RUTGERS

THE STATE UNIVERSITY OF NEW JERSEY

CS 516 Compilers and Programming Languages II

Parallel Computing-6 and hard compiler / runtime system problems

Annoucements

Project 1 deadline: Wednesday, April 1.
Do you need another extension?

Homework #3 is due Friday, April 3.
Do you need an extension?

First paper presentation starting second week in April
- what's next
- polyhedral parallelization (available on sakai under resources)

Simple Vectorization Algorithm

Allen/Kennedy Algorithm

procedure *vectorize* (L, D)

// L is the maximal loop nest containing the statements.
// D is the dependence graph for statements in L.

find the partition p of set $\{S_1, S_2, \ldots, S_m\}$ of maximal strongly-connected regions in the dependence graph D restricted to L (for example, using Tarjan's algorithm);

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 (excluding loop-carried anti-dependence cycle) then
generate a DO-loop around the statements in p_i;

else
directly rewrite p_i in vector notation, vectorizing it with respect to every loop containing it;

end

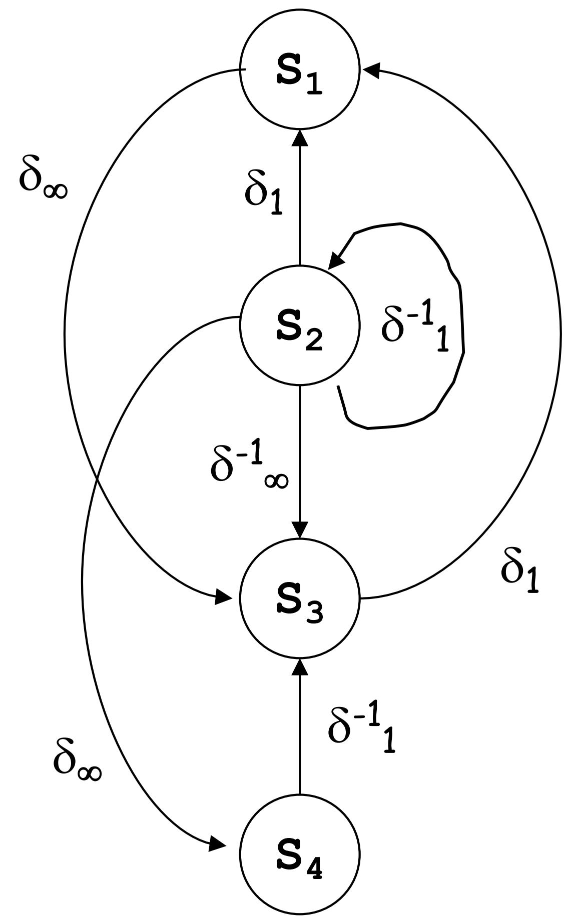
end *vectorize*

Simple Vectorization Algorithm

Statement level dependence graph D:

Example Loop L:

```
DO I = 2, 100
S1    A(I) = B(I-1) + C(I-1)+ 1
S2    B(I) = C(I)* B(I+1)
S3    C(I) = A(I) + 5
S4    D(I) = B(I)+ C(I+1)
ENDDO
```



δ_k - level k true dependence
δ^o_k - level k output dependence
δ^{-1}_k - level k anti dependence

Loop independent: $k = \infty$

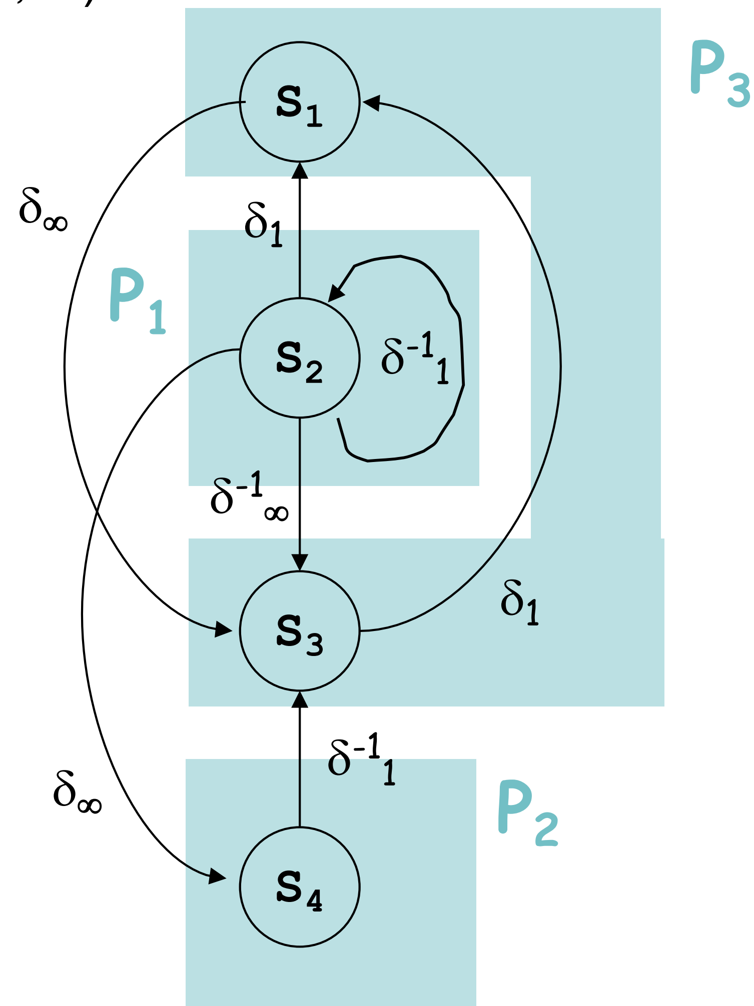
Simple Vectorization Algorithm

Statement level dependence graph D:

Example Loop L:

vectorize (*L*, *D*)

```
DO I = 2, 100
S1    A(I) = B(I-1) + C(I-1)+ 1
S2    B(I) = C(I)* B(I+1)
S3    C(I) = A(I) + 5
S4    D(I) = B(I)+ C(I+1)
ENDDO
```



δ_k - level k true dependence
δ^o_k - level k output dependence
δ^{-1}_k - level k anti dependence

Loop independent: $k = \infty$

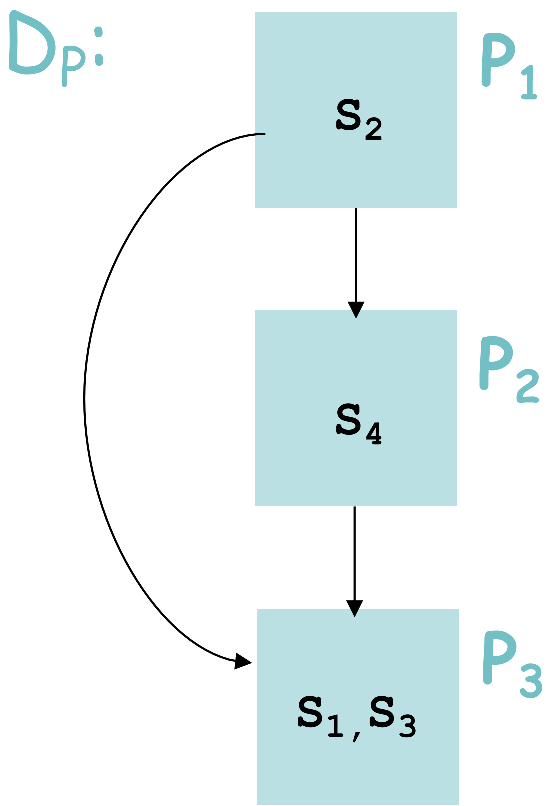
Simple Vectorization Algorithm

Statement level dependence graph D:

Example Loop L:

vectorize (*L*, *D*)

```
DO I = 2, 100
S1    A(I) = B(I-1) + C(I-1)+ 1
S2    B(I) = C(I)* B(I+1)
S3    C(I) = A(I) + 5
S4    D(I) = B(I)+ C(I+1)
ENDDO
```



D_P has to be acyclic

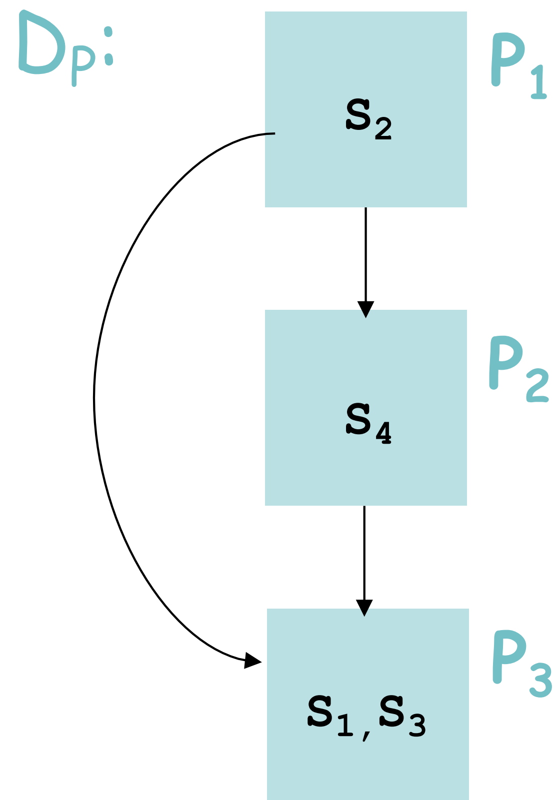
Simple Vectorization Algorithm

Statement level dependence graph D:

Example Loop L:

vectorize (*L*, *D*)

```
DO I = 2, 100
S1    A(I) = B(I-1) + C(I-1)+ 1
S2    B(I) = C(I)* B(I+1)
S3    C(I) = A(I) + 5
S4    D(I) = B(I)+ C(I+1)
ENDDO
```



In topological order:

```
S2  B(2:100) = C(2:100)* B(3:101)
S4  D(2:200) = B(2:100)+ C(3:101)
DO I = 2, 100
S1      A(I) = B(I-1) + C(I-1)+ 1
S3.     C(I) = A(I) + 5
ENDDO
```

D_P has to be acyclic

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 other loop!

Can be converted to:

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

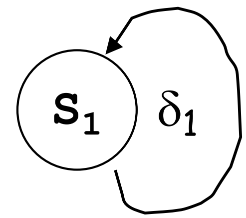
The simple algorithm does not capitalize on such opportunities. Once it sees a recurrence (dependence cycle), it gives up.

Note one exception: loop carried anti-dependence cycle

Problems With Simple Vectorization

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

Dependence graph



Observation: A level k dependence (dependence carried at level k) is satisfied if the k-level loop is executed sequentially.

Idea:

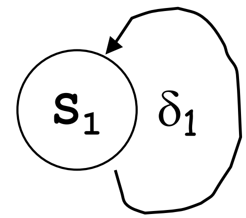
- Start from outermost level 1, apply the simple vectorization algorithm to level k, and if a strongly-connected region has a recurrence cycle, generate a sequential loop for level k for that region;
- Remove all level k dependences, and call the simple vectorization algorithm recursively for the region with only level k+1 or greater dependences.
- If no remaining recurrence cycles, generate vector statement for the remaining innermost levels.

Problems With Simple Vectorization

k = 1

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

Dependence graph



Observation: A level k dependence (dependence carried at level k) is satisfied if the k-level loop is executed sequentially.

Idea:

- Start from outermost level 1, apply the simple vectorization algorithm to level k, and if a strongly-connected region has a recurrence cycle, generate a sequential loop for level k for that region;
- Remove all level k dependences, and call the simple vectorization algorithm recursively for the region with only level k+1 or greater dependences.
- If no remaining recurrence cycles, generate vector statement for the remaining innermost levels.

Problems With Simple Vectorization

Dependence graph

k = 2

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

S_1

Observation: A level k dependence (dependence carried at level k) is satisfied if the k-level loop is executed sequentially.

Idea:

- Start from outermost level 1, apply the simple vectorization algorithm to level k, and if a strongly-connected region has a recurrence cycle, generate a sequential loop for level k for that region;
- Remove all level k dependences, and call the simple vectorization algorithm recursively for the region with only level k+1 or greater dependences.
- If no remaining recurrence cycles, generate vector statement for the remaining innermost levels.

Advanced Vectorization Algorithm

Allen/Kennedy 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 $\{S_1, S_2, \ldots, S_m\}$ of maximal strongly-connected regions in the dependence graph D restricted to R;

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

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

for i = 1 to m do begin

    if p_i is cyclic then begin

        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
    else

        generate a vector statement for p_i in r(p_i)-k+1 dimensions, where r (p_i) is the number of loops containing p_i;

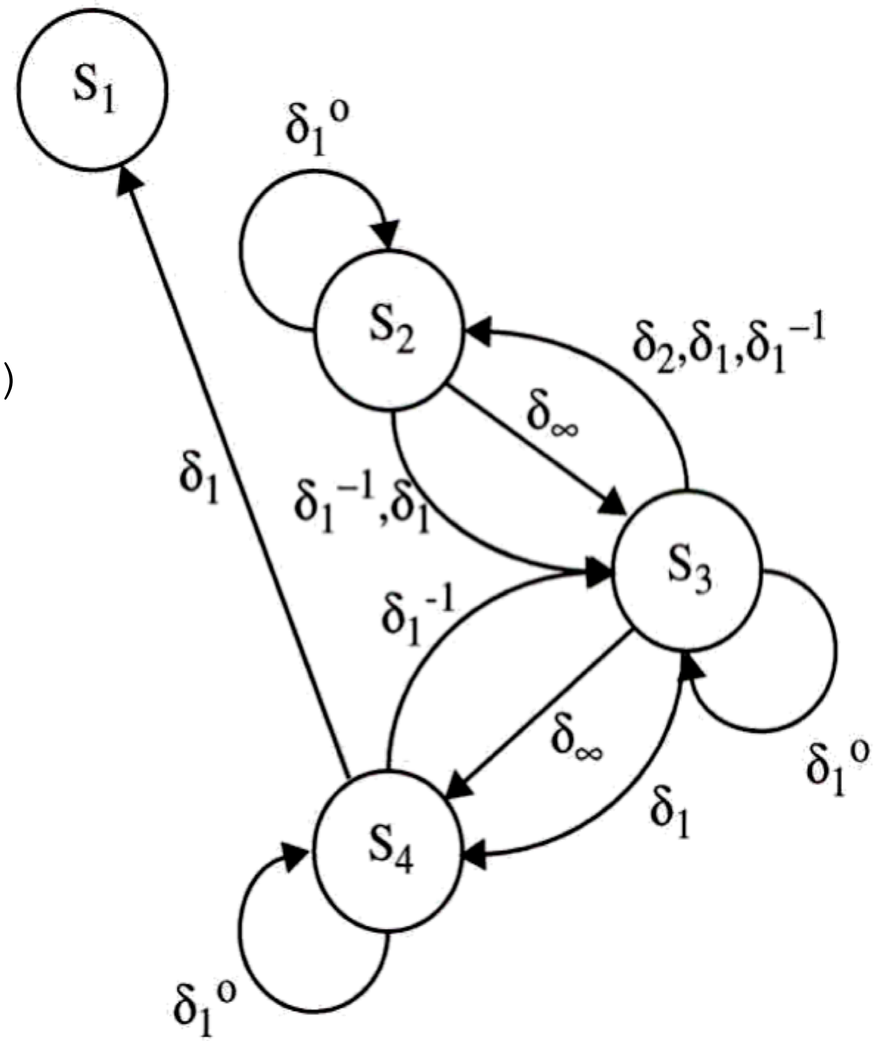
end

Homework problem

```
DO I = 2, 100
S1   D(I) = 100
     DO J = 1, 100
S2       B(I,J) = C(I-1,J+1) + 5
         DO K = 1, 100
S3         A(I,J,K)= A(I-1,J,K+1) + B(I,J+1) * 2
         ENDDO
S4       C(I,J) = D(I+1)* B(I,J)
     ENDDO
S5   E(I) = D(I) + 2
ENDDO
```

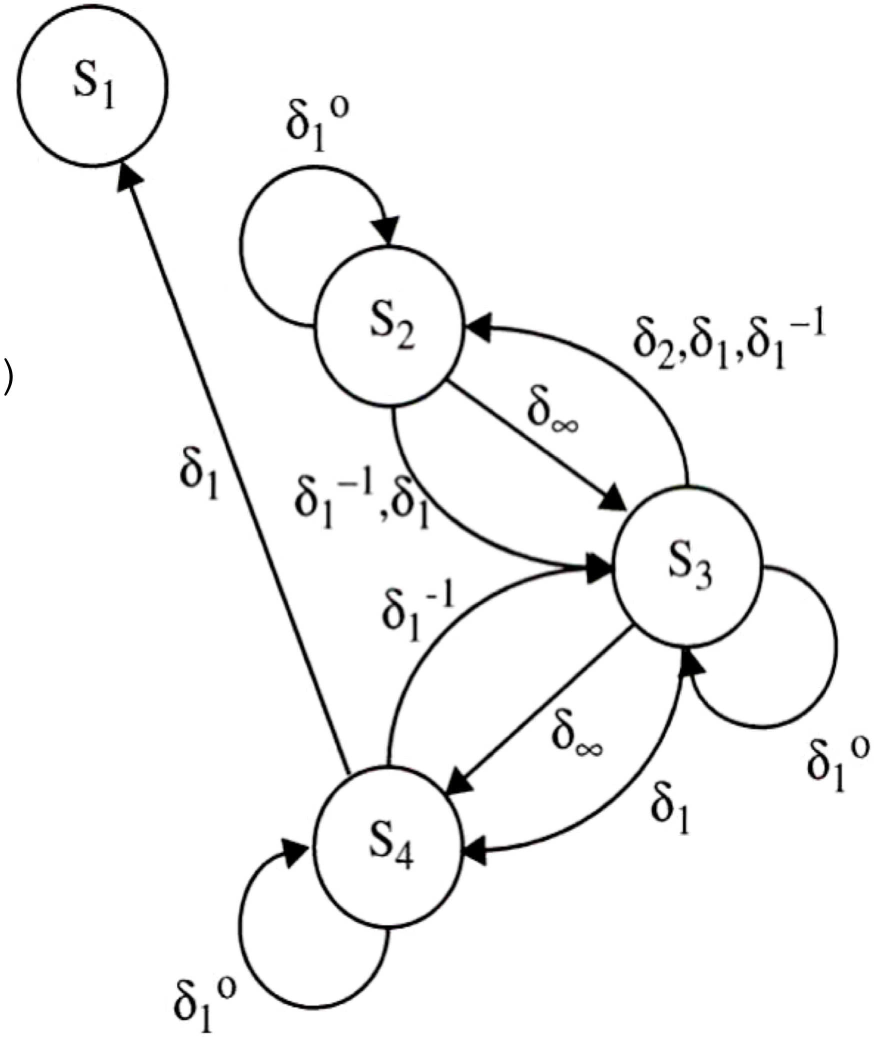
Advanced Vectorization Algorithm

```
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
```



Advanced Vectorization Algorithm

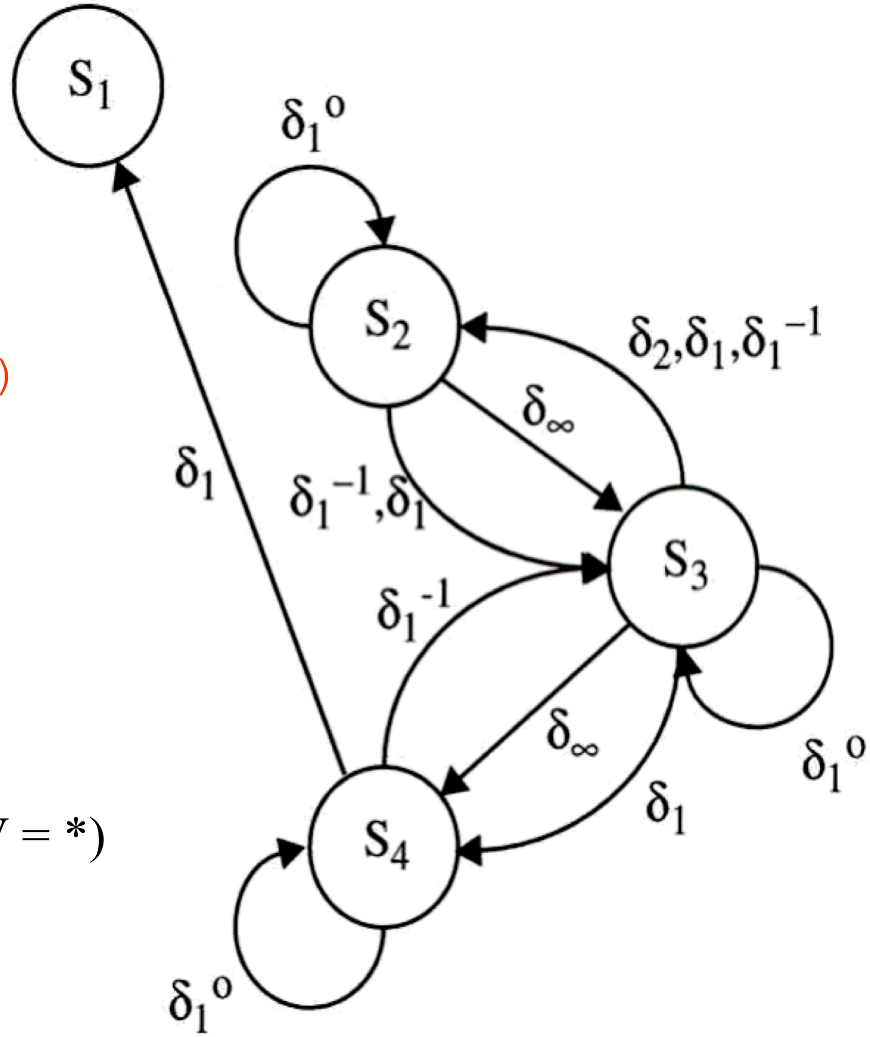
```
DO I = 1, 100
S1    X(I) = Y(I) + 10
      DO J = 1, 100
S2       B(J) = A(J,N)
         DO K = 1, 100
S3          A(J+1,K)=B(J)+C(J,K)
         ENDDO
S4       Y(I+J) = A(J+1, N)
      ENDDO
ENDDO
```



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

Initial call to vectorizer:

codegen ({S_1, S_2, S_3, S_4}, 1})

$\Rightarrow$ S_1 will be vectorized

```
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
   codegen({S2, S3, S4}, 2, D2})
ENDDO

S1 X(1:100) = Y(1:100) + 10
```

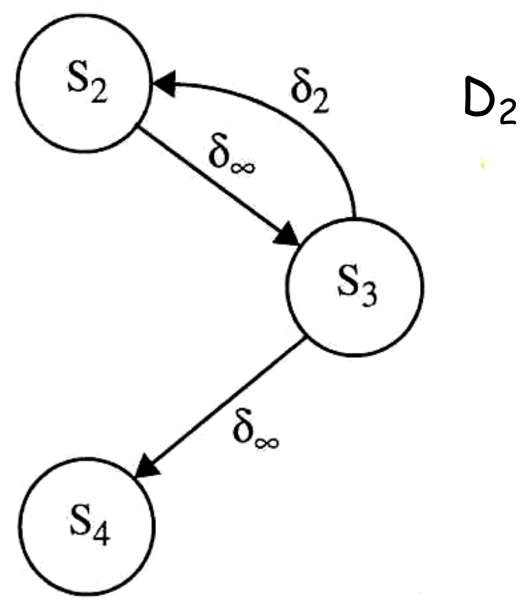
Advanced Vectorization Algorithm

- *codegen* ({S_2, S_3, S_4}, 2})
- level-1 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
      codegen({S2, S3}, 3, D3})
   ENDDO
S4 Y(I+1:I+100) = A(2:101,N)
ENDDO

S1 X(1:100) = Y(1:100) + 10
```



Advanced Vectorization Algorithm

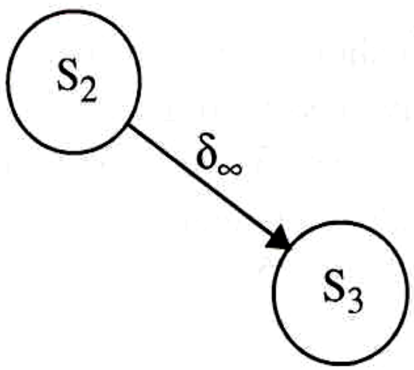
- *codegen* ({S_2, S_3}, 3})
- level-2 dependences are stripped off

```
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
   S2 B(J) = A(J,N)
   S3 A(J+1,1:100)=B(J)+C(J,1:100)
   ENDDO
S4 Y(I+1:I+100) = A(2:101,N)
ENDDO

S1 X(1:100) = Y(1:100) + 10
```

D3

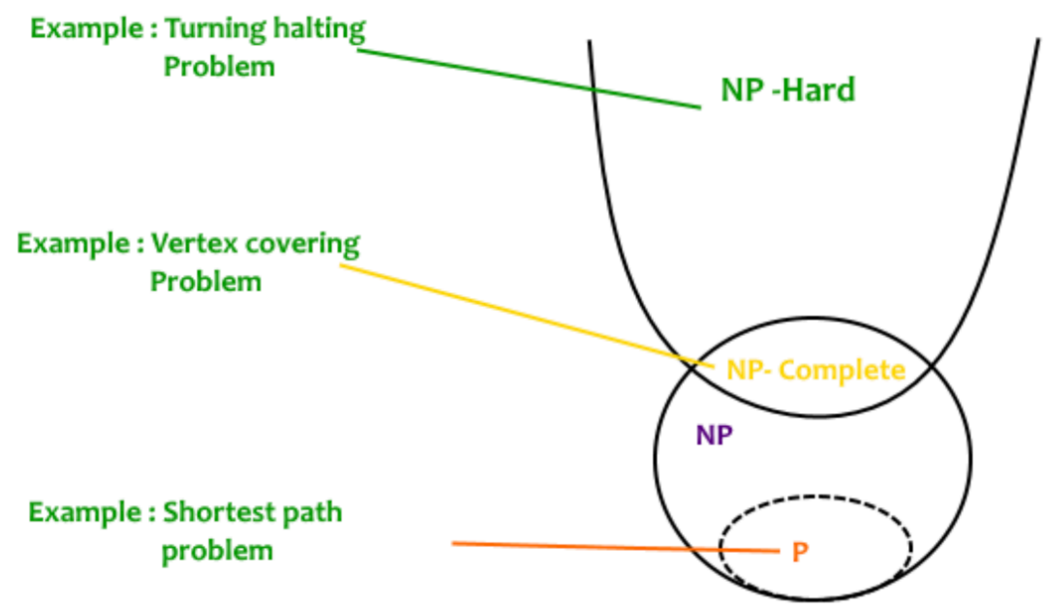


Hard compiler / runtime problems

What is a hard compiler problem?

Answer: Problems that are NP-complete (non-deterministic polynomial)

Definition: A problem X is NP-complete iff
(1) X is in NP, and
(2) Every problem Y in NP can be reduced in polynomial time to X



Here P != NP

Hard compiler problems (cont.)

How to prove that a (decision) problem X is in NP-complete?

(1) Show that you can verify in polynomial time that a given solution/witness of X is valid (polynomial time verification)
(2) Show a polynomial time reduction of any instance of a existing/known NP-complete problem to an instance of X

Example "classical" NP-complete problems
- 3 SAT (3 Conjunctive Normal Form Satiability Problem)
- Traveling salesman
- Graph coloring
- Integer programming

Example "compiler" NP-complete problems
- Register allocation
- Instruction scheduling
- Automatic data layout

Hard compiler problems (cont.)

Example proof outline (Kremer1993/1995)

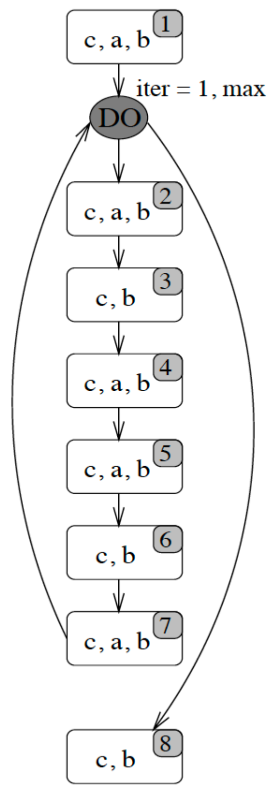
Theorem: Minimal cost dynamic data layout problem is NP-complete

Problem statement:

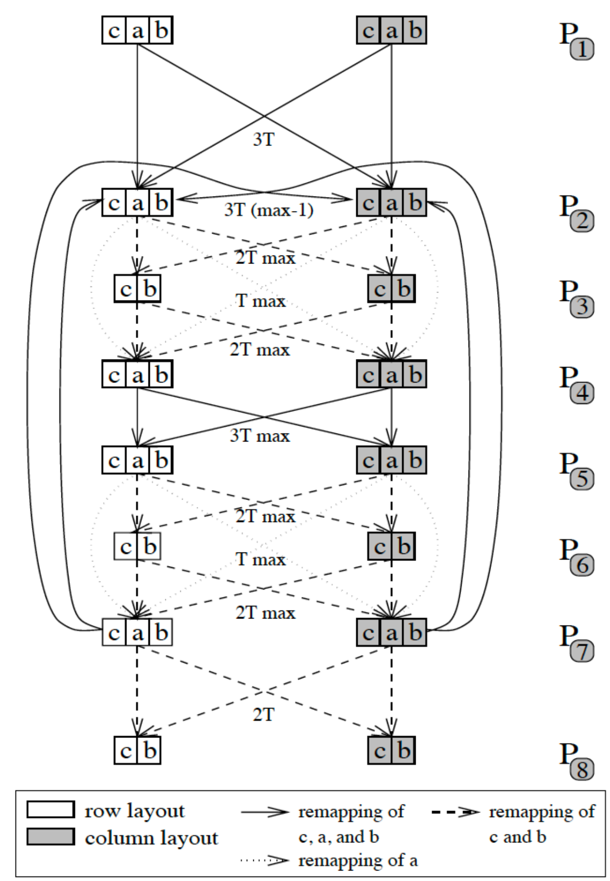
- Program consists of multiple phases
- Phases access multi-dimensional arrays
- Each array has a finite set of candidate layouts (by row, by column, blocked, …)
- Cost of array access in phase computation depends on array's layout
- Array remapping (e.g.: from row to column) may be performed between phases
- Remapping is not free, i.e., has a cost

Determine the layout of each array in each phase such that the overall computation and remapping costs are minimized.

8 phases, 3 arrays



2 candidate layouts per array



Hard compiler problems (cont.)

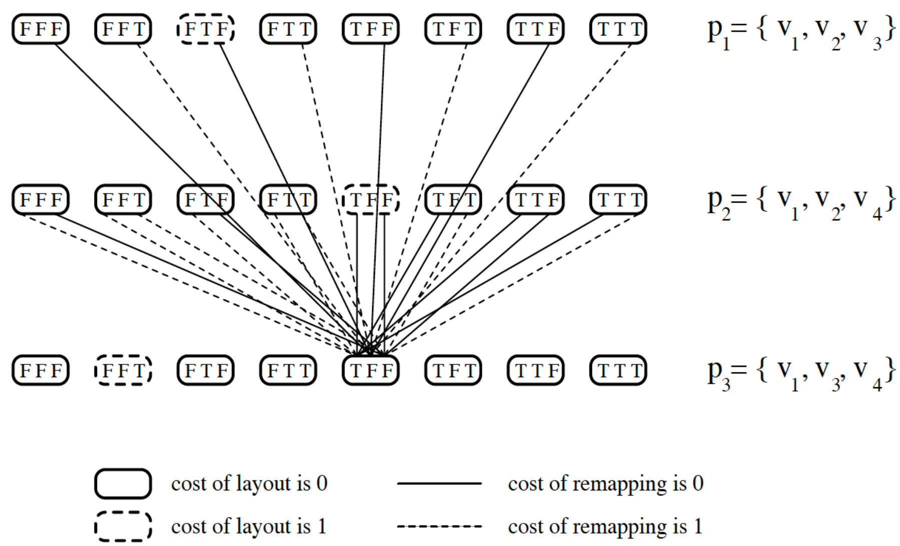
Proof Outline:
Ad (1): Given a layout for each array,
verify that it is smaller than a specific cost: just add up the phase costs with necessary remappings. This is polynomial time □

Ad (2): Reduce 3 SAT to the dynamic layout problem. Example polynomial time mapping:

Instance

$$(v_1 \vee \neg v_2 \vee v_3) \ \wedge \ (\neg v_1 \vee v_2 \vee v_4) \ \wedge \ (v_1 \vee v_3 \vee \neg v_4)$$

is mapped to a data layout problem with 3 phases, one for each term; each variable v_x has two possible layouts (true and false), resulting in 8 candidate layouts per phase; cost of a phase layout is 0 if corresponding term evaluates to true, otherwise 1; remapping for individual arrays/terms has cost of 1



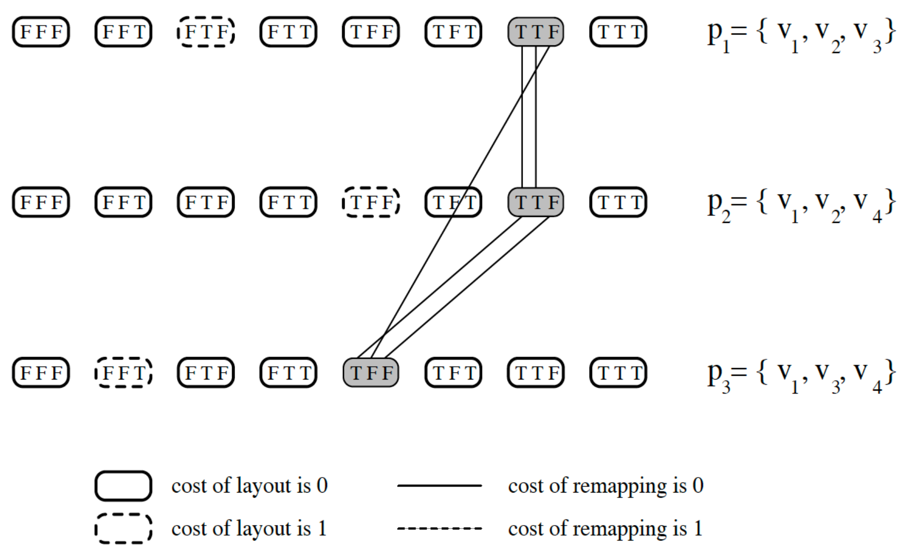
Hard compiler problems (cont.)

Proof Outline:
Ad (1): Given a layout for each array,
verify that it is smaller than a specific cost: just add up the phase costs with necessary remappings. This is polynomial time □

Ad (2): Reduce 3 SAT to the dynamic layout problem. Example polynomial time mapping:
Instance

$$(v_1 \vee \neg v_2 \vee v_3) \ \wedge \ (\neg v_1 \vee v_2 \vee v_4) \ \wedge \ (v_1 \vee v_3 \vee \neg v_4)$$

is satisfiable iff
There exists a data layout with 0 cost.



If 0 cost, no remapping (i.e., for each variable there is a fixed truth value assignment) and each phase has a 0 cost (i.e., true) term;
If the expression is satisfiable, there exists a truth value assignment such that no remapping is necessary and each phase has a 0 cost selected term candidate. □

Hard compiler problems (cont.)

"Any advanced / interesting compiler optimization problem is NP-complete" (Keith Cooper, Rice University)

So what to do?

Option 1 (current wisdom): **Use a heuristic**
This can work well since problem instances that occur in practice may have structure, i.e., exhibit properties that may not lead to exponential cost.

Option 2: Use state-of-the-art integer programming tools
This will produce the optimal solution (no computation approximation). If there is structure in the problem, the solver will exploit it. If the optimal solution takes too long to compute, return the best feasible solution within a specified time budget (heuristic solution, if needed).

WE WILL USE OPTION 2. $\Rightarrow$ **GUROBI** - MIXED INTEGER PROGRAMMING TOOL