Accepted for NSAD 2019 informal proceedings (https://staticanalysis.org/nsad2019/)

Improving the Numerical Accuracy of Parallel Programs by Data Mapping

Farah Benmouhoub[1], Pierre-Loic Garoche[2], and Matthieu Martel[1,3]

[1] LAMPS Laboratory, University of Perpignan, France.
[2] DTIS, ONERA, Toulouse, France.
[3] Numalis, Montpellier, France.
[1]{first.last}@univ-perp.fr,[2]pierre-loic garoche@onera.fr

Abstract. The first objective of parallelization is to speed up the program execution. Typically, a program is split into multiple parts that are computed on different computation cores. A usual approach is to balance the load of each core, splitting the computation evenly among them.
However, when the program performs computations in floating-point arithmetic, we should pay extra attention to their numerical accuracy. Indeed, floating-point numbers are a finite approximation of real numbers, they are therefore prone to accuracy problems due to the accumulated round-off errors. Concerning the numerical accuracy, parallelism introduces additional problems due to the order of operations between several computation units.
Rather than focusing on balancing the load, we focus here on a proper split of the problem driven by the numerical accuracy of the computation. In this paper, we describe a new technique that relies on static analysis by abstract interpretation, and which aims at improving the numerical accuracy of computations by dividing the problem, between computation units, according to the order of magnitude of data.

Keywords: Numerical accuracy, Static analysis, Mapping, scientific computing.

1 Introduction

Scientific computing is typically performed with floating-point arithmetics as defined by the $IEEE754$ standard [1, 13] and therefore sensitive to associated errors; and this problem tends to increase with parallelism. To cope with this issue, we aim at improving the accuracy of computation [3] using a new technique based on static analysis by abstract interpretation. In floating-point computations, in addition to rounding errors, the computations order may also affect the accuracy of the results. Indeed, as illustrated in Figure 1, for the same mathematical expression, eg. a sum, different summation algorithms can be applied, i.e. with different ordering of the computations.

To better illustrate, we can take an example of calculating in $IEEE754$ single precision (binary 32) the sum of three values x, y and z, where $x = 10^9$, $y = -10^9$ et $z = 10^{-9}$, we obtain:

$$((x + y) + z) = ((10^9 - 10^9) + 10^{-9}) = 10^{-9} \quad (1)$$

$$(x + (y + z)) = (10^9 + (-10^9 + 10^{-9})) = 0 \tag{2}$$

The equality as well as addition operator represent here the operations performed with floating point arithmetics. We note that, for the same values of x, y and z, and for the same arithmetic operation, we obtain two different results because of parsing the three values differently. In that specific case, an *absorption* occurred: a small value vanished when added to a large one.

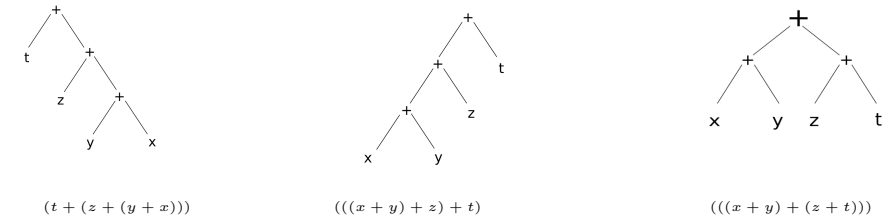

Fig. 1. The different possible sums.

Objective: The key idea of this work is to detect potential ordering of scalars that could lead to an optimization of the numerical accuracy of the computation. In the example above, the lack of accuracy was caused by computation involving scalars of different orders of magnitude. Roughly speaking, in order to keep the computation accurate, one needs to know if the variables can be ordered with respect to their order of magnitude, starting summations with the smallest elements.

Our contribution: We propose to rely on static analysis to detect such arrangements of matrix coefficients: detecting the order of magnitude of each scalar involved in the computation and the ordering (increasing, decreasing, balanced) of sequences of such. Once this ordering is accurately computed, one can choose an appropriate summation algorithm (left to right, right to left, balanced) and obtain more accurate floating point results.

Application: In the case of parallel programs [17], we aim at specializing the code of each process depending on its data, instead on focusing only on load balancing between computation cores. This specialization is based on data mapping [8, 9]. The idea is to assign to each processor sets of data that can be summed together accurately. This corresponds to a given partitioning of the sum. We propose to rely on static analysis to build these partitions before distributing the data among processors. We applied the approach to an iterative method to solve linear systems $Ax = b$. To keep the presentation simple we used the simplest iterative scheme: Jacobi's method.

In this paper, we describe the different phases of our technique. We first rely on abstract interpretation to represent properties of sequences of values

$s = x_1, x_2, \ldots x_n$. The computed properties, called *gradients*, indicate how the elements of s are ordered. Let $grad(s) \in \{\nearrow, \searrow, \rightarrow\}$ be such property, denoting increasing, decreasing, or balanced sequences of values, respectively. By balanced sequences, we mean values of the same magnitude (same exponent in base 10.) rather than constant sequences. Considering the largest sequences with the same gradient property, we can build a partitioning thanks to a greedy algorithm.

More precisely, the first step of our method is to perform a static analysis of the matrix A and the vector x in order to be able to identify the sign and the gradient of the different blocks. After this identification, each block is assigned to a computation unit with an appropriate summation algorithm.

Plan: This article is organized as follows. Section 2 briefly presents Jacobi's method, and means to parallelize it. Section 3 presents our contribution: we detail our technique with its different steps. Section 4 describes the greedy algorithm used to build partitions and the abstraction of the gradient property. Section 5 details our motivational example following by some experimental results obtained during the measurement of the efficiency of our technique. Lastly, in Section 6 we conclude and discuss about future works.

2 Jacobi's Method

2.1 A basic Iterative Algorithm to Solve Linear Systems

The Jacobi method is a well known numerical method used to solve linear systems of n equations with n unknowns. We choose it for its simplicity, and as a first algorithm on which to apply our methodology. In this method, an initial guess, an approximate solution x^0, is selected and is iteratively updated until finding the actual solution x.

In order to explain the idea of the algorithm, let us consider the following system of n linear equations $Ax = b$, where:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \text{ and } \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

The computation of the solution at each iteration is given by Equation (3) below:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1,\ i \neq j}^{n} a_{ij} x_j^k \right). \qquad i = 1, \ldots, n, \quad a_{ii} \neq 0. \tag{3}$$

Note that Jacobi's method is stable whenever the matrix A is strictly diagonally dominant (cf. Equation 4), i.e. on each line, the absolute value of the diagonal term is greater than the sum of absolute values of the other terms:

$$\forall i \in 1, \ldots, n, \qquad |a_{ii}| > \sum_{j \neq i} |a_{ij}|. \tag{4}$$

Our choice for this method is related to the existence of the summation operator, i.e. a sum is performed at each iteration of the algorithm. Because of floating-point numbers, this sum may become wrong because of accumulated errors. For example, it is well known that if we sum the small values with the large ones an absorption may arise and, as a result, a possible lack of accuracy. In order to make end to this problem, we stand for adding the values in increasing order by starting with the small ones. Note that there exists many summation algorithms in the bibliography [15, 16] and that for general arithmetic expressions or numerical algorithms, program transformation improving the accuracy have been proposed [10, 7, 14].

2.2 Related Works in the Parallelization of Jacobi's Method

Several works have been focused on parallelizing Jacobi algorithms, providing different partitioning for different goals.

Luke and Park [12] consider two parallel Jacobi algorithms for computing the singular value decomposition of an $n \times n$ matrix. By relating the algorithms to the cyclic-by-rows Jacobi method, they prove convergence of one of the algorithm for odd values of n and in the general case for the second algorithm.

Zhou and Brent [18] show the importance of sorting columns by norms in each sweep for one-sided Jacobi SVD computation. They describe two parallel Jacobi orderings. These orderings generate $n(n-1)/2$ different index pairs and sort column norms at the same time. The one-sided Jacobi SVD algorithm using these parallel orderings converges in about the same number of sweeps as the sequential cyclic Jacobi algorithm.

Coope and Macklem [4] present how to efficiently use parallel and distributed computing platforms when solving derivative-free optimization problems with the Jacobi algorithm. Convergence is achieved by introducing an elementary trust region subproblem at synchronization steps in the algorithm.

All these works focused on the parallelization of Jacobi's method. Some are focused on the improvement of the convergence of the algorithm, eg. when computing a singular value decomposition. Some others try to make the best use of the parallel architecture performance. However, none of them addressed the issue of the numerical accuracy of computations. All the methods mentioned above are specific algorithms that cannot defined automatically we aim at finding such methods (or probably slightly less efficient ones) automatically.

Our past results [6] show that improving the accuracy of computation also led to acceleration of convergence for iterative algorithms. Our motivation is therefore to parallelize the Jacobi's method focusing first on accuracy and obtaining, as a side result, a better convergence.

3 Abstract Domains

In this section, we introduce the first step of our contribution which consists in detecting increasing, decreasing or balanced patterns in vectors.

Focusing on vectors and the associated increasing, decreasing or balanced patterns, we introduce abstract domains to represent and compute properties over vector sets. Let $\mathbb{R}^n$ be the set of vectors of size n and $\wp(\mathbb{R}^n)$ the powerset of such n-sized vectors. The set $\wp(\mathbb{R}^n)$ is fitted with a partial order, the set inclusion. It is a complete lattice.

Thanks to the framework of Abstract Interpretation [5], we can define different abstractions of this lattice, and combine them to compute properties of sets of n-vectors. In this section, we introduce the various Galois connection that we do to obtain gradients. An overview of our sequence of connections is given in Figure 2. *Sign* abstraction relies on the classical sign domains to detect whether all elements of the vector have the same sign. *Grad*(ient) abstraction is used to represent the increasing, decreasing or balanced nature of the vector scalars. This property is computed over a first abstraction representing values by their floating point exponent, i.e. their order of magnitude. The partially ordered sets Exp^n, $\mathsf{Exp}_D^{\#n}$, Grad are defined in details in sections 3.1, 3.2, 3.3 respectively.

Thanks to the framework of Abstract Interpretation [5], we can define abstractions of this lattice in a modular fashion, combining them to compute properties of sets of n-vectors. In this section, we introduce a set of Galois connections that we will combine to compute our gradients properties. An overview of our sequence of connections is given in Figure 2. The concrete (hence most precise) partially ordered set is the set of n-vectors $\langle \wp(\mathbb{R}^n), \subseteq \rangle$. The *Sign* abstraction relies on the classical sign domains to detect whether all elements of the vector have the same sign. Therefore a positive abstract element denotes the set of n-vectors with only positive scales. The other main abstraction is the *Grad*(ient) abstraction, representing the increasing, decreasing or balanced nature of the vector scalars. As an example , an increase abstract element denotes the set of n-vector with increasing scalars. This gradient abstract domain is defined thanks to the introduction of simpler abstract domains. First, sets of n-vector are abstracted as n-vectors of order of magnitude, associating each scalar by its floating point exponent. Then these abstract vectors are further abstracted to produce the gradient abstraction. The partially ordered sets Exp^n, $\mathsf{Exp}_D^{\#n}$, Grad are defined in details in sections 3.1, 3.2, 3.3, respectively.

3.1 Exp^n: Order of Magnitude of Matrix Elements

Indeed, floating points numbers do not need to be exactly ordered to avoid absorption. We only need to have values that are similar or of *the same order of magnitude*. A first abstract domain represents a set of n-vectors $\wp(\mathbb{R}^n)$ by a vector of a set of exponent $\mathsf{Exp}^n = (\wp(\mathbb{Z}))^n$. Exponents are defined as signed integers where the integer n corresponds to the exponent of the real number it represents, i.e. $\lfloor log_{10}(x) \rfloor$ the integer value of $log_{10}(x)$ rounded towards $-\infty$.

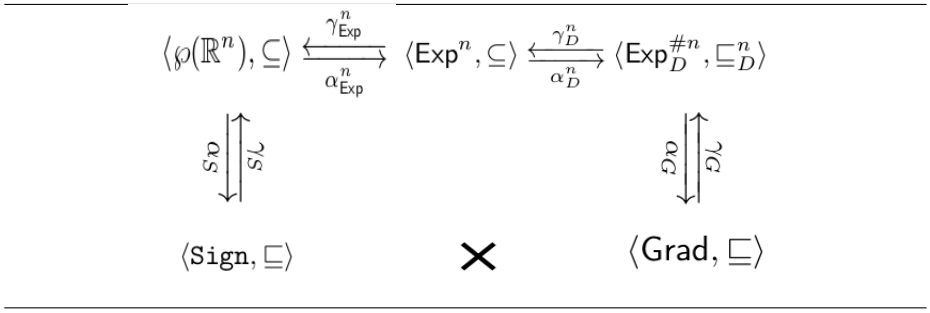

Fig. 2. Global diagram of abstractions.

Let us first formalize this abstraction: the lattice $\langle \wp(\mathbb{R}), \subseteq, \cup, \cap, \emptyset, \mathbb{R} \rangle$ is abstracted by the lattice $\langle \wp(\mathbb{Z}), \subseteq, \cup, \cap, \emptyset, \mathbb{Z} \rangle$. Let $(\alpha_{\mathsf{Exp}}, \gamma_{\mathsf{Exp}})$ be the pair of abstraction and concretization functions defined as:

$$\begin{cases} \alpha_{\mathsf{Exp}} : \wp(\mathbb{R}) \to \wp(\mathbb{Z}) \\ \qquad\qquad X \mapsto \{\log_{10}(x) : x \in X)\} \\ \gamma_{\mathsf{Exp}} : \wp(\mathbb{Z}) \to \wp(\mathbb{R}) \\ \qquad\qquad Y \mapsto \{x \in \mathbb{R} | \log_{10}(x) \in Y)\} \end{cases} \tag{5}$$

Property 1 (Exp *Galois connection).* The pair of $(\alpha_{\mathsf{Exp}}, \gamma_{\mathsf{Exp}})$ is a Galois connection.

$$\langle \wp(\mathbb{R}), \subseteq \rangle \underset{\alpha_{\mathsf{Exp}}}{\overset{\gamma_{\mathsf{Exp}}}{\leftrightarrows}} \langle \mathsf{Exp}, \subseteq \rangle$$

Proof. Both domains are sets and the functions are defined element-wise: they are monotonic. $\alpha_{\mathsf{Exp}} \circ \gamma_{\mathsf{Exp}}(Y) = Y$ is reductive while $\gamma_{\mathsf{Exp}} \circ \alpha_{\mathsf{Exp}}(X) = \{x \in \mathbb{R} | \exists x' \in X, log_{10}(x) = log_{10}(x')\} \supseteq X$ is extensive. □

The abstraction function represents a set of values by the magnitude obtained with a log_{10} function. The concretization function is the associated operation to obtain a Galois connection. As an example, $\alpha_{\mathsf{Exp}}(\{1.10^4, 2.10^4, 3.10^4\}) = \{4\}$ since all these values share the same exponent 4[1].

We can now define the lift of this abstraction to sets of n-vectors in $\wp(\mathbb{R}^n)$. The lattice $\langle \wp(\mathbb{R}^n), \subseteq, \cup, \cap, \emptyset, \mathbb{R} \rangle$ is abstracted by the lattice $\mathsf{Exp}^n = \langle (\wp\mathbb{Z})^n, \subseteq^n, \cup^n, \cap^n, \emptyset^n, \mathbb{Z}^n \rangle$ where $\subseteq^n, \cup^n$, and $\cap^n$ denote the lift of classical set operators to n-vectors. Eg. $\forall x, y \in (\wp\mathbb{Z})^n, x \subseteq^n y$ iff $\forall i \in [1, n], x_i \subseteq y_i$. Similarly $\forall x, y \in (\wp\mathbb{Z})^n, \exists z \in (\wp\mathbb{Z})^n$, st. $z = x \cup^n y$ and $\forall i \in [1, n], z_i = x_i \cup y_i$.

Let us introduce the pair of function $(\alpha^n_{\mathsf{Exp}}, \gamma^n_{\mathsf{Exp}})$:

$$\begin{cases} \alpha^n_{\mathsf{Exp}} : \wp(\mathbb{R}^n) \to (\wp\mathbb{Z})^n \\ \qquad\qquad X \mapsto z \in (\wp\mathbb{Z})^n \text{ s.t. } \forall i \in [1, n], z_i = \alpha_{\mathsf{Exp}}(\{x_i | x \in X\}) \\ \gamma^n_{\mathsf{Exp}} : (\wp\mathbb{Z})^n \to \wp(\mathbb{R}^n) \\ \qquad\qquad z \mapsto \{x \in \mathbb{R}^n | \forall i \in [1, n], x_i \in \gamma_{\mathsf{Exp}}(z_i)\} \end{cases} \tag{6}$$

[1] In practice, since the values are represented by sums of powers or 2, one could have use the binary log $log2$.

Theorem 1 (Exp^n Galois connection). *The pair of* $(\alpha^n_{\mathsf{Exp}}, \gamma^n_{\mathsf{Exp}})$ *is a Galois connection.*

$$\langle \wp(\mathbb{R}^n), \subseteq \rangle \underset{\alpha^n_{\mathsf{Exp}}}{\overset{\gamma^n_{\mathsf{Exp}}}{\leftrightarrows}} \langle \mathit{Exp}^n, \subseteq^n \rangle$$

Proof. The two domains $\wp(\mathbb{R}^n)$ and $\wp(\mathbb{Z})^n$ are sets and the functions are defined element-wise for each vector: they are monotonic. $\alpha^n_{\mathsf{Exp}} \circ \gamma^n_{\mathsf{Exp}}(z) = z$ is reductive while $\gamma^n_{\mathsf{Exp}} \circ \alpha^n_{\mathsf{Exp}}(X) = \{x \in \mathbb{R}^n | \exists x' \in X \quad \forall i \in [1,n], \alpha_{\mathsf{Exp}}(x'_i) = \alpha_{\mathsf{Exp}}(x_i)\} \supseteq X$ is extensive. □

The example below represents the abstraction of a set of vectors $\wp(\mathbb{R}^n)$ by a vector of sets of exponents $\wp(\mathbb{Z})^n$ using the abstract function α^n_{Exp}.

$$\left\{ \begin{pmatrix} 1000 \\ 100 \\ 10 \\ 1 \end{pmatrix}, \begin{pmatrix} 0.001 \\ 1 \\ 0.1 \\ 8 \end{pmatrix} \right\} \overset{\alpha^n_{\mathsf{Exp}}}{\rightarrow} \begin{pmatrix} \{-3,3\} \\ \{0,2\} \\ \{-1,1\} \\ \{1\} \end{pmatrix} \in \wp(\mathbb{Z})^n$$

3.2 $\mathsf{Exp}^{\#n}$: Abstraction of Exponents in Scalar Words

Manipulating vectors of sets is not convenient nor tractable. We need to further abstract these vectors of set of exponents $\mathsf{Exp}^n = (\wp\mathbb{Z})^n$. Since each element of these vectors is a set of integers, one can rely on the state of the art of abstract domains to represent these sets. For example let us take again the vector of sets $v = \begin{pmatrix} \{-3,3\} \\ \{0,2\} \\ \{-1,1\} \\ \{1\} \end{pmatrix}$

of section 3.1. We are going to abstract it into $v^\#$ of abstract elements. Then we need to abstract the sets {-3, 3}, {0, 2}, {-1, 1} and {1}. For this purpose, we are going to use the Kildall or interval domain.

Let $\langle D, \sqsubseteq_D \rangle$ be a sound abstraction of $\langle \wp(\mathbb{Z}), \subseteq \rangle$ associated with a proper Galois connection (α_D, γ_D). We define the set $\mathsf{Exp}^{\#n}_D = D^n$ as the n-vector of D elements. Each operator is the lift of domain D operators to vectors. Eg. $\forall x, y \in \mathsf{Exp}^{\#n}_D, x \sqsubseteq^n_D y$ iff $\forall i \in [1,n], x_i \sqsubseteq_D y_i$. of section 3.1. We are going to abstract it into $v^\#$ of abstract elements. Then we need to abstract the sets {-3, 3}, {0, 2}, {-1, 1} and {1}. For this purpose, we are going to use either the Kildall abstract domain or the interval abstract domain.

Let $\langle D, \sqsubseteq_D \rangle$ be a sound abstraction of $\langle \wp(\mathbb{Z}, \subseteq \rangle$ associated with a proper Galois connection (α_D, γ_D). We define the set $\mathsf{Exp}^{\#n}_D = D^n$ as the n-vector of D elements. Each operator is the lift of domain D operators to vectors. Eg. $\forall x, y \in \mathsf{Exp}^{\#n}_D, x \sqsubseteq^n_D y$ iff $\forall i \in [1,n], x_i \sqsubseteq_D y_i$.

We introduce the following abstraction:

$$\langle \mathsf{Exp}^n, \subseteq \rangle \underset{\alpha^n_D}{\overset{\gamma^n_D}{\leftrightarrows}} \langle \mathsf{Exp}^{\#n}_D, \sqsubseteq^n_D \rangle \tag{7}$$

where $\alpha_D^n(x) = z$ with $\forall i \in [1,n], z_i = \alpha_D(x)$. Similarly, $\gamma_D^n(z) = x \in \mathsf{Exp}^n$ with $\forall i \in [1,n], x_i \subseteq \gamma_D(z_i)$.

Instantiation. We propose to instantiate this abstraction by two basic abstract domains:

1. $\langle K, \sqsubseteq_K \rangle$ Kildall's constants domain [11] with $K = \mathbb{Z} \cup \{\bot, \top\}$
2. $\langle I, \sqsubseteq_I \rangle$ Intervals with $I = ((\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{+\infty\})) \cup \{\bot\}$

Applied to the above example, we obtain:

$$\begin{pmatrix} \{-3,3\} \\ \{0,2\} \\ \{-1,1\} \\ \{1\} \end{pmatrix} \overset{\alpha_K^n}{\rightarrow} \begin{pmatrix} \top \\ \top \\ \top \\ 1 \end{pmatrix} \qquad \begin{pmatrix} \{-3,3\} \\ \{0,2\} \\ \{-1,1\} \\ \{1\} \end{pmatrix} \overset{\alpha_I^n}{\rightarrow} \begin{pmatrix} [-3,3] \\ [0,2] \\ [-1,1] \\ [1,1] \end{pmatrix}$$

3.3 Abstraction in Gradient

To abstract the gradient from data, we define a new comparison relation according to the order of magnitude of the data noted by $\leq^\#$, such that we associate to a set of values $(x_1 \cdot x_2 \cdot .... \cdot x_n)$ one of the following five values $\bot, \nearrow, \searrow, \rightarrow$ and $\top$. We need to be able to compute an abstract *gradient* denoting the monotonic feature of a sequence of elements of D, i.e. $\mathsf{Exp}_D^{\#n}$. In order to detect the increasing properties between abstract value, we need to introduce the abstract operator $\leq_D^\#$: $D \times D \rightarrow \mathbb{B}$. Before introducing instantiations of $\leq_D^\#$ in Equations (9) and (10) for the domains of Section 3.2 we detail the general properties that $\leq_D^\#$ has to satisfy. Note that, in property 2, the order $\leq$ is the usual order on integers.

Property 2 (Soundness of $\leq_D^\#$). A sound abstract binary operator $\leq^\#$: $D \times D \rightarrow \mathbb{B}$ shall satisfy $\forall x^\# \in D, \bot \leq x^\#$ and:

$$\forall x^\#, y^\# \in D \setminus \{\bot\}, \forall x \in \gamma_D(x^\#), y \in \gamma_D(y^\#),$$
$$x^\# \leq_D^\# y^\# \implies x \leq y$$

Let $G = \{\bot, \nearrow, \searrow, \rightarrow, \top\}$ be a set of abstract gradient, fitted with the partial order $\sqsubseteq_G$ with $\forall g \in G, \bot \sqsubseteq_G g, g \sqsubseteq_G \top$ as illustrated in Figure (3).

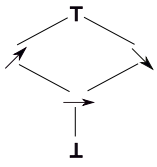

Fig. 3. Lattice of gradients.

Using $\leq_D^{\#}$, we define the abstract gradient $g \in G$, associated to the sequence of $(x_1 \cdot x_2 \cdot .... \cdot x_n) \in D^n$. We introduce the last abstraction:

$$\langle \mathsf{Exp}_D^{\#n}, \subseteq_D^n \rangle \underset{\alpha_G}{\overset{\gamma_G}{\leftrightarrows}} \langle G, \sqsubseteq_G \rangle \tag{8}$$

where

$$\alpha_G(x_1 \cdot x_2 ... \cdot x_n) = \begin{cases} \nearrow & \text{when} \quad x_1 \leq_D^{\#} x_2 \cdot ....\cdot \leq_D^{\#} x_n, \\ \searrow & \text{when} \quad x_n \leq_D^{\#} \cdot .... \cdot x_2 \leq_D^{\#} x_1, \\ \rightarrow & \text{when} \quad x_1 = x_2 \cdot ...\cdot = x_n, \\ \top & \text{otherwise.} \end{cases}$$

and,

- $\gamma_G(\nearrow) = \{(x_1, \ldots, x_n) \in D^n \mid \forall i \in [1, n[, x_i \leq_D^{\#} x_{i+1}\}$.
- $\gamma_G(\searrow) = \{(x_1, \ldots, x_n) \in D^n \mid \forall i \in [1, n[, x_{i+1} \leq_D^{\#} x_i\}$.
- $\gamma_G(\rightarrow) = \{(x_1, \ldots, x_n) \in D^n \mid \forall i \in [1, n[, x_i = x_{i+1}\}$.
- $\gamma_G(\top) = \{(x_1, \ldots, x_n) \in D^n\}$.

Theorem 2 (Exp Galois connection). *The pair of (α_G, γ_G) is a Galois connection.*

$$\langle \mathit{Exp}_D^{\#n}, \sqsubseteq_D^n \rangle \underset{\alpha_G}{\overset{\gamma_G}{\leftrightarrows}} \langle G, \subseteq_G \rangle$$

Proof. Both $\mathsf{Exp}_D^{\#n}$ and G are sets and the functions γ_G and α_G are defined component-wise for each vector: They are monotonic by construction. The composition $\alpha_G \circ \gamma_G(g) = g$ is reductive while $\gamma_G \circ \alpha_G(x_1, \ldots, x_n) = \{(x_1, \ldots, x_n) \in D^n | \exists (x'_1, \ldots, x'_n) \in \mathsf{Exp}_D^{\#n} : \quad \forall i \in [1, n], \alpha_{\mathsf{Exp}_D^{\#n}}(x'_i) = \alpha_{\mathsf{Exp}_D^{\#n}}(x_i)\} \supseteq \mathsf{Exp}_D^{\#n}$ is extensive since $\alpha_{\mathsf{Exp}_D^{\#n}}$ is extensive itself. □

Instantiation. We need to define the operator $\leq_D^{\#}$ for the domains $\langle K, \sqsubseteq_K \rangle$ and $\langle I, \sqsubseteq_I \rangle$.

- For Kildall's constant,

$$x^{\#} \leq_K^{\#} y^{\#} \triangleq \begin{cases} \mathit{true} & \text{if } x^{\#} = \bot \\ \mathit{true} & \text{when } x^{\#}, y^{\#} \in \mathbb{Z}, \gamma_K(x^{\#}) \leq \gamma_K(y^{\#}) \\ \mathit{false} & \text{otherwise} \end{cases} \tag{9}$$

- For interval,

$$x \leq^{\#} y \begin{cases} x = [a, b] \quad y = [c, d] \quad \mathit{and} \quad b \leq c \\ \mathit{or} \quad x = \bot \end{cases} \tag{10}$$

Property 3 (Soundness of $\leq_K^{\#}$). The binary operator $\leq_K^{\#}$: $D \times D \to \mathbb{B}$ satisfies:

$$\forall x^{\#}, y^{\#} \in K, \forall x \in \gamma_K(x^{\#}), y \in \gamma_K(y^{\#}), \\ x^{\#} \leq_K^{\#} y^{\#} \implies x \leq y$$

Proof. Let $x^\#, y^\# \in K$ such that $x^\# \leq_K^\# y^\#$, only three cases are possible:

1. $x^\# = \bot$, $y^\# = i$ for some $i \in \mathbb{Z}$
$$\gamma_K(\bot) = \bot, \gamma_K(i) = \{i\} \quad \textit{and} \quad \bot \subseteq \{i\}.$$
2. $x^\# = i$ for some $i \in \mathbb{Z}$, $y^\# = \top$. In this case:
$$\gamma_K(i) = \{i\}, \gamma_K(\top) = \top \quad \textit{and} \quad \{i\} \subseteq \top.$$
3. $x^\# = \bot$, $y^\# = \top$. Then,
$$\gamma_K(\bot) = \bot, \gamma_K(\top) = \top \quad \textit{and} \quad \bot \subseteq \top.$$

Property 4 (Soundness of $\leq_I^\#$). The binary operator $\leq_I^\#: D \times D \to \mathbb{B}$ satisfies:

$$\forall x^\#, y^\# \in I, \forall x \in \gamma_I(x^\#), y \in \gamma_I(y^\#),$$
$$x^\# \leq_I^\# y^\# \implies x \leq y$$

Proof. The three following cases are similar to the correspond cases in the proof of Property 2:

1. $x^\# = \bot, y^\# = \top$,
2. $x^\# = \bot, y^\# = [a, b]$ for $a, b \in \mathbb{Z}$,
3. $x^\# = [a, b]$ for $a, b \in \mathbb{Z}, y^\# = \top$.

The last case to be considered is when $x^\# = [a, b]$ and $y^\# = [c, d]$. By hypothesis $[a, b] \leq_I^\# [c, d]$, by definition of $\leq_I^\#$ in Equation 10:

$$b \leq c. \tag{11}$$

Let $x \in [a, b], y \in [c, d]$. Consequently $x \leq b$ and $c \leq y$. In addition from Equation (11) we know that $b \leq c$. We conclude that $x \leq b \leq c \leq y$.

We end this section by two examples of a set of two vectors of size 3 containing all the abstractions. The difference between the two examples is the abstraction of exponents in scalar words, such that the first example uses the abstract function α_{K}^n while the second uses the abstract function α_{I}^n.

$$\left\{ \begin{pmatrix} 100 \\ 5 \\ 0.001 \end{pmatrix}, \begin{pmatrix} 0.1 \\ 8 \\ 100 \end{pmatrix} \right\} \overset{\alpha_{\mathsf{Exp}}^n}{\to} \begin{pmatrix} \{-1, 2\} \\ \{1\} \\ \{-3, 2\} \end{pmatrix} \overset{\alpha_{\mathsf{K}}^n}{\to} \begin{pmatrix} \top \\ 1 \\ \top \end{pmatrix} \overset{\alpha_{\mathsf{G}}^n}{\to} \quad \top$$

$$\left\{ \begin{pmatrix} 10 \\ 100 \\ 1000 \end{pmatrix}, \begin{pmatrix} 1 \\ 1000 \\ 10000 \end{pmatrix} \right\} \overset{\alpha_{\mathsf{Exp}}^n}{\to} \begin{pmatrix} \{0, 1\} \\ \{2, 3\} \\ \{3, 5\} \end{pmatrix} \overset{\alpha_{\mathsf{I}}^n}{\to} \begin{pmatrix} [0, 1] \\ [2, 3] \\ [3, 5] \end{pmatrix} \overset{\alpha_{\mathsf{G}}^n}{\to} \quad \nearrow$$

4 Partitioning Data with the Abstract Gradient

We now introduce the next step of our contribution, (1) from the results provided by the static analysis in Section 3, build partition in which we select the appropriate summation arrangement. Then (2) apply our technique on one example coming from realistic problems in mechanics.

4.1 Greedy Algorithm

As mentioned in Section 1, we aim at using the domain G to abstract matrices used as inputs of numerical algorithms. Our goal is to use the information provided by G to determine the best way to perform the sums arising in these algorithms.

More precisely, we want to detect lines, columns or blocks of the matrix that could be efficiently optimized with respect to numerical accuracy.

As described in Section 3, using G, we partition data $x = x_1, x_2, \ldots x_n$ according to their signs and magnitudes. Obviously, there are many ways to partition a matrix in blocks before assigning a gradient to each block. Trying to assign or compute a single gradient for a row or a column may result in the imprecise $\top$ gradient. Therefore we aim at partitioning the matrix, assigning sequence of gradients to rows or columns.

For example, let us consider the vector $x = 1, 2, 3, 4, 4, 3, 2, 1$. While a common gradient shall be $\top$, we aim at assigning it to the sequence $\{\nearrow, \rightarrow, \searrow\}$.

Our objective, for an efficient algorithm, is to compute large blocks with homogeneous gradients. The worst case being a complete partition of the matrix or vector with as many abstract elements as concrete elements.

To build an efficient partition we introduce a greedy algorithm. We will see in our experimental results in Section 5 that this algorithm abstracts a vector of size n by few different gradients. Note that, in future works, for parallelization, we will need blocks of same size. We plan to subdivide the blocks found by our algorithm to the greatest common divisor of the sizes computed for each of them in order to obtain our final partition. Let us also mention that our greedy algorithm is not necessarily optimal. We plan to explore this point in future work.

In our study to detect the blocks of the same sign and magnitude we use a greedy algorithm given in Figure 4.

```
Grad_position = [];
position = 0;
while (position < n) do
  Grad_position = append(Grad_position, position);
  Grad_x = compare(x[position], x[position + 1]);
  position = position + 1;
  New_Grad_x = compare(x[position], x[position + 1]);
  while ((Grad_x = New_Grad_x)  and  (position < n)) do
    position = position + 1;
    Grad_x = New_Grad_x;
    New_Grad_x = compare(x[position], x[position + 1]);
  end while
  position = position + 1;
end while
return Grad_position
```

Fig. 4. The greedy algorithm to partition vector of n size in gradients.

The idea is to compare the values of the input vector component-wise. These values can be in decreasing, increasing or balanced order. The number of blocks is determined by the number of gradients calculated by this algorithm. To do so, the algorithm takes as input the solution vector $x = x_1, x_2.....x_n$ at each iteration. The outputs of the algorithm are the gradient of each block noted by $Grad\_x$ and the index of the component by which this block begins noted by $Grad\_position$. The algorithm of Figure 4 partitions a vector x of size n in gradients. This algorithm calls the function compare defined in Figure 5. The

```
function compare(x, y)
if x ≤#_D y then
  result = ↗;
end if
if x ≥#_D y then
  result = ↘;
end if
if x =#_D y then
  result = →;
else
  result = ⊤
end if
```

Fig. 5. Function compare between two values x and y.

greedy algorithm scans x from its first elements to its last (first while loop), the current position is denoted *position*. As long as the values are ordered in the same order following $\leq_I^\#$ or $\leq_K^\#$, we keep the elements in the same block (second while loop). When a change of gradient is detected, we start a new block by initiating a new iteration of the first loop (first while loop). If we consider the example presented previously $x = 1, 2, 3, 4, 4, 3, 2, 1$ we obtain as an output the following two pieces of information for each block, $[Grad\_position = 1 : Grad\_x = \nearrow]$, $[Grad\_position = 4 : Grad\_x = \rightarrow]$, $[Grad\_position = 5 : Grad\_x = \searrow]$. Our work is still in progress and our perspective is to use the abstract gradients to determine how to partition matrices in blocks in such a way that we may assign each block to different processors of a parallel machine and to specialize the summation algorithm of each processor according to its data in order to optimize the numerical accuracy without increasing the execution time (see Figure 1). Usual mappings are depicted in Figure 6, we plan to select one of these mappings according to the gradient information.

4.2 Example

Let us illustrate the method on a small example, a matrix with N=2. First, we generate the system corresponding of the flexion of one dimension beam

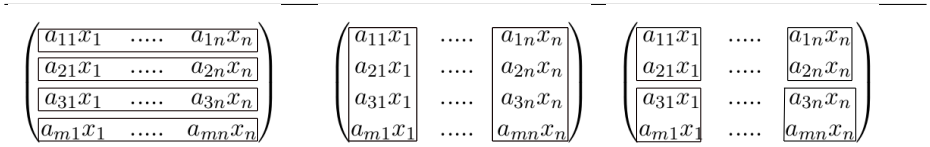

Fig. 6. Different parallelization of Jacobi's method.

introduced in a more general context in Section 5. We then compute the solution of this linear system by the Jacobi's method, and associate to each value of the solution vector its floating point exponent using the abstract function α^n_{Exp} presented in the Section 3.

$$A = \begin{pmatrix} 11778.279410 & -1778.279410 \\ -1778.279410 & 3556.558820 \end{pmatrix}, \quad b = \begin{pmatrix} 0.000662 \\ 0.001125 \end{pmatrix}$$

$$x^1 = \begin{pmatrix} \text{-5.623062e-07} \Rightarrow Exp^1 = \text{-7} \\ \text{-3.162326e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix} \quad x^2 = \begin{pmatrix} \text{-1.039753e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.443479e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix}$$

$$x^3 = \begin{pmatrix} \text{-1.082201e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.682203e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix} \quad x^4 = \begin{pmatrix} \text{-1.118244e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.703427e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix}$$

$$x^5 = \begin{pmatrix} \text{-1.121448e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.721448e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix} \quad x^6 = \begin{pmatrix} \text{-1.124169e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.723050e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix}$$

$$x^7 = \begin{pmatrix} \text{-1.124169e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.723050e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix}$$

Once this step is over, we apply the abstract function noted by α_G on the exponents generated by the first abstraction in order to represent the increasing, decreasing or balanced nature of the vector scalars. The results of this abstraction are given below:

$$x^1 = \begin{pmatrix} Exp^1 = \text{-7} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \nearrow \qquad x^2 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow$$

$$x^3 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow \qquad x^4 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow$$

$$x^5 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow \qquad x^6 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow$$

$$x^7 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow$$

As we can observed the iterates compute the same gradient starting at x^2. We may deduce two points. First, a fixed point can be found for the example by a static analysis. Second, we may chose a summation algorithm efficient for this sequence (balanced sum). We could also assign this block of computation to the same processors using only one summation algorithm.

5 Experimentations

In this section, we introduce some experimental results on a problem taken from mechanics. Section 5.1 introduces the problem and our results are presented in Section 5.2. For our experiments, our implementation is still in progress we only use single matrices instead of sets of matrices. The abstractions $\leq_I^{\#}$ and $\leq_K^{\#}$ introduced in Section 3.2 are then useless (they reduce to singletons) and we directly abstract matrices of concrete elements into gradients. Obviously, we aim at improving this point in our next developments.

5.1 Flexion of a Beam

This example consists of a physical problem arising in Mechanics: the flexion of an $1D$ elastic beam with Dirichlet boundary conditions on its extremities [2]. The discretization of this kind of problem is based on a finite element method (FEM), as typically used in engineering. The actual resolution of most finite elements problem require to solve a system of linear equations, eg. using the recursive algorithm of Jacobi's method. The representative diagram of our case study is presented in Figure 7.

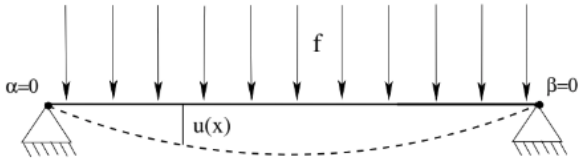

Fig. 7. Representation of the flexion of an 1D beam.

In Figure 7, u is discretized and represents the displacement such as $u_1 = \alpha$ and $u_{N+1} = \beta$, α and β the extremities where the beam is fixed such as ($\alpha = \beta = 0$). f is a constant vertical force acting on the domain interval $\Omega = [0, 1]$. The problem is formalized as:

$$\begin{cases} u''(x) = f \quad \forall x \in]0, 1] \\ u(0) = \alpha \quad \text{and} \quad u(1) = \beta \end{cases}$$

Discretizing the mesh produces a linear system to resolve. First we introduce the mesh of the domain $\Omega = [0, 1]$, considering $N + 1$ nodes $\{x_i, i = 1, \ldots, N + 1\}$ of the interval $[0, 1]$ with $x_1 = 0$, $x_{N+1} = 1$ and $x_{i+1} = x_i + h_i, \forall i = 1, \ldots, N$. Next, the domain $[0, 1]$ is discretized into N intervals (x_i, x_{i+1}) that are the finite elements of size h_i. Last, substitution of the known values (u_1, u_{N+1}) we obtain the tridiagonal system pictured in Figure 8 that has to be considered as a system

of linear equations that we interest to solve using Jacobi's method. This system is made of three blocks. The first block is a tridiagonal matrix corresponding to the beam, the second one (vector u) is the displacement and the last block is a vector giving the size of finite elements.

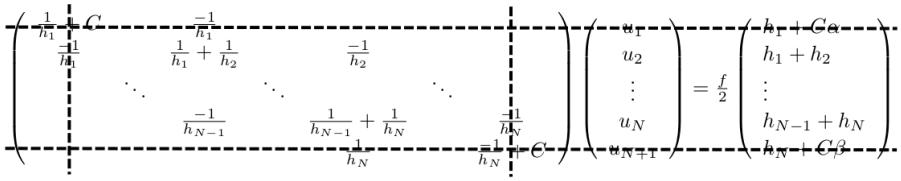

Fig. 8. Tridiagonal linear system.

For our experiments, the values of h are computed automatically with the specificity of obtaining symmetrical values of h. More precisely, we initialize the first and the last values of h and we compute the others so that $h[n-i-1] = h[i]$.

We fix the other parameters: penalization $C = 10^6$ while vertical force $f = -20N/m^2$.

We recall that our motivation is to create linear systems of different size N modeling the flexion of a $1D$ beam. Then, we calculated the real solution X using the Jacobi's method.

Once the solution is calculated, we abstract the values of the solution vector to exponent as presented in Section 3.1. After abstraction, we apply the greedy algorithm to find the best partitioning of data, more precisely blocks of values with the same sign and gradient.

5.2 Experimental Results

In this section, we present the experimental results of our study, focusing on the numerical accuracy of computations obtained using our partitioning and summation algorithms. First, we want to improve the efficiency of our technique in the detection of scalars that have the same sign and gradient which can be grouped in blocks and that could eventually lead to an optimization of the numerical accuracy of computations. The most critical case is to have as many blocks as values in the vectors.

In other words, the ratio between the total number of blocks and the size of the matrix is equal to 1. To test our method, we generate different linear systems of the form $Ax = b$ that model the example previously described in Section 5.1, after that we solve these systems using Jacobi's method, then we applying our technique on the solution vector x. For measuring effectiveness of our technique we compute the average of the gradient of a set of matrices to compare the variation of these average with those of matrices sizes, Figure 9 represents the gradients average corresponding to each matrix size from 10 to 1000.

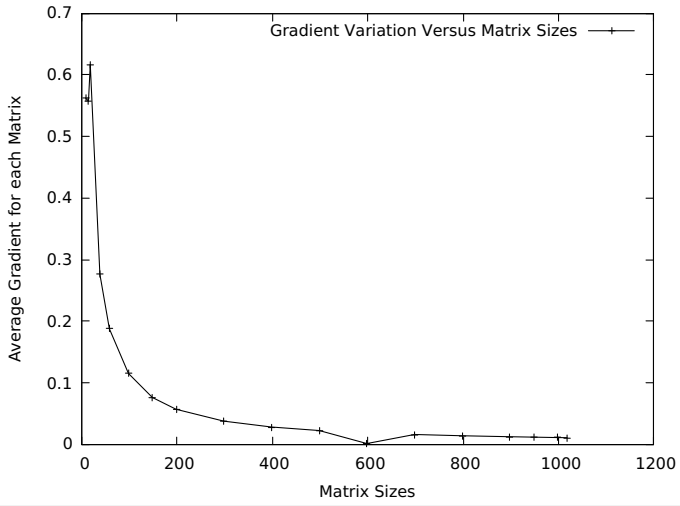

Fig. 9. Gradients average for different matrix size from 10 to 1000.

We notice that for small matrices, for example $N = 100$ the average is around 0.6, so we conclude that the number of blocks represents 60% of the size of matrix. We also notice that when the size of the matrix increases, the average decreases. From these two remarks, we deduce that the efficiency of our technique is reached when we handle large matrices.

Secondly, we want to know if for a given matrix we can generalize the division of scalars into lines, columns or blocks. As mentioned in Section 4, for a linear system of the form $Ax = b$ we apply our technique to a solution vector $x = x_0, x_1, ...x_{n-1}$ at each iteration in order to group the data accordingly to their signs and magnitudes. We associate for each sequence of data a corresponding sequence of gradients, and each block is represented by two values: the index i of the component $x_i, \forall i \in [0, n-1]$ by which it starts and the gradient associated to its sequence of scalars. For the sake of simplicity we consider a matrix of 16×16 with coefficients taken from a realistic example corresponding to the flexion of a beam developed in Section 5.1. The results of our study is given below:

$Iteration1 : [0 :\rightarrow][2 :\searrow][3 :\rightarrow][4 :\searrow][5 :\rightarrow][9 :\nearrow][10 :\rightarrow][11 :\nearrow][12 :\rightarrow]$
$Iteration2 : [0 :\rightarrow][1 :\searrow][2 :\nearrow][3 :\searrow][5 :\rightarrow][6 :\nearrow][7 :\searrow][8 :\rightarrow][10 :\nearrow][11 :\rightarrow][12 :\nearrow]$
$Iteration3 : [0 :\rightarrow][1 :\searrow][2 :\nearrow][3 :\rightarrow][4 :\searrow][5 :\nearrow][6 :\searrow][7 :\nearrow][8 :\searrow][10 :\nearrow][11 :\rightarrow]$
$Iteration4 : [0 :\searrow][1 :\nearrow][2 :\searrow][4 :\nearrow][5 :\searrow][7 :\nearrow][9 :\searrow][12 :\rightarrow]$
$Iteration5 : [0 :\rightarrow][1 :\nearrow][2 :\searrow][3 :\nearrow][4 :\searrow][6 :\nearrow][7 :\searrow][8 :\nearrow][10 :\searrow]$
$Iteration6 : [0 :\rightarrow][1 :\searrow][2 :\nearrow][3 :\searrow][4 :\rightarrow][6 :\nearrow][7 :\searrow][9 :\nearrow][12 :\searrow]$
$Iteration7 : [0 :\rightarrow][1 :\searrow][2 :\rightarrow][3 :\searrow][4 :\nearrow][6 :\searrow][7 :\nearrow][8 :\searrow][10 :\nearrow][11 :\searrow][12 :\nearrow]$
$Iteration8 : [0 :\rightarrow][1 :\nearrow][2 :\searrow][4 :\nearrow][5 :\searrow][7 :\nearrow][9 :\searrow]$
$Iteration9 : [0 :\rightarrow][2 :\searrow][3 :\nearrow][4 :\searrow][6 :\nearrow][7 :\searrow][8 :\nearrow][10 :\searrow]$
$Iteration10 : [0 :\rightarrow][1 :\searrow][2 :\nearrow][3 :\searrow][4 :\rightarrow][5 :\nearrow][7 :\searrow][9 :\rightarrow][10 :\nearrow][12 :\searrow]$
$Iteration11 : [0 :\rightarrow][1 :\searrow][2 :\nearrow][3 :\searrow][4 :\nearrow][6 :\searrow][7 :\nearrow][8 :\searrow][10 :\nearrow][11 :\searrow][12 :\nearrow]$

$Iteration12 : [0 :\rightarrow][1 :\nearrow][2 :\searrow][4 :\nearrow][5 :\searrow][6 :\rightarrow][8 :\nearrow][9 :\searrow]$
$Iteration13 : [0 :\searrow][1 :\nearrow][2 :\searrow][3 :\nearrow][4 :\searrow][6 :\nearrow][7 :\searrow][8 :\nearrow][10 :\searrow]$
$Iteration14 : [0 :\nearrow][1 :\searrow][2 :\nearrow][3 :\searrow][5 :\nearrow][7 :\searrow][9 :\nearrow][12 :\searrow]$
$Iteration15 : [0 :\rightarrow][1 :\searrow][2 :\nearrow][3 :\searrow][4 :\nearrow][6 :\searrow][7 :\nearrow][8 :\searrow][10 :\nearrow][11 :\searrow][12 :\nearrow]$
$Iteration16 : [0 :\rightarrow][1 :\nearrow][2 :\searrow][4 :\nearrow][5 :\searrow][7 :\nearrow][9 :\searrow]$
$Iteration17 : [0 :\rightarrow][1 :\nearrow][2 :\searrow][3 :\nearrow][4 :\searrow][6 :\nearrow][7 :\searrow][8 :\nearrow][10 :\searrow]$
$Iteration18 : [0 :\rightarrow][1 :\searrow][2 :\nearrow][3 :\searrow][5 :\nearrow][7 :\searrow][9 :\nearrow][12 :\searrow]$
$Iteration19 : [0 :\nearrow][1 :\searrow][2 :\nearrow][3 :\searrow][4 :\nearrow][6 :\searrow][7 :\nearrow][8 :\searrow][10 :\nearrow][11 :\searrow][12 :\nearrow]$
$Iteration20 : [0 :\searrow][1 :\nearrow][2 :\searrow][4 :\nearrow][5 :\searrow][7 :\nearrow][9 :\searrow]$
$Iteration21 : [0 :\rightarrow][1 :\nearrow][2 :\searrow][3 :\nearrow][4 :\searrow][5 :\rightarrow][6 :\nearrow][7 :\searrow][8 :\rightarrow][9 :\nearrow][10 :\searrow]$
$Iteration22 : [0 :\rightarrow][1 :\searrow][2 :\nearrow][3 :\searrow][5 :\nearrow][7 :\searrow][9 :\nearrow][12 :\searrow]$
$Iteration23 : [0 :\rightarrow][1 :\searrow][2 :\nearrow][3 :\searrow][4 :\nearrow][6 :\searrow][7 :\nearrow][8 :\searrow][10 :\nearrow][11 :\searrow][12 :\nearrow]$
$Iteration24 : [0 :\rightarrow][1 :\nearrow][2 :\searrow][4 :\nearrow][5 :\searrow][7 :\nearrow][9 :\searrow]$
$Iteration25 : [0 :\rightarrow][2 :\searrow][3 :\nearrow][4 :\searrow][6 :\nearrow][7 :\searrow][8 :\nearrow][10 :\searrow]$

If we take iteration 11 as an example, we notice that after each 4 iterations we find the same sequence of blocks, i.e. the same sequence of gradients is associate to the sequence of scalars as it is shown by iterations 15, 19, 23. In the same way, the iterations 16 and 14 are repeated after 4 iterations respectively given by the iterations 20, 24 and 18, 22. A static analyzer unfolding the loop 4 times could then find a fixed point for this example.

6 Conclusion

In this article, we have introduced our technique which relies en static analysis to improve the numerical accuracy of linear systems resolution. We have detailed its different steps and the algorithm that its used to detect potential ordering of scalars. We have tested our technique across experimental results obtained on one example coming from a mechanical problem. The results obtained show the efficiency of our technique in data analysis to identifying different groups of data accordingly to their signs and magnitudes. An interesting perspective consists on specializing the code of each process depending on its group of data. We would like to study how our technique can be extended to other iterative methods (Newton, Gauss-Seidel, etc.,). Code transformation techniques for numerical precision have proved their efficiency to accelerate the convergence of iterative methods in the sequential case. These results will be extended to parallel algorithms to improve the convergence of iterative methods.

Acknowledgments This work was supported by a scholarship from the Languedoc Roussillon region and partially by project ANR-17-CE25-0018 FEANICSES.

References

1. ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic.* SIAM, 2008.

2. M. Barboteu, N. Djehaf, and M. Martel. Numerically accurate code synthesis for gauss pivoting method to solve linear systems coming from mechanics. *Computers & Mathematics with Applications*, 77(11):2883–2893, 2019.
3. F. Benmouhoub, N. Damouche, and M. Martel. Improving the numerical accuracy of high performance computing programs by process specialization. In Matthieu Martel, Nasrine Damouche, and Julien Alexandre Dit Sandretto, editors, *TNC'18. Trusted Numerical Computations*, volume 8 of *Kalpa Publications in Computing*, pages 11–23. EasyChair, 2018.
4. I.D. Coope and M. Macklem. Parallel jacobi methods for derivative-free optimization on parallel or distributed processors. *The ANZIAM Journal*, 2004, 07 2005.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.
6. N. Damouche, M. Martel, and A. Chapoutot. Impact of accuracy optimization on the convergence of numerical iterative methods. In M. Falaschi, editor, *LOPSTR 2015*, volume 9527 of *Lecture Notes in Computer Science*, pages 143–160. Springer, 2015.
7. N. Damouche, M. Martel, and A. Chapoutot. Improving the numerical accuracy of programs by automatic transformation. *STTT*, 19(4):427–448, 2017.
8. Y. Georgiou, E. Jeannot, G. Mercier, and A. Villiermet. Topology-aware job mapping. *IJHPCA*, 32(1):14–27, 2018.
9. T. Hoefler, E. Jeannot, and G. Mercier. An Overview of Process Mapping Techniques and Algorithms in High-Performance Computing. In Emmanuel Jeannot and Julius Zilinskas, editors, *High Performance Computing on Complex Environments*, pages 75–94. Wiley, June 2014.
10. A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, volume 7460 of *Lecture Notes in Computer Science*, pages 75–93. Springer, 2012.
11. G.A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.
12. F.T. Luk and H. Park. A proof of convergence for two parallel jacobi svd algorithms. *IEEE Trans. Comput.*, 38(6):806–811, June 1989.
13. J.M. Muller, N. Brisebarre, F. De Dinechin, C.P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.
14. P. Panchekha, A. anchez-Stern, J.R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 1–11. ACM, 2015.
15. S.M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: faithful rounding. *SIAM J. Scientific Computing*, 31(1):189–224, 2008.
16. S.M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part II: sign, k-fold faithful and rounding to nearest. *SIAM J. Scientific Computing*, 31(2):1269–1302, 2008.
17. D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Perics. Trends in data locality abstractions for hpc systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):3007–3020, Oct 2017.

18. B.B. Zhou and R.B. Brent. On parallel implementation of the one-sided jacobi algorithm forsingular value decompositions. pages 401–408, 02 1995.