Detection of Recurrences in Sequential Programs with Loops

Xavier Redon and Paul Feautrier

Laboratoire MASI,
Université de Versailles-St. Quentin,
45, Avenue des Etats-Unis,
78000 Versailles, France
e-mail : redon@masi.ibp.fr, feautrier@masi.ibp.fr

Abstract. To improve the performances of parallelizing compilers, one must detect recurrences in scientific programs and subject them to special parallelization methods. We present a method for detecting recurrences which is based on the analysis of Systems of Recurrence Equations. This method identifies recurrences on arrays, recurrences of arbitrary order and multi-equations recurrences. We explain how to associate a SRE to a restricted class of imperative programs. We present a normalization of such SRE that allows the detection of recurrences by simple inspection of equations. When detected, a recurrence may be replaced by a symbolic expression of its solution. To iterate the process can lead to the identification of multi-dimensional recurrences.

1 Introduction

One of the most important challenges in present day computer science is the efficient compilation of programs for the newly emerging massively parallel architectures. In contrast of the situation for the last generation of supercomputers, which were mostly vector processors with a moderate degree of parallelism, now every last ounce of parallelism must be extracted in order to feed several hundred of vector microprocessors. It may be shown that potential parallelism exists as soon as the execution order of some operations may change without any consequence on the final result of the program. This kind of commutativity property may have two origins. The first one depends only on the pattern of use of memory cells. It is subsumed by Bernstein's conditions ([2]), and has been extensively exploited by present day parallelizing compilers. The second kind depends on algebraic properties of the operators which appears in the computation. The name "semantic parallelization" for the exploitation of these opportunities for parallelism has been coined by P. Jouvelot.

Example 1. Consider for instance the computation of the sum $\sum_{i=1}^{n} x_i$, its usual implementation is

```
s=0.
DO i=1,n
   s=s+x(i)
END DO
```

An ordinary parallelizing [1] compiler will say that this loop is sequential, due to a loop-carried dependence on `s`. As a consequence, the computation will take a time of the order of n. However, since addition is associative and commutative [2] one may divide the summation into segments of length $\frac{n}{p}$ (where p is the number of processors) then add the partial sums in a time of the order of $\frac{n}{p} + \log_2 p$. If the number of processors is very large (of the order of n) one may compute s in time $O(\log_2 n)$. Most programming languages lack a notation for expressing computations like $\sum_{i=1}^{n} x_i$. The main exception is APL, in which the name *reduction* was introduced.

It is thus seen that extracting expressions like the above sum from loops is a very important task for a parallelizing compiler. This is usually done in two steps. The first one consists in recognizing *recurrences* from their sequential expression. For instance the recurrence associated to the previous example is [3]

$$\left[\begin{array}{r} s_0 = 0 \\ \forall i \in \mathbb{N}_n^*,\ s_i = s_{i-1} + x_i \end{array}\right.$$

This is a purely algebraic task; it becomes very difficult when the sequential program gets complicated, and especially when arrays are involved. The second step consists in examining the recurrences to see whether the operators have the required properties. This is a pattern recognition step, whose performance will depend mainly on the size of the pattern base. Hence the interest of reducing this size by normalization of the recurrences.

The paper is organized as follows. The next section is a review of recent works on the subject. In section 3, we describe the basis on which our solution will be built, namely the translation of a sequential program into a System of Linear Recurrence Equations. In section 4 we discuss the pattern-matching process and describe a normal form for SLRE. Section 5 describes the normalization process and give a necessary and sufficient condition for the existence of a normal form. Section 6 describes the final step of our analysis, that is multi-dimensional recurrences detection. The conclusion includes some informations on an implementation of the method. In the interest of conciseness, most proofs have been omitted. They may be found in full in [10].

2 State of the Art

Some papers on reductions detection have already been published ([5] and [9]). The first presents a method based on symbolic stores and the last a method based on the dependence graph.

---

[1] some commercial compilers will recognize this form as an idiom and compile it efficiently.

[2] if one neglects rounding errors.

[3] Let $\mathbb{N}_n^*$ denote the set of natural integers $\{1, \ldots, n\}$.

2.1 Method Based on Symbolic Stores

The first step of this method is the analysis, by symbolic evaluation, of the bodies of the innermost loops of the program. A pattern matching step is then applied to the symbolic stores to recognize the reductions. Loop nests can be processed by propagation of the solutions of recurrences.

The principal advantage of these methods is that a normalization of the program occurs during the computation of the symbolic stores. Thus a method based on symbolic stores is somewhat indifferent to variations in the implementation of the algorithm. As against this the fact that values are considered as symbolic items is a substantial disadvantage. Indeed such a method cannot fully handle arrays (e.g. $a(i)$ and $a(i'+1)$ are different symbols but may represent the same array cell if $i = i'+1$). Therefore some recurrences on arrays cannot be detected. Moreover the imprecision due to the symbolic analysis can lead to wrongly replace a piece of code by a recurrence computation. To avoid this problem one must use an heuristic method to select loop bodies on which it is safe to apply the reduction detection. Consequently more recurrences will be missed.

2.2 Method Based on the Dependence Graph

For a detection of recurrences based on the Dependence Graph (DG) , one needs to represent loops. [9] presents a way to build such a DG. First, loops must be unrolled so as to reach normal form. A loop is in normal form when an iteration of the new loop uses only arrays cells and scalars computed in the same iteration or in the previous one. When the normal form is reached, a generic DG of the loop body is build. Since the loop is in normal form, the union of the DG of the initial iteration with the DG of an intermediate iteration and with the DG of the final iteration gives a DG for the whole loop. Then a pattern-matching step is applied to the loop DG, and for each sub-graph that involve recurrence an appropriate algorithm is generated.

Such a method allows better detection (e.g. it detects some cross-recurrences). A disadvantage is that this method does not include any normalization. This normalization must be done by classical transformations (e.g. substitution of temporaries, scalar expansion, loop interchange, etc.). Therefore this step can hardly be done without human control. Moreover, since the pattern-matching is applied to the whole graph, the time complexity increases quickly with the program size. Another limitation is that normal form for a loop exists only for the case of uniform dependences.

3 A New Method Based on SLRE Analysis

3.1 Motivation

The methods presented in the previous section all have some weak points. These are due to the lack of precision of the program representation (symbolic stores) or to the absence of normalization. An intermediate representation of programs

by a System of Linear Recurrence Equations (SLRE) seems to be well adapted to the detection of recurrences. Indeed, a program written in an imperative language (e.g. FORTRAN) can be translated into a SLRE under certain usual assumptions (i.e. the Dataflow Graph of the program is computable, see below). Moreover we are able to normalize SLRE with a powerful tool: the forward substitution.

3.2 The Dataflow Graph

In order to translate the source program into SLRE, we use the algorithm described in [4] for computing the Dataflow Graph (DFG) of the source program. The DFG deals with *operations*. An operation is a pair build with an instruction and an occurrence of the iteration vector of the instruction. $(i, \mathbf{v})$ being an operation, the DFG gives, for each reference to a scalar or array element in this operation, the source operation (that is the operation in which the scalar or the array element is computed). When the DFG is build, it is easy to translate the source program into a Single Assignment program by renaming and expansion of variables.

Dataflow Analysis has been implemented, along the lines of [4] as part of the PAF project [4]. The present software for reduction detection uses the result of this analysis. It has the same range of application as the Dataflow Analysis module: static control programs with linear subscripts (see [4] for more informations on that point).

3.3 Representation for SLRE

Our detection is based on SLRE, thus we must give a precise definition of such systems and find out a way to represent them. Some languages (e.g. the Alpha language [8]) have already been designed to describe such equations. Alpha variables are spatial variables, i.e. triplets $< D, \phi, V >$ where the function ϕ associates to each point of the convex domain D a value in V. However, the Alpha notation has been designed more for automatic processing than for ease of use. Thus we prefer to deal with equations in the usual mathematical way.

We will work with LRE equations of the form

$$z \in D_i,\ U_i(z) = f_i(U_1(I_1(z)), \ldots, U_n(I_n(z)))\ ,$$

assuming that $(U_i)_{i \in \mathbb{N}_n^*}$ is the family of variables of the system. Moreover, we assume that D_i is a bounded convex. The I_i are linear subscripts functions and the functions f_i are conditional functions such that

$$f_i(x) = \begin{cases} Exp_i^1(x) & \text{if } x \in D_i^1 \\ \quad \vdots & \\ Exp_i^m(x) & \text{if } x \in D_i^m \end{cases}$$

(The Exp_i^j are classical mathematical expressions and the D_i^j are bounded convexes). We say that f_i is an m clauses expression.

[4] PAF is a French acronym standing for Automatic Parallelization of FORTRAN.

3.4 Overview of the Method

Like every method for the detection of recurrences, our method consists of three parts. The first part is the conversion of the source program into SLRE. The second part is the normalization of the SLRE and the last part is the application of a pattern-matching on the SLRE. We want the pattern-matching phase, which is quite time consuming, to be as efficient as possible. Therefore, we apply the pattern-matching only on one equation at a time. But we want an efficient method too, thus the normalization part try to break multi-equations systems into several systems with only one equation. Note that we are working with multi-dimensional arrays and loops. We begin to detect the recurrences relative to the highest dimension (i.e. the recurrences relative to the innermost loops). A detected recurrence is replaced by its symbolic solution and the analysis is applied to the next dimension. This allows the detection of multi-dimensional recurrences, that is recurrences relative to several nested loops.

4 Validity of Detection by Pattern-Matching

We use pattern-matching for the detection of recurrences. This section shows on which conditions this must be done to be valid. First we give some definitions about equations systems and about systems graphs.

4.1 Definitions

In this paper, we use definitions and results from graph theory. Our reference is [1]. First, we must precise the notion of equations system.

Definition 1 equations system. An equations system S is a set of LRE equations [5] such that

$$\forall (e, e') \in S^2,\ v_e = v_{e'} \Rightarrow D_e = D_{e'} \wedge \forall z \in D_e,\ Exp_e(z) = Exp_{e'}(z)\ .$$

To point out the dependences between equations we build the system graph (this graph is a sub-graph of the Dependence Graph of the original program).

Definition 2 system graph. Let S be an equations system, the graph of S (denoted by $\mathcal{G}_S$) is the graph whose vertices are the equations of S and whose edges are the couples (e, e') such that the variable $v_{e'}$ appears in the expression Exp_e.

We need to introduce the notion of depth into our graphs.

Definition 3 system p-graph. Let S be an equations system, the p-graph of S (denoted by $\mathcal{G}_S^p$) is a sub-graph of $\mathcal{G}_S$ such that (e, e') is an edge of $\mathcal{G}_S^p$ if and only if there exists $z \in D_e$ and $z' \in D_{e'}$ such that $v_{e'}(z')$ is used in expression $Exp_e(z)$ and p is the largest integer verifying $z[1..p] = z'[1..p]$.

[5] We will assume that an LRE equation e has the form $\forall z \in D_e, v_e(z) = Exp_e(z)$.

4.2 Conditions of Validity

A naive method to detect recurrences in a SLRE is to scan all the clauses of the equations and compare them with a general pattern. But this syntactic criterion does not suffice to characterize a recurrence. We need two additional conditions. First, the values needed for the computation of an element of the sequence must belong to the clause, except for the initial values of the recurrence. Second, the equation must not be part of a multi-equations recurrence. Indeed, in this case, a reference to an other equation can hide an auto-reference. A sufficient condition is that, if we are detecting recurrences in respect to dimension l, the graph $\mathcal{G}_S^{l-1}$ does not have any cycle (except loops) which include the equation. This condition presents the advantage of being easily verified.

Now we can present a two level characterization of a recurrence. At the equation level we must find the recurrent clauses:

Definition 4 recurrent clause. Let S be an equations system, e an equation of S and c a clause of Exp_e. The clause c is recurrent with order o and step k for the dimension p if and only if c matches the following pattern

$$F(v_e(\phi_{p,k}(z)), \ldots, v_e(\phi_{p,o.k}(z))$$
$$\text{if } z \in D_e^c = \{a_1 \leq z_1 \leq b_1, \ldots, a_m(z_1, \ldots, z_{m-1}) \leq z_m \leq b_m(z_1, \ldots, z_{m-1})\}$$

where the vectors $\phi_{p,k'}(z)$ are of the form

$$\phi_{p,k'}(z) = (z_1, \ldots, z_p, z_p - k', x_1^{k'}(z), \ldots, x_{m-p}^{k'}(z))$$

and if and only if the images of D_e^c by the $\phi_{p,k'}$ auto-reference functions are included in D_e^c except for the initials values of the recurrence.

At the system level we must verify that equation level detection is valid:

Proposition 5 validity of pattern-matching. *Let S be an ordered system, e an equation of S and c a clause of Exp_e. If c is a recurrent clause with order o, step k and propagation function F^c for dimension l and if there is no cycle in $\mathcal{G}_S^{l-1}$ with length greater than or equal to 2 then c can be computed by a recurrence with order o, step k and propagation function F^c.*

5 System Normalization

The aim of systems normalization is to allow pattern-matching to recognize a maximum of recurrences. So, the condition of validity from the previous section must be fulfilled by a maximum of clauses. Therefore, we want to transform each system into a reduced one (i.e. a system whose graph does not have any cycle of length greater than 1). To be sure that the transformed system is equivalent to the original one we will use only *forward substitution* as a transformation tool.

First, we define the notions of substitution and transformation. Then we present the conditions on which a system can be transformed into a reduced one.

5.1 Definitions

Let us formalize the usual process of substitution in equations systems. Let e and $(e_i)_{i\in\mathbb{N}_n^*}$ be LRE equations. We denote by $e \odot \{e_1, \ldots, e_n\}$ the equation e in which all references to the variables $(v_{e_i})_{i\in\mathbb{N}_n^*}$ are simultaneously replaced by their respective expressions (i.e. the $(Exp_{e_i})_{i\in\mathbb{N}_n^*}$).

Our elementary transformation is the action of replacing one or more equations in a system S by new equations of the form $e \odot S'$ where e is the original equation and S' is a sub-system of S.

5.2 Criterion for System Reduction

The aim of this sub-section is to find the conditions on which a system is reducible (i.e. there exists a sequence of transformations such that the resulting system is a reduced system).

Proposition6 reduction of a strongly connected system. *Let S be a strongly connected system* [6]*. Then S is reducible if and only if the cycles of $\mathcal{G}_S$ have a common vertex.*

In fact, when detecting recurrences with respect to the dimension l, the fulfillment of the validity condition only requires the reduction of the $(l-1)$-graph of the system. Therefore we just need to verify that the cycles of $\mathcal{G}_S^{l-1}$ have a common vertex. This criterion stands only for strongly connected systems. In the case of an arbitrary equations system one must try to reduce the system strong components. If each component is reducible then the system is reducible.

5.3 An Algorithm for Normalization

This section presents an efficient algorithm to reduce a system.

Algorithm7 algorithm A. *let S be a strongly connected system.*
Initialization:
$S_0 = C_0 = S$
Propagation:
If there exists, In set C_i, an e'_i only referenced by e_i
Then / Replace e_i by $e_i \odot e'_i$ and remove e'_i from C_i */*
$S_{i+1} = (S_i - \{e_i\}) \cup \{e_i \odot e'_i\}$
$C_{i+1} = (C_i - \{e_i, e'_i\}) \cup \{e_i \odot e'_i\}$
Else / End of normalization */*
$S_{i+1} = S_i$
$C_{i+1} = C_i$
EndIf.
We denote by i_{final} the smallest i such as $S_{i+1} = S_i$. It is of no use to compute the sequence $((S_i, C_i))$ beyond i_{final}.

[6] A strongly connected system is a system whose graph is strongly connected. In the same way a strong component of a system is the vertex set of a strong component of the system graph.

For each strongly connected system S, this algorithm builds a system $S_{i_{\text{final}}}$. If S is reducible, $S_{i_{\text{final}}}$ is a reduced system. Moreover the complexity of $\mathcal{A}$ is linear in relation to the number of vertices of S. All these affirmations are proved in [10].

6 Symbolic Solutions for Recurrences

The major difficulty when detecting recurrences in a SLRE system is to deal with multi-dimensional sequences. We must detect recurrences relative to all dimensions. Moreover some recurrences can be relative to two or more dimensions. The first step to solve these problems is to detect recurrences from inside outward. The second step is to replace the clauses which represent a recurrence by its symbolic solution.

We can draw an analogy with differential equations: an equation $dy = f(x)dx$ may not have an algebraic solution but we always can say that y is equal to $y = \int f(x)dx$. The equation is not solved but we can work with y, for instance replace it by the integral in an expression. The symbol used to write a symbolic solution of a recurrence (i.e. the counterpart of the integral symbol) is called the recurrence operator.

We can summarize the algorithm of reduction detection by the following.

Algorithm8. *Let S be a SLRE system extracted from an imperative program and let D be the maximal dimension of equation domains.*
For p=$D-1$ Downto 0 Do
p-reduce the system S.
Recognize recurrences relative to the dimension $p+1$ and replace them by a symbolic solution.
Done
In the final system, compose recurrence operators to obtain multi-dimensional recurrences.

6.1 Recurrence Operator

Definition9 recurrence operator. An expression build with the recurrence operator is of the following form.

$$\begin{aligned}\text{Recur}(\,(o,k), \{(l, \lambda i_1 \ldots i_{l-1}.\alpha, \lambda i_1 \ldots i_{l-1}.\beta)\}, \\ \lambda i_1 \ldots i_l x_1 \ldots x_o.f, (\lambda i_1 \ldots i_l.g_s)_{s\in[1,o]}) \ .\end{aligned} \tag{1}$$

Let us give the meaning of the different terms: o is the recurrence order, k is the recurrence step (see section 4.2). The recurrence is relative to dimension l, must iterate between the lower bound α and the upper bound β, its propagation function is f and the initial values are the $(g_s)_{s\in\mathbb{N}_o^*}$.

It is easy to rewrite a recurrent clause with the recurrence operator.

Example 2. For instance, the symbolic solution of the Fibonacci sequence

$$\left[\begin{array}{rl} & u_0 = 1 \\ & u_1 = 1 \\ \forall i \in \mathbb{N} - \{0,1\}, & u_i = u_{i-1} + u_{i-2} \end{array}\right.$$

is

$$\left[\begin{array}{rl} & u_0 = 1 \\ & u_1 = 1 \\ \forall i \in \mathbb{N} - \{0,1\}, & u_i = \mathrm{Recur}((2,1), \{(1,2,\infty)\}, \lambda i x y.x + y, (\lambda i.1, \lambda i.1))(i) \end{array}\right.$$

Example 3. Let us process the following program to show how our method handle uni-dimensional recurrences.

```
x(0)=0                          (Ins1)
DO i=1,2*n
 save(i)=x(2*n-i+1)             (Ins2)
 x(i)=x(i-1)+save(i)            (Ins3)
END DO
```

The corresponding system is

$$\left[\begin{array}{l} \forall i \in \mathbb{N}^*_{2n},\ \mathrm{Ins2}_i = \left\{\begin{array}{ll} x_{2n-i+1} & \text{if } i \geq 1 \wedge i \leq n \\ \mathrm{Ins3}_{2n-i+1} & \text{if } i \geq n+1 \wedge i \leq 2n \end{array}\right. \\ \forall i \in \mathbb{N}^*_{2n},\ \mathrm{Ins3}_i = \left\{\begin{array}{ll} \mathrm{Ins2}_1 & \text{if } i = 1 \\ \mathrm{Ins3}_{i-1} + \mathrm{Ins2}_i & \text{if } i \geq 2 \wedge i \leq 2n \end{array}\right. \end{array}\right.$$

(note that replacing the array reference `save(i)` by a scalar reference to `save` in instructions Ins2 and Ins3 would lead to the same system). The 0-graph of this system is not reduced, so the system must be normalized. Let us assume that the algorithm $\mathcal{A}$ choose to replace Ins2 by its value in Ins3 expression. The new system (Ins2 become useless and is removed) is

$$\left[\begin{array}{l} \forall i \in \mathbb{N}^*_{2n},\ \mathrm{Ins3}_i = \left\{\begin{array}{ll} x_{2n} & \text{if } i = 1 \\ \mathrm{Ins3}_{i-1} + x_{2n-i+1} & \text{if } i \geq 2 \wedge i \leq 2n \\ \mathrm{Ins3}_{i-1} + \mathrm{Ins3}_{2n-i+1} & \text{if } i \geq n+1 \wedge i \leq 2n \end{array}\right. \end{array}\right.$$

The final system (after recurrence detection) is

$$\left[\begin{array}{l} \forall i \in \mathbb{N}^*_{2n}, \mathrm{Ins3}_i = \left\{\begin{array}{l} \mathrm{Recur}(\,(1,1), \{(1,1,n)\}, \\ \qquad \lambda i_1 y.y + x_{2n-i_1+1}, (\lambda i_1.0))(i) \\ \text{if } i \geq 1 \wedge i \leq n \\ \mathrm{Recur}(\,(1,1), \{(1,n+1,2n)\}, \\ \qquad \lambda i_1 y.y + \mathrm{Ins3}_{2n-i_1+1}, (\lambda i_1.\mathrm{Ins3}_{i_1-1}))(i) \\ \text{if } i \geq n+1 \wedge i \leq 2n \end{array}\right. \end{array}\right.$$

Applying some algebraic transformations on the final system give us the following result:

$$\mathrm{Ins3}_n = \sum_{i=1}^{n} (n - i + 2) x_{2n-i+1} \ .$$

Hence the original program may be useful to compute a discrete random variable expectation. Moreover this sequential program is efficient since no multiplication is used. Note that classical methods do not handle this example. Indeed, the dependence $i \rightarrow 2n - i + 1$ prevent loop normalization as presented in [9] and cannot be exploited by symbolic analysis.

The recurrence operator is designed to allow substitutions, but some precautions must be respected. We distinguish two kinds of substitution in presence of recurrence operators.

The first kind is substitution *by* a recurrence operator. Since an expression build with such an operator is independent of the domain of its clause, this expression can be moved anywhere. Thus substitution by a recurrence operator is always valid. But since detection of recurrences is done in order to reduce computation time, we must not duplicate the symbolic solution of a recurrence. Therefore this kind of substitution will be allowed only if the symbolic solution is referenced once. As a result we will not be able to reduce some reducible systems since some substitutions are forbidden.

The second kind of substitution is substitution *into* a recurrence operator. In an expression of the form $\mathrm{Recur}(a, b, c, d)$ the only terms in which doing substitutions make sense are c and d. A substitution in term c may lead to break the expression into severals symbolic solutions. There is no advantage in doing that, because we must re-compute the initial terms and because the new expressions must be computed sequentially, therefore the time complexity increases. But substitutions in term d are valid and useful. They do not involve modification in the other terms and they are necessary for multi-dimensional recurrences detection.

6.2 Multi-Dimensional Recurrences

As shown in algorithm (8) multi-dimensional recurrences are build in a final stage of recurrence operators composition. We first extend the definition of the recurrence operator to deal with multi-dimensional recurrence. Then we prove that, on certain conditions, two interlocked recurrence operators are equivalent to a recurrence operator with higher dimension.

Definition10 extension of the recurrence operator. An expression build with a recurrence operator can also have the following form (the sequence of natural numbers $(l_s)_{s \in \mathbb{N}_n^*}$ is a strictly increasing one):

$$\begin{aligned} &\mathrm{Recur}(\,(1,1),\\ &\qquad \{\,(l_1, \lambda i_1 \ldots i_{l_1-1}.\alpha_1, \lambda i_1 \ldots i_{l_1-1}.\beta_1), \ldots,\\ &\qquad\;\; (l_n, \lambda i_1 \ldots i_{l_n-1}.\alpha_n, \lambda i_1 \ldots i_{l_n-1}.\beta_n)\},\\ &\qquad \lambda i_1 \ldots i_{l_n} x.f, (\lambda i_1 \ldots i_{l_n}.g))\ . \end{aligned} \tag{2}$$

All the remarks of the previous sub-section about the incidence of the recurrence operator on equations substitution remain true with this new definition. Now we give the rule of composition for recurrence operators.

Proposition11 composition of recurrence operators. *Let R be a valid expression which have the following form*

$$\begin{aligned} R = \mathrm{Recur}(&(1,1),\{(l_1,\alpha_1,\beta_1)\},\\ &\lambda i_1 \ldots i_{l_1} x.\lambda j_1 \ldots j_\omega.\mathrm{Recur}(\,(1,1),\{(l_2,\alpha_2,\beta_2),\ldots,(l_n,\alpha_n,\beta_n)\},\\ &\qquad\qquad f,(\lambda k_1 \ldots k_{l_n}.x\ \Phi'(k_1 \ldots k_{l_n}))\\ &\qquad\qquad)\ i_1 \ldots i_{l_1}\ \Phi(i_1,\ldots,i_{l_1},j_1,\ldots,j_\omega),\\ &(\lambda i_1 \ldots i_{l_1}.\lambda j_1 \ldots j_\omega.(h\ i_1 \ldots i_{l_1}))) \ , \end{aligned}$$

where Φ is a map from $\mathbb{N}^{l_1+\omega}$ to $\mathbb{N}^{l_n-l_1}$ and Φ' a map from $\mathbb{N}^{l_n}$ to $\mathbb{N}^\omega$. If the expression f does not contain any symbol x and if the following condition is fulfilled

$$\begin{gathered} \forall I \in \mathbb{N}^{l_n},\ I_{l_2} = (\alpha_2\ I_1 \ldots I_{l_2-1}) - 1\\ \Phi(I_1 \ldots I_{l_1-1}(I_{l_1}-1),\Phi'(I)) = max^{l_2}(I_1,\ldots,I_{l_1-1},I_{l_1}-1,I_{l_1+1},\ldots,I_{l_n}) \ , \end{gathered}$$

where max^{l_r} is defined by recurrence: $\forall z \in \mathbb{N}^{l_n}$, $\forall l \in \mathbb{N}^*_{l_n}$,

$$max^{l_r}(z)_l = \begin{cases} \beta_l\ max^{l_r}(z)_1 \ldots max^{l_r}(z)_{l-1} & \text{if } l \in \{l_r,\ldots,l_n\} \\ z_l & \text{otherwise} \end{cases}$$

then R is equivalent to

$$\begin{aligned} \lambda i_1 \ldots i_{l_1+\omega}.\mathrm{Recur}(&(1,1),\{(l_1,\alpha_1,\beta_1),\ldots,(l_n,\alpha_n,\beta_n)\},\\ &f,(h))\ i_1 \ldots i_{l_1}\Phi(i_1,\ldots,i_{l_1+\omega}) \ . \end{aligned}$$

Example 4. To illustrate the detection of multi-dimensional recurrences let us process the following program

```
s=0                          (Ins1)
DO i=1,n
 DO j=1,m
  s=s+a(i,j)                 (Ins2)
 END DO
END DO
```

First we compute the system of the program

$$\left[\forall (i,j) \in \mathbb{N}^*_n \times \mathbb{N}^*_m,\ \mathrm{Ins2}_{i,j} = \begin{cases} \mathrm{Ins2}_{i,j-1} + a_{i,j} & \text{if } j > 1 \\ \mathrm{Ins2}_{i-1,m} + a_{i,j} & \text{if } j = 1 \wedge i > 1 \\ 0 + a_{i,j} & \text{if } j = 1 \wedge i = 1 \end{cases}\right.$$

The 1-graph of the system is reduced, that allows us to detect recurrences relative to the second dimension. The system becomes:

$$\begin{aligned} &\forall (i,j) \in \mathbb{N}^*_n \times \mathbb{N}^*_m,\\ &\mathrm{Ins2}_{i,j} = \begin{cases} \mathrm{Recur}((1,1),\{(2,2,m)\},\lambda i_1 i_2 x.x + a_{i_1,i_2},(\lambda i_1 i_2.\mathrm{Ins2}_{i_1,i_2}))(i,j) & \\ \qquad\qquad \text{if } j > 1 & \\ \mathrm{Ins2}_{i-1,m} + a_{i,j} & \text{if } j = 1 \wedge i > 1 \\ 0 + a_{i,j} & \text{if } j = 1 \wedge i = 1 \end{cases} \end{aligned}$$

Then, we replace s_{i_1,i_2} by its value in the first clause. Thus the system is now:

$$\forall (i,j) \in \mathbb{N}_n^* \times \mathbb{N}_m^*,$$
$$\mathrm{Ins2}_{i,j} = \begin{cases} \mathrm{Recur}((1,1),\{(2,1,m)\},\lambda i_1 i_2 x.x + a_{i_1,i_2},(\lambda i_1 i_2.\mathrm{Ins2}_{i_1-1,m}))(i,j) \\ \text{ if } j > 1 \wedge i > 1 \\ \mathrm{Recur}((1,1),\{(2,1,m)\},\lambda i_1 i_2 x.x + a_{i_1,i_2},(\lambda i_1 i_2.0))(i,j) \\ \text{ if } j > 1 \wedge i = 1 \end{cases}$$

Since the 0-graph of this new system is reduced, we can detect recurrences relative to the first dimension:

$$\forall (i,j) \in \mathbb{N}_n^* \times \mathbb{N}_m^*,$$
$$\mathrm{Ins2}_{i,j} = \begin{cases} \mathrm{Recur}(\ (1,1),\{(1,1,n)\}, \\ \qquad \lambda j_1 y.\lambda j_2.\mathrm{Recur}(\ (1,1),\{(2,1,m)\}, \\ \qquad\qquad \lambda i_1 i_2 x.x + a_{i_1,i_2}, \\ \qquad\qquad (\lambda i_1 i_2.y(m)))(j_1,j_2), \\ \qquad (\lambda j_1 j_2.0))(i,j) \\ \text{if } j > 1 \wedge i > 1 \end{cases}$$

In such a system the composition of recurrence operators is valid. The final system is:

$$\forall (i,j) \in \mathbb{N}_n^* \times \mathbb{N}_m^*,\ \mathrm{Ins2}_{i,j} = \begin{cases} \mathrm{Recur}(\ (1,1),\{(1,2,n),(2,1,m)\}, \\ \qquad \lambda i_1 i_2 x.x + a_{i_1,i_2}, \\ \qquad (\lambda i_1.0))(i,j) \\ \text{if } j > 1 \wedge i > 1 \end{cases}$$

6.3 Comparison to Other Recurrence Operators

Some other recurrence operators already exist, namely the reduction operator in the Alpha language (see [6]) and the scan primitives also known as parallel prefix operations (see [3]). However we have introduced our own recurrence operator for the following motives. The Alpha operator is an operator on un-ordered set of values, which is thus restricted to reduction by associative and commutative operators. It only gives the final result of the reduction, while we need the partial results since they can be used in the original program.

Example 5. For instance the following expression build with the Alpha operator

```
red(+, (i,j -> ), {i,j | 1<=i<=n; 1<=j<=m} : a)
```

which computes the sum $\sum_{i=1}^{n}\sum_{j=1}^{m} a_{i,j}$ can be rewritten with the recurrence operator:

$$\mathrm{Recur}((1,1),\{(1,1,n),(2,1,m)\},\lambda ijx.x + a_{i,j},(\lambda ij.0))(n,m)$$

But the set of values

$$(\mathrm{Recur}((1,1),\{(1,1,n),(2,1,m)\},\lambda ijx.x + a_{i,j},(\lambda ij.0))(i,j))_{(i,j)\in\mathbb{N}_n^*\times\mathbb{N}_m^*}$$

cannot be expressed with the `red` operator.

The scan primitives are more adapted since they use ordered sets and compute all the terms of the recurrence. Indeed the one-dimensional form of our recurrence operator and the scan primitives are very similar.

Example 6. The expressions

$$\forall i \in \mathbb{N}_n^*, \mathrm{Recur}((1,1), \{(1,1,n)\}, \lambda i x.x + a_i, (\lambda i.0))(i)$$

and

$$scan(+, [a_1, \ldots, a_n])$$

compute the same vector $[s_1, \ldots, s_n]$ with $s_i = \sum_{k=1}^{i} a_k$.

But scan primitives are designed to describe one-dimensional recurrences. It is possible, by a change of variables, to transform any multi-dimensional recurrence into a one-dimensional one. However, when doing this, the subscripts functions become non-linear and the difficulty of system analysis increases.

7 Conclusion

In summary, our method of recurrences detection, when compared with other methods, presents the following advantages: our method is based on the DFG structure which allows us to fully handle arrays. Moreover, the representation of programs as equations systems give us a way to perform a strong normalization. As a consequence the detection is not sensitive to the algorithm implementation. Lastly the introduction of the recurrence operator allows us to detect multi-dimensional recurrences.

Note that conditionals can be easily handled by our method: the structural ones (i.e. conditionals whose predicate is a positive form, linear in the loop counters and parameters of the program) are inserted in the DFG structure. The non structural conditionals are transformed into guarded instructions.

We have realized an implementation of this method in Lisp (the size of this implementation is about 5000 lines). The program is mostly a symbolic manipulation of conditionals equations. These equations are defined on convex domains. As a consequence the forward substitutions leads us to deal with convex intersections and convex simplifications. The easiest way to simplify a convex is to use an algorithm for computing its vertices, like Chernikova's algorithm. We would like to thank H. Le Verge and D. Wilde for allowing us to use the particular implementation they developed at IRISA ([7]).

Due to the effectiveness of this algorithm the final systems (after recurrences detection) have a reasonable size (less than ten clauses per equations for small examples). Moreover the final systems are simplified by the elimination of useless equations. The execution time is function of the initial system complexity. Thus sample programs with classic uni-dimensional recurrences are processed quickly (a few seconds on a low end workstation). When composition of symbolic solutions of recurrences is necessary the execution time increases. Therefore a program computing a double sum needs 30s to be analyzed and we need 60s

to process a triple sum program. The decomposition of the system into strong components allows us to deal with medium sized programs. But a real size program should be first analyzed by a front end program that finds out the portions of code where recurrences have to be detected.

The directions for future work are the following: since special recurrences can be implemented more efficiently than others in present day super-computers (i.e. reductions), we must point them out. Thus a dedicated pattern-matching phase must be developed. Moreover, in order to use the detected recurrences for parallel program construction, we plan to compute a schedule for the generated system (where recurrences are detected). Some adaptations to existing schedulers are needed since our symbolic solutions of recurrences may use unbounded fan-in operations.

References

1. C. Berge. *Graphes*. Gauthier-Villars, 1987.
2. A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on El. Computers*, EC-15, 1966.
3. G.E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1539, 1989.
4. Paul Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.
5. 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.
6. H. Leverge. Reduction operators in alpha. In D. Etiemble and J.-C. Syre, editors, *Lecture notes in Computer Science No 605*, pages 397–411, 1992.
7. Hervé Leverge. A note on chernikova's algorithm. Technical Report 1992, INRIA, May 1992. Référence à vérifier.
8. Christophe Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, Université de Rennes I, December 1989.
9. Shlomit S. Pinter and Ron Y. Pinter. Program optimization and parallelization using idioms. In *POPL'91*, 1991. to appear.
10. X. Redon. Détection des réductions. Technical Report MASI 92-52, Institut Blaise Pascal, September 1992.

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