PLUGGING ANTI AND OUTPUT DEPENDENCE REMOVAL TECHNIQUES INTO LOOP PARALLELIZATION ALGORITHMS*

PIERRE-YVES CALLAND , ALAIN DARTE , YVES ROBERT , AND FRÉDÉRIC VIVIEN†

Abstract. In this paper we shortly survey some loop transformation techniques which break anti or output dependences, or artificial cycles involving such "false" dependences. These false dependences are removed through the introduction of temporary buffer arrays.

Next we show how to plug these techniques into loop parallelization algorithms (such as Allen and Kennedy's algorithm). The goal is to extract as many parallel loops as the intrinsic degree of parallelism of the nest authorizes, while avoiding a full memory expansion. We try to reduce the number of temporary arrays that we introduce, as well as their dimension.

Key words. Anti-dependence, output dependence, dependence removal, loop parallelization algorithm.

1. Introduction. Flow (or value-based) dependences are the only "true" dependences of a program. Anti dependences[1] and output dependences[2] are due to storage re-use and can be eliminated at the price of more memory usage. Removing anti and output dependences may prove very useful to break data dependence cycles and thereby enabling vectorization and/or improving parallelization.

However, removing *all* memory-based or "false" (i.e. anti and output) dependences may have a prohibitive cost. A complete removal of false dependences is usually achieved, if feasible, via conversion of the original loop nest program into single assignment form. This turns out to be unnecessarily costly. Indeed, there are some memory-based dependences whose removal will not improve the parallelization. Rather, we should introduce as much memory overhead as needed to expose all the parallelism of the original program. As much as, but no more than, needed.

The aim of this paper is to show how to plug false dependence removal techniques into loop parallelization algorithms. The idea is to characterize those false dependences that do decrease the amount of parallelism, and to remove only these dependences. This will lead to memory savings without sacrificing performance.

The paper is organized as follows. Section 2 is devoted to a brief survey of techniques aimed at removing anti and output dependences. In Section 3 we work out an example to illustrate the key-ideas of our "integration" sketch. We summarize the general steps of our method in Section 4. Finally, we give some conclusions in Section 5.

2. False dependence removal techniques. Many papers have been devoted to the problem of eliminating anti and output dependences. Proposed methods include "array data flow analysis" [10, 13], "variable expansion" [4], "variable renaming" [14] and "node splitting" [14, 6]. See the survey papers of Banerjee, Eigenmann, Nicolau and Padua [3] and Bacon, Graham and Sharp [2], as well as the books of Wolfe [16] and Zima [17], for further references. Note that "array privatization" [11] is yet

*Supported by the CNRS–ENS Lyon–INRIA Project *ReMaP* and by the Eureka project *EuroTOPS*. Pierre-Yves Calland is supported by a grant of Région Rhône-Alpes.

† Laboratoire LIP, URA CNRS 1398, Ecole Normale Supérieure de Lyon, F - 69364 LYON Cedex 07, e-mail: `Firstname.Lastname@lip.ens-lyon.fr`

[1] Anti dependence occurs when a variable is used in one statement and reassigned in a subsequently executed statement [16].

[2] Output dependence occurs when a variable is assigned in one statement and reassigned in a subsequently executed statement [16].

another technique that can be applied, but it comes later, when moving from virtual to physical processors.

2.1. Renaming. Scalar renaming consists in giving a different name to occurrences of a scalar locally used in a program. This allows the removal of anti and output dependences due to the multiple use of the scalar. This technique can be directly extended to array renaming. Consider the loop nest in Figure 2.1(a). The dependence graph[3] (Figure 2.1(c)) contains a cycle with an anti dependence from statement S_2 to statement S_3. Renaming the array "a" in S_3 (see the code in Figure 2.1(b)) breaks this dependence: the new graph (Figure 2.1(d)) is acyclic. Loop statements can now be parallelized.

For $i = 1$ to N do
  S_1: $\mathrm{a}(i) = \sin(i)$
  S_2: $\mathrm{b}(i+1) = \mathrm{a}(i) + \mathrm{c}(i)$
  S_3: $\mathrm{a}(i) = \cos(i)$
  S_4: $\mathrm{c}(i+1) = \mathrm{a}(i)$
EndFor

(a) original code

For $i = 1$ to N do
  S_1: $\mathrm{renamed}(i) = \sin(i)$
  S_2: $\mathrm{b}(i+1) = \mathrm{renamed}(i) + \mathrm{c}(i)$
  S_3: $\mathrm{a}(i) = \cos(i)$
  S_4: $\mathrm{c}(i+1) = \mathrm{a}(i)$
EndFor

(b) code after renaming

$$\begin{tikzcd} S_1 \arrow[r, "f"] \arrow[d, "o"'] & S_2 \arrow[dl, "a"] \\ S_3 \arrow[r, "f"'] & S_4 \arrow[u, "f"'] \end{tikzcd}$$

(c) original dependence graph

$$\begin{tikzcd} S_1 \arrow[r, "f"] & S_2 \\ S_3 \arrow[r, "f"'] & S_4 \arrow[u, "f"'] \end{tikzcd}$$

(d) graph after renaming

FIG. 2.1. *Example of variable renaming*

2.2. Expansion. Consider a loop nest where a scalar variable is written at several iterations. This implies an output dependence from and to the statement involved in the multiple writings. Consider the example of Figure 2.2(a). Since scalar a is written at each iteration, there is a self output loop around statement S_2 (see the dependence graph in Figure 2.2(c)). To suppress this dependence, we expand a into a linear array, as shown in Figure 2.2(b). Again, the new graph (Figure 2.2(d)) is acyclic.

This technique can be extended to multi-dimensional loop nests for expanding scalars, or for expanding multi-dimensional arrays in the simple cases where the arrays can be considered as scalars when some loop indices are fixed.

2.3. Node splitting. This technique consists in splitting a statement into two statements, in order to break cycles in the dependence graph. Consider the example of Figure 2.3(a). The dependence graph (Figure 2.3(c)) contains a cycle involving a flow dependence from S_1 to S_2, and an output dependence from S_2 to S_1. To break this cycle, rather than writing array "a" in statement S_1, we store the evaluation of

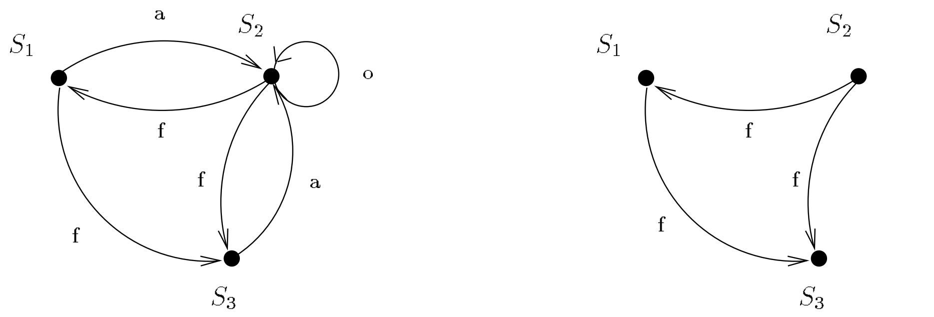
[3]In all figures, flow, anti and output dependence edges are labeled with a *"f"*, a *"a"* and a *"o"* respectively.

For $i = 1$ to N do
  S_1: $\mathrm{c}(i) = 3 + a$
  S_2: $a = i + 1$
  S_3: $\mathrm{b}(i) = \mathrm{c}(i) + a$
EndFor

(a) original code

$\mathrm{temp}(0) = a$
For $i = 1$ to N do
  S_1: $c(i) = 3 + \mathrm{temp}(i-1)$
  S_2: $\mathrm{temp}(i) = i + 1$
  S_3: $b(i) = \mathrm{c}(i) + \mathrm{temp}(i)$
EndFor
If $(N \geq 1)$ then $a = \mathrm{temp}(N)$

(b) code after expansion



(c) original dependence graph *(d) graph after expansion*

FIG. 2.2. *Example of variable expansion*

the right-hand side into a temporary array “temp”. This temporary array is read in S_2 instead of array “a”. The transformed code is given in Figure 2.3(b), and the new dependence graph is represented in Figure 2.3(d).

The previous example is due to Padua and Wolfe [14]. We generalize [6] the statement transformation as indicated in Figure 2.4. The value computed at each iteration of statement S is stored into a temporary array whose access function is the same as that of “lhs”, the left hand side of S. Obviously, if another statement instance depends upon a value “lhs($g(i)$)” computed by S, then the access to “lhs($g(i)$)” must be replaced by temp($g(i)$). This implies knowledge of the statement instances which depend upon the value calculated in S (or in S' after the transformation).

The impact of this transformation on the dependences going to and coming from a statement S is summarized in Figure 2.5. As shown in [6], if this transformation is applied to all the statements of the original loop nest, then the new dependence graph contains only flow dependence cycles and output dependence cycles. Furthermore, these cycles correspond to cycles of the initial dependence graph.

However, applying the transformation to all statements is not a good approach. First, it can be too costly. Moreover, it is useless to transform some statements. Consider for instance a statement S (a vertex of the dependence graph) with an incoming flow dependence f_{in} and an outgoing output dependence o_{out}. We check on Figure 2.5 that the path $\bullet \overset{f_{in}}{\to} S \overset{o_{out}}{\to} \bullet$ has been transformed into the path $\bullet \overset{f_{in}}{\to} S' \overset{f_{new}}{\to} S \overset{o_{out}}{\to} \bullet$. Thus a cycle containing the original path is not broken by the transformation. In fact, the paths that are broken when transforming statement S are those containing an anti or output dependence incoming to S, followed by an anti

For $i = 1$ to N do
  S_1: $\mathrm{a}(i) = \mathrm{b}(i) + \mathrm{c}(i)$
  S_2: $\mathrm{a}(i+1) = \mathrm{a}(i) + 2 \times \mathrm{d}(i)$

(a) original code

For $i = 1$ to N do
  S_1': $\mathrm{temp}(i) = \mathrm{b}(i) + \mathrm{c}(i)$
  S_1: $\mathrm{a}(i) = \mathrm{temp}(i)$
  S_2: $\mathrm{a}(i+1) = \mathrm{temp}(i) + 2 \times \mathrm{d}(i)$

(b) code after node splitting

$$\begin{tikzcd} S_1 \arrow[r, bend right, "f"'] & S_2 \arrow[l, bend right, "o"'] \end{tikzcd}$$

(c) original graph

$$\begin{tikzcd} S_1' \arrow[r, "f"] \arrow[rr, bend right, "f"'] & S_2 \arrow[r, "o"] & S_1 \end{tikzcd}$$

(d) graph after node splitting

FIG. 2.3. *Example of node splitting*

For $i = 1$ to N do
  ...
  S: $\mathrm{lhs}(f(i)) = \mathrm{rhs}(\ldots)$
  ...
  ... $= \mathrm{lhs}(g(i))$

(a) original code

For $i = 1$ to N do
  ...
  S': $\mathrm{temp}(f(i)) = \mathrm{rhs}(\ldots)$
  S: $\mathrm{lhs}(f(i)) = \mathrm{temp}(f(i))$
  ...
  ... $= \mathrm{temp}(g(i))$

(b) code after node splitting

FIG. 2.4. *Transformation of statement S*

or flow dependence going out of S. To summarize, we break paths like

$$? \xrightarrow{a,o} S \xrightarrow{f,a} ?$$

2.4. Conversion to single-assignment form. The only full transformation to single assignment form (SAF) has been proposed by Feautrier in [10]. The technique relies on an exact analysis of direct flow dependences (through parametric integer linear programming) that permits the source of each array reference to be found, i.e. the statement and the value of the surrounding loop counters where the desired element of the array has been computed. Then, the algorithm is the following:

- For each statement S surrounded by d_S loops, define a new d_S-dimensional array M_S and replace the left-hand side of S by $M_S(I)$ where I is the iteration vector associated with S.
- Replace all references in the right-hand side using the corresponding source function: if the source is defined at iteration $f(I)$ of statement S', then refer-

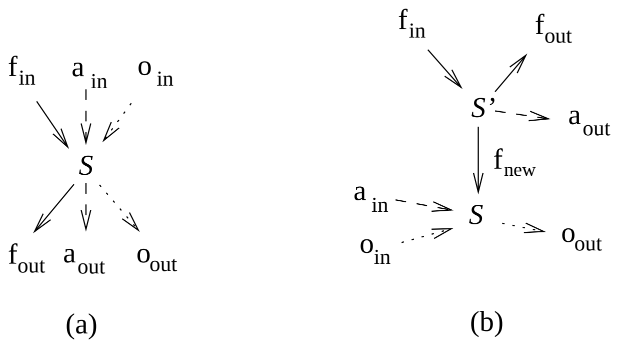


FIG. 2.5. *A statement S with in-coming and out-going dependences (a) before and (b) after generalized Padua and Wolfe's transformation.*

ence $M_{S'}(f(I))$, and if the source is empty, then keep the original reference. In other words, one temporary cell is introduced for each computation, and the right-hand side is modified to reference either a temporary cell for a value modified in the code, or the original scalar or array for a value kept unchanged. All new arrays have as many dimensions as there are loops surrounding their definition.

This transformation has two main weaknesses: the resulting code is in general very complicated, including many "if" tests in the innermost loops; and it requires a very large amount of memory (one cell per computation). Some attempts have been made to remedy these two problems: more sophisticated rewriting techniques have been proposed to move if tests into the outermost loops if possible and to minimize the memory usage (through memory folding, i.e. memory reuse) once parallelism has been detected (see the work of Chamski [7]).

In other words, Chamski's technique consists in three main steps:

1. transform the code into SAF, through full memory expansion,
2. parallelize the code,
3. reduce memory size by analyzing the life duration of each cell in the parallelized code.

In this paper, we explore an opposite approach: first determine anti and output dependences that are responsible for a loss of parallelism, remove them through memory expansion and then parallelize. We believe that this approach is more flexible and powerful to enable various parallelization strategies. We point out that a similar approach is being currently developed in [12], but with a more restricted methodology since the false dependence removal is done *with respect to* a given schedule.

3. Motivating example. We briefly review Allen and Kennedy's algorithm (AK) in Section 3.1. Next we present a simple example, upon which we apply three parallelization schemes. First, in Section 3.2, we apply AK directly on the example. Then, in Section 3.3, we apply AK to the single assignment form of the example. Finally, in Section 3.4, we integrate the false dependences removal techniques into the parallelization process.

3.1. Allen and Kennedy's parallelization algorithm. We summarize Allen and Kennedy's algorithm (AK) because we use this parallelization algorithm throughout the paper. More details on this algorithm can be found in [1, 5, 17].

AK works on a structure called reduced leveled dependence graph (RLDG), i.e. a description of the level of dependences. For a *loop carried dependence*, the *level of* *dependence* is the rank of the first non null component of the distance vectors. This is also the depth of the outermost loop which carries this dependence. For a *loop independent dependence*, the level of dependence is said to be infinite and is denoted ∞. If e is an edge of the RLDG, $l(e)$ denotes its level of dependence.

Before summarizing the algorithm in its simpler form, we need to recall some simple graph definitions:

- A *strongly connected component* of a directed graph G is a maximal subgraph of G in which for any vertices p and q ($p \neq q$) there is a path from p to q;
- The *acyclic condensation* of a graph G is the acyclic graph whose nodes are the strongly connected components $\mathcal{V}_\infty, \ldots, \mathcal{V}_\rfloor$ of G. There is an edge from $\mathcal{V}_\rangle$ to $\mathcal{V}_|$ if there is an edge $e = (x_i, y_j)$ in G such that $x_i \in \mathcal{V}_\rangle$ and $y_j \in \mathcal{V}_|$.
- Let G be a reduced leveled dependence graph. Let H be a subgraph of G. Then $l(H)$ (the *level of* H) is the minimal level of an edge of H: $l(H) = \min\{l(e) \mid e \in H\}$

AK(H, l).

- $H' = H \setminus \{e \mid l(e) < l\}$
- Build H'', the acyclic condensation of H', and number its vertices $\mathcal{V}_\infty, \ldots, \mathcal{V}_\rfloor$ in a topological sort order.
- **For** $i = 1$ **to** c **do**
  1. **If** $\mathcal{V}_\rangle$ is reduced to a single statement S, with no edge, **then** generate parallel "For" loops ("ForPar") in all remaining dimensions (i.e. for levels l to n) and generate code for S.
  2. Otherwise, let $k = l(\mathcal{V}_\rangle)$. Generate parallel "For" loops ("ForPar") for levels from l to $k-1$, and a sequential "For" loop ("ForSeq") for level k. Call AK$(\mathcal{V}_\rangle, k+1)$.

Finally, to apply AK to a reduced leveled dependence graph G, call: *AK(G, 1)*.

3.2. Direct parallelization scheme. The direct parallelization scheme consists in the application of AK on the following loop nest (with two 3-dimensional arrays a and c and one scalar b):

Motivating example.

For $i = 1$ to N do
  For $j = 1$ to N do
    For $k = 1$ to N do
      S_1: $\mathrm{a}(i,j,k) = \mathrm{a}(i,j,k+1) + \mathrm{c}(i,j+1,k) + \mathrm{c}(i,j-1,k) + b$
      S_2: $b = \mathrm{a}(i-1,j,k) + \mathrm{a}(i,j-1,k)$
      S_3: $\mathrm{c}(i,j,k) = \mathrm{c}(i+1,j,k) + \mathrm{c}(i,j,k+1)$
    EndFor
  EndFor
EndFor

The RLDG of the motivating example is drawn[4] on Figure 3.1. This RLDG contains a single strongly connected component which includes the three statements and dependences at level 1. Thus, the outermost loop (loop i) is marked "sequential" by AK. We now remove all level 1 edges: there is still a unique strongly connected component including at least one dependence at level 2. Thus, the second loop (loop j) is marked "sequential". We now remove level 2 edges: there are two strongly connected components, and each component includes dependences at level 3. Thus,

[4]Dependences are those found by Tiny [15]. Some do not actually exist. For instance the anti dependence from S_1 to S_2 (due to scalar b) only occurs at level ∞; the anti dependences at level 1, 2 or 3 are covered.

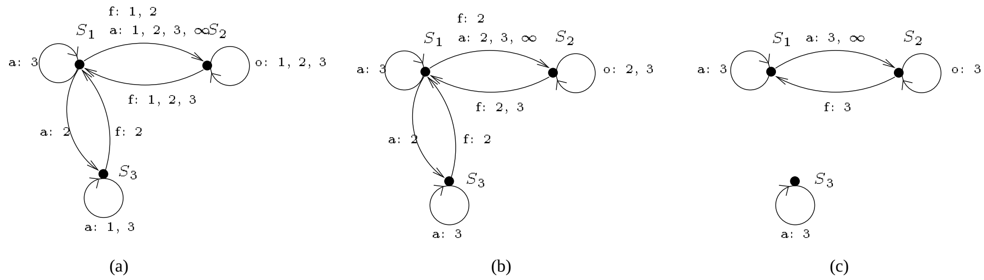


FIG. 3.1. *RLDG of the motivating example: (a) whole RLDG; (b) RLDG without level 1 dependences; (c) RLDG without level 1 and 2 dependences*

the third loop (loop k) is marked "sequential" for both components, and thus for all statements. Thus, AK finds no parallelism in this example when taking into account anti and output dependences, hence the need of removing at least some of the memory based dependences, in order to expose parallelism.

3.3. Parallelization of the single assignment form. Another approach could be first to transform the loop nest into single assignment form (thereby removing *all* memory based dependences), and then to apply the parallelization algorithm.

Motivating example in single assignment form. We first transform the code into a single assignment form, using the transformations of Section 2.4 and we get (assuming $N \geq 1$):

For $i = 1$ to N do
  For $j = 1$ to N do
    For $k = 1$ to N do
      S_1: $\mathrm{atemp}(i,j,k) = \mathrm{a}(i,j,k+1) + \mathrm{c}(i,j+1,k)$
        $+$ if $j \geq 2$ then $\mathrm{ctemp}(i,j-1,k)$ else $\mathrm{c}(i,0,k)$
        $+$ if $k \geq 2$ then $\mathrm{btemp}(i,j,k-1)$
          else if $j \geq 2$ then $\mathrm{btemp}(i,j-1,N)$
            else if $i \geq 2$ then $\mathrm{btemp}(i-1,N,N)$
              else b
      S_2: $\mathrm{btemp}(i,j,k) =$ if $i \geq 2$ then $\mathrm{atemp}(i-1,j,k)$ else $\mathrm{a}(0,j,k)$
        $+$ if $j \geq 2$ then $\mathrm{atemp}(i,j-1,k)$ else $\mathrm{a}(i,0,k)$
      S_3: $\mathrm{ctemp}(i,j,k) = \mathrm{c}(i+1,j,k) + \mathrm{c}(i,j,k+1)$
    EndFor
  EndFor
EndFor

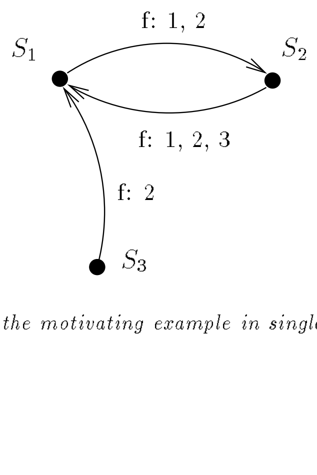


FIG. 3.2. *RLDG of the motivating example in single assignment form*

The new RLDG is drawn in Figure 3.2.

Parallelization of the single assignment form. One can easily see that AK will mark the two outermost loops (loops i and j) "sequential" for statements S_1 and S_2. All other loops will be found "parallel". This is expressed in the parallelized form written below (strictly speaking, we should also copy back atemp(i,j,k) into a(i,j,k), ctemp(i,j,k) into c(i,j,k), and btemp(N,N,N) into b).

```
ForPar i = 1 to N do
    ForPar j = 1 to N do
        ForPar k = 1 to N do
            S3: ctemp(i, j, k) = c(i + 1, j, k) + c(i, j, k + 1)
        EndFor
    EndFor
EndFor
ForSeq i = 1 to N do
    ForSeq j = 1 to N do
        ForPar k = 1 to N do
            S2: btemp(i, j, k) = if i ≥ 2 then atemp(i − 1, j, k) else a(0, j, k)
                    + if j ≥ 2 then atemp(i, j − 1, k) else a(i, 0, k)
        EndFor
        ForPar k = 1 to N do
            S1: atemp(i, j, k) = a(i, j, k + 1) + c(i, j + 1, k)
                    + if j ≥ 2 then ctemp(i, j − 1, k) else c(i, 0, k)
                    + if k ≥ 2 then btemp(i, j, k − 1)
                        else if j ≥ 2 then btemp(i, j − 1, N)
                            else if i ≥ 2 then btemp(i − 1, N, N)
                                else b
        EndFor
    EndFor
EndFor
```

The latency of this parallel program is $\Theta(N^2)$, instead of $\Theta(N^3)$ for the direct parallelized version. Hence our motivating example does contain some parallelism! However, to expose all the parallelism, we have introduced three new arrays of size N^3, which is the size of the iteration domain. We show in the next section that a clever integration of the false dependence removal techniques into the scheduling process enables maximum parallelism to be found while introducing less memory overhead.

3.4. Plugging false dependence removal techniques into the parallelization. The dataflow graph (the RLDG where only flow dependence edges are kept, see Figure 3.3(a)) tells us exactly what amount of parallelism can be found in the program. Our aim is to find in the whole RLDG as much parallelism, while introducing as less memory overhead as possible.

3.4.1. First loop. Consider the first loop: because of the flow dependences, the outermost loop (loop i) must be sequential for statements S_1 and S_2. However this loop could be made parallel for S_3. In order to expose this parallelism, S_3 should not belong any longer to a strongly connected component including some level 1 dependence. Therefore two false dependence edges must be removed, we call them incompatible edges (see Figure 3.3(b)):

- the anti dependence from S_1 to S_3. Because of this edge and of the flow dependence from S_3 to S_1, S_3 is in the same strongly connected component than S_1 and S_2, which contains an edge at level 1.

- the self anti dependence on S_3 at level 1.

Note however that there is no need to remove the output dependence of level 1 from S_1 to S_2.

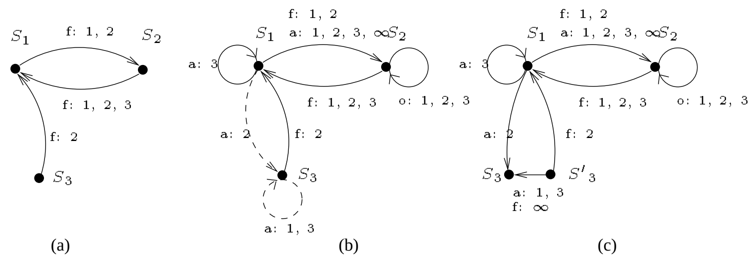


FIG. 3.3. *Motivating example: (a) Dataflow graph (b) RLDG where incompatible edges are dashed (c) RLDG after program transformation*

The two incompatible dependences can be removed by splitting the node S_3 as explained in Section 2.3. This only introduces a single new three dimensional array. The new RLDG is depicted in Figure 3.3(c). We can now apply the first step of Allen and Kennedy's algorithm and we get:

```
ForPar i = 1 to N do
    ForPar j = 1 to N do
        ForPar k = 1 to N do
            S'3: ctemp(i, j, k) = c(i + 1, j, k) + c(i, j, k + 1)
        EndFor
    EndFor
EndFor
ForSeq i = 1 to N do
    For j = 1 to N do
        For k = 1 to N do
            S1: a(i, j, k) = a(i, j, k + 1) + b + c(i, j + 1, k)
                    + if j ≥ 2 then ctemp(i, j − 1, k) else c(i, 0, k)
            S2: b = a(i − 1, j, k) + a(i, j − 1, k)
        EndFor
    EndFor
EndFor
ForPar i = 1 to N do
    ForPar j = 1 to N do
        ForPar k = 1 to N do
            S3: c(i, j, k) = ctemp(i, j, k)
        EndFor
    EndFor
EndFor
```

3.4.2. Second loop. We now consider the second step of AK for the loops surrounding S_1 and S_2 (since the rest of the code is already fully parallelized). The remaining RLDG at level 2 are depicted in Figure 3.4:

Because of the flow dependences at level 2 (see Figure 3.4), the second loop (loop j) must be sequential for statements S_1 and S_2. Removing the anti-dependence at level 2 from S_1 to S_2 will not permit to detect more parallelism with AK. The second

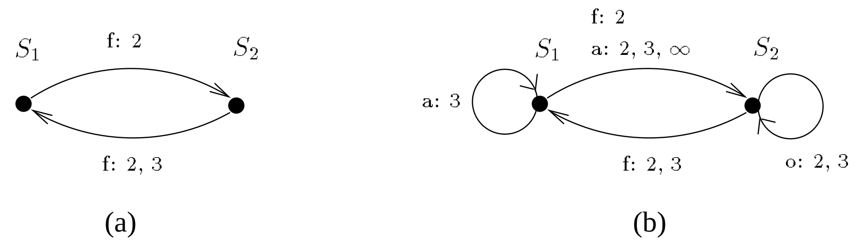


FIG. 3.4. *After first level parallelization: (a) Dataflow graph (b) RLDG*

loop is marked sequential.

3.4.3. Third loop. Considering the dataflow graph of Figure 3.5(a), we see that the innermost loop could be marked parallel because there is no cycle at level 3. In order to expose this parallelism, S_1 and S_2 should not belong any longer to a strongly connected component including some level 3 dependences. Therefore three false dependence edges must be removed (see Figure 3.5(b)):

- the self output dependence on S_2.
- the anti dependence from S_1 to S_2.
- the self anti dependence on S_1.

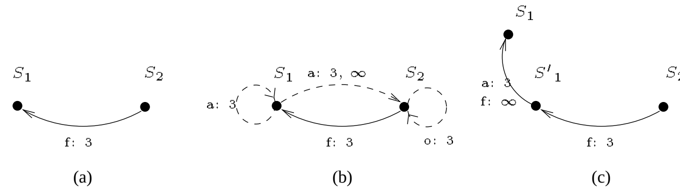


FIG. 3.5. *After second level parallelization: (a) Dataflow graph (b) RLDG where incompatible edges are dashed (c) RLDG after program transformation*

The first two dependences can be removed by expanding the scalar b as explained in Section 2.2. The third dependence can be removed by splitting the node S_1 as explained in Section 2.3. However, instead of introducing two 3-dimensional arrays to suppress these dependences, we introduce only two 1-dimensional arrays "atemp" and "btemp": this is because the outermost two loops are already sequential. Indeed, we only need to remove incompatible dependences *for each iteration of the outermost loops*. This is done as if the outermost two loops indexes were fixed. We first get the code:

ForSeq $i = 1$ to N do
  ForSeq $j = 1$ to N do
    $\mathrm{btemp}(0) = b$
    For $k = 1$ to N do
      S'_1: $\mathrm{atemp}(k) = \mathrm{a}(i,j,k+1) + \mathrm{btemp}(k-1) + \mathrm{c}(i,j+1,k)$
        $+$ if $j \geq 2$ then $\mathrm{ctemp}(i,j-1,k)$ else $c(i,0,k)$
      S_1: $\mathrm{a}(i,j,k) = \mathrm{atemp}(k)$
      S_2: $\mathrm{btemp}(k) = \mathrm{a}(i-1,j,k) + \mathrm{a}(i,j-1,k)$
    EndFor
    $b = \mathrm{btemp}(N)$
  EndFor
EndFor

whose RLDG at level 3 is depicted in Figure 3.5(c). Finally, applying the last step of AK leads to:

```
ForSeq i = 1 to N do
    ForSeq j = 1 to N do
        btemp(0) = b
        ForPar k = 1 to N do
            S_2: btemp(k) = a(i-1, j, k) + a(i, j-1, k)
        EndFor
        ForPar k = 1 to N do
            S'_1: atemp(k) = a(i, j, k+1) + btemp(k-1) + c(i, j+1, k)
                  + if j >= 2 then ctemp(i, j-1, k) else c(i, 0, k)
        EndFor
        ForPar k = 1 to N do
            S_1: a(i, j, k) = atemp(k)
        EndFor
        b = btemp(N)
    EndFor
EndFor
```

4. Plugging false dependence removal techniques into parallelization algorithms. In Section 3, we have shown that integrating the false dependence removal techniques into the parallelization scheme makes it possible to find all the parallelism while introducing less memory overhead. In our example, one 3D array and two 1D arrays have been introduced instead of three 3D arrays as needed by the parallelization of the single assignment form. Note that if external constraints had dictated that introducing a 3D array were too costly, we would have been able to cope with these constraints. We would have exposed less parallelism, but that is the price to pay!

We now summarize our methodology.

4.1. Integration scheme. Instead of removing all possible false dependences first and then applying a parallelization algorithm, we propose to combine both techniques.

Suppose that we use a parallelization algorithm that has the following properties:

- Each statement S surrounded by n_S loops in the original code is surrounded by n_S loops in the parallelized code.
- The choice of the loops surrounding S is made from the outermost to the innermost.

Note that all known parallelization algorithms have these properties: Allen and Kennedy's algorithm, Wolf and Lam's algorithm, Darte and Vivien's algorithm, Feautrier's algorithm.

We propose to plug false dependence removal techniques into loop parallelization algorithms as follows (where G is a given dependence graph):

Parallelization(G).

- Determine false dependences that could be removed by standard dependence removal techniques, as presented in Section 2. Let F be the dependence graph obtained if these dependences were removed. F is the dependence graph that exhibits as much parallelism as can be exposed.
- Apply the parallelization algorithm on F. Let PP be the parallelized program obtained. Each statement S is surrounded in PP by a sequence of n_S loops, marked either sequential or parallel.

- **For** $d = 1$ to $\max(n_S)$ **do**
  1. Mark as *incompatible* all dependences in G but not in F (i.e. false dependences that can be removed) which, if not removed at depth d, will induce a loss of parallelism if PP is taken as reference. In other words, incompatible dependences are those that prohibit the generation of the same number of parallel loops as in PP within the the $n_S - d + 1$ remaining loops surrounding S.
  2. Remove all the incompatible dependences by introducing temporary arrays of dimension as small as possible.
  3. Generate the loop.

4.2. Comments. We do not go on more formally: the integration scheme above is simply the sketch of our methodology. Many problems remain to be solved: how to characterize incompatible edges for an arbitrary parallelization algorithm, how to minimize the number of incompatible dependences that should be removed, how to minimize the dimension of the temporary arrays that have to be introduced, ...However, we have given several examples in Section 3 which should make these problems clearer.

On the dimension of temporary arrays. In theory, we can hope to introduce temporary arrays with as many dimensions as nested loops minus the number of sequential loops already generated. See for example how node splitting has been performed in Figure 3.5: we introduced only a 1D array and not a 3D array for splitting S_1. This adds a new output dependence with level 1 and 2 for S'_1 but it has no effect in terms of parallelization since the two outermost loops are already sequential.

In practice however, the dimension of the temporary arrays may be different. On one hand, we may need extra memory if the false dependence removal technique is not powerful enough to enable code generation: this is the case, for example, if the access functions are too complicated. On the other hand, memory overhead can be reduced by the use of scalar/array privatization, instead of expansion, along the parallel dimensions. This is illustrated in the example of Figure 2.3: the 1D temporary array temp could be transformed into a privatized scalar.

Incompatible edges. Plugging dependence removal techniques into Allen and Kennedy's algorithm is straightforward, because it uses only loop distribution/fusion, which corresponds in terms of graph to the detection of strongly connected components in the RLDG. We have seen in Section 3 that this permits the identification of incompatible edges. They are precisely defined as follows:

For a RLDG H, we denote by H_l the subgraph of H obtained by deleting all edges with level $< l$. Then, incompatible edges at level l are the edges which belong to (at least) one cycle C such that:

1. C contains an edge of level l,
2. C contains a vertex that only belongs in F_l to cycles of level strictly greater than l.

The characterization of incompatible edges for Darte and Vivien's algorithm is much more complicated. We refer to [8, 9] for a complete description of the algorithm. We just give here the flavor of this algorithm.

Darte and Vivien's algorithm takes as input a reduced dependence graph G whose edges are labeled by dependence polyhedra. First, the reduced dependence graph G is uniformized into a graph G_u, which contains the nodes of G and some new nodes, called virtual nodes. Then G_u is processed by the parallelization algorithm as the reduced dependence graph of a system of uniform recurrence equations, except that one has to take into account the fact that some nodes are virtual. Then, the algorithm is recursive and it relies on the construction of a particular subgraph of G_u, the subgraph of null weight multi-cycles, denoted as G'. Incompatible edges are the edges that change the structure of G', more precisely that change the structure of the vector space generated by the weight of the cycles of G' (or of one of the G' that will be recursively defined). We have not studied yet the complexity of this complete characterization.

5. Conclusion. We have presented a framework to plug false dependence removal techniques into loop parallelization algorithms. Our approach has two main advantages. First, we remove only false dependence edges that are responsible for a lesser degree of parallelism, i.e. responsible for the sequentialization of an extra loop. Second, for each dependence removal – hence for each new temporary array – we use our knowledge on the already generated loops to (try to) minimize the number of dimensions of this temporary array.

Furthermore, our algorithmic sketch can be controlled by external parameters. For instance, we can require that each statement is surrounded by the same number of sequential loops. More important, we can straightforwardly cope with external constraints on the total memory overhead which is allowed. For instance if only 2D-arrays may be introduced, we will generate only 1D or 2D temporaries, at the price of some loss in parallelism.

Further work should be devoted to a better characterization of "incompatible" edges, and to a precise estimation of the extra memory that is needed to expose all the potential parallelism.

REFERENCES

[1] J.R. Allen and K. Kennedy. Automatic translations of Fortran programs to vector form. *ACM Toplas*, 9:491–542, 1987.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4), 1994.

[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[4] Thomas Brandes. The importance of direct dependences for automatic parallelization. In *International Conference of Supercomputing*, pages 407–417, 1988.

[5] D. Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Dept. of Computer Science, Rice University, Houston, TX, 1987.

[6] Pierre-Yves Calland, Alain Darte, Yves Robert, and Fr d ric Vivien. On the removal of anti and output dependences. In J. Fortes, C. Mongenet, K. Parhi, and V. Taylor, editors, *Application Specific Systems, Architectures and Processors*, pages 353–364. IEEE Computer Society Press, 1996.

[7] Zbigniew Chamski. *Environnement logiciel de programmation d'un acc l rateur de calcul parall le*. PhD thesis, Universit de Rennes, Rennes, France, 1993. num ro 957.

[8] Alain Darte and Frédéric Vivien. A classification of nested loops parallelization algorithms. In *INRIA-IEEE Symposium on Emerging Technologies and Factory Automation*, pages 217–224. IEEE Computer Society Press, 1995.

[9] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism in polyhedral reduced dependence graphs. Technical Report 96-06, LIP, ENS-Lyon, France, April 1996.

[10] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[11] Junjie Gu, Zhiyan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing 95*, 1995.

[12] V. Lefebvre and P. Feautrier. Gestion de la mémoire dans les programmes parallèles. In Richard Castanet and Jean Roman, editors, *RenPar'8*, pages 149–152, LaBRII, Université de Bordeaux, France, May 1996. In French.

[13] Dror E. Maydan, Saman P. Amarasinghe, and Monica Lam. Array data-flow analysis and its use in array privatization. In *Principles of Programming Languages*, 1993.

[14] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[15] Michael Wolfe. The Tiny loop restructuring research tool. In H.D. Schwetman, editor, *International Conference on Parallel Processing*, volume II, pages 46–53. CRC Press, 1991.

[16] Michael Wolfe. *High Performance Compilers For Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[17] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.