Automatic Storage Management for Parallel Programs

Vincent Lefebvre and Paul Feautrier
Laboratoire PRiSM, Université de Versailles-St. Quentin,
45, Avenue des États-Unis, 78 035 Versailles cédex, FRANCE.
e-mail : {Vincent.Lefebvre,Paul.Feautrier}@prism.uvsq.fr

Abstract

This article deals with automatic parallelization of static control programs. During the parallelization process the removal of memory related dependences is usually realized by translating the original program into a single assignment form. This total data expansion has a very high memory cost. We present a technique of partial data expansion which leaves untouched the performances of the parallelization process, with the help of algebra techniques given by the polytope model.

Keywords : Automatic Parallelization, Memory Management, Array Dataflow Analysis, Scheduling.

1 Introduction

This article deals with the automatic parallelization technique based on the polytope model. This method can be applied provided that source programs are static control programs, i.e. are limited to `do` loops and assignment to array with affine subscripts. The first step is the extraction of exact dependences by array data flow analysis. All memory related dependences, which are due to reuse of data, are deleted by a total data expansion. The transformed program has the single assignment property and residual dependences constitute the data flow. The program is then parallelized by scheduling, a method which automatically satisfies the sequential constraints inherent in the data flow.
The single assignment form translation has a very high memory cost: memory size is of the same order as the iteration space (for example, matrix multiplication will take $\mathcal{O}(n^3)$ memory). This is clearly unacceptable, and can be ameliorated by producing multiple assignment parallel code. The aim of this paper is to present a new technique for partial data expansion. We show that to any schedule we can associate a parallel program with minimal memory expansion.
In section 2, we describe the polytope method with total data expansion. In Section 3, we present our technique of partial data expansion. We show that this technique can now replace the total expansion in the parallelization process of the polytope method.

2 The polytope method

All techniques and algorithmes described in this section are directly taken from the PAF compiler developped at the university of Versailles by P. Feautrier and his team.

2.1 Framework

2.1.1 $\mathbb{Z}$-polyhedra

A **polyhedron** is the set of all vectors $\vec{x}$ which satisfies a set of linear inequalities. A bounded polyhedron is called a **polytope**. A **$\mathbb{Z}$-module** is a set of integral points generated by integer combination of basis vectors. A **$\mathbb{Z}$-polyhedron** is the intersection of a $\mathbb{Z}$-module and a polyhedron.
The basic problem about $\mathbb{Z}$-polyhedra is the question of their emptiness. It is a linear integer programming question which can be solved by the Gomory cut method which is integrated in the Parametric Integer Programming (PIP) tool. A straightforward application of PIP computes the lexicographic maximum of a $\mathbb{Z}$-polyhedron [5].

2.1.2 Static Control Programs

Static control programs are built from assignment statements and `do` loops. The only data structures are arrays of arbitrary dimensions. Loop bounds and array subscripts must be affine functions in the loop counters and integral structure parameters.
An operation may be named $\langle R, \vec{x} \rangle$ where R is a statement and $\vec{x}$ the iteration vector whose components are the values of the surrounding loop counters. The component p of $\vec{x}$ is the counter of loop p. The iteration vector is constrained by the loop bounds. The iteration domain $\mathcal{D}(R)$ of a statement R, is the set of instances of R and can be described by the conjunction of all inequalities from the surrounding loops. Loop counters are integers, hence iterations domains are set of integer vectors inside polytopes. We will take as running example the program `matrix-vector`:

```
        program matrix-vector
        real s, a(n,n), b(n), c(n)
        integer i,j,n
        do i=1,n
S1        s = 0.
          do j=1,n
S2          s = s + a(i,j)*b(j)
          end do
S3        c(i) = s
        end do
        end
```

This program has three statements from $S1$ to $S3$. The operation $\langle S2, 2, 1 \rangle$ is an execution of statement $S2$ for $i = 2$ and $j = 1$ where i and j are counters of loops surrounding $S2$. The symbolic constant n is a structure parameter. The iteration domain of $S2$ is $\mathcal{D}(S2) = \{i, j \mid 1 \leq i \leq n, 1 \leq j \leq n\}$.

2.1.3 Sequential Execution Order

The lexicographic order is noted $\ll$. The expression $R \lhd S$ indicates that statement R is before statement S in the program text. N_{RS} is the number of loops surrounding both R and S. One has $\vec{x} \ll_p \vec{y} \equiv \vec{x}[1..p] = \vec{y}[1..p] \wedge \vec{x}[p+1] < \vec{y}[p+1]$ and $\ll$ is given by

$$\vec{x} \ll \vec{y} \equiv \bigvee_{p=0}^{|\vec{x}|-1} \vec{x} \ll_p \vec{y} \tag{1}$$

The fact that operation $\langle R, \vec{x} \rangle$ is executed before the operation $\langle S, \vec{y} \rangle$ is written: $\langle R, \vec{x} \rangle \prec \langle S, \vec{y} \rangle$. It is shown in [6] that:

$$\langle R, \vec{x} \rangle \prec \langle S, \vec{y} \rangle \equiv \vec{x} \ll \vec{y} \vee (\vec{x}[1..N_{RS}] = \vec{y}[1..N_{RS}] \wedge R \lhd S) \equiv \bigvee_{p=0}^{N_{RS}} \langle R, \vec{x} \rangle \prec_p \langle S, \vec{y} \rangle \tag{2}$$

where

$$\langle R, \vec{x} \rangle \prec_p \langle S, \vec{y} \rangle \Leftrightarrow \begin{cases} 0 \leq p < N_{RS} : \vec{x} \ll_p \vec{y} \\ p = N_{RS} : \vec{x}[1..N_{RS}] = \vec{y}[1..N_{RS}] \wedge R \lhd S \end{cases} \tag{3}$$

In our running example, we have: $\langle S1, 2 \rangle \prec_0 \langle S2, 3, 1 \rangle$ and $\langle S2, 2, 3 \rangle \prec_0 \langle S2, 3, 1 \rangle$

2.2 Semantic Analysis

2.2.1 Dependences

One must ensure that the parallel program is determinate and gives the same results as the sequential one. Hence one must take into account the dependences which exist between the operations of the source program. To each operation v we associate two sets: $\mathcal{R}(v)$ is the set of memory cells which are read by v; $\mathcal{M}(v)$ is the set of memory cells which are modified by v. Berstein's conditions distinguish three kinds of dependences between v and u, where $v \prec u$. If $\mathcal{M}(v) \cap \mathcal{R}(u) \neq \emptyset$, there is a **flow dependence**, written $v \, \delta \, u$. If $\mathcal{R}(v) \cap \mathcal{M}(u) \neq \emptyset$, there is an **anti-dependence**, written $v \, \overline{\delta} \, u$. If $\mathcal{M}(v) \cap \mathcal{M}(u) \neq \emptyset$, there is an **output dependence**, written $v \, \delta^\circ \, u$. One may be more precise and associate a dependence to a depth p. For instance, if two operations v and u are in flow dependence at depth p, written $v \, \delta_p \, u$, it means that: $v \prec_p u \wedge \mathcal{M}(v) \cap \mathcal{R}(u) \neq \emptyset$.

2.2.2 Array Data Flow Analysis of Static Control Programs

It is well known that output dependences and anti-dependences are artificial, i.e. due to reuse of memory. These kinds of dependences are called **false dependences**.
The real dependences which define the inherent semantic of a program, are a subset of flow dependences: the **direct flow dependences**. A direct flow dependence is a data flow from a definition by an operation v to a use by an operation w of a same memory cell c and provided there is no write on c between the executions of v and w. It means that the value read by w in c is the one produced by v. The remaining flow dependences are artificial dependences too and are called **spurious flow dependences**. The removal of artificial dependences by data restructuring, is called **data expansion**. This technique will be detailed in the next section.
Direct flow dependences are computed by data flow analysis [6]. It must determine for each memory cell c read by an operation w, the last operation in $\prec$ which gives a value to c before the execution of w. This operation is called the *source* of the read:

$$source(c, w) = \max_{\prec} \{v \mid v \, \delta \, w\} \tag{4}$$

The computation of the *source* can be done by PIP (Parametric Integer Programming) algorithm (cf [6] for more details). The result of the analysis is a quasi-affine tree or quast, i.e. a many-level conditionnal in which predicates are tests for the positiveness of affine forms in the loop counters and structure parameters. The leaves are either operation names, or $\perp$. The symbol $\perp$ indicates that the array cell under study is not modified. Sources are gathered in the Data Flow Graph (DFG).
Fig. 1 gives the DFG of the `matrix-vector` program.

memory cell referenced	in operation	source operation
s	$\langle S2, i, j\rangle$	$\begin{cases} \textbf{if } j-2 \geq 0 \\ \textbf{then } \langle S2, i, j-1\rangle \\ \textbf{else } \langle S1, i\rangle \end{cases}$
$a(i,j)$	$\langle S2, i, j\rangle$	$-$
$b(j)$	$\langle S2, i, j\rangle$	$-$
s	$\langle S3, i\rangle$	$\langle S2, i, n\rangle$

Figure 1: The DFG of the `matrix-vector` program

2.3 Program Transformations

The first step is to delete all false dependences and spurious flow dependences by a **total data expansion**. It is realized by translating the source program into a single assignment form. The residual dependences in a such program are the direct flow dependences of the DFG.
The second step is to parallelize the single assignment form program by **scheduling**. The aim is to change the operation execution order of the program, the set of operations and the data flow being left untouched.

2.3.1 Total Data Expansion

Total data expansion gives to the program the **single assignment property**: each memory cell allocated to data will only receive one value produced by one operation during all the execution of the program. In this way, one associates a memory cell to an operation. One can find the algorithm for translating a static control program into a single assignment form in [6]. The first step is a **complete renaming**: for each statement R one associates a specific data structure `InsR`, used to store all values produced by the operations instances of R. Then one **totally expands** all data structures: `InsR` is indexed by the iteration vector of R.

$$R : \texttt{A}[\vec{f}(\vec{x})] = \ldots \quad \text{becomes} \quad R : \texttt{InsR}[\vec{x}] = \ldots$$

Finally one **reconstitutes the data flow** by replacing each rhs reference by its corresponding *source*. If one translates the program `matrix-vector` into single assignment form, one obtains the following code:

```
            program matrix-vector
            real a(n,n), b(n), InsS1(n), InsS2(n,n), InsS3(n)
            do i=1,n
S1            InsS1(i) = 0.
              do j=1,n
S2              InsS2(i,j) = (if (j-2 >= 0) then InsS2(i,j-1) else InsS1(i)) + a(i,j) * b(j)
              end do
S3            InsS3(i) = InsS2(i,n)
            end do
            end
```

Notice that the total data expansion has created an one-dimensionnal array `InsS1` with n elements and a two-dimensionnal array `InsS2` with n^2 elements which replace the scalar s in $S1$ and $S2$.

2.3.2 Parallelization by Scheduling

One computes a time function θ which gives the partial execution order of the parallel program by taking into account the sequential constraints of the data flow. For any operation

u, if $\theta(u)$ is its execution time, one must have:

$$\forall c \in \mathcal{R}(u), \theta(source(c,u)) \ll \theta(u) \tag{5}$$

It defines a set of linear constraints. For complexity reasons finding the exact solution of (5) is not practicable. One limits oneself to affine one-dimensionnal and multi-dimensionnal [7] schedules. In the case of our running example, one can have the following schedule function θ:

$$\left\{ \begin{array}{l} \textbf{if } (j-2 \geq 0) \textbf{ then } \theta(S2,i,j-1) \ll \theta(S2,i,j) \\ \qquad\qquad \textbf{else } \theta(S1,i) \ll \theta(S2,i,j) \\ \theta(S2,i,n) \ll \theta(S3,i) \end{array} \right. \Rightarrow \left\{ \begin{array}{l} \theta(S1,i) = 0 \\ \theta(S2,i,j) = j \\ \theta(S3,i) = n+1 \end{array} \right. \tag{6}$$

An operation front $\mathcal{F}(\vec{t})$ gathers all operations which have the same execution time. The operations in a front can be executed in parallel. Note that an execution time $\theta(v) = \vec{t}$ is in fact a logical time on $\mathbb{N}^d$. One can imagine that it corresponds to the execution of the parallel program with an unbounded number of processors which execute one operation in unit time. Let τ be the set of all possible execution times ($\vec{t} \in \tau \Rightarrow \mathcal{F}(\vec{t}) \neq \emptyset$) enumerated in lexicographic order. The parallel program must enumerate each possible date $\vec{t} \in \tau$. To build the parallel code one must apply affine transformations to the iteration space of the program. When this is done, the operations in the original program are to be executed according to the lexicographical order in the transformed iteration space. The final lexicographical order is the one given by the schedule function. If one translates the `matrix-vector` program scheduled with (6) in Fortran 90, one gets the following code:

```
            program matrix-vector
            real InsS1(n), InsS2(n,n), InsS3(n), a(n,n), b(n)
            do t=0,n+1
              if (t .EQ. 0) then
S1              InsS1(1:n:1)=0.
              end if
              if (t .EQ. 1) then
S2              InsS2(1:n:1,t) = InsS1(1:n:1) + a(1:n:1,t)*b(t)
              end if
              if (t .GE. 2 .AND. t .LE. n) then
S2              InsS2(1:n:1,t) = InsS2(1:n:1,t-1) + a(1:n:1,t)*b(t)
              end if
              if (t .EQ. n+1) then
S3              InsS3(1:n:1) = InsS2(1:n:1,n)
              end if
            end do
            end
```

Notice that total data expansion has induced the split of $S2$ in two different statements in the parallel code.

3 Reduced Data Expansion in Parallelized Programs

Translating the source program into single assignment form has a very high memory cost. It is clear in the case of our running example: from a scalar `s` and an array `c(n)`, one gets three arrays with a data size of $\mathcal{O}(n^2)$.

Our aim is now to define a method of partial data expansion which **reduces the memory expansion** induced by parallelization and **replaces the single assignment form translation** during the parallelization process. The constraint is that the schedule which has been deduced from the DFG should remain valid in the presence of output and anti dependences. An intuitive presentation of the method is given below.

3.1 Optimized Storage Management in Parallelized Programs: an intuitive Approach

One must precise some conventions and notations. Let $\mathcal{V}(v)$ be the value produced by an operation v. Let $\mathcal{C}(v)$ be the memory cell in which $\mathcal{V}(v)$ is stored. Let $\mathcal{U}(v)$ be the set which gathers all operations u such that there is a direct data flow from v to u. $\mathcal{U}(v)$ is the set of all operations which will be executed after v and will read $\mathcal{V}(v)$:

$$\mathcal{U}(v) = \{u \mid \exists\, c \in \mathcal{R}(u),\, source(c,u) = v\} \tag{7}$$

$\mathcal{U}(v)$ is the **utilization set** of v [10].
Let $\mathcal{L}(v)$ be the execution time of the last read of $\mathcal{V}(v)$ in the parallel program. Let $L(v)$ be the subset of operations in $\mathcal{U}(v)$ which execute this last read:

$$\mathcal{L}(v) = \theta(L(v)) = \max \theta(u), u \in \mathcal{U}(v) \tag{8}$$

Consider a memory cell $\mathcal{C}(v)$ during execution of a parallel program in single assignment form. One can distinguish three periods (see Fig. 2):

1. **Period (I)**: the memory cell **stays empty** until the execution of v to which it is associated.
   In our running example, `InsS2[i,j]` (`InsS2[i,j]` $= \mathcal{C}(S2,i,j)$) stays **"empty"** until the execution of $\langle S2,i,j\rangle$ at $\theta(S2,i,j) = j$, if $1 \leq j \leq n \perp 1$.

2. **Period (II)**: the execution of v stores $\mathcal{V}(v)$ in $\mathcal{C}(v)$. The operations of $\mathcal{U}(v)$ read $\mathcal{V}(v)$ until $\mathcal{L}(v)$. During this time, $\mathcal{V}(v)$ is **useful**.
   One has $\mathcal{U}(S2,i,j) = \{\langle S2,i,j+1\rangle\}$. $\mathcal{V}(S2,i,j)$ is read by $\langle S2,i,j+1\rangle$ at $\theta(S2,i,j+1) = j+1$. This time is the last read of $\mathcal{V}(S2,i,j)$: $\mathcal{L}(S2,i,j) = j+1$

3. **Period (III)**: the memory cell is not read anymore after $\mathcal{L}(v)$, nevertheless $\mathcal{V}(v)$ is still in $\mathcal{C}(v)$ until the end of the execution of the parallel program. $\mathcal{V}(v)$ becomes **useless**.
   $\mathcal{V}(S2,i,j)$ becomes useless after $\theta(S2,i,j+1) = j+1$ and stays in `InsS2[i,j]` until the end of the program at $\theta(S3,i) = n+1$

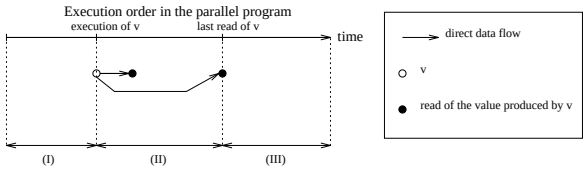

Figure 2: Use of $\mathcal{V}(v)$ in $\mathcal{C}(v)$ in the single assignment parallel program

It is clear that during periods (I) and (III), $\mathcal{C}(v)$ can store others values. If one stores others values in $\mathcal{C}(v)$, output dependences appear in the parallel program. The problem is to define an automatic method for partial data expansion which ensures that the parallel program obtained is valid.

3.2 Related Work

Many papers are devoted to the problem of eliminating false dependences. Some of them try to eliminate these dependences with a reduced memory cost. One can find many techniques

which come from the automatic parallelization community ([1], [2], [9], [12]) or the systolic community ([13], [3]). It is interesting to notice that these techniques are close to data-localization methods ([4], [14]).
Most papers from the automatic parallelization community deal with array privatization. Privatization is a technique that allows each thread to allocate a variable in its private storage. Hence if a loop is transformed into a parallel loop, privatization replaces all original reference to an array `a` by an access to a local array. One can prove that privatization is similar to scalar or array expansion. But privatization may require less space than expansion because it creates one copy per processor and the number of processors cooperating in the execution of the parallel loop is less than the number of iterations [12]. Lam [1], Padua and Tu [12] propose to optimize array privatization with the help of the DFG. If one adapts their method to partial expansion, it consists in maintaining output dependences which duplicate flow dependences.
Another solution has been proposed by the systolic community ([3], [13]). Programs that are taken into account are given in single assignment form. They try to create output dependences which don't invalidate the data flow by estimating the lifetime of each variable. Darte and al. [2] build upon results of Padua who introduced two graph transformations to eliminate false dependences [11]. They give an unified framework for such transformation and prove that the problem of determining a minimal renaming is NP-complete.

3.3 Utility Span of a Value

Our method of partial data expansion is based on the notion of **utility span of a value**. It is clear that the utility span corresponds to the period (II) (see Fig. 2): $\mathcal{V}(v)$ must reside in memory during $\vec{t} \in [\theta(v), \mathcal{L}(v)]$. The utility span of a value is a subsegment of $[\vec{0}, \vec{L}]$ where $\vec{L}$ is the latency i.e. the execution time of the last front executed in the parallel program.

Definition 1 *The utility span of $\mathcal{V}(v)$ is the time span between production of $\mathcal{V}(v)$ and its last read in the parallel program.*

$$\vec{t} \in [\theta(v), \mathcal{L}(v)] \Rightarrow \mathcal{V}(v) \in \mathcal{C}(v) \tag{9}$$

One can estimate the utility span of $\mathcal{V}(S2, i, j)$ in our running example. If $1 \leq i \leq n \wedge 1 \leq j \leq n \perp 1$, then $\mathcal{V}(S2, i, j)$ must reside in $\mathcal{C}(v)$ for $\vec{t} \in [\theta(S2, i, j), \theta(S2, i, j+1)] = [j, j+1]$. Before and after this utility span, $\mathcal{C}(v)$ can store others values without changing the data flow from v to operations in $\mathcal{U}(v)$: one can reintroduce output dependences between v and some others operations.

The atomic entity in our study is not the memory cell $\mathcal{C}(v)$ like in most previous methods, but the value $\mathcal{V}(v)$. The main advantage over the notion of variable lifetime is that it can be applied to programs which are not necessarily in single assignment form.
The next subsection show which are the conditions that an output dependence must verify to be harmless in the parallel program. Such output dependences are called **neutral dependences**.

3.4 Neutral Dependences

Consider two operations v and w. Rule (9) imposes that:

1. $\mathcal{V}(v) \in \mathcal{C}(v)$ for $\vec{t} \in [\theta(v), \mathcal{L}(v)]$
2. $\mathcal{V}(w) \in \mathcal{C}(w)$ for $\vec{t} \in [\theta(w), \mathcal{L}(w)]$

In the case of a program in single assignment form, one has systematically $\mathcal{C}(v) \neq \mathcal{C}(w)$ because there is no output dependence. Optimizing storage means that one introduces memory reuse in the parallel program, i.e. we want to have some operations v and w for which $\mathcal{C}(v) = \mathcal{C}(w)$. It is clear that this is possible iff the basic rule (9) is still verified for v and w in spite of this output dependence. Hence an output dependence is valid in the parallel program if the subsegments which are the utily spans of v and w are separate. Such an output dependence is called **neutral output dependence**.

Definition 2 *An output dependence is neutral for a schedule function θ iff it doesn't change the data flow in the parallel program built with the help of θ.*

One can precisely gives the characteristics of a **neutral output dependence** $v\,\delta^\circ\,w$ in the parallel program (see Fig. 3):

- v **must be executed before** w: $\theta(v) \ll \theta(w)$.
- **there is an access conflict**: $\mathcal{C}(v) = \mathcal{C}(w)$
- **the utility spans are separate**: $\mathcal{L}(v) \ll \theta(w)$

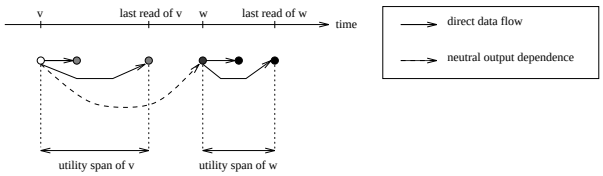

Figure 3: a neutral output dependence $v\,\delta^\circ\,w$ in the parallel program

By extension an output dependence between v and w can be considered as neutral if w is the single operation which constitutes $L(v)$. Here the utility spans of $\mathcal{V}(v)$ and $\mathcal{V}(w)$ are not separate because $\mathcal{L}(v) = \theta(w)$. Nevertheless these two operations can share the same memory cell because w must read $\mathcal{V}(v)$ before computing $\mathcal{V}(w)$. It means that the write of $\mathcal{V}(w)$ occurs after the read of $\mathcal{V}(v)$ by w.

- An output dependence between $\langle S2, i, j\rangle$ and $\langle S2, i+1, j+1\rangle$ would be not neutral, because $\langle S2, i+1, j+1\rangle$ is executed after $\langle S2, i, j\rangle$ and before the end of its utility span. Hence one must have $\mathcal{C}(S2, i, j) \neq \mathcal{C}(S2, i+1, j+1)$ in the parallel program.
- $\langle S2, i, j+2\rangle$ is executed after the utility span of $\langle S2, i, j\rangle$, hence an output dependence between $\langle S2, i, j\rangle$ and $\langle S2, i, j+2\rangle$ would be neutral in the parallel program. Hence one can have $\mathcal{C}(S2, i, j) = \mathcal{C}(S2, i, j+2)$.
- An output dependence between $\langle S2, i, j\rangle$ and $\langle S2, i, j+1\rangle$ would be neutral because one has $\langle S2, i, j+1\rangle = L(S2, i, j)$. The utility spans of $\mathcal{V}(S2, i, j)$ and $\mathcal{V}(S2, i, j+1)$ are not separate, but the two operations can be stored in the same memory cell: we are sure that $\langle S2, i, j+1\rangle$ must read $\mathcal{V}(S2, i, j)$ before writing $\mathcal{V}(S2, i, j+1)$. Hence one can have $\mathcal{C}(S2, i, j) = \mathcal{C}(S2, i, j+1)$.

Notice that if **two operations** v and w belong to the **same operations front**, an output dependence $v\,\delta^\circ\,w$ would be **non neutral** in the parallel program. Hence one must use data expansion to ensure that they are stored in two different memory cells. In fact,

the memory requirement of a parallel program is strongly linked to the parallelism degree (size of operations fronts) given by the schedule function. As we have seen in our running example, the utility span of $\mathcal{V}(S2,i,j)$ for $j < n$ is between $\vec{t} = j$ and $\vec{t} = j+1$. An output dependence between $\langle S2,i,j\rangle$ and $\langle S2,i+1,j\rangle$ would not be neutral because the two operations belong to the same front $\mathcal{F}(\vec{t}) = j$.

To decide if an output dependence is neutral in a parallel program, one must have a precise estimation of the utility span of each value $\mathcal{V}(v)$. This estimation can help us to reconstruct the data space of the program by adjusting data size to utility spans. The final purpose is to build a program with direct flow dependences and neutral output dependences. Our first approach has consisted to maintain neutral output dependences from the original program in its parallel version [8]. But this method is directly dependent on the original data space and can't be used to reduce data size of programs in single assignment form. We have decided to improve our technique to become independent from the original data: with the new method presented in this article, the output dependences existing in the program after partial expansion are not necessarily present in the original version.

3.5 Determinating The Utility Span

Consider an operation $\langle R,\vec{x}\rangle$. One wants to determine the subsegment of $[\vec{0},\vec{L}]$ which corresponds to the utility span of this operation: $[\theta(R,\vec{x}),\mathcal{L}(R,\vec{x})]$. The lower bound of this subsegment is directly given by θ. The problem is to compute the upper bound $\mathcal{L}(R,\vec{x})$. We recall that it is the last execution time in the parallel program of an operation of the utilization set $\mathcal{U}(R,\vec{x})$.
Determining this time uses techniques from data flow analysis. The main difference is that the lexicographic maximum computation is not on the sequential execution order $\prec$, but on the execution order given by the schedule function θ.
Consider two statements R and S:

$$\begin{array}{l} R\ :\ a[\vec{f}(\vec{x})]\ =\ \ldots \\ S\ :\ \ldots\ =\ \ldots a[\vec{h}(\vec{y})]\ldots \end{array}$$

The operation $L_S(R,\vec{x})$ is the last read of $\mathcal{V}(R,\vec{x})$ in the parallel program among the operations instances of S which belong to $\mathcal{U}(R,\vec{x})$. The set of candidates is $\langle S,B_{RS}(\vec{x})\rangle$ which is built by scanning the DFG. Consider the quast which gives all the operations sources of the read reference on a in instances of S. One takes into account each leaf $\langle R,\vec{l}(\vec{y})\rangle$ concerning an instance of R and $P(\vec{y})$ the conjunction of all the predicates which lead to this leaf in the quast. A candidate in $\langle S,B_{RS}(\vec{x})\rangle$, which has this leaf as source of the read $a[\vec{h}(\vec{y})]$ has the following characteristics:

- it corresponds to an existing instance of $S\ :\ \vec{y}\in\mathcal{D}(S)$
- $\langle S,\vec{y}\rangle$ is in $\mathcal{U}(R,\vec{x})$: $source(a[\vec{h}(\vec{y})],\langle S,\vec{y}\rangle) = \langle R,\vec{l}(\vec{y})\rangle \Leftrightarrow \vec{l}(\vec{y}) = \vec{x}$ if $P(\vec{y})$ is verified.

Let $B_{RS}^{\langle R,\vec{l}(\vec{y})\rangle}(\vec{x})$ be this candidate,

$$B_{RS}^{\langle R,\vec{l}(\vec{y})\rangle}(\vec{x}) = \{\vec{y}\,|\,\vec{y}\in\mathcal{D}(S)\ \wedge\ P(\vec{y})\ \wedge\ \vec{l}(\vec{y}) = \vec{x}\,\}$$

All its terms are linear equalities or inequalities, hence $B_{RS}^{\langle R,\vec{l}(\vec{y})\rangle}(\vec{x})$ is a $\mathbb{Z}$-polyhedra. $\langle S,B_{RS}(\vec{x})\rangle$ is the union of all candidates which can be built with the quast source of $a[\vec{h}(\vec{y})]$ in instances of S. Hence the set $\langle S,B_{RS}(\vec{x})\rangle$ is a disjunction of $\mathbb{Z}$-polyhedra. It

is clear that the last operation which reads $\mathcal{V}(R,\vec{x})$ between instances of S is the last one executed according to θ:

$$L_S(R,\vec{x}) = \langle S, \max_{\ll_\theta} B_{RS}(\vec{x}) \rangle \tag{10}$$

All statements which may read the data a must be taken into account. The real last read is their maximum according to θ:

$$L(R,\vec{x}) = \max_{\ll_\theta} L_S(R,\vec{x}) \tag{11}$$

Like the source function, $L(R,\vec{x})$ is a quast. To determine $\mathcal{L}(R,\vec{x})$ one just applies the function θ to each leaf of $L(R,\vec{x})$ except for leaves which are the symbol $\perp$ which are left untouched. The different utility spans are gathered in the Utility Span Graph (USG) which gives to each operations v the utility span of $\mathcal{V}(v)$ and the operation executing the last read of $\mathcal{V}(v)$. The symbol $\perp$ indicates that $\mathcal{V}(v)$ is either useless or an output value. Fig. 4 gives the USG of the `matrix-vector` program.

Operation v	$L(v)$	$\mathcal{L}(v)$	Utility span of $\mathcal{V}(v) = [\theta(v), \mathcal{L}(v)]$
$\langle S1, i\rangle$	$\langle S2, i, 1\rangle$	1	$[0, 1]$
$\langle S2, i, j\rangle$	**if** $j \leq n-1$ **then** $\langle S2, i, j+1\rangle$ **else** $\langle S3, i\rangle$	**if** $j \leq n-1$ **then** $j+1$ **else** $n+1$	**if** $j \leq n-1$ **then** $[j, j+1]$ **else** $[j, n+1]$
$\langle S3, i\rangle$	$-$	$-$	$[n+1, -]$

Figure 4: The USG of the `matrix-vector` program

3.6 Partial Data Expansion

The first step is a **partial array and scalar expansion process** that decides the shape and the index function of each statement left hand side. The second step consists in a **partial renaming process** and decides which are the statements that can share the same data structure in their left hand side.

3.6.1 Partial Array Expansion

The aims of partial array expansion for each statement R are the following:

- We want to build a structure `lhsR` which is specifically associated to the statement R. It will give the shape (number of dimensions and size of each dimension) and the index function which are the data in the left hand side of R in the restructured program.
- The specifications used to build `lhsR` is that if `lhsR` provides memory reuse, i.e. output dependences between some operations instances of R, these output dependences have to be **neutral** in the parallel program. For instance in our running example, if $\vec{F}_{S2}$ is the index function of `lhsS2`:
  - One may have `lhsS2`$[\vec{F}_{S2}(i,j)]$ = `lhsS2`$[\vec{F}_{S2}(i,j+2)] \Leftrightarrow \mathcal{C}(S2,i,j) = \mathcal{C}(S2,i,j+2)$. The output dependence beteween $\langle S2,i,j\rangle$ and $\langle S2,i,j+2\rangle$ would be neutral.
  - One must have `lhsS2`$[\vec{F}_{S2}(i,j)] \neq$ `lhsS2`$[\vec{F}_{S2}(i+1,j)] \Leftrightarrow \mathcal{C}(S2,i,j) \neq \mathcal{C}(S2,i+1,j)$. An output dependence between the two operations would be not neutral in the parallel program.

- The elaboration of `lhsR` must be independent from the original data structure in the lhs of R.

The problem is now to build `lhsR`. One recalls that a neutral output dependence can't kill a value $\mathcal{V}(R,\vec{x})$ during its utility span. To respect this rule for any instance of R, one must take into account the maximum duration that the utility span of $\mathcal{V}(R,\vec{x})$ can have in the parallel program. For an operation $\langle R,\vec{x}\rangle$ this duration is obtained by subtracting the lower bound of its utility span from the upper bound. One writes $d(R,\vec{x})$ this parameter:

$$d(R,\vec{x}) = \mathcal{L}(R,\vec{x}) - \theta(R,\vec{x}) \tag{12}$$

One considers that $\perp \perp \theta(R,\vec{x}) = \perp$. Each leaf of $d(R,\vec{x})$ is a multi-dimensionnal linear expression in term of loop counters and structure parameters.

The maximum duration $D(R)$ that the utility span of instances of R can have, is the maximum value of $d(R,\vec{x})$ on the iteration domain of R:

$$\forall \vec{x} \in \mathcal{D}(R), d(R,\vec{x}) \underline{\ll} D(R) \tag{13}$$

$D(R)$ is a multidimensionnal linear expression in term of structure parameters or the symbol $\perp$. Notice that one considers that if $d(R,\vec{x}) \neq \perp$, then $\perp \ll d(R,\vec{x})$. The maximum utility span durations in the `matrix-vector` program are given in Fig. 5. (9) implies that

Statement R	Utility span duration of an instance of R	Maximum utility span duration on R
$S1$	$d(S1,i) = 1$	$D(S1) = 1$
$S2$	$d(S2,i,j) = \left\{ \begin{array}{l} \textbf{if } j \leq n-1 \\ \textbf{then } 1 \\ \textbf{else } 1 \end{array} \right.$	$D(S2) = 1$
$S3$	$d(S3,i) = -$	$D(S3) = -$

Figure 5: The maximum duration of utility spans in the `matrix-vector` program

$\mathcal{V}(R,\vec{x})$ must be in $\mathcal{C}(R,\vec{x})$ between $\theta(R,\vec{x})$ and $\mathcal{L}(R,\vec{x}) = \theta(R,\vec{x}) + d(R,\vec{x})$. If one wants to protect each instance of R during its utility span, one must build `lhsR` in such a way that (9) is verified for the greatest utility span that an instance of R can have. Hence we have chosen to impose that no value $\mathcal{V}(R,\vec{x})$ can be killed between $\theta(R,\vec{x})$ and $\theta(R,\vec{x}) + D(R)$:

$\mathcal{V}(R,\vec{x}) \in$ `lshR` for $\vec{t}$ in $[\theta(R,\vec{x}), \theta(R,\vec{x}) + D(R)]$ where $\theta(R,\vec{x}) + d(R,\vec{x}) \underline{\ll} \theta(R,\vec{x}) + D(R)$

The algorithm that builds the data structure `lhsR` can be summarized like this:

- One starts with a scalar `lhsR`.
- The elaboration of `lhsR` is iterative, the number of iterations is equal to N_{RR} (number of loops surrounding R). Each iteration is called **partial expansion of R at depth** p where p is the depth of the loop considered ($p \in [0, N_{RR} \perp 1]$).
- A **partial expansion of R at depth** p consists in
  1. Computing the **expansion degree of R at depth** p: E_R^p. It gives the number of elements of a new dimension that one adds to `lhsR`.
  2. Indexing this new dimension of `lhsR`:

     `lhsR[`$\vec{F}'(\vec{x})$`]` becomes `lshR[`$\vec{F}'(\vec{x})$, i_{p+1} mod $E_R^p + 1$`]`

     where $\vec{F}'(\vec{x})$ is the index function built by previous iterations on p; i_{p+1} is the counter of loop $(p+1)$ (from the outer one surrounding R); "mod" is the modulo operator and E_R^p is the expansion degree computed in the previous step.

- At the end of the process, `lhsR` only provides neutral output dependences on $R, \forall p \in N_{RR}$.

The problem is now to compute E^p_R. The partial expansion of R at depth p avoids non neutral output dependences between two operations $\langle R, \vec{x}\rangle$ and $\langle R, \vec{x'}\rangle$ if $\vec{x} \ll_p \vec{x'}$. For an operation $\langle R, \vec{x}\rangle$, we build the set of candidates gathering all the operations $\langle R, \vec{x'}\rangle$ which can't share the same memory cell than $\langle R, \vec{x}\rangle$:

- **the operations exist**: $\vec{x} \in \mathcal{D}(R)$ and $\vec{x'} \in \mathcal{D}(R)$
- **the sequential execution order is**: $\langle R, \vec{x}\rangle \prec_p \langle R, \vec{x'}\rangle$
- **the utility spans are not separate**:

$$[\theta(R,\vec{x}), \theta(R,\vec{x}) + D(R)] \cap [\theta(R,\vec{x'}), \theta(R,\vec{x'}) + D(R)] \neq \emptyset$$

Let be $C^p_{RR}(\vec{x})$ the set of candidates, it can be decomposed in unions of $\mathbb{Z}$-polyhedra. Let $e^{C,p}_R$ be its lexicographic maximum:

$$e^{C,p}_R = \max_{\prec_p} C^p_{RR}(\vec{x})$$

One can't have output dependences between operations $\langle R, \vec{x}\rangle$ and $\langle R, \vec{x'}\rangle$ with:

$$\langle R, \vec{x}\rangle \prec_p \langle R, \vec{x'}\rangle \preceq_p \langle R, \vec{x}_e\rangle = e^{C,p}_R$$

From this follows the inequalities on the iteration vectors:

$$\vec{x}[p+1] < \vec{x'}[p+1] \leq \vec{x}_e[p+1]$$

If one expands `lhsR` at depth p with $E^p_{\langle R,\vec{x}\rangle} = \vec{x}_e[p+1] \perp \vec{x}[p+1] + 1$, we are sure that no non neutral output dependence at depth p can appear for $\langle R, \vec{x}\rangle$. But it must be verified for each instance of R, hence the expension degree E^p_R is the maximum value that $E^p_{\langle R,\vec{x}\rangle}$ can have for $\vec{x} \in \mathcal{D}(R)$:

$$E^p_R = \max_{\vec{x} \in \mathcal{D}(R)} E^p_{\langle R,\vec{x}\rangle} \tag{14}$$

Fig. 6 indicates the expansion degree and the structures `lhs` that must be set in the `matrix-vector` program. There can't be output dependences on S1 and S2 at depth

Statements	Expansion degrees	Final data structure	Final lhs
S1	$E^0_{S1} = n$	`lhsS1[n]`	`lhsS1[i] = ...`
S2	$E^0_{S2} = n$		
	$E^1_{S2} = 0$	`lhsS2[n]`	`lhsS2[i] = ...`
S3	$E^0_{S3} = n$	`lhsS3[n]`	`lhsS3[i] = ...`

Figure 6: The final results of the partial array expansion for the `matrix-vector` program

0, hence `lhsS1` is fully expanded and `lhsS2` becomes an one-dimensionnal array with n elements. But all output dependences on $S2$ at depth 1 will be neutral in the parallel program, hence there is no expansion at depth 1 for $S2$. Notice that for the last statement one leaves untouched the shape of the array in the lhs of $S3$ even if its values are never read. It is due to the fact that it stores the final results of the program. In fact if $D(R) = \perp$ then

the shape of `lhsR` must be at least the same that the original data structure in the lhs of R, i.e. the elaboration of `lhsR` starts in this specific case with the original datum. Finally `lhsR` must be partially expanded in such a way that all the instances of R which belong to a same front must be stored in different memory cells, i.e. the partial expansion is here totally dependent from the parallelism degree.

3.6.2 Partial Renaming

The partial renaming process must decide if two different statements can share the same data structure. Consider two statements R and T. Partial expansion builds two structures `lhsR` and `lhsT` which can have different shapes. If at the end of the renaming process R and T are authorized to share the same array, this one would have to be the rectangular hull of `lhsR` and `lhsT`: `lhsR-T`. It is clear that these two statements can share the same data iff this sharing does not generate non neutral dependence between R and T with `lhsR-T` in the left hand side of the two statements. Let $\vec{F}_{R-T}$ be the index function of `lhsR-T`. One must verify for each operation $\langle R, \vec{x} \rangle$ and $\langle T, \vec{z} \rangle$ that would be in output dependence (i.e. $\vec{F}_{R-T}(\vec{x}) = \vec{F}_{R-T}(\vec{z})$) that:

1. $\mathcal{V}(R, \vec{x})$ can't be killed by $\langle T, \vec{z} \rangle$ before the end of its utility span:

$$\theta(R, \vec{x}) \underline{\ll} \theta(T, \vec{z}) \underline{\ll} \theta(R, \vec{x}) + D(R)$$

2. $\mathcal{V}(T, \vec{z})$ can't be killed before by $\langle R, \vec{x} \rangle$ before the end of its utility span:

$$\theta(T, \vec{z}) \underline{\ll} \theta(R, \vec{x}) \underline{\ll} \theta(T, \vec{z}) + D(T)$$

As in the case of partial expansion, one can decompose candidates sets in disjunctions of $\mathbb{Z}$-polyhedra. All these $\mathbb{Z}$-polyhedra must be empty for this transformation to be legal. If there are no integral solutions, R and T can share the same data structure else they can't. Finding the minimal number of renaming is a NP-complete problem (see [2]). Our method consists in building a graph similar to an interference graph as used in code genaration process of a classical compiler to optimize registers allocation. In this graph, each vertex represents a statement of the program. There is an edge between two vertices R and T iff it has been shown that they can't share the same data structure in their left hand side: there is at least one non neutral output dependence $R \, \delta_p^\circ \, T$. Then one applies on this graph a greedy coloring algorithm. Finally it is clear that vertices that have the same colour can have the same data structure in their lhs.

In the `matrix-vector` program there is no edge in the interference graph. It means that $S1$, $S2$ and $S3$ can have the same array in their lhs. This one must be the rectangular hull of `lhsS1`, `lhsS2` and `lhsS3`, i.e. an one-dimensionnal array with n elements. In the transformed program we try to reuse the original variables as soon as possible, especially for variables which store output values like the array c. Hence this one is maintained because it exactly corresponds to the storage requirement of the parallel program. One just has to reconstrut the data flow. One replaces all rhs references by its corresponding source:

- replace a leaf of a quast of the form $\langle R, \vec{l}(\vec{y}) \rangle$ by `A[`$\vec{F}_R(\vec{l}(\vec{y}))$`]` where `A` is the data structure in the lhs of R built by partial array expansion and partial renaming;
- replace a void leaf $\perp$ by the original source reference

For the `matrix vector` program one gets:

```
        program matrix-vector
        real a(n,n), b(n,n), c(n)
        integer i,j,n
        do i=1,n
S1         c(i) = 0.
          do j=1,n
S2          c(i) = c(i) + a(i,j)*b(j)
          end do
S3        c(i) = c(i)
        end do
        end
```

It is clear that the statement $S3$ has become useless after the fusion of c and s and can be removed in the parallel program. If one builds the parallel program with (6) as schedule function without the statement $S3$, we find in Fortran 90:

```
        program matrix-vector
        real a(n,n), b(n), c(n)
        integer i,j,n
        do t=0,n
          if (t .EQ. 0) then
S1          c(1:n:1) = 0.
          end if
          if (t .GE. 1 .AND. t .LE. n) then
S2          c(1:n:1) = c(1:n:1) + a(1:n:1,t)*b(t)
          end if
        end do
        end
```

Our first aim has been reached, our method can effectively reduce the memory cost in the data expansion process of static control programs (see Fig. 7). You can see in this example that the data size is less in the parallel program than the original data size. The simplification of memory access can in some cases reduce the complexity of the parallel code. The primitive latency $\vec{L} = n + 1$ given by (6) is now reduced to $\vec{L} = n$ with the removal of $S3$. Moreover partial data expansion may avoid the split of some statements like the case of $S2$ in the Fortran 90 code in single assignment form.

Statements	Original data	After total expansion	After partial expansion
S1	s	InsS1[n]	c[n]
S2		InsS2[n,n]	
S3	c[n]	InsS3[n]	

Figure 7: The data structures generated by total and partial data expansion

3.7 A second Example: Cholesky Factorization

In this subsection, we present the results we obtain with the Cholesky factorization algorithm. The sequential version that we start from is:

```
        program choles
        integer i, j, k
        real x
```

```
        real a(n,n), p(n)
        do i=1,n
S1        x = a(i,i)
          do k = 1, i-1
S2          x = x - a(i,k)**2
          end do
S3        p(i) = 1.0/sqrt(x)
          do j = i+1, n
S4          x = a(i,j)
            do k=1,i-1
S5            x = x - a(j,k) * a(i,k)
            end do
S6          a(j,i) = x * p(i)
          end do
        end do
        end
```

The original data storage are a scalar `x`, one-dimensionnal array `p` and a two-dimensionnal array `a` respectively with n and n^2 elements. Fig, 8 gives the DFG of the `choles` program. If one translates this program into a single assignment form by **total data expansion**

memory cell referenced	in operation	source operation
$a(i,i)$	$\langle S1,i\rangle$	$-$
x	$\langle S2,i,k\rangle$	$\begin{cases}\textbf{if } k-2\geq 0\\ \textbf{then } \langle S2,i,k-1\rangle\\ \textbf{else } \langle S1,i\rangle\end{cases}$
$a(i,k)$	$\langle S2,i,k\rangle$	$\langle S6,k,i\rangle$
x	$\langle S3,i\rangle$	$\begin{cases}\textbf{if } i-2\geq 0\\ \textbf{then } \langle S2,i,i-1\rangle\\ \textbf{else } \langle S1,i\rangle\end{cases}$
$a(i,j)$	$\langle S4,i,j\rangle$	$-$
x	$\langle S5,i,j,k\rangle$	$\begin{cases}\textbf{if } k-2\geq 0\\ \textbf{then } \langle S5,i,j,k-1\rangle\\ \textbf{else } \langle S4,i,j\rangle\end{cases}$
$a(j,k)$	$\langle S5,i,j,k\rangle$	$\langle S6,k,j\rangle$
$a(i,k)$	$\langle S5,i,j,k\rangle$	$\langle S6,k,i\rangle$
x	$\langle S6,i,j\rangle$	$\begin{cases}\textbf{if } i-2\geq 0\\ \textbf{then } \langle S5,i,j,i-1\rangle\\ \textbf{else } \langle S4,i,j\rangle\end{cases}$
$p(i)$	$\langle S6,i,j\rangle$	$\langle S3,i\rangle$

Figure 8: The DFG of the `choles` program

one gets:

```
      PROGRAM choles
          real a(n,n)
          real InsS1(n)
          real InsS2(n,n-1)
          real InsS3(n)
          real InsS4(n,n-1)
          real InsS5(n,n-1,n-1)
          real InsS6(n,n-1)
          integer n,i,j,k
          DO i = 1,n,1
S1          InsS1(i) = a(i,i)
            DO k = 1,i-1,1
```

```
S2            InsS2(i,k) = (if (k-2 >= 0) then InsS2(i,k-1) else InsS1(i)) - InsS6(k,i) ** 2
            END DO
S3          InsS3(i) = 1./sqrt(if (k-2 >= 0) then InsS2(i,j-1) else InsS1(i))
            DO j = i+1,n,n
S4            InsS4(i,j) = a(i,j)
              DO k = 1,i-1,1
S5              InsS5(i,j,k) = (if (k-2 >=0) then InsS5(i,j,k-1) else InsS4(i,j)) - InsS6(k,j) * InsS6(k,i)
              END DO
S6            InsS6(i,j) = (if (i-2 >= 0) then InsS5(i,j,j-1) else InsS4(i,j)) * InsS3(i)
            END DO
          END DO
      END
```

You can see that total data expansion has created six arrays and the data size is of $\mathcal{O}(n^3)$ instead of $\mathcal{O}(n^2)$ in the original version.
The schedule function computed by the PAF compiler is given by (15).

$$\begin{cases} \theta(S1,i) = 0 \\ \theta(S2,i,k) = 3k \\ \theta(S3,i) = 3i-2 \\ \theta(S4,i,j) = 0 \\ \theta(S5,i,j,k) = 3k \\ \theta(S6,i,j) = 3i-1 \end{cases} \tag{15}$$

With the help of the DFG and the schedule function (15), we have applied our technique of partial data expansion. The first step is the estimation of the utility span of each value and the construction of the USG (see Fig. 9)

Operation v	$L(v)$	$\mathcal{L}(v)$	Utility span of $\mathcal{V}(v) = [\theta(v), \mathcal{L}(v)]$
$\langle S1,i\rangle$	$\langle S2,i,1\rangle$	3	$[0,3]$
$\langle S2,i,k\rangle$	$\begin{cases} \textbf{if } k \le i-2 \\ \textbf{then } \langle S2,i,k+1\rangle \\ \textbf{else } \langle S3,i\rangle \end{cases}$	$\begin{cases} \textbf{if } k \le i-2 \\ \textbf{then } 3k+3 \\ \textbf{else } 3k+1 \end{cases}$	$\begin{cases} \textbf{if } k \le i-2 \\ \textbf{then } [3k,3k+3] \\ \textbf{else } [3k,3k+1] \end{cases}$
$\langle S3,i\rangle$	$\langle S6,i,j\rangle$	3i-1	$[3i-2,3i-1]$
$\langle S4,i,j\rangle$	$\langle S5,i,j,1\rangle$	3	[0,3]
$\langle S5,i,j,k\rangle$	$\begin{cases} \textbf{if } k \le i-2 \\ \textbf{then } \langle S5,i,j,k+1\rangle \\ \textbf{else } \langle S6,i,j\rangle \end{cases}$	$\begin{cases} \textbf{if } k \le i-2 \\ \textbf{then } 3k+3 \\ \textbf{else } 3k+2 \end{cases}$	$\begin{cases} \textbf{if } k \le i-2 \\ \textbf{then } [3k,3k+3] \\ \textbf{else } [3k,3k+2] \end{cases}$
$\langle S6,i,j\rangle$	$\mathcal{U}(S6,i,j)$	3i	[3i-1,3i]

Figure 9: The USG of the `choles` program

From the USG, one computes the maximum duration that an utility span of a value can have (see Fig. 10). Finally the process of data restructuring by a partial data expansion gives the structures `lhs` of Fig. 11 and the Interference Graph of Fig. 12. After a partial renaming, one finally finds that on one hand $S1$ and $S2$ and on the other hand $S4$ and $S5$ can share the same data structure in their lhs. One finally gets the following code:

```
PROGRAM choles
    integer i,j,k,n
    real x(n)
    real a(n,n)
    real p(n)
    real Var1(n,n-1)
    DO i = 1,n,1
```

Statement R	Utility span duration of an instance of R	Maximum utility span duration on R
$S1$	$d(S1,i) = 3$	$D(S1) = 3$
$S2$	$d(S2,i,k) = \begin{cases} \textbf{if } k \leq i-2 \\ \textbf{then } 3 \\ \textbf{else } 1 \end{cases}$	$D(S2) = 3$
$S3$	$d(S3,i) = 1$	$D(S3) = 1$
$S4$	d(S4,i,j) = 3	D(S4) = 3
$S5$	$d(S5,i,j,k) = \begin{cases} \textbf{if } k \leq i-2 \\ \textbf{then } 3 \\ \textbf{else } 2 \end{cases}$	$D(S5) = 3$
$S6$	d(S6,i,j) = 1	D(S6) = 1

Figure 10: The maximum duration of each utility span in the `choles` program

Statements	Expansion degrees	Final data structure	Final lhs
S1	$E^0_{S1} = n$	`lhsS1[n]`	`lhsS1[i] = ...`
S2	$E^0_{S2} = n$		
	$E^1_{S2} = 0$	`lhsS2[n]`	`lhsS2[i] = ...`
S3	$E^0_{S3} = n$	`lhsS3[n]`	`lhsS3[i] = ...`
S4	$E^0_{S4} = n$		
	$E^1_{S4} = n-1$	`lhsS4[n,n-1]`	`lhsS4[i,j] = ...`
S5	$E^0_{S5} = n$		
	$E^1_{S5} = n-1$		
	$E^2_{S5} = 0$	`lhsS5[n,n]`	`lhsS5[i,j] = ...`
S6	$E^0_{S6} = n$		
	$E^1_{S6} = n-1$	`lhsS6[n,n-1]`	`lhsS6[i,j] = ...`

Figure 11: The results of the partial array expansion for the `choles` program

```
S1            x(i) = a(i,i)
              DO k = 1,i-1,1
S2                x(i) = x(i) - a(k,i) ** 2
              END DO
S3            p(i) = 1./sqrt(x(i))
              DO j = i+1,n,1
S4                Var1(i,j) = a(i,j)
                  DO k = 1,i-1,1
S5                    Var1(i,j) = Var1(i,j) - a(k,j) * a(k,i)
                  END DO
S6                a(i,j)= Var1(i,j) * p(i)
              END DO
          END DO
      END
```

One can see that the expansion is limited to the scalar `x` which gets one more dimension with n elements, and to create a two-dimensionnal array `Var1` with $n * (n \perp 1)$ elements. With total expansion the data space is of $\mathcal{O}(n^3)$, with partial expansion it is only of $\mathcal{O}(n^2)$. Moreover partial data expansion has generated no conditionnal expression. Fig. 13 gives a comparaison of results obtained by applying different existing methods of reduced memory

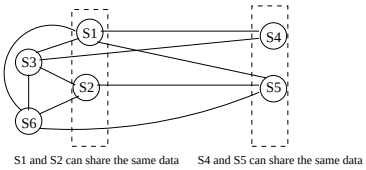

Figure 12: The interference graph of the `choles` program

expansion. The first column is the statements list. The others columns give the shape of different data structures that one finds in the lhs of statements: in the source program (1), in the single assignment form program (2), in the version restructured by Rajopadhye method applied on the single assignment form version (3) (cf [13]) and by our technique (4).

Statements	(1)	(2)	(3)	(4)
S1	`x`	`InsS1[n]`	`InsS1[n]`	`x[n]`
S2		`InsS2[n,n-1]`	`InsS2[n]`	
S4		`InsS4[n,n-1]`	`InsS4[n,n-1]`	`Var1[n,n-1]`
S5		`InsS5[n,n-1,n-1]`	`InsS5[n,n-1]`	
S3	`p[n]`	`InsS3[n]`	`InsS3[n]`	`p[n]`
S6	`a[n,n]`	`InsS6[n,n-1]`	`InsS6[n,n-1]`	`a[n,n]`

Figure 13: Comparaison with others methods

4 Conclusion

In the PAF compiler, our method can now replace the translation into a single assignment form. The parallelization process is now applied in this order:

1. Array data flow analysis
2. Scheduling for the real flow dependences.
3. Partial memory expansion
4. Construction of the parallel program

Our aim has been reached, our method can effectively reduce the memory cost in the data expansion process of static control programs. Morover we have obtained two important results:

1. Our performances are strongly linked to the parallelism degree given by the schedule function. The better the parallelism, the higher the memory cost and conversely. This can be explained simply in the following way. The mean degree of parallelism F is simply the mean size of the fronts. There must be no output dependence in a front, hence all operations must write in a different location. Hence we may control the memory expansion and improve our results by adjusting the schedule to the architecture. We recall that we have considered that the parallel program will be executed on

a target architecture with an unbounded number of processors. This program cannot in general be executed directly: the size of the fronts is in fact limited by the real parallelism provided by the target architecture. suppose that the target architecture is a pipeline Cray processor. In this case, all operations of a front are to be instances of the same statement. Moreover the size of the front is limited to the size of vector registers, 64 for instance. One can adjust the schedule in such a way that:

- There is only one parallel loop by loop nest.
- No front has more operations than 64 (this a variant of the strip mining technique).

In the case of the `matrix-vector` program, it imposes the following multidimensionnal schedule function:

$$\begin{cases} \theta(S1,i) = \begin{pmatrix} \lfloor \frac{i}{64} \rfloor \\ 0 \end{pmatrix} \\ \theta(S2,i,j) = \begin{pmatrix} \lfloor \frac{i}{64} \rfloor \\ j \end{pmatrix} \\ \theta(S3,i) = \begin{pmatrix} \lfloor \frac{i}{64} \rfloor \\ n+1 \end{pmatrix} \end{cases}$$

With this schedule, we find with our method of partial expansion that the expansion degrees at depth 0 for $S1$ and $S2$ are $E^0_{S1} = E^0_{S2} = 64$. The array in the lhs of $S1$ and $S2$ has then one dimension with only 64 elements instead of n. In this case, the Fortran 90 code generated is:

```
        program matrix-vector
        real a(n,n), b(n), c(n), s(64)
        integer i,j,n
        do t0 = 0,n-1,64
          do t1=0,n+1
            if (t1 .EQ. 0) then
S1            s(1:min(64,n-t0):1) = 0.
            end if
            if (t1 .GE. 1 .AND. t1 .LE. n) then
S2            s(1:min(64,n-t0):1) = s(1:min(64,n-t0):1) + a(t0+1:t0+min(64,n-t0):1,t1)*b(t1)
            end if
            if (t1 .EQ. n+1) then
S3            c(t0+1:t0+min(64,n-t0):1) = s(1:min(64,n-t0):1)
            end if
          end do
        end do
        end
```

Notice that the statement $S3$ has not been removed because an array of n elements is still needed in the lhs of $S3$. This approach like array privatization technique takes into account the fact that the real parallelism provided by the target architecture is less than the number of iterations.

2. Our method can be used to reduce data space of parallel programs directly provided in **single assignment form**. Consider for instance that the original version of the `matrix-vector` program is given in single assignment form. With the schedule function (6), one reduces the original data to an one-dimensionnal array with n elements like in Fig. 7. It means that our method can reduce the original data size of the program if the memory requirement necessary for the schedule function is less than

the original data size. This property is due to the fact that except for data storing output values, our partial data expansion is completly independent from the orginal program.

References

[1] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. *Array data-flow analysis and its use in array privatization.* In Principles of Programming Languages, 1993.

[2] P.Y Calland, A. Darte, Y. Robert, F. Vivien. *On the removal of anti and output dependences.* Technical report RR96-04, laboratoire LIP - École normale supérieure de Lyon, France - Feb 1996.

[3] Z. Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle.* Thèse de l'université de Rennes I, France - chapitre IV - 1993, numéro d'ordre 957.

[4] C. Eisenbeis, W. Jalby, D. Windheiser, and F. Bodin. *A strategy for array management in local memory.* In Proc. of the 4th Worlshop on Languages and Compilers for Parallel Computing, Aug. 1991.

[5] P. Feautrier. *Parametric integer programing.* RAIRO Recherche opérationnelle, 22:243-268, Sept 1988.

[6] P. Feautrier. *Dataflow Analysis of Array and Scalar References.* Int. J. of Parallel Programming, 20(1):23-53, February 1991.

[7] P. Feautrier. *Some efficient solutions to the affine scheduling problem part II : multidimensional time.* Int J. of Parallel Programming, 21(6):389-420, December 92.

[8] V. Lefebvre and P. Feautrier. *Storage Management in Parallel Programs.* In IEEE, editor, Fith Euromicro Workshop on Parallel and Distributed Processing - PDP'97, pages 181-188. London. United Kingdom, Jan 1997. IEEE Computer Society Press.

[9] Z. Li, G. and G. Lee. *Symbolic array dataflow analysis for array privatization and program parallelization.* In Supercomputing 95, 1995.

[10] C. Mongenet. *Data compiling for system of uniform recurrence equations.* Technical report 94/11, ICPS - University Louis Pasteur of Strasbourg, France. 1994.

[11] D. A. Padua and M. J. Wolfe. *Adavanced Compiler Optimizations for Supercomputers.* In Communications of the ACM, 29(12):1184-1201, December 1986.

[12] P. Tu and D. Padua. *Array privatization for shared and distributed memory machines.* In Proc. Third Workshop on Languages and Compilers for Distributed Memory Machines, Boulder, Colorado 1992.

[13] S. Rajopadhye and D. Wilde. *Memory Reuse Analysis in the Polyhedral Model.* In Bougé, Fraignaud, Mignotte and Robert, editors, Euro-Par'96 Parallel Processing, Vol I, pages 389-397. Springer-Verlag, LNCS 1123, August 1996.

[14] M. Wolf and M. Lam. *A data locality optimizing algorithm.* In Proc. ACM SIGPLAN 91 Conf. on Programming Language Design and Implementation, June 1991.