

# Independent Subexpressions Parallelism with Delayed Synchronization for Functional Logic Languages

Andrés del Pozo-Prieto                      Juan José Moreno-Navarro  
Universidad Politécnica de Madrid, Facultad de Informática  
Campus de Montegancedo s/n, Boadilla del Monte 28660 Madrid, Spain  
e-mail: andres@judith.ls.fi.upm.es                      jjmoreno@fi.upm.es

**Abstract:** The paper describes the implementation of a parallel graph narrowing machine. The machine implements a functional logic language on a shared memory multiprocessor. The model of parallelism uses an independence condition to decide which subexpressions of an expression can run in parallel. The outer expression continues its execution and the synchronization with a subtask is only performed when its result is needed. The machine is fully implemented and some experimental results are shown.

**Keywords:** Logic Programming Parallelization, Independent And-Parallelism, Functional Logic Languages

## 1 Introduction

Since the initial efforts in reconciling the styles of functional programming and logic programming, there has been significant work done regarding the design of combined languages and their efficient implementation.

Several *functional logic languages* [MR92], [BCM90, Ha94] have been proposed trying to add the advantages of both paradigms. They have functional syntax and *narrowing*, an evaluation mechanism that uses unification for parameter passing, is used as operational semantics [Re85]. As an instance of this class of languages we will use along the paper the syntax of the functional logic language BABEL [MR92], although it is clear that the techniques are applicable to many other languages. Besides the fact that functional logic languages are more expressive than pure functional and logic languages by themselves, the use of functional expressions allows the compiler to extract more information from the source code. The implementation can use several optimizations that are not possible, in principle, in logic languages, as the dynamic detection of deterministic computations [LW91].

Thus, one of the natural ways of achieve an implementation is to extend functional programming techniques by borrowing the *logical* components from Prolog implementations (mainly from the Warren Abstract Machine -WAM- [Wa83], [AK91]). *Graph narrowing machines* extend graph reduction machines, designed for the implementation of functional languages (see, for instance, [PJ87] for a survey of the extensive literature in this area), in such this way. They have been used for the sequential implementation of BABEL [KLMR90, KLMR92, MKLR90]. One advantage of the functional logic language approach, is that it guarantees that purely functional programs can be executed almost as efficiently as in the original functional machine, although some overhead due to the different parameter passing mechanism cannot be completely avoided.

We are interested in the parallel implementation of (lazy) functional logic languages. Functional logic languages can exploit the same kind of parallelism that logic languages but we can take advantage from the information of the structure of the program. A merely combination of existing techniques is a simple approach but in

order to obtain benefits from the integration of paradigms we believe that new models and techniques for the parallelization are needed. Of course, the task is more complicated but highly desirable properties can be provided. The paper presents a further step in this direction, that have been started in [KMH92].

There are three main principles in the design of our execution model:

1. *No slow-down*: The implementation must ensure that no more work than in the sequential execution is done.
2. *Subexpressions parallelism*: The exploited parallelism allows subexpressions of an expression and the outer expression itself to run in parallel, if an independence condition holds. Essentially, we can consider it as a generalization of independent and-parallelism from logic programming [He86, De84, HR90], if we look at the conjunction as the only "function" that combines subexpressions. As in [LKID89], we allow several subexpressions  $e_1, \dots, e_n$  to run in parallel with the outer expression  $e$  that uses them. The degree of parallelism is increased:  $n + 1$  tasks (instead of  $n$ , where the extra task could in turn generate more parallel tasks) run in parallel.
3. *Delayed synchronization*: The synchronization on the parallel task is done when another task needs its result. Only the *head normal form* (free variable or topmost constructor) is needed to perform the synchronization and the arguments can still run in parallel without performing the synchronization yet. At this point we are using the information of functional dependencies. This model is quite different from the usual one in independent and-parallel implementations of Prolog. In most such implementations, and due to the lack of information on directionality of arguments in logic programs (unless extensive global analysis is performed or the program is annotated), a computation  $e$ , which combines the results of some parallel computations  $e_i$ , has to be placed and evaluated after them.

The model uses a producer-consumer scheme: a task computing an expression  $e$  produces the result and the synchronization is done when a task needs its result, consuming (a part of) it. The approach shares some similarities with the Andorra model [SWYH91] in the sense that some

(deterministic) expressions are executed in parallel before the others. However, our model differs from it in several points, for instance in the delay of synchronization, as we will discuss later.

The parallel abstract machine is fully implemented and some experimental results are reported. Its design combines some techniques from parallel graph reduction machines (see, among others, [PJ87, DR81, PCH87, LKID89]) and ideas from the parallel implementation of logic programming (see [De84, He86, HG90, HR90]). Our approach is part of a more ambitious project: the design and implementation of a parallel model for the execution of lazy functional logic languages. By the moment, the generation of tasks is eager but the synchronization is lazy. We have also simplified the detection of parallelism and synchronization.

The rest of the paper is organized as follows: we first introduce functional logic languages (Section 2). Then, Section 3 presents the independent subexpressions parallelism with delayed synchronization model. The concrete implementation is discussed in Section 4, while experimental results are reported in Section 5. We conclude by comparing our work with some existing ones (Section 6) and pointing out future work (Section 7).

## 2 Functional Logic Languages

Due to the lack of space we will present the functional logic language BABEL in an informal manner. The interested reader can consult [MR92].

The language BABEL integrates functional and logic programming in a flexible and mathematically well-founded way. It is based on a constructor discipline and uses narrowing as evaluation mechanism [Re85].

In order to give a flavour of the BABEL syntax we will start with a small example, computing the reverse of a list in the naive way.

```
datatype list A := nil | (cons A (list A)).
```

```
fun app: list A × list A → list A.
  app ([ ], Ys) := Ys.
  app ([X|Xs], Ys) := [X|app (Xs, Ys)].
```

```
fun nrev: list A → list A.
  nrev ([ ]) := [ ].
  nrev ([X|Xs]) :=
    app (nrev (Xs), [X]).
```

BABEL is a typed language with a Hindley-Milner polymorphic type system. A datatype definition allow to define new (polymorphic) types as the type list  $A$ . A Prolog-like syntax for lists is allowed, i.e.  $[e \mid e']$  is equivalent to “cons ( $e$ ,  $e'$ )”,  $[e, e']$  represents “cons ( $e$ , cons ( $e'$ , nil))” and  $[]$  is the empty list “nil”. A BABEL *program* consists of a sequence of datatype definitions and a sequence of function definitions. For each function, its type and a sequence of defining rules are specified. A *rule* for a function  $f$  has the form

$$\underbrace{f(t_1, \dots, t_m)}_{\text{left hand side}} := \underbrace{\underbrace{\{b \rightarrow\}}_{\text{optional guard}} \underbrace{e}_{\text{body}}}_{\text{right hand side}}$$

where  $t_1, \dots, t_m$  are (left linear) terms,  $e$  is an arbitrary expression using the variables of  $t_1, \dots, t_m$  and  $b$  is a boolean expression which can use new free variables<sup>1</sup>. A *term* is either a (logical) *variable* (starting with a capital letter), a constant, or a data constructor application.

BABEL expressions are made from well typed applications of user defined functions, primitive symbols, or constructors to other expressions. We assume the following primitive functions:  $\neg b$  (negation),  $(b_1, b_2)$  (conjunction),  $(b_1 ; b_2)$  (disjunction),  $(b \rightarrow e)$  (guarded expression, meaning: **if**  $b$  **then**  $e$  **else** undefined),  $(b \rightarrow e_1 \square e_2)$  (conditional, meaning: **if**  $b$  **then**  $e_1$  **else**  $e_2$ ), and  $(e_1 = e_2)$  (weak equality, both expressions denote the same object), where  $b$  is a boolean expression.

Although BABEL allows higher order functions (see [KLMR92]) we restrict in this paper to the first order part of the language for simplicity.

BABEL functions are functions in the mathematical sense, i.e. for each tuple of (ground) arguments, there is only one result. This is guaranteed by some syntactic restrictions into program rules ensuring *incompatibility* (see [MR92, KLMR92]) between two rules for the same function.

The above program looks like a functional program. However, in BABEL a program can be queried with a goal expression with free variables.

$$\text{eval nrev}([1, 2 \mid L]) = [X, Y, Z, W].$$

which would return the answer **yes** under the variable bindings  $L = [Y, X]$ ,  $Z = 2$ ,  $W = 1$ .

This goal can be computed by using *narrowing*. An expression  $e$  is narrowed by applying

the minimal substitution that makes it reducible, and then reducing it. The minimal substitution  $\sigma = \sigma_{\text{in}} \dot{\cup} \sigma_{\text{out}}$  is found by unifying  $e$  with the left hand side of a rule. A new expression  $e'$  and the *answer substitution*  $\sigma_{\text{out}}$  binding some variables from  $e$  are the *outcome* of one narrowing step (denoted by  $e \Rightarrow_{\sigma_{\text{out}}} e'$ ). A *computation* from the *goal*  $e$  to the *result*  $t$  with *answer*  $\sigma$  is reached when an expression  $e$  is narrowed to the term  $t$  in several narrowing steps, where  $\sigma$  is the composition of the answer substitutions of all the individual steps. The computation fails when  $e$  cannot be narrowed further, but it is not a term. The implementation will backtrack in such a situation. The rules are tried in their textual order.

A narrowing sequence for the goal  $\text{nrev}([1 \mid L]) = [2, Z]$  in the previous example is:

$$\begin{aligned} \text{nrev}([1 \mid L]) = [2, Z] & \Rightarrow \\ \text{app}(\text{nrev}(L), [1]) = [2, Z] & \Rightarrow_{\{L=[X|Xs]\}} \\ \text{app}(\text{app}(\text{nrev}(Xs), [X]), [1]) = [2, Z] & \\ & \Rightarrow_{\{Xs=[\ ]\}} \\ \text{app}([X], [1]) = [2, Z] & \Rightarrow \\ [X \mid \text{app}([], [1])] = [2, Z] & \Rightarrow \\ [X, 1] = [2, Z] & \Rightarrow_{\{X=2, Z=1\}} \\ \text{true} & \end{aligned}$$

giving *true* as result and the answer  $L = [2]$ ,  $Z = 1$ .

### 3 Independent Subexpressions Parallelism with Delayed Synchronization

An (intermediate) goal usually contains several subexpressions, which can in principle be narrowed in parallel. In the above example, the expression  $\text{app}(\text{nrev}(Xs), [X])$  contains two function calls  $\text{app}(\dots)$  and  $\text{nrev}(Xs)$ . It is clear that it is not always advisable to narrow subexpressions in parallel: One problem is “granularity”: it is not always the case that the evaluation of an expression in parallel (i.e. spawning a process) is faster than its sequential evaluation. Even if two expressions are of adequate granularity, the existence of dependencies among the expressions to be executed in parallel can complicate the execution and, specially, the backtracking mechanism which would be a source of large overheads.

For this reason, the syntax of BABEL is extended with a special construction, called *letpar-expression*, which allows to express the desired parallelism:

<sup>1</sup>These variables are implicitly existentially quantified

**letpar**  $X_1 := e_1 \ \& \ \dots \ \& \ X_n := e_n$  **in**  $e$

where  $e_i$  has to be an application of a defined function to terms ( $1 \leq i \leq n$ ) and  $e$  may contain the new auxiliary variables  $X_1, \dots, X_n$ .

The user (or a clever compiler) can determine which expressions are complex enough, to do so. Of course, it is desirable to have an automatic parallelization of the program. It is possible based on techniques capable of predicting the granularity of the subexpressions, and the proposed language can serve as the intermediate language to which these tools translate (as it is done in the  $\&$ -Prolog system [HG90]).

The reading of a **letpar**-expression is that  $e_1, \dots, e_n$ , and  $e$  (!) will be narrowed in parallel. The first use in  $e$  or an outer expression of a variable  $X_i$  causes a synchronization with the task  $e_i$ : the execution has to wait (suspension) until the corresponding process  $e_i$  has *head normal form* (*hnf*), i.e. a free variable or a term with known topmost constructor as result, (synchronization), at which point the execution of both tasks can continue (reactivation). In our previous paper [KMH92] this delayed technique was used by in a syntactic way: as soon as  $X_i$  is syntactically used the synchronization is done. Now, we use a dynamic rule: only an access to  $X_i$  to examine its contents causes the synchronization. No synchronization is done if  $X_i$  is stored in a data structure (a list for instance).

Notice that no more work than in the innermost sequential case is done, because the subexpressions  $e_i$  of an expression  $e$  need to be narrowed before the execution of  $e$ . Now, the subexpression are again computed but in parallel with the outer expression. In the case of successful computations, the *no slow-down* property is preserved. For failing expressions, we need a dedicated scheduling policy, which gives priority to the leftmost subexpression.

Now we need to fix when a set of expressions can run in parallel. In our previous paper [KMH92] we have imposed the expressions to be *independent*, i.e. they do not share unbound variables, borrowing the ideas developed for independent and-parallelism in Prolog [HR90] (ensuring correctness and “no slow-down” properties for parallel execution). Now we are a bit more restrictive in order to simplify the model: “When an expression  $e$  is running in parallel the use of a variable  $Y$  that will be bound by  $e$  is forbidden”. The notion is not always easy to check and some particular cases of the previous statement can be fixed. For instance, the property is trivially true if  $e$  is ground. Even if  $e$  is not ground, the property holds if the free variables of  $e$  are not

bound during the execution of  $e$ . This last condition allows very interesting cases, for instance the use of accumulative parameters or difference lists. Although, it is not exactly the notion of “independency” in logic programming, the name also defines correctly this condition and we will still use it. We also request every parallel expression to be “deterministic”, so backtracking does not affect its relation with the other expressions. It is valid in the previous particular cases.

In order to show our model of parallelism, let us consider, for instance, the goal  $[X,Y,3,4] = \text{app}(\text{app}([1,2],[Z]),W)$  with the **app** function from the example of section 2, giving the result **true** and bindings  $X = 1, Y = 2, Z = 3, W = [4]$ . We can parallelize the function and the goal in the following way:

```

app ([ ], Ys) := Ys.
app ([X | Xs], Ys) := letpar
                       R := app (Xs, Ys)
                       in [X | R].

```

```

eval letpar L := app ([1,2], [Z])
in [X,Y,3,4] = app (L, W).

```

The example involves logical variables, but several call for **append** are done in parallel, as it is shown in figure 1, where the parallel execution is drawn.

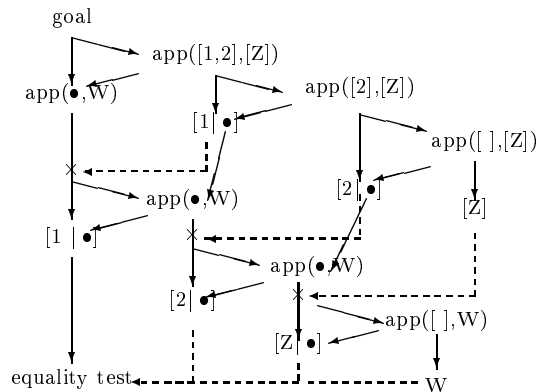


Figure 1: Parallel trace of the example program

æ

The symbol  $\times$  and the stroke line are used to denote a synchronization point. For instance, the leftmost synchronization is done during the unification of the first arguments of **app** ( $L, W$ ) (i.e. the result of **app** ( $[1, 2], [Z]$ )) with  $[ ]$  or  $[X' | X'_s]$ . Notice that only the *hnf* of  $L$  is required for the synchronization.

As we have mentioned, the imposed condition has an additional advantage: backtracking is not needed, and the model is simplified. It is

worth mentioned that the delayed synchronization technique does not depend on this condition as (partially) shown in [KMH92]. In fact, when a variable  $Y$  “belonging” to a task  $e$  is needed by another task a synchronization on this variables can be done, waiting until  $e$  binds the variable.

Backtracking is allowed when the task is not involved in a parallel execution. In other words, the parallel part cohabits with a logical part, that includes backtracking.

## 4 Implementation

In this section we are going to describe an experimental implementation of the model of parallelism introduced in the previous section. We have used an *abstract machine* since this implementation method provides the best combination between flexibility and performance.

Our family of *Babel Abstract Machines (BAM)* [KLMR90, KLMR92, MKLR90] are *graph reduction abstract machines*. The main characteristic of this kind of machines is the using of a graph for storing the state of the computation instead of a set of stacks like those machines derived from the *Warren Abstract Machine* [Wa83]. Graphs are implicit parallel data structures so the parallelization of an abstract machine based on graphs is easier than stack based ones, avoiding technical difficulties like stack trapping.

The BAM's are experimental machines so they have been designed for easy modifications and extensions. Their modular design sacrifices part of the performance while gaining a much shorter development cycle. First we briefly describe the (last version of the) sequential *innermost Babel abstract machine (IBAM)* to introduce later the parallel extensions and how delayed synchronizations are implemented.

### 4.1 BAM - Babel Abstract Machine

Due to the lack of space we will not describe the machine in detail. Instead, we will only discuss some important points to understand the parallel extension. The main components of a BAM are the program store, the graph and the interpreter of instructions. The BAM stores its state in the graph which is composed by different types of nodes that represent the data and control of an evaluation. These nodes are served on demand by a *dynamic memory manager* and no longer useful nodes are reclaimed by a *lazy reference counting* garbage collector. The goal of the memory manager is to provide the highest possible memory bandwidth while reducing the frag-

mentation. The memory manager asks the page manager for pages from a pool of preallocated pages waiting to be employed. When the pool of pages is exhausted the page manager can allocate dynamically more pages. Pages contain nodes of the same size to reduce the fragmentation of the memory. The lazy garbage collector allows an immediate reclaim of the memory wasted by discarded terms with a constant (and low) cost in time.

*Task nodes* represent the state of a function (or predicate) call. The forward and backward flow control is kept as implicit stacks made with pointers, stored as fields in the task nodes, pointing the involved task nodes. This is the main difference with WAM based machines and allows to decouple the components of the abstract machine obtaining a more modular design. This technique is less efficient than stacks but we think that performance losses are smaller than the benefits (at least for an experimental implementation). Small nodes like constants, references and free variables can live inside other nodes. This is similar to the permanent and global variables in the WAM [AK91] and shares similar advantages/disadvantages.

In Prolog (and the WAM), predicates only return success or failure as result, while in a functional logic language the result can be a term or a failure. So, the BAM must provide a method to allow functions to return a value.

The abstract machine allows some optimizations. The instruction set allows the use of improved clause indexing thanks to the typed nature of Babel. Since constants and constructors have a fixed type, perfect hashing (as in the WAM) is not needed and a simple lookup table can keep the jump addresses. Tail recursive functions and predicates are optimized with the *last call optimization* that reuses the same task node for the next tail recursive call. The machine is also capable to detect some deterministic computations and a more efficient code (avoiding backtracking) is generated.

### 4.2 PBAM – Parallel Babel Abstract Machine

The original design of the PBAM can be found in [KMH92]. The current implementation does not handle backtracking yet but this limitation is not too restrictive since and-parallelism mainly accelerates forward execution. However, a full implementation with backtracking is under development.

The implicit distributed structure of graphs

renders an immediate parallelization of the data structures so the parallelization of the BAM mainly consists in the parallelization of the control. Most of the BAM components need to be moved to reentrant components to allow simultaneous operations.

The structure of the PBAM is quite similar to the BAM but it has a new component: the *Task Scheduler (TS)*. The PBAM also has several execution units with an *interpreter of instructions (II)* for each one. Each interpreter can access a *memory manager and garbage collector (MMGC)* and several interpreters can share the same *MMGC* although the best performance is achieved with the same number of *II* and *MMGC*. There is only a *parallel page manager* since the low number of accesses makes collisions unlikely.

We have used a coding technique called *critical region snooping* in order to minimize the number of locks and collisions between concurrent execution units. When an execution unit wants to access a shared structure a reading of a flag (part of the structure) is done firstly. The flag determines whether the access is allowed or interrupted. If the access is actually done it is necessary to lock the data structure and re-consult the flag to be sure of its value. *Critical region snooping* requires that the flag would have a safe value and that its updates would be done in a safe way.

The *IIs* ask the *TS* for tasks ready to be executed. The *TS* has multiple entry points to avoid bottlenecks. As there are no constraints in the order of dispatching ready tasks, it is possible to choose for the *TS* between many correct policies. We have chosen one of the simplest policies since it renders a good performance: The *TS* is a set of stacks, one for each *II*. Tasks generated by an *II* are stored in its own stack. When a *II* asks for a task, it firstly looks into its stack. If it is empty it looks (using critical region snooping) for the non-locked stack with the highest number of tasks. The policy allows a good load balancing and a low number of collisions in the *II* access.

### 4.3 DPBAM – Delayed Synchronization

The ability of the delay of synchronizations is a direct extension of the previous design of the PBAM. Remember that synchronization is only allowed on the results of function calls. Unevaluated terms are represented by the so called *synchronization variables*. Synchronization variables can be used as normal *reference variables* except that a task trying to consult its value is forced to

wait. Synchronization variables are implemented as a new node type that can be used like other nodes to construct terms. These nodes are created in the moment of the creation of parallel tasks which are going to have delayed synchronization. When the parallel task has computed the term which is going to return, the synchronization variable is moved into a reference, pointing to the new term. Since the synchronization variable could be part of other terms the changes are immediately propagated. If a parallel task does not involve delayed synchronization, the return of results is done as in the PBAM to reduce the overhead.

A delayed synchronization is actually done when the result of the evaluation is needed. This situation happens when a term is unified with another term, when it is involved in an arithmetic operation or when it is going to be output. In the first case (unification), the term must be in *hnf*, so only the top level node is checked to be a synchronization variable. The task must wait until this synchronization variable has a concrete value. In the other two cases the task must wait until the term is in *normal form (nf)*. Since programs are well typed, there is no difference between *hnf* and *nf* for the arguments of arithmetic expressions.

The tasks waiting for a result are stored on a linked list, kept as a special component of task nodes. The synchronization variable that represents the unevaluated result has a field with a pointer to the head of the list of waiting tasks.

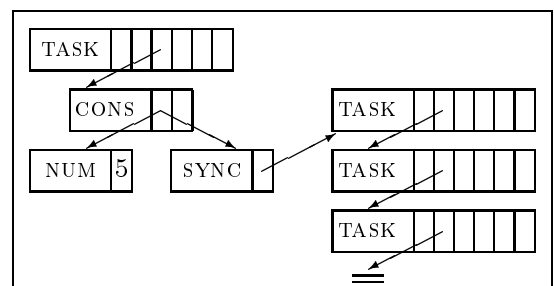


Figure 2: A synchronization variable.

## 5 Experimental Results

We have implemented the design of the DPBAM on a Sequent Symmetry computer with ten processors. The implementation is based on a previous implementation of the BAM that runs at similar speed as some WAM based Prologs. The implementation has been developed in C with the

GNU compiler and uses some of its non-standard features.

We have tried only some small programs<sup>2</sup> that show promising results. We report the result of a couple of programs with delayed synchronization.

```

sieve (2) := [2].
sieve (N) := letpar
    R1:= sieve (N-1) &
    R2:= check (R1, N)
in R2.

check ([X],N) := N mod X = 0 → [X] □ [X,N].
check ([X|Xs], N) := N mod X ≠ 0
    → [X|Xs]
    □ letpar
        R:= check (Xs, N)
    in [X|R].

eval sieve (1000).
    
```

Figure 3: Sieve of Eratosthenes.

The first program (figure 3) is the Sieve of Eratosthenes algorithm. This algorithm is not parallelizable with the standard *Independent And parallelism* model (or at least is not parallelizable in a simple way). However with our model the parallelization is achieved quite easily. The second example, Naive reverse (figure 4), inverts the order of the elements of a list using the append function.

```

nrev ([]) := [].
nrev ([X|Xs]) := letpar
    A := nrev (Xs) &
    B := app (A, [X])
in B.

app ([], L) := L.
app ([X|Xs], L) := letpar C := app (Xs, L)
in [X|R].

eval nrev ([1,2,3,4,5, ... ,150]).
    
```

Figure 4: Naive Reverse.

The execution times of both programs are shown in the tables 1 and 2. Both tables show the result of running the programs on the DPBAM allowing delayed synchronization, DPBAM without delayed synchronization and PBAM. All the times are measured in seconds and include

<sup>2</sup>At present we have not developed a BABEL to BAM-code translator

Processors	DPBAM		PBAM
	delay	sync	
1	7.76	7.04	6.19
2	3.88	-	-
4	2.31	-	-
5	1.68	-	-
6	1.49	-	-
10	1.37	-	-

Table 1: Sieve of Eratosthenes (1000)

garbage collection time but not output time. Notice that the programs are not parallelizable without delayed synchronization, so the times shown for the PBAM and DPBAM without delayed synchronization uses only one processor.

The overhead imposed by the inclusion of the delayed synchronization is about a 10% for programs that do not use the delayed synchronization (5% for *naive reverse* and 13% for *sieve*). Programs using delayed synchronization have some overhead but it is counterbalanced with the parallelism gained. Part of this overhead is caused by the excessive number of parallel tasks generated and it could be reduced by using *task granularity control* [DLH90].

æ

The figures 5 and 6 represents the speedup achieved by the delayed synchronization model with respect to the standard model. The horizontal axis shows the number of processors and the vertical axis the ratio PBAM/DPBAM. For one processor the overhead taxes the results for the DPBAM but the speedup increases peaking at about 4.0 for ten processors. These times are

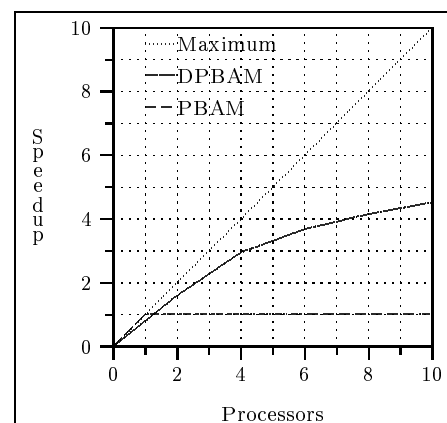


Figure 5: Sieve of Eratosthenes (1000)

better than SICStus Prolog 2.1 in the same computer and we think that the overhead can be reduced.

Processors	DPBAM		PBAM
	delay	sync	
1	5.10	3.45	3.27
2	2.71	-	-
4	1.52	-	-
5	1.17	-	-
6	0.95	-	-
10	0.87	-	-

Table 2: Naive Reverse (150).

## 6 Related Work

Although there is a lot of work on the parallelization of declarative languages, we will only review some directly related work.

Out of the BABEL environment, there is only one significant work devoted to the parallel implementation of integrated languages: the implementation of the language K-LEAF by the KWAM [BCM90]. A parallel execution model has been proposed and implemented. However, only or-parallelism is exploited. Its abstract machine is an extension of a logic abstract machine, rather than a functional one. In some sense, the solutions applied to BABEL are different and complementary to those presented for K-LEAF. Moreover, since there is also backtracking in BABEL or-parallelism can also be exploited by extending our machine with either the environment sharing or the environment copying techniques used in the combination of and- and or-parallelism in Prolog, or, perhaps, with the techniques used in K-LEAF.

æ

There are some other proposals for parallelization of functional logic languages in the context of the BABEL language. [KH91] implements dependent And-parallelism. All parallel processes first mark all the variables which they can access, but only one of them is the owner. Only the owner is allowed to bind it. When a different process wants to bind a variable, it has to wait, until the owner process is ready. Usually, the extended synchronization in such systems imposes more overhead. It is not clear at this point whether this overhead would be worthwhile. In any case, most of the techniques presented here can be adapted to this approach.

Of course, the more related work is [KMH92] because we are inspired on it. Both in [KMH92] and our approach the synchronization with a task

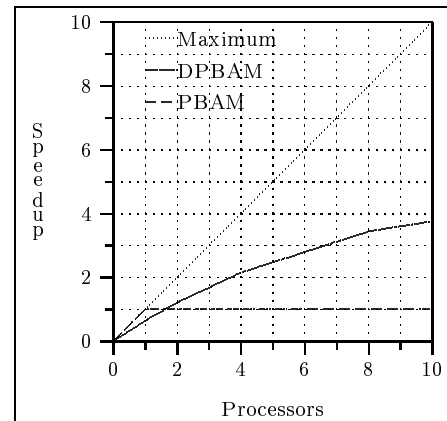


Figure 6: Naive Reverse (150)

is delayed until its result is needed. The main difference is that here this notion is dynamic while in [KMH92] is static. Certainly, the use of a variable does not mean that its value is needed. A simple static notion restricts the parallelism while the dynamic one does not. For instance, in the `app` function the tail of the list (a recursive parallel call) is needed to construct the resulting list, but it does not matter the exact content of it. Synchronization variables are managed in such a way that this dynamic decision can be done with little overhead. Furthermore, this paper presents the first (partial) implementation of [KMH92] showing that the ideas of subexpression parallelism and delayed synchronization are feasible.

A recent paper [HRSW94] introduces some techniques of independent and-parallelism Prolog into a stack-based implementation of BABEL.

In the field of logic programming, a lot of effort has been invested on the parallelization of Prolog. In particular, our model can be extended to cope with similarities from the independent and-parallelism found in Prolog [De84, He86, HG90, HR90] and exploited in  $\&$ -Prolog. As we have said, the schemes differ in the creation and synchronization of tasks. In  $\&$ -Prolog the processes for some expressions  $e_1 \& \dots \& e_n$  are not linked by an expression  $e$  in a `letpar` expression. Rather, the evaluation of  $e$  would have been placed after that of  $e_1, \dots, e_n$ , so the synchronization is not delayed. Most of the development in compile-time techniques could be extended to cope for the model presented in this paper.

Maybe, the most related approach to our work is done in the context of the Andorra model and languages. The *Andorra* model was designed by D.H.D. Warren in order to support both depen-

dent and-parallelism and or-parallelism. Goals that are deterministic can be executed concurrently. When no such goal exists, a don't know nondeterministic step is performed in some of the goals. In our model, the expressions running in parallel are deterministic, but this restriction can be eliminated. The Andorra model can be combined with our system in the spirit of the combination of the Andorra-I and &-Prolog systems presented in the IDIOM model [SWYH91]. An additional advantage is that in a functional logic language it is more easy to detect determinism.

The idea of running  $e_1, \dots, e_n$ , and  $e$  in parallel, and to synchronize  $e$  and  $e_i$ , when  $e$  needs the result of  $e_i$ , comes from the functional programming field, where a lot of work on parallelization has been done (see [PJ87, DR81, PCH87, LKID89]). However, our model includes unification and logical variables, and the sequential part (and the parallel part in the future) support backtracking. A functional language with logical variables (but no backtracking) appears in [Li87]. The paper extends a distributed graph reduction machine to allow for logic variables, while preserving lazy evaluation and determinacy.

## 7 Conclusion and Future Work

The paper presents some techniques for the parallel implementation of functional logic languages, by integrating mechanisms used in functional and logic programming implementations. The model discussed in [KMH92] is extended with a dynamic notion of delayed synchronization. If the compiler is responsible for placing the synchronization of subexpressions in the (static) code the solution must be to synchronize before the execution of the outer expression (as in &-Prolog) or, when the result is (syntactically) used (as in [KMH92]). In both cases, the potential parallelism is reduced. Synchronization variables allow for synchronization when the result is accessed, what increases the parallelism.

Furthermore, the model preserves the *no slow-down* property, ensuring that the parallel version does not produce more work than the sequential implementation. Even more, our system can execute less work than in the sequential case. When the computation finishes (either with success or a failure), all the tasks that are still running are killed. For those expression that are not used in the final result, the system can avoid their complete execution. In such these cases, the implementation behaves similarly to lazy evaluation.

A complete prototype is available. Despite the fact that several significant optimizations are

still to be implemented, the results, as we hope to have shown, are quite encouraging. They show that delayed synchronization is an interesting and applicable technique for the parallel implementation of functional logic languages.

As a future work, we plan to incorporate backtracking in our model. This supposes to relax our independency condition in order to allow multiple results of the subexpressions in parallel. The extensions, although not completely trivial, can be carried out following the ideas of [KMH92]. Again, the functional structure of programs can be used to improve the model. When some subexpression  $e_i$  running in parallel produces a new solution, only the work made after the use of this  $e_i$  needs to be redone, while for several reasons in Prolog implementations the evaluation of  $e$  generally restarts from the beginning. Also, *pseudo-intelligent backtracking* as in [HN86] can be included. Moreover, it is possible to keep some results of the parallel tasks, avoiding reevaluations. As we have mentioned, the implementation can deduce, in some cases, that a computation is deterministic. If it is involved in a parallel execution, it is not affected by backtracking. Its result is kept and can be used later on, when the re-execution of the previous expressions is successful and the value is needed again.

We also want to work on the compilation of BABEL programs to our machine code. The automatic parallelizer will reuse some techniques proposed for the parallelization of functional and logic programs (detection of independence, granularity analysis, etc.).

Finally, our ultimate aim is to design and implement a lazy parallel model for the execution of functional logic languages. The main work to be done is the lazy creation of task, because the synchronization is already lazy. The ideas presented in [MKLR90, MKMWH93] should be the basis for this work.

## Acknowledgements

This research was supported in part by the spanish project TIC/93-0737-C02-02.

## References

- [AK91] H. Ait-Kaci: The WAM: A (Real) Tutorial, The MIT Press, 1991.
- [BCM90] P. Bosco, C. Cecchi, C. Moiso, M. Porta, G. Sofi: Logic and Functional Programming on Distributed Memory Architectures, Proc. 7th ICLP, 325-339, 1990.

- [DR81] J. Darlington, M. Reeve: ALICE - a Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages, Proc. ACM FPCA, 65-76, 1981.
- [DLH90] S. Debray, N.-W. Lin, M. Hermenegildo: Task Granularity Analysis in Logic Programs, Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation, ACM Press, 1990.
- [De84] D. DeGroot: Restricted And-parallelism, Int. Conf. on Fifth Generation Computer Systems, 1984.
- [GSYH91] G. Gupta, V. Santos Costa, R. Yang, M. Hermenegildo: IDIOM: A Model Integrating Dependent-, Independent-, and Or-parallelism, ILPS'91, MIT press.
- [HRSW94] W. Hans, J.J. Ruz, F. Saenz, S. Winkler: A Babel Parallel System: VHDL Modelling for Performance Measurement, GULP-PRODE'94, Valencia (Spain), 1994.
- [Ha94] M. Hanus: The Integration of Functions into Logic Programming: A Survey to appear in the J. of Logic Programming, available as Tech. Report MPI-I-94-201, Saarbrücken, 1994.
- [He86] M. Hermenegildo: An Abstract Machine for Restricted And Parallel Execution of Logic Programs, ICLP'86, LNCS 225, 25-39, 1986.
- [HN86] M. Hermenegildo, R.I. Nasr: Efficient Management of Backtracking in And-parallelism, ICLP'86, LNCS 225, 40-50, 1986.
- [HG90] M. Hermenegildo, K.J. Green: &-Prolog and its Performance: Exploiting Independent And-Parallelism, ICLP'90, 253-268, 1990.
- [HR90] M. Hermenegildo, F. Rossi: Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions, J. of Logic Programming, 1993.
- [KLMR90] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Graph-Based Implementation of a Functional Logic Language, ESOP'90, LNCS 432 : 271-290, 1990.
- [KLMR92] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Graph-Narrowing to Implement a Functional Logic Language, Technical Report, U.P. Madrid, 1992.
- [KH91] H. Kuchen, W. Hans: An And-Parallel Implementation of the Functional Logic Language BABEL, Granada Workshop on the Integration of Functional and Logic Programming, Aachener Informatik-Bericht 91/12:119-139, RWTH Aachen, 1991.
- [KMH92] H. Kuchen, J.J. Moreno-Navarro, M. Hermenegildo: Independent And-Parallel Narrowing, PLILP'92, LNCS 631, 1992.
- [Li87] G. Lindstrom: Implementing Logical Variables on a Graph Reduction Architecture, Workshop on Graph Reduction, LNCS 279, Springer 1987, 328-400.
- [LKID89] R. Loogen, H. Kuchen, K. Indermark, W. Damm: Distributed Implementation of Programmed Graph Reduction, PARLE'89, LNCS 365:136-157, 1989.
- [LW91] R. Loogen, S. Winkler: Dynamic Detection of Determinism in Functional Logic Languages, PLILP'91, LNCS 528:335-346, 1991.
- [MH90] K. Muthukumar, M. Hermenegildo: The CDG, UDG and MEL methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-Parallelism, ICLP'89.
- [MKLR90] J.J. Moreno-Navarro, H. Kuchen, R. Loogen, M. Rodríguez-Artalejo: Lazy Narrowing in a Graph Machine, ALP'92, LNCS 456:298-317, 1990.
- [MR92] J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Logic Programming with Functions and Predicates: The Language BABEL, J. of Logic Programming:12: 191-223, 1992.
- [MKMWH93] J.J. Moreno-Navarro, H. Kuchen, J. Mariño-Carballo, S. Winkler, W. Hans: Efficient Lazy Narrowing using Demandedness Analysis, PLILP'93, Springer LNCS 714, 167-183, 1993.
- [PJ87] S.L. Peyton Jones: The Implementation of Functional Programming Languages, Prentice Hall, 1987.
- [PCH87] S.L. Peyton Jones, C. Clack, N. Harris: GRIP - a Parallel Graph Reduction Machine, Workshop on the Implementation of Functional Languages, 1987.
- [Re85] U.S. Reddy: Narrowing as the Operational Semantics of Functional Languages, IEEE Int. Symp. on Logic Progr., IEEE Computer Society Press, 138-151, 1985.
- [SWYH91] V. Santos Costa, D.H.D. Warren, R. Yang: The Andorra-I Engine, ILPS'91, The MIT Press, 1991.

- [Wa83] D.H.D. Warren: An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, California, 1983