



Toward an Automatic Parallelization of Sparse Matrix Computations

Roxane Adle, Marc Aiguier, Franck Delaplace,
LaMI, CNRS UMR 8042,
Université d'Évry Val d'Essonne,
523 Place des Terrasses,
91025 Évry, France
Tel : +33 (0)1 60 87 39 00
Fax : +33 (0)1 60 87 37 89
www : www.lami.univ-evry.fr
email: {adle,aiguier,delapla}@lami.univ-evry.fr

Version: ACCEPTED

In this paper, we propose a generic method of automatic parallelization for sparse matrix computation. This method is based on both a refinement of the data-dependence test proposed by A. Bernstein and an inspector-executor scheme which is specialized to each input program of the compiler. This analysis mixes compilation process and run-time process.

The sparsity of underlying data-structure determines a specific parallelism which increases the degree of parallelism of an algorithm. Such a source of parallelism had already been applied to many numerical algorithms such as the usual Cholesky factorization or LU-decomposition algorithms considered as the gold standards of parallelization based on sparsity. The standard automatic parallelization method cannot tackle such source of parallelism because it is based on the value of cells arrays and not merely on the memory addressing function.

Addressing the automatization of this parallelism requires to develop a mixed compile-time and runtime approach integrated in a inspector-executor process. The compilation step provides a dedicated inspector devoted to the analyzed program. The inspector computes the dependence graph at runtime which allows a dynamic parallelization of the execution.

As expressed just before, the generic scheme developed in this paper follows the design principles which have been applied, but at each time in an *ad-hoc* way, to many sparse parallelization of numerical algorithms such as Cholesky algorithm. As far as we know, no general formal framework has been proposed to automate such a method of sparse parallelization. In this paper, we propose a generic framework of sparse parallelization (i.e. numerical program independent) which can be applied to any numerical programs satisfying the usual syntactic constraints of parallelization.

Key Words: Parallelization, Dependence Analysis, Sparse Matrix Compiler

1. INTRODUCTION

Numerical scientific applications such as fluid dynamics or mechanical structure computation often use sparse matrices [9]. A matrix is sparse if many of its coefficients are zero and if there is an advantage in exploiting its zeros. The exploitation of its zeros leads to the use of dedicated sparse storage format which

involves complex algorithms, sophisticated data-structure and irregular memory references. Hence, parallel programs dealing with sparse matrices are recognized to be error-prone, hard to debug and difficult to maintain. Consequently, hiding sparsity to programmers eases the programmers' task and reduces the development time. One possibility is to consider programs written for dense storage format and automatically transforming them (i.e. restructuring) into programs which deal with sparse storage formats. This enables compiler optimizations such as parallelization before the restructuration phase. Moreover, considering sparsity during compiler's optimizations provides a new source to improve the parallelization which cannot be applied when the format is only considered to be dense. This is due to the algebraic property of zero to be absorbing in usual algebraic structures used in linear algebra (i.e. \mathbb{R} , \mathbb{Q} , \mathbb{Z} , etc.). This allows the dependence removal at run-time. Therefore, sparse matrix programming inherently contains more parallelism than programs written for dense storage formats.

This had already been observed and applied to the Sparse Cholesky Factorization method [13, 12]. In this paper, we propose to generalize these methods to any sequential numerical program satisfying the usual syntactic constraints of parallelization [10]. Dealing with sparse structures leads to extending the usual definition of dependencies to integrate these properties.

The parallelization is then divided into a compile phase and an execution phase. At compile time and at execution time, an analysis is performed on program texts and on sparse matrix structure, respectively. The parallelization follows an inspector-executor scheme, that is:

- the inspector will compute the dependence graph and schedule the tasks according to this graph;
- the executor will perform the parallel numerical computation.

Usually, the inspector consists of a separate module which performs a dynamic analysis. In the sparse parallelization, the program of the inspector is generated by the compiler according to a generic framework. Therefore, it is specialized to the analyzed program. In order to compute the dependence graph at runtime, the inspector will determine the fill-in introduced during the execution because the generation of the dependence graph depends on all nonzero locations. Figure 1 pictures the different steps which are performed at run-time by the compiler. The different inspector steps of Figure 1 briefly describe the following treatments:

- the symbolic fill determines the entries in the matrix which corresponds to nonzero elements during the execution. The filling is not merely used to accurately determine the required memory size for a given sparse matrix, but also to compute the dependence graph.
- The computation of the dependence graph, so-called *sparse dependence graph* (SDG), is based on an extension of data dependence whose conditions were given by A. Bernstein [5]. At compile-time, a dedicated program is generated to compute such a graph.
- The scheduling phase dynamically schedules program tasks. Here, tasks correspond to a restructured program operating on sparse data structures.

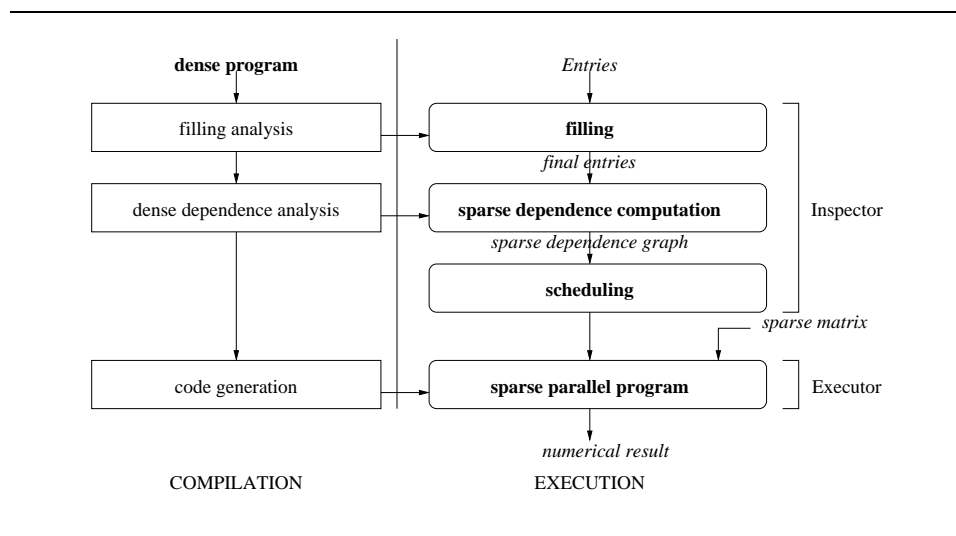


Figure 1 Overview of steps

Inspector-Executor methods induce an overhead which must be minimized. Consequently, we will favor parallel methods for inspector's algorithms. This enables us to apply the method for large matrices because, the size of the memory also scales up with the number of processors.

The rest of the paper is organized as follows. In Section 2, we describe some related works dealing with compilation of irregular programs. In Section 3, we give an overview of our method built around a simple example. Then, Section 4 presents the structure of sequential input programs. Section 5 defines the filling function which computes new entries from an input program and matrix. In Section 5.3, we generate the iteration sparse dependence graph from entries performed by the filling function. Section 6 presents both the symbolic code to generate the fill-in as well as dependences, and the numerical parallel code. Moreover, some experimental results with comments are given. Finally, we leave the recapitulation to Section 7.

We assume that the reader is conversant with the elementary definitions of denotational semantics as found in the introductory chapters of a textbook on the subject as [15].

2. RELATED WORKS

Two topics are close to our works, respectively: sparse compilers which restructure a program operating on dense arrays to programs operating on sparse arrays, and the dynamic parallelization of irregular programs which analyses dependence graphs and schedules tasks at run-time.

2.1. Sparse compilers

In this topic, two works have mainly been proposed [6, 7, 14, 19]. In [6, 7], the authors have based their compiler MT1 on a use of both data structures CRS (Compressed Row Storage) and CCS (Compressed Column Storage) for storing

sparse matrices. Program transformations are formalized using polyhedral algebras. In [14, 19], the authors have based their compiler Bernoulli on a use of a generalization of both data structures CCS and CRS, called CHS (Compressed Hyperplane Storage), for storing sparse matrices. The program semantics is given using relational algebras. They assimilate sparse arrays to a relational data-base where keys are entry coordinates. In this sense, they named their approach “data-centric”: loops which originally span iteration domains are restructured to span the entry coordinates domain. Consequently, reference values which do not belong to entries are then discarded when the computation is performed since they are never accessed.

Both works are mainly focused on the automatic conversion of sequential dense programs into semantically equivalent sequential sparse codes. Therefore, the work presented in this paper can be seen as an extension of these works. Indeed, in addition, we use sparsity to convert sequential dense programs into semantically equivalent parallel sparse codes. Besides, we will show in Section 6 how these works can be integrated to our parallelization scheme.

2.2. Dynamic parallelization of programs

Dynamic parallelization aims at scheduling an application according to a dependence graph performed at runtime. In this topic, we can cite the works of [18, 11, 22].

In [18], the authors define a parallelization scheme for programs where arrays are indirectly referenced. This method is also based on the inspector-executor scheme. The inspector computes the dependence graph, and then performs a schedule of tasks, while the executor schedules and computes tasks according to the dependence graph. The dependence analysis is performed dynamically according to Bernstein’s conditions. Our work follows the same steps except we refine the Bernstein’s conditions by taking the sparsity into account. In Section 6.3, experiments will be given in order to compare the scheduling obtained from [18]’s analysis and the scheduling obtained from the generic sparse method presented in this paper.

In [11, 22], a library named RAPID has been defined which provides a runtime support to dynamically schedule acyclic irregular graphs. Dependence graphs are explicitly defined. Our work provides an automatic framework to extract the dependence graph and the symbolic computation to fill the sparse matrix of the program. The scheduling algorithms developed in the mentioned works can be used in our framework.

3. OUTLINE OF THE APPROACH

This section aims to give an overview of our approach built around a very simple example. It provides an intuitive road map for the rest of the paper. The chosen example is a simple program given in Figure 2.

The basic principle of data dependencies relies on conditions introduced by A. Bernstein [5]. Given a program where a statement T occurs before a statement S , a dependence exists if one of the three following cases holds:

1. a statement S reads a cell memory which has previously been written by a statement T , so-called *flow-dependence*;

- Program -

```
do (I = 2, 16)
  s : A(I + 1) = (A[I - 1] * A[I - 2]) - A[I + 1]
enddo
```

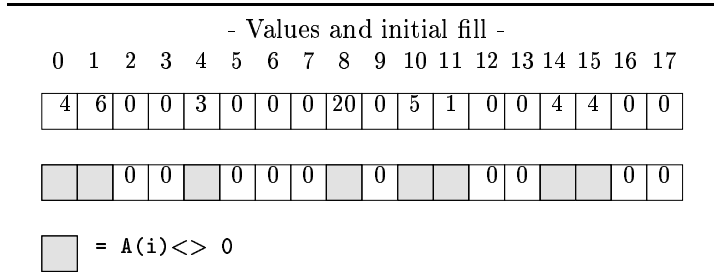


Figure 2 A simple example

2. a statement S writes in a cell memory which has previously been read by a statement T , so-called *anti-dependence*;
3. a statement S writes in a cell memory which has previously been written by a statement T , so-called *output-dependence*.

For our example, a statement is the execution of the instruction s at any iteration $2 \leq i \leq 16$. Let us denote it by $s(i)$. Therefore, the flow-dependence condition, usually noted δ^f , is expressed by the set of all pairs (i, i') of $\{1, \dots, 16\} \times \{1, \dots, 16\}$ satisfying both following conditions ¹:

1. $i < i'$
2. $i + 1 = i' - 1 \vee i + 1 = i' - 2 \vee i + 1 = i' + 1$

If we restrict our attention to the first equation $i + 1 = i' - 1$ then we obtain the following set where each pair denotes a dependence edge:

$$\delta^f = \left\{ \begin{array}{l} (2, 4), (3, 5), (4, 6), (5, 7), (6, 8), (7, 9), (8, 10), (9, 11), (10, 12), \\ (11, 13), (12, 14), (13, 15), (14, 16) \end{array} \right\}$$

However, by using the fact that the matrix is sparse, this set can be simplified due to the property of 0 to be absorbing for multiplication. For example, we can observe that $A[5]$ is equal to 0 during all the run-time of the program. Consequently, at the iteration 7, we have $A[8] = -A[8]$. Hence, whatever the content of $A[6]$ is, it does not affect the content of $A[8]$. Therefore, we deduce that the pair $(5, 7)$ can

¹We omit mentioning the instruction s because it is the single instruction of the loop.

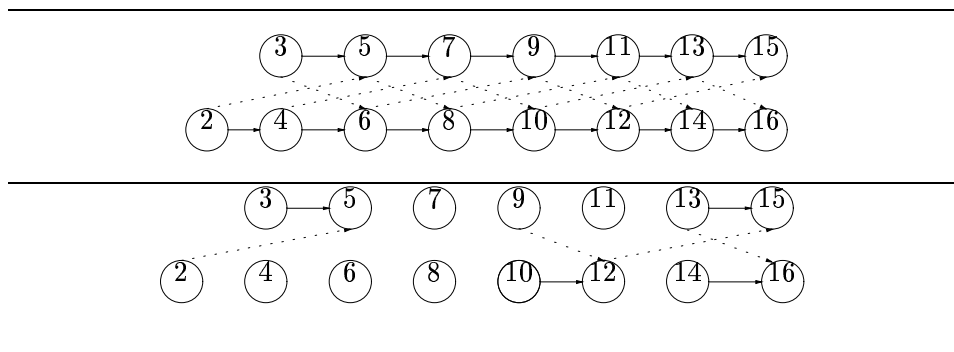


Figure 3 Dependence graph according both analysis

be removed from the previous set. By following the same analysis, the couple (4, 6) can also be removed. Iterating this process yields at the end the following set:

$$\delta^f = \{(3, 5), (10, 12), (13, 15), (14, 16)\}$$

In Figure 3, these two sets (resulting respectively from data dependence and sparse dependence analysis) are schematically represented. In this figure, lines and dashed lines represent dependencies induced by both references $A[I + 1]$ in writing and $A[I - 1]$ in reading, and $A[I + 1]$ and $A[I - 2]$, respectively. A topological sort applied to both dependence graphs provides an ASAP-scheduling of the tasks $s(i)$, $2 \leq i \leq 16$ which is described in figure 4.

time	Scheduling	
	dense dependence	sparse dependence
1	2, 3	2, 3, 7, 9, 10, 13, 14
2	4, 5	5, 12, 16
3	6, 7	15
4	8, 9	
5	10, 11	
6	12, 13	
7	14, 15	
8	16	

Figure 4 Scheduling of the program

Let us remark that in the sparse scheduling, the iterations $\{4, 6, 8, 11\}$ have been removed. They do not lead to any computation. On the contrary, the iteration 7 is preserved because the content of $A[8]$ is modified at this iteration ($A[8] = -A[8]$). Finally, we can observe from this simple example that the sparse dependence analysis reduced the completion time to 5 steps.

By this simple example, we can observe that the dependence analysis can be improved when dealing with sparse matrix. This was achieved by observing that some positions in the matrix remain at zero during all the course of the numerical execution. In other words, this analysis does not take only memory access into account,

but also values and algebraic properties of them to simplify calculus. Therefore, generalizing this method to any pair (program,matrix) first requires computing positions in the matrix the content of which will be nonzero at least once in the course of the numerical execution. In our example, such positions, so-called *entries*, are obtained from any iteration i such that both following conditions are satisfied:

1. $A[i + 1] \neq 0$ at i ;
2. for all iterations $j < i$, $A[i + 1] = 0$ at j .

Therefore, the execution of any iteration $2 \leq j \leq 16$ generates the new entry $j + 1$ if both $A[j - 2] \neq 0$ and $A[j - 1] \neq 0$. In other words, we generate a new entry $j + 1$ at the iteration j if both $j - 1$ and $j - 2$ respectively are entries. Consequently, by starting from the following set $E^0 = \{0, 1, 4, 8, 10, 11, 14, 15\}$ denoting the initial entries in the matrix and iterating the process previously defined, we obtain the final set of entries $E^{max} = \{0, 1, 3, 4, 6, 8, 10, 11, 13, 14, 15, 16, 17\}$. The interest here is that this process can be automatically computed by using the basic abstract interpretation theory. As usual, the idea is to statically collect dynamic informations about programs by using a non-standard semantics [8]. The interest is to abstract away from irrelevant matters by giving conservative approximations of the concrete behaviors of programs. Here, to generate the set of entries, we are only interested in the fact that some values remain null and the others do not. Consequently, we naturally choose to give meaning of numerical expressions in the boolean domain equipped with the usual propositional connectives. From this simple abstract interpretation, we will be able to define a function *fill*, called the *filling* function. The function *fill* will computed the whole set of entries for a given program and a matrix in input. The interest relies on the convergence of the *fill* function to a fixpoint. Formally, by applying the classical slogan: *recursive definition = fix-point equation*, the function *fill* can be defined as an iterative process, recursively specified as follows:

$$\begin{cases} E^0 = \text{initial structure of the matrix in input} \\ E^{t+1} = \text{fill}(E^t) \\ \text{fill}(E^{max}) = E^{max} \text{ (stop condition)} \end{cases}$$

The program which computes cells to be filled is automatically generated. The symbolic analysis provides affine constraints which must be satisfied to fill a cell. We will see in Section 5 that the satisfaction of these affine constraints is computable. For our example, at the iteration $2 \leq t \leq 16$, generating E^{t+1} from E^t requires to compute the set N of new entries defined by:

$$N = \{e \mid \exists 2 \leq j \leq 16, e = j + 1 \wedge j - 1 \in E^t \wedge j - 2 \in E^t\}$$

Therefore, we have $E^{t+1} = E^t \cup N$. The filling algorithm then adds new entries until no entry is computed. For our example, E^{max} is obtained in three steps. This is illustrated by Figure 5 where the black boxes denote new entries.

Obviously, all elements which do not belong to E^{max} denote positions of cells whose values will stay zero during the whole numerical execution. The idea is then to search subexpressions in the right-part of assignments under analysis, ($(A[I - 1] * A[I - 2]) - A[I + 1]$ in our example), satisfying for some iterations i that their evaluation does not affect the result of the global expression. To come back to our example, the evaluation of the subexpression $A[I - 1] * A[I - 2]$ at the iteration 7

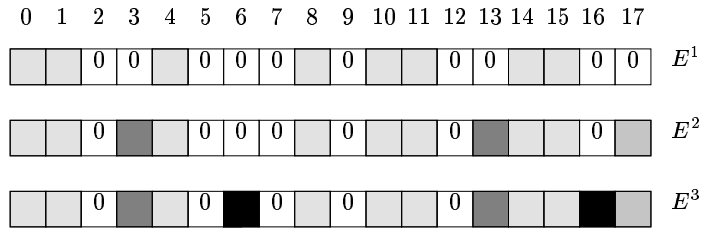


Figure 5 Example of the computation of the filling from of the example 2

satisfies such a property. Now, if some parts of form $A[-]$ of these subexpressions denote an entry at some iterations $i < 7$, obviously the pair $(i, 7)$ is not a dependence edge. In our example, $A[I - 1]$ satisfies such a condition for $i = 5$ ($6 \in E^{max}$). Indeed, we have $7 - 1 = 5 + 1$ and then $(5, 7)$ does not form a dependence.

In our analysis, this will be achieved by defining a family, indexed by the iteration space, of binary relations \mathcal{S}_i on numerical expressions, called *substitution predicates*, where the second expression will be always of the form $A[-]$. Roughly speaking, $\mathcal{S}_i(exp, exp')$ will hold if exp' is a subexpression of exp and whatever the value associated to exp' is, the evaluation of exp at the iteration i is always the same. For example, $\mathcal{S}_7((A[I - 1] * A[I - 2]) - A(I + 1), A[I - 1])$ holds because $A[5]$ is always equal to zero. Therefore, generating sparse dependencies at compile-time requires that the complement of \mathcal{S}_i can be computed. To achieve this purpose, we will still use the abstract interpretation previously defined to generate the filling function. From there, we will improve the computation of dependencies.

By returning to our example, sparse flow-dependences can be more formally defined as follows: i is flow-dependent to j if:

- $i' < i$, $<$ denotes the execution order.
- $i + 1 \in E^{max}$, the writing cell belongs to filled cells.
- $i + 1 = i' - 1$, both addressing functions reach the same memory cell for iterations i' and i .
- $(A[I - 1] * A[I - 2]) - A(I + 1), A[I - 1] \notin \mathcal{S}_j$, the substitution does not affect the result of the expression computation.

Following these ideas, the general method proposed in this paper (i.e. transforming any sequential program working on dense matrix into parallel one working on sparse matrix) follows the three following steps:

- computation of all entries, that is, positions in the matrix the content of which will become nonzero at least once in the course of the numerical execution.
- computation of iteration dependencies for generating the dependence graph. Here, we will use the previous step to refine the usual data dependence tests.
- Finally, generation of the corresponding parallel code.

4. STRUCTURE OF ANALYZED PROGRAMS

Sparse parallelization deals with programs satisfying the usual static control programming constraints [10]. To simplify the presentation, we suppose that only one array of the input program A , is sparse. In order to extend this analysis to a number of arrays, we can consider that A is made up of several sparse arrays. Consequently, we suppose that we have a sequential program as input of our compilation line. The general form of this program is inductively generated from the following rules:

- $v = exp$ where v is a scalar variable and exp is an expression;
- $A[exp_1, \dots, exp_c] = exp'$ where A is an array variable, exp_1, \dots, exp_c are integer expressions and exp' is an expression with no side effect;
- $S_1; S_2$ where S_1, S_2 are sequential programs.
- *if* exp *then* S_1 *else* S_2 where exp is a boolean expression and S_1, S_2 are sequential programs;
- *do* $(I = P, Q)$ S where I is a variable, P and Q are affine integer expressions and S is a sequential program. I is called *iteration variable*.

Given a program P and a dense matrix in input, the definition of the filling function as well as dependencies will be dependent on assignments of the form $A[exp_1, \dots, exp_c] = exp'$ encompassed in a nested loop within the program under analysis ². Thus, afterwards we will consider the following generic form of these assignments:

$$A[f(I_1, \dots, I_d)] = \mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)])$$

such that:

- A is a (dense) array where values are in \mathcal{C} (the domain ³ of the concrete semantics);
- $f : \mathbb{Z}^d \rightarrow \mathbb{Z}^c$ (resp. each $g_p : \mathbb{Z}^d \rightarrow \mathbb{Z}^c$ for every $p \in \{1, \dots, m\}$) is an affine function yielding the index of a memory cell by writing (resp. by reading) from an iteration $(i_1, \dots, i_d) \in \mathbb{Z}^d$;
- and $\mathcal{G} : \mathcal{C}^m \rightarrow \mathcal{C}$ stands for any function without side-effect.

More precisely, the function \mathcal{G} is the semantical meaning in the standard interpretation \mathcal{C} of the numerical expression $\mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)])$ inductively defined from the following set of basic numerical operators based on algebraic properties:

²Otherwise, there are not dependencies between iterations, and then there is no longer interest to generate the filling function.

³A domain is any algebraic structure, i.e., a set equipped with some internal and/or external composition laws.

operators	Definition
0	constant zero
c	any other nonzero constant
v	variable
\otimes	binary operator such that 0 is absorbing (e.g. multiply)
\oplus	binary operator such that 0 is neutral at left and/or at right (e.g. plus or minus)
μ	unary operator or function for which 0 is a fixpoint (e.g. square root)
$\odot, \bar{\mu}$	both binary and unary operators or functions the behavior of which depends on arguments (e.g. the randomize function)

We define *Expr* the whole set of numerical expressions as described above. Finally, we define *Prog* the whole set of well-formed programs (according to the inductive steps described just above) which contain at least a DO-loop nest with a statement of the form:

$$A[f(I_1, \dots, I_d)] = \mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)])$$

5. STATIC ANALYSIS

The static analysis part of the resolution aims at defining the inspector. In this section we describe how to generate at compile-time the iteration sparse dependence graph associated to a sequential program and a matrix in input. We start by defining a symbolic program, called *filling function*, for a given numerical program. The goal of this symbolic program is to statically compute the fill-in generated during the numerical computation. Moreover, from the filling function results, we improve dependences between iterations by refining the usual data-dependence (Bernstein's) conditions. The following subsections contain the main definitions of this static analysis: abstraction domain, calculation of the filling function and sparse dependencies for Do-loop nests. We give two running examples in order to help the reader's understanding of the theoretical results stated in this section. These two chosen examples are respectively the one presented in Figure 2 (cf. Section 3) and the usual Cholesky factorization algorithm whose the associated program is defined as follows:

```

SPARSE, REAL :: A(N,N)
b1 do (j = 1, N)
b2   do (k = 1, j - 1)
b3     do (i = j, N)
s1       A[i, j] := A[i, j] - A[i, k] * A[j, k]
           enddo
           enddo
s2   A[j, j] := sqrt(A[j, j])
b4   do (k2 = j + 1, N)
s3     A[k2, j] := A[k2, j] / A[j, j]
           enddo
           enddo

```

The three statements are named s_1 , s_2 and s_3 , respectively. Each one has the following characteristics:

- s_1 is encompassed by three nested loops. The function $\mathcal{G} : \mathbb{R}^3 \rightarrow \mathbb{R}$ is defined by: $(x, y, z) \mapsto y * z$. And, the affine functions f , g_1 , g_2 , and g_3 from \mathbb{N}^3

to \mathbb{N}^2 are respectively defined by: $(j, k, i) \mapsto (i, j)$, $(j, k, i) \mapsto (i, j)$, $(j, k, i) \mapsto (i, k)$, and $(j, k, i) \mapsto (j, k)$.

- s_2 belongs to a single loop. The function $\mathcal{G} : \mathbb{R} \rightarrow \mathbb{R}$ is defined by: $x \mapsto \sqrt{x}$. Both affine functions f and g_1 from \mathbb{N} to \mathbb{N}^2 are defined by: $j \mapsto (j, j)$.
- Finally, s_3 is encompassed by two nested loops. The function $\mathcal{G} : \mathbb{R}^2 \rightarrow \mathbb{R}$ is defined by: $(x, y) \mapsto x/y$. The affine functions f , g_1 and g_2 from \mathbb{N}^2 to \mathbb{N}^2 are respectively defined by: $(j, k_2) \mapsto (k_2, j)$, and $(j, k_2) \mapsto (j, j)$.

5.1. Abstraction domain

To statically collect information about programs, we define a non-standard semantics of the programming language [8]. This is based on interpreting expressions (i.e. the expressions exp' occurring in the right-part of assignments of the form: $A[exp_1, \dots, exp_c] = exp'$) in the boolean domain $\mathbb{B} = \{true, false\}$ equipped with the usual propositional connectors (mainly \wedge (and) and \vee (or)). Informally, given an expression exp' , its non-standard evaluation yields *true* if its standard numerical evaluation yields nonzero. Formally, this abstraction is defined as follows:

NOTATION 5.1. *Given a c -dimensional array, of size $(m_1, \dots, m_c) \in (\mathbb{N})^c$ in each dimensions, we define \mathbb{A} , the address space of A , the set $\{0, \dots, m_1 - 1\} \times \dots \times \{0, \dots, m_c - 1\}$.*

DEFINITION 5.2 (Abstraction domain). *Let \mathcal{C} be the domain where the standard interpretation of numerical expressions is given (e.g. natural numbers \mathbb{N} , integers \mathbb{Z} , real numbers \mathbb{R} , etc.). We define the abstraction relation $\alpha \subseteq \mathcal{C} \times \mathbb{B}$ by: $\alpha = \{(0, true), (0, false)\} \cup \{(x, true) \mid x \neq 0\}$.*

Remark 5.3. Let us notice that zero is related to both *true* and *false*. This is due to the fact that the abstract interpretation approximates the numerical results. For instance, facing an assignment of the form $A[f(I_1, \dots, I_d)] = v$ where v is a scalar variable, we cannot statically deduce if the index denoted by the expression $f(I_1, \dots, I_d)$ will be an entry or not. This depends on the value that v will have. Therefore, it is sensible to consider that the value of v is always nonzero.

Expressions are evaluated within *environments*. An environment is a mapping which associates each variable with a value of the domain. Environments are defined as follows:

DEFINITION 5.4 (Environment). *Given any domain \mathcal{C} , an environment is defined as an application $\rho_{\mathcal{D}}$ which associates:*

- the array A with a total function $f : \mathbb{A} \rightarrow \mathcal{D}$;
- a variable v with an element of \mathcal{D} ;
- each iteration index I with an element of its iteration space.

Therefore, with the notations of Definition 5.4, $\rho_{\mathbb{B}}$ will denote any environment in the domain \mathbb{B} whereas $\rho_{\mathcal{C}}$ will denote any environment in the standard domain \mathcal{C} .

Given an environment $\rho_{\mathbb{B}}$, the associated expression evaluation is defined as follows:

DEFINITION 5.5 (Expression evaluation). *Given an environment $\rho_{\mathbb{B}}$ and a numerical expression exp of $Expr$, we define $\llbracket exp \rrbracket_{\rho_{\mathbb{B}}}$ the interpretation of exp in \mathbb{B} inductively defined by the following rules:*

$$\begin{aligned}
\llbracket c \rrbracket_{\rho_{\mathbb{B}}} &= (c \neq 0). \\
\llbracket v \rrbracket_{\rho_{\mathbb{B}}} &= true. \\
\llbracket A[g_p(I_1, \dots, I_d)] \rrbracket_{\rho_{\mathbb{B}}} &= \rho_{\mathbb{B}}(A)(g_p(\rho_{\mathbb{B}}(I_1), \dots, \rho_{\mathbb{B}}(I_d))). \\
\llbracket exp_1 \otimes exp_2 \rrbracket_{\rho_{\mathbb{B}}} &= \llbracket exp_1 \rrbracket_{\rho_{\mathbb{B}}} \wedge \llbracket exp_2 \rrbracket_{\rho_{\mathbb{B}}}. \\
\llbracket exp_1 \oplus exp_2 \rrbracket_{\rho_{\mathbb{B}}} &= \llbracket exp_1 \rrbracket_{\rho_{\mathbb{B}}} \vee \llbracket exp_2 \rrbracket_{\rho_{\mathbb{B}}}. \\
\llbracket \mu(exp) \rrbracket_{\rho_{\mathbb{B}}} &= \llbracket exp \rrbracket_{\rho_{\mathbb{B}}}. \\
\llbracket \bar{\mu}(exp) \rrbracket_{\rho_{\mathbb{B}}} &= true. \\
\llbracket exp_1 \odot exp_2 \rrbracket_{\rho_{\mathbb{B}}} &= true.
\end{aligned}$$

Remark 5.6. By Definition 5.5, both operations $\bar{\mu}$ and \odot are interpreted by the two constant functions defined respectively from \mathbb{B} to \mathbb{B} and $\mathbb{B} \times \mathbb{B}$ to \mathbb{B} by: $x \mapsto true$ and $(x, y) \mapsto true$. This is due to the fact that their behavior cannot be statically deduced.

Example 5.7. Let us evaluate the expression $exp = A[i, j] - A[i, k] * A[j, k]$ of the Cholesky factorization algorithm from the environment $\rho_{\mathbb{B}}$ defined as follows:

$$\begin{aligned}
j &\mapsto n_j \text{ s.t. } n_j \in \{1, \dots, N\} \\
k &\mapsto n_k \text{ s.t. } n_k \in \{1, \dots, n_j - 1\} \\
i &\mapsto n_i \text{ s.t. } n_i \in \{n_j, \dots, N\} \\
A &\mapsto f : \{1, \dots, N\} \times \{1, \dots, N\} \rightarrow \mathbb{B} \\
&\quad (n_i, n_j) \mapsto true \\
&\quad (n_i, n_k) \mapsto false \\
&\quad (n_j, n_k) \mapsto true
\end{aligned}$$

Then, $\llbracket exp \rrbracket_{\rho_{\mathbb{B}}} = \llbracket A[i, j] \rrbracket_{\rho_{\mathbb{B}}} \vee (\llbracket A[i, k] \rrbracket_{\rho_{\mathbb{B}}} \wedge \llbracket A[j, k] \rrbracket_{\rho_{\mathbb{B}}})$. But, $\llbracket A[i, j] \rrbracket_{\rho_{\mathbb{B}}} = true$. We conclude that $\llbracket exp \rrbracket_{\rho_{\mathbb{B}}} = true$.

We obtain the following correctness result:

NOTATION 5.8. *Let ρ_C be an environment in the standard domain \mathcal{C} . Let $\rho_{\mathbb{B}}$ be an environment in the abstract domain \mathbb{B} . We say that $\rho_{\mathbb{B}}$ is compatible with ρ_C if and only if:*

- for any $(i_1, \dots, i_c) \in \mathbb{A}$, $\rho_C(A)(i_1, \dots, i_c) \alpha \rho_{\mathbb{B}}(A)(i_1, \dots, i_c)$;
- for every variable v , $\rho_C(v) \alpha \rho_{\mathbb{B}}(v)$.

PROPOSITION 5.9 (Correctness). *With the previous notation, the following relation holds:*

$$\llbracket exp \rrbracket_{\rho_C} \alpha \llbracket exp \rrbracket_{\rho_{\mathbb{B}}}$$

where $\llbracket exp \rrbracket_{\rho_C}$ denotes the usual evaluation of the expression in the standard interpretation and $\rho_{\mathbb{B}}$ is any environment compatible with ρ_C .

Proof. This proof is tedious but not difficult. It is proved by structural induction on expressions. ■

5.2. Filling

The filling deals with situations in which zero elements become nonzero. Well-known applications such as sparse Cholesky factorization use such a symbolic analysis [13], but in an ad-hoc way. The difference of one program and another depends on the definition of the *filling function* which is unique for a given a program. In this section, we will show how to statically generate the whole set of new entries. To achieve this purpose, the idea is to use the abstraction domain defined in Section 5.1 to define the filling function directly from the index space and not from the iteration space of the program under analysis. The generated program associated to the filling function will be then data-centric.

5.2.1. Entries

Given a assignment of the form:

$$A[f(I_1, \dots, I_d)] = \mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)])$$

any tuple (j_1, \dots, j_c) of \mathbb{A} will denote an entry if there is an iteration (i_1, \dots, i_d) such that both conditions hold:

1. the evaluation at (i_1, \dots, i_d) of $\mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)])$, in the abstract domain yields true; $\llbracket \mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)]) \rrbracket_{\rho_{\mathbb{B}}} = true$
2. $f(i_1, \dots, i_d) = (j_1, \dots, j_c)$.

More formally, this is expressed as follows:

NOTATION 5.10. *Let $\rho_{\mathcal{D}}$ be an environment (\mathcal{D} is any domain). Let v be any variable. An environment $\rho'_{\mathcal{D}}$ is v -equivalent to $\rho_{\mathcal{D}}$ if and only if $\rho'_{\mathcal{D}}$ is defined as $\rho_{\mathcal{D}}$ except for v .*

DEFINITION 5.11 (Entries). *Given an environment $\rho_{\mathbb{B}}$ and a program P of *Prog*, we note $\llbracket P \rrbracket_{\rho_{\mathbb{B}}}$ the subset of \mathbb{A} inductively defined by the following rules:*

- $\llbracket v = exp \rrbracket_{\rho_{\mathbb{B}}} = \emptyset$;
- $\llbracket A[f(I_1, \dots, I_d)] = exp \rrbracket_{\rho_{\mathbb{B}}}$ is the set of entries e such that for each one, there exists a tuple (i_1, \dots, i_d) of the iteration space such that both following conditions hold:
 - $e = f(i_1, \dots, i_d)$;
 - for the environment $\rho'_{\mathbb{B}}$ I_j -equivalent to $\rho_{\mathbb{B}}$ with $\rho'_{\mathbb{B}}(I_j) = i_j$ for every $j = 1, \dots, d$, we have: $\llbracket exp \rrbracket_{\rho'_{\mathbb{B}}} = true$.
- $\llbracket S_1; S_2 \rrbracket_{\rho_{\mathbb{B}}} = \llbracket S_1 \rrbracket_{\rho_{\mathbb{B}}} \cup \llbracket S_2 \rrbracket_{\rho_{\mathbb{B}}}$;
- $\llbracket if exp then S1 else S2 \rrbracket_{\rho_{\mathbb{B}}} = \llbracket S_1 \rrbracket_{\rho_{\mathbb{B}}} \cup \llbracket S_2 \rrbracket_{\rho_{\mathbb{B}}}$;
- $\llbracket do (I = P, Q) S \rrbracket_{\rho_{\mathbb{B}}} = \llbracket S \rrbracket_{\rho_{\mathbb{B}}}$.

Example 5.12. From the environment of Example 5.7, the pair (n_i, n_j) belongs to $\llbracket A[i, j] = exp \rrbracket_{\rho_{\mathbb{B}}}$. Indeed, we showed that $\llbracket exp \rrbracket_{\rho_{\mathbb{B}}} = true$.

According to definition 5.11, the difficulty lies in the calculation of iterations the execution of which yields new entries, called *significant iterations*. Indeed, the variables I_1, \dots, I_d play the role of free variables. Therefore, we must substitute them by any tuple (i_1, \dots, i_d) of the iteration space. However, the iteration space is a finite set. Consequently, the set of entries, as generated in definition 5.11, is recursive, and then, from Matiyasevich's result ⁴, the characteristic function of this set can be computed from a set of Diophantine equations. Actually, the functions f, g_1, \dots, g_m in assignments being affine, associated Diophantine equations are linear.

Below, we only define the set of Diophantine equations associated to an assignment of the form: $A[f(I_1, \dots, I_d)] = \text{exp}$. The other cases of statements can be deduced from Definition 5.11.

NOTATION 5.13. *Let us note $E_{\rho_{\mathbb{B}}}$ (resp. E_{ρ_C}) the subset of \mathbb{A} defined by: $E_{\rho_{\mathbb{B}}} = \{e \mid \rho_{\mathbb{B}}(A)(e) = \text{true}\}$ (resp. $E_{\rho_C} = \{e \mid \rho_C(A)(e) \neq 0\}$).*

DEFINITION 5.14 (Diophantine equations). *Given an assignment S of the form $A[f(I_1, \dots, I_d)] = \text{exp}$ the set $[S]_{\rho_{\mathbb{B}}}$ is generated by solving a family of Diophantine equation systems inductively defined from the expression exp as follows:*

- if exp is the constant 0 then the system is empty;
- if exp has one of the following forms: $c \neq 0$, v , $\text{exp}_1 \odot \text{exp}_2$, or $\bar{\mu}(\text{exp}_1)$ then the system is reduced to be the singleton $\{e = f(i_1, \dots, i_d)\}$ with a constraint that $e \notin E_{\rho_{\mathbb{B}}}$ and (i_1, \dots, i_d) belongs to the iteration space;
- if exp is of the form $A[g(I_1, \dots, I_d)]$ then the system is reduced to be the set $\{e = f(i_1, \dots, i_d), e' = g(i_1, \dots, i_d)\}$ with a constraint that $e \notin E_{\rho_{\mathbb{B}}}$, $e' \in E_{\rho_{\mathbb{B}}}$, and (i_1, \dots, i_d) belongs to the iteration space;
- if exp is of the form $\text{exp}_1 \oplus \text{exp}_2$ then the family of systems is defined by:

$$\mathcal{F} = \{\mathcal{F}_i\}_{i \in I_1 \cup I_2}$$

where $\{\mathcal{F}_i\}_{i \in I_1}$ (resp. $\{\mathcal{F}_i\}_{i \in I_2}$) is a family of systems of Diophantine equations and inequations defined from exp_1 (resp. exp_2);

- if exp is of the form $\text{exp}_1 \otimes \text{exp}_2$ then the family is defined by:

$$\mathcal{F} = \{\mathcal{F}_i \cup \mathcal{F}_j\}_{(i,j) \in I_1 \times I_2}$$

where $\{\mathcal{F}_i\}_{i \in I_1}$ (resp. $\{\mathcal{F}_i\}_{i \in I_2}$) is the family of Diophantine systems defined from exp_1 (resp. exp_2).

For each solution (i_1, \dots, i_d) of Diophantine systems, $f(i_1, \dots, i_d)$ then denotes a new entry.

Following Definition 5.14, Diophantine systems define characteristic functions which depend on both entry variables and iteration variables. Simplification processes can be applied on Diophantine systems in order to remove iteration variables

⁴This result states that all recursively enumerable sets are Diophantine, i.e., there exists a Diophantine equation whose set of solutions is the recursively enumerable set that we are looking for.

(see the two examples just above). *These removals lead to the definition of characteristic functions from entries to entries.* This enables us to improve the efficiency of sparse matrix code computations because the number of entries is far less than the number of iterations. For all Diophantine systems following data-centric representations, we can identify two distinct components in the definition of each \mathcal{F}_i composing it:

$$\mathcal{F}_i = \{e' = G_i(e_{i1}, \dots, e_{ini}) \mid C_i(e_{i1}, \dots, e_{ini})\}$$

where $G_i(e_{i1}, \dots, e_{ini})$ computes new entries whereas the other equations and inequations are considered as constraints. They define the predicate $C_i(e_{i1}, \dots, e_{ini})$ which is expressed as a conjunction of equalities, divisibility conditions and inequalities.

Simplification processes can be automatically performed by using tools as Omega [17] (see also [3]).

Example 5.15 (Simple Program). For the simple program given in Section 3, the interpretation leads to the following set:

$$\{i + 1 \mid i \in \mathbb{Z}, (2 \leq i \leq 16) \wedge (i + 1 \notin E_{\rho_{\mathbb{B}}}) \wedge \\ ((\exists a_0 \in E_{\rho_{\mathbb{B}}}, \exists a_1 \in \overline{E_{\rho_{\mathbb{B}}}}, \exists a_2 \in E_{\rho_{\mathbb{B}}}, (a_1 = i - 1 \wedge a_2 = i - 2) \vee a_0 = i + 1)\}$$

After simplifications, it becomes :

$$\{a_2 + 3 \mid \exists a_1 \in E, \exists a_2 \in E, (a_2 + 3 \notin E) \wedge (1 \leq a_1 \leq 15) \wedge (a_1 = 1 + a_2)\}$$

Example 5.16 (Cholesky). From the statement s_1 of the Cholesky algorithm and a given environment $\rho_{\mathbb{B}}$, the associated Diophantine system is defined by the union of the two following sets:

1. $\{(x_0, y_0) = (i, j), (x_1, y_1) = (i, j)\}$ with a constraint that $(x_0, y_0) \notin E_{\rho_{\mathbb{B}}}, (x_1, y_1) \in E_{\rho_{\mathbb{B}}}, 1 \leq j \leq N$ and $j \leq i \leq N$.
2. $\{(x_0, y_0) = (i, j), (x_1, y_1) = (i, k), (x_2, y_2) = (j, k)\}$ with as constraint that $(x_0, y_0) \notin E_{\rho_{\mathbb{B}}}, (x_1, y_1) \in E_{\rho_{\mathbb{B}}}, (x_2, y_2) \in E_{\rho_{\mathbb{B}}}, 1 \leq j \leq N, 1 \leq k \leq j - 1$ and $j \leq i \leq N$.

The first system has trivially no solution (we impose both that $(i, j) \in E_{\rho_{\mathbb{B}}}$ and $(i, j) \notin E_{\rho_{\mathbb{B}}}$). Consequently, we obtain for $\llbracket s_1 \rrbracket_{\rho_{\mathbb{B}}}$ the following set of entries:

$$\{(x_0, y_0) \mid \exists(j, k, i) \in \mathbb{Z}^3, \exists(x_1, y_1) \in E_{\rho_{\mathbb{B}}}, \exists(x_2, y_2) \in E_{\rho_{\mathbb{B}}}, \\ (i, j) \notin E_{\rho_{\mathbb{B}}} \wedge (1 \leq j \leq N) \wedge (1 \leq k \leq j - 1) \wedge (j \leq i \leq N) \\ \wedge x_0 = i \wedge y_0 = j \wedge x_1 = i \wedge y_1 = k \wedge x_2 = j \wedge y_2 = k\}$$

As the constraints $1 \leq x_1, y_1, x_2, y_2 \leq N$ are always verified (the entry coordinates are limited to the matrix bounds) the characteristic function of the set $\llbracket A(i, j) = A(i, j) - A(i, k) * A(j, k) \rrbracket_{\rho_{\mathbb{B}}}$ can be simplified as follows ⁵:

⁵Such simplifications have been automatically performed by using the symbolic computation tool Omega [17].

$$\{(x_1, x_2) \mid \exists(x_1, y_1) \in E_{\rho_{\mathbb{B}}}, \exists(x_2, y_2) \in E_{\rho_{\mathbb{B}}} \\ (x_1, x_2) \notin E_{\rho_{\mathbb{B}}} \wedge y_1 = y_2 \wedge y_1 < x_2 \leq x_1\}$$

5.2.2. Filling function

At this level of description, we are in position to define the filling function as follows:

DEFINITION 5.17 (Filling functions). *With the previous notations, from a program P and an environment $\rho_{\mathbb{B}}$ which denotes the initial environment, we define $fill : 2^{\mathbb{A}} \rightarrow 2^{\mathbb{A}}$ where $2^{\mathbb{A}}$ is the set of all subsets of \mathbb{A} (i.e. $2^{\mathbb{A}} = \{X \mid X \subseteq \mathbb{A}\}$), the total function defined by:*

$$\emptyset \mapsto E_{\rho_{\mathbb{B}}} \\ \mathbb{E} \mapsto E \cup \llbracket P \rrbracket_{\rho'_{\mathbb{B}}}$$

where $\rho'_{\mathbb{B}}$ is any environment such that $E_{\rho'_{\mathbb{B}}} = E$.

Let us show that this application fully describes an algorithm (i.e. $fill$ is recursive). To achieve this purpose, we use a classical result of the set theory due to Knaster and Tarski and known as Tarski's theorem. The statement of this theorem and all useful notations can be found in Appendix A.1. Here, we only state two theorems. The first one means that $fill$ admits a fixpoint which can be reached in a bounded steps. The second theorem states the partial correctness of the fill-in algorithm, that is, $fill$ generates all entries as performed by the numerical execution.

THEOREM 5.18 (fixpoint). *$fill$ admits a finite least fixpoint in a bounded steps.*

(The proof is given in Appendix A.2.)

The fixpoint is defined as the limit (i.e. the least upper bound) of the set $\{fill^n(\emptyset) \mid n \geq 0\}$ (i.e. $fix_{fill} = Sup \{fill^n(\emptyset) \mid n \geq 0\}$). This describes the general outline of the algorithm. The program generated from this scheme is detailed in Section 6.

THEOREM 5.19 (Partial correctness). *Let P be a program of $Prog$. Let ρ_C be an environment in the standard domain \mathcal{C} denoting the initialization of the program P . Usually, the standard interpretation of P in \mathcal{C} is defined by a finite sequence of environments $(\rho_C^1, \dots, \rho_C^n)$ such that the following conditions hold:*

- $\rho_C^1 = \rho_C$;
- for all $1 \leq i \leq n - 1$, we have:
 - $(\rho_C^{i+1}(I_1), \dots, \rho_C^{i+1}(I_d))$ is the next of $(\rho_C^i(I_1), \dots, \rho_C^i(I_d))$ according to the lexicographical order \ll on the iteration space;
 - $\rho_C^i \llbracket S \rrbracket_{\rho_C^i} \rho_C^{i+1}$ for some assignments S occurring in P .

Then, we have: $\bigcup_{1 \leq i \leq n} E_{\rho_C^i} = fix_{fill}$.

(The proof is given in Appendix A.3.)

5.3. Sparse dependence analysis

Dependence analysis consists of determining the dependence between program tasks. Thus, it aims at finding a conservative approximation of tasks that can be performed in parallel. In general, the problem of computing all dependencies at compile-time is undecidable. However, sufficient conditions of data-dependence introduced by A. Bernstein [5] ensure us such a good approximation. These conditions consist of verifying that all statements of the program under analysis do not access at the same cell memory. if we note by $R(s)$ the set of cell memories read by the statement s and $W(s)$ the set of written cell memories, then a data dependence between two statements s_1, s_2 exists if one of the three following condition holds:

1. $W(s_1) \cap R(s_2) \neq \emptyset$,
2. $W(s_2) \cap R(s_1) \neq \emptyset$, or
3. $W(s_1) \cap W(s_2) \neq \emptyset$.

In the following, we assume that iterations of nested DO-loops identify tasks (or operations [10]). For such DO-loop nests, data-dependences are defined on some assignments of the form:

$$S : A[f(I_1, \dots, I_d)] = \mathcal{G}[A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)]]$$

performed at any iteration (i_1, \dots, i_d) . Therefore, they are expressed as follows:

NOTATION 5.20 (Execution order). *Let P be a program of Prog. Let S and S' be two assignments of P . Moreover, let us suppose that S depends on iteration variables I_1, \dots, I_d and S' depends on iteration variables J_1, \dots, J_k . Let (i_1, \dots, i_d) and (j_1, \dots, j_k) be two iterations. Let us note $S(i_1, \dots, i_d)$ (resp. $S'(j_1, \dots, j_k)$) the assignment S (resp. S') performed at the iteration (i_1, \dots, i_d) (resp. (j_1, \dots, j_k)). Then, let us define $S(i_1, \dots, i_d) \prec S'(j_1, \dots, j_k)$ by:*

$$S(i_1, \dots, i_d) \prec S'(j_1, \dots, j_k) \text{ iff } \begin{cases} \text{either } (i_1, \dots, i_d) = (j_1, \dots, j_k) \text{ (among other } k = d), \\ \text{and in this case } S' \text{ occurs after } S \text{ in } P \\ \text{or } (i_1, \dots, i_d) \ll (j_1, \dots, j_k) \end{cases}$$

where \ll denotes the strict lexicographical order on the iteration space.

Roughly, $S(i_1, \dots, i_d) \prec S'(j_1, \dots, j_k)$ means that the execution of $S(i_1, \dots, i_d)$ occurs before $S'(j_1, \dots, j_k)$ in the course of the sequential execution of P .

DEFINITION 5.21 (Data dependence). *Let P be a program of Prog. Let S and S' be two assignments of P depending respectively on iteration variables I_1, \dots, I_d and J_1, \dots, J_k . Let (i_1, \dots, i_d) and (j_1, \dots, j_k) be two iterations such that $S(i_1, \dots, i_d) \prec S'(j_1, \dots, j_k)$. Then,*

flow-dependence: $(S, (i_1, \dots, i_d))$ is flow-dependent to $(S', (j_1, \dots, j_k))$, usually noted $(S, (i_1, \dots, i_d)) \delta^f (S', (j_1, \dots, j_k))$, iff:

$$W(S(i_1, \dots, i_d)) \cap R(S'(j_1, \dots, j_k)) \neq \emptyset$$

anti-dependence: $(S, (i_1, \dots, i_d))$ is anti-dependent to $(S', (j_1, \dots, j_k))$, usually noted $(S, (i_1, \dots, i_d)) \bar{\delta} (S', (j_1, \dots, j_k))$, iff:

$$R(S(i_1, \dots, i_d)) \cap W(S'(j_1, \dots, j_k)) \neq \emptyset$$

output-dependence: $(S, (i_1, \dots, i_d))$ is output-dependent to $(S', (j_1, \dots, j_k))$, usually noted $(S, (i_1, \dots, i_d)) \delta^o (S', (j_1, \dots, j_k))$, iff:

$$W(S(i_1, \dots, i_d)) \cap W(S'(j_1, \dots, j_k)) \neq \emptyset$$

When using sparse matrices, we can refine data dependence conditions by using the properties of zero to be absorbing and neutral. To achieve this purpose, we will use the abstraction domain defined in Section 5.1 as well as the following property:

NOTATION 5.22. Given an expression exp and a subexpression exp' of exp , we note $exp[exp'/x]$ the expression obtained from exp by substituting all occurrences of exp' by a fresh variable x (e.g. a variable which has not been used in the program).

DEFINITION 5.23 (Substitution property). Let (i_1, \dots, i_d) be an iteration of the iteration space. We note $\mathcal{S}_{(i_1, \dots, i_d)} \subseteq Expr \times Expr$ the binary relation defined by:

$$exp \mathcal{S}_{(i_1, \dots, i_d)} exp' \text{ iff } \begin{cases} \text{either } exp' \text{ is not a subterm of } exp, \\ \text{or for every } \rho_C \text{ such that } \begin{cases} \rho_C(I_j) = i_j, 1 \leq j \leq d \\ E_{\rho_C} = fix_{fill} \end{cases} \\ \text{we have: } \begin{cases} \llbracket exp \rrbracket_{\rho_C} = 0 & \text{if } exp = exp' \\ \llbracket exp[exp'/x] \rrbracket_{\rho'_C} = \llbracket exp \rrbracket_{\rho_C} & \\ \text{for every } \rho'_C \text{ } x\text{-equivalent to } \rho_C & \text{otherwise} \end{cases} \end{cases}$$

Definition 5.23 calls for some comments:

Let exp be the expression with the form:

$$\mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)])$$

and let exp' be the sub-expression of exp of the form $A[g_p(I_1, \dots, I_d)]$ where $p \in \{1, \dots, m\}$. Therefore, given an iteration (i_1, \dots, i_d) , $exp \mathcal{S}_{(i_1, \dots, i_d)} exp'$ means that the evaluation of exp does not depend on $A[g_p(i_1, \dots, i_d)]$ whatever its content is. This plays the role of a fresh variable. For example, this condition holds when for any expression $A[g_p(i_1, \dots, i_d)] \otimes A[g_{p'}(i_1, \dots, i_d)]$ with $p' \neq p \in \{1, \dots, m\}$, $g_{p'}(i_1, \dots, i_d)$ is not an entry (i.e. $g_{p'}(i_1, \dots, i_d) \notin fix_{fill}$).

Thus, let us suppose that there exists an iteration (j_1, \dots, j_d) of the iteration space such that $(j_1, \dots, j_d) \ll (i_1, \dots, i_d)$ and $f(j_1, \dots, j_d) = g_p(i_1, \dots, i_d)$ (i.e. the cell memory indexed by $g_p(i_1, \dots, i_d)$ was written at the iteration (j_1, \dots, j_d)). Given a statement S of the form:

$$A[f(i_1, \dots, i_d)] = \mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)])$$

we have:

$$A[g_p(i_1, \dots, i_d)] \in R(S(i_1, \dots, i_d)) \cap W(S(j_1, \dots, j_d))$$

Consequently, according to Bernstein's conditions we have a data dependence between both tasks identified by (i_1, \dots, i_d) and (j_1, \dots, j_d) . However, from $exp \mathcal{S}_{(i_1, \dots, i_d)} exp'$, we have:

$$S(i_1, \dots, i_d); S(j_1, \dots, j_d) = S(j_1, \dots, j_d); S(i_1, \dots, i_d)$$

Hence, by using substitution property, Bernstein's conditions can be refined as follows:

DEFINITION 5.24 (Sparse Bernstein's conditions). *Let P be a program of $Prog$. Let S be an assignment of P of the form:*

$$A[f(i_1, \dots, i_d)] = \mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)])$$

Let S' be an assignment of P of the form:

$$A[f'(J_1, \dots, J_k)] = \mathcal{G}'(A[g'_1(J_1, \dots, J_k)], \dots, A[g'_{m'}(J_1, \dots, J_k)])$$

Let (i_1, \dots, i_d) and (j_1, \dots, j_d) be two iterations such that $S(i_1, \dots, i_d) \prec S'(j_1, \dots, j_k)$. Then:

sparse flow-dependence: $(S, (i_1, \dots, i_d))$ is sparse flow-dependent to $(S', (j_1, \dots, j_k))$

$$\begin{aligned} \text{iff: } & f(i_1, \dots, i_d) \in \text{fix}_{fill} \\ & \wedge (\exists 1 \leq p \leq m', f(i_1, \dots, i_d) = g'_p(j_1, \dots, j_k)) \\ & \wedge (\mathcal{G}(A[g'_1(J_1, \dots, J_k)], \dots, A[g'_{m'}(J_1, \dots, J_k)]), A[g'_p(J_1, \dots, J_k)]) \notin \mathcal{S}'_{(j_1, \dots, j_k)} \end{aligned}$$

sparse anti-dependence: $(S, (i_1, \dots, i_d))$ is sparse anti-dependent to $(S', (j_1, \dots, j_d))$

$$\begin{aligned} \text{iff: } & f'(j_1, \dots, j_k) \in \text{fix}_{fill} \\ & \wedge (\exists 1 \leq p \leq m, f'(j_1, \dots, j_k) = g_p(i_1, \dots, i_d)) \\ & \wedge (\mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)]), A[g_p(I_1, \dots, I_d)]) \notin \mathcal{S}_{(i_1, \dots, i_d)} \end{aligned}$$

sparse output-dependence: $(S, (i_1, \dots, i_d))$ is output-dependent to $(S', (j_1, \dots, j_k))$

$$\begin{aligned} \text{iff: } & f(i_1, \dots, i_d) \in \text{fix}_{fill} \\ & \wedge f(i_1, \dots, i_d) = f'(j_1, \dots, j_k) \end{aligned}$$

Generating sparse dependencies at compile-time requires that the complement of the relation $\mathcal{S}_{(i_1, \dots, i_d)}$ with respect to $Expr \times Expr$ is defined. As for the filling function, we need to use the abstract interpretation defined in Section 5.1.

DEFINITION 5.25 (Unsubstitution algorithm). *For any (i_1, \dots, i_d) , let us note $\bar{\mathcal{S}}_{(i_1, \dots, i_d)} : Expr \times Expr \rightarrow \mathbb{B}$ the application inductively defined by:*

- $\bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp', exp') = \llbracket exp' \rrbracket_{\rho_{\mathbb{B}}}$ where $\rho_{\mathbb{B}}$ denotes any environment such that $E_{\rho_{\mathbb{B}}} = \text{fix}_{fill}$ and for every $j \in \{1, \dots, d\}$ $\rho_{\mathbb{B}}(I_j) = i_j$;
- $\bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp, exp') = \text{false}$ if exp' is not a sub-expression of exp ;
- $\bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_1 \otimes exp_2, exp') = \llbracket exp_1 \otimes exp_2 \rrbracket_{\rho_{\mathbb{B}}} \wedge (\bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_1, exp') \vee \bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_2, exp'))$ where $\rho_{\mathbb{B}}$ denotes any environment such that $E_{\rho_{\mathbb{B}}} = \text{fix}_{fill}$ and for every $j \in \{1, \dots, d\}$ $\rho_{\mathbb{B}}(I_j) = i_j$.
- $\bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp, exp') = \text{false}$ if exp' is not a sub-expression of exp ;
- $\bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_1 \oplus exp_2, exp') = \bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_1, exp') \vee \bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_2, exp')$
- $\bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_1 \odot exp_2, exp') = \bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_1, exp') \vee \bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_2, exp')$
- $\bar{\mathcal{S}}_{(i_1, \dots, i_d)}(\phi(exp_1), exp') = \bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_1, exp')$ where $\phi \in \{\mu, \bar{\mu}\}$.

With such an approach, we only give a rough estimate of the complement of $\mathcal{S}_{(i_1, \dots, i_d)}$ which can be symbolically computed from the entries, as it is shown by the following result:

THEOREM 5.26. $(exp, exp') \notin \mathcal{S}_{(i_1, \dots, i_d)} \implies \overline{\mathcal{S}}_{(i_1, \dots, i_d)}(exp, exp')$
(The proof is given in Appendix A.4.)

From there, we can redefine iteration dependencies such that they can be computed at the inspector phase. Indeed, it is sufficient to replace in Definition 5.24 both conditions:

1. $(\mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)]), A[g_p(I_1, \dots, I_d)]) \notin \mathcal{S}_{(j_1, \dots, j_d)}$
2. $(\mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)]), A[g_p(I_1, \dots, I_d)]) \notin \mathcal{S}_{(i_1, \dots, i_d)}$

by both following ones:

1. $\overline{\mathcal{S}}_{(j_1, \dots, j_d)}(\mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)]), A[g_p(I_1, \dots, I_d)])$
2. $\overline{\mathcal{S}}_{(i_1, \dots, i_d)}(\mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)]), A[g_p(I_1, \dots, I_d)])$

Example 5.27 (Simple example). In this example, we only study sparse flow dependencies for s . δ^f is then defined as follows:

$$\begin{array}{ll}
i_1 \delta^f i_2 \Leftrightarrow & \exists a_1 \in \text{fix}_{fill}, \exists a_2 \in \text{fix}_{fill}, \\
\text{Domain by writing} & 2 \leq i_1 \leq 16 \wedge \\
\text{Domain by reading} & 2 \leq i_2 \leq 16 \wedge \\
\text{Ordering} & i_1 < i_2 \wedge \\
\overline{\mathcal{S}} & a_1 = i_2 - 1 \wedge a_2 = i_2 - 1 \\
\text{Dep. with } A[I - 1] & [(i_1 + 1 = i_2 - 1 = a_1) \\
& \vee \\
\text{Dep. with } A[I - 2] & (i_1 + 1 = i_2 - 1 = a_2)]
\end{array}$$

The previous formulation can be simplified to become data-centric:

$$i_1 \delta^f i_2 \equiv \exists a_1 \in \text{fix}_{fill}, \exists a_2 \in \text{fix}_{fill}, i_2 = a_1 + 1 \wedge (a_2 = a_1 - 1) \\
[(i_1 = a_1 - 1 \wedge 3 \leq a_1 \leq 15) \vee (i_1 = a_1 - 2 \wedge 4 \leq a_1 \leq 15)]$$

Example 5.28 (Cholesky). Here, we will only give sparse flow dependencies on the assignment s_1 . The computation of δ^f on s_1 is then defined by:

$$\begin{array}{ll}
(s_1, (j, k, i)) \delta^f (s_1, (j', k', i')) \Leftrightarrow & \exists (x_1, y_1) \in \text{fix}_{fill}, \exists (x_2, y_2) \in \text{fix}_{fill}, \\
\text{domain by writing} & 1 \leq j \leq N \wedge 1 \leq k \leq j - 1 \wedge j \leq i \leq N \wedge \\
\text{domain by reading} & 1 \leq j' \leq N \wedge 1 \leq k' \leq j' - 1 \wedge j' \leq i' \leq N \wedge \\
\text{ordering} & (j, k, i) \ll (j', k', i') \wedge \\
\text{Dep. with } A[I, K] & \\
\text{references} & [(x_1 = i = i' \wedge y_1 = j = k' \wedge \\
\overline{\mathcal{S}} & i' = x_1 \wedge k' = y_1 \wedge j' = x_2 \wedge k' = y_2) \\
& \vee \\
\text{Dep. with } A[J, K] & \\
\text{references} & (x_1 = i = j' \wedge y_1 = j = k' \wedge \\
\overline{\mathcal{S}} & j' = x_1 \wedge k' = y_1 \wedge i' = x_2 \wedge k' = y_2)]
\end{array}$$

As previously, we can simplify the characteristic function as follows:

$$(s_1, (j, k, i))\delta^f(s_1, (j', k', i')) \Leftrightarrow \begin{aligned} &\exists(x_1, y_1) \in fix_{fill}, \exists(x_2, y_2) \in fix_{fill} \\ &y_1 = y_2 \wedge j = y_1 = k \wedge i = x_1 \wedge k' = y_2 \\ &[(i' = x_1 \wedge j' = x_2 \wedge y_1 < x_2) \vee \\ &(i' = x_2 \wedge j' = x_1 \wedge (x_1 < x_2 \vee y_1 < x_1))] \end{aligned}$$

Similarly to the analysis of the filling function, the computation of dependencies only depend on entry variables. As previously, this also constitutes a necessary condition to generate an efficient inspector program which computes dependencies.

DEFINITION 5.29 (Sparse dependence graph). *Let P be a program of $Prog$. The set of dependencies Δ_P is defined by: $\Delta_P = \delta^f \cup \bar{\delta} \cup \delta^o$.*

6. CODE GENERATION

This section deals with the code generation both at the symbolic and numerical levels. The code generation step converts any dense sequential program to an SPMD sparse program for shared memory architecture. This is divided into two parts:

1. the code generation of the symbolic program where we compute from the references of nonzero entries of the matrix, the fill-in and the sparse dependence graph.
2. The generation of the numerical parallel sparse program where we compute the scheduling of iteration sets which can be independently performed concurrently, so-called *fronts*, from dependences previously defined.

The code generation is presented here by using any parallel Fortran-like programming language and a dedicated generic data-structure library. The choice of a parallel Fortran-like programming language enables us to define a generic framework without introducing specific knowledge on languages, which facilitates the presentation. However, higher languages based on relational databases [14, 19], or using dedicated data-parallel structures [2] will certainly provide abstractions to significantly simplify the code generation.

6.1. Symbolic code generation

During the static analysis (Section 5), we computed two sets of conditions defining respectively the fill-in and the sparse dependencies. As we saw in Section 5, these conditions are *data-centric*, that is, we scan available entries to generate both new ones and dependences. Therefore, the insertion step will consist of adding new entries as well as dependencies. Obviously, the conditions can be computed in parallel.

To define the parallel implementation, we follow the OPEN-MP norm [16] which corresponds to a programming language with a single address space. The parallel loop will be defined in this article by `doall` [20].

6.1.1. Data structures for entries and dependences

Before giving the parallel code for the fill-in and sparse dependences, we must first choose suitable Sparse Data Structures, for short SDS, to store both entries

and the dependence graphs. By following [6, 7, 14], we choose hash tables as SDS. The interest of such a representation is threefold:

1. Hash tables can be considered as generic implementations of SDS. Their genericity lies on the ability to convert any address of a multi-dimensional array (space) to a homogeneous address which corresponds to the primary entry.
2. Accessing to elements is efficient.
3. Hash tables subsume the Line Storage Format (resp. Column Storage Format) where one of matrix lines (resp. matrix columns) is used as the primary hash key.

Hash tables can also be considered as suitable structures to implement dependence graph because graph implementation by matrices often leads to sparse matrices. It is sufficient to define a hashing function for any pair (\vec{j}_1, \vec{j}_2) representing dependence edges between two iterations.

Usually, hash tables are implemented by dynamically allocated arrays of arrays. In order to minimize the complexity, we will assume that elements of the second dimension will be sorted in increasing order. This will allow us to insert new elements by using a dichotomic search. Therefore, the two following functions are required:

- *bool insert(value, SDS)* : inserts a value in the SDS. The primitive returns a boolean value which is true if the entry already exists.
- *bool member(value, SDS)* : checks whether a value has already been stored or not.

Moreover, memory locking and unlocking primitives are implicitly used during the execution of insert and member primitives. This allows atomic accesses to the secondary entries without having conflicts. Finally, parallelism is expressed by using `doall` loops which scan every element in parallel. This can be implemented by OPEN-MP parallel loops which scan primary entries.

6.1.2. Code generation of the filling program

In Section 5, we saw that the filling function can be specified as follows:

$$\bigcup_{i=1,m} \mathcal{F}_i(e_{i1}, \dots, e_{in_i}) \text{ where :}$$

$$\mathcal{F}_i(e_{i1}, \dots, e_{in_i}) = \{e' = G_i(e_{i1}, \dots, e_{in_i}) | C_i(e_{i1}, \dots, e_{in_i})\}$$

From this representation, the filling program scans the current set of entries E for each entry e_{ij} of the parameter $(e_{i1}, \dots, e_{in_i})$. An improvement of this scheme is to consider only entries previously introduced because without creation the fixpoint is necessarily reached.

Without loss of generality, we assume that new entries correspond to \vec{e}_1 . The generated code for the filling function is then given in Figure 6.

Despite the complexity of this scheme which is mainly due to the genericity required to its definition, practical applications lead to a very simple code as we can see it in the two examples above.

Input:
A program P of *Prog* and an *SDS* E storing the initial entries

Output:
a final set E of entries for the program P .

program:
 $N_1 = E$
while ($N_1 \neq \emptyset$)
 $N_2 = \emptyset$
 doall ($e_{11} \in N_1$)
 ...
 doall ($e_{1n_1} \in E$)
 if ($C_1(e_{11}, \dots, e_{1n_1})$) then
 $e' = G_1(e_{11}, \dots, e_{1n_1})$
 if (insert(e' , E)) then insert (e' , N_2) endif
 endif
 enddoall... enddoall
 ...
 doall ($e_{m1} \in N_1$)
 ...
 doall ($e_{mn_m} \in E$)
 if ($C_m(e_{m1}, \dots, e_{mn_m})$) then
 $e' = G_m(e_{m1}, \dots, e_{mn_m})$
 if (insert(e' , E)) then insert (e' , N_2) endif
 endif
 enddoall... enddoall
 $N_1 = N_2$
endwhile
output: E ;

Figure 6 Filling symbolic code

```

N1 = E
while (N1 ≠ ∅)
  N2 = ∅ ;
  doall ((x0, y0) ∈ E)
    doall ((x1, y1) ∈ E)
      if (y0 = y1 ∧ y0 < x1 ≤ x0) then (x, y) = (x0, x1)
      if (insert((x, y), E)) then insert ((x, y), N2) endif
    enddoall
  enddoall
  doall ((x0, y0) ∈ E)
    doall ((x1, y1) ∈ E)
      if (x1 = y1 ∧ x1 ≤ x0) then (x, y) = (x0, x1)
      if (insert((x, y), E)) then insert ((x, y), N2) endif
    enddoall
  enddoall
  N1 = N2
endwhile

```

Figure 7 Filling code for Cholesky

Example 6.1 (Cholesky). In example 5.16, we gave conditions to generate entries associated to the assignment s_1 of the Cholesky factorization algorithm. Before giving the associated filling symbolic code, we must in addition define conditions for generating entries from the two other assignments s_2 and s_3 .

For s_2 , the associated conditions are empty because s_2 is of the form $A[f(i_1, \dots, i_d)] = \mu(A[f(i_1, \dots, i_d)])$. On the contrary, the assignment s_3 gives rise to the following set:

$$\{(x_0, y_0) \mid \exists(k_2, j) \in \mathbb{Z} \times \mathbb{Z}, \exists(x_1, y_1) \in E_{\rho_{\mathbb{Z}}}, \\ (x_0, y_0) \notin E_{\rho_{\mathbb{Z}}} \wedge (1 \leq j \leq N) \wedge (j + 1 \leq k_2 \leq N) \wedge \\ x_0 = k_2 \wedge y_0 = j \wedge x_1 = j \wedge y_1 = j\}$$

As previously, this set can be simplified as follows:

$$\{(x_0, x_1) \mid \exists(x_1, y_1) \in E_{\rho_{\mathbb{Z}}}, (x_0, x_1) \notin E_{\rho_{\mathbb{Z}}} \wedge x_1 = y_1 \wedge x_1 \leq x_0 \leq N\}$$

If we add that $x_j \leq N$ is always satisfied we have : (note that this simplification can be automatically performed)

$$\{(x_0, x_1) \mid \exists(x_1, y_1) \in E_{\rho_{\mathbb{Z}}}, (x_0, x_1) \notin E_{\rho_{\mathbb{Z}}} \wedge x_1 = y_1 \wedge x_1 \leq x_0\}$$

The generated code for the Cholesky factorization algorithm is given in Figure 7.

Example 6.2 (Simple Example). Similarly the simple example detailed in Section 3 leads to the program given in Figure 8.

The previous program can be optimized in order to reduce the dimension of the parameter space. This optimization is based on the fact that checking whether equalities are satisfied or not can be done by the member primitive. The resulting

```

 $N_1 = E$  ;
while ( $N_1 \neq \emptyset$ )
   $N_2 = \emptyset$  ;
  doall ( $a1 \in N_1$ )
    doall ( $a2 \in E$ )
      if ( $a1 >= 1 \wedge a1 \leq 15 \wedge a2 = a1 - 1$ ) then
        if (insert( $E, a2 + 3$ )) then
          insert( $N_2, a2 + 3$ );
        enddoall
      enddoall
    enddoall
   $N_1 = N_2$ 
endwhile

```

Figure 8 Filling program

code is given in Figure 9.

6.1.3. Generation of the dependence computation program

As in Section 6.1, generating dependences amounts to satisfying conditions denoting respectively: domain by writing, domain by reading, identical references and unsubstitution properties. Consequently, the generic code is like Figure 6 except where conditions are those used to generate Δ_P instead of those used to generate entries (i.e. conditions denoted by the family \mathcal{F}).

6.2. Generation of the numerical program

From a sequential program as described in Section 4, the equivalent parallel program is based on a scheduler which allows to run tasks in parallel. Since each task is identified by an iteration, scheduling is represented by an order on iterations. If we define by \mathcal{D} the iteration space then the program of Figure 10 describes a static scheduling by front for a PRAM model (shared memory). The loop with the index t schedules fronts. Each front is described by a set of iterations $\Theta(t)$. The partition of the iteration space \mathcal{D} can also be made in parallel [18] by performing a topological sort on the dependence graph. Finally, $\text{Sparse}(\mathbf{P}(\vec{I}^t))$ embodies the program rewritten for sparse structures.

This is usual in automatic parallelization [10]. However, other methods [11], such as a dynamic scheduling [18] (self-scheduling), can be applied.

The program $\mathbf{P}(\vec{I}^t)$ has all the characteristics of a sequential program with nested do loops, that is: access to memory is based on a single address space, and the control flow is sequential. Hence, we can automatically convert it into an equivalent sparse sequential code by using existing tools designed to this purpose [6, 7, 14, 19]. Moreover, the execution order of the tasks of \mathbf{P} is controlled by the scheduler. Therefore, the proposed approach in this paper is compatible with studies performed to rewrite dense code to sparse code (cf. Section 2.1). Consequently, they can be integrated into the sparse parallelization process. In order to fully describe how code

```

 $N_1 = E$ 
while ( $N_1 \neq \emptyset$ )
   $N_2 = \emptyset$ ;
  doall ( $a \in N_1$ )
     $a1 = a; a2 = a1 - 1$ ;
    if ( $a1 \geq 1 \wedge a1 \leq 15 \wedge \text{member}(a2, E)$ ) then
      if ( $\text{insert}(E, a2 + 3)$ ) then  $\text{insert}(N_2, a2+3)$ ;
       $a2 = a; a1 = a2+1$ ;
      if ( $(a2 \geq 0 \wedge a2 \leq 14) \wedge \text{member}(E, a1)$ ) then
        if ( $\text{insert}(E, a2 + 3)$ ) then  $\text{insert}(N_2, a2+3)$ ;
    enddoall
   $N_1 = N_2$ 
endwhile

```

Figure 9 Filling program

```

compute the filling
compute the sparse dependence graph
 $\Theta = \text{Partition of } \mathcal{D} \text{ to figure out a scheduling}$ 
do  $t = 1, |\Theta|$ 
  doall  $\vec{I} \in \Theta(t)$ 
    Sparse( $P(\vec{I})$ )
  enddo
  synchronization
ENDDO

```

Figure 10 Skeleton of the sparse parallel program

generation can be handled in the scope of the parallelism coming from sparsity, we give two simple frameworks to rewrite the program P:

1. The first framework is straightforward but does not use the sparsity to improve the storage. Only the sparse parallelism is considered. In this case, the program P remains unchanged. Consequently, only the control originally performed by do loops is achieved by a scheduler in the new version.
2. Another possible implementation which suits the method is to consider the sparse structure as a relational table. This follows [14]. Here, schematically, sparse formats are considered as a specialized implementation of them. This can also be implemented by a hash table. In this case, the records of the table used to implement E are extended to store values. Each record is defined by a pair $(\vec{a}, A(\vec{a}))$, where \vec{a} is an entry and $A(\vec{a})$ is its value.

These frameworks can be improved by using the same optimizations as those used by the works mentioned in Section 2.

6.3. Experiments

Experiments described in this section, compares [18]'s method (see Section 2.2) and the proposed method when applied to the program described in Example 2. In the following, we will name [18]'s analysis *DAE analysis* (Dynamic dependence Analysis restricted to Entries).

These experiments have been performed on a SGI ORIGIN 2000 (8 processors). They have been obtained from arrays of different sizes. The presented results of experiments has been performed for a vector of size 1000. Other experiments performed on arrays with other sizes lead to similar results. The curves depict two kinds of experiments which differ on how arrays are initially filled: arrays of the first class of experiments (tests A) are filled in such a way that each address of filled cell has been randomly selected in different segments of size 3. This avoids large fillings. The second class (tests B) corresponds to an initial random fill. The left curve of Figure 11 describes the evolution of the number of entries introduced during the computation when the initial density varies from 1 % to 50 % (for higher densities, arrays are considered as always filled). The curve located on the right of Figure 11 represents the maximum number of iterations for reaching the fix-point. The result of the test B emphasizes a sensibility to the initial conditions of the filling. More precisely, the appearance of three consecutive filled cells leads to a filling of all the cells located on the right of these three cells. The curve of Figure 12 represents the number of steps required to achieve the computation. This corresponds to the length of the longest path in the dependence graph. The number of iteration to reach the fixpoint is bounded by ≈ 40 iterations. In comparison to the size of the vector which corresponds to the upper bound of the computation, we can notice that the number of iteration corresponds to less than 0.5% of the maximal number of iteration. Since the complexity of the filling is the same as the complexity of the program, the time is proportional to the filled values. In practice, the time spent to fill and to perform the numerical computation of the vector is negligible in comparison to the time spent performing a computation in dense vector.

Indeed, the order of a speedup⁶ exceeds 10 when less than 40% of arrays are filled. The speedup between the sequential sparse version and the parallel sparse

⁶Let us recall that the speedup is defined as follows: Let T_{dense} be the execution time of the

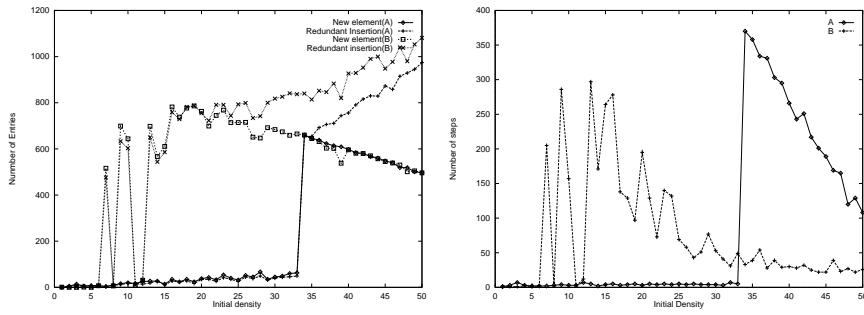


Figure 11 fill-in

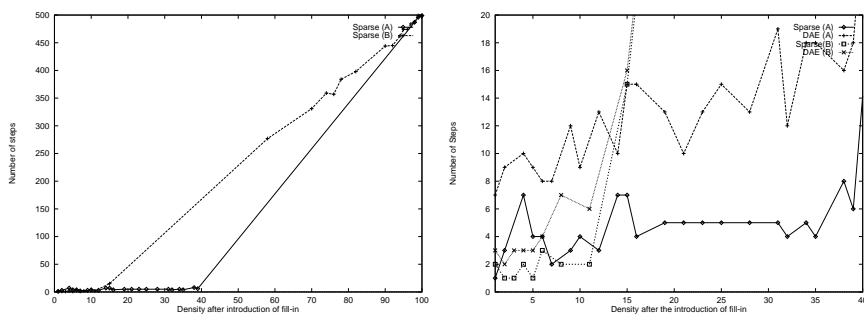


Figure 12 Scheduling

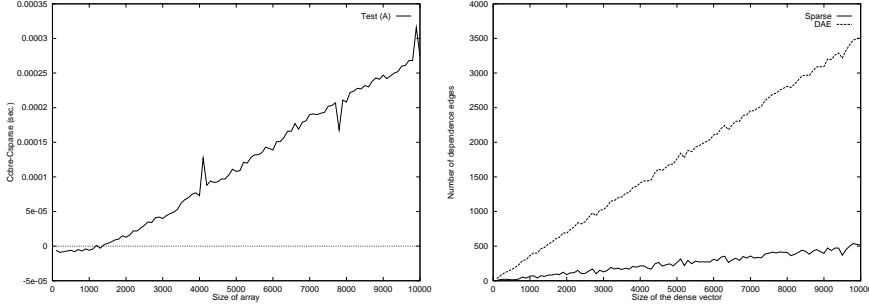


Figure 13 Dependencies

one on 8 processors is between 2 and 4. The computation time of the sparse parallel method includes the time spent to compute the filling, the dependence graph and the parallel numerical computation (and not merely the numerical computation).

This confirms the importance of the method for the parallelization process as it is admitted that a structure is considered to be sparse if less than 5% of the cells are filled. Thus, the main limitation of the method lies on the overhead induced by both management of the sparsity and computation of the dependence graph. Let us recall that the DAE method also describes a “sparse parallelizing method”. The right curve of Figure 12 represents the time of execution according to the two methods for arrays which have less than 40% of fill-in. Let us notice that the absorption properties increase the speedup by an average factor of 2 in comparison to the DAE method. If we consider cases where 5% at most of the array are filled, the speedup is 5 times greater than DAE.

Figure 13 details the difference of the dependence computation cost between the DAE method and the sparse method, that is: $(Cost_{DAE} - Cost_{sparse})$. The sizes of tested arrays are ranged from 100 to 10000 with 20% of filled cells. Since the computation of the fill-in is necessary for both methods, its cost is removed from the computation of the difference. One can remark that for a given size (1500 for the figure) the cost of the sparse dependence is less than the cost of the DAE method. This can be explained as follows : If we assume that the time access complexity in a hash table is $\log_2(|fix_{fill}|)$ then the time cost computation of the dependence graph for both methods are defined as follows :

$$\begin{aligned} Cost_{DAE} &= \alpha \cdot |fix_{fill}| \cdot |\Delta_{DAE}| + \beta \\ Cost_{sparse} &= \alpha' \cdot |fix_{fill}| \cdot (|\Delta_{sparse}| + \log_2(|fix_{fill}|)) + \beta' \end{aligned} \quad (1)$$

where α , α' , β et β' are constants which depend on the implementation. Given that the implementation of both programs are similar, we can admit that $\alpha \approx \alpha'$ et $\beta \approx \beta'$. Hence, the difference $Cost_{DAE} - Cost_{sparse} > 0$, when $\alpha \cdot |\Delta_{DAE}| - \alpha' \cdot |\Delta_{sparse}| > \log_2(|fix_{fill}|) - (\beta - \beta')/|fix_{fill}|$. Consequently, the gain of the sparse method is explained by the number of dependence edges which can be safely discarded by this method. This analytical result is experimentally confirmed by

dense (necessary) sequential program, and let T_{sparse} be the execution time of the sparse parallel program, then the speedup is the ratio $s = T_{dense}/T_{sparse}$.

the right curve of Figure 13 where the two curves represent the number of dependence edges obtained with both methods. The number of edges computed by the sparse method is much lower than the one computed by the DAE method. These Experiments put the element which favors the speedup forward: the dependence edge removal. This gives rise to the reduction of the cost of the symbolic computation because the amount of treated data decreases in comparison of the DAE method. In the example, this also leads to a reduction of the completion time of the algorithm. Actually, the number of dependence edges computed by the DAE method represents the asymptotic bound of our method. This is also the case for the number of steps needed to compute the parallelized program.

7. CONCLUSION

In this paper, we have presented an automatic method which from sequential programs working on dense matrices generates a parallel counterpart working on sparse matrices. This work is divided into a static part and a code generation part. The static part is itself split up into two steps. In the first step, we have defined the filling function to symbolically compute the indexes of the matrix in input whose the content will be nonzero at least once in the course of the execution. To achieve this purpose, we have used a non-standard semantics of numerical expressions from the propositional calculus. With such an abstraction, we have shown by Theorem 5.18 that the filling function can be computed. The resulting algorithm computes, from the program text and a matrix in input, the necessary memory which will be used to store the matrix content during the real execution of the program. Theorem 5.19 establishes the partial correctness of this algorithm. In the second part, we have generated the sparse dependence graph. This graph has been obtained from data-dependencies of DO-loop nests. These dependencies have been computed by refining the usual Bernstein's conditions to dynamical concepts. These dynamical concepts denote both that a memory cell used by two different iterations is an entry, (the whole set of entries is the result yielded by the filling function), and that the cell read in one of these iterations is not substitutable by any values (i.e. their contents are important in the numerical expression where they occur). The code generation part generates an SPMD sparse program for shared memory architecture. Therefore, from an automatic analysis, we have first defined the two symbolic programs associated respectively to the fill-in and the sparse dependence graph. we have also defined a numerical parallel sparse program using a dynamic scheduling policy with the help of the MT1 compiler [6].

A.1. Partially ordered sets and Tarski's theorem

Here, we recall some basic definitions and results about partially ordered sets, for short often called posets.

DEFINITION A.1 (Partially ordered sets). *A partial order is a reflexive, transitive and antisymmetric relation which is usually written \prec . The pair (A, \prec) consisting of a set A and a partial order \prec is called a partially ordered set or poset.*

DEFINITION A.2 (directed and complete posets). *Given a poset (X, \prec) , a subset of X is directed if any two of its elements has an upper bound in E . A poset (X, \prec) is complete if it has a least element according to the order \prec and if any directed subset E of X has an upper bound $\text{Sup } E$.*

DEFINITION A.3 (Continuous functions). *A function $f : X \rightarrow Y$ where X and Y are complete posets is continuous if for every directed subset E of X , we have: $f(\text{Sup } E) = \text{Sup } f(E)$.*

THEOREM A.4 (Tarski). *If f is a continuous endofunction (i.e. $f : X \rightarrow X$ where X is a complete poset) then it has a least fixpoint noted fix_f .*

A.2. Proof of Theorem 5.18

This is easy to see that given any set X , the set 2^X equipped with the subset relation as partial order is complete. Indeed, 2^X has \emptyset as least element and for any directed subset E , $\text{Sup } E = \bigcup_{e \in E} e$. Hence, By the Tarski's theorem, it is sufficient to show that fill is continuous.

LEMMA A.5. *fill is continuous*

Proof. Let E be a subset of $2^{\mathbb{A}}$. As $2^{\mathbb{A}}$ is a complete poset, the set E has an upper bound $\text{Sup } E$. Obviously, fill is monotone. Thus, we have: $\forall e \in E, e \subseteq \text{Sup } E$. Since fill is continuous, and $\forall E \subseteq \mathbb{A}, E \cup [P]_{\rho_{\mathbb{B}}} \subseteq \mathbb{A}$, fill has a least fixpoint fix_{fill} . This least fixpoint is defined as the limit (i.e. the least upper bound) of the set $\{\text{fill}^n(\emptyset) \mid n \geq 0\}$ (i.e. $\text{fix}_{\text{fill}} = \text{Sup } \{\text{fill}^n(\emptyset) \mid n \geq 0\}$). This describes the general outline of the algorithm. The program generated from this scheme is detailed in Section 6. ■

By definition, this algorithm stops whatever the program and the array in input are, since E is bounded by the number of cells contained in the array. Moreover, the number of steps is bounded by the cardinality of the iteration space.

A.3. Proof of Theorem 5.19

This is proven by induction on the size of the sequence $(\rho_{\mathbb{C}}^1, \dots, \rho_{\mathbb{C}}^n)$ resulting of the interpretation of P in \mathcal{C} . By definition, $E_{\rho_{\mathbb{C}}} = E_{\rho_{\mathbb{B}}}$ where $\rho_{\mathbb{B}}$ is compatible with $\rho_{\mathbb{C}}$. Then, by Definition 5.17 we have: $E_{\rho_{\mathbb{C}}} \subseteq \text{fix}_{\text{fill}}$. Let e be an entry of $E_{\rho_{\mathbb{C}}^{i+1}}$. Here, two cases have to be considered:

- There exists $j \leq i$ such that $e \in E_{\rho_{\mathbb{C}}^j}$. The conclusion is obvious;

- Otherwise, this means that there exists an assignment of the form:

$$A[f(I_1, \dots, I_d)] = \mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)])$$

and a tuple (i_1, \dots, i_d) of the iteration space so that:

- $\rho_c^i(I_j) = i_j$ for all $1 \leq j \leq d$;
- $\llbracket \mathcal{G}(A[g_1(I_1, \dots, I_d)], \dots, A[g_m(I_1, \dots, I_d)]) \rrbracket_{\rho_c^i} \neq 0$.

Consequently, the tuple (i_1, \dots, i_d) denotes a significant iteration and then $f(i_1, \dots, i_d)$ belongs to fix_{fill} .

A.4. proof of Theorem 5.26

This is proven by following the same induction step than in Definition 5.25. Here, we only give the proof for the case where exp is of the form $exp_1 \otimes exp_2$. The others cases are handled in the same way.

By Definition 5.23, this means that there are both environments ρ_c and ρ'_c with ρ'_c x -equivalent to ρ_c such that:

$$\llbracket exp_1 \otimes exp_2[exp'/x] \rrbracket_{\rho'_c} \neq \llbracket exp_1 \otimes exp_2 \rrbracket_{\rho_c}$$

Therefore, we have $\llbracket exp_1[exp'/x] \rrbracket_{\rho'_c} \neq \llbracket exp_1 \rrbracket_{\rho_c}$ or $\llbracket exp_2[exp'/x] \rrbracket_{\rho'_c} \neq \llbracket exp_2 \rrbracket_{\rho_c}$. Consequently, by the induction hypothesis, we conclude:

$$\bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_1, exp') \vee \bar{\mathcal{S}}_{(i_1, \dots, i_d)}(exp_2, exp')$$

Finally, from the inequality just above, we have:

$$(\llbracket exp_1 \otimes exp_2[exp'/x] \rrbracket_{\rho'_c} \neq 0) \vee (\llbracket exp_1 \otimes exp_2 \rrbracket_{\rho_c} \neq 0)$$

From which we directly conclude that:

$$(\llbracket exp_1 \otimes exp_2[exp'/x] \rrbracket_{\rho'_c} \delta true) \wedge (\llbracket exp_1 \otimes exp_2 \rrbracket_{\rho_c} \delta true)$$

Consequently, for any $\rho_{\mathbb{B}}$ such that $E_{\rho_{\mathbb{B}}} = fix_{fill}$ and for every $j \in \{1, \dots, d\}$ $\rho_{\mathbb{B}}(I_j) = i_j$ we obtain: $\llbracket exp_1 \otimes exp_2 \rrbracket_{\rho_{\mathbb{B}}} = true$.

References

- [1] R. Adle : “*Outils de parallélisation automatique des programmes denses pour les structures creuses*”. PhD’s thesis, University of Évry, 1999. In French. available at <ftp.lami.univ-evry.fr/pub/publications/these/PhD0199.ps.gz>
- [2] M. Ujaldon, E. L. Zapata, B. M. Chapman, H. P. Zima “*Vienna-Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation.*” IEEE Transactions on Parallel and Distributed Systems Vol 8 N 10: 1068-1083, 1997.
- [3] R. Adle and F. Delaplace : “*Extension of the Dependence Analysis for Sparse Computation*”. Technical Report, LaMI-22-97, University of Évry, 1997. available at <ftp.lami.univ-evry.fr/pub/publications/reports/1997/index.html/lami.22.ps.fz>

- [4] U. Banerjee : “*Dependence Analysis for Supercomputing*”. Kluwer Academic Publisher, Vol. 60, 1988.
- [5] A. Bernstein : “*Analysis for parallel Processing*”. In: Proc. IEEE Transactions on Electronic Computers, Vol. EC-15, No. 5, 1966.
- [6] A. Bik and H. Wijshoff : “*Advanced compiler optimizations for sparse computations*”. J. Par. Dist. Comp., 31:109-126. Academic Press, 1995.
- [7] A. Bik and H. Wijshoff : “*Automatic Data Structure Selection and Transformation for Sparse Matrix Computation*”. In: Proc. IEEE Transactions on Parallel and Distributed Systems, 7(2):109-126, 1996.
- [8] P. Cousot and R. Cousot : “*Abstract interpretation and application to logic programs*”. J. Logic Prog., 13(2-3):103-179. Amsterdam: Elsevier, 1992.
- [9] I. Duff, A. Erisman and J. Reid : “*Direct Methods for Sparse Matrices*”. Oxford Sciences Publications, 1986.
- [10] P. Feautrier : “*Techniques de parallisation*”. In: M. Cosnard, M. Nivat and Y. Robert (eds), Algorithmique parallèle, pp. 243-257. Masson, 1992.
- [11] C. Fu, and T. Yang : “*Run-time Techniques for Exploiting Irregular Task Parallelism on Distributed Memory Architecture*”. J. Par. Dist. Comp., 42:143-156. Academic Press, 1997.
- [12] A. Gupta, G. Karypis and V. Kumar : “*Highly Scalable Parallel Algorithms for Sparse Matrix Factorization*. In: Proc. IEEE Transactions on Parallel and Distributed Systems, 8(5):502-520, 1997.
- [13] M. Heath, E. Ng and B. Peyton : “*Parallel Algorithm for Sparse Linear Systems*”. Siam Review, 33(3):420-460, 1991.
- [14] V. Kotlyar, K. Pingali and P. Stoghill : “*Compiling Parallel Code for Sparse Matrix Applications*. In: Proc. SuperComputing (SC), ACM/IEEE, November 1997.
- [15] J.-C. Mitchell : “*Foundations for Programming Languages*”. Foundations of Computing. Boston: MIT Press, 1996.
- [16] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan and Jeff McDonald : “*Parallel Programming in OpenMP*”. Morgan Kaufmann Publishers. 2000.
- [17] W. Pugh and D. Wonnacott : “*An Exact Method for Analysis of Value-based Data Dependences*”. In: Proc. Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, 1993.
- [18] J. Saltz, R. Mirchandaney and K. Crowley : “*Run-Time Parallelization and Scheduling of Loops*”. In: IEEE Transaction on Computer, 40(5):603-611, 1991.
- [19] P. Stodghill : “*A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*”. PhD thesis, Cornell University, 1997.
- [20] M. Wolfe : “*Optimizing Supercompilers for Supercomputers*”. MIT Press, 1989.

- [21] M. Wolfe : “*High Performance Compilers for Parallel Computing*”. Addison Wesley, 1996.
- [22] T. Yang and C. Fu : “*Space/Time-Efficient Scheduling and Execution of Parallel Irregular Computations*”. ACM Transactions on Programming Languages and Systems (TOPLAS), 1999.