Exploiting Path Parallelism in Logic Programming*

Jordi Tubella and Antonio González

Universitat Politècnica de Catalunya
C/. Gran Capità, s/n. - Edifici D6
E-08071 Barcelona (Spain)
e-mail: {jordit,antonio}@ac.upc.es

Abstract

This paper presents a novel parallel implementation of Prolog. The system is based on Multipath [12], a novel execution model for Prolog that implements a partial breadth-first search of the SLD-tree. The paper focusses on the type of parallelism inherent to the execution model, which is called *path parallelism*. This is a particular case of data parallelism that can be efficiently exploited in a SPMD architecture. A SPMD architecture oriented to the Multipath execution model is presented. A simulator of such system has been developed and used to assess the performance of path parallelism. Performance figures show that path parallelism is effective for non-deterministic programs.

1 Introduction

The implementation of Prolog systems usually relies on the inference rule based on *SLD-resolution* [6, 8], which is used in a top-down fashion. The application of the resolution steps starts from the initial program query. Every resolution step takes a query and tries to match one of its goals against the head of a clause. Unification is used to match both predicates. If it succeeds, then resolution computes a new query or resolvent, substituting the selected goal with the subgoals in the body of the clause. This process goes on recursively until either some goal has no matching clause, or until an empty query is generated. The application of a SLD-resolution needs the definition of a selection rule (also named computation rule) to choose a goal from the query. In standard Prolog, this rule selects the *leftmost goal*.

*This work has been supported by the Ministry of Education of Spain under grant CICYT TIC 91/1036.

The fact that there can be more than one clause to match with a given goal introduces non-determinism at the resolution level. The execution of a Prolog program consists in the exploration of a search space, whose shape is a tree and which usually is referred to as the SLD-tree or search tree. Given a SLD-tree, the strategy to traverse it is known as the search rule. Sequential implementations of Prolog make use of a *depth-first left-to-right* search rule [7], that is, the SLD-tree is traversed in depth-first order, trying the clauses in a resolution step in the textual order, and backtracking to the youngest resolution with remaining clauses to match when the process described in the above paragraph finishes.

For instance, the Warren Abstract Machine [1], which is the starting point of many sequential execution models, implements the above mentioned leftmost selection rule with the depth-first left-to-right search rule.

Traditional implementations of Prolog have been based on a depth-first execution model, whereas a breadth-first search has been neglected. This is mainly due to the fact that a breadth-first search would require a huge amount of memory, resulting in an impracticable approach or in a too slow system due to the management of such big memory.

Apart from that, a breadth-first search theoretically has a number of advantages when compared with the traditional depth-first search. These advantages come from the fact that, once we have found all the solutions to a goal, they can be collected together and subsequently processed simultaneously (not necessarily in parallel). For instance, suppose the following program:

```
?- p(X),q(X),r(X).
p(0).          q(0).
p(1).          q(2).
p(2).          q(4).
```

After finding all solutions to `p/1`, we have three possible bindings for `X: X/0, X/1, X/3`. The three possible bindings are collected and processed simultaneously when `q/1` is executed. At that time, we see that only the first and third bindings are consistent with `q/1` and therefore, the second binding is discarded. When `r/1` is executed, `X` has two possible bindings: `X/0, X/2`, that are processed at the same time. In short, instead of traversing the SLD-tree carrying just one binding environment, the traversal is done processing at the same time several binding environments, one for each solution found so far.

Processing multiple binding environments simultaneously results in the following benefits:

- The overhead due to the execution of control instructions is considerably reduced. The impact of this is especially important for combinatorial search problems as it will be shown in this paper.
- The number of unifications decreases. This happens when the same unification occurs in different paths (branches) of the SLD-tree. The depth-first search repeats the unification operation every time, whereas a breadth-first search avoids these unnecessary recomputations.
- A new type of parallelism, called *path parallelism*, is exhibited by the model. This type of parallelism is exploited at each operation to be performed on any variable with multiple bindings (unification, creation, ...). The different bindings belong to different binding environments and can be processed in parallel. This type of parallelism can be efficiently exploited by a SPMD-like computer, having a main processing unit, which conducts the search of the SLD-tree, together with several unification units working in parallel.

As it usually happens, the most efficient solution may be a trade-off between two opposite designs. This led us to think about the possibility of combining a depth-first and a breadth-first search in the same execution model. This combination is called a *partial breadth-first search* and it is the search rule used by *Multipath*, a novel execution model for Prolog.

For details about the Multipath execution model and its sequential implementation we refer the interested reader to [11, 12, 3]. Multipath has been proved to be more efficient than the traditional depth-first based execution model on a sequential environment [12]. In this paper, we focus on the parallel implementation of the Multipath execution model and on those issues related to the exploitation of path parallelism.

The rest of the paper is organized as follows. Related work is reviewed in section 2. Section 3 presents a brief summary of the Multipath execution model. The main issues regarding the parallel implementation of Multipath are analyzed in section 4. Performance figures about the performance of the parallel implementation are presented in section 5. Finally, we summarize the main conclusions of this work.

2 Related work

A preliminary definition of the Multipath execution model was presented in [11]. At about the same time, in [9] D. A. Smith presents Multilog as a data parallel language that computes solutions to any arbitrary goal into a set of environments, and subsequent goals execute in this set of environments, with unification being performed in parallel. Multilog and MEM have been carried out concurrently and independently. They have similar objectives although their implementations are rather different.

Another related work is the DAP Prolog system proposed in [5], in particular its set mode operation. This proposal extends the standard implementation of Prolog in order to support sets of data and exploit parallelism for managing the different elements of a set. There are several important differences between DAP and Multipath. First, DAP extends the semantics of Prolog with sets whereas Multipath is transparent to the programmer (a standard Prolog program does not require any modification to be executed by Multipath). Second, the source of parallelism in DAP is due only to facts while, in Multipath, multiple bindings to the same variable can be obtained as a result of any type and number of clauses. Finally, the implementation of Multipath is simpler since the different relationships that may happen to have different sets makes the management of sets very cumbersome in DAP.

Path parallelism is a particular case of data parallelism. Data parallelism consists in the concurrent treatment of multiple bindings of variables. In the literature, data parallelism has been exploited in the context of OR-parallel systems [4]. In this approach, after binding a variable to multiple values, execution can continue in parallel exploring the subtrees related to each binding in an independent way. For each one of these subtrees, just one of the bindings is visible. In path parallelism, variables get multiple bindings as a result of the sequential execution of a nondeterminate goal. When all bindings are collected, parallelism is

exploited when a unify operation is to be performed on variables with multiple bindings.

Path parallelism is also different from unification parallelism [2]. In unification parallelism, parallel unifications are performed on different arguments of a goal for a single binding environment. In this case, there may be data dependences among the unifications. In path parallelism, the parallel execution corresponds to unifications of the same argument for different binding environments. In this case there are no data dependences.

3 The Multipath Execution Model

The main feature of Multipath is that it allows a given goal to be executed in depth-first, breadth-first or partial breadth-first order. The choice among these three options can be made by the programmer, the compiler and the execution model itself at run time.

In this context, a breadth-first execution of a goal means to explore all the alternative clauses that can lead to solutions of the goal before proceeding with the next goal, whereas a partial breadth-first execution means to explore in breadth-first order some (but not all) the alternative clauses, and go back to explore the remaining ones by means of backtracking. These remaining clauses may be then explored in any of the three possible orders (depth, breadth or partial breadth).

In this way, unlike a pure breadth-first execution, the amount of memory required by the execution of a partial breadth-first search can be limited. Once the amount of memory used has grown to a given size, breadth-first execution is not permitted. When some of the memory used is released (because of failures in some binding environments), a breadth-first is again allowed for those goals identified either by the programmer or compiler.

A goal that is executed in breadth or partial breadth order may result in some variables bound to multiple values. For instance, in the following code:

```
p(Y):-q(X),r(X,Y).
q(1).                r(2,3).
q(2).                r(3,4).
q(3).                r(4,5).
```

a breadth execution of `q/1` results in `X` being bound to three different values 1, 2 and 3.

The multiple bindings that result from a (partial) breadth execution of a goal are then processed all together afterwards. In the previous example, when `X` is dereferenced during the execution of `r/2` the result will be that it is bound to three different values: 1, 2 and 3. This is equivalent to say that the execution model explores several paths (branches) of the SLD-tree at the same time.

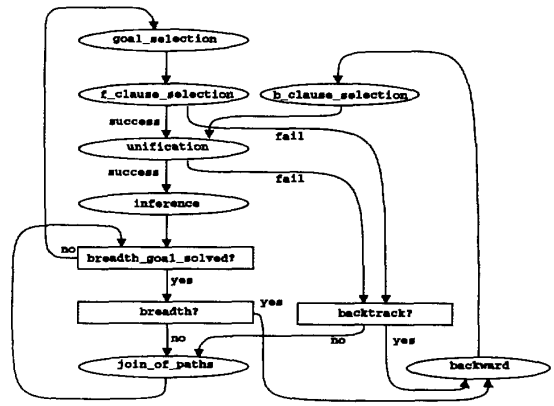

Figure 1: Basic operations of the Multipath execution model.

Each resolution step of the Multipath execution model (MEM) involves the following operations. The sequence among these operations is illustrated in figure 1 and commented in the next paragraphs. A further explanation of MEM can be found in [12].

GOAL_SELECTION selects the leftmost goal in the current query (as stated by the selection rule) in order to attempt an inference.

F_CLAUSE_SELECTION selects a clause to be unified with the chosen goal. If there are more than one candidate clause, they are tried in the order they are written. Thus, f_clause_selection always selects the first one. Note that f_ stands for forward execution to differentiate it from the clause selection operation to be performed in backward execution (see below).

Then, the selected goal (including its arguments) and the head of the selected clause are unified in the UNIFICATION operation. The unification is performed for every current binding environment. The unification operation fails if every individual unification in each binding environment fails, otherwise succeeds. The agent that is responsible for managing each different binding environment is called a *Unification Engine* (UE). Since unifications related to different binding environments are independent, and they can be carried out in parallel, we could have as many parallel threads as number of UEs. However, as we will discuss later on, it could be more effective to map several

UEs to a single thread because of granularity reasons. This source of parallelism is what we call *path parallelism* and it will be further analyzed in the following sections.

If unification succeeds, the selected goal is substituted by all the subgoals in the body of the clause, and bindings representing the most general unifier in each binding environment are added to it.

When a solution to a breadth goal is found (this condition is tested in BREADTH_GOAL_SOLVED?), it is decided whether the execution continues with a breadth-first or a depth-first search, that is, with a backward or a join of paths operation, respectively. In the current implementation, this decision depends only on the amount of available memory. That is, there is a limit on the number of binding environments (and thus, in the number of UEs) that can be used at the same time by the execution. When this number is reached, the execution shifts to a depth-first search even if the current predicate has still more alternatives. The remaining alternatives will be explored by backtracking. When a binding environment fails and it is known that it is not required any more (it is still required if backtracking to a location before the fail is possible), it is appended to the list of free binding environments and can be used to perform a breadth-first search of following goals. That is, the depth-first search shifts to a breadth-first search when there are again enough available binding environments.

Currently, more complex strategies to shift between breadth-first and depth-first search are being analyzed in order to take into account other overheads of the system like copying of binding environments, which occurs in the backward operation (see below).

In the Multipath execution model, when a unification fails, execution does not continue immediately with a backtracking operation. There is a BACKTRACK test that considers the existence of a breadth goal that has no possibilities to obtain more solutions. In this case, a join of paths operation is executed. Otherwise, a backward operation is executed.

The JOIN_OF_PATHS operation joins all solutions found so far to the youngest breadth goal of the SLD-tree and proceeds with a forward execution.

In case of a BACKWARD operation to the youngest branch alternative of the SLD-tree, the main action is to restore the computation state at that point. There are two ways to restore it: in case the binding environment has failed previously, it is done by simple backtracking; and, in case the binding environment is still active, it is done by allocating a free binding environment and copying to it the contents of the active binding environment.

B_CLAUSE_SELECTION selects the next candidate clause to be unified with the leftmost goal in the current query.

Operations described above are continually repeated until one or all solutions to the program have been found. The agent that conducts the search in this way is called *Main Engine* (ME).

4 Parallel implementation of Multipath

In this section, the main issues regarding the parallel implementation of the Multipath execution model (MEM) are analyzed.

The implementation of the MEM is done by introducing some extensions to the Warren Abstract Machine (WAM) [1]. This modified WAM is called Multipath Abstract Machine (MAM).

The main difference between WAM and MAM is the existence of two types of variables: *single* and *multiple*. The former are the conventional variables used by WAM. These variables have a unique binding shared by all paths that can be simultaneously traversed. The latter are variables that may be bound to multiple values at the same time, each one corresponding to a different binding environment.

Single variables are stored as in the WAM: in the **environments** of the **STACK** or in the **HEAP**. A new memory area, called **MBE** (Multi-Binding Environment), is added in order to store multiple variables. A multiple variable has the same address in all the binding environments where it is visible.

The MAM instructions are the same as in the WAM but their semantics are slightly modified in order to manage multiple binding environments. Those instructions referencing a multiple variable must repeat its operation for every binding environment. The operations on each binding environment are performed by different Unification Engines (UEs, see section 3), which can work in parallel depending on the underlying hardware.

A parallel architecture that efficiently exploits the features of the Multipath execution model is shown in figure 2.

The system is based on a SPMD-like computer, and consists of a *Main Unit* (MU) where the Main Engine (ME) is executed. All data structures that are required by the MU are stored in the *Shared Memory*. There are also a number of *Unification Units* (UUs). Several Unification Engines (UEs) are mapped on each

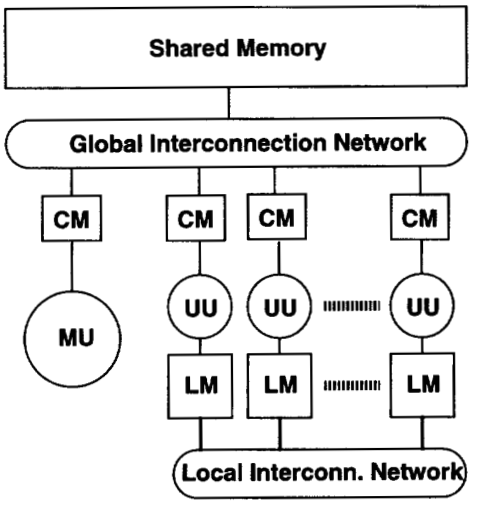

Figure 2: A parallel architecture for Multipath.

UU. We could allocate just one UE on each UU but, although this would produce a higher degree of parallelism, the system performance would be worse since path parallelism is fine-grained. Results about the performance obtained when the number of UEs per UU varies are shown in the next section.

The ME conducts the search of the SLD-tree. Every time it has to perform an operation on a multiple variable (like unification), it sends a command to all the UEs related to the binding environments currently active. As a result of it, the different UEs perform the same operation on different binding environments. Depending on the command, the ME waits until all the UEs have finished the command or continues traversing the search tree. For instance, if the command is to perform a unification, the ME waits because it must know which UEs succeed and which ones fail. However, if the command is to create a new free multiple variable, the ME continues as the command has no influence on the traversal of the search tree. Obviously, a UE cannot execute a new command until the previous one has finished. In each UU, there is a command queue where the commands sent by the ME are placed. These commands are processed in FIFO order.

The MU and the UUs communicate by means of shared variables in the Shared Memory. The *Global Interconnection Network* allows the MU and all the UUs to access to the Shared Memory. Single variables are also stored in the Shared Memory since they are manipulated by the MU and read by the UUs. The MU and each UU has a *Cache Memory* (CM) in order to speed-up the average memory access time.

Every UU has its own *Local Memory* (LM). Each LM contains the multiple variables belonging to the unification engines related to that UU. A given multiple variable may be visible for different unification engines and therefore for different UUs. However, each unification engine can only see one of its multiple values. In consequence, such variables can be allocated to several LMs but each LM stores only those values that are visible to the unification engines mapped onto that UU.

Eventually, all the multiple variables of a binding environment are required to be copied into another binding environment. As described in section 3, this happens when a backward operation is performed. A copy between two unification engines that are in the same UU is done locally. If the engines are in different UUs, it is done through the *Local Interconnection Network*. This is a high-speed network that just performs DMA transactions between two LMs.

5 Performance analysis

In order to isolate the different benefits of Multipath, we will start by analyzing the performance of a sequential implementation. That is intended to provide insight about the benefits of a partial breadth-first search in terms of reducing unifications and control overhead. Then, the additional benefits provided by path parallelism will be discussed.

The Multipath execution model has been implemented on a sequential environment by developing an interpreter of the abstract machine outlined in section 4. By tracing the sequential implementation, the execution time of the different threads suitable to be executed in parallel has been measured and these figures have been used to feed a simulator of the parallel machine described in the previous section in order to estimate the execution time in such parallel platform.

The simulator of the parallel machine makes some assumptions in order to keep the simulation cost reasonable. In particular, it is assumed no conflicts neither in the global interconnection network nor the shared memory modules.

An interpreter of the original Warren Abstract Machine, which implements the traditional depth-first search, has been used as a reference system to be compared with the Multipath execution model.

Using these interpreters, a set of benchmarks have been run on a DEC 3800 system whose CPU is an

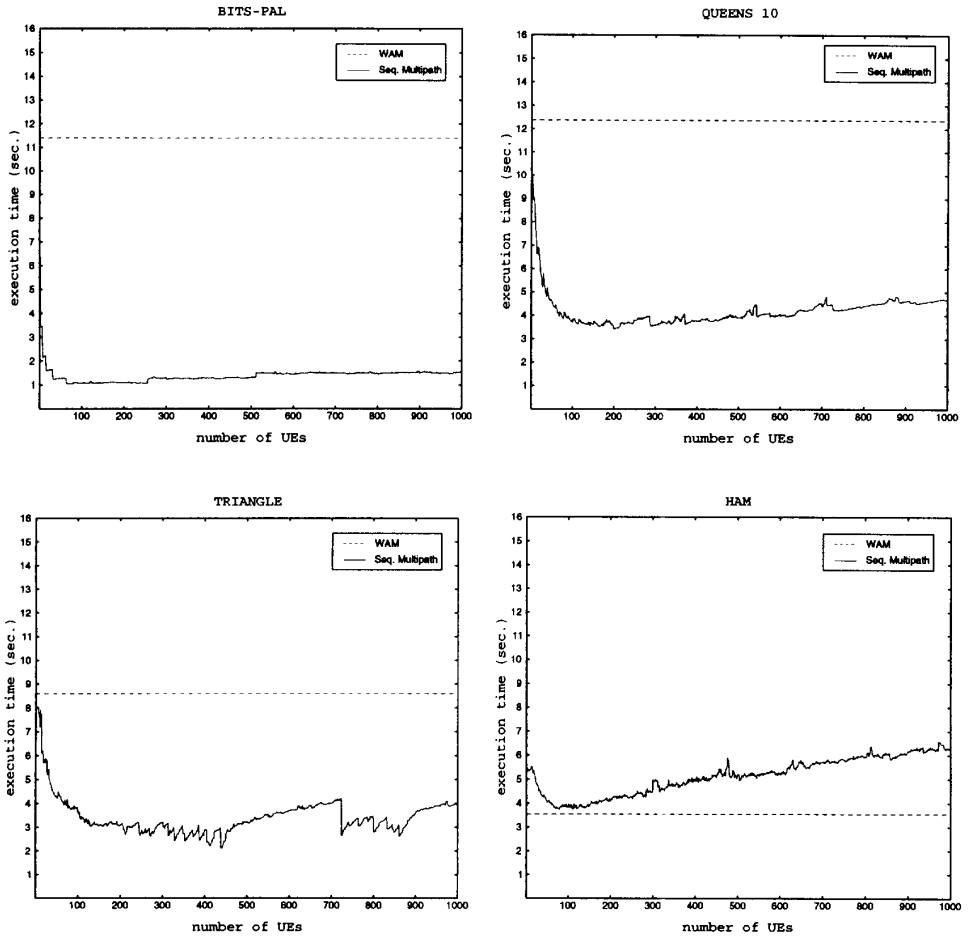

Figure 3: Sequential execution time for WAM and sequential Multipath.

Alpha 21064 microprocessor. The benchmarks are the following:

- **Queens10**: Finds all ways of placing 10 queens in a chess board without attacking among them.
- **Ham**: Finds all the hamiltonian paths on a graph.
- **Triangle**: This is the 'triangle' program (structure version, 133 solutions) of Evan Tick [10].
- **Cube**: Solves the Instant Insanity puzzle (6 cubes) from [10].
- **Zebras**: A logical constraint problem from [10].
- **Bits-pal**: Finds all palindromic lists of 17 binary elements using linear time reverse. It has been taken from [9].

5.1 Control overhead and unification cost

The benefits of a partial breadth-first search in front of a depth-first search in terms of reducing the control overhead and the number of unifications depend on the characteristics of the program. For very nondeterministic programs these benefits are expected to be high since, during the execution, Multipath will be able to traverse the SLD-tree carrying a high number of binding environments at the same time. As programs are more deterministic, the number of bind-

ing environments processed simultaneously decreases and in consequence, the reduction in control overhead and number of unifications is lower. Obviously, there is a minimum execution time for a sequential execution of Multipath, which corresponds to the amount of time needed for data computations to be performed on binding environments, plus the minimum time due to the breadth control.

On the other hand, the overhead introduced by the breadth-first search also depends on the type of program. This *breadth overhead* is mainly due to the copying of binding environments and the increase in the average memory access time caused by a bigger working set. Therefore, this overhead is higher as the size of the binding environments managed by the program execution increases. This overhead also depends on the number of unification engines because the number of times a binding environment is copied and the working set increase with the number of unification engines.

In addition to the program, the performance is also related to the strategy used to shift between depth-first and breadth-first search. Currently, it is done by having a maximum number of binding environments. Since each binding environment is processed by a unification engine (whether unification engines are processed sequentially or in parallel depends on the underlying hardware), this is equivalent to say that the execution model makes use of a maximum number of unification engines, which may vary from one execution to another.

Because of that, in first place the effect of varying the number of unification engines for different benchmarks is analyzed. Figure 3 compares the execution time of sequential Multipath and WAM, that is of a partial breadth-first search and a pure depth-first search. The results are presented for four of the benchmark programs and correspond to computing all solutions to them. The results for the other two benchmarks do not add significant differences. The benchmarks have been executed varying the number of unification engines from 1 to 1000.

In order to characterize the benchmarks, we define a *non-determinism metric* as the ratio of the average number of active unification engines over the maximum number of unification engines. This metric depends only on the benchmark, and remains constant when varying the number of unification engines.

The best results are for *bits-pal*, since it represents the type of programs where a partial breadth-first is most beneficial. This program is a typical example of a 'generate-and-test' program. Because of that, it is quite non-deterministic which results in a high utilization of the unification engines. Its non-determinism ratio is 63%. On the other hand, the binding environments are small and in consequence the breadth overhead is not much significant. As a result, we can see that Multipath is about 10 times faster than WAM with a quite low number of unification engines (about 70) and the performance is scarcely degraded when the number of unification engines increases.

Queens10 is more deterministic that bits-pal, with a non-determinism ratio of 8%. This causes that the the difference between the execution time of Multipath and WAM is not so high although still quite important (about 3 times). This also causes that the curve for Multipath goes down with a lower slope than for bits-pal, and in that case, the optimal number of unification engines is a bit higher (about 200). The size of the binding environments is also small and, like bits-pal, the degradation due to increasing the number of unification engines is almost negligible.

Triangle is even more deterministic, with a non-determinism ratio of 3%. In consequence, the optimal number of unification engines is also higher (about 440). Since binding environments are also small, the drawbacks due to increasing the number of unification engines are again almost negligible.

Finally, *ham* has the worst results. This program is quite deterministic (non-determinism ratio is equal to 7%), and therefore, the breadth-first search is quite limited. In addition, its binding environments are rather big, which results in a significant breadth overhead. In consequence, Multipath has about the same performance than WAM for about 120 unification engines, but if we increase the number of unification engines, we can observe a significant detriment in the performance. In this case, half the execution time is spent on binding environment copies. This has motived us to try to improve the implementation of Multipath by changing the copying of binding environments during its initialization to a copy on demmand of variable bindings. This is a work currently in progress.

Notice that the choice of an appropriate number of unification engines has a significant impact on the performance of the system. Because of that, we are also currently working on the development of a strategy that allows this number to be adapted dynamically, during the execution, to the requirements of the program. The implementation currently available uses a fixed number of unification engines for the whole execution of the program.

Another possible way to improve the system is by

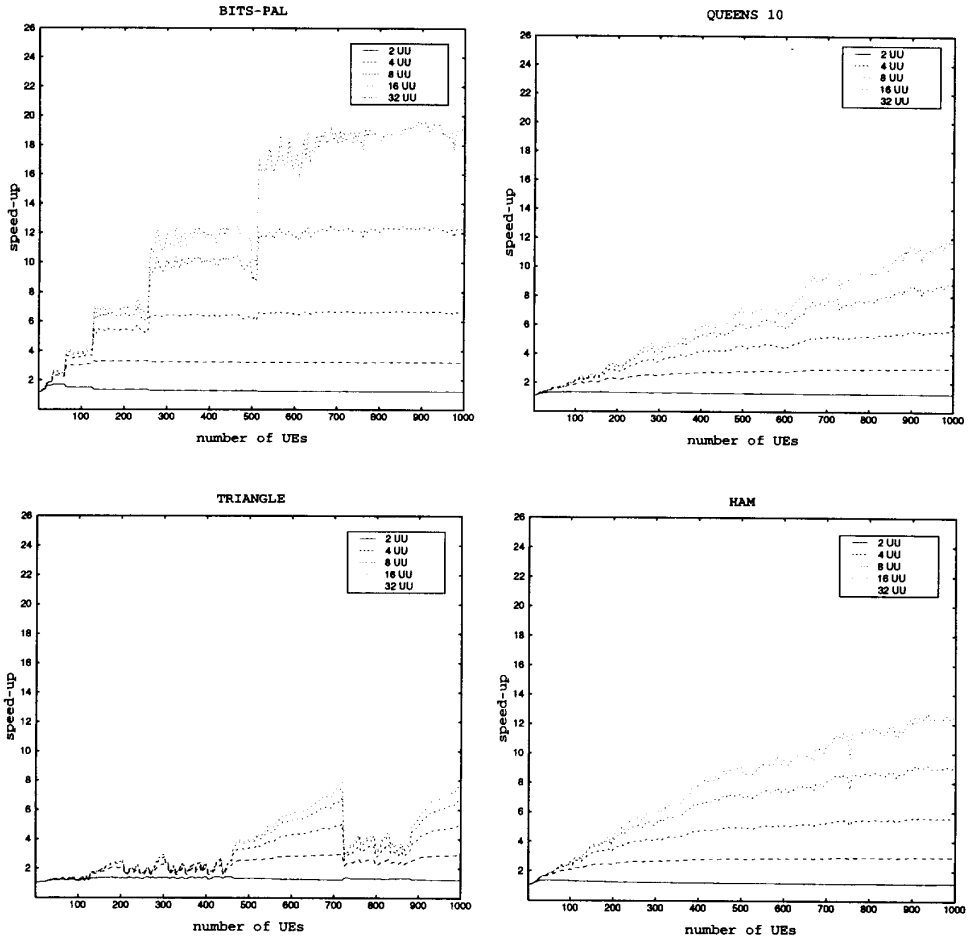

Figure 4: Speed-up of parallel Multipath.

doing a more careful analysis through abstract interpretation in order to determine which goals are more suitable for a breadth-first search and which ones are not. This is again a work currently in progress.

Table 1 summarizes the execution times of sequential Multipath and WAM for all the analyzed benchmarks as well as some significant measures as the optimal number of Unification Engines and the nondeterminism ratio. Notice that Multipath provides a significant improvement in execution time for most of the programs (it is several times faster than WAM) and in the worst case (ham), Multipath is only slightly worse than WAM.

5.2 Path parallelism

Figure 4 depicts the additional speed-up that can be obtained with the parallel architecture of figure 2. The results are shown for a different number of Unification Units (UUs) and a varying number of Unification Engines (UEs). Usually there are more Unification Engines than Unification Units. This is managed by allocating several Unification Engines to each Unification Unit. This has proved to be a good strategy because the amount of work to be performed by each Unification Engine is rather small. In this way, the granularity of the different parallel threads is increased. The results are presented for the same four benchmarks used in the previous subsection. The graphs

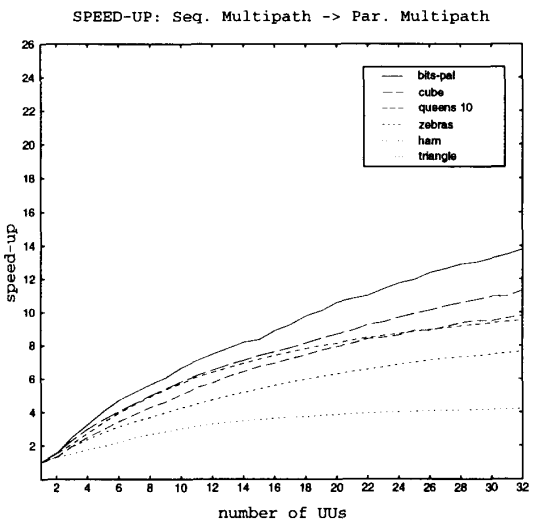

Figure 5: Speed-up of parallel Multipath for the optimal number of UEs.

correspond to the speed-up as it is usually defined: execution time with one MU and zero UUs divided by the execution time with one MU and N-1 UUs.

Notice in figure 4 that the speed-up clearly depends on the non-determinism ratio exhibited by the program. If this metric gets higher, then the speed-up increases. Also, in a parallel execution, increasing the number of UEs implies more potential parallelism to be exploited, in addition to the benefit of reducing the overhead of control instructions and unifications. This is why increasing the number of UEs respect to the sequential implementation is now beneficial.

Figure 5 depicts the speed-up for all the benchmarks varying only the number of UUs. For each number of UUs, the execution time for the optimal number of UEs is taken to compute the speed-up. Speed-ups are rather good, especially when the number of UUs is not very high. For example, with 8 UUs, the speed-ups for bits-pal, queens10, triangle and ham are 5.83, 4.05, 2.67 and 3.70, respectively. A higher number of UUs provides additional improvement in the execution time but the efficiency is reduced. This is due to the size and type of these benchmarks. For bigger programs and more non-deterministic programs we can expect a higher degree of path parallelism and in consequence, a better utilization of a higher number of unification units.

To conclude, table 1 summarizes the more significant results when analyzing the sequential as well as the parallel execution of all the benchmarks. They include the execution time (in seconds) of a WAM interpreter, the execution time and the optimal number of UEs of the sequential execution of Multipath, the speed-up or gain from WAM to sequential Multipath, the execution time and the optimal number of UEs of the parallel execution of Multipath with 8 UUs, the additional speed-up obtained by the parallel execution, and finally the total speed-up from a WAM execution to the parallel (8 UUs) execution of Multipath.

6 Conclusions

In this paper, a novel execution model for Prolog, which is called Multipath, has been presented. This execution model implements a partial breadth-first search of the SLD-tree. This search strategy has several advantages when compared with the traditional depth-first search:

- The overhead due to control instructions is reduced.
- The number of unification operations decreases.
- Introduces an inherent new type of parallelism, called path parallelism. This parallelism is a particular case of data parallelism and can be exploited quite efficiently since the amount of synchronization and data sharing that it requires is very low.

The paper analyses the benefits of Multipath by comparing it with the standard depth-first search implemented by the Warren Abstract Machine. For the analyzed benchmarks, and assuming that there are 8 Unification Units working in parallel, Multipath is about 63 to 4 times faster than WAM. The best results are for very non-deterministic programs, but even for more deterministic programs the results can be considered acceptable.

The conclusion is that a partial breadth-first search is more attractive than a pure depth-first search. In some sense, a partial breadth-first search introduces a new flexibility in the execution model: it allows the execution model to choose between a depth-first search and a breadth-first search for different parts of the SLD-tree. That is, the search strategy is dynamically adapted to the requirements of the execution.

The system presented here is a first version of such execution model. We believe that the results may

	bits-pal	*queens10*	*triangle*	*cube*	*zebras*	*ham*
WAM execution time	11.40	12.37	8.59	9.05	7.12	3.54
Seq. Multipath time	1.05	3.42	2.11	4.56	6.64	3.74
Optimal #UEs	70	200	440	65	75	120
Speed-up WAM to Seq.	10.96	3.62	4.07	2.01	1.07	0.94
Non-determinism ratio (%)	63	8	3	10	4	7
Par. Multipath (8 UUs) time	0.18	0.79	0.79	0.91	1.35	1.01
Optimal #UEs	230	1000	1000	585	710	560
Speed-up Seq. to Par.	5.83	4.05	2.67	5.01	4.91	3.70
SPEED-UP WAM to Par.	63.33	15.65	10.87	9.94	5.27	3.50

Table 1: Significant results for the analyzed benchmarks.

be still much better by improving several aspects like those outlined in the paper (i.e., copy on demmand of variable bindings instead of copy on initialization of binding environments, strategy to determine the optimal number of unification engines, scheme to determine the type of search, etc.).

References

[1] H. Aït Kaci. *Warren's Abstract Machine.* MIT Press, 1991.

[2] W. V. Citrin. Parallel Unification Scheduling in Prolog. UCB/CSD 88/415, Berkeley University, 1988.

[3] Antonio González and Jordi Tubella. The Multipath Parallel Execution Model for Prolog. In *Proceedings of the First Int'l Conf. on Parallel Symbolic Computation PASCO'94.* World Scientific Pub., 1994.

[4] P. Heuze. Using Data Parallelism in Elipsys. elipsys 003, ECRC, 1989.

[5] P. Kacsuk. DAP Prolog: A Parallel Array Extension of Prolog. In *Proceedings of CONPAR'88.* British Computer Society, 1988.

[6] R. A. Kowalski. Predicate Logic as a Programming Language. In *Information Processing 74, Stockholm*, pages 569–574. North-Holland, 1974.

[7] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[8] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.

[9] D. A. Smith. Multilog: Data Or-Parallel Logic Programming. In *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, 1993.

[10] Evan Tick. *Parallel Logic Programming.* MIT Press, 1991.

[11] Jordi Tubella and Antonio González. MEM: A New Execution Model for Prolog. *Microprocessing and Microprogramming*, 39:83–86, 1993.

[12] Jordi Tubella and Antonio González. A Partial Breadth-First Execution Model for Prolog. In *Proceedings of the 6th Int'l Conf. on Tools with Artificial Intelligence TAI'94*, To appear.