Detection of Scans in the Polytope Model

Xavier Redon* Paul Feautrier†

March 14, 2001

Abstract

Most automatic parallelizers are based on the detection of independent operations. Dependence analysis is mainly a syntactical process, in which the actual data transformations are ignored. There is another source of parallelism, which relies on semantical information, namely the detection of *reductions* and *scans*. Scans and reductions are quite frequent in scientific codes and are implemented efficiently on most parallel computers. We present here a new Scan detector which is based on the normalization of systems of recurrence equations. This allows the detection of scans in loops nests of arbitrary depth and on multi-dimensional arrays, and gives a uniform treatment for scalar reductions, array reductions, and arrays of reductions.

keywords: Automatic parallelization, reductions, scans, programs transformations, programs normalization.

1 Introduction

1.1 Motivation for Scan Detection

Most automatic parallelizers, whether industrial or academic, are based on the detection of independent operations. Operations are independent if they do not share modified data, or, in other words, if they have no memory conflicts. Detection of memory conflicts is mainly a syntactical process, in which the actual data transformations are ignored. There is another source of parallelism, which relies on semantical information, namely the detection of *reductions*. A reduction is the application of an associative binary operator to a vector of values, the result being one scalar. Reductions are quite frequent in scientific codes and are implemented efficiently on most parallel computers. Scans are similar to reductions, but the intermediate results are kept, thus giving a vector. Since there is no single rule for detecting associative operators, scan detection must

*Laboratoire LIFL, Université Lille I, 59655 Villeneuve d'Ascq Cedex, France. e-mail: Xavier.Redon@lifl.fr

†Laboratoire PRiSM, Université de Versailles-St. Quentin, 45, Avenue des Etats-Unis, 78035 Versailles, France. e-mail: Paul.Feautrier@prism.uvsq.fr

use pattern-matching, and pattern-matching must be preceded by a normalization phase. Our method is based on a representation of the program as a system of recurrence equations. This allows the detection of reductions in loops nests of arbitrary depth and on multi-dimensional arrays. Scans and reductions are represented in symbolic form with the help of the Scan operator. When scans and reductions have been detected, one may use the results to construct more efficient parallel programs. While partial solutions are known [16], the design of a universal method is beyond the scope of this paper.

1.2 The Dataflow Graph

In the area of automatic parallelization a very useful structure is the DataFlow Graph (DFG). This structure is more accurate than the classical Dependence Graph (DG) which describes the pairs of statements having memory conflicts. Some memory conflicts are due to memory reuse. They can be satisfied either by enforcing sequential execution, or by expanding the data space of the program. In the DFG, all dependences which can be satisfied by data expansion have been removed. Hence, the DFG has the potential to expose more parallelism than the ordinary Dependence Graph.

The DFG gives, for each reference to a scalar or array element in an operation, the source operation, i.e. the operation that defined the value of the scalar or array element we are interested in.

Consider, for example, the following code:

```
DO i=1,n
 s(i)=a(i)*b(i)      (i1)
END DO
s(0)=0               (i2)
DO i=1,n
 s(i)=s(i-1)+s(i)    (i3)
END DO
```

Its Dataflow Graph as given by our automatic analysis tool, PAF, is:

```
Source of s(i-1) in i3 ;
   * if [ i-2>=0 ] then (i3,i-1)
   * if [ i-1=0 ] then (i2)
Source of s(i) in i3 ;
   * if [ true ] then (i1,i)
```

Note that the source of `s(i-1)` in `i3` is a conditional expression. Such expressions occur frequently in Dataflow Analysis and are called quasts (Quasi-Affine Selection Trees) in what follows.

The results of Array Dataflow Analysis can be presented in many ways, all equivalent up to syntactical embellishments: source functions, a single assignment code, or a system of affine recurrence equations (SARE). The later

presentation is the one that fits best with our aims in this paper; we will use the ALPHA notation to represent SAREs [12]. The important point here is not notation, but the fact that an imperative program can be mechanically converted to a SARE provided it meets the following constraints:

- The only data structures are arrays.
- The only control structures are `DO` loops.
- The array subscripts are explicitly given affine functions of the loop counters.

  The DFG of our example program can be expressed as the following SARE:

```
i1[i] =  1<=i<=n   : a[i] * b[i] ;

i3[i] = case
           i=0   : 0 ;
           1<=i<=n   : i3[i-1] + i1[i] ;
        esac
```

  In this notation, conditions such as { `1<=i<=n` } define iterations domains, while the subscripts expressions like `i-1` are shorthand notations for dependence functions, in this case $\lambda i.(i-1)$.

The reason for using SAREs as the basis of our work is that SAREs are *referentially transparent*, which means that a variable can be freely replaced by its value as given by the right hand side of the equation that defines it. This is the main tool in our reduction detection method.

Let us emphasize the fact that the DFG and SARE are equivalent representations. SAREs are more self-contained and are easier to handle in an automatic system, while DFGs are susceptible of an intuitive graphical representation. In this paper, we will freely shift from one representation to the other according to the needs of the exposition.

2 Overview of the Method

2.1 What is a Reduction?

A reduction – the name comes from the APL language [7] – is any operation which takes a set of values as argument and gives a single value in return[1]. Most often, a reduction is defined as the repeated application of a binary function to the elements of the input set in some order:

$$s = f(\ldots f(a_1, a_2), \ldots, a_n). \tag{1}$$

[1] We acknowledge that this definition lacks in precision. It should be completed by complexity considerations: the "size" of the result should not increase, or, at least, increase more slowly than the total size of the arguments.

Specially interesting is the case where f is an associative function. Commutativity and the existence of a unit are also useful. With the help of these properties, definition (1) can be rearranged in various ways in order to exhibit parallelism. As a rough estimate, reduction of n values on P processors takes about $\frac{n}{P} + \log_2 P$ units of time, excluding communication delays if any.

APL was the first language to introduce a specific notation for reduction. In contrast, many ordinary imperative languages, new and old, have no such feature. In Fortran, for instance, one has to write:

```
Program reduc
s = a(1)
do i = 2,n
  s = f(s,a(i))
end do
end
```

and the situation is similar for C or Pascal. With the advent of massively parallel computers, there is a strong incentive to extract the last ounce of parallelism from sequential programs, hence the importance of detecting associative reductions.

The reader must be warned that some computer operators are only approximately associative. For instance, addition of floating point numbers is associative up to rounding errors. While there is no reason to suppose that a parallel reduction is less precise than a sequential one [13], the user must be aware that a program to which reduction parallelization has been applied may not give exactly the same results as its sequential original. The extent of the difference depends on the numerical stability of the algorithm, and this question is beyond the scope of this paper.

2.2 Detecting a reduction

The question is thus to identify reductions when they are concealed in a sequential program.

It is not too difficult to identify code such as:

```
do i = ...
  s = f(s)
end do
```

The question is then whether the assignment may be rewritten as

```
s = g(s, e(i))
```

with g an associative operation. This is a difficult mathematical question. Usually, the associativity of a binary operation has to be proved, and such a proof may be of arbitrary difficulty. Since it is not possible, at least with present technology, to include a theorem prover in a compiler, the obvious solution is to use a catalog of predefined reductions and to match this knowledge base against the object program. This is done in many commercial parallelizers, which are able to recognize elementary reductions like sums and dot products.

Such recognition of reductions by pattern matching is more complicated than it looks: consider for example the problem of summing the components of a vector. It may be written in many different ways, among which:

```
program A              program B                  program C
s = 0.                 s = a(1)            1      s = 0.
do i = 1,n             do i = 2,n                 do i = 1,n
    s = s + a(i)           s = a(i) + s    2         t = s + a(i)
end do                 end do              3         s = t
end                    end                        end do
                                                  end
```

Furthermore, most of the time, the summation loop above will be encumbered by extraneous code, which may interfere – or not – with the summation. It would need a very large knowledge base to recognize all these forms as equivalent.

The first idea is to go beyond "first order" pattern matching as is found, e.g., in parsers or in such languages as ML. For instance, the best way of deciding whether a statement `s = f(s)` is a summation is to compute (with the help of a computer algebra system) the quantity $f(s) - s$ and to test whether the result is independent of s or not. Similarly, the best way to handle extraneous code in the loop body is to analyze its dependence graph. We are thus lead to the use of "matching functions" of arbitrary complexity. The set of matching functions is the knowledge base of our reduction detector. It needs not be fixed for all times and circumstances. Special application fields may have their own type of reductions and associated matching functions. For all these reasons, it is impossible to combine the pattern matching functions into one grand pattern, as in done, for instance, in LR(k) parsers. Application of the several matching functions must be sequential, hence our insistence in having as few of them as possible.

As is well known, *normalization* of the object program is a powerful tool for reducing the size of the knowledge base. Ideally, all equivalent programs should transform into the same *normal form* and the reduction detection should be done on the normal form. This is clearly impossible, since it would give an algorithm for testing the equivalence of two programs, which is undecidable as soon as the programming language is powerful enough. The way out is both to restrict the input language – to static control programs in our case – and to accept imperfect normal forms, in which some but not all equivalent programs have the same normalization.

> For instance, with the system which is described in this paper, program A and C normalize to the same form, but program B does not. This is as it should be, since program A and B are equivalent only if $n \geq 1$. But even adding this information will not enable us to have program A and B converge to the same normal form, unless we find a way for doing formal computations on reductions, a subject for future research.

Many researchers have defined normal forms for use in reduction detection and other program transformations. The best known one is obtained by *symbolic execution* [8]. When the presence of a reduction in a loop is suspected, its

body is submitted to an algebraic interpreter. Consider for instance program C, and let s_i^0, t_i^0 be the values of `s` and `t` at the beginning of iteration i of the loop. Similarly, let s_i^k, t_i^k be the values of these variables after execution of statement k in iteration i. Symbolic interpretation of statement 2 and 3 gives successively:

$$\begin{aligned} t_i^2 &= s_i^0 + a(i) \\ s_i^3 &= s_i^0 + a(i) \end{aligned}$$

At this point, one starts the next iteration, meaning that $s_{i+1}^0 = s_i^3$. We have thus found the recurrence:

$$s_{i+1}^0 = s_i^0 + a(i).$$

Programs A and B would lead to the same recurrence. Note that adding extraneous statements to the loop (e.g. resetting `a(i)` to zero after statement 3) would not change the result of symbolic execution, thus showing that this modification does not affect the reduction.

This method is quite powerful and can be extended to handle conditional statements in the loop body [9]. It needs a formal algebra system and its power is directly proportional to the normalization power of this system. However, it cannot handle either reductions on arrays or arrays of reduction (i.e. systems of recurrence equations). The reason is that reference to arrays elements cannot considered as mathematical variables. It is not always true that two occurrences of `a[i]` refer to the same value, or that an occurrence of `a[i]` and an occurrence of `a[j]` refer to different values.

Callahan [2] has proposed a method which can handle systems of reductions provided they are linear (i.e. their solution reduces to matrix products). The basic idea is similar to symbolic execution, but the interpreter handles only linear right hand sides. Variables which are the result of non linear calculations are treated as undefined, and the undefined value has to be propagated across the symbolic calculation. At the end of the process, the linearly computed variables at the end of one iteration are expressed as functions of their values at the beginning of this iteration. Computing their final value is equivalent to the computation of a succession of matrix products, which are associative. Most of the time, the iteration matrix is sparse: Callahan gives methods to exploit this fact in order to minimize the total computation time.

The method of [14] is quite different. The aim here is to normalize the dependence graph of the loop. Since the size of the dependence graph of a loop which is iterated n times is at least $O(n)$, this is impossible in general. The authors propose a limited unfolding of the loop body until periodic patterns become apparent. The recognition of reductions is done on this unfolded graph.

Our aim here is to improve on the symbolic execution method in order to be able to handle reductions on arrays as well. Ideally, the following program

```
program D
real a(0:100)
s(0,k) = 0.0
```

```
do i = 1,n
    s(i,k) = s(i-1,k) + a(i)
end do
```

should have the same normal form as programs A and C.

2.3 Normalization Strategy

Our basic requirement is that the representation of a program must be a well defined mathematical object, to be transformed by applying the usual algebraic rules, the most important one being substitution of equals for equals. This requirement is satisfied in a limited way by symbolic execution: in a basic bloc, the value of a scalar variable after the execution of a statement is a mathematical expression, which can be subjected to ordinary algebraic calculation. This is no longer true if one is interested in arrays and if one wants to handle several loops as a whole.

In that case, one has to switch to a representation by a SARE in the manner of [15]. An object in such a system is the value of a variable at a well defined point in the execution of a program (a statement and a set of counters for the surrounding loops), or equivalently, a point in the domain of one the variables. The system gives relations between these values. Solving the system is equivalent to running the program.

The Dataflow Graph, however is not a sufficiently powerful normal form for reduction recognition. For instance, while programs A and D have the same DFG, this is not true for program C. In fact the DFG's of all simple reductions we want to recognize have a very simple form which is depicted in figure 1. The only circuit [2] in the graph is a loop and the dependence function is a

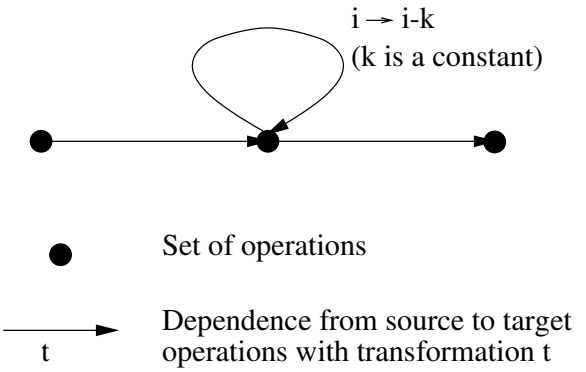

Figure 1: DFG of a simple reduction

translation. If a DFG has circuits which are not loops, we may try to shrink them by eliminating variables by substitution. We will show later that this

[2] In graph theory, a circuit is a cycle with edges oriented in the same direction.

is not always possible, and give a necessary and sufficient condition for this strategy to succeed. When a loop is found, one tries to identify a reduction by matching the associated equation with a knowledge base. Pattern matching will be discussed in details in Sect. 4.

A program may have a large number of reductions. Once one of them is found, we must set it aside, as it where, and try to find other ones. This may be done simply by introducing a new operator, `Scan`, which has the same relation to reduction as the integral sign $\int$ has to integration: it allows one to give a name to an object which is not always expressible in closed form within the underlying theory.

In graphical terms, introducing a `Scan` operator allows one to delete the loop on the corresponding node. When this is done, one may continue eliminating variables until other loops are found. This is best done working from inside outward, as in this way the simplest reductions are detected first.

The result of this analysis is a new version of the Dataflow Graph in which as many reductions as possible have been identified. The result may still be simplified by combining reductions to build higher order reductions. Consider for instance, the program

```
program E
s = 0.0
do i = 1,n
  do j = 1,i
    s = s + a(i,j)
  end do
end do
```

As a first step, we find that the j loop is a sum. The next step is to analyze the i loop, thus finding another summation which uses as data the results of the j loop summations. These results may be combined to give – we use here the ordinary mathematical notation:

$$s = \sum_{i=1}^{n} \sum_{j=1}^{i} a_{ij}.$$

3 Program normalization

The first step is always the construction of the DFG and transformation into a SARE. The reader is referred to [6] for a detailed account of this process.

3.1 Elimination of Variables Using Substitutions

As we have said, we want to keep the complexity of the pattern matching phase as low as possible. Our choice is to consider only one variable reductions. The price we pay for that is our inability to detect "systems of reductions", as Callahan does in the special case of linear recurrences. Now, computations

often use temporary variables which lead to recurrences on several variables, therefore a system normalization must be provided to collapse multi-variables recurrences onto single-variable ones if possible. A similar situation occurs in the case of systems of first order differential equations:

$$x' = v,\ v' = f(x),$$

which can be transformed into one second order differential equation $x'' = f(x)$.

The recurrence defined by the system below is a two-variables one:

```
x[i] = case
         i | i = 1  :       0 ;
         i | 2 <= i <= n  : y[i - 1] + a[i] ;
       esac ;
y[i] = case
         i | i = 1  :       0 ;
         i | 2 <= i <= n  : x[i - 1] + b[i] ;
       esac ;
```

If our pattern matching is directly applied no scan can be detected since there is no self-referencing variable. Our solution is to eliminate either `x` or `y` from the system. For example if we replace the reference to `y` in `x` by its definition, the result is:

```
x[i] = case
         i | i = 1  :       0 ;
         i | i = 2  :       0 ;
         i | 3 <= i <= n  : x[i - 2] + b[i - 1] + a[i] ;
       esac ;
```

The variable `x` is now self-referencing (with stride 2) and a mere pattern matching can point out that `x` is to be computed by summing the data `b[i-1]+a[i]`. Going from the first to the second system is not a simple textual substitution, We had to compute the expression `y[i-1]` from the definition of `y` in the initial system, then substitute it into the definition of `x`, then simplify the result by eliminating cases with empty domains. See [17] for details.

We also want our scan detector to handle scans on multi-dimensional arrays. In this context another problem arises: scans can be computed along very different paths in these arrays. The problem of detecting scans along arbitrary paths seems intractable, hence we deal only with rectilinear ones. An uni-directional scan gathers its data following one vector. An uni-directional scan associated with a multi-dimensional array represents in fact a set of scans:

The following program computes a set of scans along the diagonals of the array `a`:

```
DO i=1,n
 DO j=1,n
  a(i,j)=a(i-1,j-1)+a(i,j)
 END DO
END DO
```

The uni-directional scan underlying this code has direction $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

Several detection schemes can be used according to the required precision. The simplest algorithm consists in considering the system of equations as a whole and eliminating as many variables as possible. At this point most of the recurrences are defined by only one variable and a pattern matching can take place. The drawback of this method is that since, as we will see later, removing all circuits in a DFG is not always possible, unguided substitutions may lead into a blind alley, while a more sophisticated algorithm may have found a solution.

No scan can be extracted from the following system using this simple scheme:

```
x[i,j] = case
            i, j | i=1, j=1          :  0 ;
            i, j | 2<=i<=n, j=1      :  y[i-1,m] ;
            i, j | 1<=i<=n, 2<=j<=m :  x[i,j-1] + a[i,j] ;
          esac ;
y[i,j] = case
            i, j | 1<=i<=n, j=1      :  x[i,m] ;
            i, j | 1<=i<=n, 2<=j<=m :  y[i,j-1] + b[i,j] ;
          esac ;
```

But the third clause of `x` and the second of `y` compute sets of scans. In fact the elimination fails because the recurrences defined by this system are cross-referencing (cannot be computed in parallel).

A more complex detection scheme can be designed using multistage eliminations. The principle is to consider only some references (i.e. equations between our multi-dimensional variables). In a first stage only the references related to the innermost loops of the original program are taken into account. This is equivalent to considering the innermost loops as stand-alone programs in which the external loop counters are considered as fixed parameters. Pattern matching is then applied and closed forms are introduced for the detected scans. In the second stage we add to the graph the references relative to the loops just surrounding the innermost ones. A normalization and a pattern matching are performed again and so on. In this way the elimination is obviously more efficient and we can detect more complex scans (see section 4.1). Since closed forms appear during the detection process, pattern matching may be applied on variables already defined by a scan. In most cases this denote a scan whose path is piecewise rectilinear, a multi-directional scan.

Our previous example can be handled if the counter `i` relative to the outermost loop is considered as a parameter. In this context no elimination is to be performed since the only remaining references are the self-referencing ones in the last clauses of `x` and `y`. Let `sum` be the textual equivalent of the $\sum$ operator, closed forms can be introduced for the scans:

```
x[i,j] = case
            i, j | i=1, j=1          :  0 ;
            i, j | 2<=i<=n, j=1      :  y[i-1,m] ;
            i, j | 1<=i<=n, 2<=j<=m  :  x[i,1]+sum(j>=2,a(i,j)) ;
          esac ;
y[i,j] = case
            i, j | 1<=i<=n, j=1      :  x[i, m] ;
            i, j | 1<=i<=n, 2<=j<=m  :  y[i,1]+sum(j>=2,b(i,j)) ;
          esac ;
```

It is very important to note that the closed forms are *parametric* with respect to `i`.

A technical issue for the multistage elimination scheme is the characterization of the references to be taken into account. At the SARE level, there is no longer information about the loop nests. We replace the missing information with the help of the concept of reference "pseudo-depth". Let the reference to a variable `y` in an equation for `x` be of the form:

$$\forall i \in \mathcal{D}, \mathrm{x}(i) = \ldots \mathrm{y}(d(i)) \ldots .$$

where d is the dependence function. The pseudo-depth is the greatest integer p such that :

$$\forall i \in \mathcal{D}, d(i)[1..p] = i[1..p].$$

Since dependence functions are supposed to be affine, the pseudo-depth can be obtained by formally computing the vector $d(i) - i$ and counting its leading zero coordinates.

When normalizing loops strictly deeper than p, one has only to consider the references of pseudo-depth greater or equal than p.

Proof: Only circuits of references can cause the elimination process to fail. The references of pseudo-depth greater or equal to p which are not references relative to the loops at a level strictly greater than p cannot be included in a circuit. ∎

The following algorithm summarizes the steps of the multistage elimination method:

Algorithm 1 *Scans Detection*

- *Let S be a SARE.*
- *For all pseudo-depths p from the maximum nesting level to* 0*:*

1. *Construct the dependence graph of S restricted to references of pseudo-depth equal or greater than p.*
2. *Compute the strongly connected components* [3] *of this sub-graph.*
3. *Try eliminating all variables but one in each component.*
4. *Apply pattern matching if the total elimination succeeded and rewrite the detected scans in closed form.*
5. *Remove inter-components references of pseudo-depth equal or greater than p using substitutions.*

3.2 Criterion for Total Elimination

The goal of our elimination phase is to collapse recurrences on several variables into recurrences on only one variable. The tool we use to perform the transformation is substitution. In this section we give a criterion for deciding if a SARE can be transformed into a single variable recurrence.

To introduce our proof let us consider a system of ordinary equations.

$$\begin{cases} x_1 = f_1(x_{k_1^1}, \ldots, x_{k_{m_1}^1}) \\ \vdots \\ x_i = f_i(x_{k_1^i}, \ldots, x_{k_{m_i}^i}) \\ \vdots \\ x_n = f_n(x_{k_1^n}, \ldots, x_{k_{m_n}^n}) \end{cases} . \tag{2}$$

We do not make any assumption on the functions $(f_i)_{i\in\mathbb{N}}$ nor on the variables x_i. In this context substitution is really the only operation we can use on the system. Such a system can be represented using a graph. The vertices are the n variables and there is an edge from x_i to x_j if x_i appears in the definition of x_j. Let us consider the effect of the substitution of x_0 into x_1, x_2, ..., x_q. The edges from x_0 to the other vertices are removed and the k_0 predecessors of x_0 are added to the predecessors of the targets. Note that the new number of predecessors of, for instance, x_1 is not exactly known. If we denote by k_1 the initial number of predecessors of x_1, it is only possible to say that the new number of predecessors k_1' is between $\max(k_1, k_0)$ and $k_1 + k_0$: some of the predecessors of x_0 may also be predecessors of x_1, and are only counted once. Substitution for x_0 eliminates it if the equation for x_0 is not self recursive as shown on figure 2.

The substitution process terminates when no more variable can be eliminated. The process succeeds if, in the terminal system, all circuits are loops.

[3] A strongly connected component of a graph is a subset $\mathcal{S}$ of its vertices such that any couple of vertices from $\mathcal{S}$ can be connected by a path, i.e. a succession of edges oriented in the same way. A strongly connected graph is a graph such that the set of its vertices is a strongly connected component.

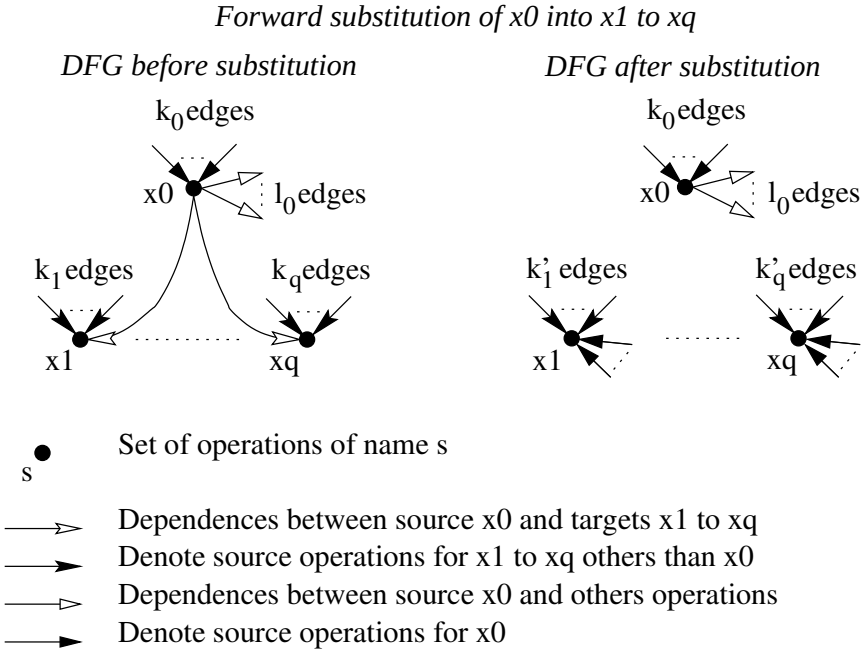

Figure 2: Effect of forward substitution on DFG

Theorem 2 (Sufficient Condition for a Total Elimination) *If the circuits of a strongly connected system* [4] *have a common vertex then the elimination process succeeds on the system.*

Proof: Without the edges going out of the common vertex the system graph is acyclic. Hence a topological sort can be performed on this sub-graph. The result of this sort is a partition of the vertices into sets $S_1, \ldots, S_p$. At this point one can substitute the definitions of the variables in S_1 into the definitions of the variables in S_2. These variables are now defined only in terms of the common vertex variable. Repeating the substitutions until S_p lead to a folding of all circuits into loops on the common vertex. ∎

Theorem 3 (Necessary Condition for a Total Elimination) *The elimination process succeeds on a strongly connected system only if its circuits have a common vertex.*

Proof: The sketch of the proof is as follows. Let G be a strongly connected graph such that for each vertex v of G there is a circuit which does

[4] A strongly connected system is a system whose graph is strongly connected. In the same way a strongly connected component of a system is the set of variables from a strongly connected component of its graph

not include v. We show that a graph G' obtained from G by applying some substitutions include a strong component with the same properties as G. Moreover such a strong component has at least two vertices, hence it has a circuit of length greater or equal to 2.

Let us now detail the proof. The basic property of transformation by substitutions is that if v^- is a predecessor of v in G, either v^- or its predecessors in G are predecessor(s) of v in G'. This implies a conservation property on paths. More precisely if there is a path p in G, then there is a sub-path of p in G which starts with the first or the second vertex of p and stop at the last vertex of p. A special case arises when the path p is in fact a circuit. In this case there is at least one circuit in G' which includes only vertices from p. From the conservation property on paths and circuits, another conservation property can be deduced on inter-circuits paths. Let c_1 and c_2 be two circuits of G, for each circuit c_2' built from c_2 in G', there is a path from a circuit built from c_1 in G' to c_2'.

Since each circuit of G' built from a circuit of G has as predecessor (via a path) at least one of the circuits of G' built from a given circuit of G (remember that G is strongly connected), there is in G' a strong component including at least one of the circuits of G' build from each circuit of G. For each vertex v of this strong component there exists a circuit of the component which does not include v. Indeed since v is not included in at least one circuit c of G, it is not included in the circuits of G' build from c (one of these is necessary in the strong component). ∎

These results apply only to strongly connected systems but they can be adapted for other systems. It suffices to compute the strong components of the system and verify that each component fulfills the requirement. Our elimination criterion formalizes an intuitive thought: it is not always possible to solve a system of equations using only substitutions. The criterion is mostly interesting when nothing is known about the functions $(f_i)_{i\in\mathbb{N}}$ of the system (2). In other cases methods using more powerful operations than substitution are generally used to solve the system.

In the context of linear equations, systems such as (2) can always be solved. This is due to the fact that the Gaussian elimination algorithm use substitutions *and also* linear simplification. Linear simplification is a powerful tool since it can remove loops from the system graph. For example in the graph of the system below there is a loop on the vertex x_1:

$$\begin{cases} x_1 = 4.x_1 + x_2 \\ x_2 = 2.x_1 \end{cases}.$$

After simplification of the first equation ($x_1 = -\frac{1}{3}x_2$) there is no longer a loop on x_1.

The criterion is also valid for a SARE. Its graph is more complex since some edges are defined only in a sub-domain of the variable definition domain. This has an influence on the elimination process. After a substitution, new edges domains are computed by intersection of the old domains. If this intersection is void, the edge does not really exist. Hence the criterion is always sufficient but no longer necessary. One may also say that the criterion is sufficient and necessary if no simplification of the clause domains occurs.

3.3 Algorithms for Variables Elimination

All algorithms for variable elimination on strongly connected components have the same pattern: find the common vertex of the system graph circuits and use a topological sort to schedule the substitutions (as described in theorem (2)). Hence the difficult point is to find the common vertex.

There is a full range of algorithms to perform this operation. Let us begin with a heuristic method, which is not very efficient but very fast. We use the fact that a graph without cycles is also without circuits [5]. Hence to find a vertex included in each circuit of a graph G, it suffices to consider each vertex v of G, to remove its outgoing edges and verify that the remaining graph G_v is acyclic. This verification can be done using the cyclomatic number [6] $\nu(G_v) = m' - n' + p'$, with m' the number of edges of G_v, n' the number of vertices of G_v (the same as the number n of vertices of G) and p' its number of connected components (i.e. 1 since G_v is a connected graph). The number of edges m' of G_v is the number of edges m of G minus the number m_v of v successors. Hence one has to check for each vertex v if the expression $m - m_v - n + 1$ is zero. The test complexity is about $O(n)$ but it is only a heuristic since it does not discriminate between cycles and circuits.

This heuristic can be transformed into an exact method. It suffices to replace the computation of the cyclomatic number by a depth-first search to insure that there is no circuit in the graph. This method complexity is of the order of $O(n.m)$.

A more efficient way for elimination is to use basic substitutions. Basic substitutions are substitutions for variables which have only one successor. The effect of a basic substitution on the DFG of a system is depicted on figure 3.

If a basic substitution is applied to a strongly connected graph, there is no circuit including x_0 and the resulting graph restricted to all vertices but x_0 is also strongly connected.

Proof: The vertex x_0 has no longer successors, so no circuit can include this vertex. Moreover if a path exists between two vertices (different from x_0) before the substitution, it also exists also after (even if the former includes x_0, since it goes through x_1 after the substitution). ∎

[5] Remember that no special orientation is required for the edges of a cycle but that the edges of a circuit must be oriented consistently.

[6] It give the number of cycles of a graph.

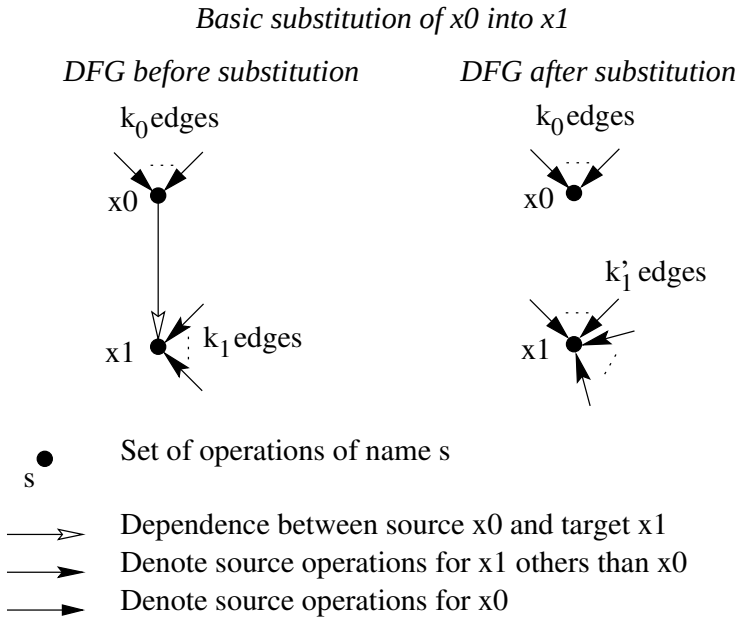

Figure 3: Effect of basic forward substitution on DFG

When no basic substitution can be applied to a graph, either its strong component is reduced to a vertex and, in the initial graph, every circuit include this vertex, or the strong component includes several vertices and a total elimination cannot be performed on the initial graph.

Proof: If the strong component is a single vertex, the elimination criterion implies that in the initial graph the circuits have a common vertex. Moreover the conservative property on circuits enforces that this vertex is the one in the strong component.

Let G' be a graph obtained from a strongly connected graph G using a basic substitution. Due to the properties of basic substitutions, if the strong component of G' is such that for each of its vertices v there exists a circuit not including v, then G has the same property. Now consider the final component including several vertices. No more basic substitution can be applied, hence each vertex has at least two successors. If a vertex v is removed, the others have still at least one successor, so there exists a circuit which does not include v. ∎

This prove that the sequential application of basic substitutions is as powerful as the application of multiple substitutions in the context of our variable elimination process. An efficient algorithm can be extracted from the method

provided that, for each vertex v, a sorted list of its predecessors `pred(`v`)` is available.

Algorithm 4 *Finding a Common Vertex*

1. *For each vertex v, compute its number of successors* `ns(`v`)` *and store one of these in* `succ(`v`)`.
2. *Initialize the stack* `s` *with the vertices having a unique successor.*
3. *Stop if* `s` *is empty or if there is only one vertex in the component.*
4. *Unstack v from* `s`. *Let v^+ be the successor in* `succ(`v`)`.
5. *Perform a sorting merge on* `pred(`v`)` *and* `pred(`v^+`)`, *for each v^- belonging to both lists subtract one to its number of successors* `ns(`v^-`)`. *When this number is equal to one stack v^- on* `s`.
6. *Store the result of the merging in* `pred(`v^+`)` *and for each v^- reset its peculiar successor* `succ(`v^-`)` *to v^+.*
7. *Goto step 3.*

The complexity of this algorithm is about $O(n.d^-)$ with n the number of vertices in the graph and d^- the maximum size of the `pred` lists. So it may even be linear if, for instance, the graph is a mere circuit.

One may remark that finding the common vertex of the system graph circuits is the instance $k = 1$ of the problem of finding the minimum cutset of size less or equal to k. Hence any one of the algorithms for finding minimal cutsets which find every cutset of size 1 are suitable for variable elimination. That excludes algorithm D of Shamir, but the one described in [10] is perfect. In fact, it works for cutsets of size 1 and it finds minimal cutsets for a number of graph classes. This last property is important since even a partial elimination is useful: one obtains a system with less variables, which can be analyzed faster by the later phases of the automatic parallelization process (e.g. scheduling).

Note that a better variable elimination can be achieved using a more precise system graph: the clause graph (whose vertices are clauses and not variables).

The system associated to the program below

```
DO i=1,2*n
  a(i)=a(2*n-i+1)
END DO
```

is

```
x[i] = case
         i | 1<=i<=n       : a[2*n-i+1] ;
         i | n+1<=i<=2*n   : x[2*n-i+1] ;
       esac ;
```

We name the clauses using the name of the variable and their rank in the variable definition. For instance the first clause of `x` is `x.0`. If `x` is included in a strong component, an elimination process may fail because of the loop in the variable graph. In contrast the clause graph does not have this loop, and so the elimination process may go further (see figure 4).

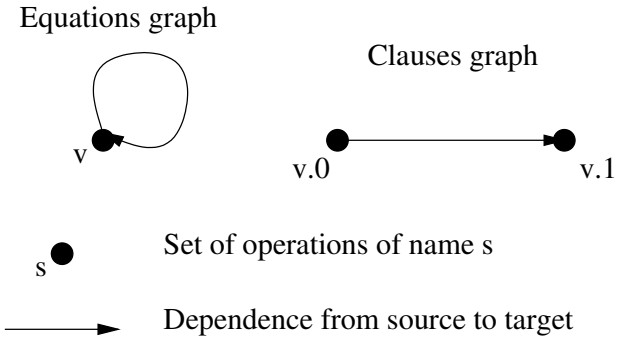

Figure 4: Example of difference between equation graph and clause graph

4 Identification of Scans

Identification of scans takes place after each phase of normalization: we thus need a way of memorizing the results before starting the next phase. We will first explain our notation for representing scans. Observe that this notation is not to be considered as a kind of function call or language statement. Its conversion to executable code is quite another problem. We will then explain how to extract scans from one-variable recurrences.

4.1 An Operator to Denote Scans

There exists some languages (mostly languages in the area of systolic arrays design) which include primitives for scan denotation. We can cite ALPHA and CRYSTAL (cf. [11]). Other formalisms like LACS [?] or PEI [?] can also describe scans in an elegant way. Some imperative languages have primitives for scans. A recent language with scan ability is HPF. It includes reduction primitives à la Fortran 90 such as `sum`.

The sum of the elements of a vector `x` can be computed by merely writing `sum(x(1:n))`. Our `Scan` operator is more general, and hence more complex:

```
Scan( i | 0 <= i <= n , ( [1] ), +, x, 0)
```

Let us explain the meaning of this notation. The binary operation to be applied (here, addition) is given as a parameter. The first two parameters (the accumulation domain and the direction) state that the reduction applies to the elements

of `x` from subscript `1` to `n` with a step of `1`. Subscript `0` is used for storing the initial value (the last parameter). Moreover the Scan operator computes the whole set of partial results. To extract the last value of the resulting vector, we use subscripting:

```
Scan( i | 0 <= i <= n , ( [1] ), +, x, 0) [n]
```

The HPF reduction primitives can use very complex patterns which are described by indirection arrays. This power of expression is interesting but the use of indirection arrays is not very convenient, since they have to be initialized beforehand. We favor a more synthetic information even if we lose some expressive power (for example, HPF allows the computation of reductions along circles, but that is not used very often in scientific programs). Our solution is to use a single vector to describe the reduction path. Two points in the scan domain are in the same path (and contribute to the same result) if one of them can be reached from the other by iterated translation along the direction vector. Points which cannot be reached in this way from other points in the domain are given initial values.

> Returning to the sum example, we easily see that there is only one path which include all points in the domain. Furthermore, point 0 cannot be reached by translation from other points, hence it is given the initial value, 0. The value of any other point, says $i \geq 1$, is obtained by applying $+$ to the value of its predecessor, $i - 1$ and to the local value, x_i.

In opposition to the reduction primitive, HPF is more restrictive concerning the scan primitives because it doesn't allow the use of indirection arrays; a scan is only possible along one dimension of the original array. Since our first transformation for program normalization uses the expansion of variables, we must deal with full scans, so we need more than a reduction operator. Hence we consider that the Scan operator compute an array of full scans; its result is of the same shape as the accumulation domain minus the initial values domain. To obtain a reduction one must reference the adequate element in the result array.

Our Scan operator has been enhanced in two aspects. Firstly, while we consider only single variable recurrences, these may have to be converted to vector or matrix scans in order to show associativity and hence parallelism. An example will be given later in Sect. 4.2.2. As a consequence, the data of a scan may be a complex object, and its operator a complex operator.

Secondly, in order to deal with scans computed along a piecewise rectilinear path, we have to introduce multi-directional scans in Sect. 5.

4.2 Scan Recognition

Suppose now that the result of the program normalization step is one (or more) self referencing equations:

$$i \in \mathcal{D} : x_i = \mathrm{Exp}(x_{d(i)}, \ldots) \tag{3}$$

where the ellipsis denotes occurrences of other variables and Exp is an arbitrary expression, which may contains several several occurrences of x with distinct dependence functions. Equation (3) is tractable if all references to x are of the form $x_{i-\delta}$ where δ is a unique integral vector. If this condition is fulfilled, δ is the direction of the scan. The domain of the scan is the domain of i, and the initial data is found from other, non recursive clauses in the system. It remains to find the operator of the scan and its data. This process will be presented here for the case $\delta = 1$. The generalization is just a matter of notations.

All expressions which are handled by a compiler are terms on a system of *operators* Ω, and a system of basic terms A. Here the basic terms are x_{i-1}, the other variables, and the constants of our language — integers, reals, truth values, and so on. To each operator ω is associated an *arity*: an integer denoted as $\partial(\omega)$. The rules for constructing legal terms are three:

1. A basic term is a term.
2. If $t_1, \ldots, t_n$ are terms, and if ω is an operator with $\partial(\omega) = n$, then $\omega(t_1, \ldots, t_n)$ is a term. ω is the *head* of this new term, and $t_1, \ldots, t_n$ are its *arguments*.
3. There are no other terms.

It is not even necessary that Ω and A be finite. All we need are well defined procedures for recognizing a basic term, recognizing an operator, and computing its arity.

The set of operators depends on the underlying programming language. It may include arithmetic operators $(+, -, *, /, \ldots)$, Boolean operators, elementary functions $(\sin, \cos, \log, \exp, \ldots)$, comparison operators, and so on. Legal terms must also conform to type rules. We will suppose that these rules have been checked by a first pass of the compiler. It is an easy matter to verify that the system of recurrence equations associated to a well typed program is well typed, and stays well typed if subjected to substitution.

4.2.1 Simple pattern matching

Let t be the term which is associated to Exp in (3). The simplest possibility is to analyze the head of t, and its arguments. Two trivial cases must be detected first. t may simply be x_{i-1}. In that case, the recurrence is a value propagation, and its solution is $x_i = x_0$. Similarly, if x_i does not occur in t, the solution is $x_i = t$. In both these situations, the recurrence has a trivial parallelization.

In other cases, we must check that the head of t is binary, and that, of its arguments, one is x_{i-1} and there is no occurrence of x_{i-1} in the other one. The head of t must belong to a list of *tractable operators*. This is the simplest form of pattern matching.

The disadvantage of simple pattern matching is that it fails as soon as t becomes too complicated. For instance, it cannot do anything with $t = (1 + x_{i-1}) + a_i$. On the other hand, adding new operators is easy. If we devise the

proper data structures, it can even be done without recompiling the detector program, simply by reading a "rule file".

4.2.2 Partial Normalization

The solution for finding the `Scan` operator in the recurrence

$$x_i = (1 + x_{i-1}) + a_i$$

is simply to rearrange t as $x_{i-1} + (1 + a_i)$ by using associativity and commutativity of $+$. Properties like associativity, commutativity, and many others can be expressed as equational axioms as in:

$$x + (y + z) = (x + y) + z.$$

Such axioms, when completed by the usual properties of equality give the set of terms the structure of an equational theory. Such a theory has a normal form if there exists a function $\mathcal{N}$ from terms to terms such that:

$$x = \mathcal{N}(x)$$

and

$$x = y \Rightarrow \mathcal{N}(x) \equiv \mathcal{N}(y),$$

where $\equiv$ denotes structural identity (two terms are structurally identical if they are constructed in the same way from the same basic terms). Since identity can always be checked mechanically, an equational theory which has a normal form is decidable provided that $\mathcal{N}$ is computable. Since we know that there are undecidable equational theories, we deduce that some axiom systems admit no computable normal forms.

In the case of addition, for example, we can obtain a normal form by sorting addends according to an arbitrary order in which constant terms come last. The normal form is obtained from the sorted expression by reducing the constants according to the rules of arithmetics.

When submitted to pattern matching, equal terms should give "equivalent" results. One way of guaranteeing this property is to normalize the given term before applying pattern matching. If we are clever enough, we can even define the normal form in such a way that pattern matching is simplified. For instance, in the additive example, we can select the ordering in such a way that variable x_{i-1} comes first.

Unfortunately, most interesting theories do not have a normalization algorithm. A heuristic solution is to use *partial normalization*, i.e. normalization which takes into account only a subset of the operators and axioms of the theory.

Let $O \subset \Omega$ and let t be a term. The *O-skeleton* of t is a construction of t in which only operators from O are used, and in which basic terms are terms from A or terms whose head does not come from O. Partial normalization can handle only terms in whose skeleton the elementary terms either are x_{i-1} or do not include x_{i-1}. Let us suppose that O has a normalization procedure. We can then normalize the skeleton of t, handling its elementary terms as baggage, and perform pattern matching on the resulting normal form.

Let us suppose that Ω is the set of operators in Fortran, and that x_i is a logical variable. Let us take the set of Boolean operators $\{\wedge, \vee, \neg\}$ as O. Viewing a Boolean expression as a tree, its O-skeleton is the "upper part" of the tree, from the root to the first occurences of a variable or of a comparison operator. Furthermore, since Fortran has no conditional expression and since the 0/1 convention of C does not hold, it is unlikely that a Boolean variable may occur beyond a comparaison operator (one would have to use a user defined function). Hence, Boolean expression have a high probability of being tractable.

Boolean algebra has, in fact, several normal forms. Conjunctive normal forms (CNF) can be used for detecting and-reductions, and disjunctive normal forms (DNF) for or-reductions.

Let us consider the following somewhat contrived recurrence:

$$x_i = (x_{i-1} \vee (\alpha_i \geq \beta_i)) \wedge (x_{i-1} \vee (\alpha_i < \beta_i)).$$

The Boolean skeleton of the right hand side is:

$$t = (x_{i-1} \vee a_i) \wedge (x_{i-1} \vee b_i),$$

where $a_i = \alpha_i \geq \beta_i$ and $b_i = \alpha_i < \beta_i$. The heads of these terms are not Boolean operators. t is in DNF, and x_{i-1} occurs twice, hence our recurrence is not an and-reduction. The CNF of t is

$$t = x_{i-1} \vee (a_i \wedge b_i),$$

which is an or-reduction. With a more powerfull normalization system, we would have noticed that $a_i \wedge b_i = \text{false}$, and hence that the recurrence has the closed form solution $x_i = x_0$. Such a normalization system must have the same power as linear programming; its construction is probably very difficult.

4.2.3 Marshalling Associative Operators

We still have to construct a list of associative operators. Some of them are well known: $+$, $*$, max, min, $\wedge$, $\vee$, etc. Is there a more systematic procedure?

Let us put the basic equation (3) into the form:

$$x_i = f_i(x_{i-1}). \tag{4}$$

This can be rewritten [4] as:

$$\begin{aligned} g_0 &= \lambda y.y, & (5)\\ g_i &= f_i \circ g_{i-1}, & (6)\\ x_i &= g_i(x_0). & (7) \end{aligned}$$

Since $\circ$ is associative, we may hope to compute the g_i's by a scan, and then to compute all x_i's in parallel. But this is a mere formal manipulation, without any practical interest, unless the complexity of g_i stays bounded as the computation

proceeds. In practical terms, this means that the functions f_i must belong to a family which can be described by a few parameters, and that this family must be closed under function composition.

As an exemple, consider the case where f_i is a polynomial in x_i. Since the composition of a polynomial of degree m with a polynomial of degree n is of degree mn, the family is closed only for the case $n \leq 1, m \leq 1$.

> Let us take as O the set $\{+, -, *\}$ with the usual properties: associativity and commutativity of $+$ and $*$, the rule of signs, distributivity of $*$ with respect to $+$, and the familiar rules of arithmetic ($2 + 2 = 4$ and so on). If t is tractable for this set of operators, its normal form is a polynomial in x_{i-1} whose coefficients are combinations of its elementary terms. We conclude that a recurrence $x_i = P(x_{i-1})$, is a scan only if polynomial P is of first degree:
>
> $$x_i = a_i x_{i-1} + b_i. \tag{8}$$
>
> There are two simple cases : $a_i = 1$, which gives a sum, and $b_i = 0$, which gives a product. In the general case, there are many ways of extracting a_i and b_i. Since practical normalization procedures are not always up to their mathematical specification, we may have to resort to "mathematical pattern matching". For instance, in the linear case, we can set $b_i = t[x_{i-1} \leftarrow 0]$, where $e[x \leftarrow f]$ is the substitution of f for x in e, and $a_i = t[x_{i-1} \leftarrow 1] - b_i$. The normalization system is then used to prove that $t - a_i x_{i-1} - b_i \equiv 0$, a much simpler proposition.
>
> Once we know a_i and b_i, recurrence (8) can be written in matrix form:
>
> $$\begin{pmatrix} x_i \\ 1 \end{pmatrix} = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ 1 \end{pmatrix},$$
>
> and hence is equivalent to the computation of matrix products, which are associative.

Another example is the family of homographic functions:

$$\mathbf{Hom}_{abcd}(x) = \frac{ax + b}{cx + d}$$

which is closed under function composition, It so happens, in fact, that if we associate to $\mathbf{Hom}_{abcd}$ the matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ then function composition for the **Hom** family is associated to the matrix product. A mathematical pattern matching routine can be devised for **Hom** provided our underlying computer algebra system is able to normalize rational fractions. It would be interesting to systematically explore closed families of functions, since each one is the basis of an efficient parallel algorithm. The study of [1] may be a step in that direction.

4.2.4 Recurrences on finite domains

Let us suppose that the variable x_i in (4) has a finite domain. It is clear that functions from a finite set to the same finite set are closed under composition. As a consequence, a recurrence on a finite domain is always a scan.

A function f on a finite domain $\{a_1, \ldots, a_n\}$ can be defined by a table of values $\{f(a_1), \ldots, f(a_n)\}$. Computing $f \circ g$ simply consists in the computation of $\{f(g(a_1)), \ldots f(g(a_n))\}$. This can be done from the tables of f and g, and is necessarily an associative operation, as it is a representation of function composition. Hence the algorithm for computing a scan (4) when x_i has a finite domain is:

1. Compute the tables associated to each f_i.
2. Combine these tables by any reduction scheme, obtaining the tables for the functions g_i of (6).
3. Compute the results of the scan in parallel by:

$$x_i = g_i(x_0).$$

It is interesting to compare the parallel and sequential scheme for complexity. Let us suppose that the domain of the variables (x_i) has n elements, that the recurrence has m steps, and that our parallel computer has P processors. Let us take the time for a table access as the unit. Let us suppose that the functions f_i are initially given as tables. In that case the sequential time is simply m. The elementary step of the scan is an algorithm for computing the composition of two tables `f1` and `f2` giving `f3`:

```
do i = 1,n
  f3(i) = f2(f1(i))
end do
```

which takes $3n$ table accesses. The total time for the reduction is thus $3n(\frac{m}{P} + \log_2 P)$. The last step takes $\frac{m}{P}$ times unit, for a total of $3n(\frac{m}{P} + \log_2 P) + \frac{m}{P}$. Supposing that m is large enough that we can neglect additional terms, we have to compare m to $\frac{(3n+1)m}{P}$. The conclusion is that the method is advantageous provided that the size of the finite domain is small compared to the number of processors.

4.2.5 Tactics

As the reader may have noticed, the methods we have presented are overlapping. The same Scan operator may be recognized by elementary pattern matching, or only after normalization, or as a computation on a finite domain. For example, a boolean recurrence can be recognized by pattern matching if its operator is $\wedge$ or $\vee$, or as a recurrence over a finite domain if not. Our proposal is first to distinguish cases according to the type of x_i. To each type is associated one special purpose normalisation system. Pattern matching is then applied to the result of the normalization. If this fails, and if it can be proved that x_i has a finite domain, then the general method of Sect. 4.2.4, which is less efficient, can be applied.

Consider the recurrence:

$$x_i = (x_{i-1} + 1)/3.$$

Suppose first that x_i is a so-called real (i.e., belongs to a subset of the rationals). Then, up to rounding errors, division distributes into addition, and the recurrence can be normalized, with the permission of the user, into:

$$x_i = 0.333 \ldots x_{i-1} + 0.333 \ldots,$$

which is linear, hence, a scan. If x_i is integral, then the above normalization is no longer valid, and the original recurrence is not a scan. Lastly, if the recurrence is slightly modified into:

$$x_i = [(x_{i-1} + 1)/3] \bmod N,$$

where N is a constant integer, then x_i belongs to the finite set $[0, N-1]$ and the recurrence is again a scan. This observation is usefull only if N is much smaller than the number of processors.

Note that scan detection can benefit from information obtained by a static analysis of the program. For instance, an expression which is apparently a second degree polynomial may be proved linear by a constant propagation which shows that the second degree coefficient is zero. A variable which is declared an integer may be found to have a finite domain by interval analysis. Conversely, the result of a scan detection may enable the computation of the DFG of an hitherto intractable part of a program, by exhibiting inductive variables or giving in closed form the values of the entries in an array which is used as a subscript. The organization of such iterative analyses is a very difficult problem.

5 Multi-directional Scans

A multi-directional scan has more than one direction vectors, other parameters being the same:

$$\mathrm{Scan}(\mathcal{D}, (e_1, \ldots, e_k), b, d, g)\ . \tag{9}$$

The direction vectors $e_1, \ldots, e_k$ span a linear subspace H and define a family of affine subspaces $H + \delta$ for an arbitrary translation vector δ. Formula (9) defines as many scans as there are affine subspaces whose intersection with $\mathcal{D}$ is not empty. b is the scan operator, and the scan order is lexicographic order on the coordinates of each point relative to its affine subspace. Points which cannot be reached from other points of $\mathcal{D}$ by a positive integral combination of direction vectors are given the initial values as defined by g. The reader may notice that this definition defaults back to the definition of a uni-directional scan when $k = 1$.

Our first observation is that multi-directional scans are more efficient than multiple uni-directional scans.

Take the example of the sum of the elements of a $n \times n$ matrix. With only uni-directional scans, one must compute n sequential scans operating on n data. With P processors, the run-time is of the order of $n(\frac{n}{P} + \log_2(P))$. With a multi-directional scan, a run-time of the order of $\frac{n^2}{P} + \log_2(P)$ is expected. In the best case (when $n = P$) the speed-up is about $1+\log_2(P)$.

In our system, multi-directional scans cannot be detected directly. They can be found using multistage elimination. When working at pseudo-depth p, a direction may be added to a scan if its initial values reference the clause c in which it lays and if the scan operator is the clause c itself. The second condition is that a new direction e_0 can be extracted from the definitions of the scan initial values. A new direction cannot be an arbitrary vector: some checks must be done on the definition of each initial value v_0 of the former Scan which are not initial values of the enhanced Scan. Namely, such a definition must be the application of the Scan operator to the data corresponding to v_0 and the data corresponding to the predecessor of v_0 according to the directions e_0 to e_k.

Practically, the problem of extracting a new direction may be solved using the PIP software [5], since testing the validity constraint is equivalent to the computation of a lexicographic maximum.

Let us consider again the summation of the elements of a matrix. The corresponding system is:

```
x[i,j] =
   case
     i,j | i=1, j=1          : a[i,j] ;
     i,j | 2<=i<=n, j=1      : x[i-1,n] + a[i,j] ;
     i,j | 1<=i<=n, 2<=j<=n : x[i,j-1] + a[i,j] ;
   esac ;
```

Pattern matching is applied for the detection of scans at pseudo-depth 1. A closed form is introduced for the clause `x.2`:

```
x[i,j] =
   case
     i,j | i=1, j=1          : a[i,j] ;
     i,j | 2<=i<=n, j=1      : x[i-1,n] + a[i,j] ;
     i,j | 1<=i<=n, 2<=j<=n :
          Scan( i',j' | 1<=i'<=n, 1<=j'<=n,
                ([0 1]), +, a[i',j'], x[i',j'] );
   esac ;
```

Removing inter-components references leads to the inclusion of the clauses `x.0` and `x.1` into the initial value of the Scan operator.

```
x[i,j] =
   case
     i,j | 1<=i<=n, 2<=j<=n :
          Scan( i',j' | 1<=i'<=n, 1<=j'<=n,
```

```
                ([0 1]), +, a[i',j'],
                 case
                    k,l | k=1, l=1     :
                         a[1,1] ;
                    k,l | 2<=k<=n, l=1 :
                         x[k-1,n] + a[k,l] ;
                 esac ; )
   esac ;
```

Since this system is already normalized for pseudo-depth 0, a pattern-matching can be applied. No new uni-directional scan can be detected. But we can try to add a direction to the previously detected scan. Two necessary conditions for scan enhancement are fulfilled: the scan *is* the clause expression, and the second clause of the initial value is referencing `x.0` at pseudo-depth 0 using the scan binary operation (here an addition). Now, a new direction e_0 must be extracted. The direction must verify that for k in $\{2, \ldots, n\}$ the point $\begin{pmatrix} k \\ 1 \end{pmatrix}$ has for predecessor in scan order defined by e_0 and e_1 the point $\begin{pmatrix} k-1 \\ n \end{pmatrix}$. A solution is the integer vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

The scan enhancement is successful, we obtain a two-directional scan:

```
x[i,j] =
   case
     i,j | 1<=i<=n, 2<=j<=n :
          Scan( i',j' | 1<=i'<=n, 1<=j'<=n,
                ([1 0] [0 1]), +,
                a[i',j'], a[1,1] ) ;
   esac ;
```

6 Conclusion

6.1 Implementation

Our scans detector prototype is written in C and Lisp. The most heavily used tools are written in C and can be accessed via an ASCII interface using a Lisp-like syntax. This interpreter combine the following tools:

- the PIP software for solving integer programming problems,
- the convex calculator from IRISA [18],
- a set of operations on systems of equations (mainly substitution).

There are two main modules in our current prototype, a module of system normalization at a given pseudo-depth and a module for detecting scans at a given pseudo-depth. The canonical way of using these modules is to apply normalization at the greater pseudo-depth and then to apply the detection module at

the same pseudo-depth. If no scan is found the process may be iterated with a lower pseudo-depth and so on. Most of the operations are executed by the C interpreter, the only major part coded in Lisp is the extraction of the binary operation and the associated data from the propagation function of recurrences. In the present version of the software, rules for detecting multi-directional scans have not been implemented yet; their implementation will lead to the creation of a third module.

Consider the following extract from a real world program:

```
DO i=1,n
  b(i)=a(i)
  a(i-1)=a(i-1)+b(i)
  a(i)=a(i)+b(i)
  a(i+1)=a(i+1)+b(i)
END DO
print *,(a(i),i=1,n)
```

The system found after the application of the normalization and detection module at pseudo-depth 0 is:

```
x[i] =
   case
      i | i=1  : a[1]+a[2] ;
      i | i>=2  :
         Scan(  i' | 0<=i'<=n , ( [1] ),
               +, a[i'+1], a[1] ) [i] ;
   esac ;

print
   case
      i | i=1  : a[1] + a[1] + x[1] ;
      i | 2<=i<=n-1  : x[i-1] + x[i-1] + x[i] ;
      i | i=n  : x[n-1] + x[n-1] ;
   esac ;
```

A scan hidden by some manipulations on the elements of the array `a` is detected. The results for some other examples (mostly from the Argonne benchmarks [3]) can be found in [17]. All of the 18 loops which are classified as reductions in the Argonne benchmarks can be solved by our software with one exception (the DFG analysis cannot be done because of the presence of a `goto` in the main loop).

6.2 Future Work

The present version of our software fulfill one of the aims we set ourselves when we began this research: to build a much more powerful scan detector than what is found in current parallelizers and vectorizers. We detect much more scans

than do others, and this is done with a very small knowledge base. Our detector is almost insensitive to variations in the syntax of the original loop nest. Last but not least, we are able to recognize scan on arrays and arrays of scans. We believe our software is the first one to implement this facility.

This work is obviously just the beginning on the road toward efficient implementations of scans and reductions. The first problem that suggests itself is how to use the results of our analyzer.These results are not an imperative program, but a set of recurrence equations embellished with `Scan` operators. We still have to convert them back to our object language, e.g. some sort of parallel or data-parallel Fortran. We gave preliminary solutions and experimental results in [16] but much more work is needed in that direction.

There are, however, other applications of scan detection besides parallelization. One of them is program checking (or reverse engineering). After scan detection — and possibly some pretty printing — a program is brought in a form which is much nearer to mathematical notations than the original. It should be easier to detect errors — e.g. a summation which is short one term — on the mathematical representation than on the imperative version.

There is another, possibly much more interesting application, *algorithm recognition*. Toward this aim, we need a complete set of operators on scans, and some way of organizing them in a semblance of a normalization algorithm, it being understood that a full normalization algorithm is probably impossible. We could then compare the result to a base of normal forms for standard algorithms, e.g. all those for which we have an efficient implementation in our library. We could then replace part of the original program by a call of the corresponding routine. If the user care to supply directives, we could even select a version which is well adapted to the task at hand, as for instance a vector or parallel version, or even a sparse version.

This proposal is not as farfetched as it seems. Consider, for example, the following code for matrix multiplication:

```
      do 1 i=1,n
        do 1 j=1,n
          s=0.
          do 2 k=1,n
2           s = s + a(i,k) * b(k,j)
1         c(i,j) = s
```

Our present scan detector translates this program into one recurrence equation:

```
c[i,j] = Scan(  i',j',k' | 1<=i'<=n, 1<=j'<=n, 0<=k'<=n ,
              ( [0 0 1] ), +, a[i',k'] * b[k',j'], 0 )[i,j,n] ;
```

Besides, it seems probable — but it has to be investigated — that most variants of this code (e.g. all those obtained by permuting the loops) will normalize to the same or to very similar equations. It thus seems that we are at the threshold of being able of recognizing simple algorithms from linear algebra.

To achieve this "semantic parallelization", a whole algebra must be build around the Scan operator; and we must find a way to translate its rules into a normalization algorithm. This will be the subject of future work.

References

[1] J. Aczél. *Lecture on Functional Equations and their Applications.* Academic Press, 1966.

[2] D. Callahan. Recognizing and parallelizing bounded recurrences. In U. Banerjee et al. (Eds.), editor, *Proc. of the Fourth International Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 266–282. Springer-Verlag, August 1991. LNCS 589.

[3] J. Dongarra D. Callahan and D. Levine. Vectorizing compilers : A test suite and results. *Proceedings of the first IEEE Supercomputing'88*, pages 98–105, November 1988.

[4] G.-R. Perrin E. Violard. Reduction in pei. In Springer Verlag, editor, *Proceeding of CONPAR'94 - VAPP VI (LNCS 854)*, pages 112–123, September 1994.

[5] Paul Feautrier. Projet vesta : Outil de calcul symbolique. In *6th Int Coll on Programming*, 1984. LNCS 167.

[6] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[7] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1), February 1991.

[8] Kenneth A. Iverson. *A Programming Language.* Jonh Wiley & Sons, New York, 1962.

[9] N. D. Jones and S. S. Muchnick. *Program Flow Analysis, Theory and Applications.* Prentice Hall, 1981.

[10] Pierre Jouvelot and Babak Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Procs. of the 3rd Int. Conf. on Supercomputing*, pages 186–194. ACM Press, 1989.

[11] Hanoch Levy and David W. Low. A contraction algorithm for finding small cycle cutsets. *Journal of Algorithms*, 9:470–493, 1988.

[12] Y.-I. Choo M. Chen and J. Li. Crystal: From functional description to efficient parallel code. In G. Fox, editor, *Proc. of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 417–433. ACM, New York, USA, 1988.

[13] Christophe Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, Université de Rennes I, December 1989.

[14] Bernard Philippe Michèle Raphalen. Précision numérique dans le cumul d'un nombre de termes. Technical Report Publication interne 253, Institut de recherche en informatique et systemes aleatoires (IRISA), April 1985.

[15] R. Pinter and S. Pinter. Program optimization and parallelization using idioms. In *ACM PoPL*, 1991.

[16] Patrice Quinton. *Automata networks in Computer Science*, chapter The systematic design of systolic arrays, pages 229–260. Manchester University Press, December 1987.

[17] S. Rajopadhye and M. Muddarangegowda. Parallel assignment, reduction and communication. In *SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, 1993.

[18] X. Redon. Détection et exploitation des récurrences dans les programmes scientifiques. In M. Cosnard et P. Fraigniaud L. Bougé, editor, *6eme rencontres francophones du parallélisme*, pages 81–85, June 1994.

[19] Xavier Redon. *Détection et exploitation des récurrences dans les programmes numériques en vue de leur parallélisation.* PhD thesis, Université P. et M. Curie, January 1995.

[20] D. Wilde. A library for doing polyhedral operations. Technical Report Internal Publication 785, IRISA, Rennes, France, Dec 1993. Also published as INRIA Research Report 2157.