Problem 1 - Dependence Analysis

Give the direction vectors, and if possible the distance vectors for all dependences in the following loop nests. State explicitly whether a dependence is a true, anti, output, or input dependence.

1. do i = 3, 100
   a(i) = a(i-1) + a(i+1) + a(i-2)
   enddo

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

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

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

Problem 2 – 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 = 3, 99
S1: a(i) = b(i-1) + c(i-1) + d(i);
S2: b(i) = a(i-2) + 3;
S3: c(i) = c(i-1) + a(i+1);
S4: d(i+1) = b(i+1) + d(i+1);
S5: e(i) = c(i-1) + 5;
endfor;

Here is our basic vectorization algorithm based on a statement-level dependence graph:

1. Construct statement-level dependence graph considering true, anti, and output dependences; in the final dependence graph, the type of the dependence is not important any more

2. Detect strongly connected components (SCC) over the dependence graph; represent SCC as summary nodes; walk resulting acyclic 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 consists of a single statement and does not have a loop-carried output or true dependence cycle, 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 is a dependence cycle on the single statement (true or output), 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.

2. Show the generated code by the vectorization algorithm described above.
Problem 3 – Dependence Testing and Statement-Level Dependence Graphs

Sketch a basic dependence tester in pseudo code that takes a perfectly nested loop as input that contains only assignment statements with array references on the right and leftmost sides (no control flow). Your program should produce a statement level dependence graph as output. Your pseudo code can assume a function “test” that takes two array references R1 and R2 as input and will return the information whether there is a dependence between R1 and R2, or whether such a dependence does not exist. If a dependence exists, the function will provide the information whether the dependence is loop carried, loop independent, or possibly both. You can write code that uses “helper” functions or “helper” constructs, for example:

test(R1, R2)

for all statements S in loop

for all array references R in S

isRead(R1) // reference R1 occurs on the RHS of an assignment

isWrite(R2) // reference R2 occurs on the LHS of an assignment

InsertDirectedEdge(S1, S2, carried, type) // S1 -> S2, loop carried or loop independent

// type: true, anti, output

...  

Note: The goal of this problem is to make you think about a program that is able to build a statement-level dependence graph. Particular implementation details or efficiencies are not of interest here.