Automatic Parallelization - 2

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Data Dependence Relations

Flow or true dependence

S1: X = ...

S2: ... = X

δ

Anti-dependence

S1: ... = X

S2: X = ...

$\bar{\delta}$

Output dependence

S1: X = ...

S2: X = ...

δ^o

Data Dependence Direction Vector

- Data dependence relations are augmented with a direction of data dependence (direction vector)
- There is one direction vector component for each loop in a nest of loops
- The *data dependence direction vector* (or direction vector) is $\Psi = (\Psi_1, \Psi_2, ..., \Psi_d)$, where $\Psi_k \in \{<, =, >, \leq, \geq, \neq, *\}$
- Forward or "<" direction means dependence from iteration i to $i + k$ (*i.e.,* computed in iteration i and used in iteration $i + k$)
- Backward or ">" direction means dependence from iteration i to $i - k$ (*i.e.,* computed in iteration i and used in iteration $i - k$). This is not possible in single loops and possible in two or higher levels of nesting
- Equal or "=" direction means that dependence is in the same iteration (*i.e.,* computed in iteration i and used in iteration i)

Direction Vector Example 1

```
    for J = 1 to 100 do {
S:      X(J) = X(J) +c
    }
```

$S \; \bar{\delta}_{=} \; S$

X(1) = X(1) +c
X(2) = X(2)+c

```
    for J = 1 to 99 do {
S:      X(J+1) = X(J) +c
    }
```

$S \; \delta_{<} \; S$

X(2) = X(1) +c
X(3) = X(2)+c

```
    for J = 1 to 99 do {
S:      X(J) = X(J+1) +c
    }
```

$S \; \bar{\delta}_{<} \; S$

X(1) = X(2) +c
X(2) = X(3)+c

```
    for J = 99 downto 1 do {
S:      X(J) = X(J+1) +c
    }
```

$S \; \delta_{<} \; S$

X(99) = X(100) +c
X(98) = X(99)+c
note '-ve' increment

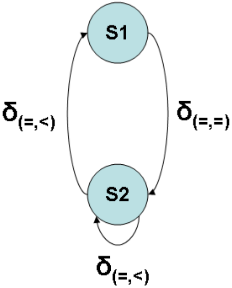
```
    for J = 2 to 101 do {
S:      X(J) = X(J-1) +c
    }
```

$S \; \delta_{<} \; S$

X(2) = X(1) +c
X(3) = X(2)+c

Direction Vector Example 2

```
      for I = 1 to 5 do {
          for J = 1 to 4 do {
S1:           A(I, J) = B(I, J) + C(I, J)
S2:           B(I, J+1) = A(I, J) + B(I, J)
          }
      }
```



Demonstration of direction vector

I=1, J=1: A(1,1)=B(1,1)+C(1,1)
B(1,2)=A(1,1)+B(1,1)
J=2: A(1,2)=B(1,2)+C(1,2)
B(1,3)=A(1,2)+B(1,2)
J=3: A(1,3)=B(1,3)+C(1,3)
B(1,4)=A(1,3)+B(1,3)

S1 $\delta_{(=,=)}$ S2

S2 $\delta_{(=,<)}$ S1

S2 $\delta_{(=,<)}$ S2

Direction Vector Example 3

$S1\ \delta_{(<,>)}\ S2$

```
      for I = 1 to N do {
        for J = 1 to N do {
S1:       A(I+1, J) = ...
S2:       ... = A(I, J+1)
        }
      }
```

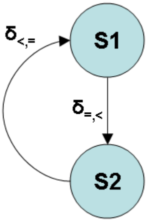
```
I = 1, J = 2
S1:    A(2,2) = ...

I = 2, J = 1
S2:    ... = A(2,2)
```

$S2\ \delta_{(<,>)}\ S1$

```
      for I = 1 to N do {
        for J = 1 to N do {
S1:       ... = A(I, J+1)
S2:       A(I+1, J) = ...
        }
      }
```

```
I = 1, J = 2
S2:    A(2,2) = ...

I = 2, J = 1
S1:    ... = A(2,2)
```

Direction Vector Example 4




```
for I = 1 to 100 do {
  for J = 1 to 100  do {
    for K = 1 to 100 do {
S1:    X(I, J+1, K) = A(I, J, K) + 10
    }
    for L = 1 to 50 do {
S2:    A(I+1, J, L) = X(I, J, L) +5
    }
  }
}
```

	I = 1	I = 2
J = 1	X(1,2,K) = A(1,1,K) A(2,1,L) = X(1,1,L)	X(2,2,K) = A(2,1,K) A(3,1,L) = X(2,1,L)
J = 2	X(1,3,K) = A(1,2,K) A(2,2,L) = X(1,2,L)	X(2,3,K) = A(2,2,K) A(3,2,L) = X(2,2,L)
J = 3	X(1,4,K) = A(1,3,K) A(2,3,L) = X(1,3,L)	X(2,4,K) = A(2,3,K) A(3,3,L) = X(2,3,L)

$\delta_{=,<}$ $\delta_{<,=}$

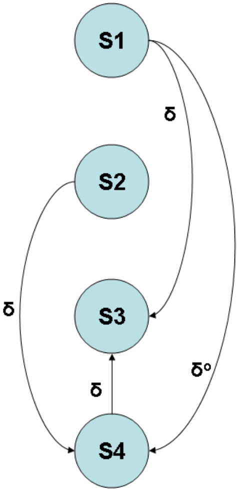
Data Dependence Graph and Vectorization

- Individual nodes are statements of the program and edges depict data dependence among the statements
- If the DDG is acyclic, then vectorization of the program is possible and is straightforward
  - Vector code generation can be done using a topological sort order on the DDG
- Otherwise, find all the strongly connected components of the DDG, and reduce the DDG to an acyclic graph by treating each SCC as a single node
  - SCCs cannot be fully vectorized; the final code will contain some sequential loops and possibly some vector code

Data Dependence Graph and Vectorization

- If all the dependence relations in a loop nest have a direction vector value of "=" for a loop, then the iterations of that loop can be executed in parallel with no synchronization between iterations
- Any dependence with a forward (<) direction in an outer loop will be satisfied by the serial execution of the outer loop
- If an outer loop L is run in sequential mode, then all the *dependences* with a forward (<) direction at the outer level (of L) will be automatically satisfied (even those of the loops inner to L)
- However, this is not true for those dependences with with (=) direction at the outer level; the dependences of the inner loops will have to be satisfied by appropriate statement ordering and loop execution order

Vectorization Example 1



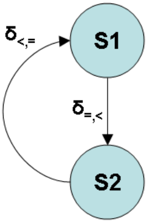

```
for I = 1 to 99 {
S1:   X(I) = I
S2:   B(I) = 100 – I
  }
for I = 1 to 99 {
S3:   A(I) = F(X(I))
S4:   X(I+1) = G(B(I))
  }
```

Loop A

Loop B

```
X(1:99) = (/1:99/)
B(1:99) = (/99:1:-1/)
X(2:100) = G(B(1:99))
A(1:99) = F(X(1:99))
```

Loop A is parallelizable, but loop B is not, due to forward dependence of S3 on S4

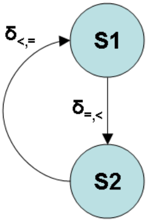



```
for I = 1 to 100 do {
  for J = 1 to 100  do {
     for K = 1 to 100 do {
S1:     X(I, J+1, K) = A(I, J, K) + 10
     }
     for L = 1 to 50 do {
S2:     A(I+1, J, L) = X(I, J, L) +5
     }
  }
}
```

	I = 1	I = 2
J = 1 $\delta_{=,<}$	X(1,2,K) = A(1,1,K) A(2,1,L) = X(1,1,L)	X(2,2,K) = A(2,1,K) A(3,1,L) = X(2,1,L)
J = 2	X(1,3,K) = A(1,2,K) A(2,2,L) = X(1,2,L)	X(2,3,K) = A(2,2,K) A(3,2,L) = X(2,2,L)
J = 3	X(1,4,K) = A(1,3,K) A(2,3,L) = X(1,3,L)	X(2,4,K) = A(2,3,K) A(3,3,L) = X(2,3,L)

$\delta_{<,=}$

Vectorization Example 2.2



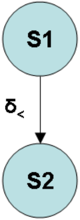

```
      for I = 1 to 100 do {
        for J = 1 to 100  do {
           for K = 1 to 100 do {
S1:             X(I, J+1, K) = A(I, J, K) + 10
           }
           for L = 1 to 50 do {
S2:             A(I+1, J, L) = X(I, J, L) +5
           }
        }
      }
```

I loop cannot be vectorized due to the cycle.

I and J loops cannot be parallelized, due to '<' direction vector. K and L loops can be parallelized

```
for I = 1 to 100 do {
     X(I, 2:101, 1:100) = A(I, 1:100, 1:100) + 10
     A(I+1, 1:100, 1:50) = X(I, 1:100, 1:50) + 5
}
```

Vectorization Example 2.3

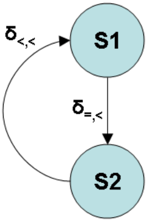


If the I loop is run sequentially, the I-loop dependences are satisfied; J-loop dependences change as shown and there are no more cycles. The loops can be vectorized. However, J-loop cannot be (still) parallelized.

```
    for I = 1 to 100 do {
      for J = 1 to 100  do {
        for K = 1 to 100 do {
S1:       X(I, J+1, K) = A(I, J, K) + 10
        }
        for L = 1 to 50 do {
S2:       A(I+1, J, L) = X(I, J, L) +5
        }
      }
    }
```

```
for I = 1 to 100 do {
    X(I, 2:101, 1:100) = A(I, 1:100, 1:100) + 10
    A(I+1, 1:100, 1:50) = X(I, 1:100, 1:50) + 5
}
```

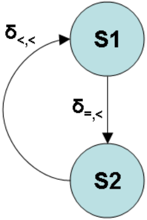
Vectorization Example 2.4




```
    for I = 1 to 100 do {
      for J = 1 to 100  do {
        for K = 1 to 100 do {
S1:        X(I, J+1, K) = A(I, J, K) + 10
        }
        for L = 1 to 50 do {
S2:        A(I+1, J+1, L) = X(I, J, L) +5
        }
      }
    }
```

	I = 1	I = 2
J = 1 $\delta_{=,<}$	X(1,2,K) = A(1,1,K) A(2,2,L) = X(1,1,L)	X(2,2,K) = A(2,1,K) A(3,2,L) = X(2,1,L)
J = 2	X(1,3,K) = A(1,2,K) A(2,3,L) = X(1,2,L)	X(2,2,K) = A(2,2,K) A(3,3,L) = X(2,2,L) $\delta_{<,<}$
J = 3	X(1,4,K) = A(1,3,K) A(2,4,L) = X(1,3,L)	X(2,4,K) = A(2,3,K) A(3,4,L) = X(2,3,L)

Vectorization Example 2.5



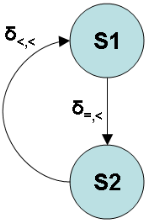
If the program is changed slightly, then dependences change as shown. I and J loops are not parallelizable. If I and J loops are interchanged and J-loop is run sequentially, I-loop can be parallelized. K and L loops are always parallelizable.

```
    for I = 1 to 100 do {
      for J = 1 to 100  do {
         for K = 1 to 100 do {
S1:         X(I, J+1, K) = A(I, J, K) + 10
         }
         for L = 1 to 50 do {
S2:         A(I+1, J+1, L) = X(I, J, L) +5
         }
      }
    }
```

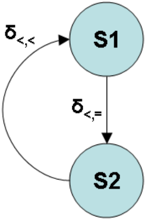
```
for I = 1 to 100 do {
     X(I, 2:101, 1:100) = A(I, 1:100, 1:100) + 10
     A(I+1, 2:101, 1:50) = X(I, 1:100, 1:50) + 5
}
```

Vectorization Example 2.6

Before interchange




```
      for J = 1 to 100 do {
        for I = 1 to 100  do {
           for K = 1 to 100 do {
S1:           X(I, J+1, K) = A(I, J, K) + 10
           }
           for L = 1 to 50 do {
S2:           A(I+1, J+1, L) = X(I, J, L) +5
           }
        }
      }
```



After interchange

	I = 1	I = 2
J = 1	X(1,2,K) = A(1,1,K) A(2,2,L) = X(1,1,L)	X(2,2,K) = A(2,1,K) A(3,2,L) = X(2,1,L)
J = 2	X(1,3,K) = A(1,2,K) A(2,3,L) = X(1,2,L)	X(2,2,K) = A(2,2,K) A(3,3,L) = X(2,2,L)
J = 3	X(1,4,K) = A(1,3,K) A(2,4,L) = X(1,3,L)	X(2,4,K) = A(2,3,K) A(3,4,L) = X(2,3,L)

$\delta_{<,=}$ $\delta_{<,<}$

Concurrentization Examples

```
      for I = 2 to N do {
        for J = 2 to N do {
S1:       A(I,J) = B(I,J) + 2
S2:       B(I,J) = A(I-1, J-1) – B(I,J)
        }
      }
```

S1 $\delta_{(<,<)}$ S2, S1 $\overline{\delta}_{(=,=)}$ S2, S2 $\overline{\delta}_{(=,=)}$ S2

	I = 1	I = 2
J = 1	A(2,2)= = A(1,1)	A(3,2)= = A(2,1)
J = 2	A(2,3)= = A(1,2)	A(3,3)= = A(2,2)
J = 3	A(2,4)= = A(1,3)	A(3,4)= = A(2,3)

If the I loop is run in serial mode then, the J loop can be run in parallel mode

```
      for I = 2 to N do {
        for J = 2 to N do {
S1:       A(I,J) = B(I,J) + 2
S2:       B(I,J) = A(I, J-1) – B(I,J)
        }
      }
```

S1 $\delta_{(=,<)}$ S2, S1 $\overline{\delta}_{(=,=)}$ S2, S2 $\overline{\delta}_{(=,=)}$ S2

	I = 1	I = 2
J = 1	A(2,2)= = A(2,1)	A(3,2)= = A(3,1)
J = 2	A(2,3)= = A(2,2)	A(3,3)= = A(3,2)
J = 3	A(2,4)= = A(2,3)	A(3,4)= = A(3,3)

The J loop cannot be run in parallel mode. However, the I loop can be run in parallel mode

Loop Transformations for increasing Parallelism

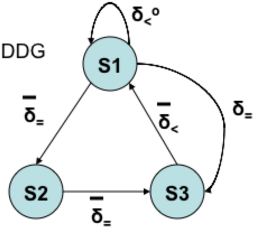
- Recurrence breaking
  - Ignorable cycles
  - Scalar expansion
  - Scalar renaming
  - Node splitting
  - Threshold detection and index set splitting
  - If-conversion
- Loop interchanging
- Loop fission
- Loop fusion

Scalar Expansion

Not vectorizable or parallelizable

```
for I = 1 to N do {
  S1:   T = A(I)
  S2:   A(I) = B(I)
  S3:   B(I) = T
}
```

Cyclic DDG



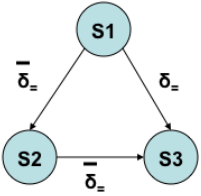
Vectorizable due to scalar expansion

```
for I = 1 to N do {
  S1:   Tx(I) = A(I)
  S2:   A(I) = B(I)
  S3:   B(I) = Tx(I)
}
```

Parallelizable due to privatization

```
forall I = 1 to N do {
  private temp
  S1:   temp = A(I)
  S2:   A(I) = B(I)
  S3:   B(I) = temp
}
```

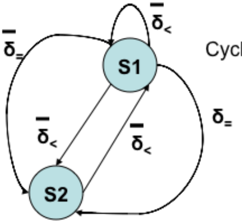
Acyclic DDG



Scalar Expansion is not always profitable

Not vectorizable or parallelizable

```
for I = 1 to N do {
  S1:   T = T + A(I) + A(I+2)
  S2:   A(I) = T
}
```

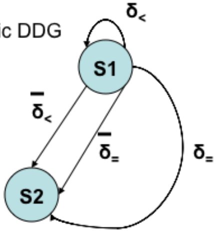


Cyclic DDG

Not vectorizable even
after scalar expansion

```
for I = 1 to N do {
  S1:   Tx(I) = Tx(I-1)+A(I)+A(I+2)
  S2:   A(I) = Tx(I)
}
```

Still cyclic DDG



Scalar Renaming

The output dependence S1 δ^o S3 cannot be broken by scalar expansion

1.

```
for I = 1 to N do {
  S1:   T = A(I) + B(I)
  S2:   C(I) = T*2
  S3:   T = D(I) * B(I)
  S4:   A(I+2) = T + 5
}
```

The output dependence S1 δ^o S3 CAN be broken by scalar renaming

2.

```
for I = 1 to N do {
  S1:   T1 = A(I) + B(I)
  S2:   C(I) = T1*2
  S3:   T2 = D(I) * B(I)
  S4:   A(I+2) = T2 + 5
}
```

3.

```
S3:   T2(1:100) = D(1:100) * B(1:100)
S4:   A(3:102) = T2(1:100) + 5(1:100)
S1:   T1(1:100) = A(1:100) + B(1:100)
S2:   C(1:100) = T1(1:100)*2(1:100)
      T = T2(100)
```

5(1:100) and 2(1:100) are vectors of constants

If-Conversion

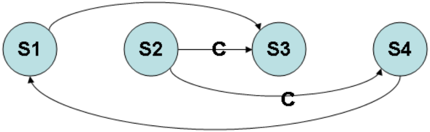
```
for I = 1 to 100 do {
   if (A(I) <= 0) then contnue
   A(I) = B(I) + 3
}
```

```
for I = 1 to 100 do {
   BR(I) = (A(I) <= 0)
   if (~ BR(I))  then
      A(I) = B(I) + 3
}
```

```
BR(1:N) = (A(1:N) <= 0)
where (~ BR(1:N))
   A(1:N) = B(1:N) + 3
```

```
          for I = 1 to N do {
S1:          A(I) = D(I) + 1
S2:          if (B(I) > 0) then
S3:             C(I) = C(I) + A(I)
S4:             D(I+1) = D(I+1) + 1
             end if
          }
```

```
          for I = 1 to N do {
S2:          temp(1:N) = B(1:N) > 0
S4:          where (temp(1:N))
                  D(2:N+1) = D(2:N+1) + 1
S1:          A(1:N) = D(1:N) + 1
S3:          where (temp(1:N))
                  C(1:N) = C(1:N) + A(1:N)
          }
```



Loop Interchange

- For machines with vector instructions, inner loops are preferrable for vectorization, and loops can be interchanged to enable this
- For multi-core and multi-processor machines, parallel outer loops are preferred and loop interchange may help to make this happen
- Requirements for simple loop interchange
  1. The loops L1 and L2 must be tightly nested (no statements between loops)
  2. The loop limits of L2 must be invariant in L1
  3. There are no statements S_v and S_w (not necessarily distinct) in L1 with a dependence $S_v\ \delta^*_{(<,>)}\ S_w$

Loop Interchange for Vectorizability

```
for I = 1 to N do {
   for J = 1 to N do {
S:      A(I,J+1) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Inner loop is not vectorizable

$S\ \delta_{(=,<)}\ S$

```
for J = 1 to N do {
   for I = 1 to N do {
S:      A(I,J+1) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Inner loop is vectorizable

$S\ \delta_{(<,=)}\ S$

```
for J = 1 to N do {
S:      A(1:N, J+1) = A(1:N, J) * B(1:N, J) + C(1:N, J)
}
```

Loop Interchange for parallelizability

```
for I = 1 to N do {
   for J = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Outer loop is not parallelizable, but inner loop is

$S\ \delta_{(<,=)}\ S$

Less work per thread

```
for J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

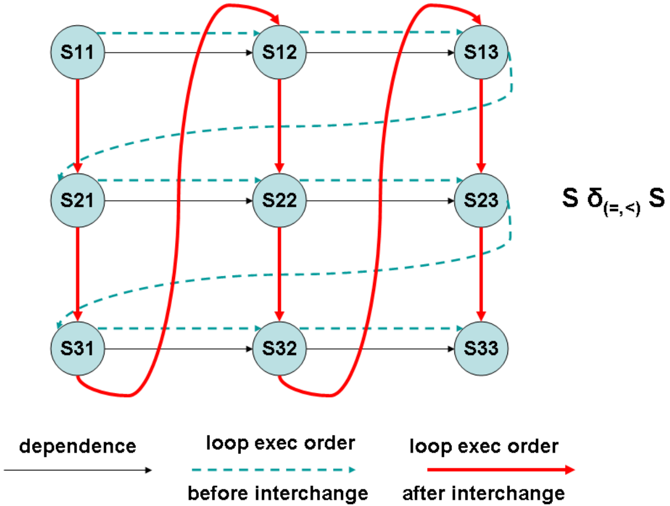
Outer loop is parallelizable but inner loop is not

$S\ \delta_{(=,<)}\ S$

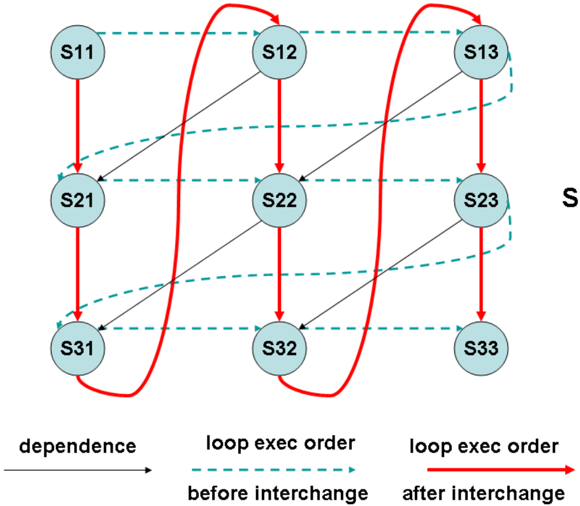
More work per thread

```
forall J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Legal Loop Interchange

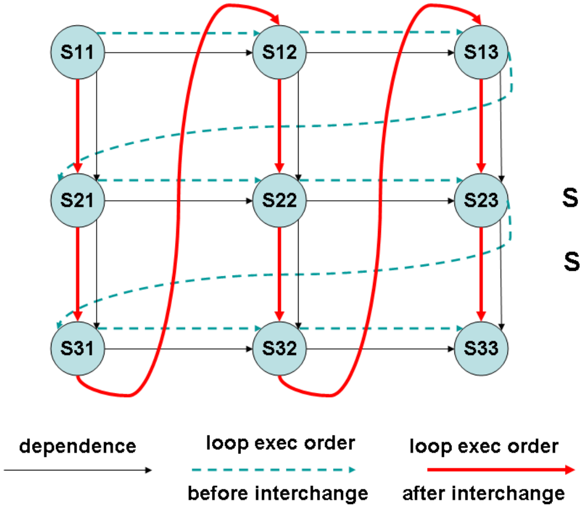


Illegal Loop Interchange



$$S\ \delta_{(<,>)}\ S$$

Legal but not beneficial Loop Interchange



$S\ \delta_{(=,<)}\ S$
and
$S\ \delta_{(<,=)}\ S$

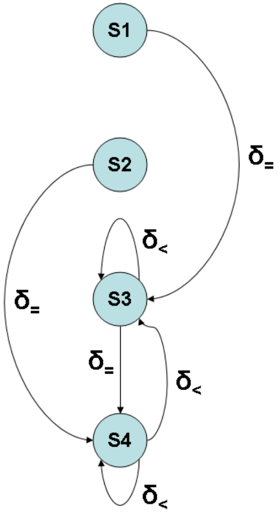
Loop Fission - Motivation

```
     for I = 1 to N do {
S1:     A(I) = E(I) + 1
S2:     B(I) = F(I) * 2
S3:     C(I+1) = C(I) * A(I) + D(I)
S4:     D(I+1) = C(I+1) * B(I) + D(I)
     }
```

The above loop cannot be vectorized

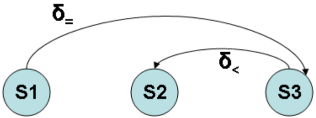
```
L1:  for I = 1 to N do {
S1:     A(I) = E(I) + 1
S2:     B(I) = F(I) * 2
     }

L2:  for I = 1 to N do {
S3:     C(I+1) = C(I) * A(I) + D(I)
S4:     D(I+1) = C(I+1) * B(I) + D(I)
     }
```

L1 can be vectorized, but L2 cannot be



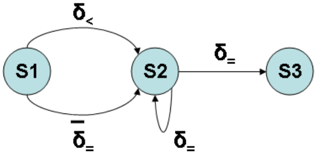
Loop Fission: Legal and Illegal

```
        for I = 1 to N do {
S1:        A(I) = D(I) * T
S2:        B(I) = (C(I) + E(I))/2
S3:        C(I+1) = A(I) + 1
        }
```



In the above loop, S3 $\delta_<$ S2, and S3 follows S2. Therefore, cutting the loop between S2 and S3 is illegal. However, cutting the loop between S1 and S2 is legal.

```
        for I = 1 to N do {
S1:        A(I+1) = B(I) +D(I)
S2:        B(I) = (A(I) + B(I))/2
S3:        C(I) = B(I) + 1
        }
```



The above loop can be cut between S1 and S2, and also between S2 and S3