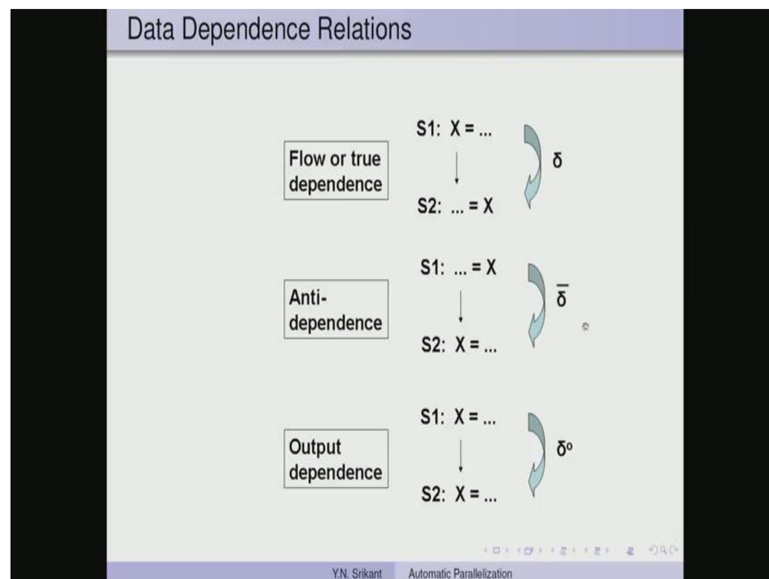
Principles of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module - 12
Lecture - 40
Automatic parallelization Part – 2

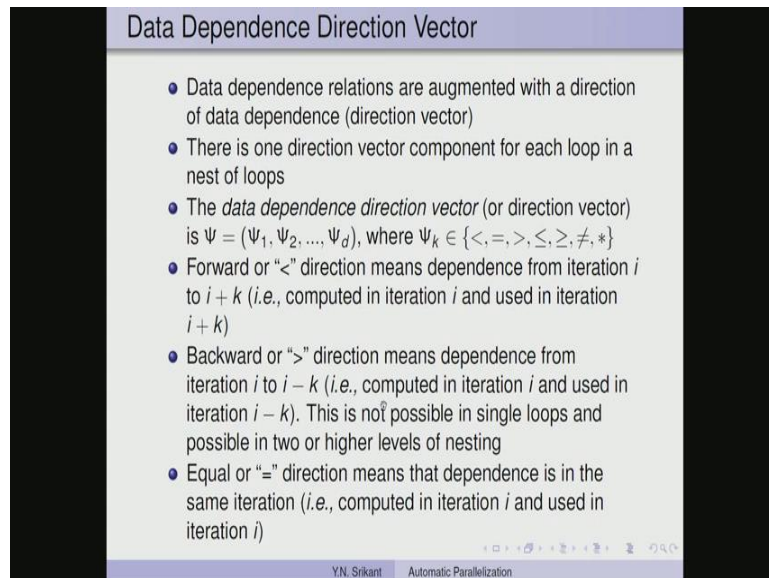
Welcome to part two of the lecture on automatic parallelization. So, in this lecture we will continue our discussion on data dependences, direction vectors and look at a couple of examples of factorization, concretization, etcetera, etcetera.

(Refer Slide Time: 00:37)



So, to do a bit of recap we know that there are three types of dependences s 1 and s 2 are two statements. And if the definition of x is used here without any modification it is a flow dependence. If the usage happens before the definition then it is anti-dependence, and if there are two definitions then it is output dependence.

(Refer Slide Time: 01:04)



The data dependence direction vector is actually an additional information attach to the dependence itself. So, there is one direction vector component for each loop in a nest of loops, so if there is a three nested loop then we have one for each of these three. The dependence data dependence direction vector is a d you know long vector where, d is a depth of nesting. And each of these components can be less than equal to or greater than then the others less than equal to greater than equal to not equal to. And star are actually derived from the principle components less than equal to and greater than. The less than direction means it is a forward direction implying some quantity computed in iteration i and used in a later iteration i plus k.

Whereas, this is the, this is a very common type of direction vector component. Backward or greater than direction means that the dependence is from i to i minus k in other words, computed in iteration i and used in iteration i minus k. If this does not look possible of course, in a single loop it is not at all possible, but in doubly nested loops are higher level loop nesting it is possible I am going to give you a examples of this later. The equal to direction vectors says the dependence is in the same iteration so computed in iteration i and used in iteration i.

(Refer Slide Time: 02:42)

Direction Vector Example 1

Loop	Dependence	Instances
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

Y.N. Srikant Automatic Parallelization

So, we saw this example last time so this is a you know x j equal to x j plus c. So, the value of j is the same in the same iteration, we actually use it first and then define it so it is a delta with equal to. Whereas, this is x j plus 1 equal to x j, so we compute first and then use it later in a different iteration so this is s delta less than s. This is x j equal to x j plus 1 plus c so we use first and then compute so this is anti dependence with a less than direction vector. And this loop is running downward x j equal to x j plus 1 plus c so you can see that x 99 is used here x 98 will be use later and so on. So, this is still a delta less than type of relation and the last one is x j equal to x j minus 1 plus c. So, again we use you know we compute first and use later so this is again a delta less than relation.

(Refer Slide Time: 03:50)

This is a different example, with two levels of nesting so i and j are the two loops. So, we have a i j equal to b i j plus c i j and we also have s 2 which is b of i comma j plus 1 equal to a of i comma j and b of i comma j. So, this is the expanded version of the two loops. So, for i equal to 1 let us say we have j equal to 1 then you know expanding s 1 we get a 1 1 equal to b 1 1 plus c 1 and expanding s 2 we get b 1 2 equal to a 1 1 plus b 1 1. So, there is a flow dependence from this to this and obviously the iteration is the same i i, iteration is the same and the j iteration is also the same. So, s 1 delta s 2 with both the directions b equal to so that is because of this. Then if you consider j equal to 1 and j equal to 2, b 1 2 is defined here in j equal to 1 and used in j equal to 2, but the value of i is the same.

So, this is again a flow dependence and computed in a earlier iteration and used in a later iteration. So, the first i loop has equal to direction vector and the second loop has less than direction vector and it is a flow dependence you know, from s 2 to s 1. Then we have a b 1 3 here this is s 2 again and it is used in you know j equal to 3, b 1 3 here. So, you can look at b 1 2 here and b 1 2 here as well so this is gain a flow dependence and similar in type, but only thing is this is between s 2 and s 2, defined in an earlier j value of s 2 and used in later j value s 2.

So, this is from s 2 to s 2 dependence is delta and the first component is equal to and the second component is less than. So, this is the direction vector in this example and the

dependence diagram is also here. So, we actually place the same dependences here from s 1 to s 2, there is delta equal to equal to so that is 1. And then there is 1 from s 2 to s 2 that is nothing but equal and less than with delta and third one is from s 2 to s 1 which is equal to and less than. So, these are the three dependences that we have along with their direction vectors.

(Refer Slide Time: 06:50)

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)
```

Y.N. Srikant Automatic Parallelization

So, this is third example of direction vector again we have two loops here in both these examples and this is supposed to show the direction vector less than and greater than. As I said greater than direction vector says, computed you know in later iteration, but used in earlier iteration. So, it does not seem to make sense, but it does when we consider doubly nested loops. So, we have s 1 as a i plus 1 comma j equal to and s 2 as equal to a i comma j plus 1.

That is expand the loops with i equal to 1 and j equal to 2 we have s 1 as a 2 2 so that is this part. And if you take a i equal to 2 and j equal to 1, then s 2 will be again equal to of a 2 2 so that is i equal to 2 and j equal to 1. So, clearly there is a dependence from s 1 to s 2, a 2 2 is being computed here and a 2 2 is being used here so this is a flow dependence. So, that is a delta all right s 1 to s 2 there is a delta, what about the data dependence direction vector. So, the value of i from here to here as increase so we compute in a lower iteration number and use in a higher iteration number.

So, the direction vector for i is less than and for j we compute in a higher iteration number j equal to 2 and use in a lower iteration number j equal to 1. So, the second component is greater than, there is no trick here it is just that the value of i is different in these two. So, the j loop starts running a fresh for every value of i so in the iterations of j corresponding to i equal to 1 we define a 2 2 at j equal to 2, but then once we go to the next i j starts running again and that is why we have the value a 2 2 here.

So, we are really using the you know value of a 2 2 in a lower iteration number, but definitely in a different iteration of i so that is you know this is quite realistic, then the second example, s 2 less than greater than delta less than greater than s 1. We have the i loop and the j loop here, we have a i j plus 1 and on the right side s 2 we have a i plus 1 comma j. So, again expand i equal to 1 j equal to 2, so s 2 is a 2 2 equal to and i equal to 2 and j equal to 1, s 1 becomes equal to a 2 2. So, again there is a you know flow dependence from here to here this is i value increases. So, the dependence is direction vector is less than for i the j value reduces so it is a greater than for s 1 for you know a second component. So, all types of dependences are possible and have given you examples of this.

(Refer Slide Time: 10:16)

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
    }
  }
}
```

S1 $\delta_{<,=}$; S1 → S2 $\delta_{=,<}$

	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)

Y.N. Srikant Automatic Parallelization

So, let us look at one more example, so here we have two nested loop for i and j and then inside j we have two independent loops, one for k and another for l. So, the only

dependence from this to this is that x is defined here and x is used here, but apart from that because of i and j you know the there are other dependences as well.

So, let us expand the loop with i equal to 1 a equal to 2 then 3 values of j, j equal to 1 2 and 3. So, with i equal to 1 and j equal to 1 we have x 1 2 and a sorry i equal to 1 and j equal to 1 we have x i j plus 1 and k, I have not expanded k because k is an independent loop here and here you know l is another independent loop here. So, we are only looking at the you know dependences corresponding to the direction vectors for i and j plus 1. So, a suitable value of k here there can always be replaced you know k and l I can just place 1 here and 1 here there is no problem.

So, x 1 2 k is defined here and then in the same value of i and with a different value of j x 1 2 l. So, I can place this make this the 1 here and 1 here so that establishes a concrete dependences between the two so that is shown here. Similarly, between x 1 3 k here and x 1 3 l here there is a dependence, I can placer k equal to a suitable value and l equal to a suitable value here, to make them the same.

Now, for the second statement we have a 2 1 l here which is used in a different value of i so a 2 1 k. So, k and l can be equalized again, the a 2 2 l is defined here and a 2 2 k is used here so this is another you know instance of the same dependence. So, in the first case it was a flow dependence so there is delta and in the second case also, it is a flow dependence so it is again delta. In the first case, the first direction vector component is marked as equal to because it is a same value of i for both these so this is a same column.

Whereas, here we have mark the first component as less than because there is a difference in the value of i from here to here. This is i equal to 1 and this is i equal to 2 and this has increased in this direction. The second component is less than here because j has increased in this direction, so we have less than here. And the value of j is the same across these two, so we have equal to as the direction vector component here.

So, if you draw the dependence diagram here so s 1 to s 2 we have delta you know equal to and less than. So, that is this and from s 2 to s 1 we have the other one delta of less than or equal to. So, you observe that this from s 2 to s 1 where this is from s 1 to s 2. So, these are the two you know adjusts in this you know dependence diagram. So, we will see how this matters a little later.

(Refer Slide Time: 13:56)

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

Y.N. Srikant Automatic Parallelization

So, far we saw examples of data dependence relations and direction vectors now, it is time to understand how to use the dependences in order to do vectorization and you know concurrentization. So, individual nodes are statements of the program and edges depict data dependence among the statements. So, we have already seen this, this is how the data dependence diagram is created, graph is created.

If the DDG is a acyclic then vectorization of the program is possible and is straight forward. So, remember the most important condition for vectorization is that data dependence diagram or the graph should be acyclic. The direction vector by itself does not pause a problem here, but it will definitely pause a problem for the concurrentization. So, vector code generation can be done using a very simple topological sort order on the data dependence graph.

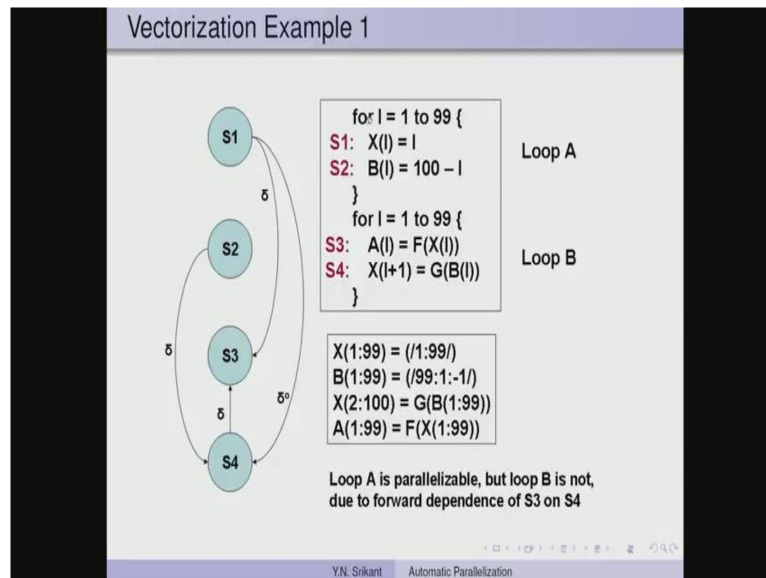
So, suppose the graph is cyclic then it is bit more complicated so we find all the strongly connected components of the data dependence graph and reduce the DDG to an acyclic graph, by treating each strongly connected component as a single node. Now, the you know once it has become acyclic we can actually now generate vector code, but as far the SSC are concerned they cannot be fully vectorised, but the final code will contain some sequential loops and possibly some vector code. So, that is how this is going to be this is bit more complex and it is not possible to provide a very simple example here.

So, we are going to emit you know rather not took it any example in this case, but will concentrate our attention here. Then in the case of a concurrentization if all the dependence relations in a loop nest have a direction vector of equal to, then the iteration of the loop can be executed in parallel with no synchronization between iterations. So, remember direction vector value of equal to for a loop, so in that particular if the direction vector value is equal to. That means, all the dependences are in the same iteration they do not flow across iterations. Therefore, iterations of loop can be executed in parallel.

There are couple of observations here which are very important, any dependence with a forward direction in an outer loop will be satisfied by the serial execution of the outer loop. So, if there is a less than direction in the outer loop you run it sequentially, then the dependence automatically satisfied for that loop. And if an outer loop l is run in sequential mode then the all the dependences with a forward direction at the outer loop of l will be automatically satisfied, even those of the inner loops. So, if we this is very important thing if we are able to if we run the outer loop in a sequential mode. Then you know we can run all the inner loops in a parallel mode provided you know the direction vectors permit. So, we do not have to worry too much once we run it sequential mode everything will be satisfied in at the inner levels.

So, we can run them in parallel only thing is the outer must have less than direction you know in all the edges. So, if some of the edges of data dependence diagram corresponding to the outer loop have equal to direction vector. Then a you know running the outer loop in sequential mode will not satisfy all the dependences so that will be a problem. However, this is not true for those dependences with equal to direction at the outer level so I already mention this. The dependence of the inner loops will have to be satisfied by appropriate statement ordering or and or loop execution order we are going to see examples of this very soon.

(Refer Slide Time: 18:34)



So, let us take an example of vectorization so here is the here is a loop i with two statements s 1 and s 2 and here is the another loop with i as the index with s 3 and s 4. The index of course, does not matter because these two are independent loops. Now, we have s 1 s 2 s 3 and s 4 the dependences among the statements which also shown here, between s 1 and s 2 there is nothing in common. So, they do not have a dependence and then x i is computed in this loop and then it is used in this loop. So, obviously between s 1and s 3 there is a flow dependence delta, I have not bother to indicate the direction vector. So, it is just dependence because vectorization does not bother about the direction vector. Then we have used x i here, but we have also computed x i plus 1 here.

So, we compute and then use so this is the way it is so x 2 and then x 1 x 3 and x 2 etcetera, etcetera. So, what happens is from s 4 we would have a dependence delta to s 3 so that is also there so this is very important. And then again we have value of b i being computed here and the value of b i being used here. So, it is a same value phi so from s 2 to s 4 we again have a delta dependence and between these two s 1 and s 4 there is also an output dependence from s 1 to s 4 that is delta O. So, these are the various dependences in our program.

Now, obviously this is a directed acyclic graph there are no cycles here so if we do a topological sort of this graph, then the statements can be vecto statements can be emitted. Now, this x i s 1 has no incoming arcs so we can emit the code for s 1 directly. So, x 1 to

99 equal to the vector constant 1 to 99 so x i equal to i means x 1 equal to 1, x 2 equal to 2 etcetera. So, this is a vector constant 1, 2, 3, 4 etcetera up to 99 so this assignment is a vector statement for this loop, this part of the loop.

The second statement is s 2 so again it is very similar, b i equal to 100 minus i becomes 99 colon 1 colon minus 1. So, we have you know i equal to 1 so this starts with 99 then goes to 98, 97 etcetera. So, this vector with a stride of minus 1 automatically indicates that is a vector with 99, 98, 97 etcetera, etcetera. So, this is s 2 so these two do not have incoming edges so this can be processed right in the beginning. Now, s 3 has an incoming edge from s 4 and it has an incoming edge from s 1 so s 3 can be processed only after s 1 and s 4 are complete, but s 4 itself can be processed once s 2 and s 1 are complete. So, we have finished code generation for these two.

So, in the execution order the vector code will be executed in the sequential order. So, this is s 4 and x i plus 1 equal to g of b i so x 2 to 100 equal to g of b 1 to 99. So, this is the vector statement corresponding to this loop, this part of the loop. And lastly, the s 3 so that would be a 1 to 99 equal to f of x colon 1 to 99. So, this is a very simple you know vectorizable set of loops so we just you know emit the code in the topological sort order and automatically it gets done.

(Refer Slide Time: 23:11)

Vectorization Example 2.1

```
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)

Y.N. Srikant Automatic Parallelization

The second example, so we have already seen this program before and we also discussed this these dependences. So, from s 1 to s 2 there is a dependence delta equal to and less

than and from s 2 to s 1 there is another dependence delta less than or equal to so this is a cyclic graph.

(Refer Slide Time: 23:33)

Vectorization Example 2.2

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

```
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
}
```

Y.N. Srikant Automatic Parallelization

And therefore, the loops cannot be vectorised so i and j loops cannot be vectorized of course, it is always possible to vectorize the k and l loops separately, that is never an issue. So, now we actually try run the i loop let us say in sequential mode so i equal to one to 100.

(Refer Slide Time: 24:03)

Vectorization Example 2.3

S1 $\delta_{<}$ S2

```
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 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 {
  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
}
```

Y.N. Srikant Automatic Parallelization

So, we run the loop in sequential mood now, the dependences corresponding to i will all be satisfied we can take out the i part from this dependence diagram.

(Refer Slide Time: 24:19)

Vectorization Example 2.2

S1

S2

$\delta_{<,=}$

$\delta_{=,<}$

```
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
}
```

Y.N. Srikant Automatic Parallelization

So here for example, so this less than this is equal to and less than so the equal to part can be taken out. And in this case this less than is automatically satisfied and equal to is never a threat for vectorization. So, we can remove this arc completely so that is what we have done here.

(Refer Slide Time: 24:41)

Vectorization Example 2.3

S1

S2

$\delta_{<}$

```
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 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 {
    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
}
```

Y.N. Srikant Automatic Parallelization

So, between this and this so you know so this the equal to arrows because of the second component, but when vectorization is performed, we are going to actually you know actually read this entire vector and then make the assignment. So, here also we are going to read the entire vector and then make the assignment so because of that the vectorization of the j loop is also possible.

So, the i loop dependences are satisfied the j loop dependences change as before. So, we first emit the vector code for s 1 and then emit the vector code for s 2 so automatically this is these are the two vectors statement inside the i loop. So, these are executed sequentially so x i comma 2 to 1 0 1 comma 1 to 100 equal to a i 1 to 100 and 1 to 100.

(Refer Slide Time: 25:53)

Vectorization Example 2.1

```
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)

Y.N. Srikant Automatic Parallelization

So, let us go back to the dependence diagram here so we are running the i loop in sequential mode. So, these are all going to be run first and then these etcetera, etcetera and the j part is you know vectorized so that is that is precisely what we are doing. So, this dependencies from this s 1 to s 2 since, all the vectors you know in a vector code all the s 1 statements are executed first and then the s 2 statements. This particular dependence will always be taken care of…

(Refer Slide Time: 26:30)

Vectorization Example 2.2

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

```
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
}
```

Y.N. Srikant Automatic Parallelization

So, that is what we have shown here so the x statement is run s 1 is completed and only then s 2 begins. So, automatically the dependence will be taken care of for this iteration for this particular two variables x and x.

(Refer Slide Time: 26:53)

Vectorization Example 2.3

S1 $\delta_{<}$ S2

```
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
    }
  }
}
```

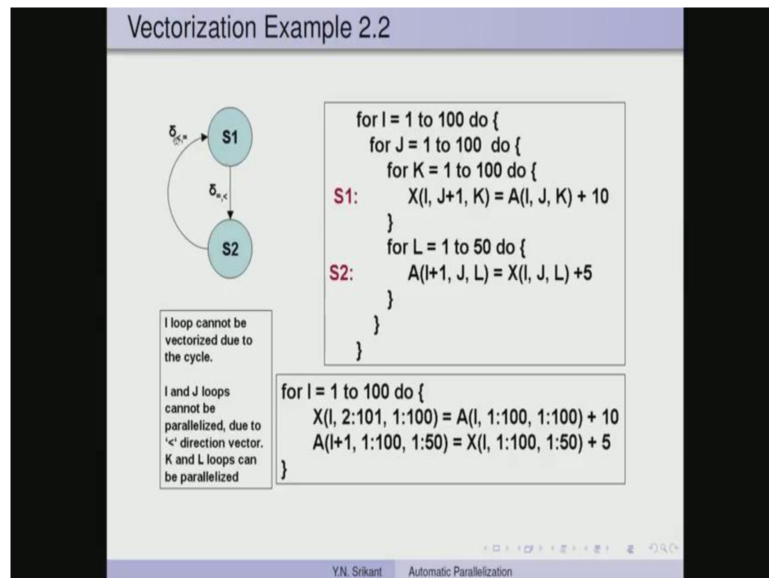
If the I loop is run sequentially, the I-loop dependences are satisfied; J-loop dependences change as shown and there are no more cycles. The loops can be vectorized. However, J-loop cannot be (still) parallelized

```
for I = 1 to 100 do {
   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
}
```

Y.N. Srikant Automatic Parallelization

So, there is also a statement here that the j loop cannot be parallelized, so that is true.

(Refer Slide Time: 27:00)



The reason being the direction vector component is less than here, but it is equal to for the i loop in this particular edge. So, what we really you know mentioned in the observations is that if the corresponding loop direction vector component is less than in all of the edges. Then you know sequentially running that particular loop will satisfy all the dependence inwards, but in this case we have equal to here and less than here.

So, even if we run the i loop in sequential mode the j loop cannot be run in parallel mode, but the k and loop, k and l loops can always be parallelized. So, assuming that we run i and j in sequential mode, then this particular part k loop can always be and the l loop these two can be vectorized and run even parallelly if necessary, but that is not advisable we will see while later. So, here is the example of the code which is slightly changed so the previous one we had you know the dependence in a slightly different fashion. So for example, the dependence ran from here to here and here to here.

(Refer Slide Time: 28:32)

Vectorization Example 2.3

S1 → S2 ($\delta_<$)

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
}
```

Y.N. Srikant   Automatic Parallelization

Whereas, here the dependences are slightly different so even the code is different so this is x i j plus 1 k a i j k and here it is a i plus 1 and j l and x i j l.

(Refer Slide Time: 28:42)

Vectorization Example 2.4

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

```
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)

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

Y.N. Srikant   Automatic Parallelization

Where as in this case it is i j plus 1 k and a i j k a i plus 1 j plus 1 l so it is not j anymore. So, what happens is the dependence is not from here to this, but it is from here to the next one, this dependence is as before. So, it is still delta i equal to and less than, but this particular dependence is from i equal to 1 to i equal to 2. So, that is less than again and here we have j equal to 1 and it is j equal to 2 so again this will also be less than. So, we

have a delta less than less than for s 2 to s 1 and we have a delta equal to and less than for this particular edge.

(Refer Slide Time: 29:37)

Vectorization Example 2.5

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

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
}
```

Y.N. Srikant   Automatic Parallelization

Now, the dependences have changed now, it is possible to interchange the i and j loops. So, there are test for conditions for loop interchange in this case they are satisfied. So, it is possible to interchange the i and j loops in other words, the j loop runs first and then the i loop runs. If that happens are obviously the the dependence direction vector components also get swapped. So, this becomes a delta less than equal to and this of course, remains as delta less than, less than. Now, both the you know edges have delta less than in the for the outer loop. So, that this would be the j loop and this would be the i loop.

(Refer Slide Time: 30:27)

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
    }
  }
}
```

	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)

After interchange

Y.N. Srikant Automatic Parallelization

So therefore, so we have the j loop and the i loop. The dependences you know even though they remain the same the it is possible to now run the j loop in parallel mode. So, whereas the sorry the j loop can be run in sequential mode so if we do that then you know the dependences of all the nested loops inside will be satisfied.

So, in other words if we run the outer loop which is j in sequential mode, then the i loop can be run in parallel mode so that is advantage that we have. So, that is about you know that is one of the examples sequential that we have for concurrentization. So, we run the outer loop in sequential mode, but then we can run the inner loop i loop in parallel mode. This is always advantageous because the inner loop being bigger the amount of work for each thread will increase. Whereas, if we had actually run the outer loop in you know if we simply say that the inner loop has very little work, then making it into a thread is of not much use.

(Refer Slide Time: 31:53)

Here, are more examples of concurrentization we have i equal to 2 to n here and we have a j equal to 2 to n here and we have s 1 and s 2 here. So, in this case again when we expand a loop i equal to 1 i equal to 2 and we have j equal to 1, j equal to 2 and j equal to 3. So, we have a 2 2 a 1 1 here a 2 3, a 1 2 here a 2 4, a 1 3 here and on this side we have a 3 2, a 2 1 and a 3 3 and a 2 2. So, a 2 2 is being used here and it is being defined here.

So, there are many you know dependences here right so for example, we have you know a 2 2 here so s 1 delta less than, less than s 2. So, from s 1 to s 2 there is a delta and that is a flow dependence s 1 to s 2 and then it is less than, less than. So, this is i equal to 1 and j equal to 1 and this is again i equal to 2 and j equal to 2. In both cases the i equal to 2 is more than i equal to 1 and j equal to 2 is more than j equal to 1 so this dependence corresponds to that.

Then we have another one, s 1 delta bar equal to equal to s 2 so that corresponds to this b i j. So, I have not shown it here, but that is easy to see this is b i j and this is b i j so there is a usage and then there is a definition. The third one is also an anti-dependence s 2 delta bar equal to equal to s 2 so there is b i j here and b i j here as well. So, this is the usage and then this is the definition so there is an anti-dependence from s 2 to s 2 as well.

So, these are the three dependences in this loop now the in this case for example, if we i loop can be this is the true dependence right, the other two are anti-dependences and that is not of much importance to us. So, if we run the outer loop in sequential mode so that is

the i loop, then the j loop can be run in parallel mode. So, that is an advantage here so we can run this in serial mode and then we can run this in parallel mode. So, obviously this will be satisfied.

Then the second example of concurrency so we have i loop here and the j loop here. So, we have s 1 delta equal to less than s 2 so again you know so we have a 2 2 here and a 2 2 here. So, it is same value of i, but different value of j so that is why this is correct. Then we have s 1 delta bar equal to equal to s 2 so that is this b i j. And then of course, s 2 delta bar equal to equal to s 2 is corresponding to these two, these three.

Now, the j loop cannot be run in parallel mode, but however the i loop can be definitely run in parallel mode. So, even here you know we cannot run the i loop or the j loop in parallel mode so that is why we resort to this sort of a thing. Whereas, here we have equal to as the component for this, this and this all three. So i loop can be definitely run in parallel mode, but the j loop cannot be, but that is perfectly because running the j loop in sequential mode gives us a lot of work for each iteration.

So whereas, here we had run i in sequential mode and then we were trying to j in parallel mode. So, if we do that than the amount of work for the j loop is a little less compare to compare to this. As it is you know if the j loop is big enough, then it could be run in parallel mode with n of work, but otherwise if the j loop has little work then it is not a good idea to run it in parallel mode.

(Refer Slide Time: 36:35)

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

Y.N. Srikant Automatic Parallelization

Now, let us look at a couple of transformations, which can increase the parallelism there are many of these. For example, recurrence breaking or cycle breaking there are ignorable cycles, then scalar expansion, scalar renaming, node splitting, threshold detection and index set splitting, if-conversion etcetera, etcetera. So, we are going to look at only a few of these to understand what goes on, then we have loop interchanging, loop fission and loop fusion.

(Refer Slide Time: 37:07)

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

Y.N. Srikant   Automatic Parallelization

Scalar expansion for example, we have a scalar in the loop t here and t here, if you look at the dependence diagram of this particular program. Then we have you know s 1 s 2 and s 3 here, from s 1 to s 2 there is a delta bar. So, from s 1 to s 2 so that is this a i and then there is from s 1 to s 3 there is delta equal to. So, that is between this t and this t and then between s 2 and s 3 there is again delta bar so s 2 and s 3 so that is this part right.

And then from s 3 to s 1 there is delta less than bar so that is from here to here that is we are using it here and then you know defining it there. And finally, there is a self loop delta less than O on s 1 so that correspond to this t. So, we write into the same location again and again so the iteration i equal to 1 must write into t and only than the iteration number two can write into t. So, this is an output dependence on t on for s 1 and since the statement is the same, we actually have a self loop and obviously it is less than because the i, i iteration numbers keep increasing.

This is obviously a cyclic group right this, this and this there is a cycle. Suppose, we make t you know into a vector that is the scalar expansion, scalar is expanded into a vector. So, we have t x of i equal to a i, a I equal to b i and b i equal to t x of i so if we do that then obviously the loop goes away. So, this loop is gone and we have from s 1 to s 2 that is very you know that is this and we also have from s 2 to s 3 so that is also there. And we have from s 1 to s 3 this is s 1, this is s 3 so this is a flow dependence rest of the dependences vanish. Now, this particular dependence diagram is cycle free so we can vectorize it using topological sort so we do this first then this and then this. So, we can very simply execute rather emit the parallel vector code for this.

The other possibility is if we are running it on multi core processors we can actually make this temp t, into a private variable separate for each core. So, assume that each iteration runs on a different core so for each core we have a space little bit of memory space available. So, we make it a private variable for each iteration so then again this becomes a cycle free you know just like this, there is no cycle here and all the dependence are within the same iteration. So, we can easily parallelize this particular loop as well.

(Refer Slide Time: 40:37)

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

Y.N. Srikant   Automatic Parallelization

Scalar expansion may not be always profitable so if you consider this program we have t equal to t plus a i plus a i plus 2 and a i equal to t. So, this is a cyclic graph right there are many dependences so you know so we have a dependence from s 1 to s to s 1, we have a

dependence from s 1 to s 2. Then we have one from s 1 to s 2 for this and then we have again you know this a i plus 2 2 to a i so there are many many and of course, one on s 1 to s 1 itself.

So, there are so many of this dependences here and it is a cyclic DDG, but making the temporary t into a vector actually still retains this as a cyclic data dependence graph. So, it does not change it all you know from the couple of them removed so this is gone, but, this remains. There is no change because of this so because of this there is a this remains and then remain we also have a cycle from a here to you know here to this and this again.

So, this to this there is no cycle, but we have a cycle right here, this is still cyclic. So, we got rid of one of them this particular thing we got rid of right and we also got rid of this to this, but this cycle still remains. So, cyclic data dependence graph cannot be vectorize will have to do this sequentially and then vectorise the rest of them otherwise we need to run it sequentially.

(Refer Slide Time: 42:33)

Scalar Renaming

The output dependence $S1\ \delta^o\ S3$ 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 $S1\ \delta^o\ S3$ 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

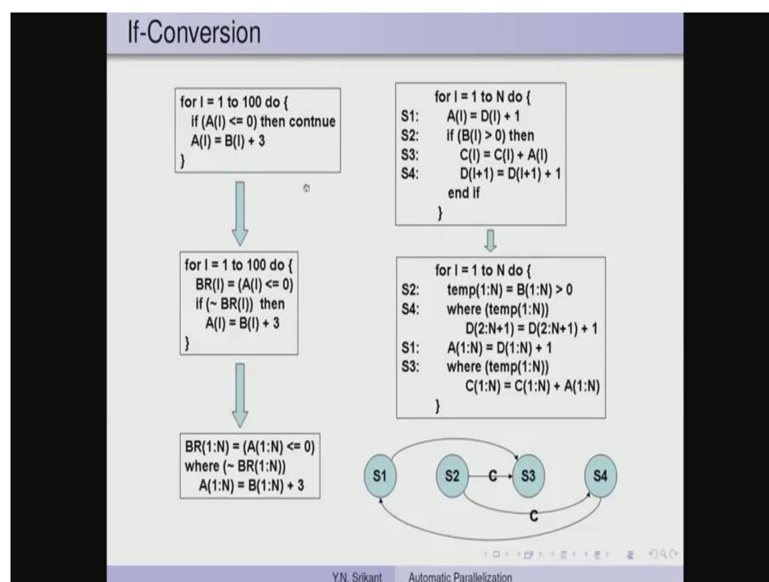
Y.N. Srikant   Automatic Parallelization

So, scalar renaming tries to you know remove anti and output dependences. So, here we have t equal to a i plus d I, v i then we again use t equal to d i plus d i star b i. So, if we introduce a separate variable for each of this so they become t 1 and t 2. So, the output dependence between s 1 and s 3 now, goes away the now you know we can vectorise this code.

So, there is no problem we the removed the by renaming the scalar t and making it separate t 1 and t 2, we have eliminated the you know output dependence and now the vector this can be vectorised. So, that can that easy to see because this t 1 does not have any dependence on it now, this this a i here and here is a i plus 2, but we are not actually executing this particular code in concurrent mode we executing we execute s 3 then s 4 then s 1 and then s 2.

So, automatically we compute here and then use it in s 1 so compute in s 4 and then use it in s 1. So, that is automatically taken care of. Then we have compute in s 3 and use in s 4 so compute in s 3 and use in s 4 that is also taken care of. And then we have compute in s 1 and use in s 2 so that is also taken care of. So, in this manner we can parallel you know make this code run in vector fashion.

(Refer Slide Time: 44:18)



So, we had looked at if-conversion a little before now if-conversion is also a way to assist in vectorization and of course, concurrentization. Here, we have a program i equal to 1 to 100, if a i less then equal to 0 then continue a i equal to b i plus 3 otherwise. So, in this case if-conversions says you know this is a conditional statements so there is no way we can make it a vector code.

So, what we do is we introduce a vector condition for a i less than equal to 0 so b r of i equal to a i less then equal to 0. So, this is a vector of conditions and we make this you know change the program by saying, if not of b r i then a i equal to b i plus 3. So, instead

of continue we made it like this and assuming that there is a masking operation available in vector machine. The masking says where ever the mask is true execute the statement and where ever it is falls do not execute it.

So, we compute the mask as before then we have this is in vector mode so instead of this now this was still in a sequential loop. Now, we made a vector of the conditions and then we said where not of b r 1 to n a 1 to n equal to b 1 to n plus 3. So, we have actually introduce this masking statement and this is still a vector code. So, this is the advantage you know if we use if-conversion, all the control dependence because of the if then else condition is automatically translated to converted to data dependence here.

Then we have another example, with s 1 s 2 s 3 s 4 inside a loop. So, we have an if then and then there are two you know there is a statements a equal to c i plus a i. And then we have another statements d of i plus 1 equal to d i plus 1 plus 1 and if. So, there are two statement here within the condition and there is one statement outside the condition. So, if you draw the dependence diagram now, there is then s 1 then s 2 they do not have any you know dependences on them.

Now, there is s 3 so s 3 is actually dependent on s 2 in by a condition so that is why it is shown as a c. And there is a dependence from s 1 to s 3 as well so that is because we are using a i here compute it here and use it here. Then we have a a you know depends from s 2 to s 4, which is again conditional just like s 2 to s 3 was a conditional dependence this is also a conditional dependence and then we have a dependence from s 4 to s 1.

So, that is because we are computing d i here and using it here. So, the these are the various dependences now you know we can actually emit vector code corresponding to this. So, what we do is we for i equal to 1 to n right sorry this i loop, actually this should be removed this is a minor mistake, there is no loop here this is basically vector code.

So, temp 1 to n is equal to b 1 to n greater than 0, so that is what we compute as the condition that is a vector. And then we have a mask execution where temp 1 to n d 2 you know colon n plus 1 equal to d 2 colon n plus 1 plus 1. So, we execute the second statement as a mass statements so what we have really done is we have executed the mask statement first.

So, that is s 2 and then we have a executed the s 4 statement because s 1 is dependent on it. So, that is a conditional mask statement then we execute s 1 so which is a none conditional statement and then we execute s 3, which is again a conditional statement so this statement is again conditional. So, the order in which this is being executed is the topological salt so first s 3 then s 4, then s 1 and finally, s 3. So, this is the execution order and this is the execution order for the vector code so just vomit this i equal to 1 because there is no loop there.

(Refer Slide Time: 49:19)

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 $S_v\ \delta^*_{(<,>)}\ S_w$

Y.N. Srikant Automatic Parallelization

Loop interchange is another very important transformation. So, for machines with vector instructions inner loops are preferable for vectorization and we can use loop inter change to enable this. For multi-core and multi-processor machines, parallel loops are preferred to be you know parallel outer loops are preferred. So, again loop interchange may be able to achieve this. There are simple conditions for loop interchange to be possible of course, l 1 and s 2 must be tightly nested, no statement between the loops and then the loop limits of s 2 must be invariant in l 1.

So, we cannot have i equal to 1 to n and then second one saying j equal to i and then something else, i to something that cannot be do done. Then most important is there are no statements S v and w in l one with dependence of S v delta star less than greater than s w delta star says it could be any of the dependence less then you know either true and t r output. So, if we have S v delta star s w with less than and greater than, then if we

interchange the loop this greater than would become the first component, which is meaningless. That is the reason why such loops cannot be interchanged.

(Refer Slide Time: 50:46)

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)
}
```

Y.N. Srikant Automatic Parallelization

So, this is a very simple these two loops and this statement the dependences is s delta equal to less than equal to s so interchanging is possible. The i loop cannot be vectorized now, you know we interchange the two loops j runs outside and i runs inside. Now, the inner loop is a definitely you know vectorizable, sorry here j loop could not be vectorize and i became the inner loop now we can vectorize the inner loops. So, the dependences have also become dependence direction vectors have change you can absorb that here. So, this is the vector code that corresponds to this particular thing so we run the outer loop in the sequential mode now the inner loop runs in vector mode.

(Refer Slide Time: 51:37)

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

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

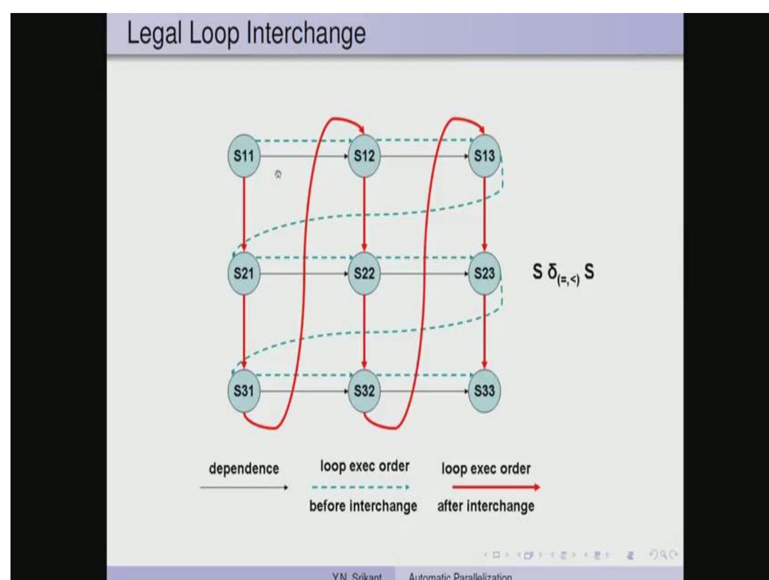
More work per thread

```
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)
   }
}
```

Y.N. Srikant Automatic Parallelization

Again, here the outer loop is not parallelizable because we have a less than so we want to exchange these two once we exchange it become equal to. And now, there is more work per thread because i loop will be now we run in sequential mode and the j loop will be run in parallel mode. So, this is how each thread will now run loop so that is how it runs.

(Refer Slide Time: 52:04)



So, if you look at the diagrams for dependences, let us say this are the various instances of statements for i equal to j equal to different values. So, and let us say this are the

dependences in black. So, if you were running the loop in this fashion to begin with now loop interchange will run them in this order right. So, you can see that with this type of a dependence there is no violations of any of this dependences. That is what is most important if the loop interchange has to be legal.

Whereas, if we have something like this then you know running it in this fashion will satisfy the dependence of this kind from s 1 to 2 s 2 1, but if we run it in this fashion this runs before this so the dependences violated. So, any of this backward dependences are going to be violated if there is loop interchange with loops of this kind.In some cases, we have dependences in both directions right rectangular. Now, loop interchange is legal, but obviously it does not give you any benefit because we cannot you know run any loop in parallel, when there are dependences in both directions. So, that is the example for loop interchange being possible, but with no benefit.

(Refer Slide Time: 53:36)

Now, loop fusion is something, which says the whole loop may not be vectorised, vectorizable, so divide the loop into smaller loops. Now, this loop vectorizable, but this is not, but still the programs speed up little bit because this loop becomes vectorizable. This is loop fusion, so we divide the loop into two parts.

(Refer Slide Time: 53:59)

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

Y.N. Srikant Automatic Parallelization

So in this case for example, there is a dependence from s 3 to s 2 you can see that s 3 to s 2 so we have c i here and c i plus 1 here, so we compute here and use here. So, if you break the loop here make these two into one loop and s 3 into another loop. Obviously, this dependence will be violated we cannot really compute something in the s 3 and then use it in s 2, because they have become in separate loops. Whereas, in this case all the dependences are in the forward direction so we can break the loop either here or here or both places. So, in other words we can make 3 loops out of these three statements and still there is going to be no violation of any of these conditions. So, that brings us to the end of the lecture and the end of the course as well.

Thank you.