CSC2/455 Software Analysis and Improvement
Vectorization

Sreepathi Pai

URCS

March 25, 2019

Outline

Outline

Loop optimizations so far

- Important applications
  - Scientific computing
  - Audio/Video processing
  - Deep Learning
- Loop Dependences
  - True, anti- and output dependences
  - Must examine dynamic trace
  - Iteration spaces, vectors, lexicographic ordering
- Identifying loop dependences
  - Restrict array index functions to affine functions
  - Formulate dependence testing as an ILP
  - Dependence exists if solutions exist
  - ILP is NP-complete
- Today
  - Vectorization

Outline

Fortran 90 Vectorization

- If a loop contains a single statement
- And there is no loop-carried dependence
  - its iterations are independent of each other
- Then its iterations can be executed in parallel
  - "vectorization"

Example #1

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

can be vectorized as (Fortran-specific syntax)

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

Example #2

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

cannot be vectorized as (Fortran-specific syntax)

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

Fortran 90 semantics say that RHS uses original values.

- Serial code computes:
  - `X(2) = X(1) + C`
  - `X(3) = X(2) + C = X(1) + C + C`
- Vectorized code computes
  - `X(2) = X(1) + C`
  - `X(3) = X(2) + C`
  - i.e. updates on the LHS are not reflected in RHS until the *entire* statement has finished executing

Example #3

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

Note loop-carried dependence S1 δ S2

Can this be vectorized?

Example #3: Vectorized by Distribution

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

- Loop "distribution"

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

Example #4

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

- Which dependences exist?
- Can this loop be vectorized by distributing?

Outline

Simple Dependence Tests

Goal: Find dependences by examining indices.

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

Is there a read-after-write dependence from `A(I + 1)` in iteration I_0 to the read `A(I)` in a subsequent iteration?

$$I_0 + 1 = I_0 + \Delta I$$

What value of ΔI satisfies this equation?

True dependence testing

$$I_0 + 1 = I_0 + \Delta I$$

is satisfied by

- $\Delta I = 1$
- $1 > 0$ (later, so true dependence, i.e. read after write)
- $1 < N$ (will execute, assuming $N > 1$)
- $d_k(i) = 1$, so $D_k(i) = (<)$

Anti-dependence testing

Is there a write-after-read dependence from the read `A(I)` in iteration I_0 to the write `A(I + 1)` in a subsequent iteration?

$$I_0 + 1 + \Delta I = I_0$$

is satisfied by:

- $\Delta I = -1$
- $-1 < 0$, (earlier, no anti-dependence (i.e. write after read) found)

What if the write was `A(I - 1)`?

Multiple (Separable) Indices

```
DO J = 1, 100
   DO I = 1, 100
S1:       A(I+1) = A(I) + B(J)
   ENDDO
ENDDO
```

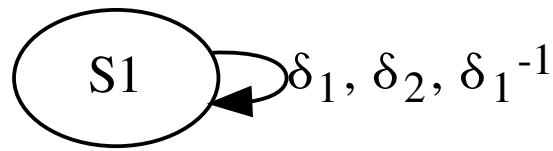
- True dependence for S1 in loop `I` is $<$
- Note that `J` does not appear in indices for `A`
  - But there is a dependence!

The * dependence direction

- Can't write equations for J though, so we assume "*" in direction vector
  - $(*, <)$
  - $(<, <), (=, <), (>, <)$
  - Level-1 (i.e. J-level) true dependence
  - Level-2 (i.e. I-level) true dependence
  - Level-1 anti-dependence

Dependence Graphs

- Nodes are statements
- Edges are dependences (from source to sink)
  - $\delta_k, \delta_k^{-1}, \delta_k^o$

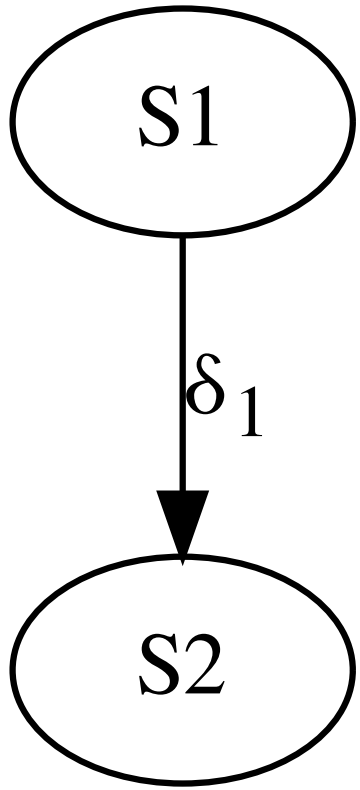


Ordering in a dependence graph

- Recall, for a moment, the data flow graph used in instruction scheduling of basic blocks
- How would you generate a linear order of instructions from the DAG that respected the dependences?

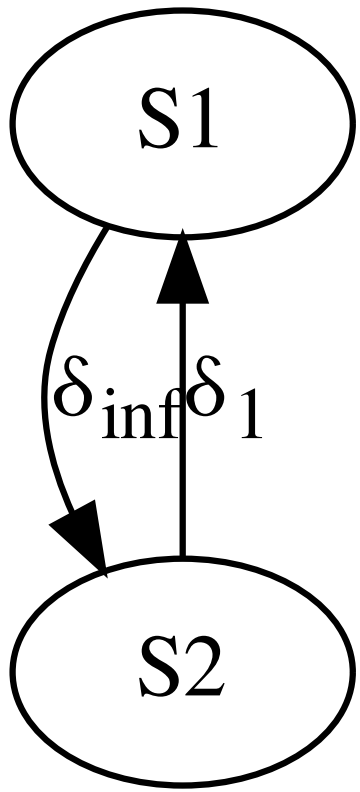
Vectorizable

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



Not Vectorizable

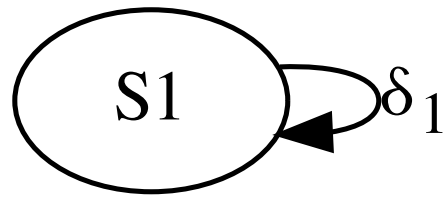
```
DO I = 1, N
S1:      B(I) = A(I) + E
S2:      A(I + 1) = B(I) + C
ENDDO
```



Outline

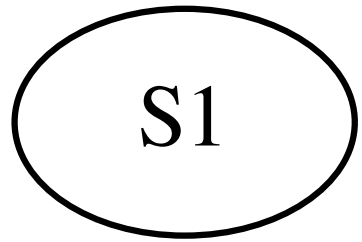
Will it vectorize? Example #5

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



Example #5: Vectorized at level 2

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



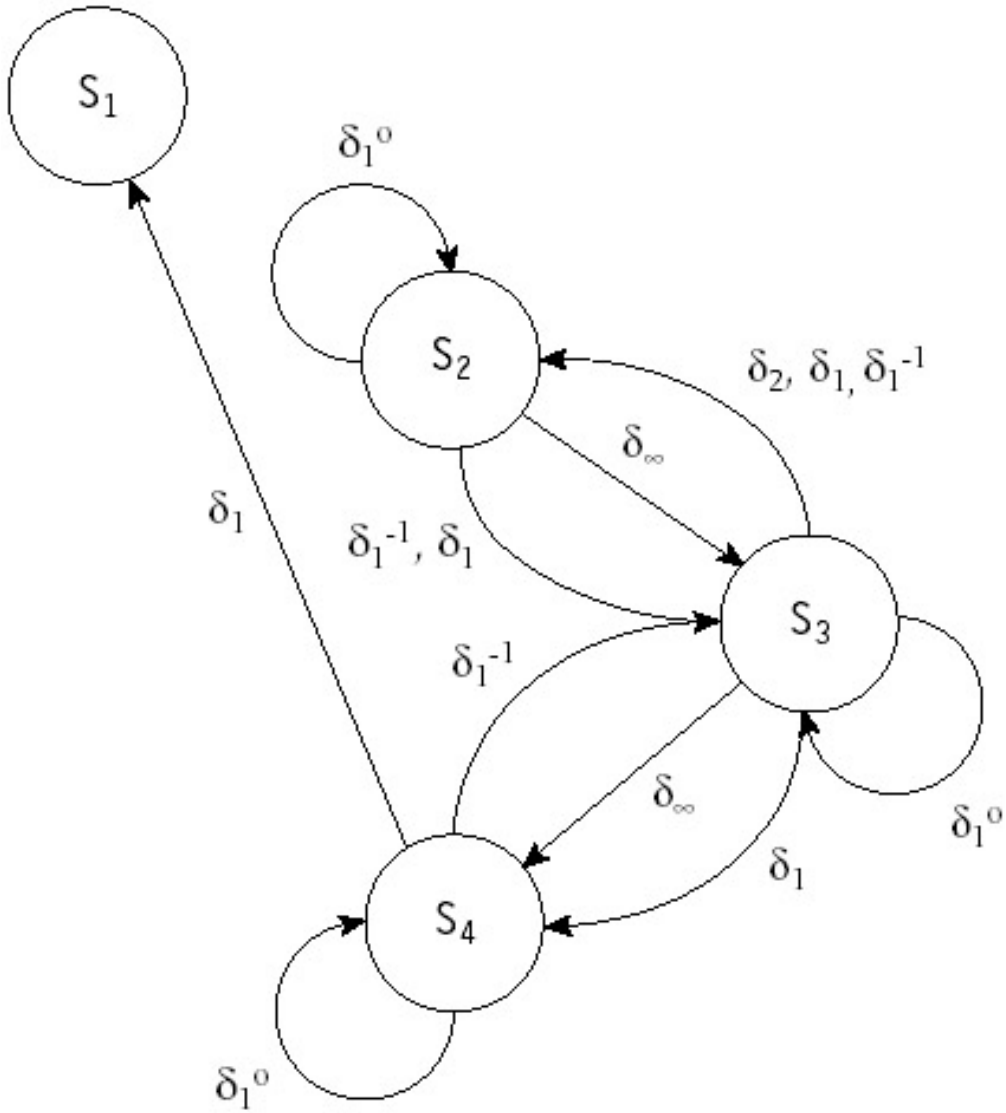
Final Example

```
  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
```

Step #1: Build Dependence Graph D

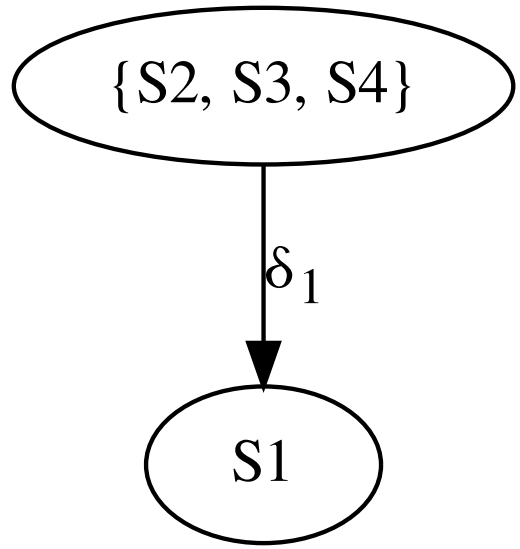


Step #2: Find Strongly Connected Components in D

- SCCs isolate cyclic regions
- Use Tarjan's algorithm
- Yields SCCs S_i

Step #3: Construct R_π

- Construct a graph R_π, where each node π_i corresponds to a SCC S_i
  - S_i is a SCC in D
- Connect nodes π_i using induced dependence graph D_π
  - I.e., if there was an edge between a node in S_i and a node in S_j, induce an edge between π_i and π_j



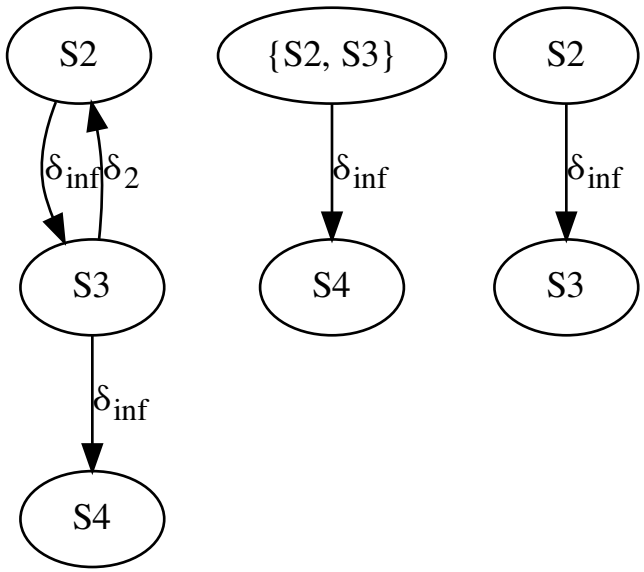
Step #4: Toposort R_π

- R_π is now a DAG
- Order nodes π_i of graph R_π using topological sort

Step #5: Recurse into π_i (if π_i is cyclic)

- If a node π_i is cyclic
  - Loop at this level must be executed serially
- However, inner loops may be vectorizable, so
  - Generate a new dependence graph with only dependences for inner levels
  - Recurse into this graph, starting from Step #1

Algorithm codegen (Figure 2.2) in Allen and Kennedy.



Step #6: Vectorize each node π_i in R_π (if possible)

- Process nodes π_j in topological order
- Is π_j acyclic?
  - Vectorize!
- Substitute all loop indices in inner dimensions with vectors

Result

```
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
```

Next steps

- More elaborate dependence testing
- Loop transformations
  - Improve locality
  - Improve parallelism

Outline

References

- Allen and Kennedy, Chapter 2, Sections 2.3 and 2.4