Automatic Parallelization - Part 1

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Automatic Parallelization

- Automatic conversion of sequential programs to parallel programs by a compiler
- Target may be a vector processor (vectorization), a multi-core processor (concurrentization), or a cluster of loosely coupled distributed memory processors (parallelization)
- Parallelism extraction process is normally a source-to-source transformation
- Requires dependence analysis to determine the dependence between statements
- Implementation of available parallelism is also a challenge
  - For example, can all the iterations of a 2-nested loop be run in parallel?

Example 1

```
for I = 1 to 100 do {
   X(I) = X(I) + Y(I)
}

can be converted to

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

The above code can be run on a vector processor in O(1) time. The vectors X and Y are fetched first and then the vector X is written into

Example 2

```
for I = 1 to 100 do {
   X(I) = X(I) + Y(I)
}

can be converted to

forall I = 1 to 100 do {
   X(I) = X(I) + Y(I)
```

The above code can be run on a multi-core processor with all the 100 iterations running as separate threads. Each thread "owns" a different I value

Example 3

```
for I = 1 to 100 do {
   X(I+1) = X(I) + Y(I)
}
```

cannot be converted to

```
X(2:101) = X(1:100) + Y(1:100)
```

because of dependence as shown below

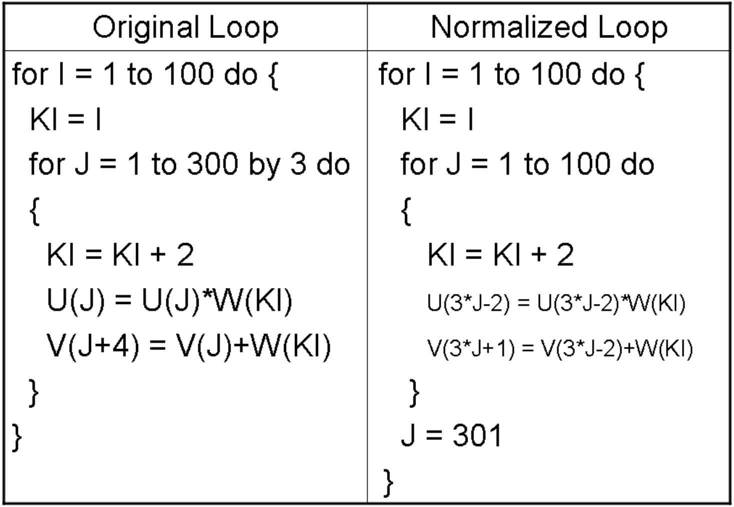
```
X(2) = X(1) + Y(1)
X(3) = X(2) + Y(2)
X(4) = X(3) + Y(3)
...
```

Transformations before Dependence Analysis

- Array subscripts should be linear functions of loop variables
- Loop lower bound should be one and the loop increment should be one
- A few loop transformations are carried out to ensure the above
    - Loop normalization
    - Induction variable substitution
    - Expression folding and forward substitution

Loop Normalization

Loop lower bound → 1, and loop increment → 1



Original Loop	Normalized Loop
for I = 1 to 100 do {	for I = 1 to 100 do {
KI = I	KI = I
for J = 1 to 300 by 3 do	for J = 1 to 100 do
{	{
KI = KI + 2	KI = KI + 2
U(J) = U(J)*W(KI)	U(3*J-2) = U(3*J-2)*W(KI)
V(J+4) = V(J)+W(KI)	V(3*J+1) = V(3*J-2)+W(KI)
}	}
}	J = 301
	}

Induction Variable Substitution

```
for I = 1 to 100 do {
  KI = I
  for J = 1 to 100 do {
    U(3*J-2) = U(3*J-2)*W(KI+2*J)
    V(3*J+1) = V(3*J-2)*W(KI+2*J)
  }
  KI = KI+200
  J = 301
}
```

Now KI is a *constant* in the J-loop. This is the inverse of *operator strength reduction*

Expression Folding and Forward Substitution

```
for I = 1 to 100 do {
  for J = 1 to 100 do {
    S1:  U(3*J-2) = U(3*J-2)*W(I+2*J)
    S2:  V(3*J+1) = V(3*J-2)*W(I+2*J)
  }
  KI = I+200 // may be deleted if KI is not live
  J = 301 // may be deleted if J is not live
}
```

Now all subscripts are linear functions of loop variables as needed for the dependence analysis.

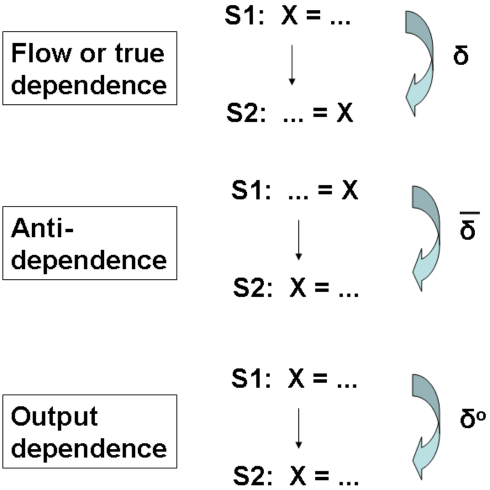
Vector Code Generation

```
I = 1, J = 1, S1: U(1) = U(1) + ...
              S2: V(4) = V(1) + ...
       J = 2, S1: U(2) = U(2) + ...
              S2: V(7) = V(4) + ...
```

- The dependence $S1\ \overline{\delta}_{(=,=)} S1$ is harmless for vectorization of S1
- But, the dependence $S2\ \delta_{(=,<)} S2$ prevents vectorization of S2

```
for I = 1 to 100 do {
    U(1:298:3) = U(1:298:3)*W(I-2:I+200:2)
  for J = 1 to 100 do {
    V(3*J+1) = V(3*J-2)*W(I+2*J)
  }
}
```

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

- 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 doubly 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)

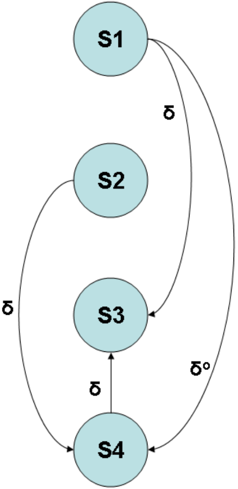
Data Dependence Graph and Vectorization

- 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 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

- 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

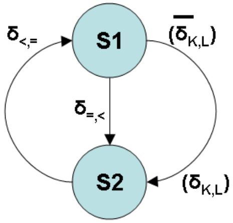
Vectorization Example 1



```
    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))
    }
```

```
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))
```

Vectorization Example 2.1

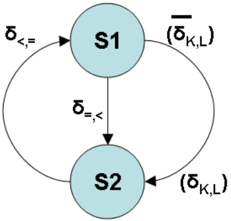


I loop cannot be vectorized due to cycle. J,K,L loops can be vectorized, provided I loop is run sequentially

```
      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 2.2



```
        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 = 1	I = 2

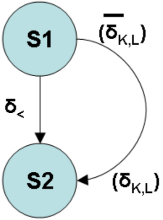
J = 1
$\delta_{=,<}$
	X(1,2,K) = A(1,1,K)
	A(2,1,L) = X(1,1,L)
		X(2,2,K) = A(2,1,K)
		A(3,1,L) = X(2,1,L)

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 2.3

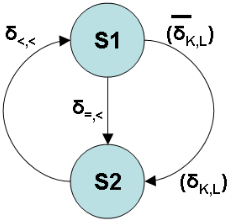


```
    for I = 1 to 100 do {
      for J = 1 to 100  do {
        for K = 1 to 100 do {
S1:       X(I, J+1, K) = A(I, J, K) + 10
        }
        for L = 1 to 50 do {
S2:       A(I+1, J, L) = X(I, J, L) +5
        }
      }
    }
```

If the I loop is run sequentially, the dependences change as shown and there are no more cycles. The loops can be vectorized.

```
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

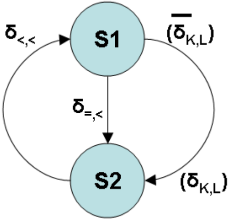


```
      for I = 1 to 100 do {
        for J = 1 to 100  do {
           for K = 1 to 100 do {
S1:           X(I, J+1, K) = A(I, J, K) + 10
           }
           for L = 1 to 50 do {
S2:           A(I+1, J+1, L) = X(I, J, L) +5
           }
        }
      }
```

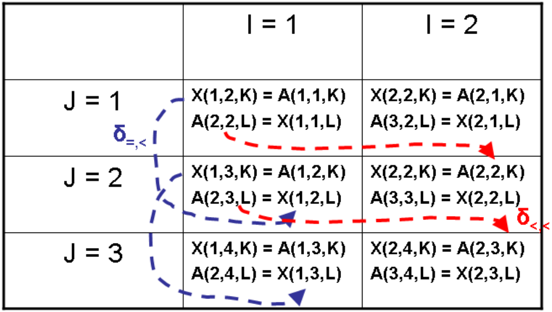
If the program changes slightly, then the dependences change as shown. If the I loop is run sequentially, J,K, and L loops can still be vectorized

```
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 2.5



```
    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	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)

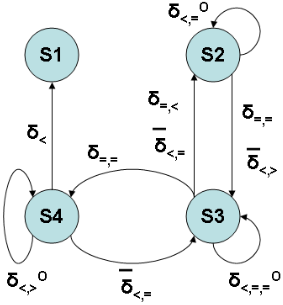
J = 2	X(1,3,K) = A(1,2,K)
	A(2,3,L) = X(1,2,L)
		X(2,2,K) = A(2,2,K)
		A(3,3,L) = X(2,2,L)

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)

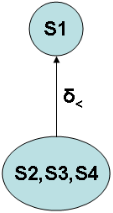
$\delta_{=,<}$
$\delta_{<,<}$

Vectorization Example 3.1

```
    for I = 1 to 100 do {
S1:     X(I) = Y(I) + 10
        for J = 1 to 100 do {
S2:         B(J) = A(J,N)
            for K = 1 to 100 do {
S3:             A(J+1, K) = B(J) + C(J, K)
            }
S4:         Y(I+J) = A(J+1, N)
        }
    }
```

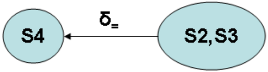


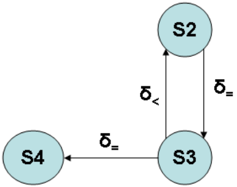
```
    for I = 1 to 100 do {
       code for S2, S3, S4
       generated at higher levels
    }
S1: X(1:100) = Y(1:100) + 10
```



Vectorization Example 3.2

```
        for I = 1 to 100 do {
           for J = 1 to 100 do {
               code for S2 and S3
               generated at
               higher levels
           }
S4:        Y(I+1:I+100) = A(2:101, N)
        }
S1:  X(1:100) = Y(1:100) + N
```

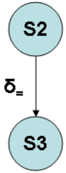




Level 2 DDG for the composite node S2S3S4

Vectorization Example 3.3

```
       for I = 1 to 100 do {
         for J = 1 to 100 do {
S2:         B(J) = A(J,N)
S3:         A(J+1, 1:100) = B(J) + C(J, 1:100)
         }
S4:     Y(I+1:I+100) = A(2:101, N)
       }
S1:  X(1:100) = Y(1:100) + N
```



Level 3 DDG for the composite node S2S3