COMP 515: Advanced Compilation for Vector and Parallel Processors

Prof. Krishna Palem
Prof. Vivek Sarkar
Department of Computer Science
Rice University
{palem,vsarkar}@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP515

Acknowledgments

- Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
  - http://www.cs.rice.edu/~ken/comp515/

Homework 1: Solution to Problem 2.2

```
DO K = 1, 100
  DO J = 1, 100
    DO I = 1, 100
      A(I+1,J,K) = A(I,J,5) + B
    END DO
  END DO
END DO
```

A statement with a single lval cannot have a loop-independent flow or output dependence

Consider flow dependence from iteration (k,j,i) to iteration (k',j',i')

- Occurs when k=5, j=j', i+1=i'. Also k ≤ k' for all plausible flow dependences ➔ direction vector for flow dependence must be (≤,=,<)

Consider anti dependence from iteration (k,j,i) to iteration (k',j',i')

- Occurs when 5=k', j=j', i=i'+1. Also k < k' for all plausible anti dependences ➔ direction vector for anti dependence must be (<,=,>)

Consider output dependence from iteration (k,j,i) to iteration (k',j',i')

- Occurs when k=k', j=j', i+1=i'+1. Not possible for a loop-carried dependence. ➔ no output dependence

Homework 1: Solution to Problem 2.3

Recap: dependence vectors for loop nest in Problem 2.2

$= \{ (\leq,=,<), (<,=,>) \}$

$= \{ (<,=,<), (=,=,<), (<,=,>) \}$

- Note that the K and I loop both carry dependences, but the middle J loop does not. Therefore, the J loop can be executed in parallel as follows:

```
DO K = 1, 100
  PARALLEL DO J = 1, 100  ! Parallel loop
    DO I = 1, 100
      A(I+1,J,K) = A(I,J,5) + B
    END DO
  END PARALLEL DO
END DO
```

Dependence: Theory and Practice

(Loop Distribution, Vectorization) Algorithm)

Allen and Kennedy, Chapter 2

Loop Distribution

- **Can statements in loops which carry dependences be vectorized?**

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

- **Dependence: $S_1 \, \delta_1 \, S_2$ can be converted to:**

```
S1    A(2:N+1) = B(1:N) + C
S2    D(1:N) = A(1:N) + E
```

Loop Distribution

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

- **transformed to:**

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

- **leads to:**

```
S1      A(2:N+1) = B(1:N) + C
S2      D(1:N) = A(1:N) + E
```

Loop Distribution

- **Loop distribution fails if there is a cycle of dependences**

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

$S_1 \ \delta_1 \ S_2$ **and** $S_2 \ \delta_1 \ S_1$

- **What about:**

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

Simple Vectorization Algorithm

procedure vectorize (L, D)

// L is the maximal loop nest containing the statement.

// D is the dependence graph for statements in L.

find the set $\{S_1, S_2, \ldots, S_m\}$ of maximal strongly-connected regions in the dependence graph D restricted to L (Tarjan);

construct L_p from L by reducing each S_i to a single node and compute D_p, the dependence graph naturally induced on L_p by D;

let $\{p_1, p_2, \ldots, p_m\}$ be the m nodes of L_p numbered in an order consistent with D_p (use topological sort);

```
for i = 1 to m do begin
        if p_i is a dependence cycle then
    generate a DO-loop nest around the statements in p_i;
        else
    directly rewrite p_i in Fortran 90, vectorizing it with respect to every loop
      containing it;
 end
```

end vectorize

Problems With Simple Vectorization

```
      DO I = 1, N
              DO J = 1, M
S1                    A(I+1,J) = A(I,J) + B
              ENDDO
      ENDDO
```

- Dependence from S_1 to itself with d(i, j) = (1,0)
- Key observation: Since dependence is at level 1, we can vectorize the inner loop!
- Can be converted to:

```
      DO I = 1, N
S1       A(I+1,1:M) = A(I,1:M) + B
      ENDDO
```

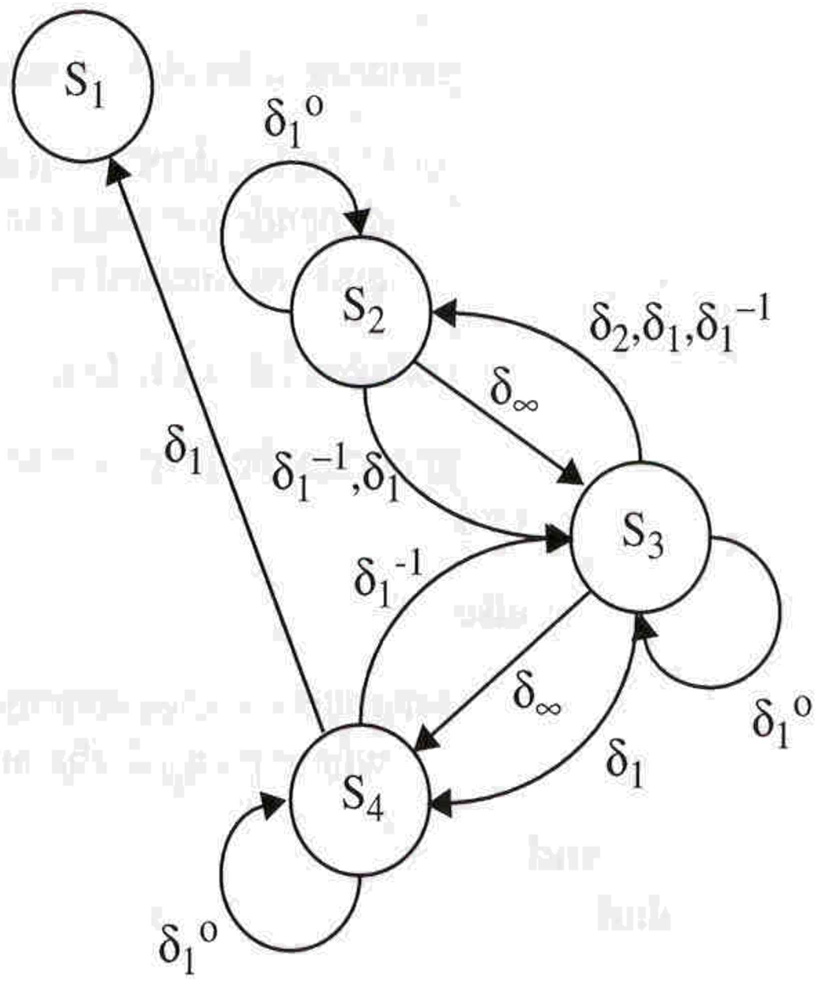
- The simple algorithm does not capitalize on such opportunities

Advanced Vectorization Algorithm

```
procedure codegen(R, k, D);
// R is the region for which we must generate code.
// k is the minimum nesting level of possible parallel loops.
// D is the dependence graph among statements in R..
find the set {S1, S2, ... , Sm} of maximal strongly-connected
    regions in the dependence graph D restricted to R;
construct Rp from R by reducing each Si to a single node and
    compute Dp, the dependence graph naturally induced on Rp by D;
let {p1, p2, ... , pm} be the m nodes of Rp numbered in an order
    consistent with Dp (use topological sort to do the numbering);
for i = 1 to m do begin
    if pi is cyclic then begin
        generate a level-k DO statement;
        let Di be the dependence graph consisting of all dependence edges in D that are at level
            k+1 or greater and are internal to pi;
        codegen (pi, k+1, Di);
        generate the level-k ENDDO statement;
    end
    else
        generate a vector statement for pi in r(pi)-k+1 dimensions, where r (pi) is the number of
            loops containing pi;
end
```

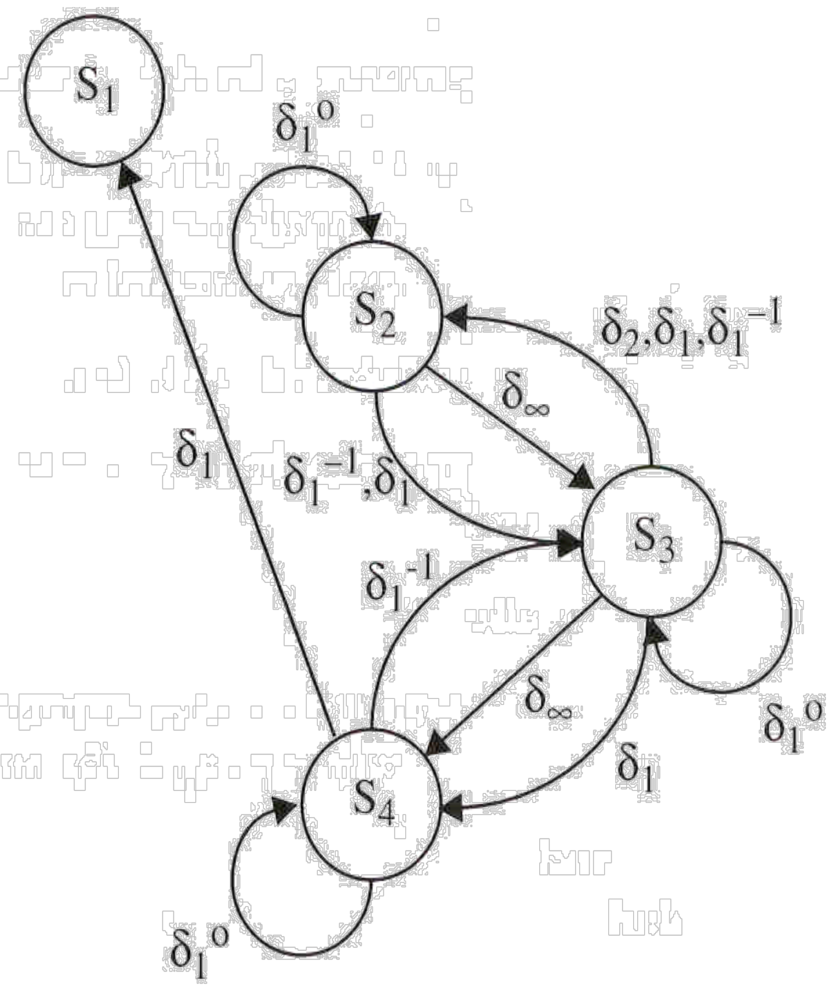
Advanced Vectorization Algorithm

```
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
```



Advanced Vectorization Algorithm

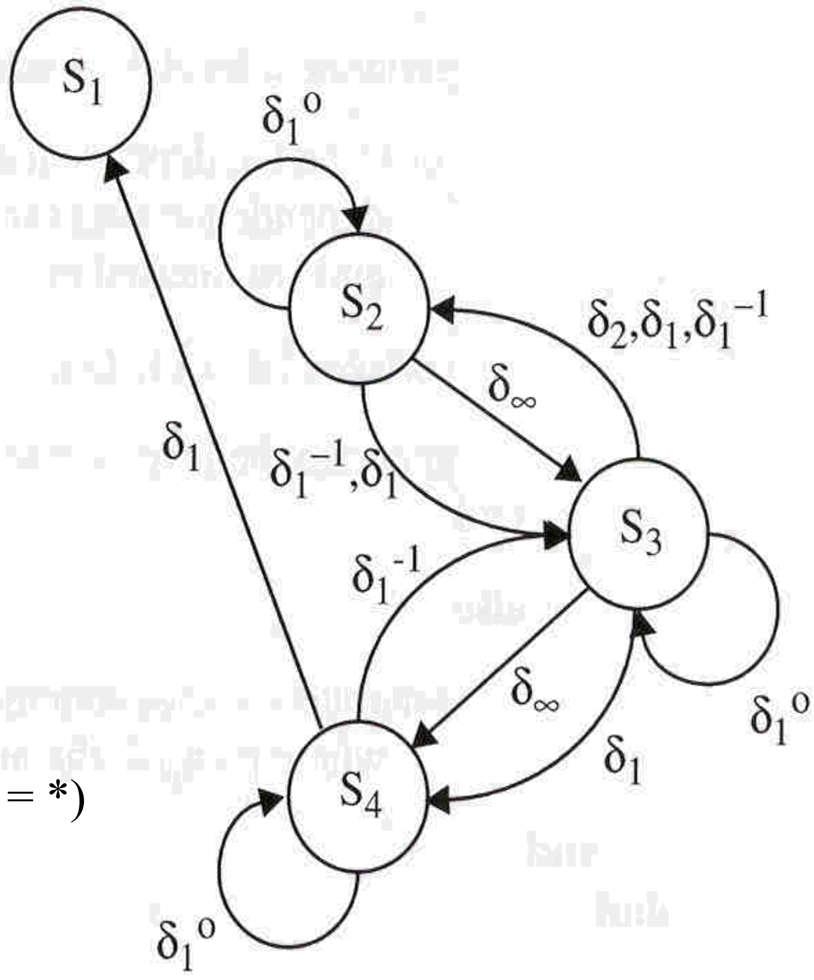
```
DO I = 1, 100
S1    X(I) = Y(I) + 10
        DO J = 1, 100
S2        B(J) = A(J,N)
            DO K = 1, 100
S3            A(J+1,K)=B(J)+C(J,K)
            ENDDO
S4        Y(I+J) = A(J+1, N)
        ENDDO
ENDDO
```



Simple dependence testing procedure:
True dependence from S_4 to S_1
$I_0 + J = I_0 + \Delta I$
$\Rightarrow \Delta I = J$
As J is always positive
$\Rightarrow$ Direction is "<"

Advanced Vectorization Algorithm

```
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
```



S_2 and S_3: dependence via B(J)
I does not occur in either subscript (D.V = *)
We get:
$J_0 = J_0 + \Delta J$
$\Rightarrow \Delta J = 0$
$\Rightarrow$ Direction vectors = (*, =)

Advanced Vectorization Algorithm

- codegen called at the outermost level
- S_1 will be vectorized, and moved later due to topological sort

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

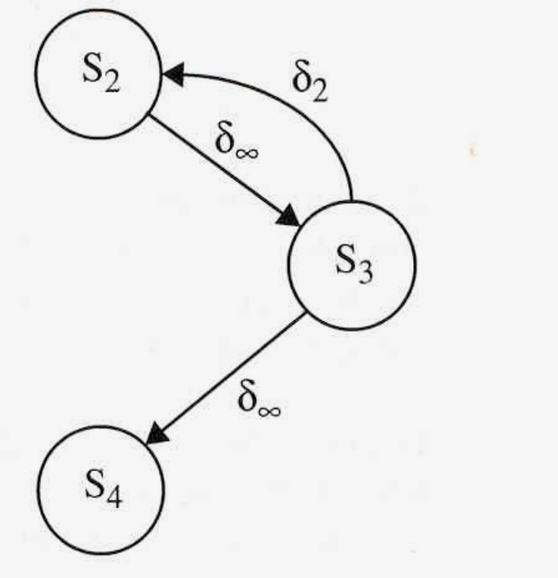
```
DO I = 1, 100
S1        X(I) = Y(I) + 10
 DO J = 1, 100
S2          B(J) = A(J,N)
          DO K = 1, 100
S3            A(J+1,K)=B(J)+C(J,K)
          ENDDO
S4          Y(I+J) = A(J+1, N)
 ENDDO
ENDDO
```

Advanced Vectorization Algorithm

- codegen $(\{S_2, S_3, S_4\}, 2\})$
- level-1 dependences are stripped off

```
DO I = 1, 100
  DO J = 1, 100
     codegen({S2, S3}, 3})
  ENDDO
S4  Y(I+1:I+100) = A(2:101,N)
ENDDO

X(1:100) = Y(1:100) + 10
```



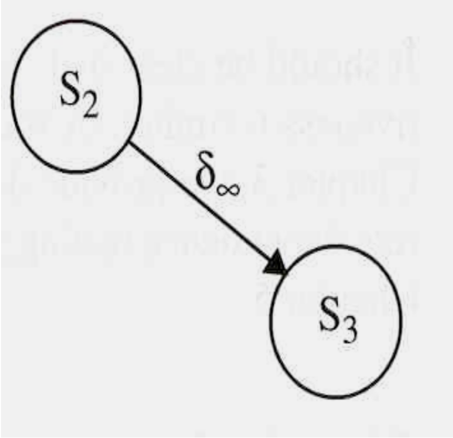
Advanced Vectorization Algorithm

- codegen ($\{S_2, S_3\}$, 3})
- level-2 dependences are stripped off

```
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
```

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


X(1:100) = Y(1:100) + 10
```



Enhancing Fine-Grained Parallelism

Chapter 5 of Allen and Kennedy

Fine-Grained Parallelism

Techniques to enhance fine-grained parallelism:

- Loop Interchange
- Scalar Expansion
- Scalar Renaming
- Array Renaming

Loop Shifting (Permutation)

- **Motivation: Identify loops which can be moved and interchange them to "optimal" nesting levels**

- Theorem 5.3 **In a perfect loop nest, if loops at level** `i, i+1,...,i+n` **carry no dependence, it is always legal to shift these loops inside of loop** `i+n+1`. **Furthermore, these loops will not carry any dependences in their new position.**

Loop Shifting

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

I J K
(=, =, <)

- **S has true, anti and output dependences on itself, hence codegen will fail as recurrence exists at innermost level**
- **Use loop shifting to shift loops I and J inside loop K:**

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

Loop Shifting

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

K I J
(<, =, =)

codegen vectorizes to:

```
 DO K = 1, N

     A(1:N,1:N) = A(1:N,1:N) + SPREAD(B(1:N,K),2)*SPREAD(C(K,1:N),1)

 ENDDO
```

Loop Selection

- Loop Shifting doesn't always find the best loop to move. Consider:

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

- Direction matrix: $\begin{pmatrix} < & < \\ = & < \end{pmatrix}$

- Loop shifting algorithm will fail to uncover vector loops; however, interchanging the loops can lead to:

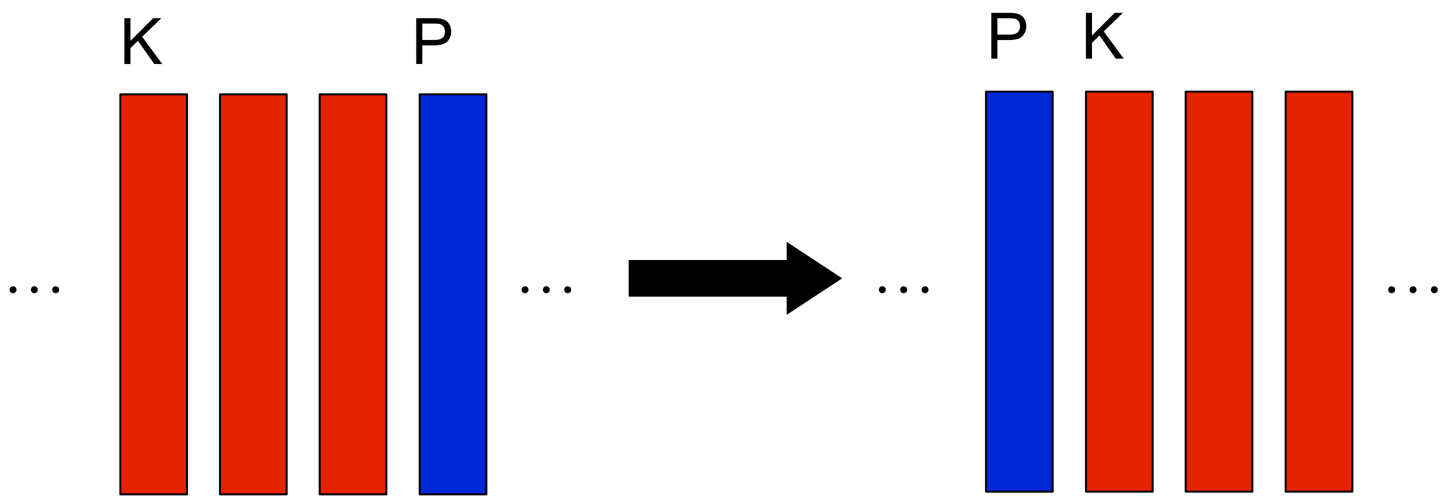
```
    DO J = 1, M
       A(2:N+1,J+1) = A(1:N,J) + A(2:N+1,J)
    ENDDO
```

$\begin{pmatrix} < & < \\ < & = \end{pmatrix}$

- Need a more general algorithm

Loop Selection

- **Loop selection:**
  - Select a loop at nesting level $p \geq k$ that can be safely moved outward to level k and shift the loops at level k, k+1, …, p-1 inside it



Fully Permutable Loop Nest

- A contiguous set of $k \geq 1$ loops, $i_j, \ldots, i_{j+k-1}$ is fully permutable if all permutations of $i_j, \ldots, i_{j+k-1}$ are legal

- Data dependence test: Loops $i_j, \ldots, i_{j+k-1}$ are fully permutable if for each dependence vector $(d_1, \ldots, d_n)$ carried at levels j … j +k-1, each of $d_j, \ldots, d_{j+k-1}$ is non-negative

- Fundamental result (to be discussed later in course): a set of k fully permutable loops can be transformed using only Interchange, Reversal and Skewing transformations into an equivalent set of k loops where k-1 of the loops are parallel

Scalar Expansion and its use in Removing Anti and Output Dependences

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

- **Scalar Expansion:**

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

- **leads to:**

```
S1       T$(1:N) = A(1:N)
S2       A(1:N) = B(1:N)
S3       B(1:N) = T$(1:N)
         T = T$(N)
```

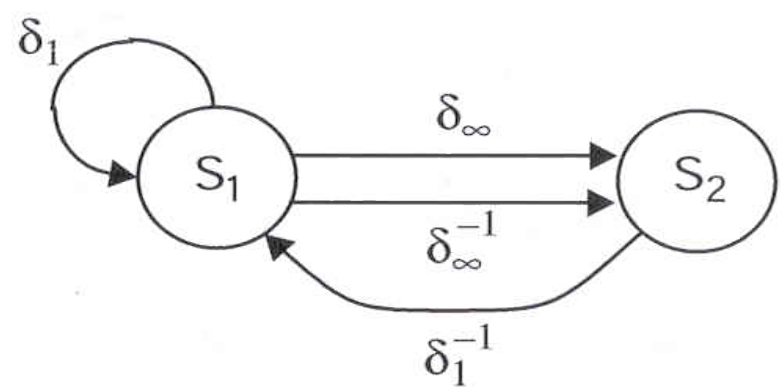
Scalar Expansion

- **However, not useful in removing true dependences. Consider:**

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

- **Scalar expansion gives us:**

```
    T$(0) = T
    DO I = 1, N
S1        T$(I) = T$(I-1) + A(I) + A(I+1)
S2        A(I) = T$(I)
    ENDDO
    T = T$(N)
```



Scalar Expansion: Safety

- Scalar expansion is always safe
- When is it useful?
  - Brute force approach: Expand all scalars, vectorize, shrink all unnecessary expansions.
  - However, we want to predict when expansion is useful i.e., when scalar expansion can enable a dependence edge to be deleted
- Dependences due to reuse of memory location vs. reuse of values
  - Dependences due to reuse of values must be preserved (true dependences)
  - Dependences due to reuse of memory location can be deleted by expansion (anti & output dependences)
    - This is also why functional languages are easier to parallelize, at the cost of increased memory overhead

Scalar Expansion: Covering Definitions

- A definition D 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 D.

```
    DO I = 1, 100
S1      T = X(I)
S2      Y(I) = T
    ENDDO
```

covering

```
    DO I = 1, 100
       IF (A(I) .GT. 0) THEN
S1         T = X(I)
S2         Y(I) = T
       ENDIF
    ENDDO
```

covering

Scalar Expansion: Covering Definitions

- **A covering definition does not always exist:**

```
     DO I = 1, 100
        IF (A(I) .GT. 0) THEN
S1             T = X(I)
        ENDIF
S2      Y(I) = T
     ENDDO
```

Scalar Expansion: Covering Definitions

- **We will consider a collection of covering definitions**

```
     DO I = 1, 100
          IF (X(I) .GT. 0) THEN
S1              T = X(I)
          ELSE
S2        T = -X(I)
          ENDIF
S3        Y(I) = T
     ENDDO
```

SSA-based definition

- **There is a collection *C* of** covering definitions **for T in a loop if either:**
  - There exists no ϕ-function at the beginning of the loop that merges versions of T from outside the loop with versions defined in the loop, or,
  - The ϕ-function within the loop has no SSA edge to any ϕ-function including itself

Scalar Expansion: Covering Definitions

- **Remember the loop which had no covering definition:**

```
     DO I = 1, 100
         IF (A(I) .GT. 0) THEN
S1           T = X(I)
         ENDIF
S2       Y(I) = T
     ENDDO
```

- **To form a collection of covering definitions, we can insert dummy assignments:**

```
     DO I = 1, 100
        IF (A(I) .GT. 0) THEN
S1           T = X(I)
        ELSE
S2           T = T
        ENDIF
S3      Y(I) = T
```

Scalar Expansion: SSA-based Algorithm

Given the collection of covering definitions, we can carry out scalar expansion for a normalized loop:

- Create an array T$ of appropriate length
- For each S in the covering definition collection C, replace the T on the left-hand side by **T$(I)**.
- For every other definition of T and every use of T in the loop body reachable by SSA edges that do not pass through S_0, the ϕ-function at the beginning of the loop, replace T by **T$(I)**.
- For every use prior to a covering definition (direct successors of S_0 in the SSA graph), replace T by **T$(I-1)**.
- If S_0 is not null, then insert **T$(0) = T** before the loop.
- If there is an SSA edge from any definition in the loop to a use outside the loop, insert **T = T$(U)** after the loop, were U is the loop upper bound.

Scalar Expansion: Covering Definitions

```
    DO I = 1, 100
        IF (A(I) .GT. 0) THEN
S1           T = X(I)
        ENDIF
S2      Y(I) = T
    ENDDO
```

After inserting covering definitions:

```
DO I = 1, 100
        IF (A(I) .GT. 0) THEN
S1           T = X(I)
        ELSE
S2           T = T
        ENDIF
S3      Y(I) = T
    ENDDO
```

After scalar expansion:

```
T$(0) = T
DO I = 1, 100
      IF (A(I) .GT. 0) THEN
S1          T$(I) = X(I)
      ELSE
S2          T$(I) = T$(I-1)
      ENDIF
S3    Y(I) = T$(I)
ENDDO
```

Deletable Dependences

- Uses of T before covering definitions are expanded as T$(I - 1)
- All other uses are expanded as T$(I)
- The deletable dependences are:
  - Backward carried antidependences
  - Backward carried output dependences
  - Forward carried output dependences
  - Loop-independent antidependences into the covering definition
  - Loop-carried true dependences from a covering definition to a use after the covering definition

Scalar Expansion: Drawbacks

- Expansion increases memory requirements
- Solutions:
  - Expand in a single loop
  - Strip mine loop before expansion
  - Forward substitution:

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

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

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 scalar T:**

```
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
```

Scalar Renaming

- **will lead to:**

S_3 `T2$(1:100) = D(1:100) - B(1:100)`

S_4 `A(2:101) = T2$(1:100) * T2$(1:100)`

S_1 `T1$(1:100) = A(1:100) + B(1:100)`

S_2 `C(1:100) = T1$(1:100) + T1$(1:100)`

`T = T2$(100)`

Scalar Renaming

- Renaming algorithm partitions all definitions and uses into equivalent classes, each of which can occupy different memory locations.
- Use the definition-use graph to:
  - Pick definition
  - Add all uses that the definition reaches to the equivalence class
  - Add all definitions that reach any of the uses…
  - ..until fixed point is reached
- Example:

```
     IF (…) THEN
S1       T = …
     ELSE
S2       T = …
     ENDIF
S3   … = T
S4     T = …
S5   … = T
```

```
IF (…) THEN
     T1 = …
ELSE
     T1 = …
ENDIF
… = T1
T2 = …
… = T2
```

Scalar Renaming: Profitability

- Scalar renaming will break recurrences in which a loop-independent output dependence or anti-dependence is a critical element of a cycle
- Relatively cheap to use scalar renaming
- Usually done by compilers when calculating live ranges for register allocation

Array Renaming

```
  DO I = 1, N
S1      A(I) = A(I-1) + X
S2      Y(I) = A(I) + Z
S3      A(I) = B(I) + C
  ENDDO
```

- $S_1\ \delta_\infty\ S_2$ $S_2\ \delta_\infty^{-1}\ S_3$ $S_3\ \delta_1\ S_1$ $S_1\ \delta_\infty^{0}\ S_3$

- **Rename *A(I)* to *A'(I)*:**

```
  DO I = 1, N
S1      A'(I) = A(I-1) + X
S2      Y(I) = A'(I) + Z
S3      A(I) = B(I) + C
  ENDDO
```

- **Dependences remaining:** $S_1\ \delta_\infty\ S_2$ **and** $S_3\ \delta_1\ S_1$

Array Renaming: Profitability

- Examining dependence graph and determining minimum set of critical edges to break a recurrence is NP-complete!
- Solution: determine edges that are removed by array renaming and analyze effects on dependence graph
- procedure array_partition:
  - Assumes no control flow in loop body
  - Identifies collections of references to arrays which refer to the same value
  - Identifies deletable output dependences and antidependences
- Use this procedure to generate code
  - Minimize amount of copying back to the "original" array at the beginning and the end

Homework #3 (Written Assignment)

1. **Solve exercise 3.6 in book**
   - This is case 4 of Lemma 3.3
   - Read Definitions 3.1, 3.2, 3.3 and Lemmas 3.1, 3.2, 3.3 before starting

- **Due in class on Tuesday, Oct 8th**
- Honor Code Policy: All submitted homeworks are expected to be the result of your individual effort. You are free to discuss course material and approaches to problems with your other classmates, the teaching assistants and the professor, but you should never misrepresent someone else's work as your own. If you use any material from external sources, you must provide proper attribution.