CS314 Spring 2023
Homework 6
Sample Solution
Due Tuesday, April 25, 11:59pm

Problem 1 – Dependence Analysis using SIV test

Use the SIV test as discussed in class (lecture 22) to prove or disprove all possible dependencies for the following loop nest:

```
   do i = 3, 100
S1:  a(i) = a(i-1) + b(2*i+4)
S2:  b(2*i-2) = a(i)
   enddo
```

1. For each possible pair of array references and type of dependence (true, anti, output), show how to apply the SIV test by listing the values of a, c_1, and c_2), and the computed distance. Interprete the result of the SIV test based on the computed distance, i.e., whether a dependence exists or not.

   The list only includes pairs of references to the same array.

source reference	sink reference	a	c_1	c_2	note	type	distance vector
S1: a(i)	S1: a(i-1)	1	0	-1	$i < i'$	true	(1)
S1: a(i-1)	S1: a(i)	1	-1	0	$i < i'$	$\Delta d = -1$:no dep.	-
S1: a(i)	S2: a(i)	1	0	0	$i \leq i'$	true	(0)
S1: a(i)	S1: a(i)	1	0	0	$i < i'$	$\Delta d = 0$ no solution:no dep.	-
S2: a(i)	S1: a(i)	1	0	0	$i < i'$	$\Delta d = 0$ no solution:no dep.	-
S2: b(2*i-2)	S1: b(2*i+4)	2	-2	4	$i < i'$	$\Delta d = -3$: no dep.	-
S1: b(2*i+4)	S2: b(2*i-2)	2	4	-2	$i \leq i'$	anti	(3)
S2: b(2*i-2)	S2: b(2*i-2)	2	-2	-2	$i < i'$	$\Delta d = 0$ no solution:no dep.	-

   Note: Potential output dependencies on single statements have been included here for completeness, i.e., S1:a(i) and S2:b(2*i-2). If an SIV test is applicable, there cannot be any feasible solution for these cases.

2. Can the above loop be parallelized (doall parallel loop)? Justify your answer.

   The loop cannot be parallelized due to loop-carried dependencies.

Problem 2 – Dependence Analysis

List all all dependencies in the following loop nests. If a dependence exists, how far apart in term of iterations are the accesses to the same memory location? This is called the *distance* of a dependence. Also, state explicitly whether a dependence is a true, anti, or output dependence.

1.
   ```
   do i = 4, 5
     a(i) = a (i+2) + a(i-3)
   enddo
   ```

   No dependence since distance is larger than upper bound minus lower bound (5-4 = 1).

2.
   ```
   do i = 1, 100
     a(2*i) = a(2*i-1) + a(2*i+1)
   enddo
   ```

   No dependence since SIV test yields distance of d = 1/2.

3.
   ```
   do i = 1, 10
     a(i) = a(5) + 1
   enddo
   ```

source reference	sink reference	type	distance vector
a(i)	a(5)	true	-
a(5)	a(i)	anti	-

4.
   ```
   do i = 1, 10
     a(i) = a(2*i) + 1
   enddo
   ```

source reference	sink reference	type	distance vector
a(2*i)	a(i)	anti	-

5.
   ```
   do i = 1, 9
     a(10-i) = a(i) + 5
   enddo
   ```

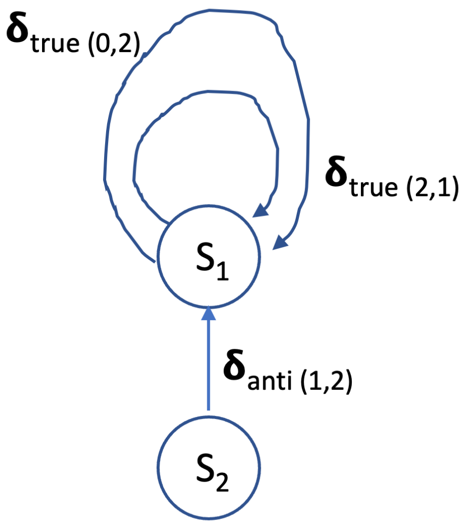
source reference	sink reference	type	distance vector
a(10-i)	a(i)	true	-
a(i)	a(10-i)	anti	-

Problem 3 – Statement level dependence graph and loop transformations

A statement-level dependence graph represents the dependences between statements in a loop nest. Nodes represent single statements, and edges dependences between statements. An edge is generated by a pair of array references that have a dependence. Edges are directed from the source of the dependence to its sink. For example, for a true dependence, the source is a write reference, and the sink is a read reference. There may be multiple edges (i.e., dependences) between two nodes in the graph.

```
do i = 1, n
  do j = 1, m
S1:  a(i,j) = a(i,j-2) + a(i-2,j-1)
S2:  b(i,j) = a(i+1,j+2) + b(i,j)
  enddo
enddo
```

1. Show the statement level dependence graph. Mark all dependencies with their type (true, anti, output) and their dependence distance.


$\delta_{\text{true }(0,2)}$
$\delta_{\text{true }(2,1)}$
S_1
$\delta_{\text{anti }(1,2)}$
S_2

2. Can you transform the above code using loop interchange and/or loop distribution to generate a program with the maximum (`doall-loop`) parallelism. Justify your answer, i.e., describe what you did and why.

   Yes, you can increase the exploitable parallelism by loop interchange. By executing the resulting outer loop sequentially, all dependencies are satisfied, leaving the inner loop parallel:

```
do j = 1, n
  doall  i = 1, m
S1:  a(i,j) = a(i,j-2) + a(i-2,j-1)
S2:  b(i,j) = a(i+1,j+2) + b(i,j)
  enddo
enddo
```

Problem 4 – Vectorization

A statement-level dependence graph represents the dependences between statements in a loop nest. Nodes represent single statements, and edges dependences between statements. An edge is generated by a pair of array references that have a dependence. Edges are directed from the source of the dependence to its sink. For example, for a true dependence, the source is a write reference, and the sink is a read reference. There may be multiple edges (i.e., dependences) between two nodes in the graph.

```
       for i = 2, 99
S1:       a(i) = b(i-1) + c(i+1);
S2:       b(i) = c(i) + 3;
S3:       c(i) = c(i-1) + a(i);
       endfor;
```

Here is a basic vectorization algorithm based on a statement-level dependence graph as discussed in class (lecture 26):

1. Construct statement-level dependence graph considering true, anti, and output dependences;
2. Detect strongly connected components (SCC) over the dependence graph (note: a single node may be an SCC by itself); represent SCC as summary nodes; walk resulting graph in topological order; For each visited node do

(a) If SCC has more than one statement in it, distribute loop with statements of SCC as its body, and keep the code sequential.

(b) If SCC is a single statement and has no loop-carried output or true dependencies, distribute loop around it and "collapse" loop into a vector instruction. For example, the loop

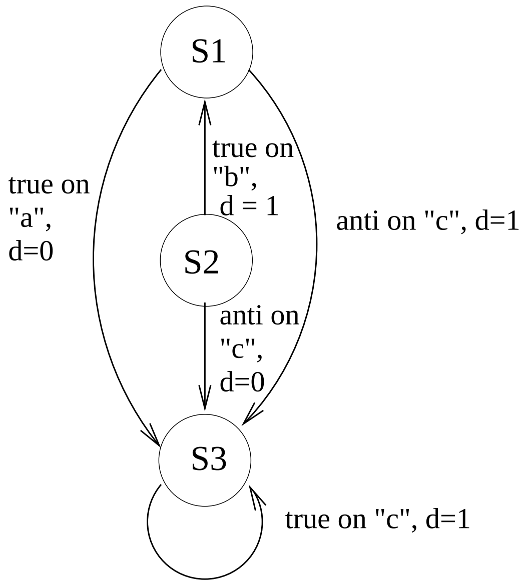
```
for i=1, 100
  a(i) = b(i) + 1;
endfor
```

can be "collapsed" into a single vector instruction

```
  a(1:100) = b(1:100) + 1;
```

. If there are loop-carried true or output dependencies on the single statement, distribute the loop around the statement and keep loop sequential.

1. Show the statement-level dependence graph for the loop with its strongly connected components. Show every dependence by a pair of array references. Note: There may be multiple depencencies between two statements.



Each node is in its own strongly connected component.

2. Show the generated code by the vectorization algorithm described above.

```
S2:     b(2:99) = c(2:99) + 3;
S1:     a(2:99) = b(1:98) + c(3:100);
        for i = 2, 99
S3:        c(i) = c(i-1) + a(i);
        endfor
```