# - FLENG Prolog -
# The Language which turns
# Supercomputers into Parallel Prolog Machines

Martin Nilsson and Hidehiko Tanaka
The Tanaka laboratory, Information Engineering, Department of Electrical Engineering,
The University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo 113

**Abstract:** This paper suggests a new way of executing logic programming languages, using small-grain parallelism on vector parallel computer architectures. The main topic is the general purpose language FLENG Prolog. This is a logic programming language for arbitrary architectures, but is especially designed to run efficiently on vector architectures. The most important contribution of the paper is the described combination of the language FLENG Prolog with vector architectures.

## 1. Introduction

In this paper we will study

- **Why** supercomputers could be suitable for logic programming;
- **What** kind of programming language to use;
- **How** to implement it and execute it.

The main, original contribution of this paper is the combined answers to these questions.

The paper emphasizes on the FLENG Prolog language (hereafter we will just write FLENG), which is a logic programming language. It is a general purpose language for all kinds of architectures, although special consideration has been paid to vector parallel architectures. FLENG uses committed choice non-determinism, and descends mainly from GHC (Ued 86a) and (Kernel) Parlog (CG 84a), (CG 85), which in their turn are strongly related to Concurrent Prolog (Sha 83). Oc (Hir 85) is a language which was developed independently, and from a completely different standpoint, but interestingly enough still is quite similar to FLENG.

The recursive acronym "FLENG Prolog" stands for "FLENG has Logic Explicit and No Guard goals, Programming logic," i.e. it is much like GHC, but does not have any guard goals (commitment occurs immiediately after head unification), and the logical semantics and control of execution is explicitly represented, not automatically "AND/ORed" like Prolog. (In fact, FLENG relates much to GHC as Kernel Parlog relates to Parlog.)

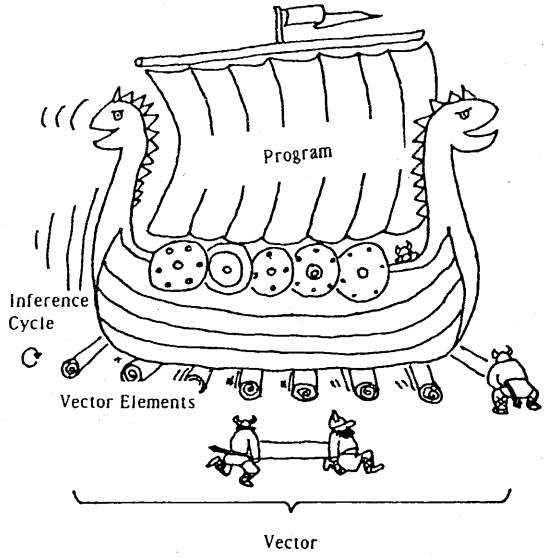
Some other differences from similar languages concern unification and system predicates. FLENG is described in more detail, with examples, in sections 3 and 4.

Although FLENG is designed to fit a variety of parallel architectures, it is particularly suitable for vector parallel (SIMD) architectures. Why? Contrastingly, current research efforts on parallel logic programming today focus on distributed (MIMD) systems, i.e. where individual processors are quite independent. Theoretically, such systems should be the most suitable for general, inhomogeneous parallel execution. However, distributed systems have serious bottlenecks in the form of communication between processors (and memories). For this reason, very few such systems exist.

On the other hand, there are already many different commercial computers with vector architecture. They don't have as serious communication problems, but their problem is that, roughly speaking, all "processors" must perform the same operation on their data. The way to execute a program in a conventional language is to use a "vectorizing" compiler. This compiler tries to detect loops in programs and convert them to special vector operations which operate on an entire matrix at a time. Vectorizing compilers are usually for Fortran, but there is also interesting ongoing work to implement a compiler for OR-parallel Prolog (Kan 85). Unfortunately, often only a small part of the processing can be done by vector operations, and the rest has to be done by a traditional scalar processor attached to the vector processor, with weak total performance as a result.

Our approach is completely different. Instead of fitting parts of a program to the architecture by a compiler, we want the programming language (FLENG) to fit the architecture from the beginning. We want this language to have very few primitive operations, which can all be executed in a fast, tight loop. Then, we can make this loop run entirely in the vector processor, with minimal

scalar processor interference. One can imagine the program as in a way "floating" or "rolling" on many processing elements.




The idea why Logic Programming Languages are especially suitable for execution on vector computers is as follows: The execution cycle of logic programming languages is only a very basic inference cycle. An inference cycle is hardly anything more than clause head unification, plus execution of system primitives (evaluable predicates) for arithmetic, etc. However, the design of the unification and the system primitives are extremely important. Primitives must be as few and as simple as possible, since they will directly influence the execution cycle size and thus the execution time. They should be real time operations whenever possible. At the same time, they must be powerful enough for a general purpose programming language. Unification is the key to process synchronization and control, but can behave very unpolitely if treated carelessly.

We do not yet have a working implementation of the language on a vector computer, but we have implemented it in a pseudo-parallel way on some conventional sequential computers. For the Vax 8600 implementation, where FLENG is compiled to Franz Lisp, the speed is about 16 kLIPS. We don't believe that the vector parallel FLENG implementation we are working on will give very impressing benchmark results, since our target computer is not at all intended for this kind of application, but we believe that with small extensions, as described in section 6, vector architectures may be very suitable for parallel execution of Logic Programming Languages.

The structure of this paper is as follows: Section 2 gives the general ideas and goals behind the design of FLENG. A definition of FLENG is informally given in section 3. This definition is illustrated with examples and motivated in more detail in section 4. In section 5, translation between FLENG and other parallel languages is discussed. Section 6 discusses implementational issues of FLENG, especially for implementation on a vector architecture. Our results so far are given in section 8, and related work is cited in section 7. The research is summarized and discussed in section 9.

## 2. FLENG Design Principles

We want the language to be on the lowest possible level, and at the same time, easy to use, powerful (especially regarding parallelism), and machine independent.

Why do we want/what do we mean by a low-level language? A low-level language fits well to the machine architecture, and is thus easy to implement, and will execute quickly on this architecture. Of course, there is a big trade-off between the language level and its ease of use. Traditional Prolog is easy to use and machine independent, but is quite sequential in nature. Its underlying control mechanism is quite involved, and does not map easily on any machine architecture. We would like our ideal language to be as simple, easy to use, and machine independent as Prolog or GHC, while making it map more closely to the machine architecture.

As for expressional power, we want the parallel abilities of GHC, Parlog, or Concurrent Prolog. We also want programs to be first-class data structures, and we do not want any discrimination of data structures. For instance, we want to be able to detect the type of all data structures occurring in the execution, including variables. This is something which is often impossible in other logic programming languages. The reason is usually that if the language has such abilities, it will be impossible to read its statements declaratively as logical assertions.

This is a dilemma. According to our experience, primitives like metacall and type detection are really needed, especially in a low-level languge, where we cannot resort to machine language subroutines. FLENG's solution is the following:

We do not require that a general FLENG clause should be readable as a statement in predicate

logic. Instead, we require that a logical statement easily can be expressed in FLENG.

This means that a purely logical language, if wanted, easily can be compiled to, or implemented on top of the machine language FLENG.

The logical interpretation of a FLENG clause is given explicitly by the clause itself (this is explained in more detail in sections 3.1 and 4.1). In practice, FLENG programs will not look very different from GHC programs. The reason for that is that the AND/OR operational semantics (imposed by guards) in GHC is not necessary for, and seldom used for control of execution: the synchronization by shared variables is quite sufficient.

Note that FLENG could be interpreted logically in the same way as GHC clauses without guard goals. However, we feel that such a semantics may be a bit artificial and not to the point of the low-level language FLENG.

## 3. FLENG Definition Outline

### 3.1 Execution

A FLENG program is a set of clauses, looking just like Prolog clauses, with a head part, and a body of goals.

$$H :- G_1, G_2, \ldots G_n .$$

A program is executed by giving it a set Q of goals. Each goal is executed independently, in any order. A goal G is executed by removing it from Q and matching it with the head of all clauses in the program. For the first clause with matching head, that clause is comitted to, and the body goals of that clause are added to Q. Matching is like unification, but as in GHC, variable bindings may not be made in G (exported) during the matching. If a match of a variable in G with a non-variable is attempted, this matching is suspended, and resumes when the variable has been bound by another matching.

The execution of a goal is not associated with success, failure, or any logical truth values.

We have an additional fairness requirement: If a clause is committed to, then the execution of every one of its body goals must be guaranteed to start in the future; there may be no indefinite postponement.

### 3.2 Datatypes

There are four basic data types in FLENG: symbols, numbers, variables, and lists. Lists ("conses") have two fields, which may contain any FLENG data type. This is the list approach of data structure construction. Another common approach in other languages is the record approach, where structures are built of varisized arrays. This latter approach is used in Dec-10 Prolog, for instance. It is not so suitable for FLENG, since it makes execution more complicated and inhomogeneous.

### 3.3 System Predicates

The most important feature of FLENG primitive predicates is that it must be possible to detect their result. In other words, they must report completion in a detectable way, for instance by binding an argument variable to a non-variable, when the primitive has completed. Sometimes this is a natural effect of execution (arithmetic), and sometimes an extra argument has to be used (unification). None of the system primitives wait for their arguments to be instatiated. Such waiting can easily be implemented inside FLENG by interface predicates (cf. section 3.4). The following three predicates are considered primitive:

• unify(Result,X,Y)
Unification of term X with term Y. The variable Result is bound to true if unification is successful, and false if it is unsuccessful. Any bindings done to variables in X and Y remain in effect, even if unification fails. When Result is not needed, we can use "=" as defined by
(X = Y) :- unify(Result,X,Y).

• compute(Operation,X,Y,Result)
This primitive is used for arithmetic, comparison of numbers or pointers, and bitwise numeric operations. The variable Result is bound to either a number for arithmetical and bitwise operations, or true or false for comparisons. The argument Operation should be a constant identifying the desired operation. It can be one of the following: $+$, $-$, $*$, $/$ (arithmetic), and, or, xor (bitwise), $=$, $\leq$, sametype (comparison). $=$ and sametype allow unbound variables as arguments.

• call(X)
This is the meta-call primitive. It does not need a result variable, since that is considered to be reported by an argument (a subterm) of the term X.

### 3.4 Convenience

For user convenience, we have decided to add one feature to FLENG, which is not strictly necessary. This feature is a "bind-to-non-variable" annotation ("%") on variables in the head of a clause. If the annotation is used on a variable in the head, it is valid for that variable in the whole head. The meaning is that during head matching, an annotated variable in the head of a candidate clause must be matched with something else than a variable, or otherwise matching suspends. The same effect can be achieved by busy waiting, in a more clumsy and inefficient way.

Using this annotation, we can easily define a waiting version of the system primitive compute:

```
waitcompute(Op,%X,%Y,R) :- compute(Op,X,Y,R).
```

## 4 Explanations, Motivations, and Examples

### 4.1 Execution

In logic programming languages with guard goals, it is necessary to associate the execution with truth values, or at least success and suspension, since the result of the execution of guard goals determines whether commitment is possible. This leads to considerable overhead, difficulties in implementation of the language, and finding a proper architecture.

Another problem with guard goals is that variable binding schemes become complicated in the general case: Vaguely expressed, variables may pass bindings inside the guard, but not outside the guard. It becomes necessary to carefully remember which variable belongs to which environment.

These are the main reasons why FLENG does not have any guard goals. Still, FLENG does not lose expressional power because the synchronization mechanism by shared variables is perfectly enough for controlling the execution. In fact, practical programs in languages like GHC relatively seldom use guard goals, and when they are used, they are often only simple calls to system predicates. Furthermore, the implicit conjunctions of executed goals are used rarely. This means that typical GHC programs easily can be rewritten to FLENG. (Actually, "Flat" GHC - i.e. GHC with only system predicates as guard goals - can comparatively easily be compiled to FLENG.)

Since we don't emphasize that a FLENG clause has to be implicitly readable as a logical statement, it might be argued that logic programming is impossible in FLENG. This is not correct. One reason is that the logic usually implicit in e.g. Prolog easily can be expressed explicitly by goal arguments in FLENG programs. Actually, compared to the implicit conjunction of goals in the body of a Prolog clause, FLENG offers greater flexibility. For instance, with the following definitions:

```
and(true,true,Result) :- Result = true.
and(false,_,   Result) :- Result = false.
and(_,false,   Result ) :- Result = false.
```

we can get the parallel AND of two goals f and g by

```
h(X,Y,Result) :-
   f(X,ResultF), g(y,ResultG),
   and(ResultF,ResultG,Result).
```

but as easily as AND, we could also define OR, exclusive OR, NOT, or any other boolean function.

Guard goals are typically used for testing arguments before commitment. This testing can as well be done before the call, as in the guard: The result of the test is passed as a parameter in the call, and is tested when matching clause heads. An example can clarify this: Consider the following GHC program:

```
prog(N) :- N < 17 | a(N).
prog(N) :- N ≥ 17 | b(N).
```

with the call prog(N). This program has a FLENG equivalent

```
prog(N) :- compute(<,N,17,Less), prog1(Less,N).
prog1(true,N) :- a(N).
prog1(false,N) :- b(N).
```

### 4.2 Unification

Recent logic programming languages for parallel execution seem to be converging, and have become quite similar. One of the most important differences is unification. Unification has central function for synchronization, control, and parameter passing. On the other hand, it can cause big problems by creating cyclic structures and deadlocks.

In this section, we consider mainly the unify system predicate. Head matching is similar but is somewhat easier to handle.

An important question is the grain size of unification: Should unification of two terms be

split up in several unification processes to be executed in parallel? Or should unification be executed as an indivisible operation? Apparently, the latter alternative is not very good, because unification may take very long time. Parlog and GHC try to split up unification in many smaller unification processes. However, in practical programs, almost all unifications are of small size. Splitting up unification in many different processes will increase the overhead, and for average small size unifications, execution may even be slowed down.

The grain size of unification in FLENG is instead intended to be slightly larger than the average size of unification. Larger unifications have to be interruptable. The structure of unification is often such that a component in unification is often an implied test, and other components are for parameter passing. Thus we do not believe that sequential unification will impair the speed seriously. We do not prescribe sequential execution of unification in FLENG, although we believe it may be the best solution in a real implementation, especially for the purpose of detection of cyclic and other ill-behaving data structures. Note that even if execution of the unify predicate is sequential, unification of two terms f(X,Y) and f(U,V) can be parallelized by

unify(R1,X,U), unify(R2,Y,V), and(R1,R2,R)

This is one of the very important roles of the result argument. Another role is to allow variables to be full-fledged data structures. Without an extra argument, it will be impossible to non-destructively determine when a unification of a variable with another variable has been completed.

Variable bindings in the unification in FLENG remain in effect even if the unification fails, e.g. because two different constants are compared with each other. In this case, the system predicate does not suspend, but completes execution and reports failure, maintaining all variable bindings made so far. Note that in this way, unify can be used for testing both equality and inequality with correct behaviour from a logical point of view: Even if variables occuring in the arguments of unify are instantiated later, the meaning of unify does not change.

The single-assignment property of variables in a logical programming languages has a hidden, serious implication: Either we must have occur check before binding a variable, or we must be able to handle cyclic structures: After we have bound a variable in the middle of unification, we cannot change our mind, if we later find that the binding was a mistake.

As can be seen from this, unification is a delicate subject. Fortunately, there seem to be unification algorithms which can solve this problem satisfactorily (NT 86). A forthcoming paper will report more detailed results on this problem.

### 4.3 The Other Two System Predicates

For any low-level logic programming architecture, it is important to reduce the system predicates to a minimum. The architecture should be kept as small and simple as possible. In particular for vector architectures, it is important to minimize the size and number of system predicates: If the vector processor has to perform a (relatively) very rare operation for some process, all the other processes will probably be idle, with very low vector utilization as a result. If necessary, such operations should be performed by a front-end processor, also running FLENG, but with an extended set of system predicates for I/O and database updates.

These primitive system predicates do not necessarily check for exceptional conditions, like division with zero. Such checking can easily be done in FLENG by interface predicates.

The compute predicate operations are intended to be the basic ones of the underlying machine, like arithmetic (+, -, *, /), bitwise operations for arithmetic (and, or, xor), and pointer and numeric comparison (<, =, sametype). All these operations have a similar form, with two input arguments, a simple (real time) operation on local data, and one output. The ideal architecture has something similar to an indirect "execute" instruction, which can take the intended operation as one of its arguments. Note that the sametype and = comparisons are meaningful also for arguments which are unbound variables. For pointers, = compares addresses.

The meta-call predicate has only one argument. This not only ensures a clean semantics (cf. (Ued 86)), but is all that is needed for a meta-call facility. The completion of the call predicate itself is not interesting. The completion of the execution of the called predicate can report completion through one of its own arguments. Thus, the meaning of call is just the same as writing the called goal in the body of the clause.

We do not think the multi-argument meta-call

approach (CG 84b) is very attractive, because it is not only a meta-call, but implies an additional control construct which is very hard to implement. (For the reader familiar with Lisp terminology, it resembles not only a call to EVAL, but a call to EVAL inside ERRORSET, or UNWIND-PROTECT.) The control problem the multi-argument type of call is intended to solve, is instead swept under the carpet, and will appear when trying to implement the language. We think that FLENG solves the problem in a nicer way by letting predicates report about completion themselves.

### 4.4 Example Program: Generating Primes

(Ued 86a) gives the following GHC program for generating primes:

```
primes(Max,Ps) :- true | gen(2,Max,Ns),sift(Ns,Ps).

gen(N,Max,Ns) :- N<=Max | Ns=[N|Ns1], N1:=N+1,
                          gen(N1,Max,Ns1).
gen(N,Max,Ns) :- N> Max | Ns=[].

sift([P|Xs],Zs) :- true | Zs=[P|Zs1], filter(P,Xs,Ys),
                          sift(Ys,Zs1).
sift([],    Zs) :- true | Zs=[].

filter(P,[X|Xs],Ys) :-
    X mod P=:=0 | filter(P,Xs,Ys).
filter(P,[X|Xs],Ys) :-
    X mod P=\=0 | Ys=[X|Ys1], filter(P,Xs,Ys1).
filter(P,[],    Ys) :- true | Ys=[].
```

To translate this program to FLENG, we have to rewrite the non-trivial guard parts. This is quite easy as can be seen from the resulting FLENG program:

```
primes(Max,Ps) :- gen(2,Max,Ns), sift(Ns,Ps).

gen(N,Max,Ns) :-
   greater(N,Max,Greater),gen1(Greater,N,Max,Ns).
gen1(false,N,Max,Ns) :-
   Ns=[N|Ns1], add1(N,N1), gen(N1,Max,Ns1).
gen1(true,N,Max,Ns) :- Ns=[].

sift([P|Xs],Zs) :-
   Zs=[P|Zs1], filter(P,Xs,Ys), sift(Ys,Zs1).
sift([],    Zs) :- Zs=[].

filter(P,[X|Xs],Ys) :- mod(X,P,Mod), zero(Mod,Zero),
   filter1(Mod,P,[X|Xs],Ys).
filter(P,[],    Ys) :- Ys=[].

filter1(true,P,[X|Xs],Ys) :- filter(P,Xs,Ys).
filter1(false,P,[X|Xs],Ys) :- Ys=[X|Ys1],
filter(P,Xs,Ys1).
```

with the following definitions:

```
greater(%X,%Y,Result) :- compute(<,Y,X,Result).
add1(%X,Y) :- compute(+,X,1,Y).
mod(%X,%Y,D) :-
   compute(/,X,Y,Q), times(Y,Q,P),
difference(X,P,D).
times(%X,%Y,P) :- compute(*,X,Y,P).
difference(%X,%Y,D) :- compute(-,X,Y,D).
zero(%X,R) :- compute(==,X,0,R).
```

For a more detailed description of the execution of the primes program, we refer to (Ued 86a). Here, we would like to point out that a typical FLENG program does not become much different from a typical (Flat) GHC program. For general GHC programs, the translation of the guard is harder, but it seems that guards are usually of a simpe kind, and even where complicated user-defined guards occur, the guard often is just a test which does not try to instantiate variables in the caller. Such a guard is also easy to translate into FLENG.

## 5. Translation between FLENG and similar Languages

### 5.1 FLENG to GHC

Except for the system primitives, FLENG should be executable in GHC. The system primitive compute in FLENG can detect and identify variables, which is impossible in GHC. Also, the unification in FLENG reports completion and failed unification, which is impossible in GHC. To execute FLENG, a GHC implementation needs to satisfy the fairness requirement that the execution of goals of committed clauses must be guaranteed to be started some time.

### 5.2 GHC to FLENG

Flat GHC can be compiled to FLENG quite easily. We have written such a compiler in Prolog, which is about 2 pages of code long. The subset of GHC that can be easily compiled to FLENG is in fact larger than Flat GHC: What the compiler needs is knowledge about which variables in guard goals will be used for export, and which will be used for import. This information can be supplied e.g. by declarations or by thorough static analysis.

Partial evaluation may be a useful approach: (SC 86) refers to work by Codish, who has used partial evaluation for translating Concurrent Prolog to Flat Concurrent Prolog. We believe this may be a very interesting approach for translating GHC into Flat GHC, and thus into FLENG.

### 5.3 Other Languages: Parlog and Concurrent Prolog

We dare not say so much about the mutual expressibility of FLENG on one hand, and Concurrent Prolog and Parlog on the other hand, until we have a clear definition of the semantics of the latter languages. In particular, a critical question is the necessary system primitives: Parlog seems to require a meta-call primitive which requires quite an elaborate control mechanism "under the surface." The semantics of unification in Concurrent Prolog is also complicated, and may not be easily expressible in FLENG.

## 6. FLENG Architectures

### 6.1 Vector Parallel Architectures

The great advantages with vector parallel architectures are that they have a shared memory, and very fast communication between "processors." Processor data are actually only elements of vector registers, so processors can communicate by just shuffling data in a vector register. Global operations such as scheduling and garbage collection are very much simplified because of the shared memory. The looming dream of a vector architecture is the possibility of scaling up by just extending the vector length.

However, there are also some disadvantages with a vector architecture: The vector processor's operations cannot be specialized for a particular program, i.e. programs cannot be compiled, but an interpreting approach has to be used. It is likely that the vector processor will spend many wasted cycles executing rare system predicates.

### 6.2 Vector Parallel Implementation

We have spent very much time and effort trying to squeeze Prolog systems to minimal size (Nil 83), (Nil 85). Studying minimal implementations has been essential for trying to understand efficient execution of logic programming languages, and particularly so for implementation on a vector architecture: On this architecture, the program interpreter must be coded as an extremely compact loop of vector operations.

For instance, consider unification. Indeed, unification is by far the most complex operation in the inference cycle, including the other system predicates. Unification 1) has to be able to detect cyclic structures, and 2) cannot use any procedure calls, because of the restricted vector operations, but has to be opencoded as a loop. The system predicate unify is very similar to head matching. It is very important that the code of head matching and the unify predicate can be shared as much as possible.

The space for this article does not allow us to go too deeply into details of our implementation for vector parallelism, but below we will roughly sketch the main mechanisms: The kernel of the program is one loop, which basically performs N inferences simultaneously, where N is the vector size. The code in the loop is similar to a traditional interpreter. The restrictions on this code are that it may not contain jump instructions or subroutine calls. Conditionals are only allowed by using mask bits, where a machine instruction can be disabled for a particular vector element by setting the corresponding bit in a mask vector.

- First in the loop, N processes (goals) are taken from a queue of waiting processes. Then, arguments are unified with candidate clause arguments.
- Here, some processes may be suspended, or their unification may already have finished. If so, bits in a mask vector are set to disable unification for those processes.
- After so many steps of unification that most processes have finished matching, i.e. had their mask bits set, unification ends. Unifications which are not yet finished will have to wait until the next turn of the main loop.
- After unification, system primitives are checked for and executed. Processes which do not contain such calls are disabled by mask bits.
- For successful head unifications, new body goals are added to of the queue. Processes which are not yet finished, e.g. because of lengthy unifications, are also put back into the queue. Depending on where in the queue new goals are put, we get different scheduling mechanisms. If new goals are put at the back of the queue, we will get breadth-first scheduling. We can also get something similar to n-bounded depth-first scheduling (Ued 86a). n-bounded depth-first scheduling is when a goal is executed depth-first, in a way similar to Prolog, for n process reductions. Then, pending goals on the stack are added to the end of the process queue, and a new process is started from the beginning of the queue. Suppose we put new processes at the

beginning of a second queue (used as a stack), and now take processes from this queue, for n cycles. Then, we take all the processes in this second queue and add them to the end of the first queue, and repeat the procedure again.

The implementation uses a structure sharing strategy since it seems that the overhead for copying will be too large; copying is not a real-time operation, so it would have to be suspendable in the same way as unification, with a consequent increase in overhead.

### 6.3 Useful Improvements of Vector Computers

A practical problem today is that commercial vector processors (supercomputers) are not so much intended for non-numerical vector processing, which is indicated by the instruction set. Most modern supercomputers have instructions for so called "list vectors" (i.e. vectors containing addresses instead of data, corresponding to indirect addressing for scalar processors). They also have masking operations, which can be used for conditional operations. But what they don't have, and what would be extremely useful for the implementation of system predicates, is an "execute" instruction, such as mentioned in section 4.3. This operation can be used for taking the code of an arithmetic operation as one of its operands. It could then execute many system predicates at the same time. Such an instruction should not be too hard to implement, as it operates only on local data.

Another, even more important new instruction would be a synchronization instruction: It often happens that several processes want to store a value in the same memory location. We want to grant exactly one of them the right. One way to do this on a vector architecture is that we first write the ID numbers of the processes (i.e. the vector indices) in those locations. We wait until all values have been written, and then we read them back. A process which reads back the same value as it wrote, is granted the permission to write the location. The disadvantage with this is the long time delay for accessing the memory, although the permissions could be found by only looking at the addresses locally, in a vector register. A much more elegant way would be to have a vector instruction which sets a bit in a mask register for exactly one vector element of several elements, containing the same address.

### 6.4 Other parallel architectures for FLENG

A distributed architecture for FLENG has the big advantage of allowing compilation and static optimization of programs. For FLENG, no "invisible" control is necessary for relating parent and child processes, as their only links are through shared variables. On a distributed architecture, several processes can be combined to run very quickly as long as they run on the same processor. However, FLENG shares a problem with other similar languages on distributed architectures, namely how to efficiently communicate bindings of shared variables.

It is worth mentioning that the independent execution of FLENG goals suggests that a dataflow architecture for FLENG may fit well.

## 7. Related work

FLENG descends directly from GHC (Ued 86) and Parlog (CG 84a). The intention has been to find a low-level language, which fits current machine architectures well, but still allows programming as easily as, say, Prolog or GHC. The basic execution features are thus very close to those of GHC and Parlog. There is another recent language called Oc (Hir 85). Oc seems to be rather formally developed, with heavy emphasis on program transformation and partial evaluation. The paper describing it is very brief, so the properties of Oc are not so clear, but its basic execution mechanism seems very similar to FLENG's. However, its treatment of unification and system predicates is quite different.

The most intricate facet of FLENG, and maybe of logic programming languages in general, is unification. Recent parallel logic programming languages differ much in this respect. GHC attempts to be as general as possible and allow minimum grain parallelism, by a property called "antisubstitutability." This rougly means that unifications of large terms automatically can be split in smaller, independently executed by inserting ("antisubstituting") intermediate variables. However, there are several dangers with this approach: The completion of unification of a variable with another variable cannot be detected. This means that, for instance, the common merge program of two streams may not work properly if we want to pass general terms in the stream. It is impossible to know whether a subterm of an element in the stream is a variable, or just waits for being bound to a non-variable. Also, this approach makes proper handling of cyclic structures hard. The antisubstitutions themselves are not to blame here: the real cause of the problem is that the completion of unification cannot be detected, and this is one of the reasons

why the unification primitive in FLENG has three arguments.

As for emphasizing on vector architectures for parallel logic programming languages, there are very few papers touching on this subject. (Kan 85) describes a possible implementation of how a vectorizing compiler could compile OR-parallel Prolog to fit a vector processor, by for instance moving tail recursive loops to the vector processor. This approach can only use relatively large-grain parallelism, and will rely much on a scalar processor. This contrasts to our method, where execution lies almost entirely in the vector processor. Our approach is entirely parallel in the sense that it is impossible to give a "program counter," or tell where in the program the computer is currently executing.

Another paper (BL 85) discusses the use of vector architectures for AI-languages. This article specializes on CAP, a computer with SIMD architecture. The article describes how to use the particular features of this computer for implementing the RETE matching algorithm for the OPS5 language, and how to implement an algorithm for semantical networks. However, these results are not applicable on vector parallel architectures in general, or for logic programming implementations.

## 8. Results

We are working on some different approaches to implementations of FLENG. So far, we have simple structure sharing and copying interpreters for FLENG written in Franz Lisp, for (sequential) Vax computers. We also have a simple compiler from FLENG to Franz Lisp. In these implementations we use n-bounded depth-first scheduling. $n = \infty$ represents depth-first search with minimal overhead, and $n = 1$ represents breadth-first search with maximal overhead, where the body goals of a reduced goal are immediately put at the end of the queue. The speed of compiled code for Vax 8600 was about 16 kLIPS for $n = \infty$, and about 4 kLIPS for $n = 1$. A normal setting of n is about 100. In this case the overhead becomes 5-10%. As a comparison, the speed of the C-Prolog interpreter on this computer is about 5 kLIPS. We estimate that the speed of FLENG can be increased by direct compilation to machine code, and some more optimizations. The structure sharing interpreter is as expected much slower than compiled code, around a factor of 20.

The different implementations of FLENG will be very different, depending on if the target architecture is a vector parallel computer, a distributed computer, or a sequential computer (pseudo-parallel implementation). There are still open questions for further research, e.g. concerning I/O and indexed data structures. Although it may be argued that these are only implementation details, we think they might influence the design so much that they had better be remembered at an early stage of the design.

## 9. Conclusions

We do not estimate the inherent speed of FLENG to be very much different from other similar languages on the same machine. On the other hand, the implementation will be comparatively simple, and may mean that a tailored architecture can be simplified, and speeded up, for instance by a RISC approach.

Our test implementations show that FLENG can be implemented easily and efficiently on a sequential computer. An implementation for an existing vector parallel computer will probably not be very fast since, the computer is not originally intended for this kind of application, and thus lacks some important features, most notably for process synchronization. It still seems, however, that a vector parallel design may be a very promising approach for future logic programming computers.

## 10. Acknowledgements

This research was possible thanks to a generous grant from the Japanese Ministry of Education. We are grateful to the members of the special interest group of inference machines, at the University of Tokyo, and to the members of the parallel programming systems working group, at the Institute for new computer technology. We would also like to thank the unknown referees of this paper for several constructive comments.

## 11. References

(BL 85) Brooks, R. and Lum, L.: "Yes, An SIMD Machine Can Be Used For AI." In Proc. of the Int. Joint Conf. on Artificial Intelligence. Los Angeles, 1985. p 73-79.

(CG 84a) Clark, K.L. and Gregory, S.: "PARLOG: Parallel Programming in Logic." Res. Rept. DOC 84/4. Dept. of Computing, Imperial College of Science and Technology, London. 1984.

(CG 84b) Clark, K.L. and Gregory, S.: "Notes on the Implementation of PARLOG." Res. Rept. DOC

84/16. Dept. of Computing, Imperial College of Science and Technology, London. 1984.

(CG 85) Clark, K.L. and Gregory, S.: "Notes on Systems Programming in PARLOG." In Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo 1984.

(Hir 85) Hirata, M.: "Description of Oc and its Applications." In Proc. Second National Conf. of Japan Society of Software Science and Technology. p 153-156. (In Japanese)

(Kan 85) Kanada, Y.: "High-speed Execution of Prolog on Supercomputers." In Proc. 26th Programming Symp., Information Processing Society of Japan. 1985. p 47-55. (In Japanese)

(Nil 83) Nilsson, M.: "FOOLOG - A Small and Efficient Prolog Interpreter." Tech. Rept. no. 20. UPMAIL, Comp. Science Dept. Uppsala, Sweden. 1983.

(Nil 84) Nilsson, M.: "The worlds shortest Prolog interpreter?" In Campbell, J. (ed): "Implementations of Prolog". Ellis Horwood Ltd., Chichester, UK. 1984. p 87-92.

(NT 86) Nilsson, M. and Tanaka, H.: "Cyclic Tree Traversal." To appear in Proc. 3rd Int. Conf. on Logic Programming. London. 1986.

(Sha 83) Shapiro, E.Y.: "A Subset of Concurrent Prolog and its Interpreter." Technical Report TR-003, Institute for New Generation Computer Technology. 1983. Tokyo.

(SC86) Sterling, L. and Codish, M.: "Pressing for Parallelism: A Prolog Program Made Concurrent." J. Logic Programming, No. 1, 1986. p. 75-92.

(Ued 86a) Ueda, K.: "Guarded Horn Clauses." Doctor's Thesis. Information Engineering Course, The University of Tokyo. 1986. (This very read-worthy thesis is combines some of Ueda's earlier papers with new material.)

(Ued 86b) Ueda, K.: "On the Operational Semantics of Guarded Horn Clauses." To appear as Technical Memorandum, Institute for New Generation Computer Technology. 1986. Tokyo.