Vectorization Using Reversible Data Dependences

Peiyi Tang and Nianshu Gao

Technical Report
ANU-TR-CS-94-08
October 21, 1994

Vectorization Using Reversible Data Dependences [*]

Peiyi Tang

Department of Computer Science
The Australian National University
Canberra ACT 0200 Australia

Nianshu Gao

Institute of Computing Technology
Academia Sinica
Beijing 100080 China

October 21, 1994

Abstract

Data dependences between statements have long been used for detecting parallelism and converting sequential programs into parallel forms. However, some data dependences can be reversed and the transformed program still produces the same results. In this paper, we revisit vectorization and propose a new vectorization algorithm using reversible data dependences. The new algorithm can generate more or thicker vector statements than traditional algorithm. The techniques presented in this paper can be incorporated in all the existing vectorizing compilers for supercomputers.

[*]This work was supported in part by **the Australian Research Council** under Grant No. A49232251

1 Introduction

Data dependences between statements have long been used by vectorizing and parallelizing compilers to detect parallelism and convert sequential programs into parallel forms [1, 2]. Two statement instances[1] are said to be data dependent if they access the same data element and at least one of the accesses is a write. The direction of the data dependence is determined by the order of sequential execution of the program.

Enforcing all the data dependences between dependent statement instances when parallelizing the program will keep the original orders of definitions and uses of data elements and, thus, guarantee that the parallelized program will produce the same results. However, to produce the same results does not necessarily require all the definition and use orders to be maintained. For example, there are n instances of statement s in the following loop: $s(1), \cdots, s(n)$.

```
   do i = 1, n
     ...
s: x = x + a(i)
   enddo
```

Assume that statement s is the only statement that accesses variable x in the loop. The variable x is defined and used by all instances of s. The original order of definitions and uses of x is such that its value defined by $s(i-1)$ is used by $s(i)$ $(2 \leq i \leq n)$ and any data dependence test will indicate that there are flow, anti and output dependences from $s(i-1)$ to $s(i)$ caused by variable x. However, the statement instances $s(1), \cdots, s(n)$ can be executed in any order as long as they are executed one after another [3].

The statements like s in the above example are called *accumulative* statements. A program can have many accumulative statements on the same variable and the variables accessed by the statements can also be array elements. For instance, both statements s_1 and s_2 in the following loop

```
   do i = 1, n
```

[1]A statement instance is an execution of the statement in loops for particular loop iteration. A statement not enclosed in any loops has only one instance.

```
s1:  x(2i) = x(2i) + a(i)
s2:  x(i+3) = x(i+3) - b(i)
   END
```

are accumulative statements. The values of the subscript functions are shown in the following table:

i	1	2	3	4	5
2i	2	4	6	8	10
i+3	4	5	6	7	8

The dependences between the statement instances are shown in Figure 1(a), where each node is a statement instance and the label of an edge is the data element that causes the dependence. For instance, both $s_1(2)$ and $s_2(1)$ access $x(4)$ and there are flow, anti and output dependences from $s_2(1)$ to $s_1(2)$ (these three dependences are represented by a single edge). The data dependence graph between the statements of the loop is shown in Figure 1(b). Due to the dependence cycle involving s_1 and s_2, the loop cannot be distributed over the statements and none of them can be vectorized by the traditional vectorization algorithm [1,4].

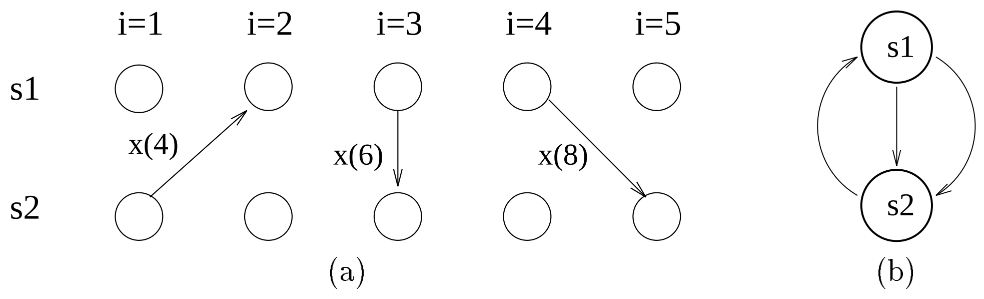


Figure 1. dependences of An Example

However, $s_1(2)$ can be executed before $s_2(1)$ even though there are data dependences from $s_2(1)$ to $s_1(2)$. The final value of $x(4)$ will still be the same: $x'(4)-b(1)+a(2)$, where $x'(4)$ is the initial value of $x(4)$ before the loop starts. This indicates that the dependence from s_2 to s_1 in Figure 1(b) can be reversed and then the loop can be vectorized as follows:

```
s1:  x(2:10:2) = x(2:10:2) + a(1:5:1)
s2:  x(6:10:1) = x(6:10:1) - b(1:5:1)
```

The dependence from s_2 to s_1 is called *reversible dependence*. In fact, the dependence from s_1 to s_2 is also a reversible dependence.

Reversible dependences can be reversed to vectorize and parallelize some parts of the programs that cannot not be vectorized or parallelized otherwise. In this paper, we revisit vectorization and propose a new vectorization algorithm which can generate more or thicker vector statements than the traditional algorithm. Our algorithm can be incorporated in any existing vectorizing compilers for supercomputers.

The paper is structured as follows. In section 2, we provide definitions of many important concepts needed for vectorization and the definition of reversible data dependences. In section 3, we present and prove the new vectorization algorithm extended from the traditional algorithm. Section 4 concludes the paper with a short discussion and a brief summary.

2 Reversible Dependences

In this section, we introduce the basic concepts of data dependence and data dependence graph used by compiler vectorization and parallelization algorithms. More detailed descriptions can be found in [1,4]. We then define a special type of data dependences called *reversible dependences*.

Data Dependence

Given a statement s enclosed in n loops with index variables $i_1, \cdots, i_n$, a statement instance of s for a particular loop index $\vec{i} = (i_1, \cdots, i_n)$, denoted by $s(\vec{i})$, is an execution of s in the loop iteration of $\vec{i}$. The sets of data elements defined and used by $s(\vec{i})$ are denoted by $\mathbf{def}(s(\vec{i}))$ and $\mathbf{use}(s(\vec{i}))$, respectively.

Definition 1 *Given two statements s and s' in a program*[2]*, there is a data dependence from $s(\vec{i})$ to $s'(\vec{i'})$, denoted by $s(\vec{i})\delta s'(\vec{i'})$, if*

1. *$s(\vec{i})$ is executed before $s'(\vec{i'})$ (this relation is denoted by $s(\vec{i}) \prec s'(\vec{i'})$), and*
2. *at least one of the following sets is non-empty:*
   (a) $\mathbf{def}(s(\vec{i})) \cap \mathbf{use}(s'(\vec{i'}))$
   (b) $\mathbf{use}(s(\vec{i})) \cap \mathbf{def}(s'(\vec{i'}))$
   (c) $\mathbf{def}(s(\vec{i})) \cap \mathbf{def}(s'(\vec{i'}))$

If $s(\vec{i})\delta s'(\vec{i'})$, and $\vec{i}|_{k-1} = \vec{i'}|_{k-1}$ and $\vec{i}(k) < \vec{i'}(k)$ where k is the level of one of their common loops, we say $S'(\vec{i'})$ is dependent on $S(\vec{i})$ *at level k*. Here, $\vec{i}|_{k-1}$ is the subvector of $\vec{i}$ that consists of the first $k-1$ elements and $\vec{i}(k)$ is the k-th element of $\vec{i}$. This data dependence is denoted by $S(\vec{i})\delta_k S'(\vec{i'})$ and is also called *loop-carried* dependence. If the corresponding indices of $\vec{i}$ and $\vec{i'}$ for all the common loops are the same, the dependence is called *loop-independent* dependence, and is denoted by $S(\vec{i})\delta_\infty S'(\vec{i'})$. Loop-independent dependences are also called dependences at level ∞.

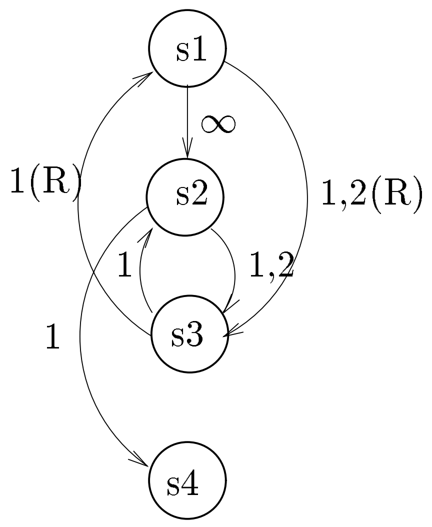
Let us have a look the program in Figure 2(a) which is extended from the example in Section 1. We can see that there are data dependences from $s_3(1,k)$ to $s_1(2,k-1)$ for $2 \leq k \leq 5$ because both access data element $x(4,k-1)$ and $s_3(1,k)$ is executed before $s_1(2,k-1)$ (i.e. $s_3(1,k) \prec s_1(2,k-1)$). This dependence is carried by loop `I` of level 1. Table 1 lists all the data dependences between the statement instances of the program in Figure 2(a).

Note that our definition of dependence does include *indirect dependences* [5]. That is, if there is a statement instance $s''(\vec{i''})$ such that $s(\vec{i}) \prec s''(\vec{i''}) \prec s'(\vec{i'})$ and $\mathbf{def}(s''(\vec{i''}))$ includes a data element in one of the three intersections of **def** and **use** sets of $s(\vec{i})$ and $s'(\vec{i'})$ in (2) of the definition above, we still say $S(\vec{i})\delta S'(\vec{i'})$ and this dependence is called *indirect dependence.* If there is no such $s''(\vec{i''})$ exits, the dependence is called *direct dependence.* For instance, the data dependences from $s_3(1,k)$ to $s_2(2,k-1)$ $(2 \leq k \leq 5)$ in our example are indirect dependences, because we can find $s_1(2,k-1)$ that $s_3(1,k) \prec$

[2]For the sake of simplicity, a program in this paper is defined as a general loop nest with the outermost loop encloses all the assignment statement.

```
   do i = 1, 5
     do j = 1, 5
s1:    x(2i,j) = x(2i,j) + a(i,j)
s2:    y(i,j) = x(2i,j) + c(i,j)
s3:    x(i+3,j-1) = x(i+3,j-1) - b(i,j)
     enddo
s4:  z(i) = y(i-1,1) + d(i)
   enddo
```

(a) program



(b) DD Graph

Figure 2. Vectorization Example

$s_1(2, k-1) \prec s_2(2, k-1)$ and it defines the same data element $(x(4, k-1))$ as $s_3(1, k)$ and $s_2(2, k-1)$ access. The reason to include indirect dependences is to make data dependence tests simple and efficient. The compiler only needs to examine the pair-wise relations between statements. This kind of dependence information is accurate enough for vectorization. As will be shown later, these indirect dependences can become direct dependences after the directions of reversible dependences are changed.

The dependence information between statement instances is too detailed to be represented in the form of graphs. dependences between statements, defined as follows, are summaries of dependences between statement instances and used by vectorizing compilers for vectorization.

Definition 2 *There is a data dependence from s to s' at level $k \in Z \cup \{\infty\}$, denoted by $S\delta_k S'$, if there are statements instance $s(\vec{i})$ and $s'(\vec{i'})$ such that $s(\vec{i})\delta_k s'(\vec{i'})$.*

Accumulative Statements

We are interested in a special class of assignment statements in loops that are called *accumulative* statements and defined as follows:

Stmt Pair	From	To	Constraint	level
s_1, s_3	$s_3(1,k)$	$s_1(2,k-1)$	$2 \le k \le 5$	1
	$s_1(3,k)$	$s_3(3,k+1)$	$1 \le k \le 4$	2
	$s_1(4,k)$	$s_3(5,k+1)$	$1 \le k \le 4$	1
s_1, s_2	$s_1(q,k)$	$s_2(q,k)$	$1 \le q \le 5$ $1 \le k \le 5$	∞
s_2, s_3	$s_3(1,k)$	$s_2(2,k-1)$	$2 \le k \le 5$	1
	$s_2(3,k)$	$s_3(3,k+1)$	$1 \le k \le 4$	2
	$s_2(4,k)$	$s_3(5,k+1)$	$1 \le k \le 4$	1
s_2, s_4	$s_2(q,1)$	$s_4(q+1)$	$1 \le q \le 4$	1

Table 1. Data dependences Between Statement Instances

Definition 3 *An assignment statement is an accumulative statement for variable x if it is of the following form:*

$$x(\mathbf{f}(\vec{i})) = x(\mathbf{f}(\vec{i})) \oplus exp(\vec{i})$$

where x is an array or scalar variable, $\vec{i}$ the index vector of the enclosing loops, $\mathbf{f}$ the subscript function[3]*, $\oplus$ an arithmetic binary operator and $exp(\vec{i})$ an expression that does not use data elements of x.*

Let $x(\vec{s})$ be an element of x accessed by the statement and

$$I_{\vec{s}} = \{\vec{i} : \mathbf{f}(\vec{i}) = \vec{s}\}$$

be the set of index vectors that access $x(\vec{s})$. Let $I_{\vec{s}} = \{\vec{i}_1, \cdots, \vec{i}_p\}$. The contribution of the statement to the value of $x(\vec{s})$ is

$$exp(\vec{i}_1) \oplus \cdots \oplus exp(\vec{i}_p)$$

assuming that $\oplus$ is commutative and associative. The order between $\vec{i}_1, \cdots, \vec{i}_p$ is not important. In our example of Figure 2, statements s_1 and s_3 are accumulative statements and statements s_2 and s_4 are not.

[3]If x is a scalar, $\mathbf{f}$ is a function that maps all the index vectors to 0 and x is interpreted as an array with only one element $x(0)$.

Interchangeable Operators

Two accumulative statements that access the same array may use different binary operators.

Definition 4 *An unordered pair of binary commutative and associative operators, $\{\oplus, \ominus\}$, defined on a real number field R is said to be interchangeable if and only if*

$$\forall a, b, c \in R, (a \oplus b) \ominus c = (a \ominus c) \oplus b.$$

Operator pairs $\{+,+\}$, $\{+,-\}$, $\{-,-\}$, $\{*,*\}$, $\{*,/\}$ and $\{/,/\}$ are all interchangeable, where `+`,`-`,`*` and`/` are binary operators of addition, subtraction, multiplication and division of real numbers, respectively. On the other hand, operator pairs $\{+,*\}$, $\{-,*\}$, $\{+,/\}$ and $\{-,/\}$ are not interchangeable.

Reversible Data dependences

Suppose that there are two accumulative statements s and s' for the same variable x as follows:

$$s: \quad x(\mathbf{f}(\vec{i})) = x(\mathbf{f}(\vec{i})) \oplus exp(\vec{i})$$

$$s': \quad x(\mathbf{g}(\vec{i'})) = x(\mathbf{g}(\vec{i'})) \ominus exp'(\vec{i'})$$

where $\{\oplus, \ominus\}$ is interchangeable. The two statements may access the same data elements of x causing data dependences between them. However, because the operators $\oplus$ and $\ominus$ are interchangeable, the orders they update the data elements are not important. Let $x(\vec{b})$ be a particular element of x and

$$I_{\vec{b}} = \{\vec{i} : \mathbf{f}(\vec{i}) = \vec{b}\} = \{\vec{i}_1, \cdots \vec{i}_p\}$$

and

$$I'_{\vec{b}} = \{\vec{i'} : \mathbf{g}(\vec{i'}) = \vec{b}\} = \{\vec{i'}_1, \cdots \vec{i'}_q\}$$

be the index sets of s and s' that access $x(\vec{b})$, respectively. The total contribution of s and s' to the value of $x(\vec{b})$ is

$$(exp(\vec{i}_1) \oplus \cdots \oplus exp(\vec{i}_p)) \ominus (exp(\vec{i'}_1) \ominus \cdots \ominus exp(\vec{i'}_q))$$

The order between the elements of $I_{\vec{b}}$ and $I'_{\vec{b}}$ is not important because $\{\oplus, \ominus\}$ are interchangeable. The data dependences between these statement instances can be reversed without changing this total contribution. Data dependences between the statements like s and s' are called *reversible dependences* and defined as follows:

Definition 5 *Given two accumulative statements for the same variable that use a pair of interchangeable operators, data dependences between, if any, are called reversible dependences.*

Reversible dependences are not difficult to detect during data dependence tests. All we need to do is to check whether the two statements are accumulative for the same variable and whether they use interchangeable operators.

Data Dependence Graph

The dependences between statements are represented by *data dependence graph* $G = (N, E)$, where the set of nodes N is the set of statements and the set of edges E represents the data dependences between the statements. The data dependence graph in this paper is constructed as follows:

1. There is an edge $e = (s, s') \in E$ labeled with $L_e = \{l_1, \cdots, l_p\}$ if and only if there is at least one data dependence from statement s to statement s' at level $l_r \in Z \cup \{\infty\}$ for each r such that $1 \leq r \leq p$. L_e is called the *set of levels* associated with edge e.

2. The edge $e = (s, s') \in E$ is marked as "reversible"(R) if and only if both s and s' are accumulative statements for the same variable and they use a pair of interchangeable operators.

The data dependence graph of our example is shown in Figure 2(b).

3 Vectorization

In this section, we provide a new vectorization algorithm extended from original Allen-Kennedy algorithm. This algorithm takes advantage of reversible dependences and produces more and thicker vector statements than the original algorithm.

A program is vectorized by calling a recursive procedure "vectorize" as follows:

$$\text{vectorize}(STAT, 1)$$

where $STAT$ is the set of all assignment statements in the program.

The new algorithm for procedure "vectorize" is shown in Figure 3. To explain what is happening and why this algorithm can generate more and thicker vector statements, we need to go back the basic theory of vectorization and have a close look at the original Allen-Kennedy Algorithm [1,4].

Original Algorithm

Given a statement s enclosed by n loops, statement instances $s(\vec{i})$ and $s(\vec{i'})$ are said to be *different at level k* if $\vec{i}|_{k-1} = \vec{i'}|_{k-1}$ and $\vec{i}(i) \neq \vec{i'}(k)$. The central question in vectorization is whether two different instances of a statement at certain levels can be executed in parallel.

Given a path in the data dependence graph $s_1 \rightarrow_{e_1} s_2 \rightarrow_{e_2} \cdots \rightarrow_{e_r} s_{r+1}$, denoted as $\pi = (e_1, \cdots, e_r)$, the *level of the path* is defined by

$$levels(\pi) = \{k \in Z \cup \{\infty\} : (\forall q \in [1,r], k \leq \max(L_{e_q})) \land (\exists q \in [1,r] \;\; s.t. \;\; k \in L_{e_q})\}$$

where $[1, r]$ is the set of integers between 1 and r. Recall that L_{e_q} is the set of levels of all data dependences associated with e_q. It can be shown that for each level $l \in levels(\pi)$ we have $1 \leq l \leq \min_{e_q \in \pi}(\max(L_{e_q}))$. If there is a cycle path C from s to s in the data dependence graph such that $k \in levels(C)$, cannot be safely executed in parallel because it is possible that there are direct or indirect data dependences between them. On the other hand, if for every cycle C from s to s we have $k \notin levels(C)$, the different instances of s at level k can be executed in parallel.

```
procedure vectorize($(R, k)$);
/* R is a set of assignment statements */
/* k is a level */
(1)   $D_k := constrain(D, k)$
(2)   $DD := D_k|R$
(3)   $G' = (N', E') := AC(DD)$
(4)   for every $n' \in N'$ in topological order do
(5)       if $n'$ is a non-trivial SCC, $G_i = (N_i, E_i)$, of DD
(6)         done := false
(7)         for each reversible edge $e_i \in E_i$ do
(8)             reverse $e_i$ of $G_i$ to form a new graph $G_i'$
(9)             $GG = (NN, EE) := AC(G_i')$
(10)            if there is only one SCC in $GG$ (i.e. NN=$\{G_i\}$ and EE=$\phi$)
(11)              discard $GG$
(12)              continue /* with loop for $e_i$ */
(13)            else
(14)              for each $nn \in NN$ in topological order do
(15)                  if $nn$ is a non-trivial SCC $G_j = (N_j, E_j)$ of $GG$
(16)                    generate level $k$ DO statement
(17)                    vectorize($N_j, k + 1$)
(18)                    generate level $k$ ENDDO statement
(19)                  else /* $nn$ is a trivial node without cycles */
(20)                    generate vector statement for $nn$
(21)                  endif
(22)              endfor /* loop for $nn$ */
(23)              done = true
(24)              break /* with loop for $e_i$ */
(25)            endif
(26)        endfor /* loop for $e_i$ */
(27)        if not done
(28)            generate level $k$ DO statement
(29)            vectorize($N_i, k + 1$)
(30)            generate level $k$ ENDDO statement;
(31)        endif
(32)      else /* $n'$ is a trivial node without cycles */
(33)        generate vector statement for $n'$
(32)      endif
(34)  endfor /* loop for $n'$ */
(35)  endprocedure
```

Figure 3. Recursive Procedure “vectorize”

The original algorithm can be obtained from the algorithm in Figure 3 by removing lines (6)-(27) and (31). The algorithm works recursively starting from level 1. Given a dependence graph $D = (N, E)$, function $constrain(D, k)$ at line (1) returns a graph D_k with the same set of nodes N, but only with those edges $e \in E$ such that $k \leq \max(L_e)$. Given $D_k = (N_k, E_k)$ and $R \subseteq N_k$, $D_k|R$ in line (2) gives the subgraph of $DD = (R, E_R)$ such that $E_R = \{(s, s') \in E_k : s \in R \wedge s' \in R\}$. The function $AC(DD)$ in line (3) gives the *acyclic condensation* (AC) after finding out all the strongly connected components (SCC) using Tarjan's algorithm. The acyclic condensation $AC(DD)$ is a graph $G_i = (N_i, E_i)$ whose nodes are SCCs of DD and there is an edge between two SCCs if there are edges between the nodes in these SCCs. The algorithm follows a topological order of the acyclic condensation and distributes the loops over the SCCs. When processing each SCCs at level k, the algorithm tests if the node $n' \in N_i$ is an non-trivial SCC or a single node without cycles.

It can be proved that node n' is an non-trivial SCC if and only if every node of the SCC is in a cycle in the original data dependence graph going through all the nodes of the SCC and the levels of the cycle l satisfy $k \leq l \leq \min_{e \in C}(\max(L_e))$.

Vectorization at level k is armed at parallelizing the instances of statements at *all* levels equal to *and* greater than k. When processing a non-trivial SCC, none of its statements can be vectorized because they are in a cycle with levels l satisfying $k \leq l \leq \min_{e \in C}(\max(L_e))$. The algorithm generates the sequential DO loop at level k for this SCC (see lines (28) and (30)) to enforce the data dependences at level k and recursively calls "vectorize" at the next level $k + 1$.

If node n' is a single node without cycle, for any cycle C in the original data dependence graph going through node n', we must have $\min_{e \in C}(\max(L_e)) < k$. These cycle data dependences must have been enforced at the previous levels and we can vectorize n' from level k through the innermost level.

Note that dependences not in cycles (i.e., dependences between SCCs of the acyclic condensation) are enforced by the loop distribution following the topological order. Since all the data dependences of the program are enforced, the vectorized program will produce the same results for all data elements as the original program.

To vectorize the program of our example in Figure 2, we first obtain the acyclic

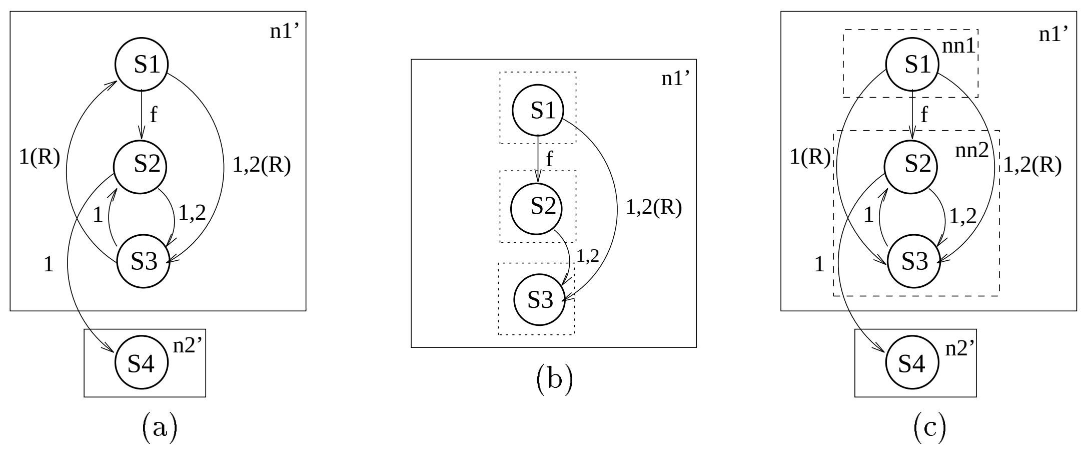


Figure 4. Vectorization

condensation of the entire program as shown in Figure 4(a). There are two SCCs in the acyclic condensation: n'_1 is a non-trivial SCC and n'_2 is a single statement without cycle. n'_2 which is s_4 is vectorized and the sequential loop `I` is generated for n'_1. Figure 4(b) shows the acyclic condensation of n'_1 at level 2. All of s_1, s_2 and s_3 can be vectorized at level 2. The vectorized program is as follows:

```
    do i = 1, 5
      s1: x(2i,1:5) = x(2i,1:5) + a(i,1:5)
      s2: y(i,1:5) = x(2i,1:5) + c(i,1:5)
      s3: x(i+3,0:4) = x(i+3,0:4) - b(i,1:5)
    enddo
s4: z(1:5) = y(0:4,1) + d(1:5)
```

Extended Algorithm

We extended the original vectorization algorithm by adding lines (6)-(26) and **if** statement in line (27) and (31). What the added lines do is to try to reverse reversible data dependence edges one at a time to see if the non-trivial SCC can be broken into smaller SCCs. If it is successful, the algorithm proceeds to process these smaller SCCs following the topological order between them. If none of the reversible edges can break the SCC, the algorithm proceeds to generate a sequential loop as in the original algorithm.

In our example, when processing the SCC for s_1, s_2 and s_3 at level 1, we change the direction of the reversible data dependence edge $e = (s_3, s_1)$. The original SCC n_1' is now broken into two smaller SCCs: nn_1 and nn_2. This situation is shown in Figure 4(c). Our algorithm proceeds to vectorize s_1 at level 1 and further process nn_2, resulting in the following vector program:

```
s1:  x(2:10:2,1:5) = x(2:10:2,1:5) + a(1:5,1:5)
     do i = 1, 5
s2:    y(i,1:5) = x(2i,1:5) + c(i,1:5)
s3:    x(i+3,0:4) = x(i+3,0:4) - b(i,1:5)
     enddo
s4:  z(1:5) = y(0:4,1) + d(1:5)
```

We can see that our algorithm vectorizes s_1 at level 1 instead of level 2 and generates a thicker vector statement for s_1 than the original algorithm.

Reversing data dependence edges changes the original data dependence graph. Reversible edges may be reversed at different times. In general, reversing an edges might create new cycles that does not exist in the original data dependence graph. To establish the correctness and effectiveness of our algorithm we need to show that

1. The extended algorithm generates at least as many vector statements as the original algorithm.

2. The vector program generated by the extended algorithm produces the same results for all the data elements as the original sequential program.

These two points are summarized in Theorem 1 and Theorem 2. In order to prove these theorems, we need the following lemmas.

Lemma 1 *If reversing a reversible $e_i \in E_i$ can break the non-trivial SCC, $G_i = (N_i, E_i)$, into smaller SCCs (i.e. the control goes to line (14)), then e_i must be included in any cycle that goes through all nodes of N_i.*

Proof: If there is a cycle that goes through all nodes of N_i does not include e_i, this cycle would still exist after e_i is reversed. This means that reversing e_i cannot break the

SCC into smaller SCCs. □

Lemma 2 *If reversing a reversible $e_i = (s, s') \in E_i$ $(s \neq s')$ can break the non-trivial SCC, $G_i = (N_i, E_i)$, into smaller SCCs (i.e. the control goes to line (14)), then the reversed edge $e'_i = (s', s)$ must be an edge between different SCCs of GG (see line (9) for the definition of GG).*

Proof: Assume that s and s' are still in the same strongly-connected component, nn, of GG after $e_i = (s, s')$ is reversed. Then there is a cycle going through s and s' in nn that does not use e_i. This means that there is a path p from s to s' in G_i that does not include e_i. Given a cycle C of G_i going through all the nodes in N_i, if it includes e_i we can replace it with p and obtain a cycle that contains all nodes but does include e_i. It is contradictory to Lemma 1. □

Lemma 3 *Each reversible dependence edge may be reversed only once and the reversed dependences are always enforced by loop distribution.*

Proof: From Lemma 2, after a reversible dependence edge is reversed, it will become an edge between the SCCs. It is enforced by the loop distribution following the topological order and will never appear in the any graphs of recursive calls at deeper levels. □

Theorem 1 *The extended algorithm will not generate more sequential DO loops than the original algorithm.*

Proof: The extended algorithm generate sequential DO loops at line (28) and (16). According to Lemma 3, a reversible edge can be reversed only once and the reversed edge is across between different SCCs. Therefore, the non-trivial SCC found in line (15) only contains the original dependence edges. It is obvious that the non-trivial SCC processed by lines (28)-(30) also contains the original dependence edges only. It follows that there must be a cycle C in the original data dependence graph that goes through all the nodes of the SCC and has a level l satisfying $k \leq l \leq \min_{e \in C}(\max(L_e))$. Therefore, the sequential DO loops generated by the extended algorithm would be also be generated by the original algorithm. $\square$

Now let us show the correctness of the extended algorithm.

Lemma 4 *If a SCC, nn, is found to be a single node without cycle in the acyclic condensation (see line (19)) after the reversible edge e_i is reversed, then any cycle originally in G_i going through nn must include e_i.*

Proof: If there is a cycle going through nn does not include e_i, then this cycle still exists after e_i is reversed. Then nn would not be single node. $\square$

Theorem 2 *The vector program generated by the extended algorithm produces the same results for all the data elements as the original sequential program.*

Proof: The extended algorithm generates vector statements at lines (33) and (20).

From Lemma 3, when "vectorize(R, k)" is called, all edges in the graph DD (see lines (1) and (2)) are original. When the SCC, n', is found to be a single node without cycle, it can be guaranteed that there is no cycle C going through n with levels l satisfying $k \leq l \leq \min_{e \in C}(\max(L_e))$. The statement can be safely parallelized in the levels from k to the innermost level (see line (30)). The cycles going through n' with levels $l < k$ are either enforced by the sequential loops generated in previous levels or by loop distribution.

If a statement is found to be single node without cycle nn at line (19), it is possible to have a cycle C going through nn with level l satisfying $k \leq l \leq \min_{e \in C}(\max(L_e))$. However, according to Lemma 4, this cycle must include the reversible edge e_i. The different instances of nn at level l can be executed in parallel because after e_i is reversed the cycle does not exist anymore. All the other data dependences in the cycle are enforced by the loop distribution. The vectorization at line (20) only change the order of the statement instances connected by e_i and all the other data dependences in the cycle are enforced.

The original data dependence graph include explicitly all the indirect data dependences between the statements. Since all the data dependences are enforced except some reversible data dependences, the final result for all the data elements of the program is the same as original sequential program. □

It is important to note that the correctness of the new algorithm relies on the fact that original data dependence graph includes all indirect dependences. In our example in Figure 2, the dependence $s_3\delta_1 s_2$ is indirect. After the reversible dependence from $s_3\delta_1 s_1$ is reversed, $s_3\delta_1 s_2$ becomes a direct dependence. The presence of indirect dependences in the data dependence graph guarantees that reversing reversible dependences will not have side effects on other data dependences.

4 Discussion and Conclusion

Almost all automatic parallelization or vectorization schemes are based on enforcing the data dependences detected in programs. In this paper, we pointed out that enforcing all the data dependences is sufficient to generate parallel programs that produce the same results, but it is not necessary. We defined a special type of data dependences whose directions can be reversed. By reversing this type of data dependences in a controllable way, we can generate parallel programs with more parallelism than the traditional schemes of parallelization. This paper is the first attempt to explore parallelism for automatic parallelization beyond the limit of data dependences.

While this paper only concentrates on vectorization, the idea should be extended to parallelization to generate parallel DOALL and DOACROSS loops. M. Wolfe suggested to include another kind of parallel loop called DOANY loop in parallel languages [6]. DOANY is a parallel loop whose iterations can be executed in any order and can be used to express the parallelism that cannot be expressed by DOALL or DOACROSS loops. The idea to exploiting reversible dependences in this paper should be used to generate DOANY loops from sequential programs. We are going to research on these problems in the future.

References

[1] R. Allen and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," *ACM Transactions on Programming Languages and Systems,*, vol. 9, no. 4, pp. 491–541, October 1987.

[2] M. Wolfe, *Optimizing Compilers for Supercomputers.* Cambridge, MA, MIT Press, 1989.

[3] R. Eigenmann, J. Hoeflinger, Z. Li and D. Padua, "Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs," in *Proceedings of the 4th (1991) of Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, No. 589* , Santa Clara, California, USA, August, 1991, pp. 65–83.

[4] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers.* New York, NY, ACM Press, 1990.

[5] W. Pugh and D. Wonnacott, "Eliminating False Data Dependences Using the Omega Test," in *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Francisco, California, June 17-19, 1992, pp. 140–151.

[6] M. Wolfe, "Doany: Not Just Another Parallel Loop," in *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, August 1992, pp. 277–282.