Enhancing Fine-Grained Parallelism

Loop vectorization,

Loop distribution,

Scalar expansion

Scalar and array renaming

Fine-Grained Parallelism

- Theorem 2.8. A sequential loop can be converted to a parallel loop if the loop carries no dependence.
- Fine-grained parallelism (vectorization)
  - Want to convert loops like:

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

    to `X(1:N) = X(1:N) + C` (Fortran 77 to Fortran 90)
  - However:

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

    is not equivalent to `X(2:N+1) = X(1:N) + C`
- Techniques to enhance fine-grained parallelism
  - Goal: make more inside loops parallelizable
  - Transform loops: Loop distribution, loop interchange
  - Transform data: scalar Expansion, scalar and array renaming

Loop Distribution

- Can dependence-carrying loops be vectorized?

```
  D0 I = 1, N
S1   A(I+1) = B(I) + C
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
```

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

- Safety of loop distribution
  - There must be no dependence cycle connecting statements in different loops after distribution

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

Loop Interchange

- Most statements are surrounds by more than one loops

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

  - Dependence from S1 to itself carried by outer loop
  - Inner loop can be parallelized

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

- Loop interchange: change the nesting order of loops

Applying Loop Distribution

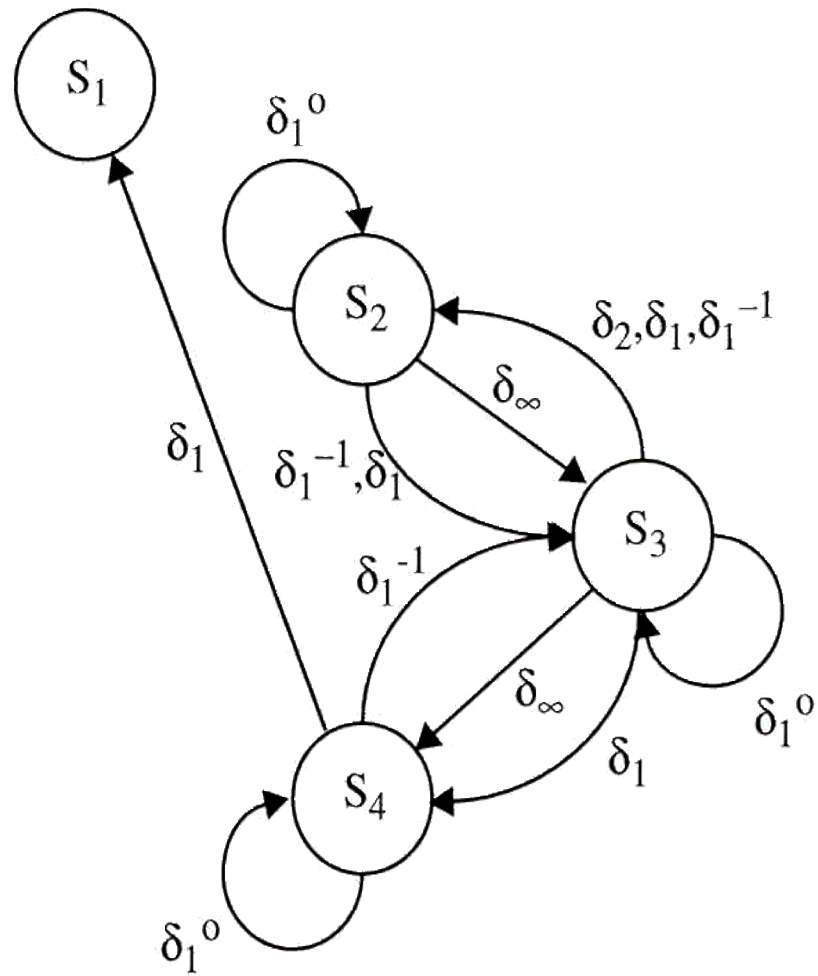
- procedure codegen(R, k, D);

  R:code to transform; k: the loop level to optimize;
  D:dependence graph for R

  - Find strongly-connected regions {S1, S2, ... , Sm} of D;
  - Rp = reduce each Si to a single node in R
    Dp = the dependence graph of Rp
  - For each node pi in topological order of nodes in Dp
    - Let Di be the dependence graph of pi at loop level k+1;
    - if Di is cyclic then
      - generate a level-k DO statement;
      - codegen (pi, k+1, Di);
      - generate the level-k ENDDO statement;
    - else
      - Try to vectorize inner loops in pi

Loop Distribution and Vectorization

```
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
```

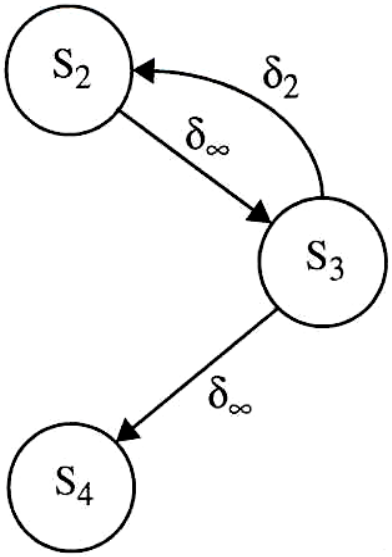


Loop Distribution and Vectorization

- *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
```



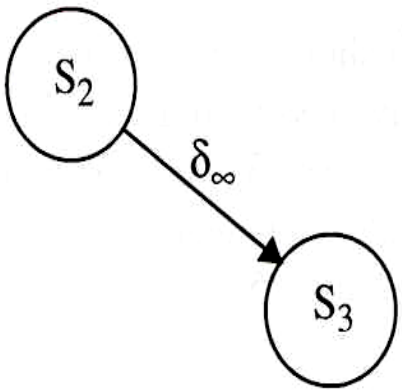
Loop Distribution and Vectorization

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

```
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
```

```
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
```



Loop Interchange

- A reordering transformation that
  - Changes the nesting order of loops
- Example

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

- Direction vector: (=, <)

  - After loop interchange

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

- Direction vector: (<, =)

  - Leads to

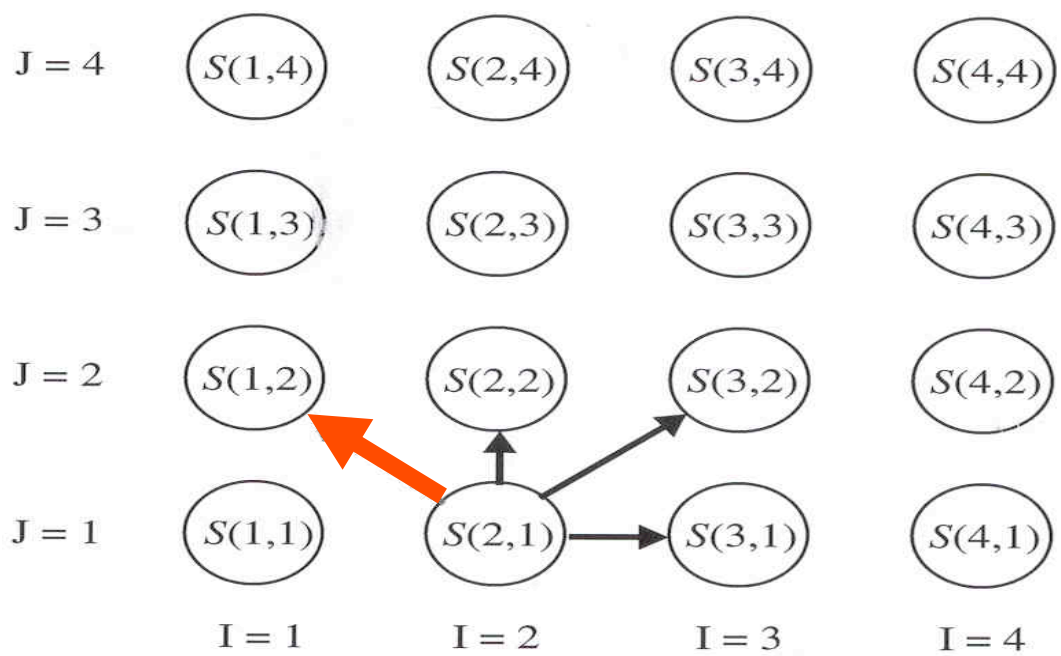
```
DO J = 1, M
S   A(1:N,J+1) = A(1:N,J) + B
ENDDO
```

Safety of Loop Interchange

- Not all loop interchanges are safe

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

Direction vector: (<, >)



Loop Interchange: Safety

- ***Direction matrix*** of a loop nest contains
  - A row for each dependence direction vector between statements contained in the nest.

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

  - The direction matrix for the loop nest is: $\begin{pmatrix} < & < & = \\ < & = & > \end{pmatrix}$
- **Theorem 5.2** A permutation of the loops in a perfect nest is legal if and only if
  - the direction matrix, after the same permutation is applied to its columns, has no ">" direction as the leftmost non-"=" direction in any row.

Loop Interchange: Profitability

- Profitability depends on architecture

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

- For SIMD machines with large number of FU's:

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

- For Vector machines: vectorize loops with stride-one access

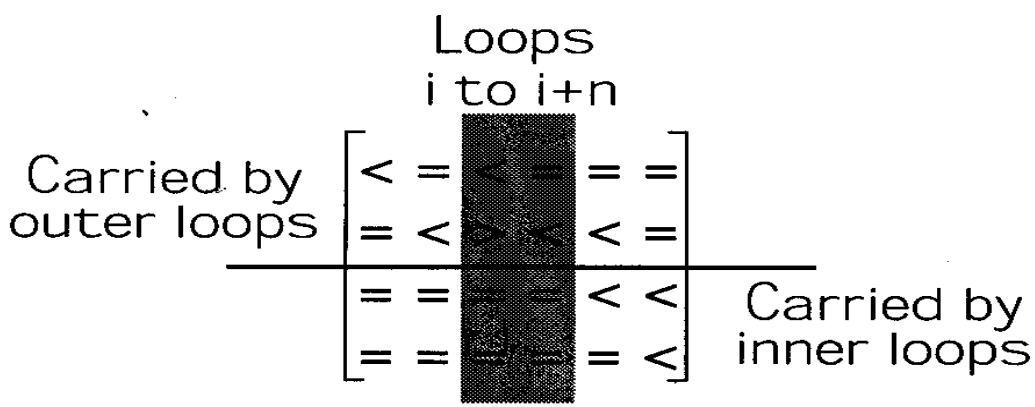
```
DO J = 1, M
   DO K = 1, L
S       A(2:N+1,J+1,K) = A(1:N,J,K) + B
```

- For MIMD machines with vector execution units: cut down synchronization costs

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

Loop Shifting

- Goal: move loops to "optimal" nesting levels
  - Apply loop interchange repeatedly when safe
- 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 Selection

- Consider:

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

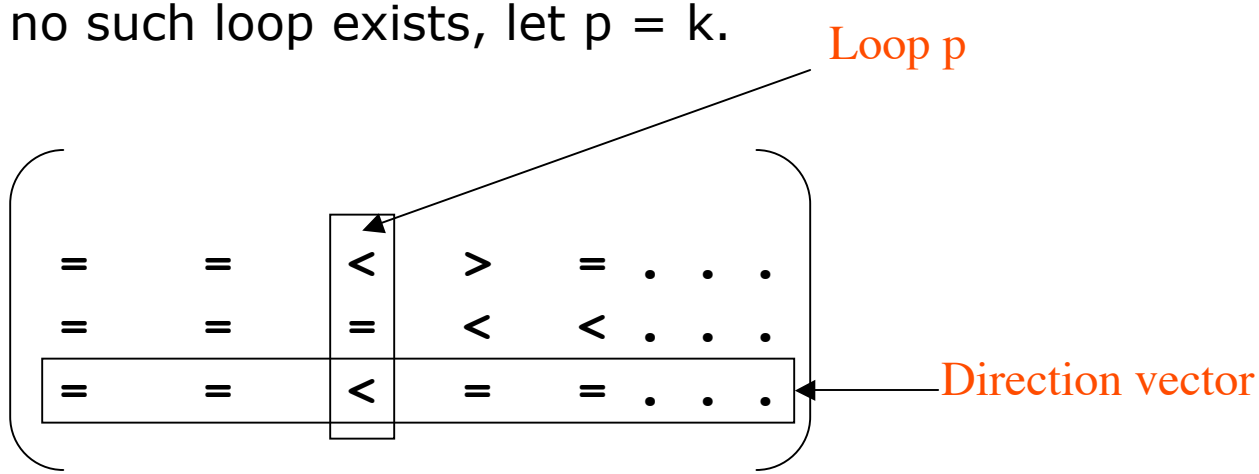
  - Direction matrix: $\begin{pmatrix} < & < \\ = & < \end{pmatrix}$
- Interchanging the loops can lead to:

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

- Which loop to shift?
  - 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

Heuristics for selecting loop level

- Goal: maximize # of parallel loops inside
  - If the level-k loop carries no dependence,
    - let p be the level of the outermost loop that carries a dependence
  - If the level-k loop carries a dependence,
    - let p be the outermost loop that can be safely shifted outward to position k and that carries a dependence direction vector d which has "=" in every position but the pth. If no such loop exists, let p = k.



Loop Shifting Example

```
      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)
```

- S has true, anti and output dependences on itself
  - Vectorization fails as recurrence exists at innermost level
- Use loop shifting to move K-loop to the outermost

```
      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)
```

- Parallelization is now possible

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

Vectorization with Loop Shifting

```
if p_i is cyclic then
    if k is the deepest loop in p_i
        then try_recurrence_breaking(p_i, D, k)
    else begin
        select_loop_and_interchange(p_i, D, k);
        generate a level-k DO statement;
        let D_i be the dependence graph consisting of
            all dependence edges in D that are at level
            k+1 or greater and are internal to p_i;
        codegen (p_i, k+1, D_i);
        generate the level-k ENDDO statement
    end
end
```

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)
```

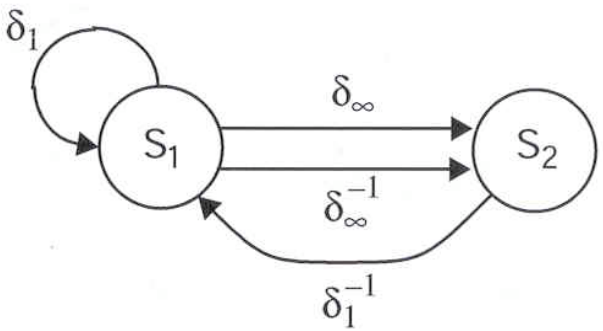
```
S1      T$(1:N) = A(1:N)
S2      A(1:N) = B(1:N)
S3      B(1:N) = T$(1:N)
        T = T$(N)
```

- Goal: remove anti-dependences inside loops
  - Use a different memory location (indexed by loop iterations) for each new value
  - Can eliminate dependence cycles inside loops
- Not profitable is scalar variables carry true dependences
  - Dependences due to reuse of values must be preserved

Profitability of Scalar Expansion

- 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)
```

  - Cannot eliminate the dependence cycle

Scalar Expansion: Tradeoffs

- 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
```

After strip-mining

```
DO I1 = 1, N, 10
   DO I=I1,I1+9
      T = A(I) + A(I+1)
      A(I) = T + B(I)
   ENDDO
 ENDDO
```

Scalar Expansion: Covering Definitions

- A definition S of variable x is a covering definition for loop L
  - If no other definition of x at the beginning of L can reach uses of x(S) in L
  - That is, if inside L, all uses of x reachable from S has a single definition S (can we apply forward expression substitution?)

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

covering

not covering

Scalar Expansion: Covering Definitions

- A single covering definition may not exist for a loop L
  - 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
      ENDDO
```

- To compute a set of covering definitions for variable x in L
  - Find the first definition S1 of x in L
  - Find all the paths that circumvent S1 to reach uses of x
  - Insert a dummy assignment for x in each of the path found

Scalar Expansion Using Covering Definitions

- Given a set C of covering definitions for variable T, assuming loop L has been normalized
  - Create an array T$ of appropriate length
  - For each S in the covering definition collection C,
    - replace T on the left-hand side by T$(I).
  - For every use of T in the loop body reachable by C
    - If the use is after C in the loop body, replace T by T$(I)
    - If the use is before C in the loop body, replace T by T$(I-1)
  - If definitions before the loop L can reach use of T in L, insert T$(0) = T before the loop L
  - If T is used after loop L, insert T=T$(U) after the loop, where 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
             T$(I) = T$(I-1)
       ENDIF
S2     Y(I) = T$(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
```

```
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)
```

- Goal: partition defs/uses into equivalent classes, each of which can occupy different memory locations:
  - Pick a definition S, add all uses that S reaches
  - Add all definitions that reach any of the uses…
  - ..until fixed point is reached
- Often 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
```

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

- 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
  - Analyze effects on dependence graph
- Algorithm (assumes no control flow in loop body)
  - Identify collections of array references which refer to the same value
  - Identify output and anti-dependences to eliminate
  - When renaming arrays, minimize amount of copying back to the "original" array at the beginning and the end

So Far...

- Uncovering potential vectorization in loops by
  - Loop Distribution
  - Loop Interchange
  - Scalar Expansion
  - Scalar and Array Renaming
- More transformations
  - Loop Skewing
  - Node Splitting
  - Recognition of Reductions
  - Index-Set Splitting
  - Run-time Symbolic Resolution
- Putting it together

Loop Skewing

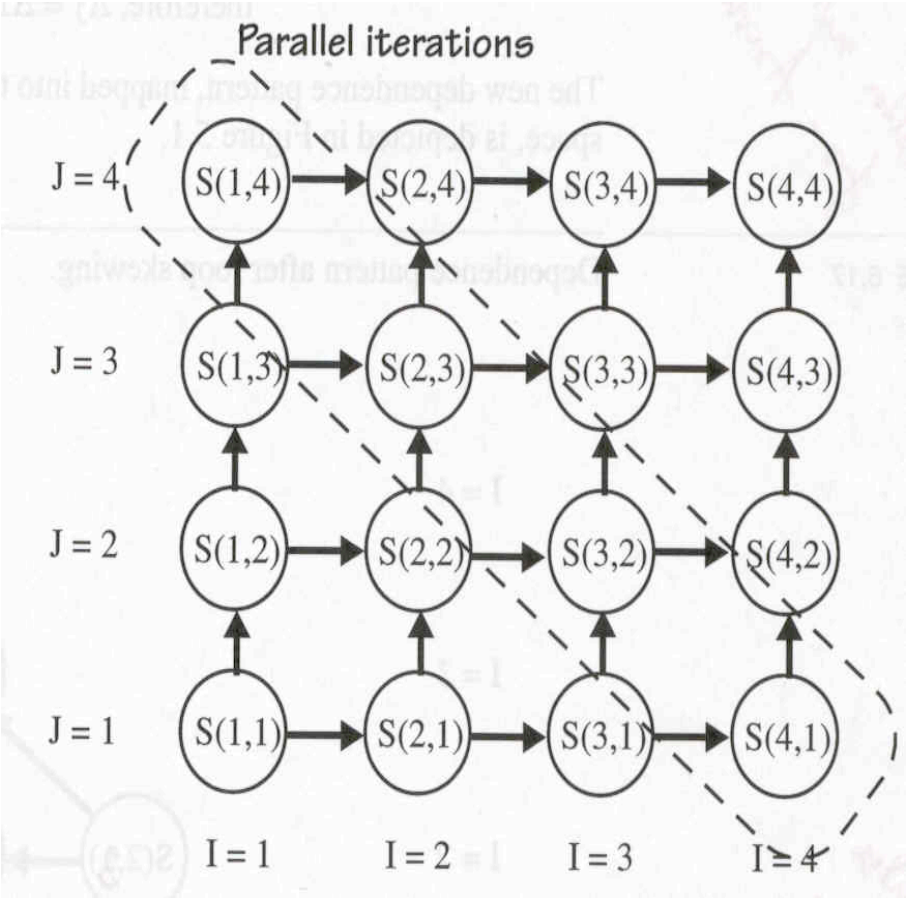
- Reshape Iteration Space to uncover parallelism

```
DO I = 1, N
   DO J = 1, N
      (=,<)
S: A(I,J)=A(I-1,J)+A(I,J-1)
      (<,=)
   ENDDO
ENDDO
```

  - Dependence Matrix

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

  - Parallelism not apparent



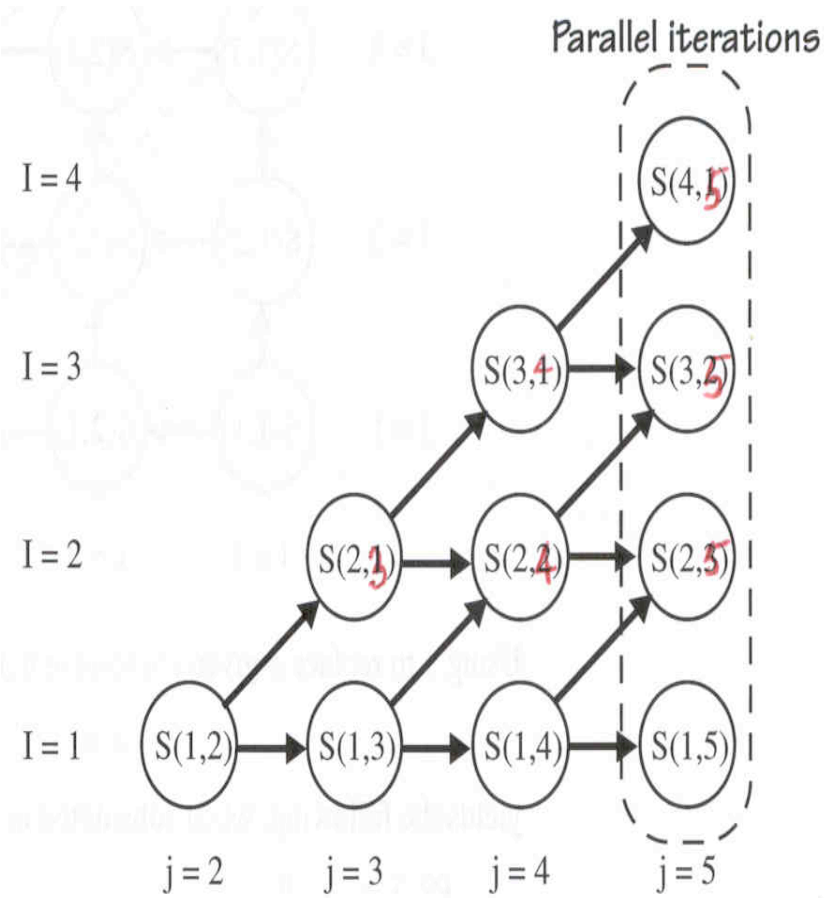
Loop Skewing Transformation

- Skew iterations of inner loop based on outer loop
  - J goes from I+1,I+N instead of 1,N

```
DO I = 1, N
   DO j = I+1, I+N
        (=,<)
S:  A(I,j-I)=A(I-1,j-I)+A(I,j-I-1)
        (<,<)
    ENDDO
ENDDO
```

- NOTE: dependence matrix changes

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$



Loop Skewing + Loop Interchange

```
DO I = 1, N
    DO j = I+1, I+N
S:    A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
    ENDDO
ENDDO
```

Loop interchange to..

```
DO j = 2, N+N
    DO I = max(1,j-N), min(N,j-1)
S:    A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
    ENDDO
ENDDO
```

Vectorize to..

```
DO j = 2, N+N
    FORALL I = max(1,j-N), min(N,j-1)
S:    A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
    END FORALL
ENDDO
```

- Disadvantages:
  - After interchange, inner loop evaluates different numbers of iterations
    - Outer loop needs twice as much number of iterations
    - Not profitable if N is small
  - If vector startup time is more than speedup time, this is not profitable
  - Vector bounds must be recomputed on each iteration of outer loop
- Apply loop skewing if everything else fails

Node Splitting

```
DO I = 1, N
S1: A(I) = X(I+1) + X(I)

S2: X(I+1) = B(I) + 32
ENDDO
```

- Recurrence kept intact by renaming algorithm
  - Antidependence and true dependence involving the same statement
- Make copy of the source data of antidependence
  - Anti-dependence now involves a different stmt
  - Goal: break dependence cycle

```
DO I = 1, N
S1':X$(I) = X(I+1)

S1: A(I) = X$(I) + X(I)

S2: X(I+1) = B(I) + 32
ENDDO
```

Vectorized to

```
X$(1:N) = X(2:N+1)
X(2:N+1) = B(1:N) + 32
A(1:N) = X$(1:N) + X(1:N)
```

Node Splitting

- Determining minimal set of critical antidependences is in NP-C
  - Perfect job of Node Splitting is difficult
- Heuristic:
  - Select antidependences
  - Delete it to see if acyclic
  - If acyclic, apply Node Splitting

Recognition of Reductions

- Reducing an array of values into a single value
  - Sum, min/max, count reductions

```
S = 0.0
DO I = 1, N
        S = S + A(I)
ENDDO
```

Not directly vectorizable

- Assuming commutativity and associativity

```
S = 0.0
DO k = 1, 4
  SUM(k) = 0.0
ENDDO
DO I = 1, N, 4
   SUM(1:3) = SUM(1:3) + A(I:I+3)
ENDDO
DO k = 1, 4
  S = S + SUM(k)
ENDDO
```

Useful for vector machines with 4 stage pipeline

Recognition of Reductions

- Reduction recognized by
  - Presence of self true, output and anti dependences
  - Absence of other true dependences

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

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

Index-set Splitting

- Subdivide loop into different iteration ranges to achieve partial parallelization
  - Loop Peeling [Weak Zero SIV]
  - Threshold Analysis [Strong SIV, Weak Crossing SIV]
  - Section Based Splitting [Variation of loop peeling]

- Loop Peeling
  - Source of dependence is a single iteration

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

Loop peeled to..

```
A(1) = A(1) + A(1)
DO I = 2, N
    A(I) = A(I) + A(1)
ENDDO
```

Vectorize to..

```
A(1) = A(1) + A(1)
A(2:N)= A(2:N) + A(1)
```

Threshold Analysis

- Threshold Analysis

```
DO I = 1, 100
    A(I+20) = A(I) + B
ENDDO
```

Strip mine to..

```
DO I = 1, 100, 20
   DO i = I, I+19
    A(i+20) = A(i) + B
   ENDDO
ENDDO
```

Vectorize to..

```
DO I = 1, 100, 20
  A(I+20:I+39) =
   A(I:I+19)+B
```

- Crossing thresholds

```
DO I = 1, 100
    A(100-I) = A(I) + B
ENDDO
```

Strip mine to..

```
DO I = 1, 100, 50
   DO i = I, I+49
    A(101-i) = A(i) + B
   ENDDO
ENDDO
```

Vectorize to..

```
DO I = 1, 100, 50
  A(101-I:51-I) = A(I:I+49)+B
ENDDO
```

Section-based Splitting

```
DO I = 1, N
   DO J = 1, N/2
S1:  B(J,I) = A(J,I) + C
   ENDDO
   DO J = 1, N
S2:  A(J,I+1) = B(J,I) + D
   ENDDO
ENDDO
```

- J Loop bound by recurrence due to B
- Only a portion of B is responsible for it

- Partition second loop into loop that uses result of S1 and loop that does not

```
DO I = 1, N
   DO J = 1, N/2
S1:  B(J,I) = A(J,I) + C
   ENDDO
   DO J = 1, N/2
S2:  A(J,I+1) = B(J,I) + D
   ENDDO
   DO J = N/2+1, N
S3:  A(J,I+1) = B(J,I) + D
   ENDDO
ENDDO
```

Run-time Symbolic Resolution

- Breaking conditions

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

Transformed to..

```
IF(L.LE.0) THEN
  A(L:N+L)=A(1:N)+B(1:N)
ELSE
  DO I = 1, N
   A(I+L) = A(I) + B(I)
  ENDDO
ENDIF
```

- Identifying minimum number of breaking conditions to break a recurrence is in NP-Complete
- Heuristic:
  - Identify when a critical dependence can be conditionally eliminated via a breaking condition

Putting It All Together

- Good Part
  - Many transformations imply more choices to exploit parallelism
- Bad Part
  - Choosing the right transformation
  - How to automate transformation selection?
  - Interference between transformations
- An effective optimization algorithm must
  - Take a global view of transformed code
  - Know the architecture of the target machine

- **Example of Interference**

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

Sum Reduction gives..

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

While Loop Interchange and Vectorization gives..

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

Performance on Benchmark

Vectorizing Compiler	Total			Dependence			Vectorization			Idioms			Completeness		
	V	P	N	V	P	N	V	P	N	V	P	N	V	P	N
PFC	70	6	24	17	0	7	25	4	5	5	0	10	23	2	2
Alliant FX/8, Fortran V4.0	68	5	27	19	0	5	20	5	9	10	0	5	19	0	8
Amdahl VP-E, Fortran 77	62	11	27	16	1	7	21	8	5	11	1	3	14	1	12
Ardent Titan-1	62	6	32	18	0	6	19	5	10	9	0	6	16	1	10
CDC Cyber 205, VAST-2	62	5	33	16	0	8	20	5	9	7	0	8	19	0	8
CDC Cyber 990E/995E	25	11	64	8	0	16	6	8	20	3	1	11	8	2	17
Convex C Series, FC 5.0	69	5	26	17	0	7	25	4	5	11	0	4	16	1	10
Cray series, CF77 V3.0	69	3	28	20	0	4	18	3	13	9	0	6	22	0	5
CRAX X-MP , CFT V1.15	50	1	49	16	0	8	12	1	21	10	0	5	12	0	15
Cray Series, CFT77 V3.0	50	1	49	17	0	7	8	1	25	7	0	8	18	0	9
CRAY-2, CFT2 V3.1a	27	1	72	5	0	19	3	1	30	8	0	7	11	0	16
ETA-10, FTN 77 V1.0	62	7	31	18	0	6	18	7	9	7	0	8	19	0	8
Gould NP1, GCF 2.0	60	7	33	14	0	10	19	7	8	8	0	7	19	0	8
Hitachi S-810/820,	67	4	29	14	0	10	24	4	6	14	0	1	15	0	12
IBM 3090/VF, VS Fortran	52	4	44	12	0	12	19	3	12	5	1	9	16	0	11
Intel iPSC/2-VX, VAST-2	56	8	36	15	0	9	17	8	9	6	0	9	18	0	9
NEC SX/2, F77/SX	66	5	29	17	0	7	21	5	8	12	0	3	16	0	11
SCS-40, CFT x13g	24	1	75	7	0	17	6	1	27	5	0	10	6	0	21
Stellar GS 1000, F77	48	11	41	14	0	10	20	9	5	4	1	10	10	1	16
Unisys ISP, UFTN 4.1.2	67	13	20	21	3	0	19	8	7	10	2	3	17	0	10