A Babel Parallel System: VHDL Modelling for Performance Measurement*

F. Sáenz†, W. Hans†, J.J. Ruz‡, S. Winkler†

† RWTH Aachen, Lehrstuhl für Informatik II, D-52056 Aachen, Germany
e-mail: {hans,winkler}@zeus.informatik.rwth-aachen.de
‡ Universidad Complutense de Madrid (UCM), Departamento de Informática y Automática, 28040 Madrid, Spain
e-mail: {fernan,jjruz}@dia.ucm.es

Abstract

In this work we present a stack-based abstract machine designed for the independent And-parallel execution of higher order innermost Babel programs. It is accomplished upon a shared memory model with local memory areas. We validate the abstract machine and study its performance with VHDL tools. Our approach is based on the exploitation of independent And-parallelism which deals with the parallelization of both strict and built-in non-strict functions. The specification is carried out in the hardware description language VHDL offering simulation facilities and having a temporal model which allows the carrying out of a performance study of the parallel system. We present the results obtained from the simulation of the VHDL specification, showing several significant factors of the parallel system.

1 Introduction

Babel [9] is a functional logic programming language which embodies the advantages of both the functional and logical paradigms, i.e., higher order functions, a strong type system, logical variables, unification, and nondeterminism. Its operational semantics is based on *narrowing*, a mechanism that subsumes SLD-resolution and reduction by performing rewriting of terms and unification.

Our aim is to speed-up Babel programs by exploiting a suitable form of independent And-parallelism (IAP) [3]. We follow the strict parallel model [11] modified to deal with the parallel execution of built-in non-strict functions[1]. Under this kind of parallelism, we allow the parallel evaluation of *independent* expressions on the right hand side of a Babel rule. In an innermost functional language, several expressions are said to be independent if no expression is a subexpression of any other expression. Since the Babel language embodies logical variables, the independence refers also to the sharing of common logical variables. This condition is imposed because the common variables may be bound to different values by the evaluation of the different expressions.

*This work was supported by the Spanish PRONTIC project TIC92-0793-C02-01 and by the German DFG-grant In 20/6-1.

[1] A function f is said to be strict in its i-th argument iff $f(a_1, \ldots, a_i, \ldots, a_n)$ is undefined whenever a_i is undefined.

Several parallel abstract machines have been proposed for Babel [5, 12, 7]. All of them adapt the graph-based sequential machine [6] to a parallel system. Our approach incorporates the more sophisticated memory management [8] relying on an efficient stack-based mechanism used in other implementations (e.g. in the WAM [13]).

In this work we first informally sketch the parallel model, then we point out the ideas present in the stack-based model, and finally we turn to the low-level specification in the hardware description language VHDL. This language provides a versatile set of description facilities for modelling systems from the behavioural level to the gate level. Moreover, it has a temporal model capable of the simulation of the fine grain features required to study the performance of our abstract machine. Instead of evaluating the system in the most usual way by relying on a software layer (e.g., a concurrent operating system) which isolates the underlying system from the abstract machine, we simulate the system at the memory level so that it is possible to carry out the performance study of fine grain features present in the abstract machine. We have designed a VHDL model for the parallel system assuming a shared memory over a single bus and local memory to reduce the bus traffic. In this model we can specify several parameters of the system such as the memory access times (register, shared memory and local memory). The simulation provides the actual computation time of the parallel system, which can be compared with the computation time of the sequential system. Moreover, we can carry out the study of the factors which explain the way the parallel system behaves. For instance, we may evaluate the data bus contention, different cache memory policies, and others which we may analyze and adjust to try different alternatives leading to a better performance.

This paper is organized as follows: Section 2 introduces the innermost version of Babel we deal with and the source of parallelism we exploit. In Section 3, an approach to exploit independent And-parallelism is proposed by presenting the parallel non-strict stack-based parallel model, the parallel system, the instruction set and the underlying memory model. VHDL is used in Section 4 to specify and validate the parallel model. Section 5 presents some preliminary results obtained from the simulation of the specification. Finally, Section 6 summarizes the conclusions and points out future work.

2 Parallelizing Babel

Here we present the basics of the functional logic language Babel and describe the extension to meet the expressiveness of parallelization. Furthermore we sketch informally how the abstract machine performs the parallel evaluation.

2.1 The Babel Language

Babel is a functional logic language with a constructor discipline and a polymorphic type system. It has a functional syntax and uses narrowing as evaluation mechanism. For the sake of clarity, we will consider the first order subset of Babel with the leftmost

innermost narrowing strategy applied[2].

A Babel program consists of a finite sequence of function definitions and can be queried with a goal expression. Each function f is a finite sequence of defining rules, where each rule has the form:

$$\underbrace{f(t_1,\ldots,t_n)}_{\text{left hand side } (lhs)} \quad := \quad \underbrace{\underbrace{\{B \rightarrow\}}_{\text{optional guard}} \quad \underbrace{M}_{\text{body}}}_{\text{right hand side } (rhs)} .$$

where B is a Boolean expression, and M is an arbitrary expression. Babel functions are functions in the mathematical sense, i.e. for each tuple of (ground) arguments, there is at most one result. This is ensured by special syntactic restrictions (see [9] for details).

A *term* t is either a (logical) *variable* or an application of a n-ary data constructor ($n \geq 0$) to n argument terms:

$t ::= X$ % variable
$\mid c(t_1 \ldots t_n)$ % application of a n-ary data constructor c

An expression M has the form:

$M ::= X$ % variable
$\mid \varphi(M_1,\ldots,M_n)$ % φ is a n-ary function or constructor symbol
$\mid B \rightarrow M_1\{\square M_2\}$ % if B then M_1 else undefined { else M_2}

Several built-ins conjunction ($\wedge$), disjunction ($\vee$), and Boolean negation ($\neg$) are supported, too.

The operational semantics of Babel is based on narrowing [10].

Let $M \equiv f(M_1,\ldots,M_n)$ be an expression and let $f(t_1,\ldots,t_n) := M'$ be a variant of a rule sharing no variables with M. Moreover, let $\sigma \circ \lambda$ be the *most general unifier* of M and $f(t_1,\ldots,t_n)$, i.e. the minimal substitution (of variables by expressions) such that $M\sigma$ is syntactically identical to $f(t_1,\ldots,t_n)\lambda$. Then, M can be *narrowed* (in one step) to $M'' \equiv M'\lambda$ with *answer substitution* σ (denoted by $M \Rightarrow_\sigma M''$). Moreover, if for some i $(0 \leq i \leq n)$ $M_i \Rightarrow_\sigma M_i'$, then $\varphi(M_0,\ldots,M_i,\ldots,M_n) \Rightarrow_\sigma \varphi(M_0\sigma,\ldots,e_i',\ldots,M_n\sigma)$.

In order to cope with the built-in non-strict functions, the basic narrowing mechanism is extended with the following narrowing rules:

$$\begin{array}{llll} true \wedge B & \Rightarrow_\epsilon B & false \wedge B & \Rightarrow_\epsilon false \\ true \vee B & \Rightarrow_\epsilon true & false \vee B & \Rightarrow_\epsilon B \\ true \rightarrow M_1\{\square M_2\} & \Rightarrow_\epsilon M_1 & false \rightarrow M_1 \square M_2 & \Rightarrow_\epsilon M_2 \end{array}$$

where ϵ denotes the empty substitution $(\forall X \in Var : \epsilon(X) = X)$.

A sequence of steps $M \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_m} M'$ is called a *computation* for the goal M. If M' is a term, we have found a *solution* consisting of the *result* M' and the *answer* $\sigma := \sigma_1 \circ \ldots \circ \sigma_m$. If M cannot be further narrowed, but it is not a term, then the computation *fails*. The implementation will backtrack in such a situation. The rules are tried in their textual order.

In the following we consider only the (sequential) leftmost innermost narrowing strategy. This means that expressions at leftmost innermost positions are narrowed first.

[2]The extension to the higher order version introduces no restrictions.

When dealing with conjunction and disjunction functions we call false and true *definitory values*, respectively, because this result of one operand determines the outcome of the whole function. To end with this short introduction to Babel, we will present the following example.

Example 1: The classical procedure for *append* is written in Babel:

$append([], Xs) := Xs.$
$append([X|Xs], Ys) := [X|append(Xs, Ys)].$

2.2 Exploiting And-parallelism

We focus our attention on a suitable form of independent And-parallelism for the functional logic programming paradigm: we allow the parallel execution of the functional arguments on the right hand side of the rules. The term And-parallelism originates from Prolog and denotes that kind of parallelism that is exploited for the arguments (goals) of the conjunction. This kind of parallelism is adopted for Babel and applies not only to the conjunction but also to all the other built-ins and user defined functions[3].

2.2.1 The Strict Parallel Model

An expression M defining the *rhs* of a rule has, in general, functional arguments (expressions with a function symbol as the root node). The parallel evaluation consists of the parallel execution of those functional arguments (which are subexpressions of M) which meet the following independence conditions:

- the functional arguments are *independent*, i.e., two arguments are independent if they do not share any variables at run time.
- one of the functional arguments is not a descendant in the syntactic tree of another functional argument.

Example 2: Consider the recursive rule for flattening lists:

$flatten([H|T]) := append(flatten(H), flatten(T)).$

The system presented in [11] generates the following parallel rule.

$$\begin{array}{l} flatten([H|T]) := \text{let } cpar(indep(H,T), \\ \qquad\qquad A_1 := flatten(H),\ A_2 := flatten(T)) \\ \qquad \text{in } append(A_1, A_2). \end{array}$$

$flatten(H)$ and $flatten(T)$ will be executed in parallel if H and T are independent (first condition), otherwise a sequential execution will be performed. Anyway, *append* must wait for their computed results (second condition).

2.2.2 The Non-strict Parallel Model

In this section we extend the basic strict parallel model in order to cope with some non-strict functions. The logical conjunction and disjunction, together with the conditional and biconditional functions belong to this class. The declarative semantics

[3]Although another name, e.g. subexpression parallelism, for this kind of parallelism could be more adequate, we still call it And-parallelism in order to follow the previous nomenclature [5, 12, 7].

of the conjunction when one of the arguments yields *false* is also *false*, whatever the results of the remaining arguments were. If we allow in the parallel scheme that any argument might define the result of a non-strict function, then we get nondeterminism and unpredictable behaviour. Rather than waiting for an arbitrary argument that yields a definitory value, we expect the same solutions as in the sequential scheme and maintain the sequential order of solutions, so that the aforementioned nondeterminism is avoided.

The arguments of a non-strict function are allowed to be executed in parallel. As soon as a definitory value for some argument is computed, only the arguments to its left are taken into account for the *outcome* (i.e., the result together with the success substitution). If not all of them have been computed, new definitory values may be delivered and this process must be repeated. If there is no definitory value at all, we are committed to wait for all the arguments. Note that this mimics exactly strict functions.

If a failure is computed for some argument position on the first invocation of the parallel call (i.e., *inside state* [2]), a wait for the arguments to its left must be performed, because if one of these arguments returns a definitory value the need of backtracking is avoided and the forward execution can be resumed. Otherwise, an (intelligent) backtracking out of the parallel call is performed.

If the backtracking course reaches a previously computed parallel call (i. e. *outside state*), the rightmost argument position with pending alternatives is searched for a *redo*. The arguments to its right are unwound and a new solution is requested. Note that it is possible to optimize the backtracking scheme by suppressing the unwinding of deterministic computations. Meanwhile, arguments to the right are spawned in parallel, therefore anticipating work. From this point, the behaviour is like in the forward computation.

3 An Efficient Implementation of And-parallelism

We rely on the work previously done for stack-based implementation of (sequential) narrowing [8] and show the extension to an efficient parallel scheme. Section 3.1 gives the description of the parallel model, Section 3.2 gives a slight description of the system introducing some needed data structures to support parallelism, Section 3.3 lists the most representative instructions (i.e. , parallel related instructions), and finally, Section 3.4 describes the underlying shared memory system.

3.1 A Stack-based Parallel Model

One advantage of stack-based implementations is the fast deallocation of data frames (choice points, environments, ...) during backtracking by simply changing the register contents. This cannot be straightforwardly extended when parallel computations are performed. Then data structures may be shared by several machines leading to effects like the garbage slot and the trapped goal problems [2]. Nevertheless, we maintain

this feature[4] in our model by committing to the following strategy: only *newer* frames are allocated on top of the corresponding stacks.

Each intermediate state of a sequential narrowing computation can be represented by means of an And-Or tree, in which *And* and *Or* nodes alternate (see Figure 1). An And node represents the head of a rule whose sons are the applications of the right hand side of the rule. These applications are the Or nodes whose sons are the heads of the rules that are unifiable with each Or node. The construction of such a tree is guided by the eager computation mechanism. A partial ordering is defined regarding that a frame a is newer than a frame b whenever b is found first in the narrowing tree in an innermost left-to-right order. The position of a node in the tree denotes definitely its age.

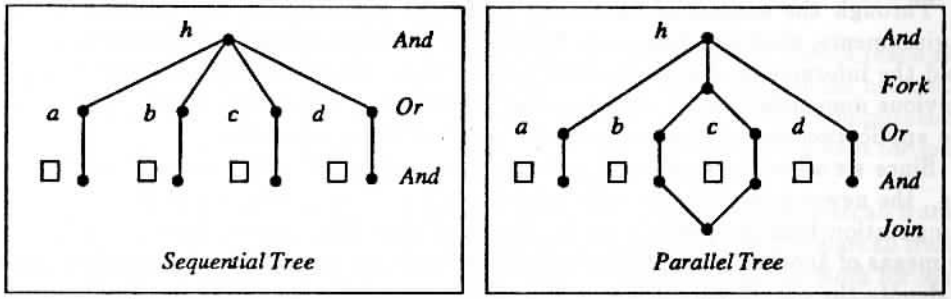

Figure 1: Sequential and Parallel Execution Trees

Furthermore, each intermediate state of a parallel narrowing computation can be represented by adding to the previous scheme *Fork* nodes, which represent the parallel scheduling of their son Or nodes, and *Join* nodes, which represent join points. In this scheme, Or nodes as well as Fork nodes may be sons of a given And node. Of course, we are able at this point to definitely associate a relative age between Or nodes and its sons, as well as between sons of Fork nodes. We can also set relative ages between the sons of a given And node, provided they may be Fork or Or nodes. But we are not able to compare any ancestor node with descendant nodes of a Fork node until the Join node is reached, because the corresponding subtree configuration for the Fork node is only known after the join point. Note that, in general, several subtrees may be tried by the backtracking mechanism before reaching the successful configuration.

We follow this restriction present in this partial order when pushing new frames on top of the stacks so that on backtracking, the frames to be deallocated are found on top of the stack.

[4]Although other features are embodied in our model, due to the lack of space, we skip them in this presentation since they are not related with the parallel behaviour. Please, consult [1] to get a detailed presentation of this work.

3.2 The Parallel System

We focus on the aspects concerning the management of parallelism, omitting the details concerning the narrowing system, which can be consulted in [8] and [1]. The forward and backward computation are described.

3.2.1 Forward Computation

Briefly, one parallel narrowing machine is designated to perform the narrowing of a given query. Whenever a fork point is reached on applying narrowing steps, other machines which comply with the precedence restriction are allowed to narrow the available parallel branches.

Through the sequential narrowing path, the usual frames are considered (i.e., environments, choice points, ...). To support parallel calls we use another frame to hold the information due to the fork point, the so-called *parcall* frame, following the previous nomenclature [2]. Among other information, it holds for every parallel call: the application name, its state, and the result of the computation.

Since we allow the local computation of parallel tasks[5] in the *active* parcall frame (i.e., the newer parcall frame with branches not yet computed), the different parallel computation branches, which are local sons of the father parcall frame, are isolated by means of another frame, the so-called *local task marker* which contains information to denote the parent parcall frame and the corresponding parallel branch. Moreover, in order to keep computation threads in remote machines also isolated we will use a similar frame, the so-called *input task marker*, which contains similar information to the local task marker.

3.2.2 Backward Computation

We distinguish three states in which a failure may happen:

- A failure is computed during *inside* state for a parallel branch. If necessary, we kill the remote computations and wait for the kill acknowledgement before switching the local computation to backward. The restoration of the previous state is efficiently done by restoring the corresponding registers, and by unwinding the variables annotated in the trail, as in sequential systems. Moreover, this is a parallel operation as far as remote machines perform this operation in parallel.

- A sequential thread runs out of alternatives provided that the next choice point is below the last (i.e. , topmost) join point (*outside* state). In order to detect this situation another frame is used: the so-called *waiting marker* that holds the parcall frame it corresponds to. The waiting marker is pushed onto the run-time stack whenever a join point has been successfully reached. The rightmost branch with alternatives is redoed, while the branches to its right are deallocated.

[5]High workload may lead to circumstances such that there are no available resources to pick up the newly created tasks.

- A failure is computed for a local (resp. remote) parallel branch, provided that the next choice point is below the topmost local (resp. input) task marker and the state is *outside*. Unlike inside state, we try a redo for the rightmost branch b to the left of the failed one, killing the tasks to the right of b.

3.3 The Instruction Set

The criteria that we follow in all the solutions we propose is that parallelism will not only be exploited in terms of And-parallelism, but also in all the duties related to the machine operation. For this purpose, idle machines deal with the duties which are allowed to be executed in parallel. For reasons of space we give only sketch of the parallelism related instructions, i.e., process instructions. For a complete and detailed description of the instruction set consult [1].

`Par` n $(l_1,\ldots):\ldots:(l_n,\ldots)$. It pushes a new parcall frame with n slots (each slot represents a parallel branch) onto the run-time stack and initializes all the necessary components. The list of tuples denotes the entry points of the parallel computations as well as some other data needed for initialization. The leftmost slot is kept for local execution. This instruction represents the fork point.

`Wait`. It waits for the termination of the parallel sons of the current parcall frame. This represents the join point of the parallel computation. If the active parcall frame still has further available work, then the machine is allowed to steal this work for local execution, therefore pushing a local task marker on its own run-time stack. Otherwise the machine waits for the remaining remote computations. In the wait state, an external kill notification may occur. For instance, because one of the parallel sons or some ascendant leads the local kill. When all the parallel sons have been successfully computed, a waiting marker is pushed on top of the run-time stack, and the sequential computation thread is resumed.

`ConjWait, DisjWait`. They wait for the termination of the parallel arguments of the non-strict conjunction and disjunction functions. They are placed instead of the `Wait` instruction, and implement the above explained forward non-strict behaviour, taking care of the definitory values.

`NotifyResult`. It finishes a successful computation of the corresponding parallel son. The computed result is located on top of the data stack and waits until the father parcall frame can be updated. Afterwards, the result is copied onto the corresponding parcall frame and the machine performs the proper actions on the father machine (e.g. , deciding to kill needless parallel arguments of a non-strict function). Finally, the machine is turned to the looking for work state. In this state, the machine performs the scheduling strategy on its own, respecting the precedence condition. There is no task queue, therefore saving the local machine from the management of such an structure. The machine scans in an efficient way the corresponding ancestor stacks looking for suitable parallel branches to be stolen. Whenever the machine picks up a suitable stack for remote computation, an input task marker is pushed on top of the run-time stack. When looking for work, a kill or redo message may arrive referring to the topmost input task marker.

`NotifyFail`. It notifies the (local) father about the computed failure. The local machine deals with the backward computation mechanism. Kill notifications may be a result of this mechanism. They consists only of the increasing of a kill notification counter located at each machine. In fact, this is the only information needed to be sent, since the other information is kept in the corresponding task marker. The kill procedure is widely executed in parallel, because the machines which send kill notifications to the target machines only wait for the acknowledgements when it is really necessary. This is why several kill notifications may arrive at a given machine before completing a kill.

3.4 Shared Memory System

Shared and distributed memory systems are the best representative memory models for parallel systems. In the shared memory model, a common memory is shared by the processors in the network. The efficiency of these systems is bound by the bandwidth of the memory since the processors request access to the same address space. In the distributed memory model, each processor with local memory is typically connected to others through links according to some topology. Whenever some data is requested from another node in the network, a message passing mechanism is activated driving the messages through the links. In this case, the efficiency is bound by the maximum message flow which depends eventually on the topology considered and the message bandwidth.

Both models have advantages and drawbacks that make them more suitable for certain applications. The shared memory model is more adequate for problems in which frequent access to common memory areas are required, but a memory access protocol must be supported. If local data areas are more often accessed, then the distributed memory system is more suitable, but a costly message passing mechanism is needed.

We extend the shared memory model with local memory, which is intended to save bus requests by holding the local data areas. Moreover, accesses to the local memory area are faster than accesses to the shared memory area. Since the code area is read-only, it can be placed at local memory. The data stack is also placed at local memory, so that the calls to and returns from functions are speeded-up. To manage remote readings to registers (there are no remote updates), they are copied to the shared memory when another processor requests it.

We consider a shared memory system in which n processors with local memory are connected to the shared memory through several lines and buses. The paths, common to typical shared memory systems, are:

- the *data bus*, which drives data from and to the shared memory,
- the *address bus*, in which each processor deposits the address to be read or written,
- the *mode line*, indicating the read, write, or wait mode[6] ,

[6]Read and write modes are used to denote respectively a read or write request, while the wait mode is used to denote a wait on an critical section.

- the *request line*, which is activated by the corresponding processor requesting access to the memory, and
- the *acknowledge line*, which is activated by the memory system to allow the corresponding processor to access the bus.

The data bus grant is controlled by the protocol policy which is abstracted in the shared memory system. We have designed a First-Come-First-Serve protocol that grants the access control to the first processor requesting the usage of the data bus. If several processors request access at the same time, then this protocol behaves in a round-robin way. With this protocol all the processors have the same priority in accessing the data bus.

Mutual exclusion management for critical sections has been designed specifically for our model. The usual semantics of semaphores is supported, providing a wait statement on critical sections (i.e., a parcall frames) and a signal statement. We have extended this semantics in order to cope with kill interrupts, so that whenever a kill notification is received for a given processor, the execution of the corresponding processor is resumed for attending the kill interrupt.

4 VHDL Specification

VHDL is a hardware description language developed to gain uniformity in the description of design specifications [4]. It was designed to fulfil a number of needs in the design process. Firstly, it allows the specification of the behaviour of a design using familiar programming language forms. Secondly, it allows the description of the structure of a design, i.e. the decomposition into sub-designs and their interconnection. Thirdly, as a result, it allows a design to be simulated before being implemented, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping.

A VHDL program is a set of concurrent processes communicating by means of signals, following an event-driven model. The behaviour of processes is described by means of a sequential algorithm making use of common imperative constructions. To handle process communication, the language provides the `WAIT` and the *signal assignment* (`<=`) statements. The signal assignment statement projects future values for signals. The general syntax is:

```
signal <= expression AFTER time expression;
```

The `expression` is evaluated and the result is scheduled to become the current value of the `signal` after the delay indicated by `time expression`. The `WAIT` statement makes it possible to suspend the execution of a process, and to express the conditions for its resumption. The general syntax for a wait statement is:

```
WAIT ON sensitivity list UNTIL condition FOR time expression;
```

The logical `condition` is evaluated whenever an event happens in a signal which is in the `sensitivity list`. If the result is false, the process remains suspended, else it is resumed after the time-out interval denoted by `time expression`. The time is only increased when a `WAIT` statement is executed.

The general syntax of a process is:

```
name : PROCESS declarative region
       BEGIN sequential statements
       END PROCESS;
```

Subprograms, variables and so on are declared in the declarative region of the process, and the body sequential statements defines the functional behaviour.

We have used this language to specify the shared memory system in which we distinguish three main categories: the processor component, the memory component, and the paths linking them (i.e., the data and address bus, and the control lines). Since the processor and the memory components are isolated working elements, we specify them as VHDL processes. They perform communication through the paths, which are specified as VHDL signals. The processor and the memory are declared as VHDL components which allows easily to declare as many processors connected to the memory as a constant indicates. The system can therefore be tested easily for different numbers of processors.

The processor component implements the parallel narrowing unit. The body of the processor component consists of an initialization phase, followed by the parallel narrowing procedure. The skeleton of the processor component is depicted below.

```
BEGIN
initProcessor;
WHILE NOT v_Halt LOOP
  fetch(v_OpCode);
  CASE v_OpCode IS
    ...
    WHEN Par => ...
    ...
  END CASE;
END LOOP;
END PROCESS;
```

initProcessor initializes the processor state by loading the program code, resetting the registers and so on. A WHILE loop controls the processing of instructions. The instructions are read by the fetch procedure, which are processed by the corresponding branch of the selection statement CASE.

The basic shared memory system is specified as an asynchronous system in which the memory and each processor component are synchronized by means of the Request and Acknowledge lines. The VHDL specification corresponding to the read operation performed by a processor is shown below.

```
Mode <= readMode;
Address <= ... ;
Request <= TRUE;
WAIT UNTIL Acknowledge = TRUE;
v_Data := InData;
Request <= FALSE;
WAIT UNTIL Acknowledge = FALSE;
```


Whenever a read operation is requested by any processor, the Mode and Address lines are set with the corresponding values. Then, the Request line is set. The processor will wait until the memory decides to serve such request. The (shortened) synchronization part of the memory is as follows.

```
outdatas(v_Target) <= ... ;
acknowledges(v_Target) <= TRUE;
WAIT UNTIL requests(v_Target) = FALSE;
WAIT FOR v_Delay;
acknowledges(v_Target) <= FALSE;
```

Once the acknowledgement has been noticed by the processor, it can pick up the requested data from the corresponding signal, and then it resets the Request line. The memory, in turn, is synchronized again by the resetting of this line. The accessing delay for each data area is taken into account through the sentence WAIT FOR v_Delay;. Finally, the processor can resume its computation when the Acknowledge line is reset by the memory. From this point, the memory is free for attending to other requests.

This scheme is the basics for controlling the access to shared data areas. It has been upgraded in order to serve the local memory accesses. The key of such an upgrade is to save the WAIT sentence of the local data areas which occurs at the memory, being placed in both the read and write operations of the processor component. These local delays apply only to the processor, though the memory component serves the data.

The memory component embodies not only the data management (by serving read or write requests) but also the access protocol. The reason is the following: since the data stack and register local data areas are to be remotely accessed, we have abstracted this mechanism in such a way that the memory system will be responsible for the management of those tasks, freeing the processor system. Therefore, the memory system will serve all the data requests, but retaining the conceptual behaviour of the local data areas, so that the access to them will not interfere with the access to the global data areas. We show below the body of the memory process that implements the intended behaviour.

```
BEGIN
WAIT ON requests;
attend_local_requests;
LOOP
  IF no_pending_requests THEN
    EXIT l_memory;
  END IF;
  select_asker(v_PriorityArray, v_Target);
  attend_request(v_Target);
END LOOP;
END PROCESS;
```

The first sentence implies a wait on the change of the requests signal. Whenever any processor sets its own request line, the change is noticed by the memory. After,

the memory first attends to the local requests (`attend_local_requests`), which do not need to be refereed by the access protocol, because they are *per se* local requests. Then, the memory system scans the active requests deciding which request to serve (`select_asker`), finally serving it (`attend_request`). With the use of the `LOOP` statement, all the eligible requests at a given time are attended, and the information needed by the access protocol is held in `v_PriorityArray`. The process delimiter `BEGIN` and `END PROCESS` define an overall loop, i.e., when all the request have been served, the process continues with the first statements and waits for further requests.

The extended behaviour of semaphores are specified in the `select_asker` procedure. Whenever the memory component detects a kill notification for the processor requesting a wait statement, the memory component serves the request returning a kill notification to the processor, respecting the protocol policy.

5 Results

In this section we give the results of the current VHDL implementation. We have studied the performance of the system by running the Fibonacci program on different number of processors. We have considered the following factors: the gain of a shared memory system with local memory over the shared memory system without local memory, the speed-up, the local hits rate, and the latency.

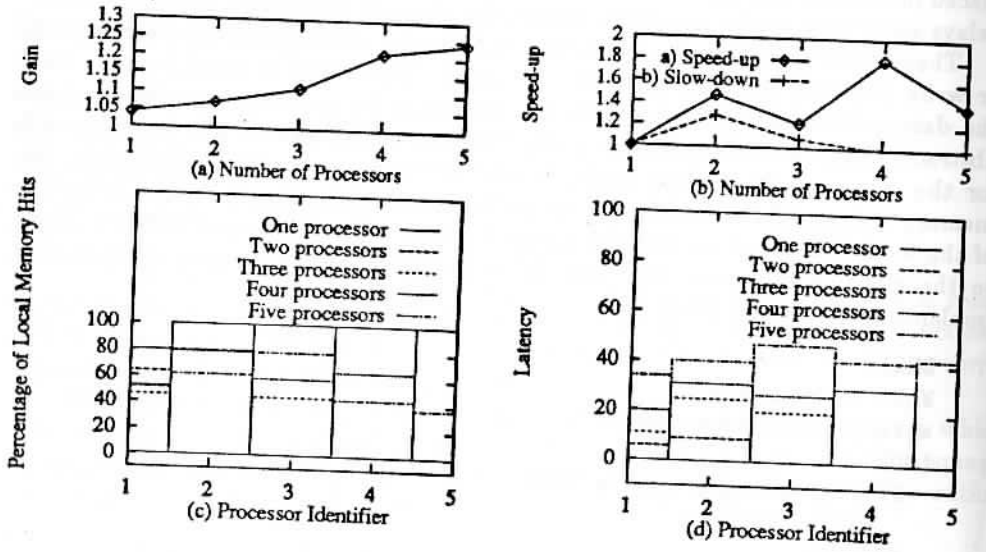

Figure 2: First results

Graph (a) shows that the greater the number of processors the greater the gain achieved when comparing both systems. This is because the bus traffic increases in the non-local memory case. A gain close to 24 percent is obtained with five processors, this demonstrates that if we add a cache memory to the parallel system (not only by

means of the local memory) we would obtain better speed-ups.

As would be expected, programs with insufficient granularity demonstrate no speed-up but actually slow-down as can be observed in graph (b)-b). This is clear due to the extra cost induced by the parallelism management. On the other hand, the speed-up curve in graph (b)-a) shows no increase in speed-up for the cases of three and five processors. This can be explained because of the scheduling policy together with the augmentation of the bus traffic. This example shows that the imposed precedence restriction results in this undesirable effect.

Graph (c) shows the rate between local and remote accesses. As the number of processors grows, this rate decreases because each processor needs more accesses to the part of the shared memory where remote data areas are held.

Finally, graph (d) shows the latency of the single bus. It is depicted as the rate between wait time and computation time. As the number of processors grow, the rate increases because bus traffic grows.

Further improvements of the profiling procedure will deal with more benchmarks and statistics. For instance, we may consider the processors that remain idle due to the precedence condition, the time spent in the parallel related operations, and others which will help us to tune the system for further efficiency.

6 Conclusions and Future Work

We have presented a computational model for the parallel execution of Babel that relies on a more efficient memory management than in previous approaches. Another important extension is the parallel execution of non-strict functions, a very essential concept of functional programming. All these ideas have been incorporated in an abstract model, assuming a shared-memory multiprocessor system with local memory over a single bus. Further refinements have delivered a low level specification in VHDL and a prototype implementation. We have tested the system with a small set of benchmarks that shows the applicability of our procedure in the design of the parallel system.

Further investigation will deal with several topics. For instance, the study of interleaved, distributed, and cached memory models. An interesting topic is the integration of the distributed and the shared memory model, together with the combined exploitation of And-parallelism and Or-parallelism, since And-parallelism is well suited to exploitation with shared memory clusters, while Or-parallelism can be better exploited on a distributed network. Finally, another topic is the extension of the non-strict parallel model to deal with lazy narrowing.

References

[1] W. Hans, J.J. Ruz, F. Sáenz, and S. Winkler. A VHDL Specification of a Shared Memory System for Babel. Technical Report 93–19, Aachener Informatik Berichte, 1993.

[2] M.V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[3] M.V. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[4] Institute of Electrical and Electronic Engineers, Inc. *IEEE Standard VHDL Language Reference Manual*, March 1988.

[5] H. Kuchen and W. Hans. An AND-Parallel Implementation of the Functional Logic Language BABEL. In *Workshop on the Integration of Functional and Logic Programming*, Granada, Spain, September 1990.

[6] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph Narrowing to Implement a Functional Logic Language. Technical Report DIA 92/4, Departamento de Informática y Automática, UCM, 28040 Madrid, Spain, November 1991.

[7] H. Kuchen, J.J. Moreno-Navarro, and M.V. Hermenegildo. Independent AND-Parallel Implementation of Narrowing. In *Book*, 1992.

[8] R. Loogen. Stack-based Implementation of Narrowing. In *CCPSD, Tapsoft, LNCS 494*. Springer-Verlag, 1991.

[9] J.J. Moreno and M. Rodríguez-Artalejo. BABEL: a Functional and Logic Programming Language Based on a Constructor Discipline and Narrowing. In *Conference on Algebraic and Logic Programming, LNCS 343*, pages 223–232. Springer-Verlag, 1988.

[10] U.S. Reddy. Narrowing as the Operational Semantics of Functional Programs. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 138–151. IEEE Computer Society Press, July 1985.

[11] F. Sáenz and J. J. Ruz. Parallelism Identification in BABEL Functional Logic Programs. In *7th Conference on Logic Programming (GULP'92)*, Milan, Italy, June 1992.

[12] M. Sassin. Design of an Abstract Shared Memory Machine for AND-Parallel BABEL with Dependency Information. In *Workshop on the Integration of Functional and Logic Programming*, Granada, Spain, September 1990.

[13] D. H. D. Warren. An Abstract Prolog Instruction set. 309 Technical Note, SRI International, 1983.

On the parallel implementation of the higher-order logic language λProlog*

Francesca Arcelli
Dipartimento di Ingegneria dell'Informazione e di Ingegneria Elettrica – Universitá di Salerno

Ferrante Formato
Centro di Ricerca in Matematica Pura ed Applicata Salerno

Giulio Iannello
Dipartimento di Informatica e Sistemistica Universitá di Napoli

Abstract

Higher-order logic languages seem particularly suitable to be implemented on parallel architectures for at least two reasons. On one hand, they offer more opportunities for parallelization than first-order logic languages. On the other hand, the greater computational costs of higher-order unification of typed λ-terms should allow the exploitation of parallelism with a larger granularity, possibly leading to higher efficiency. In spite of their potential interest, there are a few implementations of higher-order logic languages, and, to our knowledge, no experience has been made about their parallelization, up to now. We report here a preliminary study about parallelization of an interpreter for the higher-order logic language λProlog. Starting from the available experience in parallelizing first-order logic languages, the possible sources of parallelism in execution of λProlog programs are presented, and a characterization for their actual exploitation on parallel hardware is given. The analysis confirms that there are many opportunities for parallelization, and that a number of significant differences exists with respect to the Prolog case. An execution model for λProlog based on the results of our analysis is sketched, and finally some hints on the effectively parallelism exploitation from real λProlog applications, such as theorem-provers, are outlined.

*This work has been supported by Consiglio Nazionale delle Ricerche under funds of "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" and by MURST under funds 40%.

1 Introduction

In the context of logic programming languages, several attempts have been made to build higher-order extensions to this paradigm and especially two natural approaches have been proposed, based respectively, on a higher-order logic level and on a meta-logical level. In this work we analyze parallel implementation issues of higher-order features in logic programming, following the higher-order logic approach, and starting in particular from the λProlog language proposed by Nadathur and Miller [17].

λProlog is a logic language that extends Prolog by incorporating notions of higher-order functions, λ-terms, higher-order unification, and polymorphic types. These new features are provided by extending the classical first-order theory of Horn clauses to the intuitionistic higher-order theory of Hereditary Harrop formulas, which extends Horn clauses in two orthogonal directions: embedding higher- order notions, and providing primitives for specifying new search operations.

λProlog have been used for many purposes [7,12,13]. These experiments have indicated the usefulness of the language, especially of its higher-order features, and thus it is of considerable value to have at hands efficient implementations of λProlog. Several approaches have been proposed for providing good implementations [4,6,14] however their efficiency should be further improved for practical applications of the language.

Also for these reasons, it seems interesting to analyze a parallel implementation and its derived advantages. In fact, higher-order logic languages, such as λProlog, seem particularly suitable to be implemented on parallel architectures for at least two reasons. On one hand, they offer more opportunities for parallelization than first-order logic languages (as for example the exploitation of parallelism in the higher-order unification process). On the other hand, the greater computational costs of higher-order unification of typed λ-terms should allow the exploitation of parallelism with a larger granularity, possibly leading to higher speedups.

This paper aims to present a preliminary examination of the possibilities to parallelize the λProlog interpreter, with the aim to reduce its execution time. We are not concerned with any extension of the language for the explicit control of parallelism, but we try to point out the possible sources of inherent parallelism, and to characterize their nature. In particular, we start our analysis from the available experience in parallelizing first-order logic languages, and we discuss similarities and differences with respect to the higher-order case. Even though we do not propose yet practical solutions to the several problems pointed out throughout the paper, we believe that our analysis helps clarify a number of relevant issues and provides the necessary starting point towards a parallel execution model for λProlog. With some minor changes our analysis can apply also to other higher-order logic languages so as the Isabelle system [19] and the Elf language [20].

We concentrate on the version of the language based on higher-order Horn clauses, which extends Prolog essentially by replacing first-order unification with higher-order unification. Our decision of not considering here the other main feature of λProlog i.e. the ability of using implication and universal quantification in goals, is motivated by two main reasons. First, the extension of program clauses to Hereditary Harrop formulas does not introduce new sources of non-determinism in the interpreter, and no further parallelism can therefore be exploited with respect to the higher-order Horn clauses case. Second, experience matured in sequential implementations shows that the new features of λProlog are in a sense orthogonal to each other, and there is little interference among the mechanisms developed for each of these features [4, 18]. We therefore argue that our work can be a sound basis even for analyzing the impact of the Hereditary Harrop formulas on the parallelization of the λProlog interpreter.

The paper is articulated in the following sections. In section 2, the behavior of the interpreter of λProlog is described, and implementation issues relevant for parallelization are discussed. In section 3, we examine the different points of non-determinism present in the resolution process, and we give independence conditions for exploiting AND-parallelism in λProlog. In section 4, we discuss architectural issues for the development of parallel implementations of λProlog, we make some considerations towards a parallel execution model of the language and we analyze how non trivial programs, such as theorem provers, could benefit from a parallel implementation of the language. Finally we briefly discuss future research directions.

2 The λProlog abstract interpreter

In this section, we analyze the behavior of the λProlog interpreter to point out the sources of non-determinism present in the execution model of the language. As in the Prolog case, sequential implementations of the interpreter must include rules for selecting alternatives arising from the non-deterministic resolution process. When parallel execution is considered, this non-determinism gives rise to potential sources of parallelism that could be exploited to speed up the computation.

Our description of the abstract execution model of λProlog is that one reported in [18], where further details can be found. In the following, the symbols $\mathcal{P}$, $\mathcal{G}$, $\mathcal{D}$, and θ denote a set of higher-order Horn clauses, sets of goal formulas, disagreement sets and substitutions, respectively. Moreover, let $SIMPL$ denote the function that simplifies a disagreement set to flexible/rigid (F/R) pairs and flexible/flexible (F/F) pairs only, and let $MATCH$ be the function that, given an F/R pair, produces the set of imitation and projection substitutions for that pair. At last, the symbol $\mathbf{F}$ represents a disagreement set that has no unifiers, and an atom is said to be rigid if its top-level symbol is not a variable, flexible otherwise.

A tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2 \rangle$ is said to be $\mathcal{P}$-derivable from the tuple $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1 \rangle$ if $\mathcal{D}_1 \neq \mathbf{F}$ and one of the following situations hold[1]:

1. *(Goal reduction step)* $\theta_2 = \emptyset$, $\mathcal{D}_2 = \mathcal{D}_1$ and for some $G \in \mathcal{G}_1$ it is the case that:

   (a) G is $\top$ and $\mathcal{G}_2 = \mathcal{G}_1 - \{G\}$, or

[1] We have excluded from the rules the transformation introduced in [18] for solving flexible goals (i.e. a goal that has a variable as its head), because it is a simplification in the $\mathcal{P}$-derivation process that sacrifices completeness (its use can possibly lead to failure, even though solutions do exist for a given initial query [16]). Since parallel execution of the interpreter affects the set of solutions actually computed by the interpreter, we prefer to get rid of this rule in our analysis.

(b) G is $G_1 \wedge G_2$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G_2, G_1\}$, or

(c) G is $G_1 \vee G_2$ and, for $i = 1$ or $i = 2$, $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G_i\}$, or

(d) G is $\exists x\, P$ and, $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{((\lambda x\, P)Y)\}$ where Y is a new (free) variable.

2. *(Backchaining step)* $\theta_2 = \emptyset$, and for some rigid atom $G \in \mathcal{G}_1$, $G' \supset A$ is a clause in $\mathcal{P}$ instantiated with new variables and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G'\}$ and $\mathcal{D}_2 = SIMPL(\mathcal{D}_1 \cup \{\langle G, A\rangle\})$.

3. *(Unification step)* For some F/R pair $\chi \in \mathcal{D}_1$, either $MATCH(\chi) = \emptyset$ and $\mathcal{D}_2 = \mathbf{F}$, or $\theta_2 \in MATCH(\chi)$ and $\mathcal{G}_2 = \theta_2(\mathcal{G}_1)$ and $\mathcal{D}_2 = SIMPL(\theta_2(\mathcal{D}_1))$.

A sequence of the form $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i\rangle_{1 \le i \le n}$ is a *$\mathcal{P}$-derivation sequence* for a goal formula G if (i) $\mathcal{G}_1 = \{G\}$, $\mathcal{D}_1 = \emptyset$ and $\theta_1 = \emptyset$, and (ii) for $1 \le j < n$, the $(j+1)^{th}$ tuple is $\mathcal{P}$- derivable from the j^{th} tuple. Such a sequence terminates with failure if $\mathcal{D}_n = \mathbf{F}$ and with success if $\mathcal{G}_n = \emptyset$ and $\mathcal{D}_n$ is either empty or contains only F/F pairs. In the latter case, we say that the sequence is a $\mathcal{P}$-derivation for G. Such a sequence embodies in it a solution to the query G in the context of the program $\mathcal{P}$. The *computed answer* provided by this solution is obtained by composing $\theta_n \circ \cdots \circ \theta_1$ and restricting the resulting substitution to the free variables of G. An abstract interpreter for λProlog may be thought of as a procedure that, given a program $\mathcal{P}$, attempts to construct a $\mathcal{P}$-derivation for goal formulas. A state in this derivation process is characterized by a set of goals and a disagreement set, and the search for a solution is progressed by applying a simple step to one of these components. In [16] it is shown that this derivation process is complete, and that the order in which rules are applied in correspondence of the choice points does not affect the final result. As a consequence, non-determinism can be exploited to extract some parallelism, provided that the resulting process can be proved to be equivalent to some $\mathcal{P}$-derivation sequence.

The above set of transformation rules contains five points of non- determinism:

1. choice of a program clause for backchaining;
2. choice of a substitution returned by $MATCH$;
3. choice of an atom for backchaining;
4. choice of an F/R pair as an argument of $MATCH$;
5. choice between backchaining and unification.

As to goal reduction rules, they can be solved at compile time and do not give rise to actual processing load. In fact, complex goals containing logical constants, such as $\wedge, \vee, \exists$, can be statically reduced to sets of atomic goals [18].

We want now to make some comments about several aspects of sequential implementations of λProlog which have relations with our discussion.

Higher-order unification is semi-decidable, and it is an infinitary problem: if there are solutions they can be found, otherwise the search may diverge, and there may be several most general unifiers, sometimes a countable infinity. This has two consequences on implementations:

- Pre-unification [8] instead of complete higher-order unification is actually computed by the unification procedure. This choice requires that attempts to solve an unification problem must at times be suspended until the application of a backchaining step may introduce further constraints.
- The search procedure for pre-unifiers may have non-terminating branches in the matching tree (see examples in [4]. However, all implementations choose a depth-first strategy which can possibly lead to non terminating computations even when solutions exist.

There are two further observations regarding λ-unification. First, most unification problems in higher-order logic languages are first-order problems, or other special cases that do not require the complexity of the full higher-order unification procedure. This means that implementations should be able to recognize such cases and solve them cheaply with specialized algorithms [4,11]. Second, techniques developed to implement basic operations on λ-terms, such as $SIMPL$ or β-reduction, make extensive use of lazy evaluation to postpone work until it is actually needed, and to reduce memory requirements at run-time [18].

A second aspect is that, in sequential implementations, completeness of the derivation process is sacrificed for enforcing sequentialization in the execution model. The third consideration is that explicit representation of the disagreement set, even though avoidable in principle as in the first-order case, it is desirable for pragmatic reasons: in a situation where choices have to be made in substitutions, it appears best to use all available constraints on these choices [18]. Note that this unification strategy tends to reduce non-determinism, since it may, at the very least, curtail the branching in search. Moreover, explicit representation is also required by the need of suspending the unification procedure when backchaining steps are to be performed.

In this respect, a last observation is that also the switching strategy between backchaining and unification may play an important role in curtailing branching in the search procedure. In [18], the strategy followed is very simple: whenever possible, unification is always performed before backchaining. In other words, backchaining steps are delayed until no F/R pair remains in the disagreement set. However, in [11] it has been observed that this strategy can result in thrashing when certain combinations of unification problems have to be solved by extensive backtracking. A different strategy has then been embedded in the operational semantics of the higher-order logic language Elf [20], that attempts to solve only equations that lie within an appropriate extension of L_λ [15]. All other equations (solvable or not) are postponed as constraints. Even though we will not discuss further the matter here, it is apparent that this scheduling problem deserves a careful consideration when parallelism is introduced in the λProlog interpreter.

3 From Non-Determinism to Parallelism

We now analyze how the non-determinism present in the $\mathcal{P}$-derivation process can give rise to several forms of parallelism. The analysis aims to give a characterization of these forms of parallelism, and to identify the main implementation issues to be dealt with in order to achieve a parallel implementation of higher-order logic features. For those forms of parallelism present also in Prolog, references to solutions proposed for parallelizing execution of first-order logic programs and similarities and differences between the two cases are more extensively analyzed in [2]. We concentrate us in particular on the sources of parallelism typical of the higher-order language which we can exploit in a particular application of the language as pointed out in section .

3.1 Choice of a program clause for backchaining

This kind of non-determinism is present also in Prolog, where it has been classified as OR-parallelism. As in Prolog, OR-parallelism arises when the predicate corresponding to the head of a rigid atomic goal is defined through multiple program clauses, and hence the backchaining step can be attempted in several ways. For every program clause that can be selected, a different disagreement set is obtained after the application of $SIMPL$, and as many $\mathcal{P}$-derivations can be started. All these $\mathcal{P}$-derivations are independent subcomputations that correspond to different branches in the proof-search tree. Exploring in parallel several branches of this tree can speed up the computation when multiple answers to the initial query must be computed, or when the *depth-first* strategy would produce a lot of backtracking before the "right" branch is explored. The implementation of OR-parallelism in λProlog requires to solve essentially three problems:

- managing the binding environments of several independent resolvents in parallel;
- scheduling the corresponding subcomputations, so as work is balanced among processors and the system does not collapse under too many activities;
- making available to every subcomputation a coherent copy of the disagreement set.

The first two issues are typical problems also of the parallel implementation of first-order logic languages [5] (see [2] for a comparison). The third issue is specific of the higher-order case, and it represents a further source of overhead that should be accounted for. It is worth noting that the multiple copies of the disagreement set differ only for the new pairs added by the backchaining step. As a consequence techniques based on sharing or incremental copying are likely to exhibit a better performance than those requiring a complete duplication of the disagreement set.

3.2 Choice of a substitution returned by $MATCH$

Multiple solutions returned by the $MATCH$ procedure correspond to alternatives in the pre-unification algorithm that can be tried in parallel. Each alternative consists in applying a different substitution to the λ-terms in the disagreement set, giving rise to as many unification problems. The situation is quite similar to that discussed in the previous subsection. The originated parallelism can then be classified as a kind of OR-parallelism, and its exploitation requires to solve the same problems outlined above. Nevertheless, this form of parallelism represents a novelty with respect to Prolog, and it exhibits a number of peculiarities that it is worth noting.

First of all, in the sequential case when a substitution is chosen and applied to the disagreement set, a certain amount of work must be done to ensure that the substitution can be undone upon backtracking, so as another substitution can be tried in turn [4, 18]. We argue that this further overhead can make exploitation of this parallelism more convenient than the traditional OR-parallelism. In fact, applying substitutions to λ-terms in a reversible way seems a task comparable to that of making these same terms sharable among several parallel activities[2]. Hence, since parallelism substitutes backtracking, the overhead produced by the duplication of λ-terms could turn out to be partly hidden by the work saved for making substitutions reversible. Note again that sharing or incremental copying techniques should behave more favorably even in this respect.

A second observation is that parallel traversal of the matching tree affects the set of solutions actually computed by the unification procedure. This seems particularly important if the semi-decidable and infinitary nature of λ-unification is considered. In fact, on one hand, substituting depth-first search with a parallel one could improve the probability of finding solutions to unification problems having non-terminating branches. On the other hand, when more (or all) branches are tried in parallel, non-terminating searches are likely to take place, and a scheduling strategy capable to detect and abort such endless computations should be adopted. Hence, techniques for exploiting OR-parallelism in unification should pay special attention to policies used to schedule parallel subcomputations. A last point concerns the amount of OR-parallelism in unification that is present in real λProlog programs. As we have already mentioned, most unification problems arising during actual executions are not higher-order, or do not require extensive backtracking. Hence, parallel implementations should check for these special cases and limit parallelism exploitation only to cases of true higher-order problems. Note that such an hybrid approach does not introduce further difficulties if a multisequential model of execution is adopted.

3.3 Choice of an atom for backchaining

Parallelism arising from this form of non-determinism is present also in Prolog. It has been usually referred to as AND-parallelism, as it consists of developing in parallel goals that are implicitly connected by a logical AND operator: we have *independent* AND-parallelism, i.e. only goals not sharing variables are executed in parallel and *dependent* AND-parallelism, i.e. concurrent execution of goals sharing variables is allowed, but synchronized by producer/consumer relationships on these variables. In principle, the basic conditions for exploring independent AND-parallelism in λProlog

[2] This statement is justified by the observation that both tasks require a potentially complete traversal of the structure of λ-terms and, at least, a copy of parts of these terms.

do not differ from those given in the Prolog case. In practice, however, detecting such a condition in the λProlog case calls for different solutions, since variable bindings may be implicitly established also through the constraints represented by the disagreement set. This implies tight interactions with the unification step. In particular, AND-parallelism interacts with parallelism arising from the choice of an F/R pair as an argument of the MATCH procedure. Since the latter kind of parallelism is specific of the higher-order case, we will examine in more detail conditions for goal independence in the next subsection.

3.4 Choice of an F/R pair as an argument of *MATCH*

This form of parallelism has tight relations with AND-parallelism; in fact establishing goal independence requires extracting binding information from the disagreement pairs, so interactions with the unification algorithm are needed and parallel application of *MATCH* to several F/R pairs gives rise to a compatibility problem similar to that arising in the context of AND-parallelism. In particular when *MATCH* is applied to an F/R pair, a set of substitutions is returned for the head of the flexible term. If *MATCH* is applied in parallel to another F/R pair, one must ensure that the substitution applied in one case is compatible with the substitution applied in the other case. Again run-time checks enforcing coherency could be introduced, but they would be likely too expensive as in the case of AND-parallelism in Prolog. Instead, if only independent F/R pairs (i.e. not sharing variables) are allowed to be solved in parallel, the subcomputations will provide non-conflicting substitutions and no expensive coordination is required. However, even though independent pairs are selected, the corresponding subcomputations are not completely independent. To show this, let us indicate with $FV(\mathcal{T})$ the set of free variables appearing in $\mathcal{T}$, where $\mathcal{T}$ is a collection of λ-terms, and let us consider a partition of the disagreement set $\mathcal{D}$ in the subsets $\mathcal{D}_1$ and $\mathcal{D}_2$, for which the following condition holds:

$$FV(\mathcal{D}_1) \cap FV(\mathcal{D}_2) = \emptyset$$

In this hypothesis, the two subsets can be unified independently, and hence two non-interacting subcomputations can be started. Consider for instance the subcomputation that operates on the portion $\mathcal{D}_1$ of the disagreement set. Since the unification algorithm actually computes pre- unifiers for $\mathcal{D}_1$, if at some stage only F/F pairs remain in the disagreement set $\mathcal{D}'_1$ derived from $\mathcal{D}_1$, one or more backchaining steps must be performed until an F/R pair is added to $\mathcal{D}'_1$, or the goal set is empty. Now, assume that backchaining is performed on a rigid goal, of the form $G(t,u)$, where t and u are subterms containing the free variables $X \in FV(\mathcal{D}_1)$ and $Y \in FV(\mathcal{D}_2)$, respectively. After backchaining, the pairs $\langle t, r\rangle$ and $\langle u, s\rangle$ are added to $\mathcal{D}'_1$ (r and s are two subterms of the head of the clause selected for the backchaining step), and hence $\mathcal{D}'_1$ is transformed in a disagreement set that is not independent any more from $\mathcal{D}_2$. Again, we find a tight interaction between backchaining and unification in exploiting the form of parallelism discussed in this section. This consideration seems to suggest that the two sources of parallelism should be considered together in order to state

conditions for independent AND-parallelism. Under such an assumption, independent AND-parallelism in the context of a $\mathcal{P}$-derivation process can be characterized as follows.

Theorem 1 *Given a definite program $\mathcal{P}$, and given the tuple $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1\rangle$ at some stage of the $\mathcal{P}$-derivation process, if $\mathcal{G}_1$ and $\mathcal{D}_1$ can be partitioned into two sets $\mathcal{G}'_1$, $\mathcal{G}''_1$ and $\mathcal{D}'_1$, $\mathcal{D}''_1$, respectively, and the following condition holds:*

$$(FV(\mathcal{G}'_1) \cup FV(\mathcal{D}'_1)) \cap (FV(\mathcal{G}''_1) \cup FV(\mathcal{D}''_1)) = \emptyset \qquad (1)$$

then the $\mathcal{P}$-derivation of $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1\rangle$ can be performed by $\mathcal{P}$-deriving independently the tuples $\langle \mathcal{G}'_1, \mathcal{D}'_1, \theta_1\rangle$ and $\langle \mathcal{G}''_1, \mathcal{D}''_1, \theta_1\rangle$, and taking as an answer the composition of all the substitutions produced by these two $\mathcal{P}$-derivations.

The proof of the theorem can be found in [2]. Note that the theorem can be easily extended to partitions of $\mathcal{G}_1$ and $\mathcal{D}_1$ of any (same) cardinality. Condition (1) captures the interaction between goal set and disagreement set when AND-parallelism is exploited. In fact, it states that if partitions of $\mathcal{G}_1$ and $\mathcal{D}_1$ can be found satisfying the condition, then the subsets of $\mathcal{G}_1$ can be solved in parallel, independently. Unfortunately, such an independence condition, does not seem easy to detect at run-time. So it cannot be used directly to exploit independent AND-parallelism of a λProlog program. We will not investigate further the matter here, and more work is needed to verify if this kind of parallelism can be detected through static analysis of the program text. Alternatively, cheaper run-time tests detecting particular cases of condition (1) could be used. For instance, similarly to what happens in Prolog, if a goal clause G is closed (i.e. $FV(G) = \emptyset$), the subsets:

$$\mathcal{G}'_1 = G, \quad \mathcal{G}''_1 = \mathcal{G}_1 - \{G\}, \quad \mathcal{D}'_1 = \emptyset, \quad \mathcal{D}''_1 = \mathcal{D}_1$$

would satisfy trivially condition (1), and G could be solved in parallel with the remaining part of the goal set, without further overhead.

3.5 Choice between backchaining and unification

As we have already pointed out, backchaining steps and unification steps must actually be interleaved to cope with disagreement sets consisting of only F/F pairs. The situation is well modeled if the interpreter is thought of as made up by two cooperating coroutines, implementing unification and backchaining, respectively. It is then quite natural to think of parallelizing execution of these two activities, with proper synchronization points when the results of one of them must be conveyed to the other. However, when backchaining and unification proceed in parallel, maintaining coordination between them is not a trivial task due to the consistency maintenance. To clarify this point, let B and U be the processes carrying out backchaining and unification, respectively. There are two problems to be solved so as concurrent execution of B and U preserves correctness of the $\mathcal{P}$-derivation process. The first problem is what information must be exchanged between B and U. The second problem is relating

to the coordination of the two processes when backtracking has to take place. From the considerations developed in [2] it seems clear that for parallel execution of B and U it is not enough to exchange periodically the information produced, but a more tight coordination is needed. It is beyond the scope of this paper to provide practical solutions to the problems pointed out, so we do not discuss the matter further here, and refer the reader to the next sections for some more comments on future work concerning this point.

4 Discussion

After having examined what are the main sources of parallelism in the execution model of λProlog, in this section we will complete our analysis touching some other issues on the subject, and we will briefly discuss how parallelism can be actually exploited in implementations of λProlog . Finally we give an example of successful exploitation of independent AND-Parallelism which satisfies Theorem1 given in section 3.4. For a discussion on some other sources of parallelism, as for example the parallelization of SIMPL and MATCH functions or the exploitation of parallelism in the operations to be performed on each λ-term, see [2].

4.1 Architectural considerations

The effort needed to develop parallel implementations of λProlog depends on the architectural characteristics of the target hardware. We expect that, as in Prolog, the more effective way to develop such implementations on currently available parallel hardware is to derive a multisequential model from existing sequential implementations. This approach should be fairly simple for shared memory multiprocessors, even though the more complex algorithms and data structures needed to cope with higher-order features require more sophisticated techniques not to incur in non-acceptable overhead. In particular, if parallelism between backchaining and unification is to be exploited, efficient solutions to implement the merging phase must be found. For distributed memory architectures (multicomputers), two approaches can be followed. A Shared Virtual Address Space [22] support could be used to minimize the changes needed with respect to the multiprocessor case. Memory management and scheduling issues are likely to be changed, but mechanisms developed for shared memory machines could be reused on multicomputers. The second approach could consist in extending closed environments techniques [5] proposed in the Prolog case to λProlog. Even though they seem more suitable to distributed memory hardware, their use in the λProlog case could give rise to further difficulties because of the role played by sharing and lazy evaluation in representation and manipulation of λ-terms. At last, as already pointed out, fine-grain concurrent computers seem to be very promising for parallel implementation of λProlog and other logic languages. The main drawback of this solution, besides current unavailability of proper hardware, is the need to redesign many of the algorithms and data structures to exploit parallelism at a finer level than it is allowed by conventional architectures. In this respect, we guess that the basic

model of computation used for programming fine-grain architectures, i.e. Actors [1], has many attractive features. In fact, actor languages provide a notation to naturally express both parallel recursive computations like $SIMPL$, and distributed data structures like those needed to perform β-reduction in parallel. Furthermore, the functional notation of most actor languages allow the user to easily express lazy and eager evaluation of expressions [10, 24].

4.2 Towards a parallel execution model of λProlog

Considerations developed earlier have outlined that a number of difficulties for exploiting parallelism between backchaining and unification exist. On the other hand, the characterization of independent AND-parallelism given in section 3.4 suggests that even AND-parallelism cannot be easily exploited in a direct fashion. These observations would lead us to follow a conservative approach, limiting our attention only to OR-parallel models for execution of the λProlog interpreter. However, we claim that a serious attempt should be done to propose an execution model based on parallelism between backchaining and unification, before turning our attention towards simpler OR-parallel models. The reasons for this are not, of course, the hope to gain large speedups directly from this choice: this kind of parallelism is not scalable and it is limited to a degree of two that, after all, is the minimum parallelism conceivable! Rather, we believe that, structuring the interpreter as two cooperating processes corresponding to the basic steps of the $\mathcal{P}$-derivation process, may turn out to be very useful for several reasons. In fact, even though our considerations are not supported by strong evidence, and further work is needed even for better defining a concrete parallel execution model like that we are suggesting, at least three motivations can be given to support our belief:

1. In a parallel environment, concurrent execution of backchaining and unification is a more natural solution, and it provides a flexible basis to test different scheduling strategies, included adaptive ones. In this respect, parallel execution of B and U, if properly managed, could exhibit a favorable behavior, allowing a larger pruning of the search tree than it is possible with any sequential strategy. The situation resembles the one that can be found in some branch-and-bound problems, where searching in parallel allows better bounds to be found earlier, and more subtrees to be pruned before traversing them [21]. As a consequence, the parallel version of these algorithms do less work than the sequential one, possibly leading to superlinear speedups.

2. No further difficulties arise in principle to implement OR-parallelism. When a choice point is encountered, if search must be split into subcomputations, a pair of processes B and U must be created for each alternative. Since OR-parallelism substitutes backtracking, extensive exploitation of OR-parallelism may even lead to a simplification in the coordination between backchaining and unification.

3. If backchaining and unification are implemented as distinct processes, and general mechanisms are provided to manage their coordination, further parallelism

can be exploited. The B process can exploit AND-parallelism discussed in section 3.3. This time, however, since backchaining is performed independently of unification, all the goals can be reduced in parallel. In other words, goal reductions are now all independent, and no complex conditions must be checked any more. On the other hand, substitutions for different F/R pairs not sharing free variables can be computed and applied in parallel by the U process. This is the same parallelism discussed in section 3.4. This time, however, a simpler condition suffices to exploit such parallelism inside unification.

It is worth noting that in both cases mentioned in point 3, parallelism is exploited at a level of granularity that it is likely to be finer than that present between backchaining and unification. This means that, depending on the underlying architecture, goals and F/R pairs could have to be grouped so as to increase the granularity to an acceptable level.

4.3 Exploiting parallelism in theorem-provers implementation

. Applications in which there is a good amount of search are typical cases which derive the most significant advantages, in terms of a parallel implementation of the meta-language interpreter. Theorem provers constitute a meaningful example in which the process of discovering a proof involves traversing an often very complex and large search space under some controlled techniques.

In [3] after analyzing the Felty's implementation of tactic style theorem provers in λProlog we pointed out the sources of parallelism that we can exploit in the primitive control operations of the interpreter defined in [7], and provided through the modules maptac, tacticals and goalred. All the definite clauses in the maptac, tacticals and goalred modules provide a complete implementation of *tacticals*. The module maptac handles all the goal structures corresponding to the λProlog search operations and represents one of the basic control primitives or *tacticals* of the interpreter.

We proceed by analyzing the possible sources of parallelism arisen from the relation between tactic specifications and λProlog search operators. The following clause 2 of the maptac module is an interesting case of successful application of AND parallel computation processes, as we show with the following theorem.

Theorem 2 .*The invocation of the following clause:*

```
maptacTac(InGoal1&&InGoal2)(OutGoal1&&OutGoal2) : -
    maptacTacInGoal1OutGoal1,maptacTacInGoal2OutGoal2.        (2)
```

occurs always with the variable 'Tac' instantiated with a ground term.

Proof. by structural induction on the computational steps (see [3]).

As a consequence of the above result, we find some assumptions under which the execution of the clauses in the body of clause 2 could be carried out by two indipendent

AND parallel processes. This result has be shown in [3] using the dynamical model of tuples described in [16].

In order to execute the two goals in the body of clause 2 using two independent AND parallel processes, the assumption that *λ-terms instantiating variables in InGoal1 and InGoal2 must not share free variables.* is necessary.
In fact, a possible sharing of variables will cause inconsistency problems between the two processes solving the subgoals, which are the arguments of the (metalevel) conjunction of the body of - clause 2.
Examples which put in evidence the relation between the conditions above and the evolution of the computation, highlighting the occurrences of AND independent parallelism, and the analysis of the exploitation of other sources of parallelism of the tactic style interpreter for theorem provers, (that once exploited, will give a flavor of a parallel implementation of the meta-language interpreter) can be found in [3].

5 Conclusions and future work

In this paper we have presented a preliminary analysis aiming to point out how parallelism can be exploited in implementing the higher-order logic language λProlog. Starting from the observation that non-determinism is the main source of parallelism in standard Prolog, we have analyzed non-determinism inherent to the $\mathcal{P}$-derivation process, i.e. the resolution algorithm implemented by existing λProlog interpreters. In doing that, a number of analogies and a number of differences with respect to the Prolog case have been pointed out, and new sources of parallelism have been discussed and the main problems to deal with them pointed out. In particular, the importance of finding effective policies to manage the exploration of the search tree and to schedule parallel activities has been stressed. Also, the relevance in λProlog of sharing in λ- terms manipulation suggested that shared memory models or incremental copying should perform better than closed environments techniques proposed for implementing Prolog on multicomputers.
Another result of our analysis is the independence condition for AND-parallelism given in section 3.4, which represents the basis for detecting independent AND-parallelism through static analysis or run-time checks. Moreover, after some general comments about architectural issues, a possible parallel execution model for λProlog has been sketched on the basis of the analysis developed. Finally we started to analyze the conditions that allow the execution of the theorem prover by means of parallel processes to give evidence of how much parallelism can be effectively extracted from real λProlog applications. A further examination of actual parallelism inherent to real λProlog applications is needed before proceeding further in parallelizing the interpreter.In this respect, first of all we plan to start from available programs and data, like those reported in [11], in order to characterize quantitatively inherent parallelism in λProlog. Second, existing sequential implementations should be reviewed to understand how much of the effort spent to develop them can be actually reused in a parallel environment.In particular, we are looking for a better characterization of the execution model outlined in section 4.2. At last, the impact of static analysis on par-

allelization should be considered.λProlog is a much more structured language than Prolog, and sophisticated analysis could be carried out to derive useful properties. In particular, starting from the characterization given in section 3.4 of independent AND-parallelism, it would be interesting to find methods for statically detecting independent goals.

Acknowledgments

We would like to thank M. Martelli and D. Miller for their helpful comments and encouragement to begin this research.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass, 1986.

[2] F. Arcelli, F. Formato and G. Iannello. On the parallel implementation of the higher-order logic language λPrologTR-CPS- 014-93, July 1993.

[3] F. Arcelli, F. Formato and G. Iannello. On the Parallel Implementation of λProlog as a Specification Theorem Prover Language. TR-CPS-024-94, feb., 1994.

[4] P. Brisset and O. Ridoux. The compilation of lambda- Prolog and its execution with Mali. Rapport de Recherche n. 1831, INRIA, Jan. 1993.

[5] S.A. Delgado-Rannauro. OR-Parallel Logic Computational Models. In Kacsuk and Wise, editors, *Implementation of Distributed Prolog*, pages. 3-26,Wiley, 1992.

[6] C. Elliot and F. Pfenning. A semi-functional implementation of a higher-order logic programming language. In P. Lee, editor, *Topics in Advanced Language Implementation*, pages 289-325, MIT Press, 1991.

[7] A. Felty. Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language. Ph.D Thesis, University of Pennsylvania, Aug. 1989.

[8] G. Huet. A unification algorithm for typed λ-calculus. In *Theoretical Computer Science*, 1, pages 27-57, 1975.

[9] B. Jayaraman and G. Nadathur. Implementation techniques for scoping constructs in logic programming. In K. Furukawa, editor, *Proc. 8th Int. Conf. on Logic Programming*, pages 871-886, MIT Press, Paris, France, 1991.

[10] C.R. Manning. Acore: Design of an Actor Core Language and its Compiler. MIT Master's Thesis in Computer Science, May 1987.

[11] S. Michaylov and F. Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Proc. Conference Record of the Workshop on the λProlog Programming Language*, Philadelphia, July-August 1992.

[12] D. Miller and G. Nadathur. Some uses of higher- order logic in computational linguistic. In *Proc. of the 24th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, Morristown, New Jersey, 1986.

[13] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symposium on Logic Programming*, S. Haridi, editor, Sep.1987.

[14] D. Miller and G. Nadathur. λProlog version 2.7., Distributed in C-Prolog and Quintus Prolog source code, Aug. 1988.

[15] D. Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. In *Journal of Logic and Computation*, **1**, 4, pages 497-536, 1991.

[16] G. Nadathur. A higher-order logic as the basis for logic programming. Ph.D Dissertation, University of Pennsylvania, May 1987.

[17] G. Nadathur and D. Miller. An overview of λProlog. In Bowen and Kowalski, editors, *Proc. of the Fifth International Conference and Symposium on Logic Programming*, pages 810-827, MIT Press, 1988.

[18] G. Nadathur, B. Jayaraman and D.S. Wilson. Implementation Considerations for Higher-Order Features of Logic Programming. Submitted for publications, Nov. 1992.

[19] L.C. Paulson. Natural deduction as higher-order resolution. In *Journal of Logic Programming*, **3**, pages 237- 258, 1986.

[20] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149-181, Cambridge University Press, 1991.

[21] M.J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, 1987.

[22] S. Raina, D.H.D. Warren and J. Cownie. Parallel Prolog on a Scalable Multiprocessor. In Kacsuk and Wise, editors, *Implementation of Distributed Prolog*, pages 27-44, Wiley, 1992.

[23] A. Takeuchi. *Parallel Logic Programming*, Wiley, 1992.

[24] H. Waldemar. Concurrent Smalltalk on the Message- Driven Processor. Tech. Rep. 1321, MIT Artificial Intelligence Laboratory, Sept. 1991.