Storage Management in Parallel Programs

Vincent Lefebvre
Laboratoire PRiSM
Université de Versailles
45, avenue des Etats-Unis, 78 035 Versailles Cédex, France
Vincent.Lefebvre@prism.uvsq.fr

Paul Feautrier
Laboratoire PRiSM
Université de Versailles
45, avenue des Etats-Unis, 78 035 Versailles Cédex, France
Paul.Feautrier@prism.uvsq.fr

Abstract

We have been interested in this article on the data stuctures generation as part of the polyedric technique designed in PAF (Paralléliseur Automatique pour Fortran). The removal of dependences which are not data flows in a program is generally realized by a total memory expansion of data structures. We present a new technique which allows to reduce the memory cost by expanding carefully selected parts of code only. It consists in limiting the memory expansion process in accordance with contraints imposed by the schedule determined for the parallel program.

1 Introduction

The polyedric method, an automatic parallelization technique, uses explicit schedules. A schedule has to satisfy constraints which are given by dataflow analysis. The goal is to determine the execution date of each operation of the source program. Operations which have the same execution date are gathered in wavefronts, which can be executed in parallel. Dependences which don't belong to the dataflow are called false dependences. A partial removal of false dependences, is the price to pay to preserve the correctness of the parallel program. It is realized by data expansion. One generally builds a single assignment form for the source program. Total data expansion has a high memory cost. For instance, in matrix multiplication, the single assignment form has a data space of $O(n^3)$ memory words, instead of $O(n^2)$ in the classical version. This paper presents a new technique which limits memory expansion in accordance with contraints imposed by the schedule of the parallel program. We will first restate several classical techniques of program semantic analysis (array dataflow analysis) and transformations (scheduling, existing memory management techniques). Finally, we will present our optimized storage technique for parallel programs.

2 Semantic Analysis of Static Control Programs

2.1 Static Control Programs

We focuse on automatic parallelization of static control programs. For Static control programs, one may describe the set of operations which are going to be executed in a given program run. Let be E the operations set of a program. 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 are affine functions in the loop counters and integral structure parameters. An operation is one execution of a statement. It may be named $\langle R, \vec{x} \rangle$ where R is a statement and $\vec{x}$ the iteration vector built from the surrounding loop counters (from the outside to the inside). 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 for the surrounding loops. It gives values that the iteration vector $\vec{x}$ can have. We will take as a running example the sequential program of figure (1).

```
      PROGRAM scalar
      INTEGER s,i,j,n
      DO i = 1,n
{S1}    s = 0
        DO j = 1,n
{S2}      s = s + 1
        ENDDO
      ENDDO
      END
```

Figure 1. The source program

2.2 Sequential Execution Order

Let us introduce the following notations.

- The k-th entry of vector $\vec{x}$ is denoted by $\vec{x}[k]$.
- The subvector built from component k to l is written as: $\vec{x}[k..l]$.
- 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.

The fact that operation $\langle R, \vec{x} \rangle$ is executed before operation $\langle S, \vec{y} \rangle$ is written: $\langle R, \vec{x} \rangle \prec \langle S, \vec{y} \rangle$. It is shown in [5] that:

$$\begin{array}{c} \langle R, \vec{x} \rangle \prec \langle S, \vec{y} \rangle \equiv \\ \vec{x}[1..N_{RS}] \ll \vec{y}[1..N_{RS}] \vee (\vec{x}[1..N_{RS}] = \vec{y}[1..N_{RS}] \wedge R \lhd S) \end{array} \quad (1)$$

The sequential order can be split with respect to depths:

$$\langle R, \vec{x} \rangle \prec \langle S, \vec{y} \rangle \equiv \bigvee_{p=0}^{N_{RS}} \langle R, \vec{x} \rangle \prec_p \langle S, \vec{y} \rangle \quad (2)$$

where

$$\begin{array}{c} 0 \leq p < N_{RS} : \\ \langle R, \vec{x} \rangle \prec_p \langle S, \vec{y} \rangle \Leftrightarrow (\vec{x}[1..p] = \vec{y}[1..p]) \wedge (\vec{x}[p+1] < \vec{y}[p+1]) \end{array} \quad (3)$$

$$\langle R, \vec{x} \rangle \prec_{N_{RS}} \langle S, \vec{y} \rangle \Leftrightarrow \vec{x}[1..N_{RS}] = \vec{y}[1..N_{RS}] \wedge R \lhd S \quad (4)$$

2.3 Dependences

Two operations $\langle R, \vec{x} \rangle$ and $\langle S, \vec{y} \rangle$ are independent if their order of execution can be reversed without changing the global effect on the program store. If not, the operations are said to be dependent. The goal of automatic parallelization is to build a parallel program which exactly gives the same results as the sequential program. $\mathcal{R}(R, \vec{x})$ is the set of memory cells which are read by $\langle R, \vec{x} \rangle$ and $\mathcal{M}(R, \vec{x})$, is the set of memory cells which are modified by $\langle R, \vec{x} \rangle$. Supposing for instance that $\langle R, \vec{x} \rangle \prec \langle S, \vec{y} \rangle$, one can distinguish three kinds of dependences:

- **flow dependence** ($\mathcal{M}(R, \vec{x}) \cap \mathcal{R}(S, \vec{y}) \neq \emptyset$, written $\langle R, \vec{x} \rangle \, \delta \, \langle S, \vec{y} \rangle$);
- **anti-dependence** ($\mathcal{R}(R, \vec{x}) \cap \mathcal{M}(S, \vec{y}) \neq \emptyset$, written $\langle R, \vec{x} \rangle \, \overline{\delta} \, \langle S, \vec{y} \rangle$);
- **output dependence** ($\mathcal{M}(R, \vec{x}) \cap \mathcal{M}(S, \vec{y}) \neq \emptyset$, written $\langle R, \vec{x} \rangle \, \delta^o \, \langle S, \vec{y} \rangle$).

One may be more precise and associate a dependence to a depth p. For instance, if one writes $\langle R, \vec{x} \rangle \, \delta_p \, \langle S, \vec{y} \rangle$, it indicates that $\mathcal{M}(R, \vec{x}) \cap \mathcal{R}(S, \vec{y}) \neq \emptyset \wedge \langle R, \vec{x} \rangle \prec_p \langle S, \vec{y} \rangle$.

2.4 Array Dataflow Analysis

The sole real dependences inherent to the algorithm are direct flow dependences from a definition to a use of the same memory cell (data flows). All others dependences which are called false dependences, are due to memory reuse and can be deleted by data expansion. Direct flow dependences are detected by dataflow analysis technique. If a memory cell c is read in an operation $\langle S, \vec{y} \rangle$, dataflow analysis determines the latest writing into c, which is given by the $source$ function [5]:

$$source(c, \langle S, \vec{y} \rangle) = \max_{\prec} \{ \langle R, \vec{x} \rangle \in E \mid \langle R, \vec{x} \rangle \delta \langle S, \vec{y} \rangle \} \quad (5)$$

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 and leaves are either operation names, or $\perp$. $\perp$ indicates that the array cell under study is not modified. For our example, we have:

$$source(s, \langle S2, i, j \rangle) = \left\{ \begin{array}{l} \textbf{If } j \geq 2 \\ \textbf{Then } \langle S2, i, j-1 \rangle \\ \textbf{Else } \langle S1, i \rangle \end{array} \right. \quad (6)$$

3 Program Transformations

3.1 Parallelization by Scheduling

From constraints given by dataflow analysis, one deduces a schedule which gives a logical execution time to each operation of the source program. It must also respect the constraints implied by the $source$ functions. If $\theta(S, \vec{y})$ is the schedule of $\langle S, \vec{y} \rangle$, one must have:

$$\forall \langle S, \vec{y} \rangle \in E, \forall c \in \mathcal{R}(S, \vec{y}) : \theta(source(c, \langle S, \vec{y} \rangle)) \ll \theta(S, \vec{y}) \quad (7)$$

For complexity reasons, finding the exact solution of (7) is not practicable. One limits oneself to affine one-dimensionnal ([6]) or multidimensionnal schedules ([7]). In the case of our example, one must have:

$$(\textbf{if } (j \geq 2) \textbf{ then } \theta(S2, i, j-1) \textbf{ else } \theta(S1, i)) \ll \theta(S2, i, j) \quad (8)$$

One may show that $\theta(R, i) = 0$ and $\theta(S, i, j) = j$ is the best schedule for our example, i.e gives the largest operations fronts. From a schedule given by θ, one deduces operations fronts:

$$\mathcal{F}(\vec{t}) = \{ \langle R, \vec{x} \rangle \in E \mid \theta(R, \vec{x}) = \vec{t} \} \quad (9)$$

There is no dataflow between operations of a given front. Hence, all such operations can be executed in parallel. The parallel program must enumerate all lexicographical executions dates :

$$\begin{array}{ll}\{\,\vec{t}\,| & \vec{t} \in \tau \\ & \text{execute in parallel operations in } \mathcal{F}(\vec{t}) \\ & \text{synchronize} \\ \}\end{array} \quad (10)$$

The set τ is the lexicographical enumeration of each possible execution date.

3.2 Changing Data Structures

However, using any execution order which satisfies (7) for constructing a parallel program will give an incorrect result, because output dependences, anti-dependences and spurious flow dependences (flow dependences which are not dataflows) have not been taken into account. One can get rid of these false dependences by data expansion. Several techniques have been proposed in the litterature.

3.2.1 Total Memory Expansion

The easiest solution consists in translating the source program in single assignment form. This transformation is independent from scheduling but needs results given by dataflow analysis. Generally, total memory expansion is realized before the parallelization.

There is a strong relation between output dependences and anti-dependences. Consider two operations $\langle S, \vec{y}\rangle$, $\langle T, \vec{z}\rangle$, and c a cell memory, such as $c \in \mathcal{R}(S, \vec{y})$ and $c \in \mathcal{M}(T, \vec{z})$. In a correct program, each variable must be set before being read. So, there is necessarily an operation $\langle R, \vec{x}\rangle$ which sets c and which is executed before $\langle S, \vec{y}\rangle$: $\langle R, \vec{x}\rangle \prec \langle S, \vec{y}\rangle \prec \langle T, \vec{z}\rangle$. There is also a output dependence between $\langle R, \vec{x}\rangle$ and $\langle T, \vec{z}\rangle$. From this, one may deduce that if all output dependences are deleted, then anti-dependences and spurious flow dependences also disappear. Total memory expansion consists in assigning one distinct memory cell to each operation. The following algorithm presented in [3] establishes the single assignment form of a static control program:

1. **Renaming :** for each statement R, with $\vec{x}$ as iteration vector, associate a specific data structure `InsR`:

$$R : a[\vec{f}(\vec{x})] = \ldots \rightarrow \texttt{InsR}[\vec{f}(\vec{x})] = \ldots$$

2. **Expanding:** for each instruction R, replace the subscript function $\vec{f}(\vec{x})$ in `InsR` by $\vec{x}$ in left hand-sides:

$$R : \texttt{InsR}[\vec{f}(\vec{x})] = \ldots \rightarrow \texttt{InsR}[\vec{x}] = \ldots$$

3. **Reconstructing the dataflow:** replace all read reference by its new representation as given by the source function. The value produced by $\langle R, \vec{x}\rangle$ is stored in $\texttt{InsR}[\vec{x}]$. So if one finds the following source function for a memory cell c in an operation $\langle S, \vec{y}\rangle$: $source(c, \langle S, \vec{y}\rangle) \equiv \langle S, \vec{x}\rangle$, then c must be replaced by $\texttt{InsR}[\vec{x}]$ in the single assignment program.

Renaming deletes all output dependences which appear between two operations instances of two different instructions. Expanding deletes output dependences which appear between two operations instances of the same instruction. The single assignment form version of our running example is given in fig. (2). It is clear that the memory cost is high. Starting from a scalar s, one gets an array of n elements and another one with n^2 elements.

A first intuitive approach can easily show that deleting all false dependences is not necessary. During an execution of a parallel program in single assignment form, a memory cell $\texttt{InsR}[\vec{x}]$ is empty until the execution of $\langle R, \vec{x}\rangle$ at $\theta(R, \vec{x})$. Moreover in many cases, a value stored in a memory cell can become useless in memory after a limited delay. Consider `InsS2[i,j]` in our running example:

- In the parallel program scheduled by θ, this memory cell is empty until the execution of $\langle S2, i, j\rangle$ at $\theta(S2, i, j) = j$.
- The value produced by $\langle S2, i, j\rangle$ is read by $\langle S2, i, j+1\rangle$ at $\theta(S2, i, j+1) = j+1$. After this time, the value is useless but still resides in memory.

```
        PROGRAM scalar
        INTEGER i, j, n, InsS1[n], InsS2[n,n]
        DO i = 1,n
{S1}      InsS1[i] = 0
          DO j = 1,n
{S2}        InsS2 [i,j] = if (j >= 2) then InsS2 [i,j-1]
                          else InsS1 [i] + 1
          ENDDO
        ENDDO
        END
```

Figure 2. The `scalar` program in single assignment form

3.2.2 Previous Techniques to Reduce Memory Cost

Some methods try to eliminate false dependences with a reduced memory cost. Wolfe in [11] defines the method of array contraction for vector architectures. After scalar expansion and loop interchange, he performs array contraction because the vector instructions only concern the innermost loop of each loop nest. Maydan and Lam in [8], Li and Lee in [9] define a method which optimize array privatization after a renaming phase. Privatization is equivalent to expansion. They don't delete an output dependence between operations instances of a same instruction R, if it is masked by

a dataflow. Darte, Vivien, Calland and Robert in [1] intoduce two graph transformations to eliminate anti and output dependences by renaming. They give an unified framework for such transformation and prove that the problem of determining a minimal process of renaming is NP-complete. Values Lifetime Analysis is a technique which comes from the "systolic" community. It takes into account single assignment form programs and try to generate output and anti-dependences without changing the dataflow([2],[10]).

4 Minimal Memory Expansion With Respect to a Schedule

Our method tries to maintain as many false dependences as possible from the original program to the parallel one. One takes into account the original data structures, the results given by data dependences and data flow analysis, the schedule function. One generates a program with new data structures which is still sequential but can be parallelized according to the scheme (10).

4.1 Neutral Dependences

Consider an operation $\langle R, \vec{x} \rangle$ instance of an assignment statement R. Let $\mathcal{U}(R, \vec{x})$ be the set of operations such that there is a dataflow from $\langle R, \vec{x} \rangle$ to each operation $\langle S, \vec{y} \rangle$ of $\mathcal{U}(R, \vec{x})$:

$$\mathcal{U}(R, \vec{x}) = \{\langle S, \vec{y} \rangle \in E \mid source(c, \langle S, \vec{y} \rangle) \equiv \langle R, \vec{x} \rangle\} \tag{11}$$

Let be $\mathcal{V}(R, \vec{x})$ the value produced by $\langle R, \vec{x} \rangle$, $\mathcal{V}(R, \vec{x})$ must absolutly reside in memory for $\vec{t} \in [\theta(R, \vec{x}), \max_{\mathcal{U}(R,\vec{x})} \theta(S, \vec{y})]$. Before and after these dates this value is useless in memory. Suppose that one has an output dependence at depth p between $\langle R, \vec{x} \rangle$ and an operation $\langle T, \vec{z} \rangle$ (written $R \, \delta_p^\circ \, T$) in the sequential program. If $\theta(T, \vec{z}) \gg \max_{\mathcal{U}(R,\vec{x})} \theta(S, \vec{y})$, it is clear that this output dependence can be maintened in the parallel program, because $\mathcal{V}(R, \vec{x})$ is useless in memory at $\theta(T, \vec{z})$. To improve this idea, we will develop the concept of neutral dependences.

Definition 1. *An output dependence is neutral for a schedule θ, which satifies (7), iff keeping this dependence doesn't change the sequential dataflow in the parallel program obtained from θ by scheme (10).*

An output dependence can be maintained in a parallel program iff it is neutral. In this case, the results of the parallel program are still valid. The following proposition gives specific conditions that an output dependence must verify to be neutral.

Proposition 1. *A output dependence $R \, \delta_p^\circ \, T$ (R and T are two statements) is neutral for θ iff:*

$$\mathcal{M}(R, \vec{x}) = \mathcal{M}(T, \vec{z}) \wedge \langle R, \vec{x} \rangle \prec_p \langle T, \vec{z} \rangle \Rightarrow \theta(R, \vec{x}) \ll \theta(T, \vec{z}) \tag{12}$$

and

$$\theta(T, \vec{z}) \gg \max_{\mathcal{U}(R,\vec{x})} (\theta(S, \vec{y})) \tag{13}$$

(12) ensures that the execution order between $\langle R, \vec{x} \rangle$ and $\langle T, \vec{z} \rangle$ is the same in the sequential and parallel programs. (13) verifies that dataflow between $\langle R, \vec{x} \rangle$ and operations in $\mathcal{U}(R, \vec{x})$ won't be affected by $\langle T, \vec{z} \rangle$. This condition ensures that $\mathcal{V}(R, \vec{x})$ is present in memory when $\vec{t} \in [\theta(R, \vec{x}), \max_{\mathcal{U}(R,\vec{x})} (\theta(S, \vec{y}))]$, even if the output dependence is not removed in the parallel program.

We can extend this definition to anti-dependences and flow dependences which are not dataflows. For these kinds of dependences it is just necessary to verify that execution order of operations in dependance is the same in the sequential and parallel programs.

Definition 2. *An anti-dependence between two instructions S and T is neutral for a schedule function θ which satisfies (7) iff the execution order of these operations is the same in the sequential and parallel programs.*

The definition is the same for a spurious flow dependence.

Proposition 2. *A anti-dependence $S \, \overline{\delta_p} \, T$ is neutral according to θ iff:*

$$\mathcal{R}(S, \vec{y}) \cap \mathcal{M}(T, \vec{z}) \neq \emptyset \wedge \langle S, \vec{y} \rangle \prec_p \langle T, \vec{z} \rangle \Rightarrow \theta(S, \vec{y}) \ll \theta(T, \vec{z}) \tag{14}$$

(14) ensures that if this dependence is not deleted, it will still be verified in the parallel program.

Proposition 3. *A spurious flow dependence $R \, \delta_p \, T$ is neutral for θ iff:*

$$\mathcal{M}(R, \vec{x}) \cap \mathcal{R}(S, \vec{y}) \neq \emptyset \wedge \langle R, \vec{x} \rangle \prec_p \langle S, \vec{y} \rangle \Rightarrow \theta(R, \vec{x}) \ll \theta(S, \vec{y}) \tag{15}$$

4.2 Tests of Neutrality

4.2.1 Neutral Output Dependences

Let's consider:

$$\begin{array}{ll} R: & a[\vec{f}(\vec{x})] = \ldots \\ T: & a[\vec{g}(\vec{z})] = \ldots \end{array}$$

Consider the output dependences between operations instances of R and T at depth p. A dependence $R \, \delta_p^\circ \, T$, is characterized by the following conditions:

- $\langle R, \vec{x} \rangle$ and $\langle T, \vec{z} \rangle$ must exist: $\vec{x} \in \mathcal{D}(R)$, $\vec{z} \in \mathcal{D}(T)$;
- Access conflict: $\vec{f}(\vec{x}) = \vec{g}(\vec{z})$;
- Sequencing Predicate at depth p: $\langle R, \vec{x} \rangle \prec_p \langle T, \vec{z} \rangle$

Therefore, there is a dependence iff, system $Q^p_{RT}(\vec{x},\vec{z})$,

$$Q^p_{RT}(\vec{x},\vec{z}) = \{ \quad \begin{array}{l} \vec{x} \in \mathcal{D}(R) \wedge \\ \vec{z} \in \mathcal{D}(T) \wedge \\ \vec{f}(\vec{x}) = \vec{g}(\vec{z}) \wedge \\ \langle R,\vec{x}\rangle \prec_p \langle T,\vec{z}\rangle \} \end{array} \tag{16}$$

has a solution. To verify (12), one must have a dependence in the sequential program, which must still be verified in the parallel program. Therefore, in the parallel program, we must have: $\theta(R,\vec{x}) \ll \theta(T,\vec{z})$. If this execution order is not respected for only one of the operations instances of R and T linked by this dependence, the condition (12) is not verified. So we simply consider that (12) is verified if for no operation of R and T in dependence, one has $\theta(T,\vec{z}) \underline{\ll} \theta(R,\vec{x})$ that is to say if the system $N^p_{RT}(\vec{x},\vec{z})$,

$$N^p_{RT}(\vec{x},\vec{z}) = \{ \quad \begin{array}{l} \vec{x} \in \mathcal{D}(R) \wedge \\ \vec{z} \in \mathcal{D}(T) \wedge \\ \vec{f}(\vec{x}) = \vec{g}(\vec{z}) \wedge \\ \langle R,\vec{x}\rangle \prec_p \langle T,\vec{z}\rangle \wedge \\ \theta(T,\vec{z}) \underline{\ll} \theta(R,\vec{x}) \} \end{array} \tag{17}$$

has no solution. $Q^p_{RT}(\vec{x},\vec{z})$ is a $\mathbb{Z}$-polyhedron. $\theta(R,\vec{x})$ and $\theta(T,\vec{z})$ are vectors of affine functions in the loop counters. Hence $N^p_{RT}(\vec{x},\vec{z})$ is a disjunction of $\mathbb{Z}$-polyhedra which must all be empty. So verifying the emptiness of $N^p_{R,T}(\vec{x},\vec{z})$ can be easily done by the PIP (Parametric Integer Programing) tool (see [4] for more explanations). Remember that in our example, we have chosen the schedule function $\theta(R,i) = 0$ and $\theta(S,i,j) = j$. Let's verify (12) for program `scalar`. For the $R\,\delta^\circ_0\,R$ dependence, one has **if** $1 \leq i \leq n$ **then** $N^0_{RR}(i) \neq \emptyset \Rightarrow$ this dependence is not neutral. For others dependences, one can find that (12) is verified for $R\,\delta^\circ_0\,S$, $R\,\delta^\circ_1\,S$ and $S\,\delta^\circ_1\,S$ dependences and not verified by $S\,\delta^\circ_0\,R$ and $S\,\delta^\circ_0\,S$ dependences (hence these dependences are not neutral).

Theorem 1. *The condition (13) is verified for a given output dependence iff all anti-dependences generated by this dependence, are neutral.*

Proof: consider the operations of $\mathcal{U}(R,\vec{x})$. If there is an output dependence between $\langle R,\vec{x}\rangle$ and an operation $\langle T,\vec{z}\rangle$ at depth p, there is also an anti-dependence between any operation $\langle S,\vec{y}\rangle \in \mathcal{U}(R,\vec{x})$ and $\langle T,\vec{z}\rangle$ at depth p':

$$\prec: \quad \begin{array}{ll} \langle R,\vec{x}\rangle : & c = \ldots \\ \langle S,\vec{y}\rangle : & \ldots = \ldots c \ldots \\ \langle T,\vec{z}\rangle : & c = \ldots \end{array}$$

If every dependence $S\,\overline{\delta_{p'}}\,T$ is neutral, it ensures that $\theta(S,\vec{y}) \ll \theta(T,\vec{z})$ (according to (14)) . Therefore $\theta(T,\vec{z}) \gg \theta(S,\vec{y}), \forall \langle S,\vec{y}\rangle \in \mathcal{U}(R,\vec{x})$, hence $\theta(T,\vec{z}) \gg \max_{\mathcal{U}(R,\vec{x})} \theta(S,\vec{y})$. So (13) is verified.

4.2.2 Neutral Anti-dependences

Consider:

$$\begin{array}{ll} S: & \ldots = \ldots a[\vec{h}(\vec{y})] \ldots \\ T: & a[\vec{g}(\vec{z})] = \ldots \end{array}$$

One must determine if the $S\,\overline{\delta_p}\,T$ dependence is neutral, that is to say verify (14). To determine if (14) is respected, one has to verify that the execution order between $\langle S,\vec{y}\rangle$ and $\langle T,\vec{z}\rangle$ stays the same in the parallel program for the operations instances of S and T which are linked by this dependence. Also the dependence $S\,\overline{\delta_p}\,T$ is neutral iff the system $N^p_{ST}(\vec{y},\vec{z})$

$$N^p_{ST}(\vec{y},\vec{z}) = \{ \quad \begin{array}{l} \vec{y} \in \mathcal{D}(S) \wedge \\ \vec{z} \in \mathcal{D}(T) \wedge \\ \vec{h}(\vec{y}) = \vec{g}(\vec{z}) \wedge \\ \langle S,\vec{y}\rangle \prec_p \langle T,\vec{z}\rangle \wedge \\ \theta(T,\vec{z}) \underline{\ll} \theta(S,\vec{y}) \} \end{array} \tag{18}$$

has no solution.

When one knows that an anti-dependence is not neutral, one knows that for the associated output dependence the condition (13) is invalidated and the dependence is not neutral. Suppose, one has the following situation: $c = \mathcal{M}(R,\vec{x}) = \mathcal{M}(T,\vec{z})$ and $\langle R,\vec{x}\rangle \equiv source(c,\langle S,\vec{y}\rangle)$. If the $S\,\overline{\delta_p}\,T$ dependence is not neutral, then the operation $\langle T,\vec{z}\rangle$ kills the value produced by $\langle R,\vec{x}\rangle$ and stored in c before it is read by $\langle S,\vec{y}\rangle$ in the parallel program. This situation would have occurred if the output dependence between $\langle R,\vec{x}\rangle$ and $\langle T,\vec{z}\rangle$ was not deleted. So the output dependence between $\langle R,\vec{x}\rangle$ and $\langle T,\vec{z}\rangle$ is not neutral. We know the depths p and p' of $S\,\overline{\delta_p}\,T$ and $R\,\delta_{p'}\,S$ dependences. We must determine the depth p'' of $R\,\delta^\circ_{p''}\,T$ dependence. With the $S\,\overline{\delta_p}\,T$ dependence, we have: $\langle S,\vec{y}\rangle \prec_p \langle T,\vec{z}\rangle \Leftrightarrow (\vec{y}[1..p] = \vec{z}[1..p]) \wedge (\vec{y}[p+1] < \vec{z}[p+1])$. With the $R\,\delta_{p'}\,S$ dependence, we have: $\langle R,\vec{x}\rangle \prec_{p'} \langle S,\vec{y}\rangle \Leftrightarrow (\vec{x}[1..p'] = \vec{y}[1..p']) \wedge (\vec{x}[p'+1] < \vec{y}[p'+1])$. We must consider, three cases:

1. $p = p' : (\vec{x}[1..p] = \vec{z}[1..p]) \wedge (\vec{x}[p+1] < \vec{z}[p+1]) \Rightarrow \langle R,\vec{x}\rangle \prec_p \langle T,\vec{z}\rangle \Rightarrow p'' = p$
2. $p < p' : (\vec{x}[1..p] = \vec{z}[1..p]) \wedge (\vec{x}[p+1] < \vec{z}[p+1]) \wedge (\vec{x}[p+1] < \vec{z}[p+1]) \Rightarrow \langle R,\vec{x}\rangle \prec_p \langle T,\vec{z}\rangle \Rightarrow p'' = p$
3. $p > p' : (\vec{x}[1..p'] = \vec{z}[1..p']) \wedge (\vec{x}[p'+1] < \vec{z}[p'+1]) \wedge (\vec{x}[p'+1] < \vec{z}[p'+1]) \Rightarrow \langle R,\vec{x}\rangle \prec_{p'} \langle T,\vec{z}\rangle \Rightarrow p'' = p'$

So, if the $S\,\overline{\delta_p}\,T$ dependence is not neutral, then the $R\,\delta^\circ_{min(p,p')}\,T$ dependence is not neutral either. In our running example, consider the $S\,\overline{\delta_0}\,S$ dependence, we have:

$$source(s,\langle S2,i,j\rangle) = \left\{ \begin{array}{l} \textbf{If } j \geq 2 \\ \textbf{Then } \langle S2,i,j-1\rangle \\ \textbf{Else } \langle S1,i\rangle \end{array} \right.$$

The first leaf of the source function concerns a instance of $S2$, so one must determinate if the $S2\,\overline{\delta_0}\,S2$ dependence is neutral. One finds that $N^0_{S2S2}(i,j) \neq \emptyset \Rightarrow$ so this dependence is not neutral, and $S2\,\delta^\circ_0\,S2$ dependence is not neutral either. The second leaf of the source function concerns an instance of $S1$, hence the dependence $S1\,\delta^\circ_0\,S2$ is not neutral. For others anti-dependences, one finds that dependence $S2\,\overline{\delta_1}\,S2$ is neutral and that $S2\,\overline{\delta_0}\,S1$ is not neutral. As a consequence the dependences $S2\,\delta^\circ_0\,S2$ and $S1\,\delta^\circ_0\,S1$ are not neutral.

Finally, we have for the output dependences in our running example: $S1\,\delta^\circ_0\,S1$, $S2\,\delta^\circ_0\,S2$ $S2\,\delta^\circ_0\,S1$ and $S1\,\delta^\circ_0\,S2$ which are not neutral; $S1\,\delta^\circ_1\,S2$ and $S2\,\delta^\circ_1\,S2$ which are neutral.

4.2.3 Neutral Spurious Flow Dependences

Theorem 2. *It is useless to verify if a flow dependence, which is not a dataflow, is neutral.*

Proof : consider the following operations:

$$\begin{array}{rll} \prec: & \langle R,\vec{x}\rangle: & c = \ldots \\ & \langle T,\vec{z}\rangle: & c = \ldots \\ & \langle S,\vec{y}\rangle: & \ldots = \ldots c \ldots \end{array}$$

Suppose that $\langle S,\vec{y}\rangle \in \mathcal{U}(T,\vec{z})$. Dependence $R\,\delta_p\,S$ is not a dataflow, because the value stored in c by $\langle R,\vec{x}\rangle$ is killed by $\langle T,\vec{z}\rangle$ before the reading of c by $\langle S,\vec{y}\rangle$. In the parallel program, one has $\theta(T,\vec{z}) \ll \theta(S,\vec{y})$ according to (7). We must consider two cases:

1. If the output dependence between R and T is not neutral, then it must be removed in the parallel program and the flow dependence has disappeared.
2. If this output dependence is neutral, one has also $\theta(R,\vec{x}) \ll \theta(T,\vec{z}) \Rightarrow \theta(R,\vec{x}) \ll \theta(S,\vec{y})$ hence (15) is verified and it means that the dependence $R\,\delta_p\,S$ is neutral.

4.3 Exploitation of Results

The examination of neutrality of output dependences will help us to decide if we must add a dimension or new elements in a specific dimension (*minimal expanding*) or if we must proceed or not in renaming a data structure used by two different instructions (*minimal renaming*). We have developped the following algorithm which gives an optimized storage for data of a parallel static control program:

1. **Minimal expansion for each statement** R: if a is the data structure in the left hand side of R, one must find the minimal shape that a can have in R. The goal is to eliminate all output dependences $R\,\delta^\circ\,R$ which are not neutral. If an output dependence at depth p between operations instances of R is not neutral, one must expand a according to $\vec{x}[p+1]$:
   - one adds one dimension to a. The size of this dimension is the number of iterations of the loop $p+1$ which surrounds R;
   - This new dimension must be indexed by the counter of this loop in left hand side of R.

   $$R\,:\,a[\vec{f}(\vec{x})] = \ldots \rightarrow a[\vec{f}(\vec{x}),\vec{x}[p+1]] = \ldots$$

   In our running example, for $S1$, the dependence $S1\,\delta^\circ_0\,S1$ is not neutral hence

   $$S1\,:\,(s = \ldots \rightarrow s[i] = \ldots)$$

   The scalar s is now an array of n elements because there are n iterations in the loop i. In $S2$, the dependence $S2\,\delta^\circ_0\,S2$ is not neutral so it must be deleted, the dependence $S2\,\delta^\circ_1\,S2$ is neutral, so it can be maintained:

   $$S2\,:\,(s = \ldots \rightarrow s[i] = \ldots)$$

   With these new subscript functions, we are sure that every output dependences which only concern operations instances of a single statement R and which are not neutral, are deleted.
2. **Correcting the dependence graph:** the minimal expansion can suppress some output dependences which appear between operations instances of different instructions. Consider our previous statements R and T ($R \neq T$). Suppose that in the next steps ot this algorithm, one doesn't proceed in renaming the array a shared by the statements. After minimal expansion, one gets two data structures which can be different. If there is no renaming, the data struture shared by R and T must be in fact the rectangular hull of the union of the two data structures defined by minimal expansion of R and T. Imagine that there is an output dependence $R\,\delta^\circ T$ at depth p in the original program with $p \in N_{RT}$. If, for instance, one had expanded a in R according to $\vec{x}[p+1]$, it adds the following constraint in $Q^p_{RT}(\vec{x},\vec{z})$ which is $\vec{x}[p+1] = \vec{z}[p+1]$. One knows that $\langle R,\vec{x}\rangle \prec_p \langle T,\vec{z}\rangle \Rightarrow \vec{x}[p+1] < \vec{z}[p+1]$. Hence, now $Q^p_{RT}(\vec{x},\vec{z})$ has no solution and the output dependence has disappeared. In our running example, minimal expansion deletes the dependences $S2\,\delta^\circ_0\,S1$ and $S1\,\delta^\circ_0\,S2$.
3. **Minimal renaming**: we must take into account all residual output dependences between R and T, $\forall p \in N_{RT}$. If only one of these dependences is not neutral, we must rename a in T, because all these kind of dependences must be deleted. If all dependences are neutral, the data structure may remain the same in the two statements. Finding the minimal number of data structures to rename is a NP-complete problem, as it shown in [1]. We suggest the following heuristics: one builds a graph for each data structure a which appears at least once in a left hand side of a statement in the original program. Each vertex represents a statement where a is the left hand side. There is an edge from a vertex R to another one T iff there is a residual $R\,\delta^\circ_p\,T$ dependence which is not neutral ($\forall p \in N_{RT}$). Then one can apply on this graph a greedy coloring algorithm. Finally it is clear that vertices that have the same colour

can share the same data structure. In our example, the residual output dependence between R and S is $R\,\delta_1^\circ\,S$ which is neutral. So it is unnecessary to rename s in S. The final shape of each data structure shared by many statements must be the rectangular hull of the union of all shapes built form minimal expansion. The program is reconstructed with the new data structures and their subscripts functions.

Finally, one gets the program of figure (3). The removal of the conditional expression is due to the fact, that s has not been renamed.

```
         PROGRAM scalar
         INTEGER i, j, n, s[n]
         DO i = 1,n
{S1}       s[i] = 0
           DO j = 1,n
{S2}         s[i] = s[i] + 1
           ENDDO
         ENDDO
         END
```

Figure 3. The `scalar` program in single assignment form

The array (4) gives an overview on the shape of different data structures generated for the scalar program by the different techniques referenced in this article: in the source program (1), in the single assignment program (2), in the program generated by the Chamsky's method (3), by Dror and Lam's method (4) and by our technique (5).

(1)	(2)	(3)	(4)	(5)
s	InsS1[n]	InsS1[n]	InsS1[n]	s[n]
	InsS2[n,n]	InsS2[n,2]	InsS2[n]	

Figure 4. Data structures generated by different methods

The program has now the appropriate data structures and can be parallelized with the model given by (10).

5 Conclusion

Notice that if one builds a schedule function equivalent to the sequential execution order, one finds that all dependences are neutral, so there is no expanding and no renaming and we keep the scalar s. We have then obtained a very satisfying result: inherently sequential programs are fixed points for our parallelization method. Our method effectively reduces the memory cost in the data expansion process for static control programs. Our performances are strongly linked to the parallelism degree (size of operations fronts) given by the schedule. Hence one can go further and improves our results by adjusting the scheduling to the architecture. Consider for instance, that the target architecture is a pipeline processor Cray. In this case, the real size of a front is limited to 64 which is the size of a vector register. One can easily adjust the schedule function such as no front has more than 64 operations. In the case of our running example, the memory requirement is reduced to an array of 64 elements. The interest of our method is that it can have result on one hand on the expansion and on the other hand on renaming. All previous methods focused on only one of these two topics. The technique has been implemented in Lisp within the PAF project. This methods takes the place of single assignment form translation.

To conclude one gives our results obtained with the `cholesky` program:

- Original version:

```
         program choles
         integer i, j, k
         real x
         real a(10,10), p(10)
         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
```

- Single assignment form version:

```
       PROGRAM choles
           real a(10,10)
           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)
                            - ins6(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
```

- Version with minimal data expansion:

```
      PROGRAM choles
          integer i,j,k,n
          real x(n)
          real a(10,10)
          real p(10)
          real sqrt
          real insS4(n,n-1)
          DO i = 1,n,1
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                insS4(i,j) = a(i,j)
                  DO k = 1,i-1,1
S5                    insS4(i,j) = insS4(i,j) -
                                   a(k,j) * a(k,i)
                  END DO
S6                a(j,i)= insS4(i,j) * p(i)
              END DO
          END DO
      END
```

References

[1] 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 - Feb 1996.

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

[3] P. Feautrier. *Array expansion*. ACM Int. Conf on Supercomputing, pages 429-441, 1988.

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

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

[6] P. Feautrier. *Some efficients solutions to the affine scheduling problem, I, one dimensionnal time*. Int J. of Parallel Programming, 21(5):313-348, October 1992.

[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] D. E. Maydan, S. P. Amarasinghe, M. S. Lam. *Array Data-Flow Analysis and its Use in Array Privatization*. In Proc. of ACM Conf. on Principles of Programming Languages, pages 2-15, January 1993.

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

[10] 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.

[11] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman 1989.