Published in *Proceedings of EURO–PAR'95*, Sweden

Relating Data–Parallelism and (And–) Parallelism in Logic Programs*

Manuel V. Hermenegildo and Manuel Carro

Universidad Politécnica de Madrid
Facultad de Informática
28660 Boadilla del Monte
Madrid — Spain
{herme,mcarro}@fi.upm.es

Abstract. Much work has been done in the areas of and–parallelism and data–parallelism in Logic Programs. Both types of parallelism offer advantages and disadvantages: traditional (and–) parallel models offer generality, whereas data–parallelism techniques offer increased performance for a restricted class of programs. The thesis of this paper is that these two forms of parallelism are not fundamentally different and that relating them opens the possibility of obtaining the advantages of both within the same system. Some relevant issues are discussed and solutions proposed. The discussion is illustrated through visualizations of actual parallel executions implementing the ideas proposed.

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* is an iteration where the number of repetitions is known before the iteration is initiated.

Data–parallelism has been exploited in many languages, including C* [28], Data Parallel C [12], *LISP [27], 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 [29, 3, 4].

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 [6, 10, 18, 19, 1, 17, 26]. 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 paper, the other being to explore the intimately related issue of fast task startup.

* The authors have been partially supported by ESPRIT project 6707 *ParForce*

1.1 Data–Parallelism and And–Parallelism

It is generally accepted that data–parallelism is a restricted form of and–parallelism:[1] 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. All and–parallel systems impose certain restrictions on the goals or threads which can be executed in parallel (such as independence and/or determinacy, applied at different granularity levels [15, 21, 7, 16]) which 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 have to meet the same restrictions from this point of view. This is generally referred to as the "safeness" conditions in the context of data–parallelism.

However, one central idea in data–parallelism is to impose *additional* restrictions to the parallelism allowed, in order to make possible further optimizations in some important cases. These restrictions limit the amount of parallelism which can be obtained with respect to a more general purpose and–parallel implementation. But, 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. [4]). Often, a *a priori* knowledge of the sizes of the data structures being operated on is required (but this data is also obtained dynamically in other cases).

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 obtained in as many programs as possible, and at the same time have the implementation be able to take advantage of the optimizations possible in data–parallel computations 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; this problem is common to both control– and data–parallel systems, and equally difficult in both. The solution of allowing the programmer to explicitly declare such properties or use special constructs (such as "parallel map," "bounded quantifications" [2], etc.) which have built–in syntactic restrictions may help, but it is also true that this solution 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 realized in the runtime machinery to exploit the special cases. Admitting 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"

[1] Note, however, that data–parallelism can also be exploited as or parallelism [24].

version of the more general machine. 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 related to, for example, optimizations in memory management.

However, it should be noted that if the particular case is identified, the same optimizations can also be done in general–purpose abstract machines supporting and–parallelism (such as, for example, the RAP–WAM [14], the DAS–WAM [26] or the Andorra–I engine [25]), and without losing the capability of handling the general case, as shown in [23, 22].

On the other hand, a number of optimizations, generally related to the "Reform Compilation" done in Reform Prolog [20], are more fundamental. We find these optimizations particularly interesting because they bring attention upon a very important issue regarding the performance of and–parallel systems: that of the speed in the creation and joining of tasks. We will essentially devote the rest of the paper to this issue, because of the special interest of this subject, and given that, as pointed out before, the other intervening issues have already been addressed to some extent in the literature.[2]

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 result a ground list. We use as `process_element/2` a 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.
```

2.1 The Naive Approach

This program can be naively parallelized as follows using "control–parallelism" (we will use throughout &–Prolog [14] syntax, where the "&" operator represents a potentially parallel conjunction):

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

[2] Improving the performance of parallel systems in the presence of fine–grained computations can also be addressed by performing "granularity control" [9, 11, 31]. This issue can be treated orthogonally to the techniques discussed in this paper.

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 how the tasks are started due to the textual ordering of the goals. In a system like &–Prolog, using one of the the standard schedulers (which we will assume throughout the examples), the 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. However, the approach, or, at least, the naive program presented, also has some drawbacks.

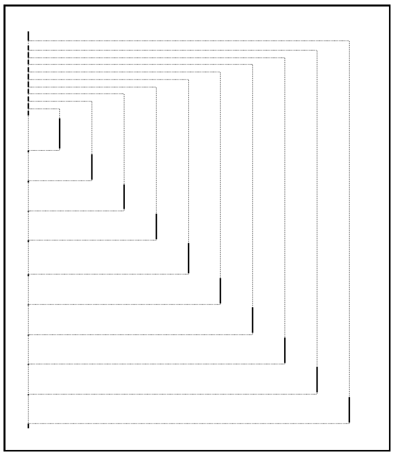

Fig. 1. List operation (10 el./1 proc.)

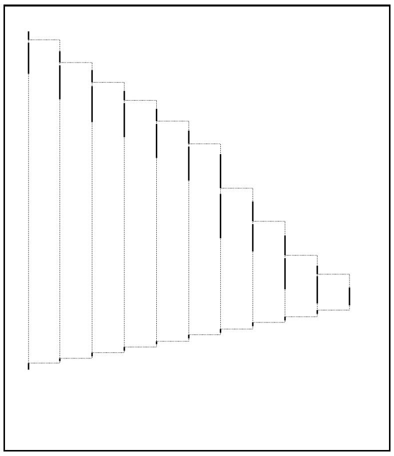

Fig. 2. List operation, giving away recursion (10 el./8 proc.)

In order to illustrate this, we perform the experiment of running the previous program with the goal "`vproc(V, R)`", where `V` has been instantiated to the list of integers from 1 to 10, and `R` is a free variable. This program (as well as the others) is run in &–Prolog on a Sequent Symmetry shared memory multiprocessor and instructed to generate trace files, which can be visualized using VisAndOr [5]. In VisAndOr graphs, time goes from top to bottom. Vertical solid lines denote actual execution, whereas vertical dashed lines represent waits due to scheduling or dependencies, and horizontal dashed lines represent forks and joins.

Figure 1 represents the execution of the benchmark in one processor, and serves as scale reference. The result of running the benchmark in 8 processors is depicted in Figure 2. 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 its length to Figure 1. However, the speedup obtained is 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.

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, to keep the recursive goal local, and not even push it on to the goal stack:

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

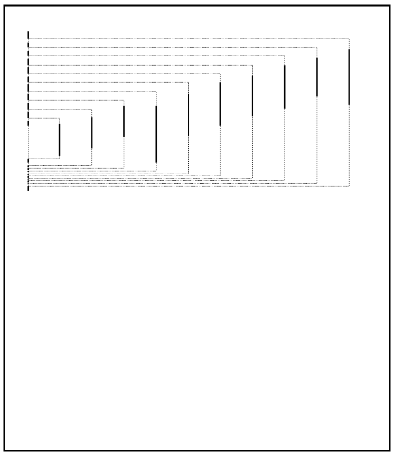

Fig. 3. List operation, keeping recursion (10 el./8 proc.)

The result of running this program is depicted in Figure 3, which uses the same scale as Figures 1 and 2. The first process keeps the recursion local and thus creates the tasks much faster, resulting in substantially more speedup. This transformation is normally done by the &–Prolog parallelizing compiler, unless the user explicitly annotates the goal for parallel execution by hand.

In applications which 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) the speed of the process creating the tasks will become a bottleneck, and speeding up tasks creation can greatly impact the performance of systems where the number of processes/threads cannot be statically determined and their creation is triggered at runtime.

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. Assuming that the recursion has already been identified as suitable for this technique, this system converts the list into a vector, noting the length on the way, and then creates the tasks associated with each element in a tight, low level loop. 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:

```
vproc([H1,H2,H3,H4,H5,H6,H7,H8,H9,H10], [R1,R2,R3,R4,R5,R6,R7,R8,R9,R10]):-
    process_element(H1,R1) &
    process_element(H2,R2) &
    .
    .
    .
    process_element(H10,R10).
```

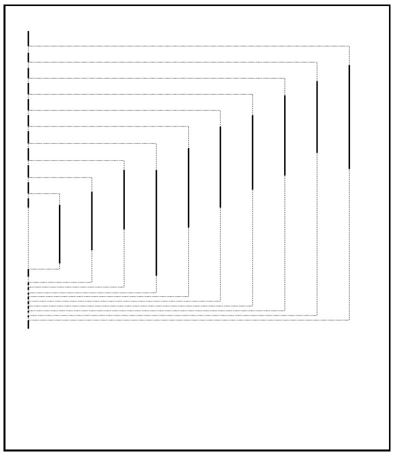

Fig. 4. List operation, keeping recursion (10 el./8 proc.)

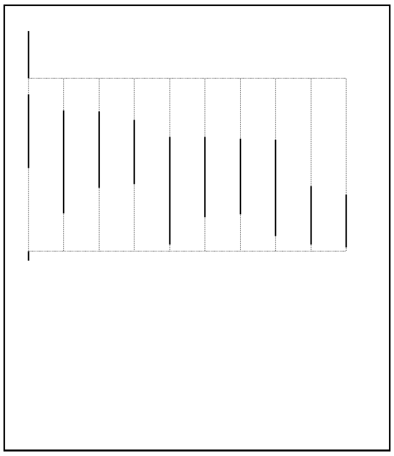

Fig. 5. List operation, flattened for 10 elements (10 el./8 proc.)

Figure 4 represents the same execution as Figure 3, but at a slightly enlarged scale; this scale will be retained throughout the rest of the paper, to allow easy comparisons among different pictures.

The execution of the "data–parallel" program is depicted in Figure 5, which uses the same scale as Figure 4. The clear improvement is 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 unifying 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. 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 this case, unification parallelism can be worthwhile [3].

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. 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 an unfolding of the original recursion. Note that 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 [2]). On the other hand it is not really the case in general and thus some other solution must be explored.

2.4 A More Dynamic Unfolding

If the depth of the recursion is not known at compile time the previous scheme cannot be used. But instead of resorting directly to the naive approach, we can try to perform a more flexible task startup. The program below 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],
      [R1,R2,R3,R4|TR]):- !,
    vproc(T,TR)  &
    process_element(H1,R1) &
    process_element(H2,R2) &
    process_element(H3,R3) &
    process_element(H4,R4).
vproc([H1,H2,H3|T],[R1,R2,R3|TR]):- !,
    vproc(T,TR)  &
    process_element(H1,R1) &
    process_element(H2,R2) &
    process_element(H3,R3).
vproc([H1,H2|T],[R1,R2|TR]):- !,
    vproc(T,TR)  &
    process_element(H1,R1) &
    process_element(H2,R2).
vproc([H|T],[R|TR]):- !,
    vproc(T,TR) &
    process_element(H,R).
vproc([],[]).
```

The results are shown in Figure 6, which has the same scale as Figures 4 and 5. A group of four tasks is created; one of these tasks creates, in turn, another group of four. The two remaining tasks are created inside the latter group. 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 has been studied formally by Millroth[3], 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 the ideal unfolding level depends both on the number of processors and the size of lists. 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 task creation speed (for example, using low level instructions) but this has a limit, and it will also eventually become a bottleneck. Another solution is to use from the start, and instead of

[3] And has been used in &–Prolog compilation informally (see e.g. [30] and some of the standard &–Prolog benchmarks, in `http://www.clip.dia.fi.upm.es`).

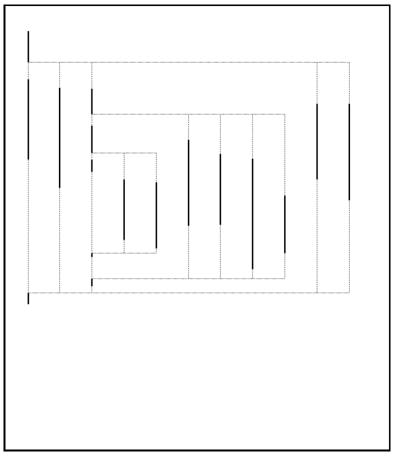

Fig. 6. List operation with fixed list flattening (10 el./8 proc.)

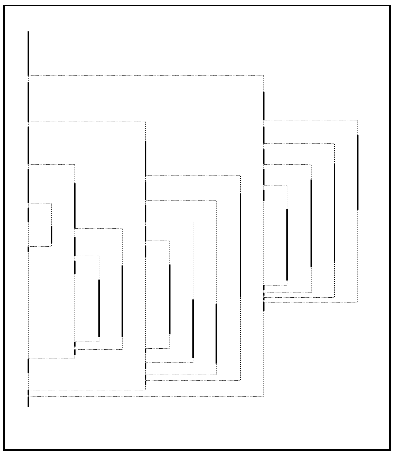

Fig. 7. List operation with flexible list flattening (10 el./8 proc.)

lists, more parallel data structures, such as vectors (we will return to this in Section 3).

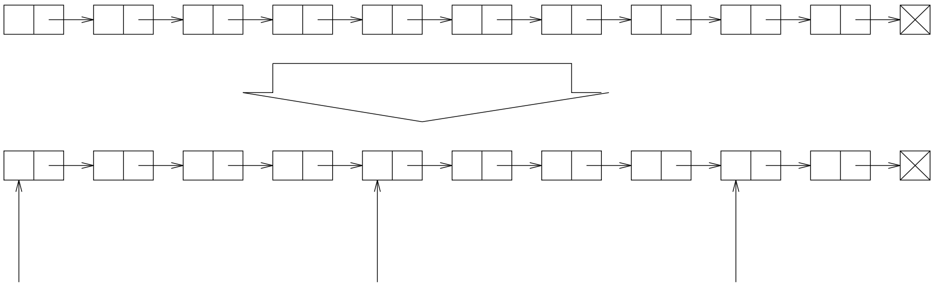

Fig. 8. "Skip" operation, 10 elements in 4

2.5 Dynamic Unfolding in Parallel

We now propose a different solution which tries to address at the same time the two problems above. 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, applied to lists, is depicted in Figure 8. Instead of 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 suffix `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 = []`). Several calls to `skip(L,N,LS,NS)` using the output list `LS` as input list `L` in each call will return pointers to equally–spaced suffixes of L, until no sufficient elements remain. Figure 8 depicts the pointers returned by skip(L,N,LS,NS) to a 10 elements list, with an "unfolding level" N = 4. This builtin can be defined in Prolog as follows (but can, of course, be implemented more efficiently at a low level):

```
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). The entry point is vproc_opt/3:

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

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

We have included the skip/4 predicate as a C builtin in the &–Prolog system and run the above program. The result is shown in Figure 7. The relatively large delays are due to the traversal of the list made by skip/4. Note, however, how the tasks are created in groups of four corresponding to the dynamically selected increment, which can now be made arbitrarily large.

It is worth noting that, in this case, the predicate skip/4 not only returns pointers to sublists of a given list, but is also able to construct a new list filled with free variables. This allows spawning 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.

If we want the splitting of the list to be used afterwards (for example, because it is needed in some further similar processing), we can construct a list containing pointers to suffixes of a given list, or, under a more logical point of view, a list describing sublists of the initial list by means of difference lists. Figure 9 depicts this situation, and Figure 10 shows the result of an execution where the input and output lists have been pre-processed using this technique. This list preprocessing does not appear in Figure 10, as an example of the reuse of a previously traversed list.

2.6 Performance Evaluation

In order to assess the relative performance of the various techniques discussed, we have run the examples on a larger (240 element) list. The execution times are presented in Table 1. The column *Relative Speedup* refers to the speedup with respect to the parallel execution on one processor, and the column *Absolute*

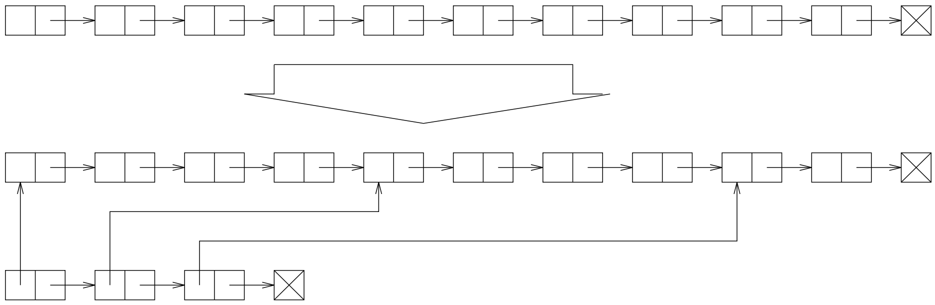

Fig. 9. "Skiplist" operation, 10 elements in 4

Method	Time (ms)	Relative Speedup	Absolute Speedup
Sequential	127	—	1
Parallel, 1 processor	153	1	0.83
Giving away recursion	134	1.14	0.94
Keeping recursion	41	3.73	3.09
Skipping (8)	30	5.1	4.23
Skipping (30)	28.5	5.36	4.45
Pre–built skipping list (8)	28	5.4	4.53
Pre–built skipping list (30)	26.5	5.77	4.79
Reform Compilation (8)	27	5.6	4.7
Data Parallel	26	5.88	4.88

Table 1. Times and speedups for different list access, 8 processors.

Speedup measures the execution speed with respect to the sequential execution. The numbers between parentheses to the right of some benchmark names represent the skipping factor chosen.

The speedups suggested by Figures 4 to 9 may not correspond with those in the table — the length of the benchmark and the skip/unfolding increment chosen in the two cases is different, and so is the distribution of the tasks. Processing larger lists can take more advantage from the proposed techniques, because the relative overhead from traversing the list is comparatively less, and tasks with larger granularity can be distributed among the processes.

Overheads associated with scheduling, preparing tasks for parallel execution, etc. make the parallel execution in one processor be slower than the sequential execution. This difference is more acute in very small grained benchmarks, as the one we are dealing with.

It can also be noted how a pre–built skipping list with a properly chosen increment beats the reformed program. Of course a reformed program with the same unfolding level would, in principle, at least equal the program with the pre–built list. But the point is that the reformed program was statically transformed, whereas the skiplist version can change dynamically, and be useful in cases where the same data is used several times in the same program.

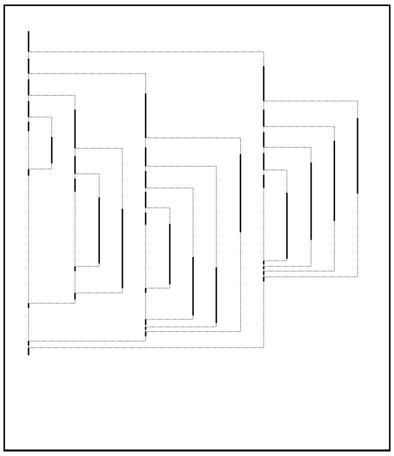

Fig. 10. List operation with prebuilt list (10 el./8 proc.)

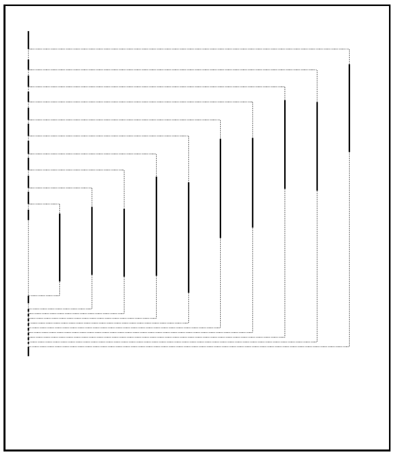

Fig. 11. Vector operation, constant time access arrays (10 el./8 proc.)

3 Using Constant Time Access Arrays

Finally, and for the sake of argument, we propose a simple-minded approach to the original problem using standard Prolog terms, i.e., the real "arrays" in Prolog. The use of this technique is limited by the fact that term arity is limited in many Prolog implementations, but this could be easily cured. The "vector" version of `vproc/3` receives a vector represented using a structure (which can have been either created directly or from a list) and its length. The access to each element is done in constant time using `arg/3`.

```
vproc(0,_,_).
vproc(_,V,VR):-
    I>0, I1 is I-1,
    vproc(I1,V,VR) & process_element_vec(I,V,VR).
```

The execution of this program is presented in Figure 11, where we are using a simple minded loop which creates tasks recursively. The same techniques illustrated in previous examples can be applied to this "real array" version: it is easy now to modify the above program in order to 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.

The result appears in Figure 12. It may seem that there is no performance improvement, but is due to the fact that the execution depicted is very small, and the added overhead of calculating the "splitting point" becomes a sizeable part of the whole execution. In Table 2 larger arrays and skipping factors were chosen, achieving better speedups than the simple parallel scheme. Since no real traversal is needed using this representation, the amount of items skipped can be dynamically adjusted with no extra cost.

A more even load distribution than that obtained with the simple recursion scheme can be achieved using a binary split. This is equivalent to dynamically

Method	Time (ms)	Relative Speedup	Absolute Speedup
Sequential	149	—	1
Parallel, 1 processor	174	1	0.85
Keeping recursion	45	3.8	3.31
Binary startup	38	4.5	3.92
Skipping (8)	31.2	5.57	4.77
Skipping (30)	29.5	5.89	5.05

Table 2. Times and speedups for vector accesses

choosing the splitting step to be half the length of the sub–vector assigned to the task. Figure 13 depicts an execution following this scheme. As in Figure 12, the comparatively large overhead associated with the determination of the splitting point makes this execution appear larger than that corresponding to the simple recursive case. But again, Table 2 reflects that for large enough executions, its performance can be placed between the simple recursion scheme and a carefully chosen skipping scheme.

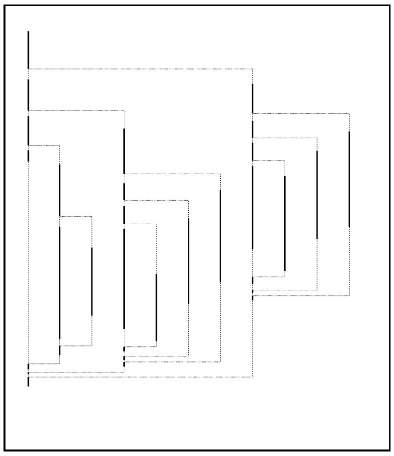

Fig. 12. Vector operation, constant time access arrays, skipping, 10 el./8 proc.

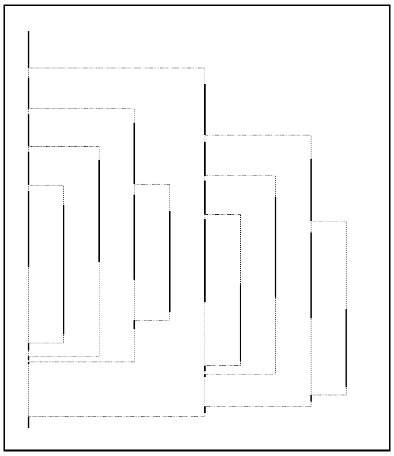

Fig. 13. Constant time access arrays, binary startup, 10 el./8 proc.

Some conclusions can be drawn from Tables 1 and 2. First, the structure–based programs are slightly slower than their list–based counterparts. This is understandable in that using structures as arrays involves an explicit index handling that is less efficient (or, rather, that has been less optimized) than in the case of lists. But the fact that accessing any element in a structure is, in principle, a constant–time operation, allows a comparatively efficient implementation of the dynamic *skip* strategy. This is apparent in that the speedups attained with the arrays version of the skipping technique are better than those corresponding to the list–based programs. The absolute speed is less, which can be attributed to the fact that the version of &–Prolog used has the `arg/3` builtin written in

C, with the associated overhead of calling and returning from a C function. This could be improved making `arg/3` (or a similar primitive) a faster, WAM–level instruction. Again, if we want (or have to) use lists, a low–level `vectorize/2` builtin could be fast enough to translate a list into a structure and still save time with respect to a list–based implementation processing the resulting structure in a divide–and–conquer fashion.

Finally, following on on this idea, we would like to point out that it is possible to 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. The solution we propose in [13] 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.

4 Conclusions

We have argued that data–parallelism and and–parallelism are not fundamentally different and that by relating them in fact the advantages of both can be obtained within the same system. We have also argued that the difference lies in two main issues: memory management and fast task startup and management. Having pointed to recent progress in memory management techniques in and–parallelism we have concentrated on the issue of fast task startup, discussed the relevant issues and proposed a number of solutions, illustrating the point made through visualizations of actual parallel executions implementing the ideas proposed. In summary, we argue that both approaches can be easily reconciled, resulting in more powerful systems which can bring the performance benefits of data–parallelism with the generality of traditional and–parallel systems.

Our work has concentrated on speeding up task creation and distribution in a type of symbolic or numerical computations that are traditionally characterized by structural recursion over lists or arrays. We have shown some transformation techniques relying on a dynamic load distribution that can improve the speedups obtained in a parallel execution. However, the overhead associated with this dynamic distribution is large in the case of lists; better speedups can be obtained using data structures with constant access time, in which arbitrarily splitting the data does not impose any additional overhead.

There are other kinds of computations where the iteration is performed over a numerical parameter. While not directly characterizable as "data–parallelism" this type of iteration can also benefit from fast task startup techniques. This very interesting issue has been recently and independently discussed by Debray [8], and shown to also achieve significant speedups for that class of problems.

The techniques discussed in this paper probably cannot always match the performance of a native data–parallel system. But it is also true that, in principle, low level mechanisms can be designed which fit seamlessly within the machinery of a more general parallel system. In particular, entries in goal stacks *à la* &–Prolog can be modified to include pointers to the data the goals have to work with, thus tightly encoding both the thread to be executed and the relevant data. The resulting system should be able to achieve both maximum speedup for data–parallel cases while at the same time supports general and–parallelism.

Acknowledgments

We would like to thank Jonas Barklund, Johan Bevemyr, and Håkan Millroth for discussions regarding Reform Prolog and Bounded Quantifications, as well as Saumya Debray, Enrico Pontelli, and Gopal Gupta for interesting discussions regarding this work.

References

1. 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.
2. Henrik Arro, Jonas Barklund, and Johan Bevemyr. Parallel bounded quantification—preliminary results. *ACM SIGPLAN Notices*, 28:117–124, 1993.
3. Jonas Barklund. *Parallel Unification*. PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1990.
4. 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.
5. M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.
6. J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.
7. M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.
8. S. Debray and M. Jain. A Simple Program Transformation for Parallelism. In *1994 International Symposium on Logic Programming*, pages 305–319. MIT Press, November 1994.
9. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
10. D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.
11. P. López García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific Publishing Company, September 1994.
12. Philip J. Hatcher and Michael J. Quinn. *Data-parallel Programming on MIMD Computers*. MIT Press, Cambridge, Mass., 1991.
13. M. Hermenegildo and M. Carro. A Note on Data–Parallelism and (And–Parallel) Prolog. Technical report CLIP 6/94.0, School of Computer Science, Technical University of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, June 1995.
14. M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
15. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

16. M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 123–133. Springer-Verlag, May 1994.
17. L. Kale. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Fourth International Conference on Logic Programming*, pages 616–632. Melbourne, Australia, May 1987.
18. Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. MIT Press, August 1988.
19. E. Lusk et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.
20. Håkan Millroth. Reforming compilation of logic programs. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 485–502, San Diego, USA, 1991. The MIT Press.
21. L. Naish. Parallelizing NU-Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. University of Washington, MIT Press, August 1988.
22. E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.
23. E. Pontelli, G. Gupta, D. Tang, M. Hermenegildo, and M. Carro. Efficient Implementation of And–parallel Prolog Systems. Technical Report CLIP4/95.0, T.U. of Madrid (UPM), June 1995.
24. S. Prestwich. On parallelisation strategies for logic programs. In Springer-Verlag, editor, *Proceedings of the International Conference on Parallel Processing*, number 854 in Lecture Notes in Computer Science, pages 289–300, 1994.
25. V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 825–839. MIT Press, June 1991.
26. K. Shen. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In *Proc. Joint Int'l. Conf. and Symp. on Logic Prog.* MIT Press, 1992.
27. Thinking Machines Corp., Cambridge, Mass. *The Essential *LISP Manual*, 1986.
28. Thinking Machines Corp., Cambridge, Mass. *C* Programming Guide*, 1990.
29. Andrei Voronkov. Logic programming with bounded quantifiers. In Andrei Voronkov, editor, *Logic Programming—Proc. Second Russian Conf. on Logic Programming*, LNCS 592, Berlin, 1992. Springer-Verlag.
30. R. Warren and M. Hermenegildo. Experimenting with Prolog: An Overview. Technical Report 43, MCC, March 1987.
31. X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 809–816. Institute for New Generation Computer Technology (ICOT), June 1992.

This article was processed using the LaTeX macro package with LLNCS style