facultad de informática

universidad politécnica de madrid

**A Simulation Study on Parallel
Backtracking
with Solution Memoing
for Independent And-Parallelism**

Pablo Chico de Guzmán
Amadeo Casas
Manuel Carro
Manuel Hermenegildo

A Simulation Study on Parallel Backtracking with Solution Memoing for Independent And-Parallelism

Authors

Pablo Chico de Guzmán
`pchico@clip.dia.fi.upm.es`
Computer Science School
Universidad Politécnica de Madrid (UPM)

Amadeo Casas
`amadeo.c@samsung.com`
Samsung Research, USA.

Manuel Carro
`mcarro@fi.upm.es`
Computer Science School
Universidad Politécnica de Madrid (UPM)

Manuel Hermenegildo
`herme@fi.upm.es`
Computer Science School
Universidad Politécnica de Madrid (UPM)
and IMDEA Software, Spain

Abstract

Goal-level Independent and-parallelism (IAP) is exploited by scheduling for simultaneous execution two or more goals which will not interfere with each other at run time. This can be done safely even if such goals can produce multiple answers. The most successful IAP implementations to date have used recomputation of answers and sequentially ordered backtracking. While in principle simplifying the implementation, recomputation can be very inefficient if the granularity of the parallel goals is large enough and they produce several answers, while sequentially ordered backtracking limits parallelism. And, despite the expected simplification, the implementation of the classic schemes has proved to involve complex engineering, with the consequent difficulty for system maintenance and expansion, and still frequently run into the well-known trapped goal and garbage slot problems. This work presents ideas about an alternative parallel backtracking model for IAP and a simulation studio. The model features parallel out-of-order backtracking and relies on answer memoization to reuse and combine answers. Whenever a parallel goal backtracks, its siblings also perform backtracking, but after storing the bindings generated by previous answers. The bindings are then reinstalled when combining answers. In order not to unnecessarily penalize forward execution, non-speculative and-parallel goals which have not been executed yet take precedence over sibling goals which could be backtracked over. Using a simulator, we show that this approach can bring significant performance advantages over classical approaches.

Contents

1   Introduction

Widely available multicore processors have brought renewed interest in languages and tools to efficiently and transparently exploit parallel execution — i.e., tools to take care of the difficult [KB88] task of automatically uncovering parallelism in sequential algorithms and in languages to succinctly express this parallelism. These languages can be used to both write directly parallel applications and as targets for parallelizing compilers.

Declarative languages (and among them, logic programming languages) have traditionally been considered attractive for both expressing and exploiting parallelism due to their clean and simple semantics. A large amount of work has been done in the area of parallel execution of logic programs [GPA+01], where two main sources of parallelism have been exploited: parallelism between goals of a resolvent (And-Parallelism) and parallelism between the clauses of a predicate (Or-Parallelism). Among the systems to efficiently exploit Or-Parallelism we can cite Aurora [LBD+88] and MUSE [AK90], and among those executing exploiting And-Parallelism, &-Prolog [HG91] and DDAS [She96] are among the best known ones. In particular, &-Prolog exploits *Independent And-Parallelism*, where goals to be executed in parallel do not attempt to bind the same variables at run time and are launched following a nested fork-join structure. Other systems such as &ACE [PGH95], AKL [Jan94] and Andorra [San93] have approached a combination of both or- and and-parallelism. In this paper, we will focus on independent and-parallelism.

A critical area in the context of IAP that has received much attention is the implementation of backtracking. Since in IAP by definition goals do not affect each other, an obvious approach is to generate all the solutions for these goals in parallel independently, and then combine them [Con87]. However, this approach has several drawbacks. First, copying solutions, at least naively, can imply very significant overhead. In addition, this approach can perform an unbounded amount of unnecessary work if, e.g., only some of the solutions are actually needed, and it can even be non-terminating if one of the goals does not fail finitely.

For these reasons the operational semantics typically implemented in IAP systems performs an ordered, right-to-left backtracking. For example, if execution backtracks into a parallel conjunction such as `a & b & c`, the rightmost goal (`c`) backtracks first. If it fails, then `b` is backtracked over while `c` is recomputed and so on, until a new solution is found or until the parallel conjunction fails. The advantage of this approach is that it saves memory (since no solutions need to be copied) and keeps close to the sequential semantics. However, it also implies that many computations are redone and a large amount of backtracking work can be essentially sequential.

Herein we propose an improved solution to backtracking in IAP aimed at reducing recomputation and augmenting parallelism while preserving efficiency. It puts together memoization of answers to parallel goals (to avoid recomputation), out-of-order backtracking (the right-to-left rule is not followed) to exploit parallelism on backtracking, and incremental computation of answers, to reduce memory consumption and avoid termination problems.

We present some implementation ideas about this approach and we provide experimental simulated data that shows that the amount of parallelism exploited increases due to the parallel backward execution, while keeping competitive performance for first-answer queries. Also, super-linear speedups are achievable thanks to memoization of previous answers.

2 An Overview of IAP with Parallel Backtracking

In this section we provide a high-level view of the execution algorithm we propose.

The IAP + parallel backtracking model we propose behaves in many respects as classical IAP approaches, but it has as main difference the use of speculative computation (if possible) to generate eagerly additional solutions both in forward computation and when backtracking. This brings a number of additional changes which have to be accommodated.

Forward execution:

when a parallel conjunction is first reached, its goals are started in parallel. When a goal in the conjunction fails without returning any solution, the whole conjunction fails. When all goals have found a solution, execution proceeds as in classical IAP. However, if a solution has been found for some goals, but not for all, the agents which did finish may speculatively look for more solutions for the goals they executed, unless there is a need for agents to execute work which is not speculative. This in turn brings the need to stash away the generated solutions in order to continue searching for more answers (which are also saved). When all goals find a solution, those which were speculatively executing are suspended (to preserve the no-slowdown property [HR95]), their state is saved to be resumed later, and their first answer is reinstalled.

Backward execution:

we only perform backtracking on the goals of a parallel conjunction which are on top of the stacks. If necessary, stack sections are reordered to move trapped goals to the top of the stack. In order not to impose a rigid ordering we allow backtracking on these goals to proceed in an arbitrary order (i.e., not necessarily corresponding to the lexical right-to-left order). This opens the possibility of performing backtracking in parallel, which brings some additional issues to take care of:

- When some of the goals executing backtracking in parallel finish, backtracking stops by suspending the rest of the goals and saving their state.

- The solution found is saved in the memoing area, in order to avoid recomputation.

- Every new solution is combined with the previously available solutions. Some of these will be recovered from the memoization memory and others may simply be available if they are the last solution computed by some goal and thus the bindings are active.

- If more solutions are needed, backwards execution is performed in parallel again. Goals which were suspended resume where they suspended.

All this brings the necessity of saving and resuming execution states, memoing and recovering answers quickly, combining previously existing solutions with newly found solutions, assigning agents to speculative computations only if there are no non-speculative computations available, and managing computations which change from speculative to non speculative. Note that all parallel backtracking is speculative work, because we might need just one more answer of the rightmost parallel goal, and this is why backwards execution is given less priority than forward execution. Note also that at any point in time we only have an active value for each

variable. While performing parallel backtracking we can change the bindings which will be used in forward execution, but before continuing with forward execution, all parallel goals have to suspend to reinstall the bindings of the answer being combined.

3   An Execution Example

In this section we present an example illustrating different aspects of the approach, and specially how the execution of a parallel program may benefit from memoization of answers and parallel backtracking.

We will use the following program:

```
main(X, Y, Z, T) :- a(X, Y) & b(Z, T).
a(X, Y) :- a1(X) & a2(Y).
b(X, Y) :- b1(X) & b2(Y).
```

We will assume that `a1(X)`, `a2(Y)`, `b1(X)` and `b2(Y)` have two answers each, which take 1 and 7 seconds, 3 and 13 seconds, 2 and 10 seconds, and 4 and 25 seconds, respectively. We will also assume there are no dependencies among the variables in the literals of these clauses, and that the cost of preparing and starting up parallel goals is negligible. Finally, we will assume that there are two agents available to execute these goals at the beginning of the execution of the predicate `main/4`. Figure 1 summarizes the evolution of the stack of each agent throughout the execution of `main/4`.

Once the first agent starts the execution of `main/4`, `a/2` is published for parallel execution and `b/2` is executed locally. The second agent steals `a/2`, publishes `a1/1` for parallel execution and executes `a2/1` locally, while the first agent marks `b1/1` as parallel and executes `b2/1`. The execution state can be seen in Figure 1(a). When the second agent finds then the first answer for `a2/1`, it marks `a2/1` to be executed in a speculative manner. However, since `a1/1` and `b1/1` are still pending, the second agent will start executing one of them instead. We will assume it starts executing `a1/1`. Once it finds an answer, `a1/1` is marked to be executed speculatively. Since `a2/1` is also marked as such, then the entire predicate `a/2` can be configured to be executed speculatively. However, the second agent will now execute `b1/1` since it is pending and has higher priority than speculative execution (Figure 1(b)).

Figure 1(c) shows the execution state when the first agent finds an answer for `b2/1`. In this case, since there is no other parallel goal to execute, the first agent starts the execution of `b2/1` speculatively, until the second agent finishes the execution of `b1/1`. When that happens, the first agent suspends the execution of `b2/1` and the first answer of `main/4` is returned, as shown in Figure 1(d).

In order to calculate the next answer of `main/4`, both agents will backtrack over `b2/1` and `b1/1`, respectively. Note that they would not be able to backtrack over any of the subgoals in `a/2` because they are currently trapped. Once the second agent finds the second answer of `b1/1`, the first agent suspends the execution of `b2/1` and returns the second answer of `main/4`, combining all the existing answers of its literals.

In order to obtain the next answer of `main/4`, the second agent continues with the execution of `b1/1`, and the first agent fails the execution of `b2/1` and starts computing the next answer of `a1/1`, since that goal has now been freed, as shown in Figure 1(e). Whenever the answer of `a1/1` is completed, shown in Figure 1(f), the execution of `b2/1` is again suspended and a set of
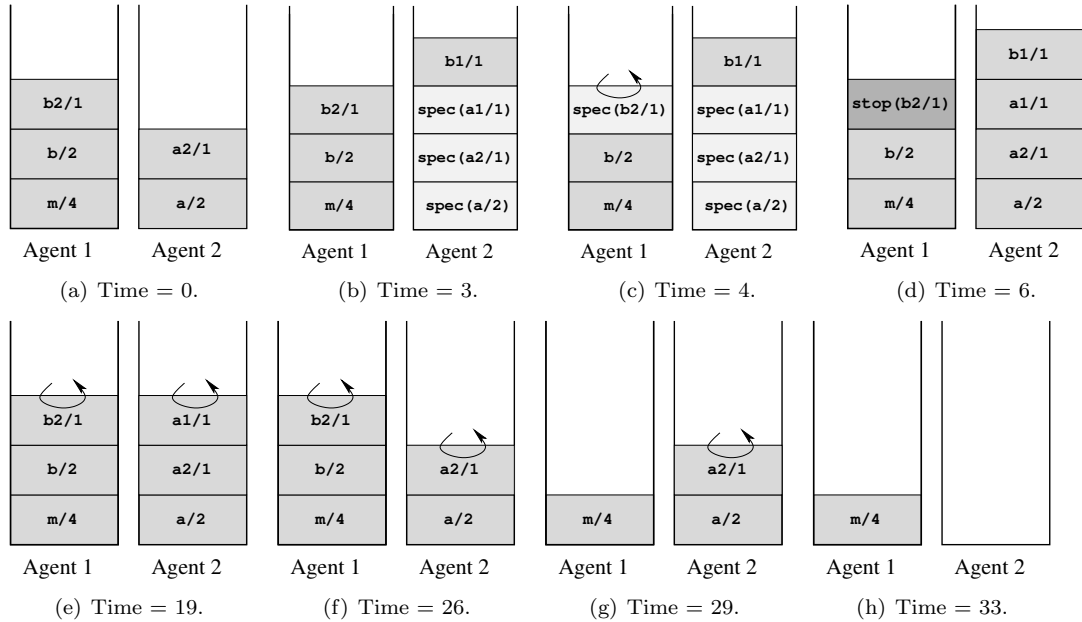
Figure 1: Execution of main/4 with memoization of answers and parallel backtracking.

new answers of main/4 not involving a new answer for a2/1 and b2/1 can be returned, again as a combination of the already computed answers of its subgoals. To obtain the rest of the answers of predicate main/4, the first agent resumes the execution of b2/1 and the second agent starts calculating a new answer of a2/1 (Figure 1(g)). The first agent finds the answer of b2/1, suspends the execution of the second agent, and returns the new answers of main/4. Finally, Figure 1(h) shows how the second agent continues with the execution of a2/2 in order to obtain the rest of the answers of main/4.

Note that in this example memoization of answers avoids having to recompute expensive answers of parallel goals. Also note that all the answers for each parallel literal could have been found separately and then merged, producing a similar total execution time. However, the computational time for the first answer would have been drastically increased.

4 Memoization vs. Recomputation

Classic IAP uses recomputation of answers: if we execute a(X) & b(Y), the first answer of each goal is generated in parallel. On backtracking, b(Y) generates additional answers (one by one, sequentially) until it finitely fails. Then, a new answer for goal a(X) is computed in parallel with the recomputation of the first answer of b(Y). Successive answers are computed by backtracking again on b(Y), and later on a(X).

However, since a(X) and b(Y) are independent, the answers of goal b(Y) will be the same in each recomputation. Consequently, it makes sense to store its bindings after every answer is generated, and combine them with those from a(X) to avoid the recomputation of b(Y). Memoing answers does not require having the bindings for these answers on the stack; in fact they should be stashed away and reinstalled when necessary. Therefore, when a new answer is
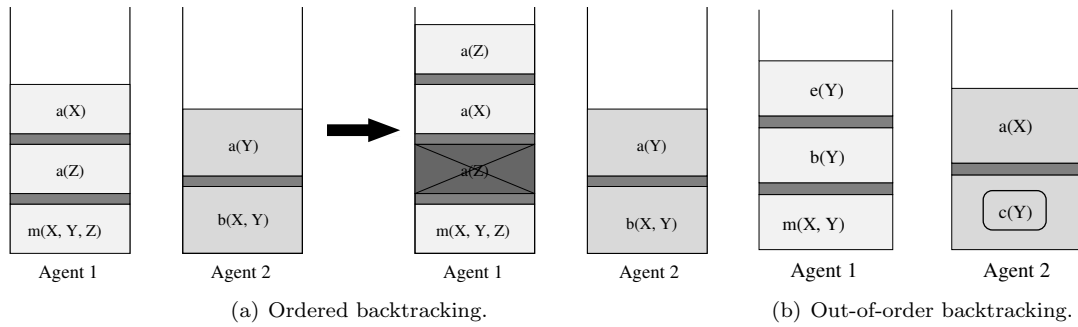
(a) Ordered backtracking.   (b) Out-of-order backtracking.

Figure 2: Trapped goal problem with ordered and out-of-order backtracking in IAP.

computed for a(X) the previously computed and memorized answers for b(Y) are restored and combined.

5   Backtracking Order, Trapped Goals, and Parallel Backtracking

The classical, right-to-left backtracking order for IAP is known to bring a number of challenges, among them the possibility of *trapped goals*: a goal on which backtracking has to be performed becomes *trapped* by another goal stacked on top of it. Normal backtracking is therefore impossible. Consider the following example:

```
m(X,Y,Z) :- b(X,Y) & a(Z).
b(X,Y) :- a(X) &  a(Y).
a(1). a(2).
```

Figure 2(a) shows a possible state of the execution of predicate m/3 by two agents. When the first agent starts computing m/3, b(X, Y) and a(Z) are scheduled to be executed in parallel. Assume that a(Z) is executed locally by the first agent and b(X,Y) is executed by the second agent. Then, the second agent schedules a(X) and a(Y) to be executed in parallel, which results in a(Y) being locally executed by the second agent and a(X) executed by the first agent after computing an answer for a(Z). In order to obtain another answer for m/3, right-to-left backtracking requires computing additional answers for a(Z), a(Y), and a(X), in that order. However, a(Z) cannot be directly backtracked over since a(X) is stacked on top of it: a(Z) is a *trapped goal*.

Several solutions have been proposed for this problem. One of the original proposals uses *continuation markers* [Her86, SH96] to *skip* over stacked goals. This is, however, difficult to implement properly and needs to take care of a large number of cases. It can also leave unused sections of memory (*garbage slots*) which are either only reclaimed when finally backtracking over the parallel goals, or require quite delicate memory management. A different solution [CCH08] is to move the execution of the trapped goal to the top of the stack. This simplifies the implementation somewhat, but it also leaves garbage slots in the stacks.

5.1   Out-of-Order Backtracking

In order to greatly reduce the likelihood of the appearance of trapped goals and garbage slots we propose to take an alternative approach: relaxing the sequential backtracking order. The key idea is to allow backtracking (and therefore the order of solutions) to dynamically adapt to the configuration of the stacks.

The obvious drawback of this approach is that it may alter solution order with respect to sequential execution, and in an unpredictable way. However, we argue that in many cases this may not be a high price to pay, specially if the programmer is aware of it and can have a choice. Programs where solution order matters, typically because of efficiency, are likely to have dependencies between goals which would anyway make them not amenable for IAP. For independent goals we argue that allowing out-of-order backtracking represents in some way a return to a simpler, more declarative semantics that has the advantage of allowing higher efficiency in the implementation of parallelism. Solution order is also relaxed traditionally in or-parallel systems, and a similar situation arises in tabling, where solutions are also generated in an order which does not necessarily match that of SLD. In return, termination is ensured for a large class of interesting programs (making the operational semantics closer to the declarative one) and other, already terminating programs, are greatly sped up.

The alternative we propose herein consists of always backtracking over the goal that is on top of the stack, without taking into account the original goal execution order. In the case of backwards execution over predicate m/3 in Figure 2(a), both agents may be able to backtrack over a(X) and a(Y), without having to move the execution of a(Z). Note that even though the order of the answers for predicate m/3 may change with respect to the sequential execution, the first answer will remain the same.

5.2   Parallel Backtracking

Once we allow backwards execution over any parallel goal on the top of the stacks, we can perform backtracking over all of them in parallel. Consequently, each time we perform backtracking over a parallel conjunction, each of the parallel goals of the parallel conjunction can start speculative backwards execution.

6   The Simulator

The simulation consists of two steps. First, the simulation performs a controlled execution of the program we want to simulate to generate a trace of the main events of the execution. Then, the second step is a simulation of this trace to obtain the speedups simulating from one to eight agents.

6.1   Generating the Parallel Execution Trace

There is a program transformation to create a new program from the original one. This new program will generate the parallel execution trace when it is executed. The events which can appear in this trace are: TIME(N), START(Goal List), RESTART(Goal List), ANSWER and FAIL.

TIME is used to obtain a virtual execution time of the program. It is obtained from the number of arguments of each head of a executed clause. This is a simple approach to forecast how long is going to take the program execution. It is not a real time execution but there is a relation between the execution time and the number of unified heads and then, it can be used to obtain simulated speedups. Moreover, arithmetics operation are simulated with 10 units of time and I/O disk operation are simulated with 1000 units of time. The predicate `pause/1` is used to force a simulated time given by its argument.

START(Goal List) is used to indicate that several goals are going to be published. In other words, it indicates the execution of a goal parallel conjunction. RESTART(Goal List) indicates backtracking over a goal parallel conjunction. ANSWER indicates that a parallel goal has found an answer and FAIL indicates than a parallel goal fails.

When the transformed program is executed to generate the parallel trace, the initial goal is associated to the parallel goal execution number 1, and the coming events are associated with the parallel execution number 1. Each time a parallel conjunction appears, its parallel goals are executed to generate their parallel execution trace. Each parallel goal is associated with a new consecutive number which is used to make references by START and RESTART events. The final trace is a sequence of parallel goal numbers, each one associated with a parallel execution trace.

Note than we are not taking into account parallel execution overheads for answer memorization, goal publication, threads coordination and so on and then, the obtained speedups are idealistic speedups for the simulated program.

6.2   Simulating the Parallel Execution Trace

The parallel execution trace is used to obtain the simulated speedups. We are simulating two IAP approaches to be later compared, classical sequential backtracking approach (*seqback*) and parallel backtracking with answer memorization (*parback*), the IAP approach we have introduced in this paper. Note than, as long as we are not taking into account parallel execution overheads, the simulation with one processor for the classical IAP approach is valid as a sequential execution simulation and the simulation with one processor for the parallel backtracking with answer memorization IAP approach is valid as a sequential execution using memorization.

The simulation uses a virtual stack for each thread. The initial goal is associated with one of these virtual stacks and it starts to process events of the initial goal. Each time a TIME(N) event is processed a global time counter is incremented on N units. When a START(Goal List) event is processed, each goal in Goal List is published. If there are available threads, they can steal any of the published parallel goal. Each simulation step looks the next event of the active virtual stacks. If all the next events are TIME(N) events, the global time counter is incremented on the minimum N value of these TIME(N) events, and the remaining time of TIME(N) events is decremented on this same value. When N comes zero, the TIME(0) events is eliminated of the list. If there are an coming events which is not a TIME(N) event, it is processed first without incremented the global time counter because we are not taking into account parallel execution overheads.

When all the parallel goals of a parallel conjunction have executed an ANSWER event, the virtual stack of the parallel goal which executed the parallel conjunction is activated. It is recorded the global time counter when the initial goal computes its first solution and when it finally fails.

The simulation is quite similar to what a parallel scheduler would do. There a published parallel goal list, a list of backtrackable parallel goals... but it is quite easier to implement than a real parallel scheduler because there is not synchronization issues. START, RESTART, ANSWER and FAIL events are used to updated these lists and to know which parallel goals can be executed.

The only difference between sequential backtracking IAP approach and parallel backtracking IAP approach is the precessing of ANSWER, FAIL and RESTART events. *seqback* stops the execution of the current goal when an ANSWER event is processed. When the right most parallel goal in a parallel conjunction processes an ANSWER event, the parallel goal where the parallel conjunction is executed is reactivated. On the other hand, *parback* can continue with the simulation of a parallel goal when an ANSWER event is processed is there is not more available work. This is speculative work, but it is worth because the thread will become handle otherwise. When the last parallel goal of a parallel conjunction processes an ANSWER event, the rest of parallel goals in this parallel conjunction are stopped (suspended) and the parallel goal where the parallel conjunction is executed is reactivated.

seqback looks a new answer of the next left parallel goal in a parallel execution when a FAIL event is processed, and the actual failed parallel goal is published again. *parback* does not do anything with FAIL event because the rest parallel goals in the parallel conjunction are already performing backtracking. When the last parallel goal finally fails, the parallel goal where the parallel conjunction is executed is reactivated to simulate backtracking.

seqback only activates the right most parallel goal in a parallel conjunction when RESTART event is processed. On the other hand, *parback* reactivates all the parallel goals of the parallel conjunction to simulate parallel backtracking.

7   Simulation Results of IAP Models

In this section, we present a comparison between the simulation of classical implementation of IAP [CCH08] (seqback) with our proposed approach (parback).

We show simulation speedups of both parback and seqback on deterministic benchmarks to determine the accurately of our simulation. The deterministic benchmarks used are the well-known Fibonacci series (*fibo*), matrix multiplication (*mmat*) and QuickSort (*qsort*). *fibo* generates the 22th Fibonacci number switching to a sequential implementation from the 12th number downwards, *mmat* uses 50x50 matrices and *qsort* is the version which uses append/3 sorting a list of 10000 numbers. The GC suffix means task granularity control [LGHD96] is used for lists of size 300 and smaller.

The selected nondeterministic benchmarks are *checkfiles*, *illumination*, and *qsort_nd*. *checkfiles* receives a list of files, each of which contains a list of file names which may exist or not. These lists are checked in parallel to find nonexistent files which appear listed in all the initial files; these are enumerated on backtracking. *illumination* receives an $N \times N$ board informing of possible places for lights in a room. It tries to place a light in each of the columns, but lights in consecutive columns have to be separated by a minimum distance. The eligible positions in each column are searched in parallel and the distance condition is checked at the end. *qsort_nd* is a QuickSort algorithm where the order between the list elements is a partial one. *checkfiles* and *illumination* are synthetic benchmarks which create 8 parallel goals and which exploit memoization heavily. *qsort_nd* is a more realistic benchmark which creates over one thousand

parallel goals.

Table 1 shows the speedups obtained. The speedups shown in this table are calculated with respect to the sequential execution of the original, unparallelized benchmark (simulation using only one thread). Therefore, the column tagged 1 corresponds to the slowdown coming from executing a parallel program on a single processor. Since we are not taking into account any kind of parallel overhead, the value of this column is always one.

For nondeterministic benchmarks we show a comparison of the performance results obtained both to generate the first solution (seqback$_{first}$ and parback$_{first}$) and all the solutions (seqback$_{all}$ and parback$_{all}$). We show speedups relative to the execution in parallel with memoing in one thread, but there would be an inherent speedup because of the memorization here respect to a sequential execution.

The simulated speedups obtained are enought accurate for the case of deterministic benchmarks. It gives us some confidency about taking conclusions for non-deterministic benchmarks.

For non-deterministic benchmarks, the behavior of parback and seqback is quite similar in the case of *qsort_nd* when only the first answer is computed because there is not backtracking here. For all answer queries, parallel backtracking plus memoization brings clear advantages.

In the case of *checkfiles* and *illumination* backtracking is needed even to generate the first answer, and memoing plays an important role. The speedups obtained by parback increase in some way closer to the increase in the number of processors —with some superlinear speedup which is normal when search does not follow, as in our case, the same order as sequential execution. In the case of seqback, which performs essentially sequential backtracking, speedups remain constant.

When all the answers are required, speedups grow noticeably. This can be traced to the increased amount of backtracking that is performed in parallel. This behavior also appears, to a lesser extent, in *qsort_nd*.

Note that the speedups of *checkfiles* and *illumination* stabilize between 4 and 7 processors. This is so because they generate exactly 8 parallel goals, and there is one dangling goal to be finished. In the case of *checkfiles* we get superlinear speedup because there are 8 lists of files to check. With 8 processors the first answer can be obtained without traversing (on backtracking) any of these lists. This is not the case with 7 processors (but it could have been if backtracking were done in the right order), and so there is no superlinear behavior until we hit the 8 processor mark.

8 Conclusions

We have present a high-level parallel backtracking approach for independent and-parallelism which shows great advantages in the simulation of non-deterministic parallel calls due to the avoidance of having to recompute answers and due to the fact that parallel goals can execute backwards in parallel, which was a limitation in previous similar implementations. Our simulation studio clearly shows that the amount of exploitable parallelism increases thanks to perform backwards execution in parallel.

Benchmark	Approach	Number of threads							
		1	2	3	4	5	6	7	8
Fibo	seqback	1	1.99	2.98	3.98	4.98	5.97	6.97	7.96
	parback	1	1.99	2.98	3.98	4.98	5.97	6.97	7.96
QSort	seqback	1	1.85	2.64	3.33	3.81	4.19	4.53	4.87
	parback	1	1.85	2.64	3.33	3.81	4.19	4.53	4.87
MMat	seqback	1	1.97	2.95	3.94	4.92	5.91	6.91	7.90
	parback	1	1.97	2.95	3.94	4.92	5.91	6.91	7.90
CheckFiles	seqback_{first}	1	1.15	1.18	1.21	1.23	1.24	1.25	1.25
	seqback_{all}	1	1.13	1.16	1.18	1.19	1.19	1.20	1.20
	parback_{first}	1	2.50	2.93	4.97	4.97	4.97	4.97	12.12
	parback_{all}	1	1.89	2.83	3.78	3.78	3.82	3.82	7.56
Illumination	seqback_{first}	1	1.22	1.26	1.28	1.29	1.29	1.29	1.30
	seqback_{all}	1	1.17	1.20	1.22	1.24	1.24	1.25	1.25
	parback_{first}	1	2.23	2.76	4.46	4.46	4.46	4.46	11.6
	parback_{all}	1	1.98	2.98	3.97	3.97	3.97	3.97	7.96
QSortND	seqback_{first}	1	1.82	2.56	3.22	3.73	4.08	4.44	4.76
	seqback_{all}	1	1.25	1.28	1.30	1.31	1.31	1.32	1.42
	parback_{first}	1	1.82	2.56	3.22	3.73	4.08	4.44	4.76
	parback_{all}	1	1.83	2.61	3.27	3.75	4.10	4.42	4.77

Table 1: Simulation speedups for several benchmarks and implementations.

References

[AK90]   K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Perfor-mance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[CCH08]   A. Casas, M. Carro, and M. Hermenegildo. A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism. In M. García de la Banda and E. Pontelli, editors, *24th International Conference on Logic Program-ming (ICLP'08)*, volume 5366 of *LNCS*, pages 651–666. Springer-Verlag, December 2008.

[Con87]   J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, 1987.

[GPA$^+$01] 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.

[Her86]   M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Ar-chitecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, U. of Texas at Austin, August 1986.

[HG91]   M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HR95]   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.

[Jan94]   Sverker Janson. *AKL. A Multiparadigm Programming Language.* PhD thesis, Upp-sala University, 1994.

[KB88]   A.H. Karp and R.C. Babb. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, September 1988.

[LBD$^+$88] E. Lusk, R. Butler, T. Disz, R. Olson, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, P. Brand, M. Carlsson, A. Ciepielewski, B. Hausman, and S. Haridi. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2/3):243–271, 1988.

[LGHD96]   P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granu-larity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Compu-tation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.

[PGH95]   E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Paral-lel Prolog System. In *International Parallel Processing Symposium*, pages 564–572. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Com-puter Society, April 1995.

[San93]   Vítor Manuel de Morais Santos-Costa. *Compile-Time Analysis for the Parallel Ex-ecution of Logic Programs in Andorra-I.* PhD thesis, University of Bristol, August 1993.

[SH96]   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.

[She96]   K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, November 1996.