Automatic parallelization of sparse matrix computations: a static analysis

Roxane Adle, Marc Aiguier, and Franck Delaplace

Université d’Évry Val d’Essonne, CNRS EP738, LaMI, F-91025 Évry Cedex, France
{adle,aiguier,delaplace}@lami.univ-evry.fr
fax number: 33 (+1) 69 47 74 72

Abstract. This article deals with the definition of a new method for the automatic parallelisation of sequential programs working on dense matrices for generating a parallel counterpart working on sparse matrices. Parallelisation of numerical programs dealing with sparse matrices implies a symbolic analysis for defining memory locations where the computation takes place. This symbolic analysis allows us to define a specific dependence analysis which refines the usual Bernstein’s conditions. Thus, it provides more parallelism than analysis strictly based on the Bernstein’s conditions due to the “sparsity” of matrices.

Keywords. fill-in, non-standard semantics, sparse dependence analysis, Bernstein’s conditions.

Introduction

Numerical applications using sparse matrices are ubiquitous in science and engineering such as dynamic fluids or mechanical structure computations. Parallel programs dealing with sparse matrices are considered to be error-prone, hard to conceive and difficult to support. Thus, it is important to develop restructuring compilers to automatically transform numerical programs into equivalent ones making sparse matrix computations. In this way, two works have mainly been proposed [4, 5, 9, 11]. In [4, 5], 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 the compiler Bernoully [11], P. Stoghill uses a generalization of both data structures CCS and CRS, called CHS (Compressed Hyperplane Storage), for storing sparse matrices. The semantics of programs is given using relational algebras. Thus, program transformations are formalized using this formalism. However, both works are mainly focused on the automatic conversion of sequential dense programs into semantically equivalent sequential sparse codes. Whereas, numerical programs have very important computational times. Thus, it is interesting to define a framework for automatically extracting the parallelism of such programs. In this paper, we propose the definition of a new method for the automatic parallelisation of sequential programs working on dense matrices for generating a parallel counterpart working on sparse matrices.

A sparse matrix contains many zero elements. This leads to define dedicated sparse storage formats to discard zero elements. However, it is not so straightforward to parallelize a program working on sparse storage formats. Indeed, programs using sparse storage format involves indirect addressing which inhibit symbolic analysis [7]. Thus, in order to parallelize programs with a dense data structure but operating on sparse matrices, we have to analyse dependencies by using the dense data structure. Finally, from the dependence graph computed, we deduce the parallel program. The main idea of our approach is to symbolically compute dependencies
from both the text program and the matrix in input by using the sparsity of matrices. Indeed, the sparsity of matrices leads to get more parallelism than dense matrices. This computation is split up into two steps:

1. computation of entries, that is to say, positions in the matrix the content of which will become different from zero in the course of the numerical execution.
2. computation of iteration dependencies for generating the dependence graph.

Here, we will use the previous step to refine usual dependence tests essentially based on the Bernstein’s conditions [3].

The compilation line can be sketched as follows:

![Compilation Line Diagram]

This article is focused on the top part of this compilation line. Then, the paper is organized as follows: in Section 2, we define the filling function which from a program and a matrix in input computes the new entries. In Section 3, we generate the iteration sparse dependence graph by using results of the filling function defined in the previous section. Section ?? is mainly devoted to succinctly present some experimental results with comments. Finally, we leave the recapitulation to Section 5.

By lack of space, no proof of theorems is given in this paper. However, all these proofs can be found in the preliminary version of this paper [2].

1 Working context

For the sake of simplicity, we reduce the analysis to one array for storing the matrix in input. As far as we know, no problem occurs to extend the scope of the analysis to several arrays. Thus, we suppose to have in input of our compilation line a sequential program the form of which is inductively generated from assignments of form both $v = exp$ and $A[exp_1, \ldots, exp_c] = exp$ where $v$ is a scalar variable, $A$ is an array variable, $exp_i$ are integer expressions $1 \leq i \leq c$ and $exp$ is an expression, and using the sequence operator, both conditional and Do-loop constructions as generators.

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 $A[exp_1, \ldots, exp_c] = exp$ contained in a Do-loop nest within the program $P$. Thus, afterwards we will consider the following generic form of these assignments:

$$A[i_1, \ldots, i_d] = \mathcal{G}(A[g_1(i_1, \ldots, i_d)], \ldots, A[g_m(i_1, \ldots, i_d)])$$

so that $A$ is the array of values in $\mathcal{C}$ (the domain of the concrete semantics) used to stock the dense matrix, $f : \mathbb{Z}^d \rightarrow \mathbb{Z}^c$ (resp. each $g_p : \mathbb{Z}^d \rightarrow \mathbb{Z}^c$ for every $1 \leq p \leq m$) is an affine application yielding the index of a memory cell by writing (resp. by reading) from an iteration $(i_1, \ldots, i_d) \in \mathbb{Z}^d$, and $\mathcal{G} : \mathbb{C}^m \rightarrow \mathcal{C}$ stands for an application without side-effect.

More precisely, the application $\mathcal{G}$ is the semantical meaning in the standard interpretation $\mathcal{C}$ of the numerical expression $\mathcal{G}(A[g_1(i_1, \ldots, i_d)], \ldots, A[g_m(i_1, \ldots, i_d)])$ inductively generated from the following set of numerical operators:

---

$^1$ Otherwise, there are not dependencies between iterations. Thus, there is not interest anymore to generate the filling function.
operators Definition

| c   | constant          |
| v   | variable          |
| ⊗   | binary operator so that 0 is absorbing |
| ⊕   | binary operator so that 0 is neutral at left and/or at right |
| μ   | unary operator or function for which 0 is a fixed point (e.g. square root) |
| ⊕, μ| both binary and unary operators or functions the behaviour of which depends on arguments (e.g. the randomize function) |

We note $\mathit{Expr}$ the whole set of numerical expressions as described above. Finally, we note $\mathit{Prog}$ the whole set of well-formed programs which contains at least a DO-loop nest with a statement of the form: $A[i_1, \ldots, i_s] = 0(A[j_1, \ldots, j_s], \ldots, A[g_m(i_1, \ldots, i_s)]).$

2 Symbolic analysis

Herein, we describe how to generate at compile-time a symbolic program, called filling function, for a given numerical program. The goal of this symbolic program is to compute the fill-in introduced during the numerical computation. The fill-in deals with situations in which zero elements become nonzero. Well-known applications such as sparse Cholesky factorization use such symbolic analysis [8]. The difference from a program to another relies upon the definition of the "filling function" which is unique for a given a program.

As we propose to generate the filling function at compile-time, the fill-in have to be derived from both the program text and the dense matrix in input. Usually, to statically collect dynamic informations about programs, it is natural to use a non-standard semantics of the programming language [6]. The interest of such semantics is to abstract away from irrelevant matters by giving conservative approximations of the concrete behaviours of programs.

To compute the fill-in, we will use the elementary abstract interpretation theory by reinterpreting numerical expressions in the abstract domain $\mathbb{B} = \{ \mathit{true}, \mathit{false} \}$ provided with the usual propositional connectors (principally $\land$ and $\lor$). Roughly speaking, given an assignment $A[\mathit{exp}_1, \ldots, \mathit{exp}_s] = \mathit{exp}$, "true" will mean that the evaluation of $\mathit{exp}$ in the concrete semantics yields a value different of 0. Thus, the expression $\mathit{exp}_1, \ldots, \mathit{exp}_s$ will denote an entry, that is to say, an index the content of which is different of 0 in the course of the numerical execution. Succinctly, the idea is to use this abstraction to define an endofunction (the filling function) directly from the index space (not anymore from the iteration space of the program under analysis). Thus, we abstract ourselves of numerical execution. Then, we will be able to statically generate the set of new entries.

Afterwards, we choose to give as running example a simplified version of the Cholesky factorization algorithm. We write the associated program as follows:

```plaintext
SPARE, REAL : a(i,j,k)  
  b_1  do (j=1, n)  
    b_2  do (k=1, j)  
  s    a(i,j) = a(i,j) - a(i,k) * a(j,k)  
  enddo  
  enddo
```

The statement $s$ belongs to a triple loop. Thus, the application $G : \mathbb{R}^3 \to \mathbb{R}$ is defined by: $(x, y, z) \mapsto x - y \cdot z$. Finally, the affine functions $f$, $g_1$, $g_2$, and $g_3$ from $\mathbb{N}^3$ to $\mathbb{N}^2$ are respectively defined by: $(i, j, k) \mapsto (i, j)$, $(i, j, k) \mapsto (i, j)$, $(i, j, k) \mapsto (i, k)$, and $(i, j, k) \mapsto (j, k)$.

2.1 Abstraction domain

**Notation 1.** Given a $c$-dimensional matrix, there exists a tuple $(m_1, \ldots, m_c) \in (\mathbb{N}^+)^c$ so that the underlying array $A$ used to stock it is of size $(m_1 \times \ldots \times m_c)$. We note $\mathcal{A}$, so-called the index space of $A$, the set $\{0, \ldots, m_1-1\} \times \ldots \times \{0, \ldots, m_c-1\}$. 
Definition 1. Let $\mathcal{C}$ be the domain where the standard interpretation of numerical expressions is defined (e.g., natural numbers $\mathbb{N}$, integers $\mathbb{Z}$, real numbers $\mathbb{R}$, etc.). We defined the abstraction relation $\delta \subseteq \mathcal{C} \times \mathbb{B}$ by: $\delta = \{(0, \text{true}), (0, \text{false})\} \cup \{(x, \text{true}) \mid x \neq 0\}$.

Remark. Understandably, $\text{exp} \delta \text{ true}$ means that the expression $\text{exp}$ provides a value which differs from zero. In this context, we can notice that zero is both linked with $\text{true}$ and $\text{false}$. This comes from the fact that some statements have their behaviour which is strongly dependent on the execution. For instance, facing an assignment of the form $A[f(I_1, \ldots , I_d)] = v$ where $v$ is a scalar variable, we cannot statically deduce if the index denoted by the expression $f(I_1, \ldots , I_d)$ will be an entry or not. It depends on the value that $v$ will have. Then, it is sensible to consider that the value of $v$ is always different of 0.

As usual, expressions are evaluated from environments. Thus, given any domain $\mathcal{D}$, an environment $\rho_{\mathcal{D}}$ will associate the array $A$ with an element of $[\mathbb{A} \rightarrow \mathcal{D}]$, a variable $v$ with an element of $\mathcal{D}$, and each iteration indice $I$ with an element of its iteration space.

Definition 2. Given an environment $\rho_{\mathcal{B}}$ and a numerical expression $\text{exp}$ of $\text{Expr}$, we note $[\text{exp}]_{\rho_{\mathcal{B}}}$ the interpretation of $\text{exp}$ in $\mathbb{B}$ inductively defined by the following rules:

- $[c]_{\rho_{\mathcal{B}}} = (c \neq 0), [c]_{\rho_{\mathcal{B}}} = \text{true},$ and $[A]_{\rho_{\mathcal{B}}}(I_1, \ldots , I_d)]_{\rho_{\mathcal{B}}} = \rho_{\mathcal{A}}(A)[\rho_{\mathcal{B}}(I_1), \ldots , \rho_{\mathcal{B}}(I_d))$.
- $[\text{exp}_1 \circ \text{exp}_2]_{\rho_{\mathcal{B}}} = [\text{exp}_1]_{\rho_{\mathcal{B}}} \circ [\text{exp}_2]_{\rho_{\mathcal{B}}}.$
- $[\text{exp}_1 \circ \text{exp}_2]_{\rho_{\mathcal{B}}} = [\text{exp}_1]_{\rho_{\mathcal{B}}} \circ [\text{exp}_2]_{\rho_{\mathcal{B}}}.$
- $[\mu(\text{exp})]_{\rho_{\mathcal{B}}} = [\text{exp}]_{\rho_{\mathcal{B}}}.$
- $[\rho(\text{exp})]_{\rho_{\mathcal{B}}} = \text{true}.$
- $[\text{exp}_1 \circ \text{exp}_2]_{\rho_{\mathcal{B}}} = \text{true}.$

Remark. By Definition 2, both operations $\mu$ and $\circ$ have to be interpreted as the constant function defined from $\mathbb{B} \times \mathbb{B}$ to $\mathbb{B}$ by: $(x, y) \mapsto \text{true}$.

Notation 2. Given an environment $\rho_{\mathcal{C}}$ for the standard domain $\mathcal{C}$, we note $\rho_{\mathcal{D}}$ every environment so that for every variable $v$ we have: $\rho_{\mathcal{C}}(v) \delta_{\rho_{\mathcal{D}}}(v)$ ($\delta$ being a relation, $\rho_{\mathcal{C}}$ is not unique). We say that $\rho_{\mathcal{D}}$ is compatible with $\rho_{\mathcal{C}}$.

Proposition 3. For every $\rho_{\mathcal{C}}$ and every $\rho_{\mathcal{D}}$ compatible with $\rho_{\mathcal{C}}$, we have: $[\text{exp}]_{\rho_{\mathcal{C}}} \delta [\text{exp}]_{\rho_{\mathcal{D}}}.$

Proposition 3 establishes the correctness of the abstract interpretation.

2.2 Calculation of the filling function

In this section, by using the abstraction interpretation given in Section 2.1 we show how to statically generate the whole set of new entries. To reach this purpose, the idea is not to iterate anymore on iterations of DO-loop nests but on the entries themselves for generating new ones.

Notation 3. 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 4. Given an environment $\rho_{\mathcal{B}}$ and a program $P$ of $\text{Prog}$, we note $[P]_{\rho_{\mathcal{B}}}$ the subset of $\mathbb{A}$ inductively defined by the following rules:

$^2$ Given two sets $N$ et $M$, the notation $[N \rightarrow M]$ denotes the whole set of applications from $N$ to $M$. 
\[-[v = \exp[\rho_0] = \emptyset.\]
\[-[A[f(I_1, \ldots, I_d)]] = \exp[\rho_0] \text{ is the set of entries } e \text{ so that for each cell, there exists a tuple } (i_1, \ldots, i_d) \text{ of the iteration space so that both following conditions hold:}\]
\[\bullet \; e = f(i_1, \ldots, i_d),\]
\[\bullet \; \text{for the environment } \rho_o \text{ 's } I_j \text{-equivalent to } \rho_0 \text{ with } \rho_0'_{j}(I_j) = i_j \text{ for every } j = 1, \ldots, d, \text{ we have: } \exp[\rho_0']_{j} = \text{true}.\]
\[-[S_1; S_2][\rho_0] = [S_1][\rho_0] \cup [S_2][\rho_0].\]
\[-[\text{if } \exp \text{ then } S_1 \text{ else } S_2][\rho_0] = [S_1][\rho_0] \cup [S_2][\rho_0].\]
\[-[do \; (i = P, Q) \; S][\rho_0] = [S][\rho_0].\]

**Notation 4.** Given any domain \( D \) and an environment \( \rho_D \), we note \( E_{\rho_D} \) the subset of \( \mathcal{A} \) defined by: \( E_{\rho_D} = \{ e \mid \rho_D(A)(e) = \text{true} \}. \)

**Example 1.** From the statement \( s_1 \) of the Cholesky algorithm and a given environment \( \rho_B \), we obtain for \( [A(i, j) = A(i, j) - A(i, k) \cdot A(j, k)]_{\rho_0} \) the following set of entries:
\[
\{(i, j) \exists (j, k, i) \in \mathbb{Z}^2, \exists (x_0, y_0) \in E_{\rho_0}, \\
\exists (x_1, y_1) \in E_{\rho_0}, \exists (x_2, y_2) \in E_{\rho_0}, \\
((i, j) \not\in E_{\rho_0}) \wedge (1 \leq j \leq N) \wedge (1 \leq k \leq j) \wedge (j \leq i \leq N) \\
\wedge ((x_0 = i) \wedge (y_0 = j)) \lor ((x_1 = i \wedge y_1 = k \wedge x_2 = j \wedge y_2 = k)) \}
\]

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 \( [A(i, j) = A(i, j) - A(i, k) \cdot A(j, k)]_{\rho_0} \) can be simplified as follows:
\[
\{(x_1, x_2) \exists (x_1, y_1) \in E_{\rho_0}, \exists (x_2, y_2) \in E_{\rho_0}, \\
(x_1, x_2) \not\in E_{\rho_0} \wedge y_1 = y_2 \wedge x_1 < x_2 \leq x_1 \}
\]

Such simplifications are automatically performed by using symbolic computation tools as Omega [10].

According to Definition 4, the difficulty lies in the calculation of iterations the execution of which yield new entries, called significant iterations. As usual, we solve it by generating the set \( [S][\rho_0] \) by solving a family of Diophantine equation systems with variables in the iteration space (a complete definition is given in [2]). Thus, given an assignment \( S \) of the form \( A[f(I_1, \ldots, I_d)] = \exp, \) for each solution \( (i_1, \ldots, i_d) \) of the associated Diophantine equation systems, \( f(i_1, \ldots, i_d) \) denotes a new entry.

**Definition 5.** With the previous notations, from a program \( P \) and an environment \( \rho_0 \), which denotes the initial environment, we note \( \text{fill} : 2^{\mathcal{A}} \rightarrow 2^{\mathcal{A}} \) where \( 2^{\mathcal{A}} \) is the set of all subsets of \( \mathcal{A} \) (i.e. \( 2^{\mathcal{A}} = \{ X \mid X \subseteq \mathcal{A} \} \)), the application defined by: \( \emptyset \rightarrow E_{\rho_0} \) and \( E \rightarrow E \cup [P][\rho_0], \) where \( \rho_0' \) is any environment so that \( E_{\rho_0'} = E. \)

To show that this application fully describes an algorithm, we use a classical result of set theory: The Tarski’s theorem. Indeed, \( (2^{\mathcal{A}}, \subseteq) \) is a complete partial order (\( \emptyset \) is the least element and for any directed subset \( E \), the upper bound \( \text{Sup} E = \bigcup_{e \in E} e \)).

Moreover, \( \text{fill} \) is obviously monotone. Then, \( \text{fill} \) is continuous (indeed, we have: \( \forall e \in E, e \subseteq \text{Sup} E \)). By the Tarski’s theorem, \( \text{fill} \) has a least fixedpoint, usually noted \( \text{fix}_{\text{fill}}. \) Consequently, our algorithm is inductively defined by: \( E^0 = E_{\rho_0} \) with \( E^{n+1} = \text{fill}(E^n) \)

as stop condition: \( E^n = \text{fill}(E^n). \) By Definition 5, this algorithm stops whatever the program and the matrix in input (the worst case is bounded by the cardinality of the iteration space). We still have to show that our algorithm generates all entries as performed by the numerical execution.

**Theorem 6.** Given a program \( P \) of Prog and an environment \( \rho_C \), we have: \( E_{[P][\rho_C]} \subseteq \text{fix}_{\text{fill}}. \)
3 Sparse dependence analysis

3.1 Understanding

Dependence analysis consists of determining the tasks of a program not being able to be performed independently. Thus, two tasks can be performed in parallel if for any order in which they are performed, the result is the same. In general, the problem of computing all dependencies at compile-time is undecidable. However, we have sufficient conditions introduced by Bernstein [3] which ensure us such a result. These conditions consist of verifying that all statements of the program under analysis do not access in the same time at an identical cell memory. Usually, these conditions are split up into three subconditions: flow-dependence, anti-dependence, and output-dependence. Here, due to the lack of space, we are only interested by the most important of them for parallelisation, the flow-dependence (a complete study of Bernstein's conditions is given in [2]).

Herein, tasks represent iterations of DO-loop nests. By following section 1, each DO-loop nest of the program under analysis has the following generic form:

```
  do (I1 = P1, Q1)
    \ldots
    do (I2 = P2, Q2)
      \ldots
      \ldots
    enddo
  \ldots
enddo
```

For such a DO-loop nest, the flow-dependence condition is expressed: let us note \( S(i_1, \ldots, i_d) \) to mean that the assignment \( S \) is performed at the iteration \( (i_1, \ldots, i_d) \). Let \( R(S) \) (resp. \( W(S) \)) be the set of cell memories read (resp. written) by the statement \( S \). Then, for any \( (i_1, \ldots, i_d) \ll (j_1, \ldots, j_d) \) where \( \ll \) denotes the lexicographical order on the iteration space and means the execution order, \( (i_1, \ldots, i_d) \) and \( (j_1, \ldots, j_d) \) are flow-dependent if and only if \( W(S(i_1, \ldots, i_d)) \cap R(S(j_1, \ldots, j_d)) \neq \emptyset \).

3.2 Sparse dependencies for DO-loop nests

We propose to refine the Bernstein's conditions by using the properties of 0 to be absorbing and neutral. To reach this purpose, we will use the abstraction domain defined in section 2.1 as well as the following relation:

**Notation 5.** Given an expression \( exp \) and a subexpression \( exp' \) of \( exp \), we note \( exp | x \) the expression obtained from \( exp \) by substituting all occurrences of \( exp' \) by a fresh variable \( x \).

**Definition 7.** Let \((i_1, \ldots, i_d)\) be an iteration of the iteration space. We note \( S_{(i_1, \ldots, i_d)} \subseteq Expr \times Expr\) the binary relation defined by:

\[
exp S_{(i_1, \ldots, i_d)} exp' \text{ iff } \begin{cases} \text{either } exp' \text{ is not a subterm of } exp, \\
\text{or for every } \rho_c \text{ so that } \{ \\text{ref } (I_j) = i_j, 1 \leq j \leq d \} \\
\{ \text{ref } \rho_c = f \in \text{init} \} \text{ we have: } [exp]_{\rho_c} = 0 \\
[exp | x]_{\rho_c} = [exp]_{\rho_c} \\
\text{for every } \rho_c, \text{ } \rho_c \text{ -equivalent to } \rho_c \text{ otherwise} \end{cases}
\]

Let us suppose that \( exp \) has the form: \( \sigma(A[g_1(I_1, \ldots, I_d)], \ldots, A[g_m(I_1, \ldots, I_d)]) \), and \( exp' \) is of the form \( A[g_p(I_1, \ldots, I_d)] \) where \( 1 \leq p \leq m \). Then, given an iteration \((i_1, \ldots, i_d)\), \( exp S_{(i_1, \ldots, i_d)} exp' \) means that the evaluation of \( exp \) does not depend on \( A[g_p(i_1, \ldots, i_d)] \) whatever its content. For example, this condition holds when we are facing an expression \( A[g_p(i_1, \ldots, i_d)] \odot A[g_{p'}(i_1, \ldots, i_d)] \) with \( p' \neq p \) so that \( g_{p'}(i_1, \ldots, i_d) \) is not an entry (i.e. \( g_{p'}(i_1, \ldots, i_d) \notin \text{ref } f_{\text{init}} \)).
Thus, let us suppose that there exists an iteration \((j_1, \ldots, j_d)\) of the iteration space so that \((i_1, \ldots, i_d) \leq (j_1, \ldots, j_d)\) and \(f(j_1, \ldots, j_d) = g_p(i_1, \ldots, i_d)\) (i.e. the cell memory indexed by \(g_p(i_1, \ldots, i_d)\) was written at the iteration \((j_1, \ldots, j_d)\)).

Given a statement \(S\) of the form: \(A_j[f(i_1, \ldots, i_d)] = g(A[j_1, i_1, \ldots, i_d], \ldots, A[j_m, i_1, \ldots, i_d])\), we have \(A[j_1, i_1, \ldots, i_d] \in R(S(i_1, \ldots, i_d)) \cap W(S(j_1, \ldots, j_d))\). Thus, according to the Bernstein’s condition we have a dependence between both iterations \((i_1, \ldots, i_d)\) and \((j_1, \ldots, j_d)\). However, from \(\exp S_{i_1, \ldots, i_d} \exp'\), we have: \(S(i_1, \ldots, i_d); S(j_1, \ldots, j_d) = S(j_1, \ldots, j_d); S(i_1, \ldots, i_d)\).

We use this property to refine the flow-dependence condition in order to compute the sparse one.

**Definition 8.** With the previous notations, given two iterations \((i_1, \ldots, i_d)\) and \((j_1, \ldots, j_d)\) so that \((i_1, \ldots, i_d) \leq (j_1, \ldots, j_d)\), we have \((i_1, \ldots, i_d)\) is flow-dependent to \((j_1, \ldots, j_d)\), usually noted \((i_1, \ldots, i_d)\delta_f (j_1, \ldots, j_d)\), if:

\[
f(i_1, \ldots, i_d) \in fix_{f_{i_1}} \\
\wedge (\exists \leq p \leq m, f(i_1, \ldots, i_d) = g_p(j_1, \ldots, j_d)) \\
\wedge (g(A[j_1, i_1, \ldots, i_d], \ldots, A[j_m, i_1, \ldots, i_d]), A[g_p(i_1, \ldots, i_d)]) \in S(i_1, \ldots, i_d) \\
\]

Generating sparse dependencies at compile-time requires that the complement of the relation \(S_{i_1, \ldots, i_d}\) with respect to \(\exp \times \exp\) be algorithmically definable. As for the filling function, we need to use the abstract interpretation defined in Section 2.1.

**Definition 9.** For any \((i_1, \ldots, i_d)\), let us note \(\overline{S}_{i_1, \ldots, i_d} : \exp \times \exp \rightarrow \mathbb{B}\) the application inductively defined by:

- \(\overline{S}_{i_1, \ldots, i_d}(\exp, \exp') = \exp'_{\rho_{\exp}}\) where \(\rho_{\exp}\) denotes any environment so that \(E_{\rho_{\exp}} = fix_{f_{i_1}}\) and for every \(j \in \{1, \ldots, d\}\) \(\rho_{\exp}(I_j) = i_j\).

- \(\overline{S}_{i_1, \ldots, i_d}(\exp, \exp') = false\) if \(\exp'\) is not a subterm of \(\exp\).

- \(\overline{S}_{i_1, \ldots, i_d}(\exp_1 \oplus \exp_2, \exp') = [\exp_1 \oplus \exp_2]_{\rho_{\exp}} \wedge (\overline{S}_{i_1, \ldots, i_d}(\exp_1, \exp') \lor \overline{S}_{i_1, \ldots, i_d}(\exp_2, \exp'))\)

where \(\rho_{\exp}\) denotes any environment so that \(E_{\rho_{\exp}} = fix_{f_{i_1}}\) and for every \(j \in \{1, \ldots, d\}\) \(\rho_{\exp}(I_j) = i_j\).

- \(\overline{S}_{i_1, \ldots, i_d}(\exp, \exp') = false\) if \(\exp'\) is not a subterm of \(\exp\).

- \(\overline{S}_{i_1, \ldots, i_d}(\exp_1 \oplus \exp_2, \exp') = \overline{S}_{i_1, \ldots, i_d}(\exp_1, \exp') \lor \overline{S}_{i_1, \ldots, i_d}(\exp_2, \exp')\)

where \(\varnothing \in \{\emptyset, \ominus\}\)

- \(\overline{S}_{i_1, \ldots, i_d}(\varnothing(\exp_1), \exp') = \overline{S}_{i_1, \ldots, i_d}(\exp_1, \exp')\) where \(\varnothing \in \{\mu, \varnothing\}\).

With such an approach, we only give a rough estimate of the complement of \(S_{i_1, \ldots, i_d}\) as it is shown by the following result:

**Theorem 10.** \((\exp, \exp') \notin \overline{S}_{i_1, \ldots, i_d} \Rightarrow \overline{S}_{i_1, \ldots, i_d}(\exp, \exp')\)

From there, we can redefine iteration dependencies in such way that they can be automatically generated. Indeed, it is sufficient to replace in Definition 8 the condition about the relation \(S_{j_1, \ldots, j_d}\) by the following one:

\[
\overline{S}_{j_1, \ldots, j_d}(\emptyset(A[i_1, \ldots, I_d], \ldots, A[j_m, I_1, \ldots, I_d]), A[g_p(I_1, \ldots, I_d)])
\]

**Example 2.** Due to the lack of space, we only present the analysis for the flow-dependencies. Two flow-dependencies are going to be computed from the analysis of the statement \(s_1: (\delta_f'_{1})\) defined from both \(A(i, j)\) and \(A(i', k')\), and \((\delta_f'_{2})\) defined from both \(A(i, j)\) and \(A(j', k')\). The computation of both flow-dependencies \((\delta_f'_{1})\) and \((\delta_f'_{2})\) providing the same characteristic function, we will only give computations for \((\delta_f'_{1})\):
\[ j(\delta_f)_{j'} = \exists (x_1, y_1) \in \text{fix}_{j', k}, \exists (x_2, y_2) \in \text{fix}_{j, k}, \exists (k, i, k', i') \in \mathbb{Z}^4, \]

**Domain by writing** \( 1 \leq j \leq N \wedge 1 \leq k \leq j \wedge 1 \leq i \leq N \wedge \)

**Domain by reading** \( 1 \leq j' \leq N \wedge 1 \leq k' \leq j' \wedge 1 \leq i' \leq N \wedge \)

**Identical references** \( x_1 = i = i' \wedge y_1 = j = k' \wedge \)

**Ordering** \( j < j' \wedge \)

\( \mathbb{F}(j, j', i', i') \)

\( i' = x_1 \wedge k' = y_1 \wedge j' = x_2 \wedge k' = y_2 \)

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

\[ j(\delta_f)_{j'} = \exists (x_1, y_1) \in \text{fix}_{j', k}, \exists (x_2, y_2) \in \text{fix}_{j, k}, \]

\[ j = y_2 \wedge j' = x_2 \wedge y_2 = y_1 \wedge y_2 < x_2 \leq x_1 \]

### 4 Hints on the sparse code generation and experiments

#### 4.1 Code generation

The code generation is not addressed in this article (see [1] for this purpose). However, a code generation has been done which converts any dense sequential program to an SPMD sparse program for share memory architecture. Roughly speaking, this code generation is split up into two parts:

1. The code generation of the symbolic program where we take the references of the non-zero entries of the matrix, the \textit{fill} function and the dependence relation \( \delta_f \). The aim of this program will be to compute the set of final entries as well as the dependence graph.

2. The generation of the numerical parallel sparse program where we compute the scheduling of fronts \(^3\) from dependences previously defined. Then, from this scheduling, we generate a mid-sparse parallel program \(^4\) by removing insignificant iterations. Finally, this mid-sparse program is restructured to provide a sparse code, by using the MT1 compiler.

#### 4.2 Experiments

From the “generic” method presented in this paper, two experiments have been done. Due to the lack of space, we are going to succinctly present them by describing comparison on performances (more details on these experiments can be found in [1]).

The first experiment concerns the Cholesky factorization algorithm. This algorithm has been chosen because there exists a had-oc method primary studied in [8], the parallelism of which is based on “sparse dependences”. The second experiment has been done to show the capability of our method to obtain a speed-up by computing the sparse dependence analysis for other programs.

The experiments made on the Cholesky factorization are focussed on the comparison of both sequential symbolic factorization and parallel fix-point computation. The main difference with the ad-hoc method (in addition that our method is not only devoted to be applied on this algorithm) is the use of a parallel fix-point computation. The parallel fix-point computation ran on an SP2, 8 nodes, 64Mo per node. The used sparse format is a coordinate scheme format [7]. For the sequential computation (1 node), the standard symbolic factorization has been used. Table 1 describes several speed-up obtained from a variety of matrices only coming from the Harwell boeing collection. Two remarks can be done:

1. a moderate speed-up (2 in average),

\(^3\) A \textit{front} is the set of iterations which can be independently performed at a same instant.

\(^4\) “mid-sparse” meaning that the scheduling has taken into account the sparsity of the matrix but the data structure used to stock it is always dense.
2. the parallel version offer the ability to compute the symbolic factorization for large matrices in parallel. For example, the computation of bcsstk30 has not be performed for one and two nodes due to the lack memory space.

In Table 1, only the time computation is reported.

<table>
<thead>
<tr>
<th>N. Proc</th>
<th>s3rm3m3</th>
<th>bcsstk14</th>
<th>bcsstk27</th>
<th>11388bus</th>
<th>bcsstk30</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.2</td>
<td>2.2</td>
<td>1.5</td>
<td>1.1</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2.4</td>
<td>2.8</td>
<td>1.2</td>
<td>0.8 sec</td>
</tr>
<tr>
<td>8</td>
<td>2.7</td>
<td>2.1</td>
<td>3.8</td>
<td>2</td>
<td>0.8 sec</td>
</tr>
</tbody>
</table>

**Table 1.** Speed-up obtained for the Cholesky symbolic factorization

Second experiments report the application of the analysis to the following program:

```fortran
DO i=2,n
   A(j+1)=(A(j-2)*A(j-1)-A(j))/A(j+1)
ENDDO
```

Usually, only a sequential dense execution is performed for such programs. It has been chosen because the computational part is reduced. Thus, the symbolic computation is time-consuming in comparison of the numerical computation (50% of the overall computation time). 2% of the overall vector is filled which corresponds to a density frequently encontoured for sparse matrices. The figure 4.2 reports a comparison between both sequential and parallel execution including symbolic filling computation, dependence graph computation and parallel execution. From a given size and for low fill density, the results highlight that this method can be a source of effective speedup for sparse “arrays” even though the symbolic cost is an important part of the computation.

![Fig. 1. time comparison between sequential dense execution and sparse parallel execution for the fill density: 1% for the left figure and 2% for the right one](image)

**5 Conclusion**

The research briefly exposed here is only the symbolic part of a complete method for the automatic parallelisation of sequential programs working on dense matrices for generating a parallel counterpart working on sparse matrices [1]. This static part is split up into two steps. In the first step, we define the filling function to symbolically compute the indexes of the matrix in input whose the content will become
different of zero in the course of the execution. To reach this purpose, we use a non-standard semantics of numerical expressions from the propositional calculus. With such an abstraction, we show that the filling function is algorithmically definable. 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 6 establishes the partial correctness of this algorithm. In the second part, we generate the sparse dependence graph. This graph is obtained from iteration dependencies of DO-loop nests. These dependencies are computed by refining the usual Bernstein’s conditions to dynamical concepts. These concepts are that both a memory cell used by two different iterations be 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 not be substitutable by any values (i.e. their contents are important in the numerical expression where they occur).

References


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