International, 1983.