Optimizing Storage Size for Static Control Programs in Automatic Parallelizers

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. 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 for global memory architectures which leaves untouched the performances of the parallelization process.

1 Introduction

This article deals with the automatic parallelization method 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 statements to array with affine subscripts. During the parallelization process, the removal of memory related dependences is usually realized by translating the program into a single assignment form. This total data expansion has a very high memory cost. The aim of this paper is to present a new technique to produce multiple assignment parallel code which leaves untouched the performances of the parallelization process. Synchronous computers with global memory are our target architectures. In section 2, we describe the polytope method with total data expansion. In section 3, we present our technique of partial data expansion.

2 The polyedric method

2.1 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 are affine functions in the loop counters and integral structure parameters. An operation may be named $\langle R, \mathbf{x}\rangle$ where R is a statement and $\mathbf{x}$ the iteration vector built from the values of the surrounding loop counters. The iteration domain $\mathcal{D}(R)$ of a statement R, is the set of instances of R.

2.2 Sequential Execution Order

The lexicographic order is noted $\ll$. The expression $R \triangleleft S$ 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 $\mathbf{x} \ll_p \mathbf{y} \equiv \mathbf{x}[1..p] = \mathbf{y}[1..p] \wedge \mathbf{x}[p+1] < \mathbf{y}[p+1]$ and $\ll$ is given by $\mathbf{x} \ll \mathbf{y} \equiv \bigvee_{p=0}^{|\mathbf{x}|-1} \mathbf{x} \ll_p \mathbf{y}$. The fact that operation $\langle R, \mathbf{x}\rangle$ is executed before the operation $\langle S, \mathbf{y}\rangle$ is written: $\langle R, \mathbf{x}\rangle \prec \langle S, \mathbf{y}\rangle$.
It is shown in [4] that $\langle R, \mathbf{x}\rangle \prec \langle S, \mathbf{y}\rangle \equiv \mathbf{x} \ll \mathbf{y} \vee (\mathbf{x}[1..N_{RS}] = \mathbf{y}[1..N_{RS}] \wedge R \triangleleft S) \equiv \bigvee_{p=0}^{N_{RS}} \langle R, \mathbf{x}\rangle \prec_p \langle S, \mathbf{y}\rangle$

where $\langle R, \mathbf{x}\rangle \prec_p \langle S, \mathbf{y}\rangle \Leftrightarrow \begin{cases} 0 \le p < N_{RS} : \mathbf{x} \ll_p \mathbf{y} \\ p = N_{RS} : \mathbf{x}[1..N_{RS}] = \mathbf{y}[1..N_{RS}] \wedge R \triangleleft S \end{cases}$

2.3 Array Data Flow Analysis

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 $v \prec_p u$. If $\mathcal{M}(v) \cap \mathcal{R}(u) \neq \emptyset$, there is a **flow dependence** at depth p, written $v\,\delta_p\,u$. If $\mathcal{R}(v) \cap \mathcal{M}(u) \neq \emptyset$, there is an **anti-dependence**, written $v\,\bar{\delta}_p\,u$. If $\mathcal{M}(v) \cap \mathcal{M}(u) \neq \emptyset$, there is an **output dependence**, written $v\,\delta_p^{o}\,u$.

The real dependences which define the inherent semantic of a program, are a subset of flow dependences: the **direct flow dependences**. All others dependences are due to memory reuse and are artificial. **Direct flow dependences** are computed by data flow analysis [4]. 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* function of the read. 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 $\perp$. $\perp$ indicates that the array cell under study is not modified.
Sources functions are gathered in the Data Flow Graph (DFG). The `matrix-vector` program is taken as running example in this article. It is given with its DFG.

```
    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
```

The DFG of the matrix-vector program
$source(s, \langle S2, i, j\rangle) = \begin{cases} \textbf{if } j-2 \ge 0 \\ \textbf{then } \langle S2, i, j-1\rangle \\ \textbf{else } \langle S1, i\rangle \end{cases}$
$source(a(i,j), \langle S2, i, j\rangle) = \perp$
$source(b(j), \langle S2, i, j\rangle) = \perp$
$source(s, \langle S3, i\rangle) = \langle S2, i, n\rangle$

2.4 Parallelization by Scheduling

All memory related dependances can be deleted by a total memory expansion which 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 a memory cell is associated to an operation. The algorithm of translation of a static control program into a single assignment form is described in [4].

Finally the program is parallelized by scheduling. A time function θ is computed 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: $\forall c \in \mathcal{R}(u), \theta(source(c,u)) \ll \theta(u)$.

It defines a set of linear constraints. One limits oneself to affine one-dimensionnal and multi-dimensionnal schedules [5]. In the case of our running example, one can have the following schedule function θ:

$$\theta(S1,i) = 0 \quad \theta(S2,i,j) = j \quad \theta(S3,i) = n+1 \qquad (1)$$

An operation front $\mathcal{F}(\mathrm{t})$ gathers all operations which have a same execution time. The operations of a same front can be executed in parallel. If one translates in Fortran 90 the parallel program built with (1) as new operations execution order, 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
```

Translating the sequential program into a 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 space of $\mathcal{O}(n^2)$.

3 Reduced Data Expansion in Parallelized Programs

3.1 Related Work

Most of papers from the automatic parallelization community deal with array privatization. Privatization is a technique that allows each thread on a processor to allocate a distinct instance of a variable. It may require less space than total 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 [7]. Lam proposes to optimize array privatization with the help of the DFG [1].

De Greef and Catthoor have adressed the memory reuse problem for static control programs. They stop at the formulation of the constraints to be satisfied [2].

Another solution has been proposed by the systolic community [8]. Programs in this case are directly 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.

3.2 Utility Span of a Value

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.

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 usually called the **utilization set** of v. 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 operation which executes this last read.
Consider a memory cell $\mathcal{C}(v)$ during the execution of a parallel program in single assignment form. One can distinguish three periods:

1. **Period (I)**: the memory cell **stays empty** until the execution of v with which it is associated.
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**.
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**.

It is clear that during the periods (I) and (III), $\mathcal{C}(v)$ can store others values. Our method of partial data expansion is based on the notion of **utility span of a value.** It is clear that it corresponds to the period (II): $\mathcal{V}(v)$ must reside in memory during $\mathbf{t} \in [\theta(v), \mathcal{L}(v)]$.
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. Such output dependences are called **neutral dependences**.

3.3 Neutral Dependences

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 give the characteristics of a **neutral output dependence** between two operations v and w in the parallel program: **v must be executed before w** **($\theta(v) \ll \theta(w)$), there is an access conflict** ($\mathcal{C}(v) = \mathcal{C}(w)$) **and the utility spans are separate** ($\mathcal{L}(v) \ll \theta(w)$).

By extension an output dependence between v and w can be considered as neutral if w is $L(v)$, i.e. the operation which executes the last read of $\mathcal{V}(v)$. The write of $\mathcal{V}(w)$ occurs after the read of $\mathcal{V}(v)$ by w.

To decide if an output dependence is neutral in a parallel program, one must have a precise estimation of an utility span of each value $\mathcal{V}(v)$. Then 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 to its parallel version [6]. But this method is directly dependent from the original data space and can't be used to reduce data size of programs provided 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.

3.4 Determinating Utility Span

Consider the utility span of an operation $\langle R, \mathbf{x}\rangle$: $[\theta(R,\mathbf{x}), \mathcal{L}(R,\mathbf{x})]$. The lower bound of this time subsegment is directly given by θ. The problem is to compute the upper bound $\mathcal{L}(R,\mathbf{x})$. 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:

$$\begin{aligned} R &: a[\mathbf{f}(\mathbf{x})] = \ldots \\ S &: \ldots = \ldots a[\mathbf{h}(\mathbf{y})] \ldots \end{aligned}$$

The operation $L_S(R,\mathbf{x})$ is the last read of $\mathcal{V}(R,\mathbf{x})$ in the parallel program among the operations instances of S which belong to $\mathcal{U}(R,\mathbf{x})$. Let $\langle S, B_{RS}(\mathbf{x})\rangle$ be this set of candidates which is built by scanning the DFG. It is clear that the last operation which reads $\mathcal{V}(R,\mathbf{x})$ between instances of S is the last one executed according to θ: $L_S(R,\mathbf{x}) = \langle S, \max_{\ll_\theta} B_{RS}(\mathbf{x})\rangle$.

All statements which may read the data a must be taken into account. The real last read is their maximum according to θ: $L(R,\mathbf{x}) = \max_{\ll_\theta} L_S(R,\mathbf{x})$. Like the source function, $L(R,\mathbf{x})$ is a quast.

To determine $\mathcal{L}(R,\mathbf{x})$ one just applies the function θ to each leaf of $L(R,\mathbf{x})$ except for leaves which are the symbol $\perp$ which are left untouched. The symbol $\perp$ indicates that $\mathcal{V}(v)$ is either useless or an output value. The utily spans in our running example are:

Operation v	$L(v)$	$\mathcal{L}(v)$	Utility span of $\mathcal{V}(v)$
$\langle S1, i\rangle$	$\langle S2, i, 1\rangle$	1	$[0, 1]$
$\langle S2, i, j\rangle$	$\begin{cases} \textbf{if } j \le n-1 \\ \textbf{then } \langle S2, i, j+1\rangle \\ \textbf{else } \langle S3, i\rangle \end{cases}$	$\begin{cases} \textbf{if } j \le n-1 \\ \textbf{then } j+1 \\ \textbf{else } n+1 \end{cases}$	$\begin{cases} \textbf{if } j \le n-1 \\ \textbf{then } [j, j+1] \\ \textbf{else } [j, n+1] \end{cases}$
$\langle S3, i\rangle$	$\perp$	$\perp$	$[n+1, \perp]$

3.5 Partial Data Expansion

The first step is a **partial array and scalar expansion process** that decides the shape 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.

Partial Array Expansion 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 constitute the data in the left hand side of R in the restructured program. The aim is that lhsR provides memory reuse, i.e. **neutral** output dependences between some operations instances of R. Moreover the elaboration of lhsR must be independent from the original data structure in the lhs of R.

One recalls that a neutral output dependence can't kill a value $\mathcal{V}(R, \mathbf{x})$ 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 $\mathcal{V}(R, \mathbf{x})$ can have in the parallel program. For an operation $\langle R, \mathbf{x}\rangle$ this duration is obtained by subtracting the lower bound of its utility span from the upper bound. One writes $d(R, \mathbf{x})$ this parameter. One considers that $\perp - \theta(R, \mathbf{x}) = \perp$. Each leaf of $d(R, \mathbf{x})$ is a multi-dimensionnal linear expression in terms 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 $d(R, \mathbf{x})$ on the iteration domain of R: $\forall \mathbf{x} \in \mathcal{D}(R), d(R, \mathbf{x}) \leqslant\!\!\leqslant D(R)$. $D(R)$ is a multidimensionnal linear expression in terms of structure parameters or the symbol $\perp$. Notice that one considers that if $d(R, \mathbf{x}) \neq \perp$, then $\perp \ll d(R, \mathbf{x})$.

$\mathcal{V}(R, \mathbf{x})$ must be in $\mathcal{C}(R, \mathbf{x})$ between $\theta(R, \mathbf{x})$ and $\mathcal{L}(R, \mathbf{x}) = \theta(R, \mathbf{x}) + d(R, \mathbf{x})$. If one wants to protect each instance of R during its utility span, one must build lhsR in such a way that no value $\mathcal{V}(R, \mathbf{x})$ can be killed between $\theta(R, \mathbf{x})$ and $\theta(R, \mathbf{x}) + D(R)$.
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.
- A **partial expansion of R according to** p consists in 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) and indexing this new dimension of lhsR.

The problem is now to compute E_R^p. The partial expansion of R at depth p avoids non neutral output dependences between two operations $\langle R, \mathbf{x}\rangle$ and $\langle R, \mathbf{x}'\rangle$ if $\mathbf{x} \ll_p \mathbf{x}'$. For an operation $\langle R, \mathbf{x}\rangle$, we build the set of candidates gathering all the operations $\langle R, \mathbf{x}'\rangle$ which can't share the same memory cell than $\langle R, \mathbf{x}\rangle$ because their utility spans are **not separate**. Let $C_{RR}^p(\mathbf{x})$ be the set of

candidates and let $e_R^{C,p}$ be its lexicographic maximum. One can't have output dependences between operations $\langle R, \mathbf{x}\rangle$ and $\langle R, \mathbf{x}'\rangle$ with $\langle R, \mathbf{x}\rangle \prec_p \langle R, \mathbf{x}'\rangle \preceq_p \langle R, \mathbf{x}_e\rangle = e_R^{C,p}$. From this follows the inequalities on the iteration vectors : $\mathbf{x}[p+1] < \mathbf{x}'[p+1] \leq \mathbf{x}_e[p+1]$.
If lhsR is expanded at depth p with $E^p_{\langle R,\mathbf{x}\rangle} = \mathbf{x}_e[p+1] - \mathbf{x}[p+1] + 1$, we are sure that no non neutral output dependence at depth p can appear concerning $\langle R, \mathbf{x}\rangle$. But it must be verified for each instance of R, hence the expension degree E^p_R is the maximum value that $E^p_{\langle R,\mathbf{x}\rangle}$ can have for $\mathbf{x} \in \mathcal{D}(R)$: $E^p_R = \max_{\mathbf{x}\in\mathcal{D}(R)} E^p_{\langle R,\mathbf{x}\rangle}$. For our running example, one obtains the following results:

Statements	Maximum utility span duration	Expansion degrees	Final data structure	Final lhs
S1	$D(S1) = 1$	$E^0_{S1} = n$	lhsS1[n]	lhsS1[i] = ...
S2	$D(S2) = 1$	$E^0_{S2} = n$	lhsS2[n]	lhsS2[i] = ...
		$E^1_{S2} = 0$		
S3	$D(S3) = \perp$	$E^0_{S3} = n$	lhsS3[n]	lhsS3[i] = ...

Notice that the array in the lhs of $S3$ is left untouched even if its values are never read, because it stores output values.

Partial Renaming For 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 lhs of the two statements. Let $\mathbf{F}_{R-T}$ be the index function of lhsR-T. One must verify for each operation $\langle R, \mathbf{x}\rangle$ and $\langle T, \mathbf{z}\rangle$ that would be in output dependence (i.e. $\mathbf{F}_{R-T}(\mathbf{x}) = \mathbf{F}_{R-T}(\mathbf{z})$) that $\mathcal{V}(R, \mathbf{x})$ can't be killed by $\langle T, \mathbf{z}\rangle$ before the end of its utility span and that $\mathcal{V}(T, \mathbf{z})$ can't be killed by $\langle R, \mathbf{x}\rangle$ before the end of its utility span.

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 generation 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 lhs. 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 our running example, $S1$, $S2$ and $S3$ have the same colour. The memory requirement is finally an one-dimensionnal array with n elements which can be the array c.

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

After partial expansion the Fortran 90 version of the program is :

```
program matrix-vector
real s(n), 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)
S2        c(1:n:1) = c(1:n:1) + a(1:n:1,t)*b(t)
        end if
      end do
      end
```

4 Conclusion

Our aim has been reached, our method can effectively reduce the memory cost in the data expansion process of static control programs. In our running example the data size is less in the parallel program than the original data size. Moreover the statement $S3$ has become useless after the fusion of c and s and has been removed. Notice that if one builds a schedule function equivalent to the sequential execution order, one finds as final structure the scalar s and the array c. It means that if the source program is provided in single assignment form for instance, then our method reduces the two arrays in the lhs of $S1$ and $S2$ to a single scalar. We have then obtained an important result: 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.

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 - Feb 1996.
3. E. De Greef, F. Catthoor and H. De Man. *Reducing storage size for static control programs mapped to parallel architectures.* - presented at Dagstuhl Seminar on Loop Parallelization, April 1996.
4. P. Feautrier. *Dataflow Analysis of Array and Scalar References.* Int. J. of Parallel Programming, 20(1):23-53, February 1991.
5. 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.
6. V. Lefebvre and P. Feautrier. *Storage Management in Parallel Programs.* In Proc. of the Fith Euromicro Workshop on Parallel and Distributed Processing Conf, Pages 181-188. London. Jan 1997.
7. 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.
8. 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, Aug 1996.