A note on Data-Parallelism and (And-Parallel) Prolog*

Manuel V. Hermenegildo
Manuel Carro

Universidad Politécnica de Madrid (UPM)
Facultad de Informática
28660-Boadilla del Monte, Madrid – Spain
{herme,mcarro}@fi.upm.es

(Extended abstract)

1 Introduction

The term *data parallelism* is generally used to refer to a parallel semantics for (definite) iteration in a programming language such that all iterations are performed simultaneously, synchronizing before any event that directly or indirectly involves communication among iterations. It is often also allowed that the results of the iterations be combined by reduction with an associative operator. In this context a *definite iteration* as an iteration where the number of repetitions is known before the iteration is initiated.

Data parallelism has been exploited in many languages, including Fortran-90 [33], C* [42], Data Parallel C [20], *LISP [41], etc. Recently, much progress has been reported in the application of concepts from data-parallelism to logic programming, both from the theoretical and practical points of view, including the design of programming constructs and the development of many implementation techniques [43, 37, 5, 8, 28, 47, 34, 4, 6, 7].

On the other hand, much progress has also been made (and continues to be made) in the exploitation of parallelism in logic programs based on control-derived notions such as and-parallelism and or-parallelism [11, 13, 14, 27, 21, 30, 31, 44, 32, 1, 2, 18, 19, 17, 16, 29, 40, 45, 38]. It appears interesting to explore, even if only informally, the relation between these two at first sight different approaches to the exploitation of parallelism in logic programs. This informal exploration is one of the purposes of this note (the other being to explore the related issue of fast task startup).

1.1 Data Parallelism and And-Parallelism

It is generally accepted that data parallelism is a restricted form of and-parallelism: the threads being parallelized in data-parallelism are usually the iterations of a recursion, a type of parallelism which is obviously also supported in and-parallel systems. The particular restrictions imposed over general purpose and-parallelism vary slightly from one proposal to another. In general, only recursions of a certain type are allowed to be executed in parallel. Also, limitations are posed on the level of nesting of these recursions (e.g. sometimes no nesting is allowed). Often, a priori knowledge of the sizes of the lists (or arrays) being operated on is required (but this data is also obtained dynamically in other cases). Furthermore, other "safeness"-related restrictions are imposed among the iterations being parallelized, such as requiring them to be deterministic, to have only one alternative, and/or to be independent.

It is interesting to note that the restrictions that general purpose systems impose on the goals which can be executed in parallel (such as independence and/or determinacy applied at different granularity levels [23, 36, 38, 12, 24] are generally the minimal ones needed in order to ensure vital desired properties

such as correctness of results or "no-slowdown", i.e. that parallel execution be guaranteed to take no more time than sequential execution. Data-parallel programs, since they are after all and-parallel programs, have to meet the same restrictions from this point of view and this is the motivation for the "safeness" conditions mentioned before.

One of the central ideas in data-parallelism, as presented in many proposals, is to impose *additional* restrictions to the parallelism allowed, in order to make possible further optimizations in some important cases, in return for a certain loss of parallelism due to not being able to deal with the general case. I.e., the additional restrictions imposed have the obvious drawback that they limit the amount of parallelism which can be obtained with respect to a more general purpose and-parallel implementation. On the other hand, when the restrictions are met, many optimizations can be performed with respect to an unoptimized general purpose and-parallel model, in which the implementation perhaps has to deal with backtracking, synchronization, dynamic scheduling, locking, etc. A number of implementations have been built which are capable of exploiting such special cases in an efficient way (e.g. [6, 7]).

In a way, one would like to have the best of both worlds: an implementation capable of supporting general forms of and (and also or!) parallelism, so that speedups can be exploited in as many programs as possible, and at the same time have the implementation be able to take advantage of the optimizations present in data-parallel implementations when the conditions are met.

1.2 Compile-time and Run-time Techniques

In order to achieve the above mentioned goal of a "best of both worlds" system, there are two classes of techniques which have to studied. The first class is related to detecting when the particular properties to be used to perform the optimizations hold. However, this problem is common to both control- and data-parallel systems. The concept of "data parallelism" does not in any way make the task of the compiler or the implementation simpler in this regard. Note that the solution of allowing the programmer to explicitly declare such properties or use special constructs which have built-in syntactic restrictions can be applied indistinctly in both of the approaches under consideration. Thus, we will not deal herein with how the special cases are detected.

The second class of techniques are those related to the actual optimizations in the abstract machine. Given, as we have argued before, that data-parallelism constitutes a special case of and-parallelism, one would in principle expect the abstract machine used in data-parallelism to be a "pared down" version of the more general machines. We believe that this is in general the case, but it is also true that the data-parallel machines also bring some new and interesting techniques.

For the sake of discussion, we will concentrate on the abstract machine of Reform Prolog [6, 7]. In many aspects, the Reform Prolog abstract machine can in fact be viewed as a "pared-down" version of a general-purpose and-parallel abstract machine such as the RAP-WAM/PWAM[26, 21], the DASWAM [40], or the Andorra-I engine [39]. For example, there are a number of agents or workers which are each essentially a WAM. Also, the dynamic scheduling techniques are very similar to the goal stealing method used in the RAP-WAM.

Understandably, there are also some major differences. A first class of such differences is related to the special case of and-parallelism being dealt with. For example, because of the restrictions posed on backtracking among parallel goals, structures like the "markers" of the RAP-WAM are gone. However, it should be noted that the same optimizations can also be done in machines such as the RAP-WAM if the particular case is identified, and without losing the general case [25, 10, 15]. This is also the case with some other optimizations.

On the other hand, a number of optimizations, generally related to the "Reform Compilation" done in Reform Prolog [35] are more fundamental. We find these optimizations particularly interesting because they bring attention upon a very interesting issue the performance of and-parallel systems: that of the speed in the creation and joining of tasks. Because of the special interest of this subject, we will essentially devote to it the rest of this note.

2 The Task Startup and Synchronization Time Problems

The problem in hand can be illustrated with the following simple program:

```
vproc([],[]).
vproc([H|T],[HR|TR]) :-
        process_element(H,HR),
        vproc(T,TR).
```

which relates all the elements of two lists. Throughout the discussion we will assume that the `vproc/2` predicate is going to be used in the "forwards" way, i.e. a ground list of values and a free variable will be supplied as arguments (in that order), expecting as a result a ground list.

2.1 The Naive Approach

This program can be naively parallelized as follows using "control-parallelism" (we will use throughout &-Prolog [22] syntax, where the "&" operator represents a potentially parallel conjunction):

```
vproc([],[]).
vproc([H|T],[HR|TR]) :-
        process_element(H,HR) &
        vproc(T,TR).
```

This will allow the parallel execution of all iterations. Note that the parallelization is safe, since all iterations are *independent*. The program can be parallelized using "data-parallelism" in a similar way.

However, it is interesting to study the differences in how the tasks are started in both approaches. In a system like &-Prolog, using one of the the standard schedulers (we will assume this scheduler throughout the examples), the initial agent, running the call to `vproc/2` would create a process corresponding to the recursion, i.e. `vproc(T,TR)`, make it available on its goal stack, and then take on the execution of `process_element(H,HR)`. Another agent might pick the created process, creating in turn another process for the recursion and taking on a new iteration of `process_element(H,HR)`, and so on. In the end, parallel processes are created for each iteration. Note that all process creation has been a simple consequence of the application of the parallel conjunction operator semantics. This is very attractive in that the same operator which allows parallelism among two goals in any general case, also yields in this particular case the desired result of parallelizing all the iterations of a "loop". However, the approach or, at least, the naive program presented above, also has some drawbacks.

In order to illustrate this, we perform the experiment of running the previous program in the following context. We assume a query "`?- makevector(10,V), main(V,VR).`", where `makevector(N,L)` simply instantiates `L` to a list of integers from 1 to `N`. Thus, we have a list of 10 elements. We use as `process_element/2` a relatively small-grained numerical operation, which serves to illustrate the issue:

```
process_element(H,HR) :-
        HR is ((((H * 2) / 5)^2)+(((H * 6) / 2)^3))/2.
```

Finally, in order to observe the phenomenon, we run the program in &-Prolog on 8 processors on a Sequent Symmetry and generate a trace file, using the following commands:

```
main(V,VR) :-
        start_event_trace,
        vproc(V,VR),
        stop_event_trace,
        save_trace('Eventfile').
```

The trace is then visualized with VisAndOr [9]. The result is depicted in Figure 1 (In VisAndOr graphs, time goes from top to bottom. Vertical solid lines denote actual execution, whereas vertical dashed lines

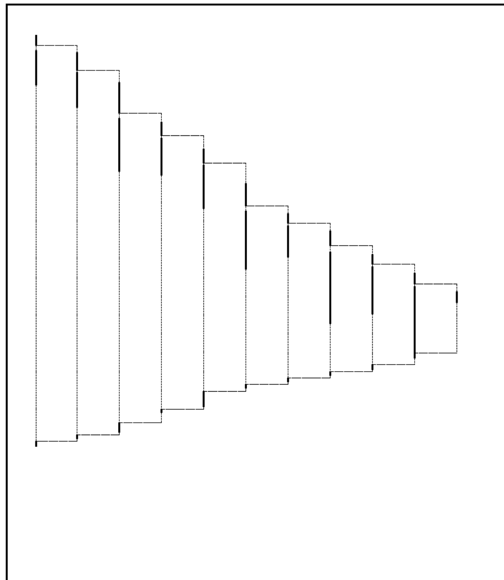

Figure 1: Vector operation, giving away recursion (10 el./8 proc.)

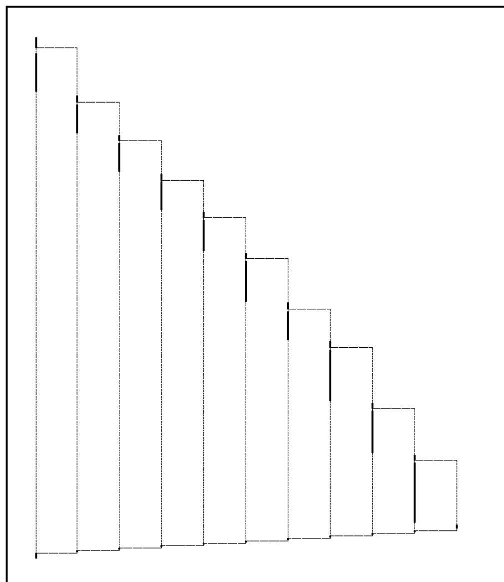

Figure 2: Vector operation (10 el./1 proc.)

represent waits due to scheduling or dependencies. Horizontal dashed lines represent forks and joins.) As can be seen, the initial task forks into two. One is performed locally whereas the other one, corresponding to the recursion, is taken by another agent and split again into two. In the end, the process is inverted to perform the joins. A certain amount of speedup is obtained. This can be observed by comparing to figure 2 which corresponds to the execution of the same program on only one processor – the total amount of time is less. However, the speedup obtained is in fact quite small for a program such as this with obvious parallelism. This low speedup is in part due to the small granularity of the parallel tasks, and also to the slow generation of the tasks which results from giving out the recursion [9].

2.2 Keeping the Recursion Local

One simple transformation can greatly alleviate the problem mentioned above – reversing the order of the goals in the parallel conjunction:

```
vproc([],[]).
vproc([H|T],[HR|TR]) :-
        vproc(T,TR) &
        process_element(H,HR).
```

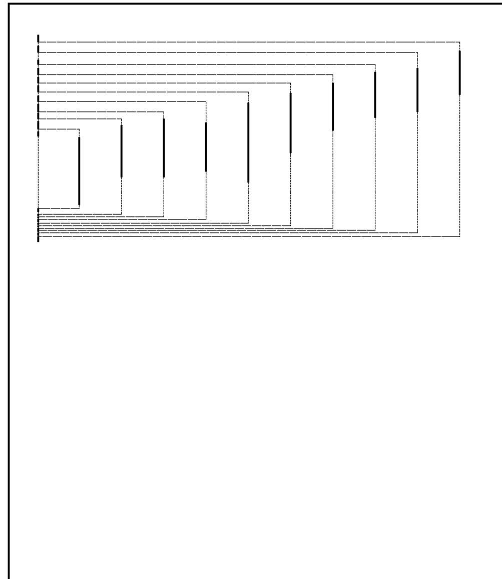

Figure 3: Vector operation, keeping recursion (10 el./8 proc.)

The result of running this program is depicted in Figures 3 (which uses the same scale as Figures 1 and 2) and 4 (which uses full scale to show more detail). The first process can now be observed to keep the recursion local and thus create the tasks much faster, resulting in substantially more speedup. Note that this transformation is in fact in most cases done automatically by the &-Prolog parallelizing compiler. However, the compiler leaves hand-parallelized code as is and this has allowed us before to write and run the program that hands out the goals in the "wrong" way.

Keeping recursions local can speed up the process of task creation, and in most applications, which in general show much larger granularity than this example, task creation speed is not a problem. On the other hand, in numerical applications such as those targeted in data-parallelism, task creation using linear recursion will still be a problem: the speed of the process creating the tasks will become a bottleneck.

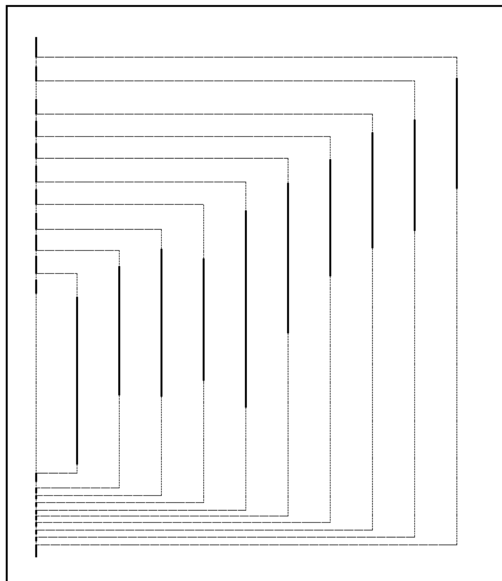

Figure 4: Vector operation, keeping recursion (10 el./8 proc.) (full scale)

2.3 The "Data-Parallel" Approach

At this point it is interesting to return to the data-parallel approach and, in particular, to Reform Prolog. The way this system tackles the problem (we assume that it has already been identified that the recursion is suitable for this technique) is by first converting the list into a vector (and noting the length on the way) and then creating in a tight, low level loop the corresponding tasks, which are simply represented by a pointer to the element of the vector which the task should operate on. The following program allows us to both illustrate this process without resorting to low level instructions and measure inside &-Prolog the benefit that this type of task creation can bring (once the parallel conjunction is set up, each task creation in and-prolog in fact corresponds to pushing two pointers on to a goal stack – the overhead in the previous cases was coming from the recursion and the setup time for each parallel conjunction):

```
main(V,VR) :-
        % Only valid for a 10 element vector!!
        length(V,10),
        start_event_trace,
        vproc(V,VR),
        stop_event_trace,
        save_trace('Eventfile').

vproc([H1,H2,H3,H4,H5,H6,H7,H8,H9,H10],
     [HR1,HR2,HR3,HR4,HR5,HR6,HR7,HR8,HR9,HR10]) :-
        process_element(H1,HR1) &
        process_element(H2,HR2) &
        process_element(H3,HR3) &
        process_element(H4,HR4) &
        process_element(H5,HR5) &
        process_element(H6,HR6) &
        process_element(H7,HR7) &
        process_element(H8,HR8) &
```

```
process_element(H9,HR9) &
process_element(H10,HR10).
```

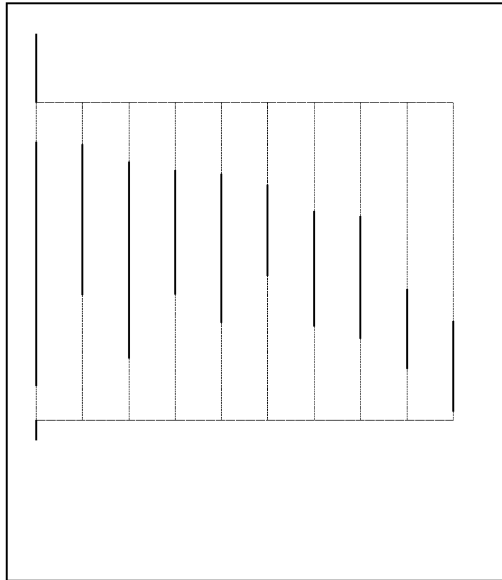

Figure 5: Vector operation, flattened for 10 elements (10 el./8 proc.)

The result is depicted in Figure 5, which uses the same scale as Figure 4. The improvement is clear and due to the much faster task creation and joining (and also to having only one synchronization structure for all tasks). Note, however, that the creation of the first task is slightly delayed due to the need for traversing the whole list before creating any tasks and for setting up the tasks themselves. This small delay is compensated by the faster task creation, but can eventually be a bottleneck for very large vectors. However it must be noted that the combined length of all the segments starting a recursion step in Figure 4 is less than the large segment corresponding to the head unification in Figure 5. This is because in the last case the overheads corresponding to the recursive calls are not present. Eventually, in a big computation with a large enough number of processors, the head unification will tend to dominate the whole computation (c.f. Amdahl's law).

In our quest for merging the techniques of the data-parallel and and-parallel approaches, one obvious solution would be to incorporate the techniques of the Reform Prolog engine into the PWAM abstract machine for the cases when it is applicable.[2] This may indeed be useful and is something we are currently collaboratively exploring. In fact, we believe that very little modification to the PWAM would be necessary. On the other hand, it is also interesting to study how far one can go with no modifications (or minimal modifications) to the machinery.

The last program studied is in fact a straightforward unfolding of the original recursion. Note that, interestingly, such unfoldings can always be performed at compile-time, provided that the depth of the recursion is known. In fact, knowing recursion bounds may actually be frequent in traditional data-parallel applications, (and is often the case when parallelizing bounded quantifications [3]). On the other hand it is not really the case in general and thus some other solution must be explored.

[2] In fact, a "map" builtin was indeed tried at some point in time [46] and showed substantial improvements for some benchmarks.

2.4 A More Dynamic Unfolding

The following program is an attempt at making the unfolding more dynamic, while still staying within the source-to-source program transformation approach:

```
vproc([H1,H2,H3,H4|T],[HR1,HR2,HR3,HR4|TR]) :-
        !,
        vproc(T,TR)  &
        process_element(H1,HR1) &
        process_element(H2,HR2) &
        process_element(H3,HR3) &
        process_element(H4,HR4).
vproc([H1,H2,H3|T],[HR1,HR2,HR3|TR]) :-
        !,
        vproc(T,TR)  &
        process_element(H1,HR1) &
        process_element(H2,HR2) &
        process_element(H3,HR3).
vproc([H1,H2|T],[HR1,HR2|TR]) :-
        !,
        vproc(T,TR)  &
        process_element(H1,HR1) &
        process_element(H2,HR2).
vproc([H|T],[HR|TR]) :-
        !,
        vproc(T,TR) &
        process_element(H,HR).
vproc([],[]).
```

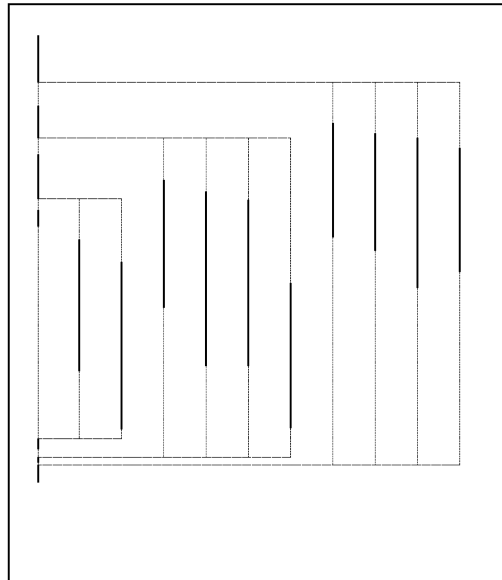

Figure 6: Vector operation, flattening (10 el./8 proc.)

The results are shown in Figure 6, which has the same scale as Figures 5 and 4. Two groups of four tasks are created one after the other, and the the two remaining tasks are created after a slight delay. The speed is not quite as good as when the 10 tasks are created at the same time, but the results are close.

This "flattening" approach, which has been used in &-Prolog compilation informally (see e.g. [46] and some of the standard &-Prolog benchmarks), has been studied formally Millroth [34], which has given sufficient conditions for performing these transformations for particular cases such as linear recursion.

There are still two problems with this approach, however. The first one is how to chose the "reformant level", i.e. the maximum degree of unfolding used, which with this technique is fixed at compile-time. In the previous example the unfolding was stopped at level 4, but could have gone on to a higher level. The ideal unfolding level depends both on the number of processors and the size of lists. For large lists a large unfolding may be desirable. However, the program size also grows, as well as the chain of intermediate unifications made by the last iterations. The other problem, which was pointed out before, is the fact that the initial matching of the list (or the conversion to a vector) is a sequential step which can become a bottleneck for large data sets. A solution is to increase the speed of creation of tasks, but that has a limit. In fact, it will also eventually become a bottleneck, even if low level instructions are used. Another solution is to use from the start, and instead of lists, more parallel data structures, such as vectors (we will return to this later).

2.5 Dynamic Unfolding In Parallel

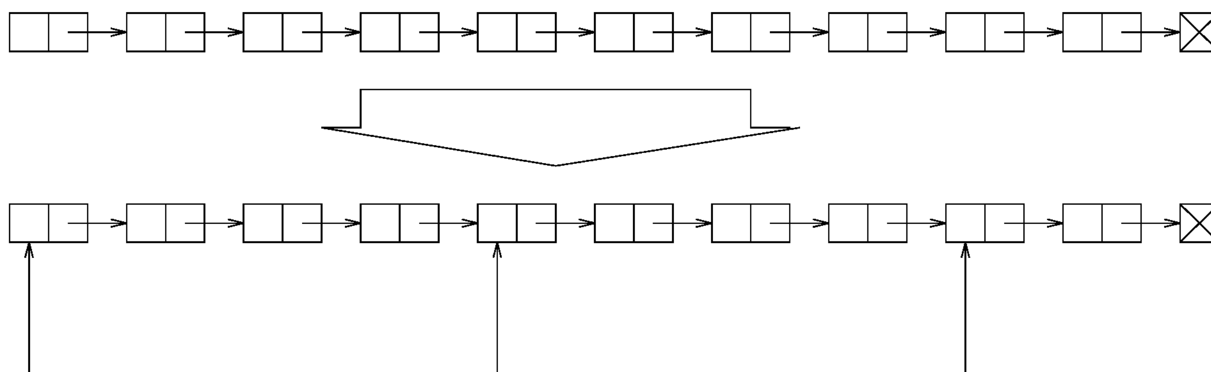

Figure 7: "Skip" operation, 10 elements in 4

We now propose a different solution which tries to address at the same time the two problems above. We give the solution for lists. The transformation has two objectives: speeding up the creation of tasks by performing it in parallel, and allowing a form of "flexible flattening". The basic idea is depicted in Figure 7. Instead of simply performing a unification of a fixed length as encoded at compile-time, a builtin, `skip/4`, is used which will allow performing unifications of different lengths.

The predicate `skip(L,N,LS,NS)` relates a list `L` and an "unfolding increment" `N` with a sublist `LS` of `L` which is placed at most at `N` positions from the starting of `L`. `NS` contains the actual number of elements in `LS`, in case that `N` is less than the length of `L` (in which case `LS = []`). The utility of `skip(L,N,LS,NS)` is that several calls to it using the output list `LS` as input list `L` in each call will return pointers to equally–spaced sublists of `L`, until no sufficient elements remain. Figure 7 depicts the pointers returned by `skip(L,N,LS,NS)` to a 10 elements list, with an "unfolding level" `N = 4`. This builtin is assumed for efficiency to be implemented at a low level, but it can be defined in Prolog as follows:

```
skip(L,N,LS,NS) :-
        skip(L,N,LS,NS,0).
```

```
skip(LS,0,LS,NS,NS) :- !.
skip([],_,[],NS,NS).
skip([_|Ls],N,LRs,Ns0,Ns) :-
        N1 is N-1,
        Ns1 is Ns+1,
        skip(Ls,N1,LRs,Ns0,Ns1).
```

We now return to our original program and make use of the proposed builtin (note that the "flattening parameter" N can be now chosen dynamically):

```
% Query: makevector(10,V), N=4, main(V,N,VR).

main(V,N,VR) :-
        start_event_trace,
        vproc_opt(V,VR,N),
        stop_event_trace,
        save_trace('Eventfile').

vproc_opt([],[],0).
vproc_opt(L,LR,N) :-
        skip(L,N,LS,NS),
        skip(LR,N,LRS,NS),
        ( vproc_opt(LS,LRS,NS) & vproc_opt_n(L,LR,NS) ).


vproc_opt_n(_,_,0).
vproc_opt_n([L|Ls],[LR|LRs],N) :-
        ( N1 is N-1, vproc_opt_n(Ls,LRs,N1) ) &
        process_element(L,LR).
```

The result is shown in Figure 8. The large delays are due to the fact that `skip/4` is defined in Prolog in this experiment, but, as mentioned before, it could be made much faster as a builtin. Note, however, how the tasks are created in groups of four corresponding to the dynamically selected increment, which can now be made arbitrarily large. We believe that this idea would also be useful even at a lower level.

It is worth noting that, in this case, the predicate `skip/4` not only returns pointers to sublists of a given list, but also is able to construct a new list composed with free variables. This allows spawning independent parallel processes, each one of them working in separate segments of a list. This, in some sense, mimics the so–called *poslist* and *neglist* identified in the Reform Compilation at run–time. Though this solution gives, obviously, poorer performance than a compile–time approach, we feel that a low–level implementation could give good results.

Note also that other builtins similar to skip could be proposed for other types of data structures and for each type of traversal allowed by each of those data structures.

As an example, we may want the splitting of the list to be used afterwards (for example, because it is needed in some further similar processing). We can use the `skip/4` predicate to build a `skiplist/3` predicate as follows:

```
skiplist([], _N, []):- !.
skiplist(L, N, [L|LSs]):-
        skip(L, N, LS, _M),
        skiplist(LS, N, LSs).
```

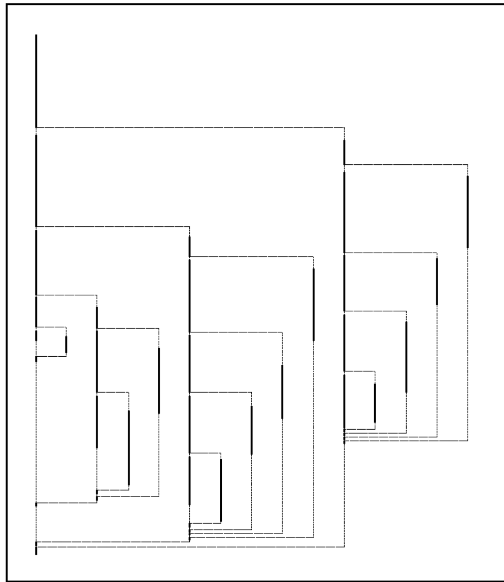

Figure 8: Vector operation, flexible flattening (10 el./8 proc.)

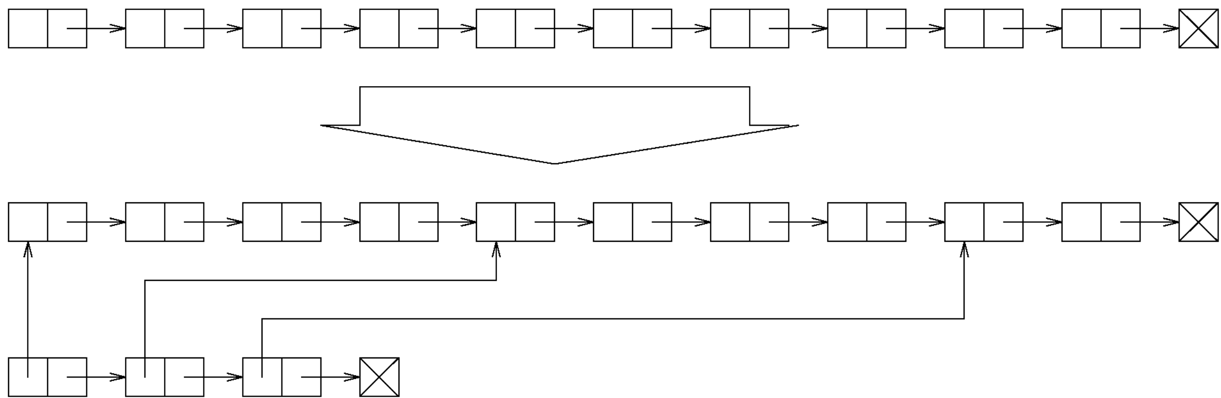

Figure 9: “Skiplist” operation, 10 elements in 4

A typical call to skiplist/3 would be done with the two first arguments instantiated; the third argument would return pointers to subslists of the first argument or, under a more logical point of view, the third argument describes a set of sublists of the first argument by means of difference lists. Figure 9 depicts this situation.

3 Constant Time Access Arrays in Prolog?

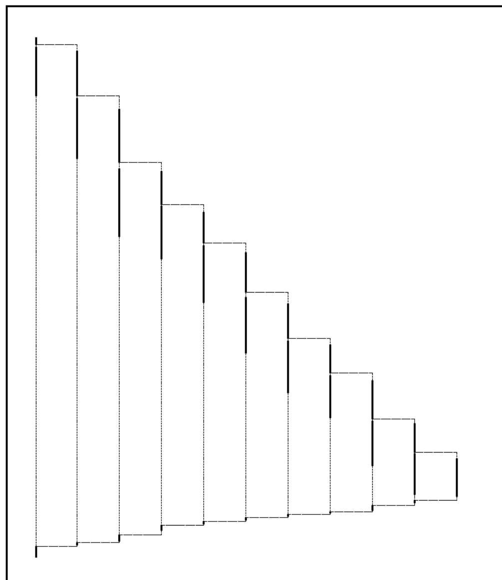

Figure 10: Vector operation, constant access arrays (10 el./8 proc.)

Finally, and for the sake of argument, we propose a simple-minded approach to the original problem using the real "arrays" in standard Prolog, i.e. terms. Of course the use of this technique is limited by the fact that *term arity is limited in many Prolog implementations*, but this could be cured. In the query we create a vector of length N using functor/3, initialize it, and then pass it on to a "vector" version of vproc (we could, of course, also start with a list, as in previous examples, and convert it into a vector before calling the "vector" version of vproc):

```
%Query: N=10, functor(V,a,N), functor(VR,a,N), fillvector(N,V), main(V,VR,N).

main(V,VR,N) :-
        start_event_trace,
        vproc(V,VR,N),
        stop_event_trace,
        save_trace('Eventfile').
```

Where the parallelized "vector" version of vproc would be as follows:

```
vproc(_,_,0).
vproc(V,VR,I) :-
        ( I>0,
          process_element(V,VR,I)
        )
```

```
        &
        ( I1 is I-1,
          vproc(V,VR,I1)
        ).
```

Element access is done in constant time using `arg/3`:

```
process_element(V,VR,I) :-
        arg(I,V,H),
        HR is ((((H * 2) / 5)^2)+(((H * 6) / 2)^3))/2,
        arg(I,VR,HR).
```

The results are presented in Figure 10. In this example we are using a simple minded loop which creates tasks recursively, but the same techniques illustrated in previous examples could be applied to this "real array" version: it is easy now to modify the above program as in the previous examples in order create the tasks in groups of N, but now without having to previously traverse the data structure, as was the case when using the `skip` builtin!

Finally, following on on this idea, we illustrate how one could even build a quite general purpose "FORTRAN-like" constant access array library without ever departing from standard Prolog or, eliminating the use of "`setarg`", even from "clean" Prolog. It is not that we are supporting the use of these data structures, but rather we are simply trying to make the point that if one really, really, wants them, then the arrays are there. The solution we propose is related to the standard "logarithmic access time" extensible array library written by D.H.D.Warren. In this case, we obtain constant (rather than logarithmic) access time, with the drawback that arrays are, at least in principle, fixed size.

We begin by defining the "type" array. Essentially, an array is a term of arity two which contains as its first argument a list of integers which correspond to the dimensions of the array (thus we can have arrays of arbitrary dimensions) and as its second argument a term whose arity is the total number of cells in the array (and thus represents the total amount of storage needed by the array):

```
% Type definition
is_array(matrix(D,S)) :-
        functor(S,storage,L),
        multiply_list(D,L).

multiply_list([],1).
multiply_list([I|Is],N) :-
        multiply_list(Is,N1),
        N is N1 * I.
```

Arrays can be created, in full FORTRAN tradition, by performing a call to `dimension/2`, where the first argument is a list with the dimensions of the array and the second argument returns the array:

```
dimension(D,matrix(D,S)) :-
        multiply_list(D,Nelements),
        functor(S,storage,Nelements).
```

Note, however, that with judicious use of delays one can also create arrays a call to the type definition predicate.

All elements of the "storage" part are accessible in constant time (as arguments of a structure):

```
% Element access
access(matrix(D,S),I,X) :-
        compute_offset(I,D,Offset),
        arg(Offset,S,X).
```

```
compute_offset([I],[D],I) :-
        I>0,
        I=<D,
        !.
compute_offset([I|Is],[D|Ds],Offset) :-
        I>0,
        I=<D,
        !,
        compute_offset(Is,Ds,Offset1),
        I1 is I-1,
        Offset is D * I1 + Offset1.
compute_offset(_,_,_) :-
        format("Warning: access out of bounds in array.",[]).
```

Finally, if one really, really wants to have everything one has in FORTRAN, then even destructive assignment is available:

```
setel(matrix(D,S),I,X) :-
        compute_offset(I,D,Offset),
        setarg(Offset,S,X).
```

However, one would hope that compilation technology would make the need for resorting to these extremes unnecessary.

Note that the definitions should at least be changed to compute with an accumulating parameter, but they have been left as is for clarity. Also, use of delay can make them fully reversible. Realistically, all these operations should be builtins for performance reasons. Note that calls to dimension, access, set, etc. could in any case often be very efficiently compiled in-line to a specialized call to `functor`, `arg`, etc.

References

[1] K.A.M. Ali. Or-parallel Execution of Prolog on the BC-Machine. In *Fifth International Conference and Symposium on Logic Programming*, pages 253–268, Seattle,Washington, 1988. MIT Press.

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

[3] Henrik Arro, Jonas Barklund, and Johan Bevemyr. Parallel bounded quantification—preliminary results. *ACM SIGPLAN Notices*, 28:117–124, 1993.

[4] Jonas Barklund. *Parallel Unification.* PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1990.

[5] Jonas Barklund and Håkan Millroth. Nova Prolog. UPMAIL Tech. Rep. 52, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1988.

[6] J. Bevemyr, T. Lindgren, and H. Millroth. Exploiting recursion-parallelism in Prolog. In *Proc. PARLE'93*, Berlin, 1993. Springer-Verlag.

[7] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: the language and its implementation. In *Proc. 10th Intl. Conf. Logic Programming*, Cambridge, Mass., 1993. MIT Press.