Vectorization

- Determine whether statements in an inner loop can be vectorized by directly rewriting them to in Fortran 90

```
DO I = 1, N
  X(I) = X(I) + C
ENDDO
```

Vectorization →

```
X(1:N) = X(1:N) + C
```

- Semantics: read all values on the right hand side, apply operations, store all results on the left hand side

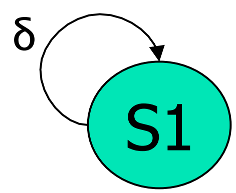
Example

```
DO I = 1, N
S1   X(I+1) = X(I) + C
                   <1>
ENDDO
```

→

```
X(2:N+1) = X(1:N) + C
```

This transformation is incorrect!



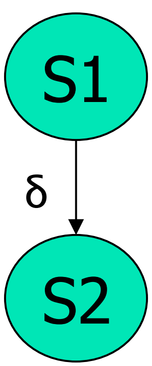
Example

```
DO I = 1, N
S1  A(I+1) = B(I) + C
          <1>
S2   D(I) = A(I) + E
ENDDO
```

```
A(2:N+1) = B(1:N) + C
D(1:N) = A(1:N) + E
```

This right!

```
DO I = 1, N
   A(I+1) = B(I) + C
ENDDO
DO I = 1, N
   D(I) = A(I) + E
ENDDO
```



Legality of Vectorization

- Theorem: A statement contained in at least one loop can be vectorized by directly rewriting in Fortran 90 if the statement is not included in any cycle of dependence

A Simple Algorithm

```
Vectorization(L, D)  // L is loop nest, D is dependence graph for L
{
   find the set {S_1, S_2, …, S_m} of maximal strongly connected regions
      in the dependence graph D restricted to L;
   construct L_π from L by reducing each S_i to a single node and
       compute D_π, the dependence graph naturally induced
       on L_π by D
  Let {π_1, π_2, …, π_m} be the m nodes of L_π numbered
    in topological order
  for i=1 to m do {
      if π_i is a dependence cycle then
         generate a DO-loop around the statements in π_i
      else
         directly vectorize the single statement in π_i
  }
}
```

Example

DO I = 1, N

δ_1

A(I) = A(I-1) + 10

δ_∞^{-1}

B(I) = B(I) + 10

δ_∞ δ_∞

C(I) = A(I) + B(I)

ENDDO

Advanced Vectorization

- Consider the following example. The simple algorithm skips it

```
DO I = 1, N
   DO J = 1, M
      A(I+1, J) = A(I,J)+B

   ENDDO          <1, 0>
ENDDO
```

We'd like to vectorize inner loops if applicable

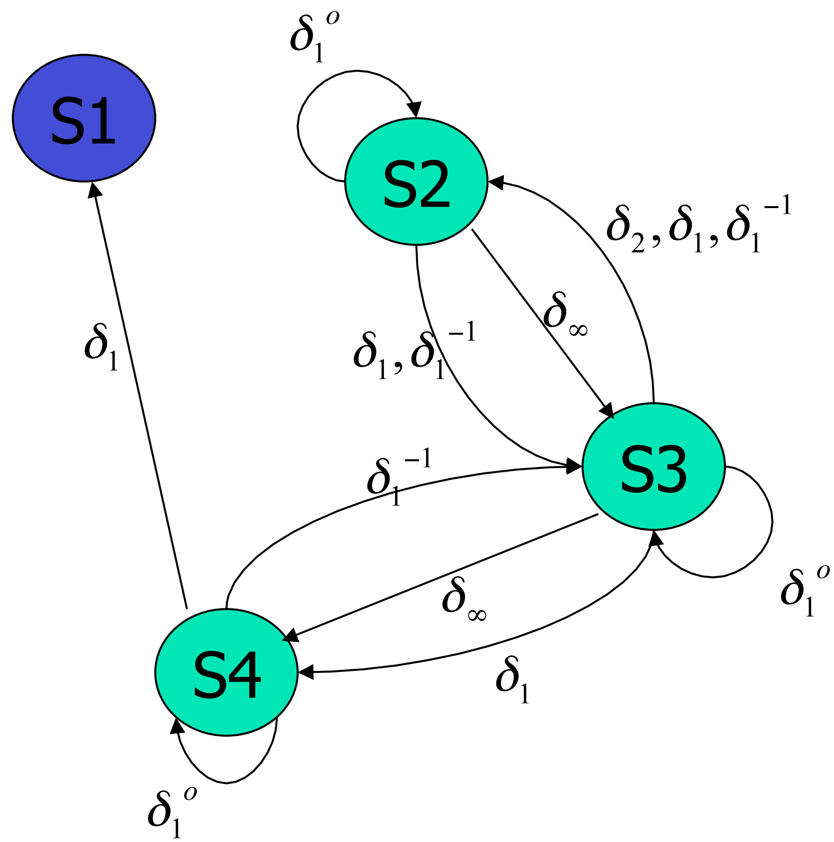
```
DO I = 1, N
    A(I+1, 1:M) = A(I, 1:M)+B
 ENDDO
```

An Advanced Algorithm

```
AdvacedVectorization(R, k, D)  // R is the region of concern
                  // k is the loop level // D is dependence graph for R
{
    find the set {S1, S2, …, Sm} of  maximal strongly connected regions
         in the dependence graph D restricted to R;
    construct Rπ from R by reducing each Si to a single node and
        compute Dπ, the dependence graph naturally induced on Rπ by D;
   Let {π1, π2, …, πm} be the m nodes of Rπ numbered in topological order;
   for i=1 to m do {
       if πi is a dependence cycle then
           generate a level-k DO statement;
           Let Di be the dependence graph consisting all dependence edges in D
              that are at level k+1 or greater and are internal to πi;
          AdvancedVectorization(πi, k+1, Di);
          generate a level-k ENDDO statement;
       else
          directly vectorize the single statement in πi;
 }
}
```

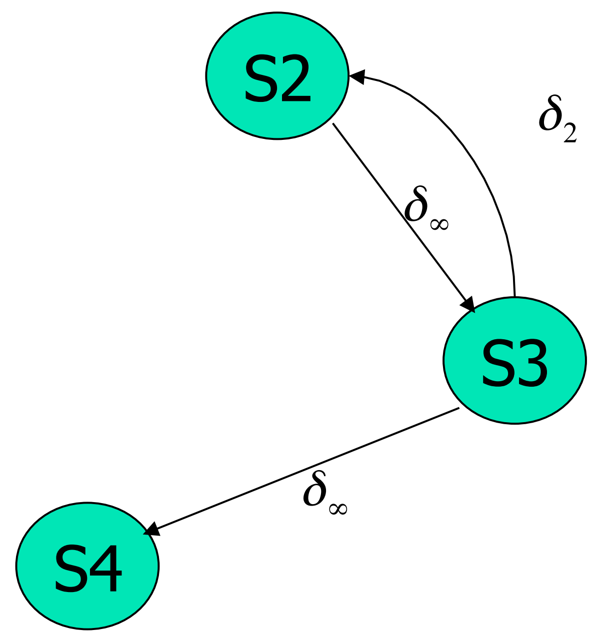
Example

```
DO I = 1, 100
S1   X(I) = Y(I) +10
     DO J = 1, 100
S2      B(J) = A(J, N)
        DO K = 1, 100
S3         A(J+1, K) = B(J) + C(J,K)
        ENDDO
S4      Y(I+J) = A(J+1, N)
     ENDDO
ENDDO
```



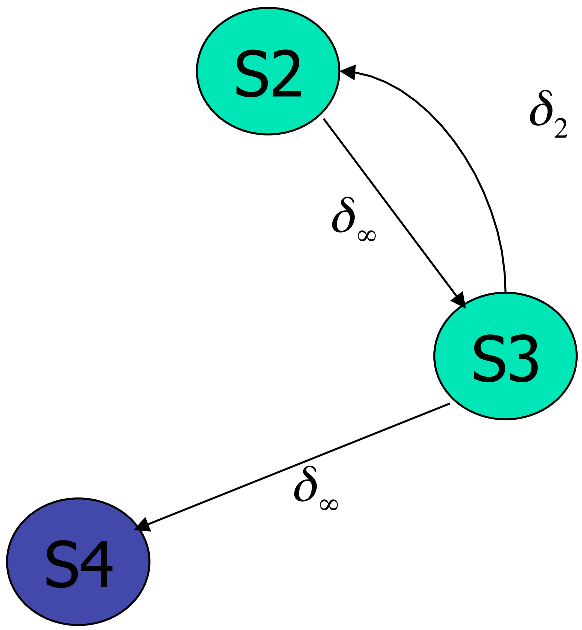
Example cont.

```
DO I = 1, 100
  AdvancedVectorization({S2,S3,S4},2)
ENDDO
X(1:100) = Y(1:100) + 10
```



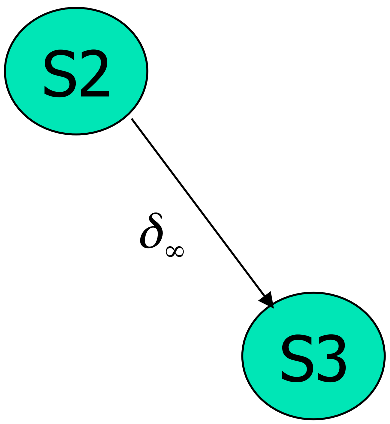
Example cont.

```
DO I = 1, 100
   DO J = 1, 100
     AdvancedVectorization({S2,S3},3)
   ENDDO
   Y(I+1:I+100) = A(2:101, N)
ENDDO
X(1:100) = Y(1:100) + 10
```



Example cont.

```
DO I = 1, 100
   DO J = 1, 100
      B(J) = A(J, N)
      A(J+1, I:100) = B(J) + C(J, 1:100)
   ENDDO
   Y(I+1:I+100) = A(2:101, N)
ENDDO
X(1:100) = Y(1:100) + 10
```



Loop Interchange

- Loop interchange
  - Switching the nesting order of two loops in a perfect nest
  - Can move inner loop-carried dependence out so as to vectorize the new inner loop

```
DO I = 1, N
   DO J = 1, M
      A(I, J+1) = A(I,J)+B
   ENDDO
ENDDO
```

interchange

```
DO J = 1, M
   DO I = 1, N
      A(I, J+1) = A(I,J)+B
   ENDDO
ENDDO
```

```
DO J = 1, M
  A(1:N, J+1) = A(1:N,J)+B
ENDDO
```

Profitability of Interchange

- It’s best to vectorize the least significant dimension
  - First dimension in Fortran

```
DO I = 1, N
   DO J = 1, M
     DO  K = 1, L
        A(I+1, J+1, K) = A(I,J, K)+B
   ENDDO
ENDDO
```

```
DO I = 1, N
   A(I+1, 2:M+1, 1:L)=A(I, 1:M, 1:L)+B
ENDDO
```

Not efficient for hardware

↓ interchange

```
DO J = 1, M
   DO K = 1, L
     DO  I = 1, N
        A(I+1, J+1, K) = A(I,J, K)+B
   ENDDO
ENDDO
```

```
DO J = 1, M
   DO K = 1, L
      A(2:N+1,J+1, K) = A(1:N,J,K)+B
   ENDDO
ENDDO
```

Loop Shifting

- It's best to vectorize the least significant dimension
  - First dimension in Fortran

```
DO I = 1, N
   DO J = 1, N
      DO  K = 1, N
         A(I, J) = A(I,J)+B(I,K)*C(K,J)
      ENDDO
   ENDDO
ENDDO
```

shifting

```
DO K = 1, N
   DO I = 1, N
      DO  J = 1, N
         A(I, J) = A(I,J)+B(I,K)*C(K,J)
      ENDDO
   ENDDO
ENDDO
```

```
DO K = 1, N
   FORALL (J=1,N)
      A(1:N, J) = A(1:N,J)+B(1:N,K)*C(K,J)
   END FORALL
ENDDO
```

Loop Shifting through Interchange

- Theorem
  - In a perfect loop nest, if loops at level i through $i+n$ carry no dependence – that is, all dependences carried by loops at levels less than i or greater than $i+n$ – it is always legal to shift these loops inside of the loop at level $i+n+1$. Furthermore, these loops will not carry any dependences in their new position

```
Select_loop_and_interchange(πi, k, Di)
{
  if the outmost carried dependence in πi is at level p>k
    shift loops at level k, k+1, …, p-1 inside the level-p loop
}
```

Advanced Algorithm with Shifting

```
AdvacedVectorization(R, k, D)  // R is the region of concern
            // k is the loop level // D is dependence graph for R
{
   find the set {S1, S2, …, Sm} of  maximal strongly connected regions
      in the dependence graph D restricted to R;
   construct Rπ from from R by reducing each Si to a single node and
      compute Dπ, the dependence graph naturally induced on Rπ by D;
  Let {π1, π2, …, πm} be the m nodes of Rπ numbered in topological order;
  for i=1 to m do {
     if πi is a dependence cycle then
        select_loop_and_interchange(πi, k, Di)
        generate a level-k DO statement;
        Let Di be the dependence graph consisting all dependence edges in D
          that are are at level k+1 or greater and are internal to πi;
       AdvancedVectorization(πi, k+1, Di);
       generate a level-k DO statement;
     else
       directly vectorize the single statement in πi;
  }
}
```

General Loop Selection and Interchange

- In following example, both loops carry dependences.
  - So simple loop shifting does not work
- But interchange will create an opportunity for vectorization for the first dimension

```
DO I = 1, N
   DO J = 1, M
      A(I+1,J+1)=A(I,J)+A(I+1, J)
                        (<, <)
                                 (=, <)
   ENDDO
ENDDO
```

```
DO J = 1, M
   A(2:N+1, J+1)=A(1:N, J)+A(2:N+1, J)
ENDDO
```

General Loop Selection Algorithm

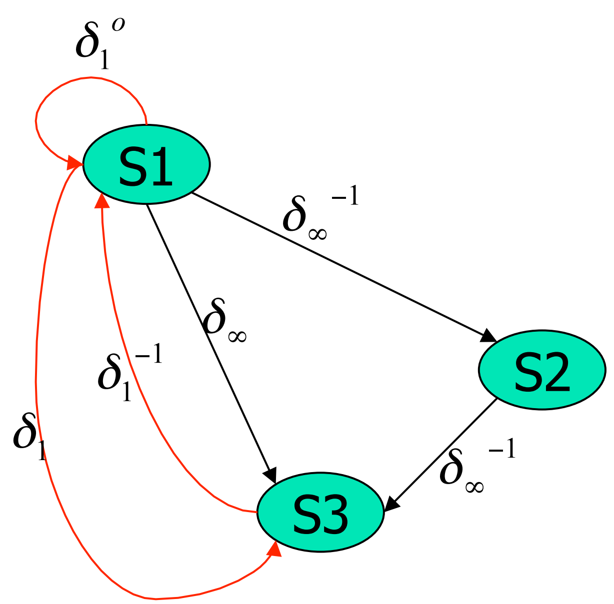
Find the loop
which must
be sequentialized

```
Select_loop_and_interchange(πi, k, Di)
{
  Let N be the deepest loop level;
  let p be the level of the outermost carried dependence
  if (p==k) { // a new case
      not_found = true;
      p++;
      while (not_found and p <= N) {
          if the level-p loop can be safely shifted outward to level k
              and there exists a dependence d carried by the loop such that
              the direction vector for d has "=" in every position but p then
              not_found = false;
          else p = p+1;
      }
      if (p >N) p = k;
  }
  if (p>k)
     shift loops at level k, k+1, …, p-1 inside the level-p loop
}
```

Scalar Expansion

- Scalar expansion: expand a scalar reference to an array reference so as to remove dependences
  - Eliminate dependences that arise from reuse of memory locations
- Example
  - Swap the contents of two vectors

```
DO I = 1, N
S1   T = A(I)
S2   A(I) = B(I)
S3   B(I) = T
ENDDO
```

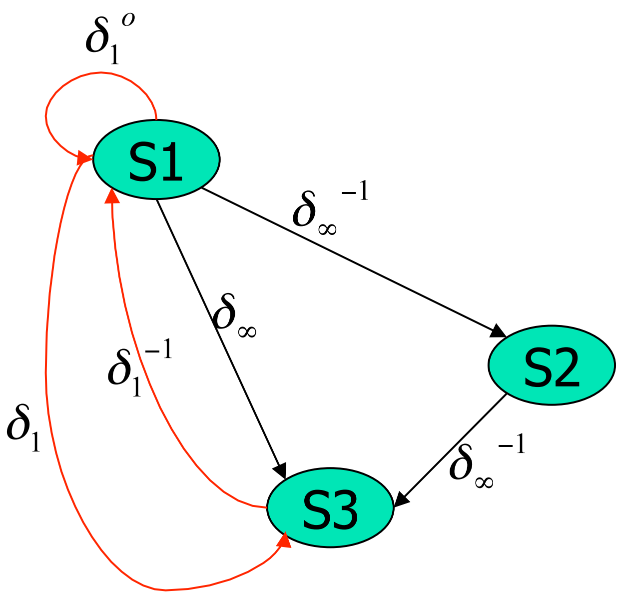


We cannot vectorize this loop without scalar expansion

Scalar Expansion

```
DO I = 1, N
S1   T = A(I)
S2   A(I) = B(I)
S3   B(I) = T
ENDDO
```

```
DO I = 1, N
S1   T$(I) = A(I)
S2   A(I) = B(I)
S3   B(I) = T$(I)
ENDDO
T = T$(N)
```




```
T$(1:N) = A(1:N)
A(1:N) = B(1:N)
B(1:N) = T$(1:N)
T = T$(N)
```

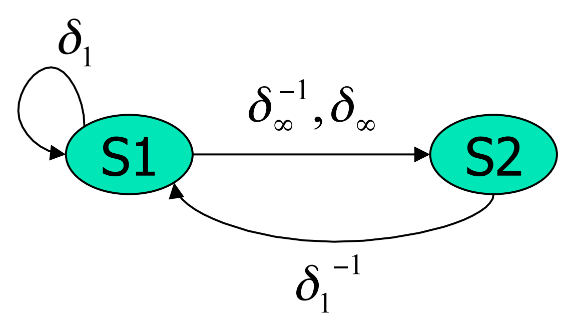
Scalar expansion eliminate red edges

Profitability

- Scalar expansion does not always yield an opportunity for vectorization
  - We need to find which edges can be deleted by scalar expansion

```
DO I = 1, N
S1   T = T + A(I) + A(I+1)
S2   A(I+1) = T
ENDDO
```

```
DO I = 1, N
S1   T$(I) = T$(I-1) + A(I) + A(I+1)
S2   A(I+1) = T$(I)
ENDDO
```



Finding Deletable Edges

- We'd like to know which edges can be deleted by scalar expansion
- *Covering definition* and a *collection of covering definitions*
  - A definition X of a scalar S is a covering definition for loop L if a definition of S placed at the beginning of L reaches no uses of S that occur past X. Whenever there're multiple covering definitions, the term "covering definition" will refer the earliest
  - Extends to a collection of covering definitions

```
DO I = 1, N
S1   T = X(I)
S2   Y(I) = T
ENDDO
```

```
DO I = 1, N
  IF (A(I).GT.0) THEN
     T = X(I)
  ELSE
     T = -X(I)
  Y(I) = T
ENDDO
```

Finding Deletable Edges

- Theorem
  - If all uses of a scalar T before any member of the collection of covering definitions are expanded as T$(I-1) and all other uses and definitions are expanded as T$(I), then the edges that will be deleted with the scalar expansion are
    1. Backward-carried anti-dependences
    2. All carried output dependences
    3. Loop-independent anti-dependences prior to the covering definition
    4. Redundant forward-carried true dependences

Scalar Renaming

```
DO I = 1, 100
S1   T = A(I) + B(I)
S2   C(I) = T + T
S3    T = D(I) – B(I)
S4    A(I+1) =T*T
ENDDO
```

renaming

```
DO I = 1, 100
S1   T1 = A(I) + B(I)
S2   C(I) = T1 + T1
S3    T2 = D(I) – B(I)
S4    A(I+1) =T2*T2
ENDDO
```

```
S3   T2$(1:100) = D(1:100) – B(1:100)
S4   A(2:101) =T2$(1:100)*T2$(1:100)
S1   T1$(1:100) = A(1:100) + B(1:100)
S2   C(1:100) = T1$(1:100) + T1$(1:100)
      T=T2$(100)
```

Scalar renaming is essentially free

Array Renaming

- Renaming arrays to remove unnecessary (loop independent) anti- and output dependences because of reuse of memory location

```
DO I = 1, 100
   A(I) = A(I-1) + X
   Y(I) = A(I) + Z
   A(I) = B(I) + C
ENDDO
```

δ_∞ δ_∞^{o} δ_∞^{-1} δ_1

renaming

```
DO I = 1, 100
   A$(I) = A(I-1) + X
   Y(I) = A$(I) + Z
   A(I) = B(I) + C
ENDDO
```

δ_∞ δ_1

Node Splitting

- Certain anti-dependence cannot be removed by array renaming because name conflicts
- Creates a new copy to split the name space
  - Can use a scalar for the copy, and scalar expansion will make it an array
- Algorithm
  - For a loop independent constant anti-dependence, create a copy to a new temporary, T$, from the source.
  - Replace the source with T$
  - Update dependence graph

Node Splitting

```
DO I = 1, 100

    A(I) = X(I+1) + X(I)
```

δ_∞^{-1} δ_1

```
    X(I+1) = B(I) + 10

ENDDO
```

Two name spaces cannot split by renaming

```
DO I = 1, 100
    X$(I) = X(I+1)
```

δ_∞^{-1}

```
    A(I) = X$(I) + X(I)
```

δ_1 δ_∞^{-1}

```
    X(I+1) = B(I) + 10

ENDDO
```

```
X$(1:N) = X(2:N+1)
X(2:N+1) = B(1:N) + 10
A(1:N) = X$(1:N)+X(1:N)
```