To be published in the *Computer Languages Journal*

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

Manuel V. Hermenegildo
Manuel Carro

Universidad Politécnica de Madrid (UPM)
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. Such work has proceeded to a certain extent in an independent fashion. Both types of parallelism offer advantages and disadvantages. Traditional (and–) parallel models offer generality, being able to exploit parallelism in a large class of programs (including that exploited by data parallelism techniques). Data parallelism techniques on the other hand 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.

Keywords: Parallel Logic Programming, And-Parallelism, Data-Parallelism, Fast Task Startup, Scheduling.

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 [MR90], C* [Thi90], Data Parallel C [HQ91], *LISP [Thi86], 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 [Vor92, NT88, BM88, Bla92, Kac90, Wis86, Mil90, Bar90, BLM93a, BLM93b].

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 [Con83, DeG84, DeG87, Her86a, HG90, KK84, LK88, War87, Lus90, Ali88, AK90, GJ89, GSCYH91, GHPC94, Fag87, Kal87, She92, War88, SCWY90, Ric89, Kar92, Car90]. 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.

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 steps of a recursion or the iterations a loop. This type of parallelism is obviously also supported in and–parallel systems: each thread in the data parallel approach would correspond to the parallel execution of different recursion steps / loop iterations of the same body.

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 [HR95, Nai88, SCWY90, GHM93, HtCg94]) 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, since they are after all and–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. Such conditions are imposed among the iterations being parallelized (examples are requiring them to be deterministic, to have only one alternative, and/or to be independent).

However, one of the central ideas in data–parallelism, as presented in many proposals, is to impose *additional* restrictions to the degree of 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. [BLM93a, BLM93b]). 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 data structures (generally lists or arrays) 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 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 be studied. The first class is related to detecting when the particular properties to be used to perform the optimizations hold. However, this problem is

[1]Note, however, that data parallelism can also be exploited as or parallelism [Pre94, CDO88].

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. The solution of allowing the programmer to explicitly declare such properties or use special constructs (such as "parallel map," "bounded quantifications" [ABB93], 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 abstract machine to exploit the special cases. 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 [BLM93a, BLM93b]. 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 [Her86b, HG90], the DASWAM [She92], or the Andorra-I engine [SCWY91]. 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 optimizations in memory management which are possible with respect to general purpose abstract machines due 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, which delimit stack sections corresponding to different goals and to different backtracking points, are not necessary. However, it should be noted that the same optimizations can also be done in general–purpose abstract machines supporting and–parallelism, such as the RAP–WAM, if the particular case is identified, and without losing the general case [Her86a, PGH95, SH94, PGT95, PGT+96]. Both dynamic and static detection of such special cases has been studied. A similar argument can be made regarding some other minor optimizations that, for lack of space, will not be addressed explicitly.

On the other hand, a number of optimizations, generally related to the "Reform Compilation" done in Reform Prolog [Mil91], 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]Improving the performance of and–parallel systems in the presence of fine-grained computations can also be addressed by performing "granularity control", where goals that could have been run in parallel but are too small-grained are executing them sequentially. This is usually done by determining (statically or dynamically) the cost of goals and sequentializing them or grouping them when such cost falls below a given threshold. This very interesting issue can be treated orthogonally to the techniques that we discuss in this paper. Relevant work can be found in [DLH90, KS90, LHD94, ZTD+92] and their references.

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 the paper &–Prolog [HG91] syntax, where the “&” operator instead of 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, due to the textual ordering of the goals. In a system like &–Prolog, using one of the standard schedulers (we will assume this scheduler throughout the examples), the initial agent, running the call to `vproc/2`, would create a task[3] 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 task, creating in turn another task for the recursion and taking on a new iteration of `process_element(H,HR)`, and so on. In the end, parallel tasks are created for each iteration. Note that all task 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 previously parallelized 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 small–grained numerical operation, which serves to illustrate the issue:

[3]The notion of task does not correspond with that of the underlying operating system here. &-Prolog tasks are internal executions of goals. Each &-Prolog agent can pick up goals made available for parallel execution and execute them. When the goal is finished, the agent is free to search for more work. Other implementations might choose to actually use a dedicated (perhaps lightweight) O.S. process for each goal.

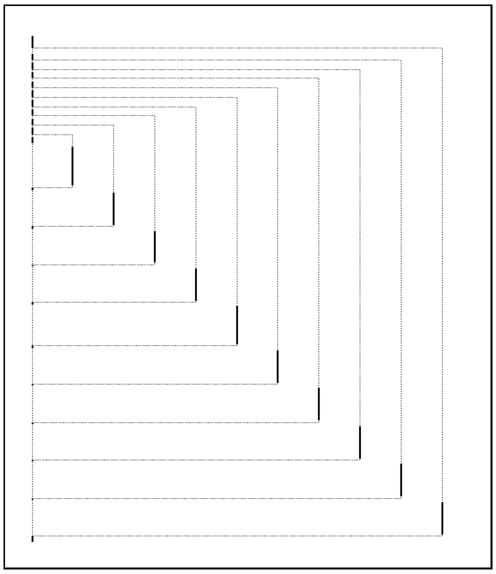

Figure 1: Vector operation (10 el./1 proc.)

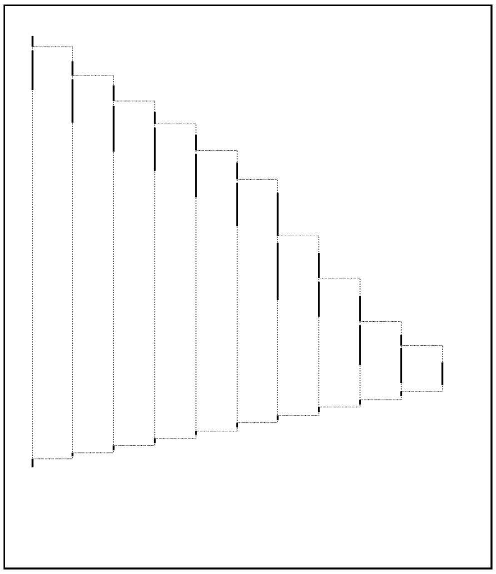

Figure 2: Vector operation, giving away recursion (10 el./8 proc.)

```
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 for this execution. This trace file contains a description of the execution, including the starting and ending time of every task, as well as the dependencies among tasks.

The trace is then visualized with VisAndOr [CGH93]. 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. Each solid vertical segment represents a task corresponding to one invocation of the `process_element/2` sequential goal. These are executed consecutively in time, after having been made available for parallel execution.

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 to Figure 1 — 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 [CGH93].

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, so that the recursive goal is kept local, and not even pushed on to the goal stack:

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

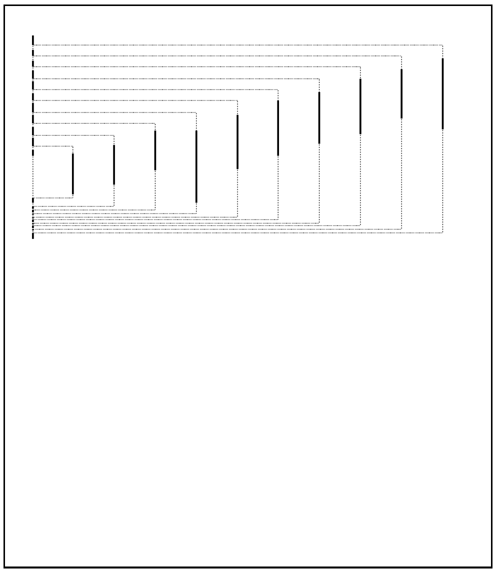

Figure 3: Vector operation, keeping recursion (10 el./8 proc.)

The result of running this program is depicted in Figure 3, which uses the same scale as Figures 1 and 2. The first process can now be observed to keep the recursion local and thus create the tasks much faster, resulting in substantially more speedup. It should be noted 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.

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):

```
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).
```

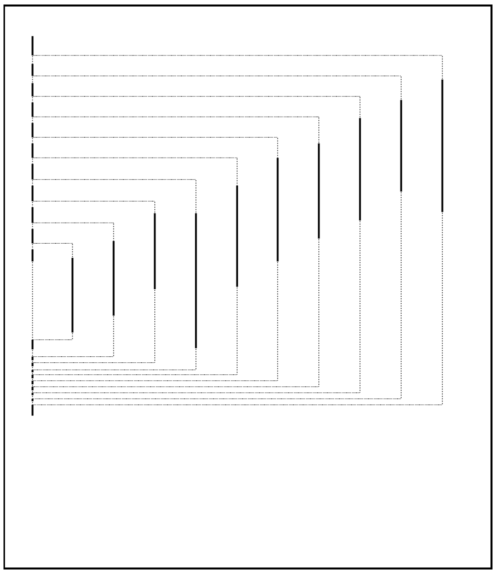

Figure 4: Vector operation, keeping recursion (10 el./8 proc.)

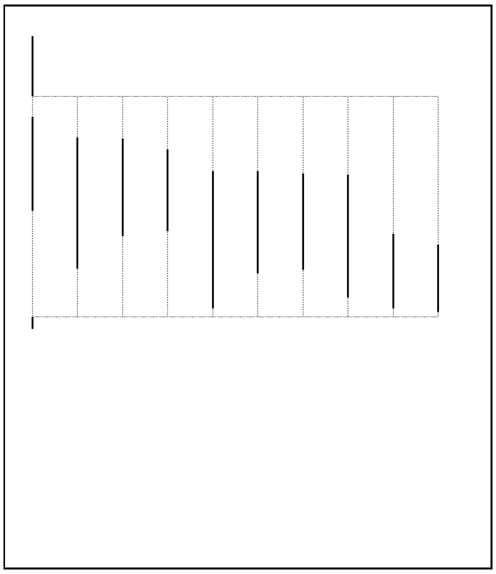

Figure 5: Vector operation, flattened for 10 elements (10 el./8 proc.)

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 of the pictures.

The result of the execution of this "data–parallel" program 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 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: 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 [Bar90].

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. In fact, we believe that very little modification to the PWAM would be necessary, and we will address this issue in Section 4. 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 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 [ABB93]. 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 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],[HR1,HR2,HR3]) :-
        !,
        process_element(H1,HR1) &
        process_element(H2,HR2) &
        process_element(H3,HR3).
vproc([H1,H2],[HR1,HR2]) :-
        !,
        process_element(H1,HR1) &
        process_element(H2,HR2).
vproc([H],[HR]) :-
        !,
        process_element(H,HR).
vproc([],[]).
```

In this program the lists are traversed in steps of four elements, and clauses for the cases of lists with 3, 2, 1, and 0 elements are provided. Another alternative would be to restrict the number of special cases (for example, taking into account only the partial lists whose length is a power of two, and making a recursive call in each of these cases) to avoid the number of clauses to grow linearly with the skipping factor.

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[4] [Mil90], which has given sufficient conditions for performing these transformations for particular cases such as linear recursion.

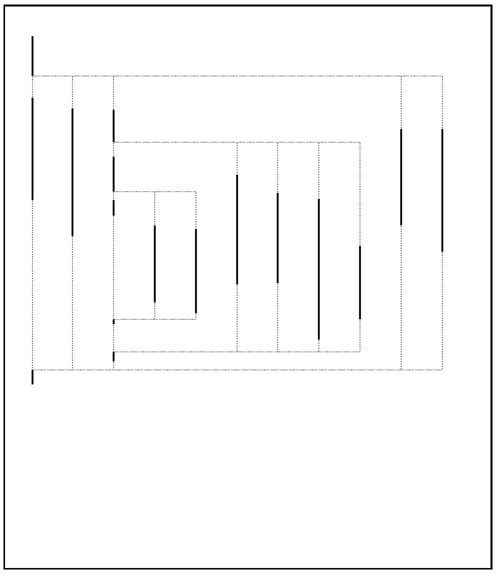

Figure 6: Vector operation with fixed list flattening (10 el./8 proc.)

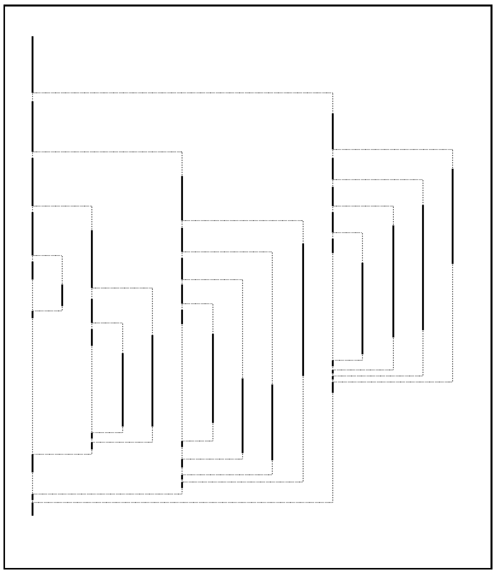

Figure 7: Vector operation with flexible list flattening (10 el./8 proc.)

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 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 in Section 3).

2.5 Dynamic Unfolding In Parallel

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 8. Instead of simply performing a unification of a fixed

[4]And has been used in &–Prolog compilation informally (see e.g. [WH87] and some of the standard &–Prolog benchmarks).

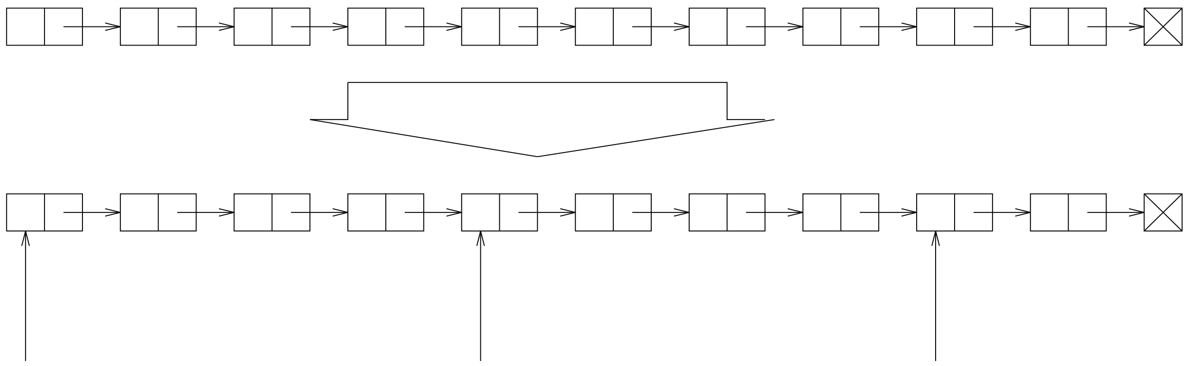

Figure 8: "Skip" operation, 10 elements in 4

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 8 depicts the pointers returned by skip(L,N,LS,NS) to a 10 element 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):

```
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 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. We believe that this idea would also be useful when implemented at an even lower level (Section 4).

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 composed of 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.

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.

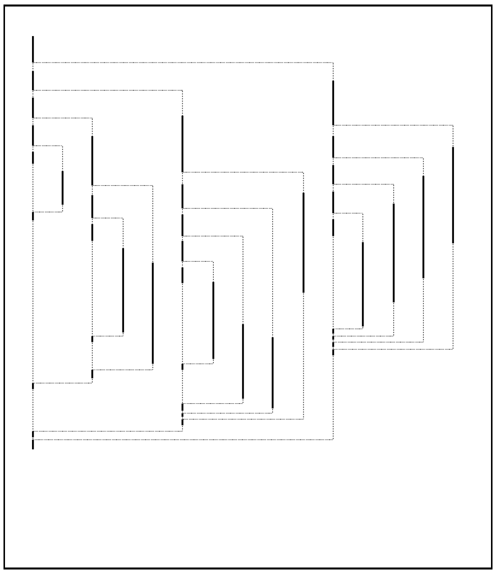

Figure 9: Vector operation, list prebuilt (10 el./8 proc.)

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).
```

A typical call to skiplist/3 would be done with the two first arguments instantiated; the third argument would return pointers to sublists of the first argument or, under a more

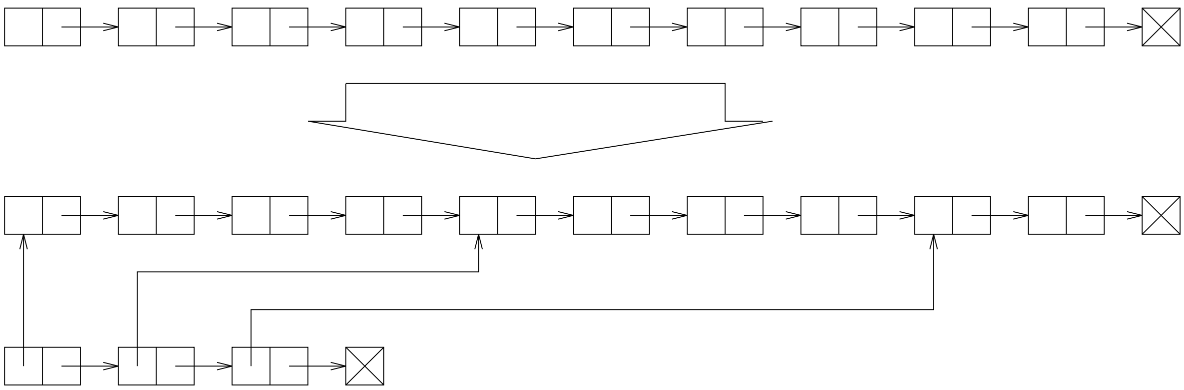

Figure 10: “Skiplist” operation, 10 elements in 4

logical point of view, the third argument describes a set of sublists of the first argument by means of difference lists. Figure 10 depicts this situation, and Figure 9 shows the result of an execution where the input and output data has been preprocessed using this predicate. This list preprocessing does not appear in Figure 9, 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 elements) list. The results presented in Table 1 show the corresponding execution times. The column *Relative Speedup* refers to the speedup with respect to the parallel execution in one processor, and the column *Absolute 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.

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.

The speedups suggested by Figures 4 to 9 may not correspond with those in the table — the length of benchmark run and the skip/unfolding increment chosen in the two cases is different, and so is the distribution of the tasks. In fact, some figures suggest a slowdown where the table shows a speedup. On the other hand, this indicates that processing larger lists can take more advantage from the proposed techniques, because the relative overhead from traversing the list is comparatively less.

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.

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.

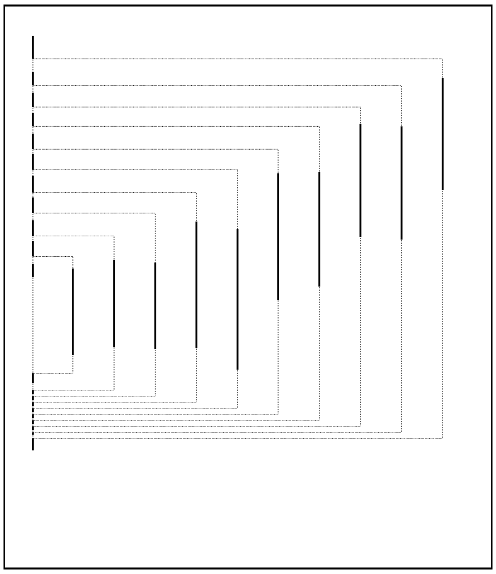

Figure 11: Vector operation, constant access arrays (10 el./8 proc.)

3 Constant Time Access Arrays in Prolog?

One obvious disadvantage of the techniques previously proposed is that a traversal of the input data needs to be performed at some point (in the best case, only once at the beginning of the execution, and several times in worse cases). This adds a sequential component to some programs which can limit the attainable speedups. Arrays, on the other hand, have the desirable property of allowing constant–time access to any element through an arithmetically manipulable index. This, in our case, means that the cost of skipping elements to split the computation does not depend on the number of elements skipped, or on the total length of the array. We will apply the techniques proposed in the previous sections to the case of arrays, and we will compare the results in this case with those obtained for lists.

3.1 Traversing and Splitting Arrays

For the sake of argument, we propose a simple-minded approach to the original problem using the real "arrays" in standard Prolog, i.e., compound 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 very easily 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 parallelized "vector" version of `vproc`):

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

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

```
process_element(I,V,VR) :-
        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 11. 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 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. From this figure it may seem that there is no performance improvement derived from using this strategy. This 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. As in Table 1, in Table 2 larger lists 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 traversed can be dynamically adjusted with no extra cost.

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

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

to be half the length of the sub–vector assigned to the task. Figure 13 depicts 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.

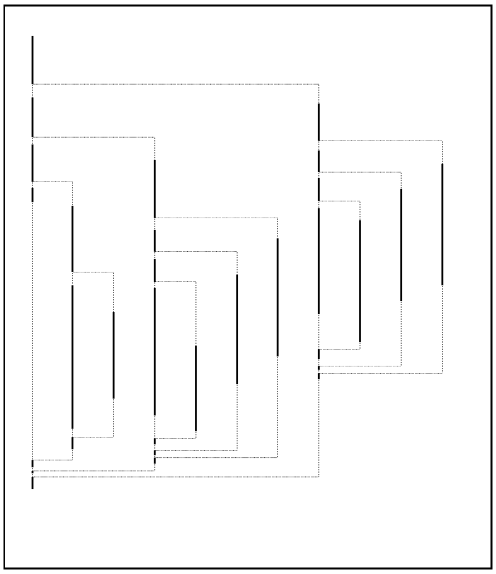

Figure 12: Vector operation, constant time access arrays, skipping, 10 el./8 proc.

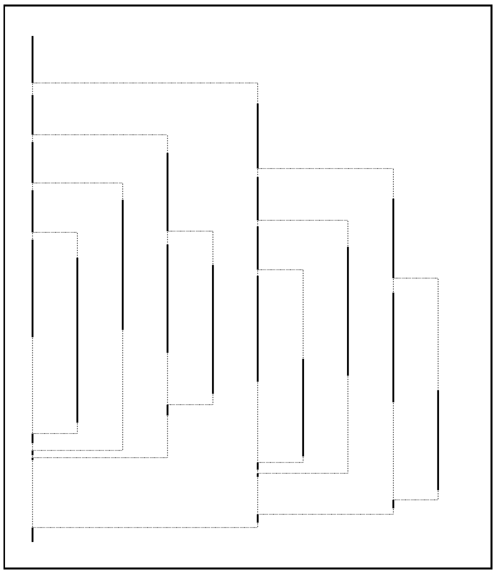

Figure 13: Vector operation, constant time access arrays, binary startup, 10 el./8 proc.

It is clearly also trivial to convert from a list representation to a "vector representation" — e.g. for the one dimension case:

```
vectorize(L,V) :- vectorize(L,0,V) .

vectorize([],N,V) :-
        functor(V,storage,N).
vectorize([H|T],N,V) :-
        N1 is N+1,
        vectorize(T,N1,V),
```

```
arg(N1,V,H).
```

Comparing Tables 1 and 2 some conclusions can be drawn. First, the structure–based programs are slightly slower than their list–based counterparts. This is understandable in that using structures as arrays involves an 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; this can be attributed to the fact that the &–Prolog version with which these times were taken 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.

3.2 Building Arrays in Prolog

Finally, following on on the idea of this section, 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. 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 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.

The "type" array can be defined as 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). From that idea, Prolog code to check the array object, to create arrays, and to consult and update (even destructively) them is easy to derive. More realistically, all these operations should be builtins (or, even better, native instructions) for performance reasons. Note that these primitives could in any case often be very efficiently compiled in–line to specialized calls to `functor`, `arg`, etc.

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):

```
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 (or in a CLP language) one can also create arrays through a simple call to the type definition predicate.

All elements of the "storage" part are accessible as arguments of a structure in time proportional to the number of dimensions of the matrix (and thus fixed for each array, regardless of the actual number of elements in it):

```
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.

The definitions above are meant as a description of the logical meaning of the operations on arrays (except for the destructive assignment, of course). From a practical point of view, these definitions should at least be changed to compute with an accumulating parameter. Also, use of delay (or CLP) can make them fully reversible. More realistically, all these operations should be builtins (or, even better, native instructions) 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.

4 Adding Lower Level Support

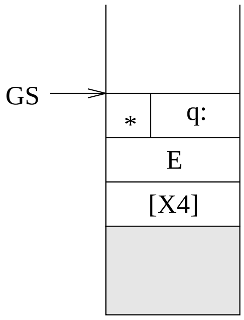

Figure 14: Multigoal Goal Stack Entry

We hope to have convincingly argued that much "data–parallel" computation can be done in an efficient way in the context of more general and–parallel systems, with the obvious advantage of not having to trade generality for efficiency in a special case. However, it would be incorrect to conclude that the techniques proposed achieve the same task startup efficiency as a native data–parallel system, specially for fine–grained computation. Clearly, there is merit for certain cases in supporting fast task creation at a lower level. However, we argue that this can be easily done, again with the advantage of avoiding losing generality (i.e., the ability to support general and–parallelism in other parts of the program), by adding either a special purpose builtin or a few special instructions to the instruction set of a typical general–purpose and–parallel engine.

As an example, we propose two simple extensions to the PWAM abstract machine in order to natively support a very simple, bounded quantification–like construct: `p_forall/2`, which checks that a certain property holds for all elements of a list, in the spirit of some of the constructs proposed in [ABB93]. We illustrate this extension through an example. Consider the following source construct:

```
..., p_forall(X in L, q(X,Y,a)), ...
```

which checks that a property `q/3` holds for `Y`, `a`, and each of the elements of the list `L`. If we suppose that the different `q/3` calls for each element in list L are independent (according to the notion of independence followed throughout this paper), then those calls can be safely executed in independent "and" parallelism.

A first implementation would be through a builtin directly corresponding to this construct, which would create standard PWAM tasks in the goal stack but in a tight low–level loop, rather than in a Prolog recursion (a "`pmap`" builtin, somewhat in this spirit, was present in early versions of the &–Prolog system [WH87]). Assuming knowledge of the determinacy of the iterations, these tasks could be of the "`det_pcall`" type [Her86a], which substantially reduce marker overhead.

Alternatively, a special class of entries in the goal stack can be defined where a single such entry identifies a collection of goals. The skeleton of the object code corresponding to the example above, in a slightly extended PWAM instruction set, follows. We assume that permanent variable `Y1` points to the list `L` and that `Y` is given by `Y2`:

```
        ...
        vectorize Y1,Y3,X4
        det_pcall 1,forall(X4,q:)
        pop_wait
        ...
q:      put_constant  a,X3
        put_value     Y2,X2
        put_indexed   Y3,X1
        call q/2
        return_par
        ...
```

All instructions used are essentially the same as in the PWAM, except for the `vectorize` instruction, taken directly from the Reform Prolog abstract machine, the `put_indexed` instruction, and the new "forall" type of argument to the `det_pcall` instruction. The `vectorize` instruction is essentially a low–level implementation of the `vectorize/2` builtin described in the previous section, returning the vector version of `Y1` in `Y3` and the length of the vector in `X4`. The `det_pcall` instruction is the same as in the PWAM, creating a goal stack entry for other processors to steal. However, the goal is marked specially (through the `forall` label) so that, in additional to the usual pointer to the current environment and to the goal code (`q:`), also an index field is added, which is initialized to the contents of `X4`. The goal is also marked specially ("*") to signal that it actually represents several goals (see Figure 14). Any processor stealing the goal now placed in this goal stack will see that it is a special "forall" goal and will save the value of the index in a special register before executing starting execution at address `q:`. Also, the value of the index in the goal stack will be decremented. Finally, if when decrementing this index the last goal is reached, the goal stack entry is deleted from the goal stack, as in the case of standard entries. Of course, the usual bookkeeping operations on the *parcall* frame [Her86a] such as setting up task return counters (i.e., the simplified versions implied by a `det_pcall` instruction) are also performed.

The advantage that the approach outlined inherits from its data–parallelism / Reform Prolog lineage is that all tasks are created in only one, constant time operation, and that the operation of traversing the input list is done once ahead of time and in a tight, low–level loop. However, the full PWAM machinery remains available for supporting general and–parallelism in other parts of the computation. Through these instructions or slight variations of them, it is also possible to implement several optimizations, such as splitting a large computation into several computations, each of them represented by one such goal stack entry, as well as in general all the techniques mentioned in the previous sections. The point again is not so much to describe a precise instruction set, but rather to illustrate how data–parallel techniques can be incorporated at a low level in an and–parallel abstract machine without losing the support for general purpose and–parallelism.

5 Conclusions

We have argued that data–parallelism and and–parallelism are not fundamentally different and that by relating them the advantages of both can in fact be obtained within the same system. We have 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. We illustrated 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 iteration over data structures that are 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 results 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 [DJ94], and shown to also achieve significant speedups for that class of problems.

Acknowledgments

We would like to thank Jonas Barklund, Johan Bevemyr, and Hakan 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.

The authors have been partially funded by ESPRIT Project 6707 ParForce.

References

[ABB93] Henrik Arro, Jonas Barklund, and Johan Bevemyr. Parallel bounded quantification—preliminary results. *ACM SIGPLAN Notices*, 28:117–124, 1993.

[AK90] 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.

[Ali88] 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.

[Bar90] Jonas Barklund. *Parallel Unification*. PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1990.

[Bla92] Jens Blanck. Abstrakt maskin för Nova Prolog. Internal report, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1992.

[BLM93a] J. Bevemyr, T. Lindgren, and H. Millroth. Exploiting recursion-parallelism in Prolog. In *Proc. PARLE'93*, Berlin, 1993. Springer-Verlag.

[BLM93b] 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.

[BM88] Jonas Barklund and Håkan Millroth. Nova Prolog. UPMAIL Tech. Rep. 52, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1988.

[Car90] M. Carlsson. *Design an Implementation of an OR–Parallel Prolog Engine.* PhD thesis, SICS and the Royal Institute of Technology, S-164 28 Kista, Sweden, March 1990.

[CDO88] M. Carlsson, K. Danhof, and R. Overbeek. A Simplified Approach to the Implementation of And-Parallelism in an Or-Parallel Environment. In *Fifth International Conference and Symposium on Logic Programming*, pages 1565–1577. MIT Press, August 1988.

[CGH93] 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.

[Con83] 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.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[DeG87] D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.

[DJ94] 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.

[DLH90] 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.

[Fag87] B. S. Fagin. *A Parallel Execution Model for Prolog.* PhD thesis, The University of California at Berkeley, November 1987. Technical Report UCB/CSD 87/380.

[GHM93] 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.

[GHPC94] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–110. MIT Press, June 1994.

[GJ89] G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.

[GSCYH91] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: Integrating Dependent and-, Independent and-, and Or-parallelism. In *1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

[Her86a] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[Her86b] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[HG90] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[HG91] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HQ91] Philip J. Hatcher and Michael J. Quinn. *Data-parallel Programming on MIMD Computers.* MIT Press, Cambridge, Mass., 1991.

[HR95] 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.

[HtCg94] 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.

[Kac90] Péter Kacsuk. *Execution Models of Prolog for Parallel Computers.* Pitman, London, 1990.

[Kal87] L. V. Kalé. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Fourth International Conference on Logic Programming*, pages 616–632. Melbourne, Australia, MIT Press, May 1987.

[Kar92] R. Karlsson. *A High Performance OR-Parallel Prolog System.* PhD thesis, SICS and the Royal Institute of Technology, S-164 28 Kista, Sweden, March 1992.

[KK84] V. Kumar and L.N. Kanal. Parallel branch-and-bound formulations for and/or tree search. *IEEE transactions on pattern analysis and machine intelligence*, 6:768–778, November 1984.

[KS90] A. King and P. Soper. Granularity analysis of concurrent logic programs. In *The Fifth International Symposium on Computer and Information Sciences*, Nevsehir, Cappadocia, Turkey, October (1990).

[LHD94] 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.

[LK88] 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.

[Lus90] E. Lusk et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

[Mil90] Håkan Millroth. *Reforming Compilation of Logic Programs.* PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1990.

[Mil91] 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.

[MR90] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford Univ. Press, Oxford, 1990.

[Nai88] L. Naish. Parallelizing NU-Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. University of Washington, MIT Press, August 1988.

[NT88] Martin Nilsson and Hidehiko Tanaka. A Flat GHC implementation for supercomputers. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Intl. Conf. Symp. on Logic Programming*, pages 1337–1350, Cambridge, Mass., 1988. MIT Press.

[PGH95] 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.

[PGT95] E. Pontelli, G. Gupta, and D. Tang. Determinacy Driven Optimizations of And-Parallel Prolog Implementations. In *Proc. of the Twelfth International Conference on Logic Programming.* MIT Press, June 1995.

[PGT+96] E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. Hermenegildo. Improving the Efficiency of Nondeterministic And–parallel Systems. *Computer Languages*, 1996.

[Pre94] 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.

[Ric89] L. Ricci. *Compilation of Logic Programs for Masively Parallel Systems.* PhD thesis, Università di Pisa, December 1989.

[SCWY90] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, April 1990.

[SCWY91] 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.

[SH94] K. Shen and M. Hermenegildo. Divided We Stand: Parallel Distributed Stack Memory Management. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 185–203. Kluwer Academic Press, 1994.

[She92] 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.

[Thi86] Thinking Machines Corp., Cambridge, Mass. *The Essential *LISP Manual*, 1986.

[Thi90] Thinking Machines Corp., Cambridge, Mass. *C* Programming Guide*, 1990.

[Vor92] 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.

[War87] D.H.D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *International Symposium on Logic Programming*, pages 92–102. San Francisco, IEEE Computer Society, August 1987.

[War88] D.H.D. Warren. The Andorra Model. Presented at Gigalips Project workshop. U. of Manchester, March 1988.

[WH87] R. Warren and M. Hermenegildo. Experimenting with Prolog: An Overview. Technical Report 43, MCC, March 1987.

[Wis86] M. J. Wise. Experimenting with epilog: Some results and preliminary conclusions. In *13th Annual International Symposium on Computer Architecture*, pages 130–139. IEEE Computer Society, June 1986.

[ZTD+92] 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.