Sparse Computation with Pei

Voisin-Demery Frédérique and Guy-René Perrin
ICPS, University Louis Pasteur,
pôle API, boulevard Sébastien Brant
67400 Illkirch, France
{voisin,perrin}@icps.u-strasbg.fr

1999

Abstract

Pei formalism has been designed to reason and develop parallel programs in the context of data parallelism. In this paper, we focus on the use of Pei to transform a program involving dense matrices into a new program involving sparse matrices, using the example of the matrix-vector product.

Keywords: irregular computation, sparse matrix, data parallelism, program transformation

1 Introduction

A wide range of research work on the static analysis of programs forms the foundation of parallelization techniques which improve the efficiency of codes: loop nest rewriting, directives to the compiler to distribute data or operations, etc. These techniques are based on geometric transformations either of the iteration space or of the index domains of arrays.

This geometric approach entails an abstract manipulation of array indices to define and transform the data dependences in the program. This requires the ability to express, compute and modify the placement of the data and operations in an abstract discrete reference domain. The programming activity may then refer to a very small set of primitive issues to construct, transform, or compile programs. Pei is a program notation and includes such issues.

In this paper we focus on the way Pei deals with sparse computations. For any matrix operation defined by a program, a sparse computation is a transformed code which adapts computations efficiently in order to respect an optimal storage of sparse matrices [2]. Such a storage differs from the natural memory access, since the location of any non-zero element of the matrix has to be determined. Due to the reference geometrical domain in Pei, we show how a dense program can be transformed to meet some optimal memory storage.

2 Definition of the Formalism PEI

2.1 An Introduction to Pei Programming

The theory Pei was defined [13, 14] in order to provide a mathematical notation to describe and reason on programs and their implementation. A program can be specified as a relation between some *multisets* of value items, or, roughly speaking, its *inputs* and *outputs*. Of course, programming may imply putting these items in a convenient organized directory, depending on the problem terms. In scientific computations, for example, items such as arrays are functions on indices. The index set, that is, the reference domain, is a part of some $\mathbf{Z}^n$. In Pei such a multiset of value items mapped on a discrete reference domain is called a *data field*.

For example, the multiset of integral items $\{1, -2, 3, 1\}$ can be expressed as a data field, say `A`, each element of which is recognized by an index in $\mathbf{Z}$ (e.g., from 0 to 3). Of course, this multiset may be expressed as another data field, say `M`, which places the items on points $(i, j) \in \mathbf{Z}^2$ such that $0 \leq i, j < 2$. These two data fields `A` and `M` are considered equivalent in Pei, since they express the same multiset. More formally, there exists a bijection from the first arrangement onto the second one, e.g. $\sigma(i) = (i \bmod 2, i \operatorname{div} 2)$. We note this by the equation:

$$\mathtt{M} = \mathtt{align::A} \qquad (1)$$

illustrated by Fig.#1 where

$$\mathtt{align} \quad = \quad \lambda i \mid (0 \leq i \leq 3) \,.\, (i \bmod 2, i \operatorname{div} 2) \qquad (2)$$

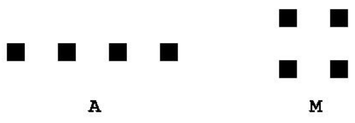

Figure 1: Example of change of basis

Any Pei program is composed of such unoriented equations, as shown in the following example of the matrix-vector multiplication.

2.1.1 Example of Matrix-Vector Multiplication

For any i in $[1..n]$, compute $x_i = \sum_{j=1}^{j=n} a_{i,j} \times v_j$.

Any x_i can be defined by a recurrence on the domain $\{j \mid 1 \leq j \leq n\}$ of $\mathbf{Z}$ by:

$$\left\{ \begin{array}{lcllr} s_{i,1} & = & a_{i,1} \times v_1 & & (a) \\ s_{i,j} & = & s_{i,j-1} + (a_{i,j} \times v_j) & 1 < j \leq n & (b) \\ x_i & = & s_{i,n} & & (c) \end{array} \right.$$

where the $s_{i,j}$ are intermediate results. The recurrence equation (b) emphasizes uniform dependencies $(0, 1)$ for the calculation of $s_{i,j}$.

In PEI, this definition can be expressed in the following way: matrix A is expressed as a data field `A` mapped on the domain $\{(i,j) \mid 1{\leq}i,j{\leq}n\}$. Vectors v and x are expressed as data fields `V` and `X` mapped on $\{i \mid 1{\leq}i{\leq}n\}$. This means that for any i, the values v_i and x_i are located at point i. The resulting PEI program is shown in Fig.#2. Of course, any program which lies on another mapping for `A`, `V` or `X` defines an equivalent program, provided its operations result in an equivalent data field. The recurrence defining the variable s suggests mapping the data field `S` in $\mathbf{Z}^2$. Since the v_j are used by the $s_{i,j}$, for any i, the values of `V` are also mapped in $\mathbf{Z}^2$, by a kind of data *strip-mining* on data field `B`. Its values are then broadcast to *localize* the right values onto the right locations in order to compute the *recurrence* steps.

```
MatVec: (A,V) $\mapsto$ X
matrix::A = A
align::B = V
P = prod $\triangleright$ (A/&/(B $\triangleleft$ spread))
S = add $\triangleright$ (P/&/(S $\triangleleft$ shift))
X = project::(S $\triangleleft$ last)
matrix   =  $\lambda(i,j)\ |(1{\leq}i{\leq}n, 1{\leq}j{\leq}n) . (i,j)$
align    =  $\lambda(i,j)\ |(i{=}1, 1{\leq}j{\leq}n) . j$
project  =  $\lambda(i,j)\ |(1{\leq}i{\leq}n, j{=}n) . i$
shift    =  $\lambda(i,j)\ |(1{\leq}i{\leq}n, 1{<}j{\leq}n) . (i,j{-}1)$
last     =  $\lambda(i,j)\ |(1{\leq}i{\leq}n, j{=}n) . (i,j)$
spread   =  $\lambda(i,j)\ |(1{\leq}i{\leq}n, 1{\leq}j{\leq}n) . (1,j)$
add      =  id # $\lambda(a{\cdot}b) . a{+}b$
prod     =  $\lambda(a{\cdot}b) . a{\times}b$
```

Figure 2: Matrix-vector multiplication in PEI

For the sake of simplicity, this intuitive explanation referred to the data parallel programming issues. Nevertheless, this imperative presentation must not confuse the reader: PEI is a declarative language which expresses equations on data fields. Therefore, the following is a more complete comment on every equations of Fig.#2:

- The first equation defines the mapping of the input data field `A`. Function `matrix`, defined on $\mathbf{Z}^2$, applies a change of basis (notation `::`), which is the identity on the domain $[1..n]\times[1..n]$. It means that the values are placed onto a square $[1..n] \times [1..n]$.
- The second equation implicitly defines `B`. The function `align`, defined on $\mathbf{Z}^2$, applies a change of basis, which expresses that the projection of its argument `B`, supposed to be a row, on a one-dimensional domain, is equal to `V`. Therefore, `V` is *aligned* on a two-dimensional domain. Notice that functions such as `align` define the context of the program. They are expressed as λ-expressions, in which the separator "|" allows the definition of the function domain, and "." begins the function body.

- In the third equation, the operation `/&/` builds the pairs of value items of `A` and (`B` $\lhd$ `spread`), which are placed at the same location. The last expression applies a *geometrical operation* (notation $\lhd$), which broadcasts the values of the "first row" of data field `B` in the direction $(1, 0)$ in $\mathbf{Z}^2$.
- The data field `S` results from the application of a so-called *functional operation* (notation $\rhd$) on two data fields. The first one results from a geometrical operation on `S` which expresses the dependency $(0, 1)$ in $\mathbf{Z}^2$. Note that `S` also collects the partial sums computed during the reduction.
- The last equation defines the data field `X`, by a change of basis. It projects the part of `S` mapped on $\{(i, n) \mid 1 \le i \le n\}$ onto $\mathbf{Z}$. Therefore, the result `X` is one-dimensional.

Figure 3 represents the geometrical operations detailed above.

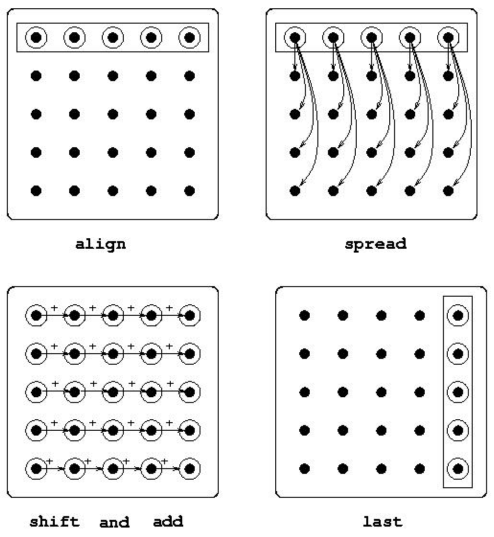

Figure 3: Matrix-vector multiplication in Pei.

2.1.2 Translation into HPF

The previous program can be translated into HPF [7], thanks to a Pei-to-HPF translator developed at ICPS. More information on this tool design can be found in [6]. At first, the program is transformed into a translatable one. For instance, new data fields can be added to allow the translation. As far as we are concerned, the computation of data field P requires such transformation. The equation

$$\texttt{P} \quad = \quad \texttt{add} \rhd (\texttt{A/\&/(B} \lhd \texttt{spread)}) \tag{3}$$

has to be split into

$$\begin{aligned} \mathtt{D} &= \mathtt{B} \lhd \mathtt{spread} & (4) \\ \mathtt{P} &= \mathtt{add} \rhd (\mathtt{A/\&/D}). & (5) \end{aligned}$$

A type-checker verifies that there are no contradictory definitions in the program and returns the domain of each data field, allowing the declaration of variables. Each equation can then be translated. Applications of change of basis are translated into `align` directives after a template with which all variables can be aligned is computed, followed by initializations of variables according to the change of basis. Functional operations are translated using `forall` statements, as well as geometrical operations. Translation of recurrent equations results in a phase of initialization for the variable that will collect the result, followed by the computation of the recurrent equation. The resulting HPF code for the Pei program of Fig.#2 is shown on Fig.#4.

```
      program MatVec
      real A(1:n,1:n), V(1:n), B(1:n,1:n)
      real D(1:n,1:n), P(1:n,1:n), S(1:n,1:n)
      real X(1:n)+
!hpf$ template T(1:n,1:n)
!hpf$ align A,B,P,S with T
!hpf$ align V(j) with T(1,j)
!hpf$ align X(i) with T(i,n)
      forall (j=1:n) B(0,j)=V(j)
      forall (i=1,n, j=1:n) D(i,j)=B(1,j)
      forall (i=1:n, j=1:n) P(i,j)=A(i,j)*D(i,j)
      forall (i=1:n) S(i,1)=0
      forall (i=1:n)
        do j=1,n S(i,j)=S(i,j-1)+P(i,j)
      endforall
      forall (j=1:n) X(j)=S(j,n)
```

Figure 4: Translation of the Pei matrix-vector multiplication program into HPF

2.2 Syntactic Issues

The previous example allows shows that data fields are main features in Pei. Besides data fields, partial functions define operations on these objects. In the following, `A`, `B`, `X`, etc. denote data fields, whereas `f`, `g`, etc. denote functions. The Pei notation for partial functions is derived from the lambda-calculus. Any function `f` of domain $dom(\mathtt{f}) = \{x \mid P(x)\}$ is denoted as $\lambda x \mid P(x) . \mathtt{f}(x)$. Moreover a function `f` defined on disjunctive sub-domains is denoted as a partition $\mathrm{f}_1 \# \mathrm{f}_2$ of functions.

Expressions are defined by applying operations on data fields. Pei defines one internal operation on the data field set, called *superimposition.*

It is denoted as `/&/`and builds the sequences of values of its arguments[1]. Three external operations are defined and associate a data field with a function:

- The *functional operation*, denoted as $\triangleright$, applies a function `f` (which may be a partial function) on value items of a data field `X` (notation $\texttt{f} \triangleright \texttt{X}$).
- The *change of basis* (denoted as `::`). Let `h` be a bijection, `h::X` defines a data field that maps the value items of `X` onto another discrete reference domain.
- *The geometrical operation*, denoted as $\triangleleft$, moves the value items of a data field on its domain. The data field $\texttt{X} \triangleleft \texttt{g}$ is such that the item mapped on some index z, $z \in \mathit{dom}(\texttt{g})$, "comes from" `X` at index $\texttt{g}(z)$.

2.3 Semantics and Program Transformations

The semantics of Pei are founded on the notion of discrete domain associated with a multiset. We call *drawing* of a multiset M of values in V a partial function v from some $\mathbf{Z}^n$ in V, whose image is M.

Such a drawing should define a natural functional interpretation $[\![\texttt{X}]\!] = v$ of a data field `X`. As we have said it at the beginning of this section, any other drawing can be deduced by applying any bijection, say h. As soon as its domain $\mathit{dom}(h)$ contains $\mathit{dom}(v)$, its interpretation should also be equal to $v \circ h^{-1}$. This is not sound, and we have to indeed consider a data field `X` as the abstraction of any drawing of a given multiset. A *data field* `X` is a pair $(v : \sigma)$, composed of a drawing v of a multiset M and of a bijection σ such that $\mathit{dom}(v) \subset \mathit{dom}(\sigma)$.

This definition induces a sound interpretation of a data field $\texttt{X} = (v : \sigma)$ as the function $[\![\texttt{X}]\!] = v \circ \sigma^{-1}$. It founds the semantics of operations on data fields.

The semantics allow the definition of program transformations in Pei as soon as a data field is substituted for any equivalent one in an expression, or by applying some *algebraic law* on the operations. A few examples of such laws (a detailed list is given in [15]), assuming right and left expressions are valid expressions, include:

$$\texttt{f1} \triangleright (\texttt{f2} \triangleright \texttt{X}) = (\texttt{f1} \circ \texttt{f2}) \triangleright \texttt{X} \quad (6)$$

$$\texttt{X} \triangleleft (\texttt{g1} \circ \texttt{g2}) = (\texttt{X} \triangleleft \texttt{g1}) \triangleleft \texttt{g2} \quad (7)$$

$$(\texttt{X1/\&/X2}) \triangleleft \texttt{g} = (\texttt{X1} \triangleleft \texttt{g})\texttt{/\&/}(\texttt{X2} \triangleleft \texttt{g}) \quad (8)$$

$$\texttt{h::(X1/\&/X2)} = \texttt{(h::X1)/\&/(h::X2)} \quad (9)$$

$$\texttt{h::}(\texttt{f} \triangleright \texttt{X}) = \texttt{f} \triangleright (\texttt{h::X}) \quad (10)$$

$$\texttt{h::}(\texttt{X} \triangleleft \texttt{g}) = (\texttt{h::X}) \triangleleft \texttt{h} \circ \texttt{g} \circ \texttt{h}^{-1} \quad (11)$$

[1]We use two operators on sequences: an associative constructor denoted as "," and the function "`id`" which is the identity on sequences of one element.

3 PEI and Sparse Computations

3.1 Introduction to Sparse Computation

Sparse computations occur in linear algebra when dealing with sparse matrices. A matrix is said to be sparse when it has a certain amount of zero entries. Such matrices arise in the resolution of partial differential equations using finite elements for example. They require special handling, as only the non-zero entries have to be stored, saving both space-storage and computational time (no computation is done with the zero elements).

There are several sparse storages available, some of which are described in [1]. The most simple is *coordinate storage.* It requires three arrays: one for the non-zero entries, another for the corresponding row indices, and the third one for the corresponding column indices. The FORTRAN code for matrix-vector multiplication $v = Ab$ is given in Fig.#5, each array being respectively `a`, `ia`, `ja`, and `nnz` being the total number of non-zero elements. Another popular storage [12] is *CSR* or *compress sparse row.*

```
do i=1, nnz
  v[ia[i]] = v[ia[i]]+b[ja[j]]*a[i]
enddo
```

Figure 5: Matrix-vector multiplication using Coordinate Storage

It also requires three arrays: one for non-zero entries, stored row-wise; one for the corresponding column indices; and one as a pointer on the beginning of each new row in the previous arrays, separating the rows from each other. The same storage can be used in a column-wise storage, the non-zero values being stored column-wise and the pointer array pointing to the beginning of each column. The storage is then *compress sparse column.* Figure 6 shows the FORTRAN code for the matrix-vector multiplication $v = Ab$ using CSR, assuming that the values are stored in `a`, the column indices in `ja` and the row pointers in `ia`.

```
do i=1,n
  do j=ia[i],ia[i+1]-1
    v[i] = v[i]+b[ja[j]]*a[j]
  enddo
enddo
```

Figure 6: Matrix-vector multiplication using CSR storage

Such programs are hard to analyze due to indirections such as `b[ja[i]]`, or those appearing in the loop bounds (compare to dense code shown in Fig.#7). They are also hard to write, each storage leading to a different code. Their parallelization is also a problem, because data-accesses are irregular, making data dependences and potential parallelism difficult to find.

```
do i=1,n
  do j=1,n
    v[i] = v[i]+b[j]*a[j]
  enddo
enddo
```

Figure 7: Dense matrix-vector multiplication

A solution to the problem can be to write sparse code from dense code, but in an "automatic" way. We intend to use Pei as a formal framework to apply transformations to a dense program in order to get an equivalent sparse program.

3.2 Pei Approach

Pei can deal with sparse computations by adapting a code in order to respect an optimal storage of sparse data structures. Such a storage differs from the natural memory access since the location of any non-zero element of the matrix has to be determined. In terms of Pei features, this means that the optimal storage and the dense array are two equivalent data fields. The change of basis from one data field onto the other one defines the way the sparse matrix is stored. Therefore, transforming the code consists in applying this change of basis to the dense program.

Let us consider again the example of a matrix-vector product, assuming the matrix is composed of a few bands parallel to the diagonal (see Fig.#8, where non-zero elements are located at indices such that either $i = j + 2$, or $i = j$, or $i = j - 2$).

First, a sparse storage has to be chosen so that the dense program can be transformed. In this example, a modified version of CSR called MSR (*modified sparse row*) has been chosen. This storage consists of two arrays, A and JA, the main diagonal being stored apart at the beginning of A. The rest of the matrix is stored using CSR (or CSC, for MSC), except that IA is replaced by the first positions of JA (those corresponding to the diagonal elements in A, as shown in Fig.#8).

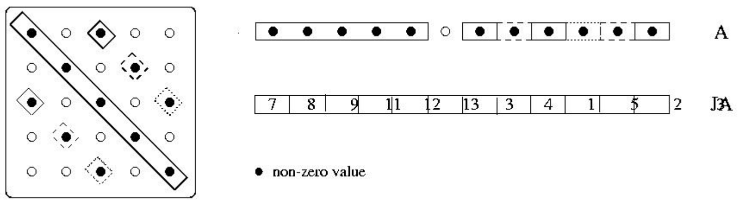

Figure 8: Sparse matrix in dense representation and MSR storage

3.3 Program Transformation in Pei

The non-zero value items of the matrix and the vector value items must belong to the domain of the change of basis to be applied in Pei. Therefore, the vector can no longer be aligned with the first row (that may be sparse); otherwise, dropping the zero values with the change of basis would suppress some vector values. That is why a first transformation step consists, for example, of aligning the vector with the main diagonal. MSR storage is then well-adapted. This alignment can be defined by rewriting `spread` as (see Fig.#9):

$$\texttt{spread} = \texttt{pivot} \circ \texttt{diagonal} \tag{12}$$

where

$$\begin{aligned} \texttt{pivot} &= \lambda(i,j)\ \mid (1{\le}i{\le}n, j{=}i)\,.\,(1,j) & (13)\\ \texttt{diagonal} &= \lambda(i,j)\ \mid (1{\le}i{\le}n, 1{\le}j{\le}n)\,.\,(j,j). & (14)\end{aligned}$$

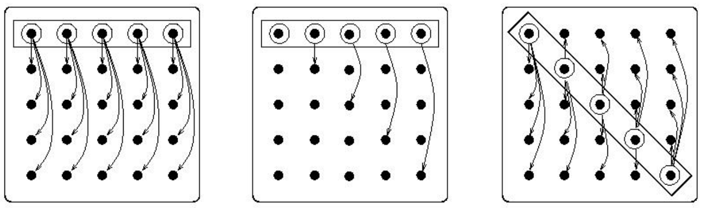

Figure 9: Geometrical functions `spread`, `pivot` and `diagonal`

We obtain the new program in Pei:

```
{
matrix :: A = A
align :: B = V
P = prod ▷ ( A /&/( (B ◁ pivot) ◁ diagonal ) )
...
}
```

The change of basis itself (Fig.#8) is defined by the following function:

$$\begin{aligned} \texttt{gather} = &\quad \lambda(i,j)\ \mid (1{\le}i{\le}2, j{=}i{+}2)\,.\,(n{+}i{+}1)\\ &\#\ \lambda(i,j)\ \mid (3{\le}i{\le}n{-}2, j{=}i{+}2)\,.\,(n{+}2i{-}1)\\ &\#\ \lambda(i,j)\ \mid (1{\le}i{\le}n, j{=}i)\,.\,(i)\\ &\#\ \lambda(i,j)\ \mid (3{\le}i{\le}n{-}2, j{=}i{-}2)\,.\,(n{+}2i{-}2)\\ &\#\ \lambda(i,j)\ \mid (n{-}1{\le}i{\le}n, j{=}i{-}2)\,.\,.(2n{+}i{-}3). \end{aligned} \tag{15}$$

Its inverse is

$$\begin{array}{rcll} \texttt{gather}^{-1} & = & & \lambda(i)\ \mid(1\le i\le n)\,.\,(i,i)) \\ & & \# & \lambda(i)\ \mid(n{+}2\le i\le n{+}3)\,.\,(i{-}n{-}1, i{-}n{+}1) \\ & & \# & \lambda(i)\ \mid(n{+}4\le i\le 3n{-}5), (i{-}n)\mathrm{mod}2{=}0)\,. \\ & & & \quad((i{-}n{-}1)/2+2), (i{-}n{-}1)/2) \\ & & \# & \lambda(i)\ \mid(n{+}4\le i\le 3n{-}5), (i{-}n)\mathrm{mod}2{=}1)\,. \\ & & & \quad((i{-}n{-}1)/2{+}1), (i{-}n{-}1)/2{+}3) \\ & & \# & \lambda(i)\ \mid(3n{-}4\le i\le 3n{-}3)\,.\,(i{-}2n{+}3, i{-}2n{+}1). \end{array} \quad (16)$$

Applying the change of basis to P leads to this definition, using rule (10) previously given in section 2.3:

$$\begin{array}{rcl} \texttt{gather::P} & = & \texttt{prod} \rhd (\texttt{gather::A /\&/} \\ & & \quad \texttt{gather::}((\texttt{B} \lhd \texttt{pivot}) \lhd \texttt{diagonal})). \end{array} \quad (17)$$

We finally obtain, applying rule (11) to $\texttt{gather::}((\texttt{B} \lhd \texttt{pivot}) \lhd \texttt{diagonal})$:

$$\begin{array}{rcl} \texttt{P} & = & \texttt{prod} \rhd (\texttt{gather::A /\&/} \\ & & (\texttt{gather::}(\texttt{B} \lhd \texttt{pivot})) \lhd (\texttt{gather} \circ \texttt{diagonal} \circ \texttt{gather}^{-1})) \end{array} \quad (18)$$

Let

$$\texttt{P'} = \texttt{gather::P} \quad (19)$$
$$\texttt{A'} = \texttt{gather::A} \quad (20)$$
$$\texttt{T'} = \texttt{gather::}(\texttt{B} \lhd \texttt{pivot}). \quad (21)$$

We can write:

$$\texttt{P'} = \texttt{prod} \rhd (\texttt{A'/\&/}(\texttt{T'} \lhd \texttt{spread'})) \quad (22)$$

where the new function `spread'`, illustrated in Fig.#10, is defined below.

$$\begin{array}{rcll} \texttt{spread'} & = & \multicolumn{2}{l}{\texttt{gather} \circ \texttt{diagonal} \circ \texttt{gather}^{-1}} \\ & = & & \lambda(i)\ \mid(1\le i\le n)\,.\,(i) \\ & & \# & \lambda(i)\ \mid(n{+}2\le i\le n{+}3)\,.\,(i{-}n{+}1) \\ & & \# & \lambda(i)\ \mid(n{+}4\le i\le 3n{-}5, (i{-}n)\mathrm{mod}2=0)\,.\,((i{-}n{-}1)/2) \\ & & \# & \lambda(i)\ \mid(n{+}4\le i\le 3n{-}5, (i{-}n)\mathrm{mod}2=1)\,.\,((i{-}n{-}1)/2+3) \\ & & \# & \lambda(i)\ \mid(3n{-}4\le i\le 3n{-}3)\,.\,(i{-}2n{+}1) \end{array} \quad (23)$$

In order to apply the change of basis `gather` in the equation defining `S`, the data field definition is constrained on the domain of `gather` by geometrical operation `sparse` that describes the matrix non-zero structure:

$$\begin{array}{rcll} \texttt{sparse} & = & & \lambda(i,j)\ \mid(3\le i\le n, j{=}i{-}2) \\ & & \# & \lambda(i,j)\ \mid(1\le i\le n, j{=}i) \\ & & \# & \lambda(i,j)\ \mid(1\le i\le n{-}2, j{=}i{+}2) \end{array} \quad (24)$$

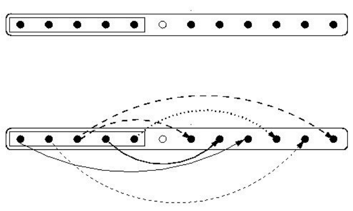

Figure 10: New geometrical functions align and spread'

When we apply sparse to S, we get

$$\begin{aligned}\texttt{S} \lhd \texttt{sparse} &= \texttt{add} \rhd (\texttt{P} \lhd \texttt{sparse /\&/} \\ &\qquad (\texttt{S} \lhd \texttt{shift} \circ \texttt{shift}) \lhd \texttt{sparse}) \\ &= \texttt{add} \rhd (\texttt{P} \lhd \texttt{sparse /\&/S} \lhd (\texttt{shift} \circ \texttt{shift} \circ \texttt{sparse})) \\ &= \texttt{add} \rhd (\texttt{P} \lhd \texttt{sparse /\&/} \\ &\qquad \texttt{S} \lhd (\texttt{sparse} \circ \texttt{shift} \circ \texttt{shift} \circ \texttt{sparse})) \\ &= \texttt{add} \rhd (\texttt{P} \lhd \texttt{sparse /\&/(S} \lhd \texttt{sparse)} \lhd \texttt{sparse\_shift}) \qquad (25)\end{aligned}$$

where, in this particular case,

$$\begin{aligned}\texttt{sparse\_shift} &= \texttt{shift} \circ \texttt{shift} \circ \texttt{sparse} \\ &= \quad \lambda(i,j)\ \vert (3{\le}i{\le}n, j{=}i)\,.\,(i,j{-}2) \\ &\quad \#\ \lambda(i,j)\ \vert (1{\le}i{\le}n{-}2, j{=}i{+}2)\,.\,(i,j{-}2). \qquad (26)\end{aligned}$$

We have now to apply gather to the summation. Let

$$\texttt{S1} = \texttt{S} \lhd \texttt{sparse}. \qquad (27)$$

We can write, using rule (10) followed by rule (11) :

$$\begin{aligned}\texttt{gather::S1} &= \texttt{add} \rhd (\texttt{gather::(P} \lhd \texttt{sparse) /\&/} \\ &\qquad \texttt{gather::(S1} \lhd \texttt{sparse\_shift})) \\ &= \texttt{add} \rhd (\texttt{gather::(P} \lhd \texttt{sparse) /\&/} \\ &\quad (\texttt{gather::S1}) \lhd \texttt{gather} \circ \texttt{sparse\_shift} \circ \texttt{gather}^{-1}) \qquad (28)\end{aligned}$$

Note that since gather only applies to the non-zero elements, we have

$$\texttt{P'} = \texttt{gather::(P} \lhd \texttt{sparse}). \qquad (29)$$

Let

$$\texttt{S'} = \texttt{gather::S1}. \qquad (30)$$

We finally get

$$\texttt{S'} = \texttt{add} \rhd (\texttt{P'/\&/(S'} \lhd \texttt{shift'})) \qquad (31)$$

where the new geometrical operation `shift'` illustrated in Fig.#11 is defined by

$$\begin{aligned}\texttt{shift'} &= \texttt{gather} \circ \texttt{sparse\_shift} \circ \texttt{gather}^{-1}\\ &= \quad \lambda(i)\ \mid (n{+}2{\le}i{\le}n{+}3)\,.\,(i{-}n{-}1)\\ &\quad \#\ \lambda(i)\ \mid (n{+}4{\le}i{\le}3n{-}5, (i{-}n)\mathrm{mod}2{=}1)\,.\,((i{-}n{-}1)/2{+}1)\\ &\quad \#\ \lambda(i)\ \mid (3{\le}i{\le}n{-}2)\,.\,(n{+}2i{-}2)\\ &\quad \#\ \lambda(i)\ \mid (n{-}1{\le}i{\le}n)\,.\,(2n{+}i{-}3). \end{aligned} \tag{32}$$

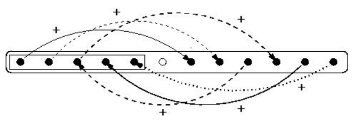

Figure 11: New geometrical function `shift`

We finally have to find the result of the matrix-vector multiplication. The equation defining `X` in Fig.#2 has to be split, so that the transformation of `Z` = `S` ◁ `last` is computed first. After that, the new solution `X'` can be found. The dense data field `S` has to be constrained to the sparse domain, so that `S` ◁ `last` becomes (`S` ◁ `sparse`) ◁ `sparse_last`, which can be simplified into `S1` ◁ `sparse_last` where `S1` was defined in (27) and

$$\begin{aligned}\texttt{sparse\_last} &= \quad \lambda(i,j)\ \mid (1{\le}i{\le}n{-}2, j{=}i{+}2)\\ &\quad \#\ \lambda(i,j)\ \mid (n{-}1{\le}i{\le}n, j{=}i). \end{aligned} \tag{33}$$

Then we can apply the change of basis leading to

$$\begin{aligned}\texttt{gather::Z} &= \texttt{gather::(S1} \lhd \texttt{sparse\_last)}\\ &= \texttt{S'} \lhd (\texttt{gather} \circ \texttt{sparse\_last} \circ \texttt{gather}^{-1}) \end{aligned} \tag{34}$$

where `S'` was defined in (30). With

$$\texttt{Z'} = \texttt{gather::Z} \tag{35}$$

and

$$\begin{aligned}\texttt{last'} &= \texttt{gather} \circ \texttt{sparse\_last} \circ \texttt{gather}^{-1}\\ &= \quad \lambda(i)\ \mid (n{+}1{\le}i{\le}n{+}2).(i)\\ &\quad \#\ \lambda(i)\ \mid (n{+}3{\le}i{\le}3n{-}5, (i{-}n)\mathrm{mod}2 = 1).(i)\\ &\quad \#\ \lambda(i)\ \mid (n{-}1{\le}i{\le}n).(i) \end{aligned} \tag{36}$$

we have the equation illustrated in Fig.#12 and displayed below:

$$\texttt{Z'} = \texttt{S'} \lhd \texttt{last'}. \tag{37}$$

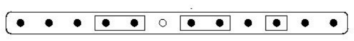

Figure 12: New geometrical function `last'`

The solution vector can be aligned with the matrix diagonal by applying
`project'` to `Z'`, where

$$\texttt{project'} = \lambda(i)\ |(1{\le}i{\le}2).(i{+}n) \;\#\; \lambda(i)\ |(2{\le}i{\le}n{-}2).(2i+n-1) \;\#\; \lambda(i)\ |(n{-}1{\le}i{\le}n).(i), \quad (38)$$

leading to

$$\texttt{X'} = \texttt{project'::Z'}. \quad (39)$$

Finally, Eqs.#(37) and (39) can be merged to provide

$$\texttt{X'} = \texttt{project'::(S'} \lhd \texttt{last')}. \quad (40)$$

This completes the program transformations in PEI, and defines a sparse solution. The final sparse program is written in Fig.#13, where `sparsematrix` = `gather`∘`matrix` and `align'` = `gather`∘`pivot`∘`align`. Functions `spread'`, `shift'`, and `last'` are described in Eqs.#(23), (32), and (36).

```
SparseMatVec: (A,V) ↦ Z
{
sparsematrix::A = A
align'::B = V
P = prod ▷ (A/&/(B ◁ spread'))
S = add ▷ (P/&/(S ◁ shift'))
X = project'::(S ◁ last')
}
```

Figure 13: Sparse matrix-vector multiplication with PEI

Notice that PEI expressions are linked to the definition of the arrays in MSR storage. For example, let `second` = $\lambda(i,j).(j)$. We clearly have

$$JA(i) = \texttt{second} \circ \texttt{gather}^{-1}(i) \quad \text{for} \quad i > n+1. \quad (41)$$

However, PEI does not require implementing the first $n+1$ elements of JA because references to rows are useless since the vector is explicitly aligned with the computation points by the change of basis. References to rows are thus hidden in the function `gather`.

3.4 Other Storages

The previous example showed the transformation of a dense matrix vector multiplication into a sparse code where the matrix is stored using MSR storage. However, other storages can be described, including, for instance, coordinate storage. Let us consider the following matrix A and its storage:

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 3 & 0 & 4 & 0 \\ 5 & 0 & 6 & 0 & 7 \\ 0 & 8 & 0 & 9 & 0 \\ 0 & 0 & 10 & 0 & 11 \end{pmatrix}. \tag{42}$$

A can be stored as

$$\begin{aligned} a &= \boxed{1}\boxed{4}\boxed{6}\boxed{7}\boxed{3}\boxed{9}\boxed{2}\boxed{5}\boxed{8}\boxed{10}\boxed{11} \\ ia &= \boxed{1}\boxed{2}\boxed{3}\boxed{3}\boxed{2}\boxed{4}\boxed{1}\boxed{3}\boxed{4}\boxed{5}\boxed{5} \\ ja &= \boxed{1}\boxed{4}\boxed{3}\boxed{5}\boxed{2}\boxed{4}\boxed{3}\boxed{1}\boxed{2}\boxed{3}\boxed{5}. \end{aligned}$$

The change of basis corresponding to this storage is given by :

$$\begin{aligned} \texttt{gather2} = \quad & \lambda(i,j) \ |(i=1, j=1).(1) \\ \# \ & \lambda(i,j) \ |(i=1, j=3).(7) \\ \# \ & \lambda(i,j) \ |(i=2, j=2).(5) \\ \# \ & \lambda(i,j) \ |(i=2, j=4).(2) \\ \# \ & \lambda(i,j) \ |(i=3, j=1).(8) \\ \# \ & \lambda(i,j) \ |(i=3, j=3).(3) \\ \# \ & \lambda(i,j) \ |(i=3, j=5).(4) \\ \# \ & \lambda(i,j) \ |(i=4, j=2).(9) \\ \# \ & \lambda(i,j) \ |(i=4, j=4).(6) \\ \# \ & \lambda(i,j) \ |(i=5, j=3).(10) \\ \# \ & \lambda(i,j) \ |(i=5, j=5).(11). \end{aligned} \tag{43}$$

It is a point-to-point function, but it can still be applied to the whole program. The vector cannot be aligned with the first row, as it is sparse, but we can still align it with the diagonal, and produce a new Pei program, following the same steps as in previous section.

3.5 HPF Translation

Just as for the dense program, we would like to have a translation of the Pei program into HPF. However, the translator design is not advanced enough. First of all, the typechecker does not yet deal with functions defined on domain partitions, as it is the case for the functions involved in the sparse program. There is another difficulty in the translation of the reduction. A space-time mapping has to be specified so that there is an order in the sum computation. But our domain is one-dimensional, and time is not a dimension of the geometrical domain, whereas vector $(0,1)$ could be seen as a time direction in the dense program. As we are not able to provide a time direction, the translator cannot translate. At last, another difficulty is that the translator does not handle HPF2 statements for irregular distribution of data. In the case of coordinate storage, alignment cannot be expressed.

However, Fig.#14 shows the code as it could be generated for the program of Fig.#13 if all the previous problems were tackled.

The use of `if` statements implies overhead in the program execution. If no regular pattern could be found in the matrix, there would be much more `if` than in Fig.#14, for in this case functions `spread'`, `shift'`, ... would be point-to-point functions. One way to handle this would be to use indirections arrays computed by the translator instead of `if` statements, generating automatically sparse code with any storages.

4 Other Approaches

The idea of obtaining sparse code by analyzing or transforming dense code has already been developed. For instance, Bik and Wijshof's apprroach [2, 3, 4] is to build a sparse compiler that takes a dense program and compiles it, taking into account the fact that some data are sparse. The sparse data-structure is chosen according to the program. An analysis is done, where the computations involving sparse data structures are detected. `If` statements are inserted to separate the parts involving non-zero elements from the ones only using zero elements, which can sometimes be eliminated. The set of indices referring to the non-zero values of A is denoted $E(A)$, and the function representing the mapping of these values onto the sparse data structure A' is denoted as σ. The following example shows the kind of transformation:

```
                          if (i,j) ∈ E(A)
                            x = A'(σ(i,j))
x = A(i,j)    becomes     else
                            x = 0.
```

The introduced function σ can be viewed as a change of basis in PEI, but the user does not choose it in the case of the sparse compiler. The chosen sparse storage depends both on the structure of the matrix (provided by the user or analyzed, the matrix being read from a file) and on the accesses of these non-zero elements during computations. But only storages where an access direction is available (such as CSR) are allowed, and loops in the original program are replaced by loops whose indices refer to the sparse storage. Unlike PEI, those transformations are not done through functions composition, but by computing indirection arrays that in some way represent σ^{-1}. The idea of representing the change of basis with an array would be a good idea in the code generation phase (as suggested in section 3.5), but this is not the goal of PEI itself, as it is a formal framework working on functions that transform the dense code in a sparse formulation thus proven to be correct.

The approach of Bik and Wijshoff does not clearly solve the parallelization problem. The sparse storage choice should take parallelism into account so that contiguous data still remain contiguous in the case of

```
      program SparseMatVec
      real A(1:3*n-4), V(1:n), B(1:3*n-4)
      real D(1:3*n-4), P(1:3*n-4), S(1:3*n-4), Z(1:3*n-4)
!hpf$ template W(1:3*n-4)
!hpf$ align A,B,P,S,Z with T
!hpf$ align V(1:n) with T(1:n)
      forall (i=1:n) T(i)=V(i)
      forall (i=1:3*n-4)
        if (1.lt.i .and. i.lt.n) D(i)=T(i)
        if (n+2.lt.i .and. i.lt.n+3) D(i)=T(i-n+1)
        if (n+4.lt.i .and. i.lt.3*n-5) then
           if (mod((i-n),2).eq.0) then
              D(i)=T((i-n-1)/2)
           else
              D(i)=T((i-n-1)/2 +3)
        endif
        if (3*n-4.lt.i .and. i.lt.3n-3) D(i)=T(i-2*n+1)
      endforall
!
      forall (i=1:3*n-4) P(i)=A(i)*D(i)
!
      forall (i=1:3*n-4)
        if (1.lt.i .and. 2.lt.i) S(i)=P(i)
        if (i.eq n+1)  S(i)=P(i)
        if (n+4.lt.i .and i.lt.3*n-5) then
          if (mod(i,2) .eq. 0) S(i)=P(i)
        endif
        if (i.eq.3*n-4) then S(i)=P(i)
      endforall
      forall (i=3:n+3)
        if (3.lt.i .and. i.lt.n-2) S(i)=S(i)+S(n+2*i-2)
        if (n-1.lt.i .and. i.lt.n) S(i)=P(i)+S(2*n+i-3)
        if (n+2.lt.i .and. i.lt.n+3) S(i)=P(i)+S(i-n-1)
      endforall
      forall (i=n+4:3*n-5)
        if (mod(i,2) .eq. 1) S(i)=P(i)+S((i-n-1)/2+1)
      endforall
      forall (i=1,n)
        if (1.lt.i .and. i.lt.2) X(i)=S(i-n)
        if (3.lt.i .and. i.lt.n-2) X(i)=S(2*i+n-1)
        if (n-1.lt.i .and. i.lt.n) X(i)=S(i)
       endforall
```

Figure 14:

parallelism. However, they suggest that the sparse compiler could insert directives for a parallel compiler so that parallelism easyly found in the dense code still remains clear in the sparse code. As PEI is designed to representing program in a data parallel context, each point in the domain being a virtual processor, parallelization in this case is easily achieved.

Another approach is to compare sparse data to relational queries. This is the approach of Kotlyar, Pingali and Stodghill [8, 9]. Data accesses in loop nests consist of enumerations of tuples satisfying a given relational query. In the case of matrix-vector multiplication as in Fig.#7, the matrix A can be viewed as a database relation of tuples $\langle i, j, a\rangle$, i and j being rows and columns indices, a being values. The same remark applies to V with $\langle j, v\rangle$. Finally, the iteration space $1 \leq i \leq n, 1 \leq j \leq n$ is a relation R of $\langle i, j\rangle$ tuples. Selecting the set of data for an iteration consists of selecting the non-zero values a and v of A and V for i and j respecting R.

The main problem relies on the efficient evaluation of the queries, that is discovering *joins* to avoid cross-products, but this problem can be overcome. In contrast to the previous work, most of the sparse storage structure is hidden from the compiler. Only access methods are available, hiding the storage implementation. This allows the programmer to choose his own storage methods, provided the access methods are available. The parallelization is done through distributed query [10] in the SPMD programming model, where all global data are partitioned among processors, leading to a local sparse storage. This complicates the evaluation of the relational queries which refer first to the global representation of data. As PEI is a tool that relies on the data parallel programming model we do not come across this peculiar difficulty, because the data only have one representation. The parallelization is natural in our case. However, both approaches are similar in the sense that a high-level framework (functional approach or relational approach) is given that allows the expression of both parallelism and sparsity.

PEI does not itself handle the problem of code generation. It allows to transform the program so that the new formulation is proven equivalent to the original one. Some of the difficulties tackled by both the alternative approaches (such as efficient data accesses) are left to the HPF translator.

5 Conclusion

The presented technique shows the interest of using PEI for program transformation to address sparse computations. The example considered here is based on a tridiagonal matrix, which is a special case of sparse matrix. Unfortunately, sparse matrices do not always have this special shape, and the main difficulty is then to define the different functions used to write the sparse program. It can be achieved by reading the matrices from a Harwell-Boeing formatted file [5], for example.

Several types of storage can be expressed using PEI, among them sparse general pattern [11], compress diagonal storage, jagged diagonal storage or other types of storage described in [1], since a storage in PEI is just a way to map data on a geometrical domain through the convenient change of basis. In fact, the non-zero elements of the matrix can be stored in any chosen order; the vector alignment with the matrix is then done through the application of the change of basis to the whole dense program. It provides a theoretical framework in which the changes applied to the dense program are proven to be correct by composition of functions using formal calculus, independent from routines written in a given language.

Finally, Pei provides parallel code. Instead of scanning the sparse data-structure to perform computation, we consider it parallel data, and we perform parallel computation after proper data alignment has been done. Therefore, Pei can be used in order to build a sparse computation library in the data parallel programming model expressed, for example, in HPF. The given example of sparse matrix vector could be used in the parallelization of a conjugate gradient algorithm (see [1]), for instance. However, the definition of the library requires the definition of a Pei to HPF translator.

References

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[2] A. Bik and H. Wijshoff. Advanced compiler optimizations for sparse computations. *J. of Parallel and Distributed Computing*, 31:14–24, 1995.

[3] Aart J. C. Bik, Peter M. W. Knijnenburg, and Harry A. G. Wijshoff. Reshaping access patterns for generating sparse codes. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 406–420, Ithaca, New York, August 8–10, 1994. Springer-Verlag.

[4] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations:. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):109–126, February 1996. also on LNCS.

[5] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's guide for Harwell-Boeing sparse matrix test problems collection. Tech. Report RAL-92-086, Computing and Information Systems Department, Rutherford Appleton Laboratory, Didcot, UK, 1992.

[6] Stphane Genaud. On deriving HPF code from pei programs. Technical Report RR97-04, ICPS, September 1997.

[7] HPF Forum. *High Performance Fortran Language Specification*, January 1997. Version 2.0.

[8] V. Kotlyar, K. Pingali, and P. Stodghill. A relational approach to the compilation of sparse matrix programs. *Lecture Notes in Computer Science*, 1300:318–, 1997.

[9] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Compiling parallel sparse code for user-defined data structures. In *Proceedings of Eights SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, March 1997.

[10] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Unified framework for sparse and dense SPMD code generation (preliminary report). Technical Report TR97-1625, Cornell University, Computer Science, March 10, 1997.

[11] Serge Petiton and Nahid Emad. *The data parallel programming model: foundations, HPF realization, and scientific applications*, chapter A data parallel scientific computing introduction, pages 45–64. Springer-Verlag Inc., 1996.

[12] Yousef Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. Version 2.

[13] E. Violard and G.-R. Perrin. PEI : a language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 1992.

[14] E. Violard and G.-R. Perrin. PEI : a single unifying model to design parallel programs. *PARLE'93, LNCS*, 694:500–516, 1993.

[15] E. Violard and G.-R. Perrin. Reduction in PEI. *CONPAR'94, LNCS 854*, pages 112–123, 1994.