Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

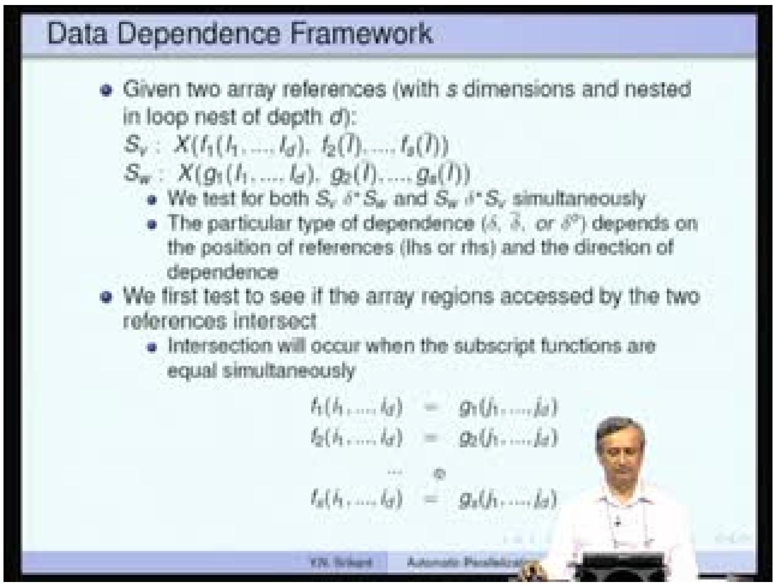
Indian Institute of Science, Bangalore

Module No. # 14

Lecture No. # 27

Automatic Parallelization-Part 4

(Refer Slide Time: 00:20)



Data Dependence Framework

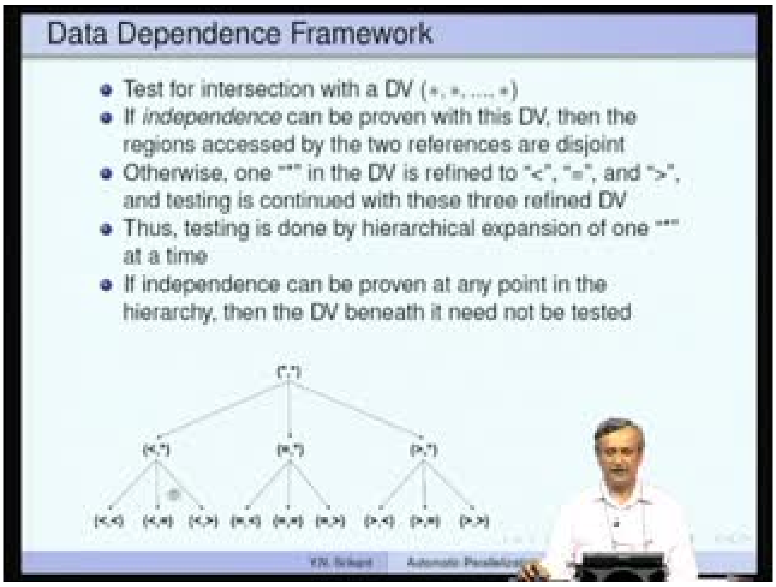
- Given two array references (with s dimensions and nested in loop nest of depth d):
  $S_v : X(f_1(I_1, \ldots, I_d), f_2(\bar{I}), \ldots, f_s(\bar{I}))$
  $S_w : X(g_1(I_1, \ldots, I_d), g_2(\bar{I}), \ldots, g_s(\bar{I}))$
  - We test for both $S_v\ \delta^* S_w$ and $S_w\ \delta^* S_v$ simultaneously
  - The particular type of dependence (δ, $\bar{\delta}$, or δ^o) depends on the position of references (lhs or rhs) and the direction of dependence
- We first test to see if the array regions accessed by the two references intersect
  - Intersection will occur when the subscript functions are equal simultaneously

$$\begin{aligned} f_1(I_1, \ldots, I_d) &= g_1(J_1, \ldots, J_d) \\ f_2(I_1, \ldots, I_d) &= g_2(J_1, \ldots, J_d) \\ \cdots &\ \cdots \\ f_s(I_1, \ldots, I_d) &= g_s(J_1, \ldots, J_d) \end{aligned}$$

Welcome to part 4 of the lecture on automatic parallelization. We looked at data dependence frame work in the last lecture. Let us recapitulate a little bit because it is very important. If you look at the loop, let us say the loop nest is of depth J, array depth d. There are two statements: S v and S w and both of them have an array reference each. So, the first array reference is X; with the subscripts being f 1, f 2, f 3, etcetera, each of which is a function of I 1 to I d the loop counter values. Similarly, S w has another reference to the same array X, with subscripts g 1, g 2, g 3, etcetera, each of which is again is a function of A 1 to I d. The question is whether these two references intercede and access the same memory location. First of all we are not worried about the type of dependence to begin with. Any type of dependence is ok. First of all we determine whether there is dependence at all and then refine it to see if it is delta, delta bar, and delta o etcetera. And in this process, when we find the intersection we also find the direction vector.

So, the subscript functions must be equal simultaneously, only then there is going to be an intersection. For all the dimensions the subscripts must have equal values - f 1 equal to g 1; f 2 equal to g 2, etcetera, f s equal g s. So, this must happen with the same values of I 1 to I d. Otherwise, the intersection cannot happen.

(Refer Slide Time: 02:06)



Data Dependence Framework

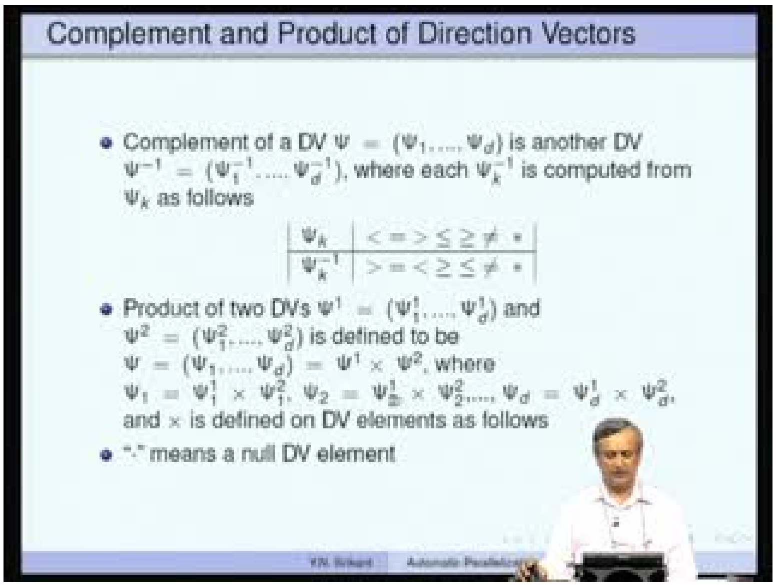
- Test for intersection with a DV (∗, ∗, ..., ∗)
- If *independence* can be proven with this DV, then the regions accessed by the two references are disjoint
- Otherwise, one "∗" in the DV is refined to "<", "=", and ">", and testing is continued with these three refined DV
- Thus, testing is done by hierarchical expansion of one "∗" at a time
- If independence can be proven at any point in the hierarchy, then the DV beneath it need not be tested

Y.N. Srikant | Automatic Parallelization

We start with the most general direction at star, star, etcetera. We can apply the Banerjee's test or the GCD test at this level. So, if we prove independence at this level nothing more to do and there will be no intersection at all. And if there is dependence as proven by the Banerjee or GCD test, one of the stars must be expanded. So, here we have an example, it is star, star; suppose, the dependence is proven to be true we just conservative. So, we expand first star to less than and then tries less than, stars, equal to, star, and greater than, star. So, each of these is going to be tested with Banerjee's test. Depending on which one of these returns independence we do not actually expand the tree. Suppose, this will be obviously always independent, because first components is greater than. So, we do not have to do this re-test at all. But suppose, this also returns independents, then this will also not be expanded further, only this will be expanded (Refer Slide Time: 02:06).

So, then we test each of these three direction writers and we apply Banerjee's test, see which one of these is actually true.

(Refer Slide Time: 03:29)



Complement and Product of Direction Vectors

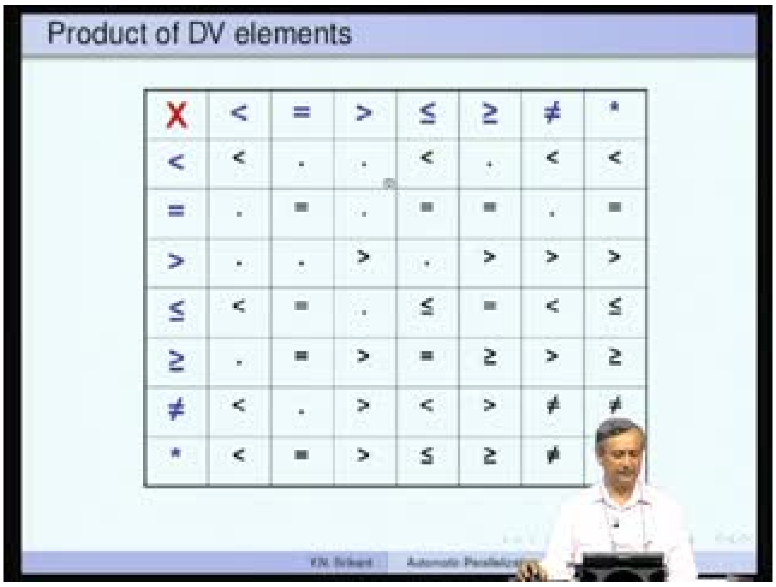
- Complement of a DV $\Psi = (\Psi_1, \ldots, \Psi_d)$ is another DV $\Psi^{-1} = (\Psi_1^{-1}, \ldots, \Psi_d^{-1})$, where each Ψ_k^{-1} is computed from Ψ_k as follows

Ψ_k	$<$	$=$	$>$	$\leq$	$\geq$	$\neq$	$*$
Ψ_k^{-1}	$>$	$=$	$<$	$\geq$	$\leq$	$\neq$	$*$

- Product of two DVs $\Psi^1 = (\Psi_1^1, \ldots, \Psi_d^1)$ and $\Psi^2 = (\Psi_1^2, \ldots, \Psi_d^2)$ is defined to be $\Psi = (\Psi_1, \ldots, \Psi_d) = \Psi^1 \times \Psi^2$, where $\Psi_1 = \Psi_1^1 \times \Psi_1^2$, $\Psi_2 = \Psi_2^1 \times \Psi_2^2, \ldots, \Psi_d = \Psi_d^1 \times \Psi_d^2$, and $\times$ is defined on DV elements as follows
- "." means a null DV element

Y.N. Srikant | Automatic Paralleliz

(Refer Slide Time: 04:06)

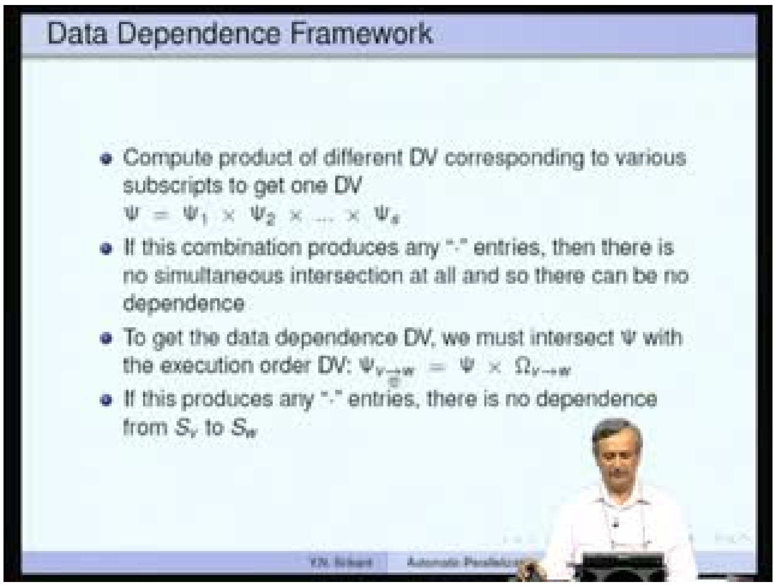


Product of DV elements

X	$<$	$=$	$>$	$\leq$	$\geq$	$\neq$	$*$
$<$	$<$	.	.	$<$	.	$<$	$<$
$=$	.	$=$	.	$=$	$=$	.	$=$
$>$	.	.	$>$	.	$>$	$>$	$>$
$\leq$	$<$	$=$	.	$\leq$	$=$	$<$	$\leq$
$\geq$	.	$=$	$>$	$=$	$\geq$	$>$	$\geq$
$\neq$	$<$	.	$>$	$<$	$>$	$\neq$	$\neq$
$*$	$<$	$=$	$>$	$\leq$	$\geq$	$\neq$	[illegible]

Y.N. Srikant | Automatic Paralleliz

(Refer Slide Time: 04:21)



Data Dependence Framework

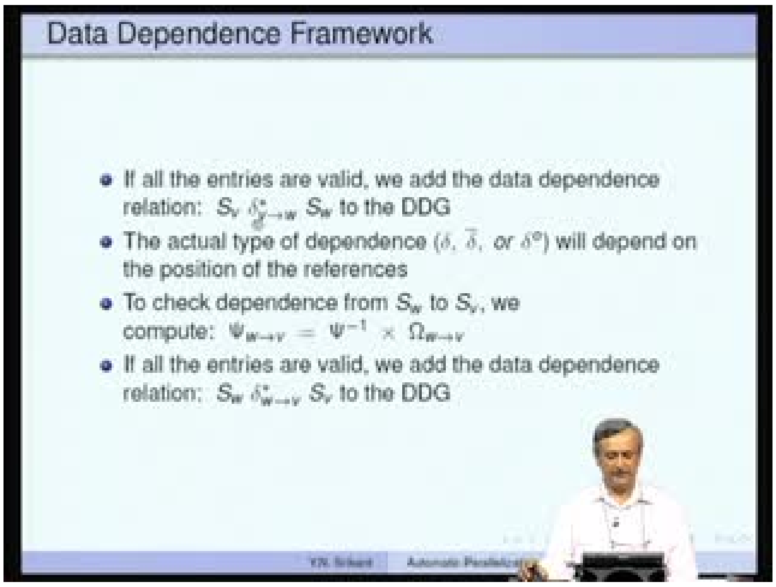
- Compute product of different DV corresponding to various subscripts to get one DV
  $\Psi = \Psi_1 \times \Psi_2 \times \ldots \times \Psi_s$
- If this combination produces any "·" entries, then there is no simultaneous intersection at all and so there can be no dependence
- To get the data dependence DV, we must intersect Ψ with the execution order DV: $\Psi_{v \xrightarrow{=} w} = \Psi \times \Omega_{v \to w}$
- If this produces any "·" entries, there is no dependence from S_v to S_w

We also define the complement and product of direction vectors. These are needed to compute psi inverse and psi 1 cross psi d, etcetera. So, if psi k is this and then psi k inverse is defined to be the complement if it is less than, then psi k inverse is greater than etcetera. And with the product, we simply take the two direction vectors psi 1 and psi 2; take the products of the components, where the product of the components is defined in this table. So, one of them is less than the other, one is less than it is less than, if it is less than and equal to it is nil, that means, it is illegal type of direction vector at this point and so on. Once, we know this, how you use it? We actually compute the product of different direction vectors belonging to the various subscripts; various dimensions have various subscripts; so, we compute the product which really indicates intersection of all this direction at us. So, that is done psi 1 cross psi 2 etcetera.

If this combination produces any dot entries then, there is no simultaneous intersection of all these subscripts. We can say there is no dependence and they will never intersect. But if the all the entries are valid we need to find the actual dependence. So, this can be done by intersecting the direction vector that we get here; with the execution order direction vector for that particular construct. Depending on which type of a loop it is, you would have computed the direction vector psi of v to w bye looking at the syntax of the loop. So, we intersect it with rather the omega; omega v to w is the execution order direction vector. So, we intersected its psi to get the direction vector from v to w that is - we are resuming that know the dependence is from S v to S w because, this is dictated by the omega.

So, if this provides any nil entries then, there is no dependence from S v to S w. Otherwise, whatever is given as the direction vector will give you the dependence from S v to S w.

(Refer Slide Time: 06:06)

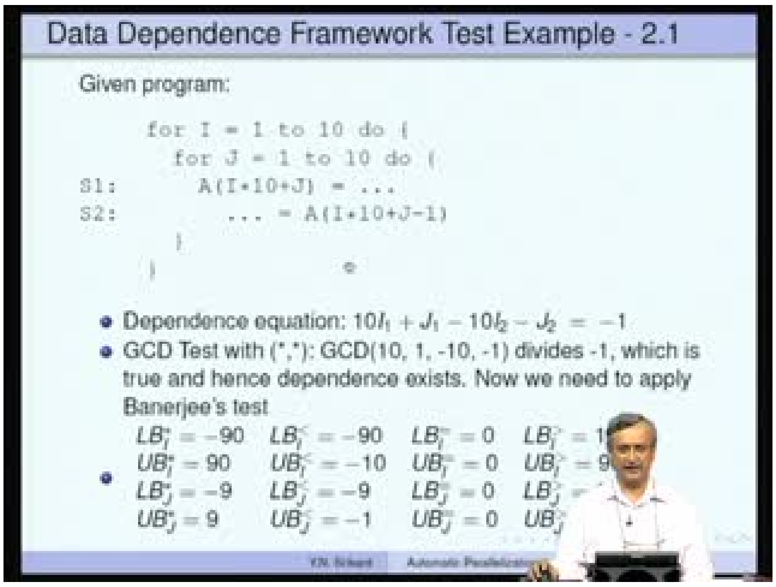


Data Dependence Framework

- If all the entries are valid, we add the data dependence relation: $S_v\ \delta^*_{\psi\to w}\ S_w$ to the DDG
- The actual type of dependence ($\delta,\ \overline{\delta},\ or\ \delta^o$) will depend on the position of the references
- To check dependence from S_w to S_v, we compute: $\Psi_{w\to v} = \Psi^{-1} \times \Omega_{w\to v}$
- If all the entries are valid, we add the data dependence relation: $S_w\ \delta^*_{w\to v}\ S_v$ to the DDG

Y.N. Srikant Automatic Paralleliza

(Refer Slide Time: 07:10)



Data Dependence Framework Test Example - 2.1

Given program:

```
      for I = 1 to 10 do {
        for J = 1 to 10 do {
S1:       A(I*10+J) = ...
S2:         ... = A(I*10+J-1)
        }
      }
```

- Dependence equation: $10I_1 + J_1 - 10I_2 - J_2 = -1$
- GCD Test with (*,*): GCD(10, 1, -10, -1) divides -1, which is true and hence dependence exists. Now we need to apply Banerjee's test
- $LB^*_i = -90$ $\quad LB^<_i = -90$ $\quad LB^=_i = 0$ $\quad LB^>_i = 1$
  $UB^*_i = 90$ $\quad UB^<_i = -10$ $\quad UB^=_i = 0$ $\quad UB^>_i = 9$
  $LB^*_j = -9$ $\quad LB^<_j = -9$ $\quad LB^=_j = 0$ $\quad LB^>_j$
  $UB^*_j = 9$ $\quad UB^<_j = -1$ $\quad UB^=_j = 0$ $\quad UB^>_j$

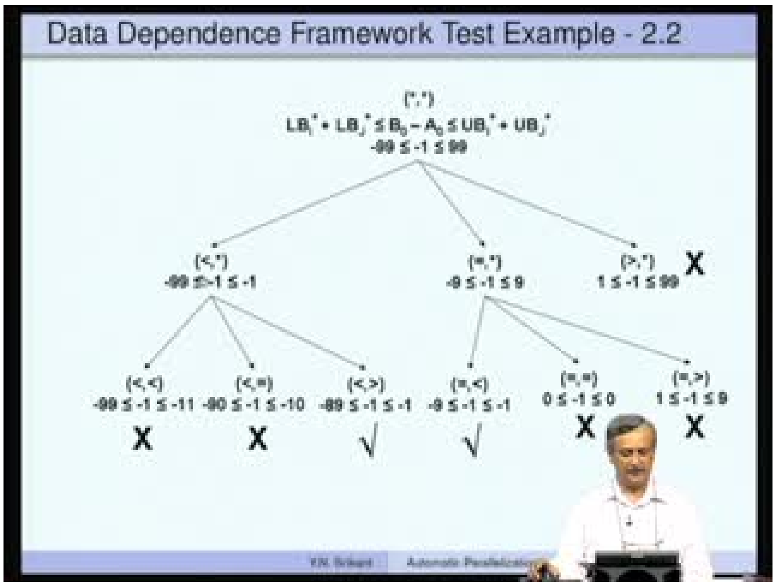
Y.N. Srikant Automatic Paralleliza

In that case, we still have not found the type of dependence delta, delta bar and delta o that is - actually decided by looking at the position of the 2 statements. If the flow dependence means S v is on left side; and S w is on right side; so on and so forth. So, if some of the entries are nil then, we need to check whether there is dependence from S w to S v. We compute psi inverse and then intersect it with omega w to v, w to v again would have been computed using the execution order direction vector that is what this is. So, we would have computed this as the direction vector of the execution order using the syntax of the construct. So, psi inverse we know how to compute? The direction vector from w to v now, if all the entries are valid we add S w delta star w to v and which type of dependence that is, decided by the position of the 2 references. Now, let us look at the second example. We saw one example last time. So, here there are 2 loops, but A is a single dimensional array, it uses both loop holders. So, the expression is i star 10 plus j on the left side; and I star 10 plus J minus 1 on the right side; so, the dependence equation is 10 I 1 plus J 1 minus 10 I 2 plus J 2 minus J 2 equal to minus 1.

That would be transferred to the left side that is, why this becomes 10 I 1 plus J 1 minus 10 I 2 minus J 2 and this remains on the right side; so that is minus 1. Now, the GCD test says s dependence exists and that would be with the star comma star.

Now, we need to apply the Banerjee's test and the dependence frame work. This is how we compute all the required lower and upper bounds for star, less than, equal to, greater than etcetera for both the indices I and J. So, the formulas we looked at in the last class itself.

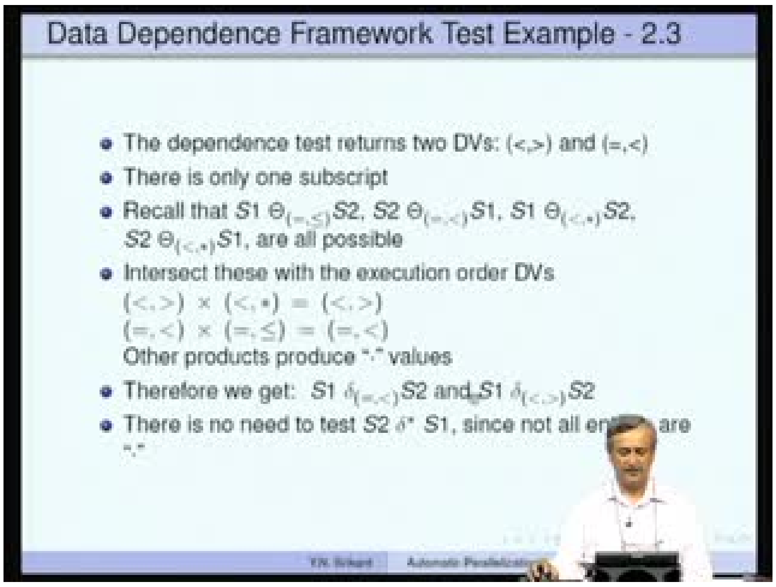
(Refer Slide Time: 08:26)



Data Dependence Framework Test Example - 2.2

So, once we compute those, we can check the constraints at the various levels. At this level, the constraints are some of the l B's of I J with star as the direction vector; must be less than or equal to B not minus a not; that must in turn will less than or equal to the some of the u B's of I N J with star as the direction vector. So, this will be easily satisfied so, we can see that. So, we expand the left. The left star so, we have less than star, equal to star, and greater than star; this is not a feasible thing not possible so, we stop at this point. Whereas here, this returns to minus 99; less than or equal to minus 1; that is true. So, we expanded the second star again and checked to all the 3 dependences using Banerjee's test. Two of them returned falls and one returned yes so, here again this returned yes and the other 2 returned false. So, these are the only 2 direction vectors, which may be possible less than, greater than and equal to less than.

(Refer Slide Time: 09:31)



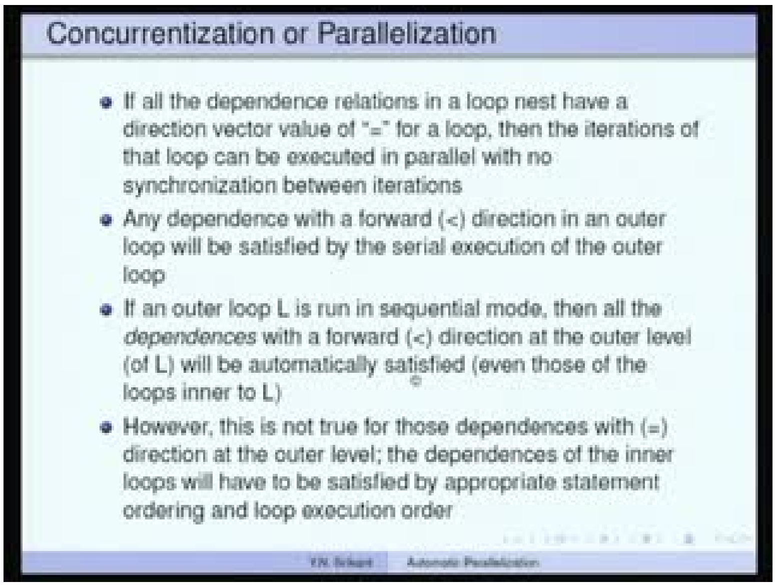
Data Dependence Framework Test Example - 2.3

- The dependence test returns two DVs: (<,>) and (=,<)
- There is only one subscript
- Recall that $S1\ \Theta_{(=,\leq)} S2$, $S2\ \Theta_{(=,<)} S1$, $S1\ \Theta_{(<,*)} S2$, $S2\ \Theta_{(<,*)} S1$, are all possible
- Intersect these with the execution order DVs
  $(<,>) \times (<,*) = (<,>)$
  $(=,<) \times (=,\leq) = (=,<)$
  Other products produce "-" values
- Therefore we get: $S1\ \delta_{(=,<)} S2$ and $S1\ \delta_{(<,>)} S2$
- There is no need to test $S2\ \delta^{*}\ S1$, since not all en are "-"

Y.N. Srikant | Automatic Parallelization

It is not yet decided. Now, there is only 1 subscript. So, there is no need to do any intersection of the 2 subscripts at all since there is only 1. Now, the execution order dependence direction vectors are all given for that single loop. We know that - S 1 theta equal to less than or equal to S 2; S 2 theta equal to less than S 1; S 1 theta less than star less than comma star S 2; S 2 theta less than comma star S 1 are all possible.

(Refer Slide Time: 11:34)



Concurrentization or Parallelization

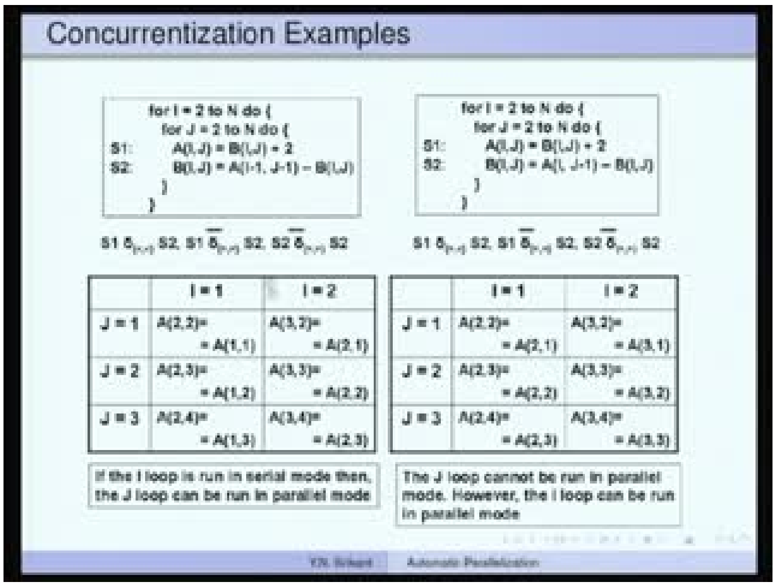
- If all the dependence relations in a loop nest have a direction vector value of "=" for a loop, then the iterations of that loop can be executed in parallel with no synchronization between iterations
- Any dependence with a forward (<) direction in an outer loop will be satisfied by the serial execution 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 (=) direction at the outer level; the dependences of the inner loops will have to be satisfied by appropriate statement ordering and loop execution order

Y.N. Srikant | Automatic Parallelization

So, we are going to actually look at less than, greater than. So, any of this with first component less than will be useful, anything with something other less than will return a null entry. So, less than comma star; S 1 theta less than comma star; S 2 is possible; so when we intersect, we just get the same less than, greater than. So, when we look at equal to comma less than again, unless the first 1 less than or equal to; or equal to there is no point in intersecting it; so we have equal to less than or equal to here, S 1 theta equal to less than or equal to that uses the same vector equal to and less than. So, all other products are dot values, but there are two of them which are not dot. So, these are indeed the dependences with this direction vectors. We have possibly S 1 delta equal to comma less than S 2; and S 2 delta less than comma greater than S 2. So, the position of the 2 references already indicates see for example, S 1 has this on the left side and S 2 as this on the right side. It indicates that - it is a delta not delta r or delta o and since this has been proven true; there is no need to test S 2 delta star S 1. So, that is how we use the dependence frame work and Banerjee's test to compute the dependences in more complicated expressions and loops. So, that is about the dependence framework. Now, let us see what can be done about parallelization, concurrentization etcetera.

So, this statement I already mentioned few lectures ago. If all the dependence relation in a loop enhanced have direction at value of equal to in a loop then, the iterations of that loop can be executed in parallel with no synchronization between the iterations. So, each iteration within the iteration the order of statements is going to be the same as in the program, we are not going to change the order of the statements unnecessarily. So, if we do that if since all their dependences at that level are equal to, we can simply execute that loop in parallel no violations will occur. This is the very performed result, even though it looks non-intuitive any dependence with a forward direction vector in an outer loop will be satisfied by the serial execution of the outer loop. So, if an outer loop l is run in sequential mode then, all the dependences with the forward direction at the outer level will be automatically satisfied. So, if this is even for the inner loops of l, I will show you an example for this, but if the outer loop has dependence equal to then whatever is said hear will not be true. Even if the outer loop is run in sequential mode the dependences may not be completely satisfied.

(Refer Slide Time: 13:18)



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

Y.N. Srikant | Automatic Parallelization

Let us take this example, I equal to 2 to N then, J equal to 2 to N, S 1 has A I J equal to B I J plus 2; and S 2 has B I J equal to A I minus 1 J minus 1 minus B I J. So, one of some of this dependences are obvious this A I J and this A I J; A minus 1, J minus 1, have a dependence. So, when this is 2, this is 1; when this is 3, this is 2; etcetera. Similarly, when this is 2, this is 1; this is 3; this is 2; etcetera. So, both this dependences are carried forward so, S 1 delta less than S 2 is because of these 2 references. Then we have this and this, the indices are the same; the subscripts are the same so, this will have actually anti-dependence with S 2, S 1 delta bar S 2, with both the direction extra components being equal to and equal to. Finally, we have this and this in same S 2 so, this is B I J; this is also B I J so, this is read first and then return in to so, it is an anti-dependence from S 2 to S 2 with equal to and equal to. Let us expand the loop. So, we get A (2, 2) equal to A (1,1) equal to etcetera and for various values of J under various values of I. So, we have a less than in the outer loop between S 1 and S 2 so, that dependence has a less than. Whereas for the others S 1 and S 2, it is equal to and S 2 and S 2 it is also equal to.

So, this I loop cannot be run in parallel, it should have been equal to. Now suppose, we run this I loop in sequential mode; so, even the J loop cannot be run in parallel because, S 1 delta less than S 2 has the second loop, second component also as less than. But let us observe something. Let us run the I loop in sequential mode that means, I loop will run like this with J equal to 1; J equal to 2 etcetera; and then I equal to 2 will be run etcetera; I equal to 3 and so on and so forth.

Let us see if we run I loop sequentially and if you run the J loop in parallel, we have J equal to 1; J equal to 2; J equal to 3 etcetera all of them running in parallel. That is what it really means. So, all these will run in parallel, but these will begin only after these complete. So, that is sequential for I parallel for J. So, that happens if the dependences get satisfied.

So, between S 1 and S 2 there is delta less than so, let us see where that is. So A (2, 2) is here and A (2,2) is here. So, since this is going to run first and then this, automatically this will be computed first and then this would be computed and this particular dependence would be automatically satisfied. See that even if run JS in parallel, it would be only here and not across. So, this will be computed and then use and this particular dependence would be automatically satisfied. This poses no problem because; it is equal to equal to in both directions. So, we have very little to worry about S 1 delta bar S 2 and that would be only between S 1, S 2 in the same iteration of I and J. So, that would be automatically satisfied. Then this is also between S 2 and S 2 in the equal to equal to so, that means it is in the same iteration of I and J. So, this will be also be automatically satisfied even if we run J in parallel it will there will be nothing wrong.

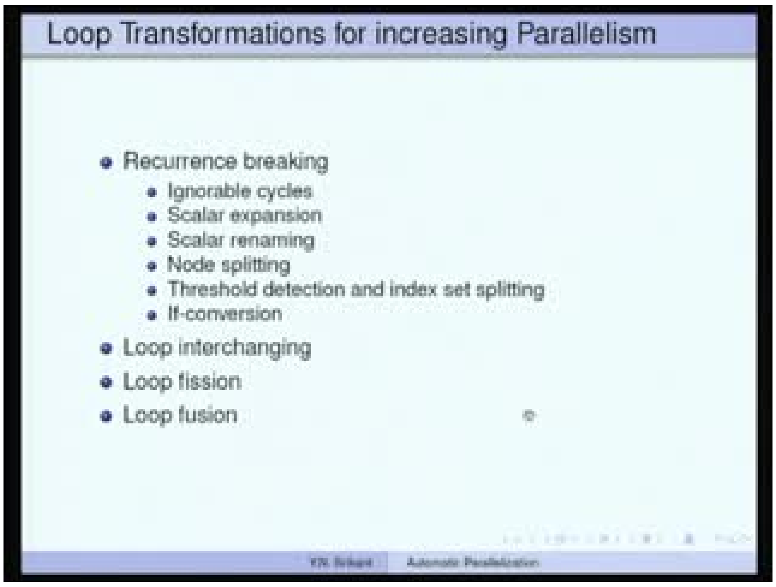
So, when we ran I loop in sequential mode; the second component which is less than also got automatically satisfied that is, what really said here? So, any dependence with less than forward less than direction vector in an outer loop will be satisfied by the serial execution of the outer loop that is obvious. But then the second one said, if we run the outer loop in sequential mode then, all other inner loops can be run in parallel mode; provided the outer loop had a less than direction vector.

So, that is true. The outer loop had less than; inner loop also had less than, but once we ran the outer loop in sequential mode; inner loop could also be run in parallel mode. So, that is very important. Let us look at another example. Here we have I equal to 2 to N; and J equal to 2 to N; so, A I J is B I J plus 2; B I J is A I comma J minus 1 minus B I J. So, we have between S 1 and S 2; we have dependence between these 2 that is - S 1 delta equal to coma less than S 2 that is, between these 2; I and I are the same, J and J minus 1 give less than.

S 1 delta bar S 2 and S 2 delta bar S 2 remain the same as before, no change. Now, we can run the I loop in parallel because, the first component is equal to in all of them no problem, but the second loop cannot be run in parallel. The second loop has less than as the direction vector component. What I want to show you here is, even if we run the I loop in sequential mode; you cannot run the J loop in parallel mode; that is, what really is the problem.

So, let us say we run the I loop in sequential mode; so, this gets run first and then this and so on and so forth. So, the dependence S 1 delta I equal to comma less than S 1 is here, it is between this and this. So, even if we run I equal to 1 and then I equal to 2 etcetera. If we run J 1, J 2, J 3 etcetera in parallel this dependence will be violated. That is the reason why we cannot run the J loop in parallel mode; and it does not matter whether we run the outer loop in parallel mode or sequential mode. So, if we run the I loop in parallel mode then, these iterations get in parallel and the J loops will all be run in sequential mode so, automatically the dependence will be satisfied.

(Refer Slide Time: 20:22)



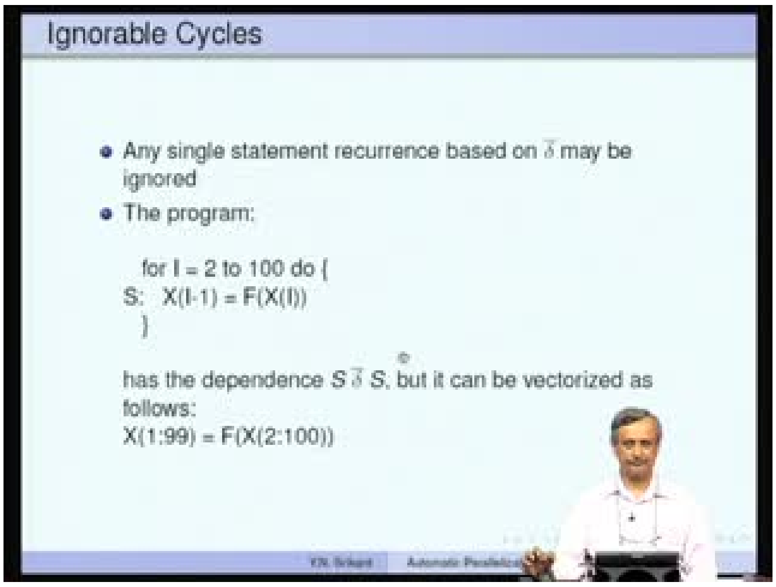
Loop Transformations for increasing Parallelism

- 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 some of the loop transformations for increasing parallelism. What we have seen so far, or to check whether existing loops can be run in parallel mode or in sequential mode. We did not propose any transformations, which can change the loop and make it possible to run it in parallel mode. So, we know that cycles that is recurrences create problems. So, recurrence breaking is 1 of the loop transformations. So, within that there are many of them. Ignorable cycles, scalar expansion, scalar renaming, note splitting, threshold detection and index set splitting, if conversion. We will see one example of each. Then loop interchange is a very important transformation, loop fission and finally, loop fusion.

(Refer Slide Time: 21:29)



Ignorable Cycles

- Any single statement recurrence based on $\bar{\delta}$ may be ignored
- The program:

```
  for I = 2 to 100 do {
S:  X(I-1) = F(X(I))
  }
```

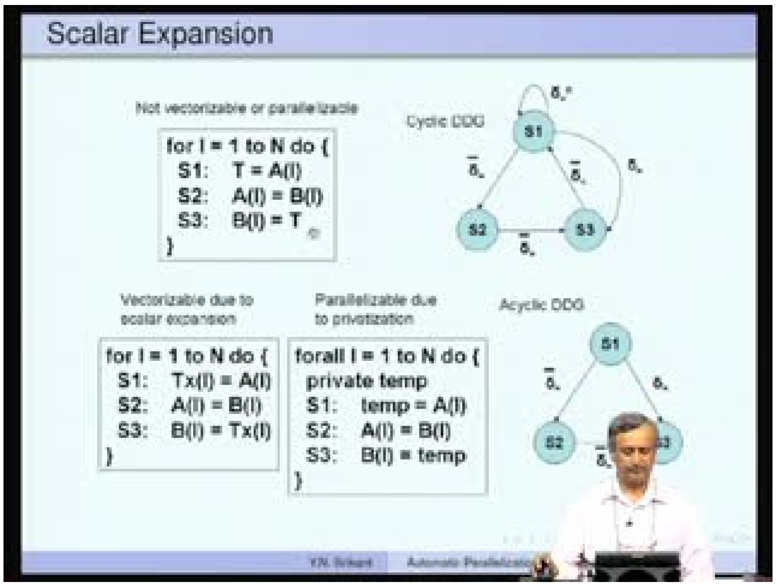
has the dependence $S \overset{\oplus}{\bar{\delta}} S$, but it can be vectorized as follows:
X(1:99) = F(X(2:100))

What are ignorable cycles? It says any single statement recurrence based on delta may be ignored. Let us take this example, X of I minus 1 equal to F of X of I; I runs from 2 to 100. So, what has happened is, if you expand this; this becomes X 2 equal to F of X 1; X 3 equal to F of X 2 etcetera. So, that means, there is a dependence so, you will have to read this first and then you know this will be X. Let us say, I equal to 2; so, this becomes X 1 right. This is X 1 and this is X 2, this is X 3, this is for I equal to 2, 3, this becomes X 2 and this becomes X 3 and so on and so forth.

So, we would have read X I first and then return into X I so, X 2 is read then return in to the next iteration. So, if this happens there is a dependence from S to S, but that is an anti-dependence. So, what this statement says is if it is an anti-dependence on the same statement then, it can be ignored; that is because even if it is vectorized the semantics of the statement say, the right hand side must be fetched first, evaluated first and then the left hand side should be return in to and that is precisely what this is really saying.

So, there is nothing wrong in, observe that - this is reading ahead and then writing it to it. Nowhere does it actually read something, which has been computed in some other cycle. So, because of that this is perfectly ok. X 2 to 100 reads all the values; puts them in some buffer; and then computes the function f on it and finally, assigns X 1 to 99 to these are assigned these values. So, there no violation of any dependence here that is why, any single statement recurrence based on delta may be ignored, but it must be delta, delta bar may be ignored.

(Refer Slide Time: 23:51)



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
}
```

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
}
```

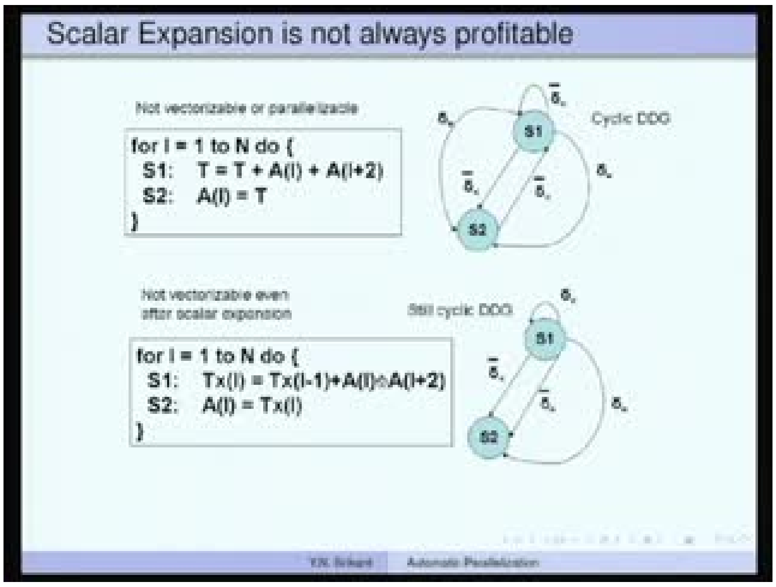
What is scalar expansion? Again it is very intuitive. So, let us take this loop, I equal to 1 to N; S 1 is to t equal to A I; S 2 is A I equal to B I; and s 3 is B I equal to T; T is a scalar, A and B are arrays. Now, it is easy to see that from the dependence diagram S 1 to S 3 there is flow dependence. So, S 1 to S 3; there is a S 1 to S 3; there is a flow dependence T is computed here and T is used here, but then if you expand this particular loop once, there will be another S 1, S 2, S 3 here; and the next iteration I equal to 2 would have S 1 with T equal to A of 2. So, this T and the next T would be tight together; there is dependence that would be anti-dependence. This is reading here and that is righting there. So, we have from S 3 to S 1 an anti-dependence with less than. So, this was I equal to 1; the next 1 would be I equal to 2.

So, that is another. Then we have a dependence from S 1 to S 1. Again look at the expanded version of this, there would be another S 1, S 2, S 3 with T. So, that T; the T which is in the next S 1 and the T in this S 1 would have an output dependence and so, that is - what is shown here. And obviously, between S 1 and S 2 there is an anti-dependence and between S 2 and S 3 there is another anti-dependence. So, this is your diagram. It has many cycles 1 here and another here. So, this cannot be vectorized that is very clear.

Now, suppose we do not want to this problem and we replace T with a vector. So, let us say, T becomes T X so, this is called scalar expansion. T is a scalar; it was expanded to a vector. So, instead of using same t again and again we are using different locations of the array T X. So, instead of T equal to I; we have T X of I to A I and instead B I equal to T, which is supposed to use the same value of T we have B I equal to T X of I. The advantage with this is some of the dependences go away so for example, from S 1 to S 2 the dependence between this A I's remains the dependence between from S 2 to S 3 between the B I's also remains, but when you look at the output dependence on S 1 it disappears because, we are not going to write into the same location again and again. This is T X of I; it writes into T X 1, 2, 3, 4 etcetera, it never writes into the same location. So, this particular loop that the self-dependence has gone. Then we also do not have the output dependence from S 3 to S 1. Remember B I equal to t and then the next S 1 would have t equal to A I so, that anti-dependence was present here, but that does not exist here because, the same location is never return into in I equal to it would have be the T X of 2.

So, this S 3 to S 1 disappears. What we have is just S 1 to S 3 with delta equal to so, that is because this and this are the same; so this is the delta equal to that we are talking about. This code can be vectorized absolutely no problem. But then, during parallelization we do not need such vectors. What we really do is each of this iteration is going to run as a thread that is what re parallelization is all about. So, we say that at this variable T which is here, is a thread private variable called temp. In such a case, the program remains as it is temp equal to A I; A I equal to B I; and B I equal to temp; but for I equal to 1 it is temp 1; which is private to the thread for I equal to 1 and for I equal to 2; it would be temp 2 which is deferent from temp 1 and temp 2 is private to the thread owning I equal to 2. So, that is how this for all loops executes with temp as private to the thread corresponding to a particular iteration value. If that is so, this particular diagram still holds so, we do not have any cycles and it can be run in parallel mode.

(Refer Slide Time: 29:01)



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
S1
S2
$\bar{\delta}$
δ

Not vectorizable even after scalar expansion

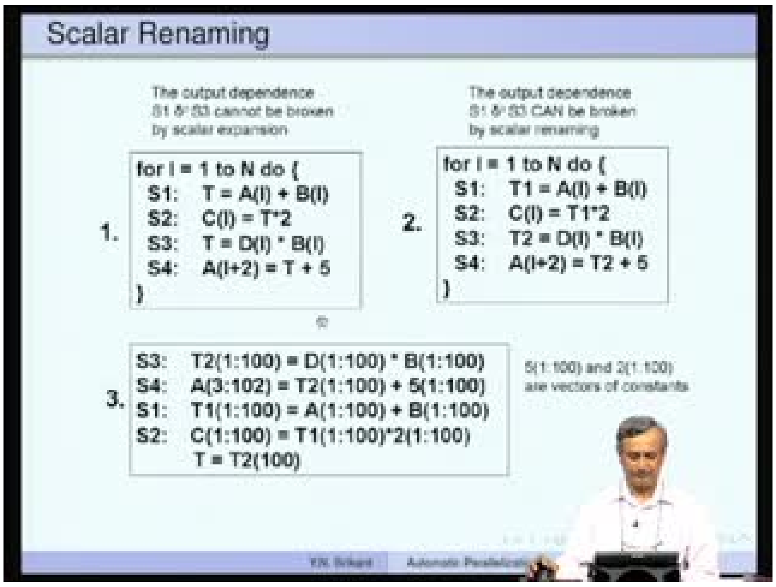
```
for I = 1 to N do {
  S1:   Tx(I) = Tx(I-1)+A(I)+A(I+2)
  S2:   A(I) = Tx(I)
}
```

Still cyclic DDG
S1
S2
$\bar{\delta}$
δ

But then scalar expansion is not always profitable or beneficial. Let us look at an example, I equal to 1 to N; S 1 is T equal to T plus A I plus A I plus 2 and S 2 is A I equal to T. So, if you draw the dependence diagram for this particular loop so, of course, we have the anti-dependence between these 2 T's so, that is here. So, what was computed in the previous iteration is used here. Then we have many other dependences. So, between S 1 and S 2 we have this anti-dependence A I and A I so, between S 1 and S 2 there is an anti-dependence so, that is from here. Then we have A I plus 2 and A I so, this is if you have A I equal to 1 you would have A 3, A 1; A 4, A 2; A 5, A 3; and so on and so forth.

So, that is a anti-dependence from S 1 to S 2 with less than as the direction vector S 1 to S 2 with anti-dependence that is another. Then of course, you have this dependence from S 1 to S 2 which is a flow dependence with equal to direction vector. So, that is also from S 1 to S 2 so, that already exists. So, that is here. Then we have this so, there are many dependences this is if you expand this we have another S 1, S 2 so, there would be another flow anti-dependence from S 2 to S 1 as well. So, we have this dependence from S 2 to S 1 so, that is - anti-dependence with less than direction vector. So, these are the various dependences that we have in this particular program. Obviously, we have a cycle so, it is not vectorisable. Suppose, we do scalar expansion that is, expand this T to an array, you get T x of I equal to T x of I minus 1 plus A I plus A I plus 2; A I equal to T x of I. So, with this we get this particular dependence diagram that is - you have these some of this dependences remain the anti-dependence is the problem here. So, we have T x I equal to T x of I minus 1 plus something. So, we still have dependence from T x of I minus 1 to T x S 1 to S 1 and now this becomes a delta less than. It was actually a delta bar before, but now because it is an array we are using the old value and computing a new value here. So, this becomes a delta less than; so, there is another loop here. Otherwise, the previous loop went away the delta equal to remains; delta bar equal to remains; delta bar less than remains, but then this also remains, but you should observe that this particular S 2 to S 1 actually has reversed. So, S 2 to S 1 there is no dependence any more that is - the dependence from S 1 to S 2 with equal to as the direction vector. It was less than before, but now it is equal to; that is - the difference between this loop and this loop; some of these dependences have changed; some of them have a vanished, but this loop has changed to a reverse actually, it is delta less than; and it was delta bar less than before. So, even after variable scalar variable expansion, we still have a loop and this loop cannot be vectorized. So, even those scalar expansion is always legal it may not be profitable. So, that is the moral of this particular slide.

(Refer Slide Time: 33:37)



Scalar Renaming

The output dependence S1 δ° 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° 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)
```

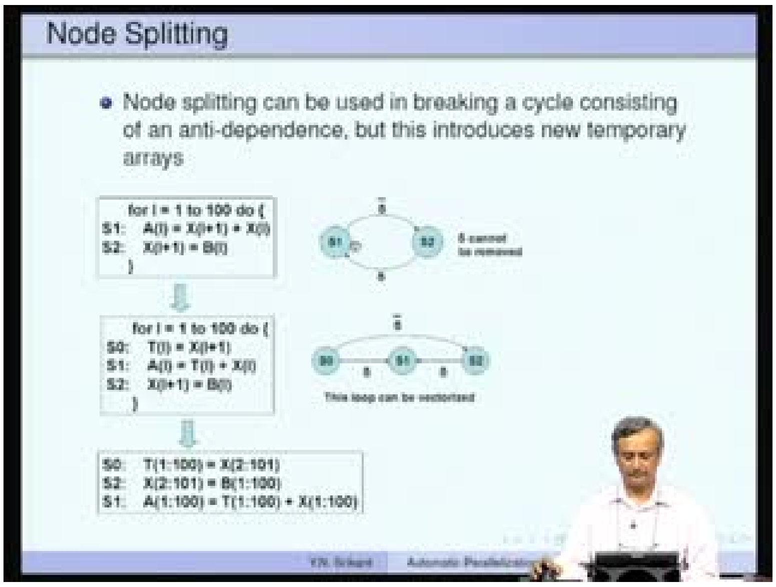
5(1:100) and 2(1:100) are vectors of constants

Y.N. Srikant   Automatic Parallelization

The next one is scalar renaming. So, you have this program with S 1, S 2, S 3, S 4. Now, you have an output dependence from S 1 to S 3; T equal to something and T equal to something here also, the output dependence S 1 delta o S 3 just cannot be broken by scalar expansion. Why? You have T I equal to something here and T I equal to something here so, they both write into the same location of the array, output dependence does not vanish at all. The trick here is we should actually, use a different temporary and not use the same temporary so, the first one becomes T 1; the second one becomes T 2; so, this dependence is from T 1; T 1 here and the second dependence is from T 2 to T 2. Once, we do that this particular program can benefit by scalar expansion. So, you now make T 1 into a vector and we also make T 2 in to a vector. Automatically, the scalar program can be converted to a vector program with T 2, 1 to 100 equal to something etcetera.

So, why is T 2 coming first and why is S 3, S 4 with order why is it in this order S 3, S 4, S 1, S 2. Well that is just that the dependences have to be satisfied appropriately and that is the reason why we are. So, this S 4 is computing something and then supplying it to S 1 that is - the reason why S 3 and S 4 come first and then, because S 1 and S 2 are dependent on it they come later. Finally, the value of 40 is T 2 of 100, which should be retained as it is. So scalar renaming and scalar expansion are two different transformations and they may be applied that together as well.

(Refer Slide Time: 35:38)



Node Splitting

- Node splitting can be used in breaking a cycle consisting of an anti-dependence, but this introduces new temporary arrays

```
    for I = 1 to 100 do {
S1:    A(I) = X(I+1) + X(I)
S2:    X(I+1) = B(I)
    }
```

```
    for I = 1 to 100 do {
S0:    T(I) = X(I+1)
S1:    A(I) = T(I) + X(I)
S2:    X(I+1) = B(I)
    }
```

```
S0:   T(1:100) = X(2:101)
S2:   X(2:101) = B(1:100)
S1:   A(1:100) = T(1:100) + X(1:100)
```

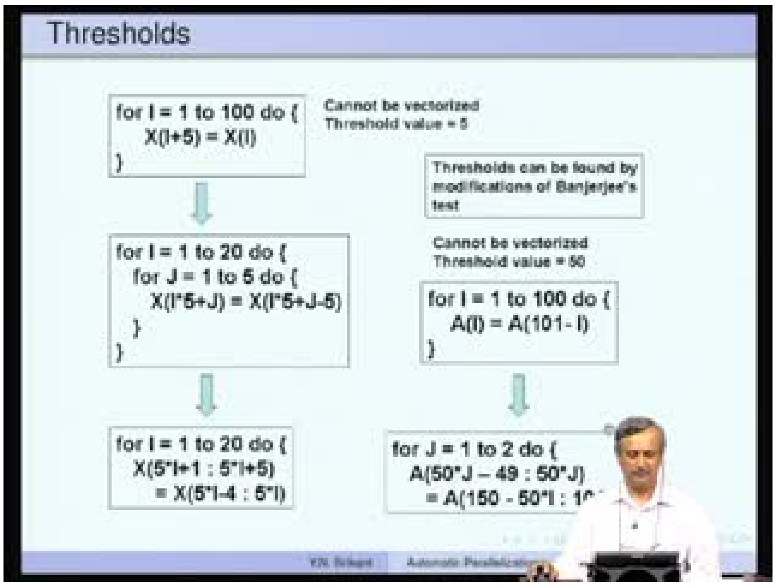
$\bar{\delta}$
S1
S2
δ
$\bar{\delta}$ cannot be removed
S0
S1
S2
$\bar{\delta}$
δ
δ
This loop can be vectorized
Y.N. Srikant
Automatic Parallelization

Then, we come to node splitting. So suppose, there is a cycle consisting of an anti-dependence. If we split a node, we may be able to get away with this dependence and then, we may be able to actually vectorize the program. But then, we are going to introduce new array so, there is a space expression possible. So, let us take an example, I equal to 1 to 100; so, S 1 is A I equal to X I plus 1 plus X I; and S 2 is X I plus 1 equal to

B I so, there is a dependence cycle between these 2. So, S 1 to S 2; there is a delta bar so, X I plus 1 is read here and X I plus 1 written into so, that is the delta bar cycle. Between S 2 and S 1 this is writing in to X 2; this is reading from X 1 then, this writes in to X 3 and this reads X 2 that means, there is a dependence of the flow type between S 2, S 1. So, there is a dependence from S 2 S 1 the delta, this is a cycle. We can never get rid of delta anyway. So, if we break this into S 1 into 2 let us see what happens.

We get s zero. T I equal to X I plus 1 so, this part is assigned to a temporary. Then we have A I equal to T I plus X I so, instead of X I plus 1 then, X i plus 1 equal to B I. So, we have broken S 1 in to 2 statements: S 0 and S 1. Let us see what happens here. So, between S 0 and S 1 there is an anti-dependence for the T I; S 0 flow dependence between this T I and this T I so, that is here, and then we have from S 0 this particular X I plus S 1 and this X I plus have an anti-dependence so, that remains as it is. And then we have the between S 1 and S 2; we have a flow dependence from S 2 to S 1; so, because we broke this node in to 2 now, the dependence cycle has vanished. So, there is no cycle both this dependences from S 0 and S 2 to S 1 actually, are in the same direction towards S 1. So, there is no cycle at all and this loop can be vectorized now.

(Refer Slide Time: 38:39)



Thresholds

```
for I = 1 to 100 do {
    X(I+5) = X(I)
}
```

Cannot be vectorized
Threshold value = 5

```
for I = 1 to 20 do {
   for J = 1 to 5 do {
      X(I*5+J) = X(I*5+J-5)
   }
}
```

```
for I = 1 to 20 do {
   X(5*I+1 : 5*I+5)
      = X(5*I-4 : 5*I)
```

Thresholds can be found by modifications of Banjerjee's test

Cannot be vectorized
Threshold value = 50

```
for I = 1 to 100 do {
   A(I) = A(101- I)
}
```

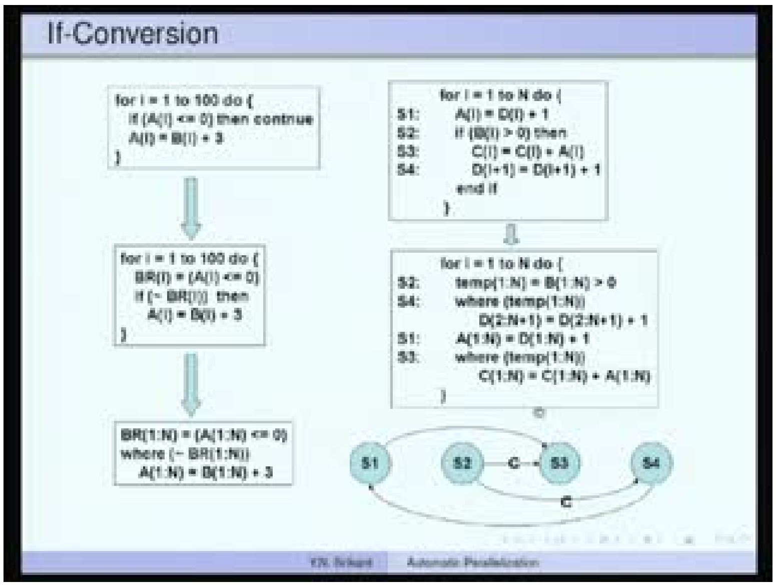
```
for J = 1 to 2 do {
   A(50*J – 49 : 50*J)
   = A(150 - 50*I : 1
```

Y.N. Srikant   Automatic Parallelization

So, we have T 1, 1 to 100 equal to etcetera vectorization works here. So, this is another important in the transformation. Remember if you have an anti-dependence cycle we can try to break it using node splitting. So, if there are thresholds sometimes we detect thresholds they can be used for vectorization as such. So, if you take this program with a loop, which has 1 statement X I plus 5 equal to X I. If we observe it, once you cross the value 5 you would have return X X equal to X 1; X 7 equal to X 2; X 8 equal to X 3 etcetera then, we have X 5 plus 5; X 10 equal to X 5 and then, you get X 11 equal to X 6. So, you have written X 6 in the first iteration and then, in the sixth iteration you are using X 6 that means, there is a flow dependence with this and that flow dependence actually can be somehow gotten rid of by, if we detect that this threshold value is 5. So, we divide the loop in to 2: first loop runs from I to I equal to 1 to 20; second loop runs from J equal to 1 to 5; and inside we have X of I star 5 plus J equal to I star 5 plus J minus 5. So, it is the same loop it is just that we have 2 loop indices here. So, the index values of X here and here will be the same. They will become the same actually, the valued at the same as I plus 5 and I respectively.

Once we have this the J loop can be vectorized. So, there is no dependence at the J level so, we can vectorize the J loop and the I loop provisioned on in sequentially mode. So, we have a very short vector, but still something is better than nothing so, we have reduce this iterations space rather the number of iterations to 20. So, if here the threshold value is 50; I equal to 1 to 100; A I equal to 101 minus I; so, up to 50 the dependence is in 1 direction; after 50 the dependence becomes in the other direction. So, because of that you cannot vectorize as it is, but once you divide the loop in to 2 parts: I equal to 1 to 2; and then J equal to 1 to 50 you can vectorize the J loop and run the I loop in so, rather the other way, you can vectorize the I loop and run the J loop 1 to 2 in sequential mode. So, that is what we have shown here.

(Refer Slide Time: 41:32)



If-Conversion

```
for i = 1 to 100 do {
  if (A(i) <= 0) then continue
  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
        }
```

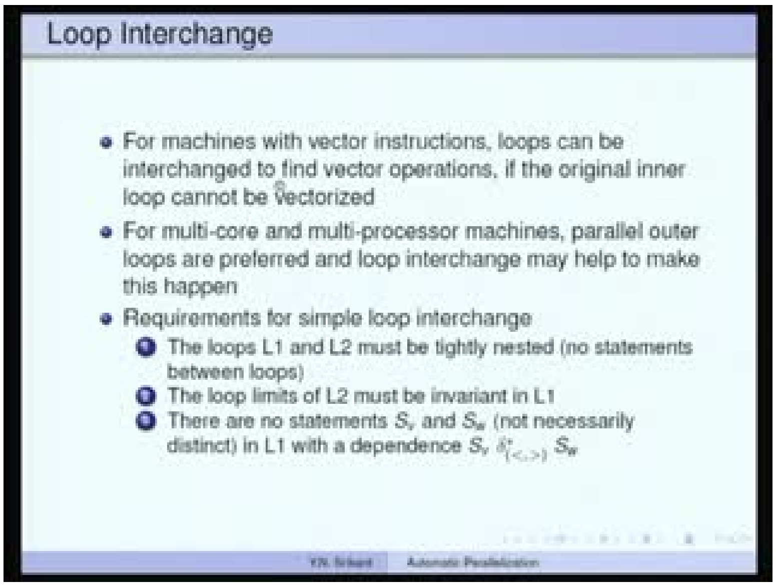
```
        for i = 1 to N do {
S2:       temp(1:N) = B(1:N) > 0
S4:       where (temp(1:N))
               D(2:N+1) = D(2:N+1) + 1
S1:       A(1:N) = D(1:N) + 1
S3:       where (temp(1:N))
               C(1:N) = C(1:N) + A(1:N)
        }
```

So, that is about thresholds. And the next transformation that we consider is the if conversion. So, here the problem is you have conditions in the program and if there are conditions then, the control can flow in any direction if, the condition is true it flow flows in 1 direction and if the condition is false it flows in the other direction. Because of this, there is no way we can actually run in vector mode. So, as it is if there are branches there can be no vectorization possible. A solution to this is to compute the condition also as a vector and for example, here if A I less than equal to 0 then continue; A I equal to B I plus 3; so, if A I is greater than 0, then we execute this otherwise, we just go to the next iteration of the loop. So, what we do is we compute a vector B R I equal to A I less than or equal to 0. So, B R 1 stores a 1 less than or equal to 0; B R 2 stores a 2 less than or equal to 0 and so on. Then we use this B R I as a mask wherever B R I set as true. For example, not of B R I is true whenever B R I is falls and it is falls whenever B R I is true. So, here we wanted to execute A I equal to B I plus 3 whenever, A I is greater than 0 that is, whenever B R I is false. So, we added a mask here. Whenever this particular thing has not of B R I that is, the B R I if it is set is 0 then, the mask for this particular instruction is also set as 1 and the instruction gets executed. So, there must be some hardware support for it. If this B R I set as true not of B R I becomes 0. So, this particular instruction A I equal to B I plus 3; will not be executed. So, there is a mask which executes the instruction if it is set to 1; it does not execute the instruction if it is set to 0. So, with that hardware we can actually, vectorize this particular loop. So, B R 1 to N is a 1 to N less than or equal to 0 so, that is the vector and if not of B R 1 to N so, this is a mask A 1 to N equal to B 1 to N plus 3. So, if this is possible then, this masking is useful. Another bigger example, we have 4 sentences and condition inside with 2 sentences inside the condition.

If this condition is true you want to execute both so, first of all we execute the condition rather compute the condition as a vector. Next we have a masked statement where temp 1: N execute D 2 something so, in vector mode. Then again we have S 1, A 1 to N equal to D 1 n plus 1 and another mask statement where temp 1:N, C 1:N equal to etcetera. So, this reordering of statements is because of this particular dependence diagram where, this C shows control dependence so for example, if B I greater than 0 is the condition, S 3 and S 4 are dependent on it. So, that is why, there is an arc from S 2 to S 3; this is the condition and S 2 to S 4. So, once we have this mask automatically, we can reorder statements assuming that this condition dependence has gone.

So, in other words, this is gone. So, we can execute S 2 first which does not have any predecessors but then, you cannot execute S 3 you will have to execute S 4 which does not have any predecessors now, then S 1 and finally, S 3. So, that is how the conversion of if statements helps in vectorization.

(Refer Slide Time: 46:10)



Loop Interchange

- For machines with vector instructions, loops can be interchanged to find vector operations, if the original inner loop cannot be vectorized
- 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

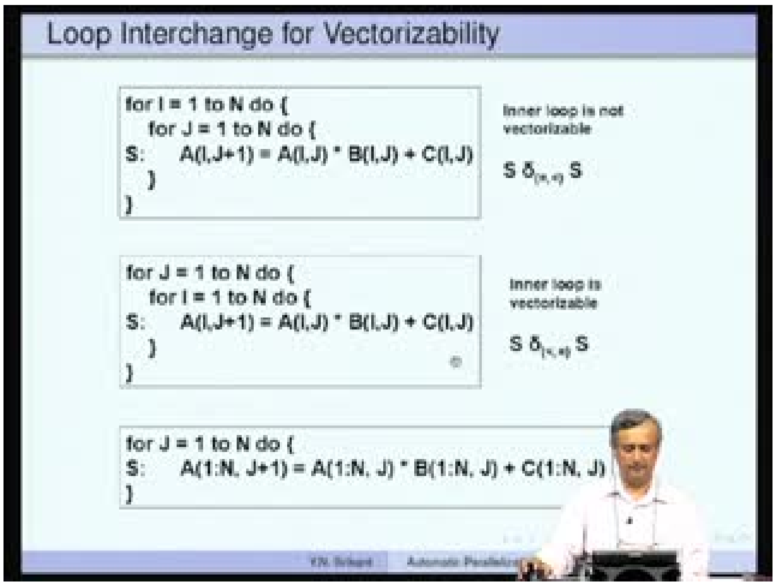
Loop interchange is a very important transformation. So, let us understand loop interchange. What really we do is, if there is a loop I outside and another loop J inside we actually, check whether I or J loop can be run in vector or parallel mode. If it cannot be then, we check a condition and see if the I and J loop can be swapped. So, the J becomes the outer loop and I becomes the inner loop. Now, it may be possible to execute one or more of these loops in vector or concurrent mode.

So, that is the motivation for the loop interchange. Now, there are 3 very simply conditions for simple loop interchange loops L 1 and L 2 must be tightly nested, no statements between the loops. So, all statements are in both L 1 and L 2. There are no statements in L 1 which are in not L 2.

The loop limits of L 2 must be invariant in L 1. So, in other words, you cannot have the inner loop in a limit dependent on the outer loop inducts. There are no statements S v and S w not necessarily distinct in L 1, which is the outer loop with a dependence S v delta star less than comma greater than S w.

So, what really has happened is, if you try to swap the loops L 1 and L 2 this dependence this is intuition reverses. So, we should really get greater than comma less than which is an illegal loop direction vector. And that is the reason why we should not have such a dependence in the first place if you want to do loop interchange.

(Refer Slide Time: 48:06)



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)
   }
}
```

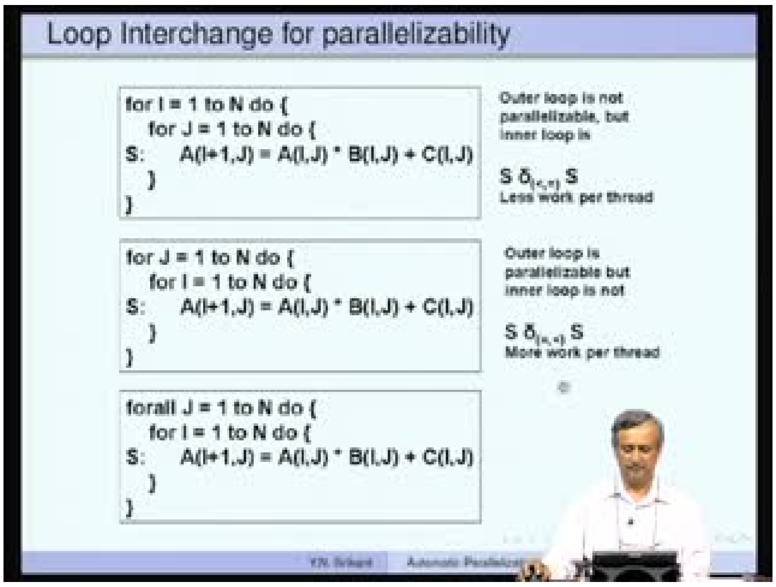
Inner loop is vectorizable

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

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

For example, in this loop the inner loop is not vectorisable. So, we have this dependence S to S with less than the inner loop. Now, if we swap the 2 loops I becomes the inner loop and J becomes the outer loop the dependence changes less than and equal to. Now, the inner loop is definitely vectorisable.

(Refer Slide Time: 48:34)



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)
   }
}
```

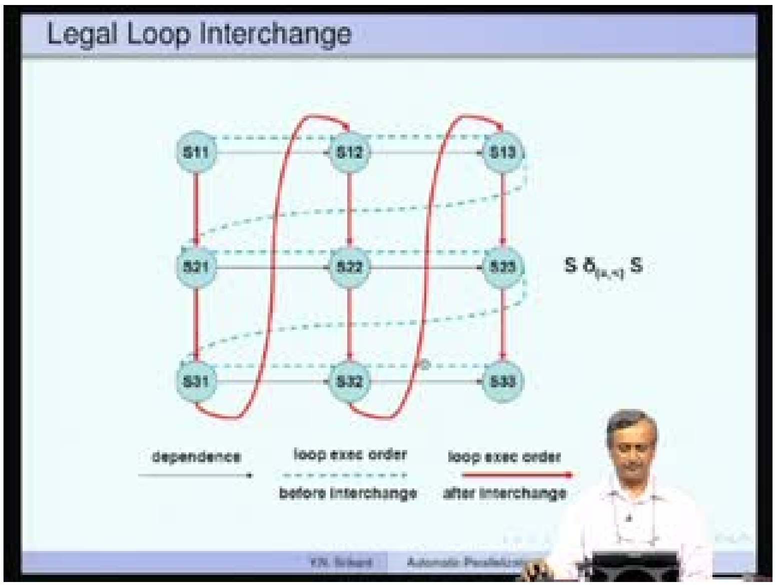
Y.N. Srikant Automatic Paralleliza

In this case, outer loop can be run in parallel, but in this case, the outer loop cannot be run in parallel. Here the dependence from S to S is less than and equal to. So, you cannot run the outer loop in parallel mode. So, let us see if you can swap. You can because, there is nothing like less than greater than. So, once we do that it becomes equal to and less than. You can actually paralyze the outer loop.

So, in vectorization you really have a only 1 statement inside the you cannot have vectorize the outer loop because, you cannot run a complete loop in vector as a vector instruction. A single in a vector instruction cannot be a complete loop. In other words. And in the case of concurrentization, if you actually have very less work for the outer loop suppose, you vectorize the inner loop.

Now, there is very less work for the inner loop just this statement. So, there may be too much overhead in executing such iterations in parallel. But once you swap and outer loop becomes parallelizable for each one of the iterations, parallel iteration, threads of the outer loop, we have a complete loop. The I complete I loop runs in sequential mode. So, there is more work per thread and therefore, this is desirable for parallelization.

(Refer Slide Time: 49:59)

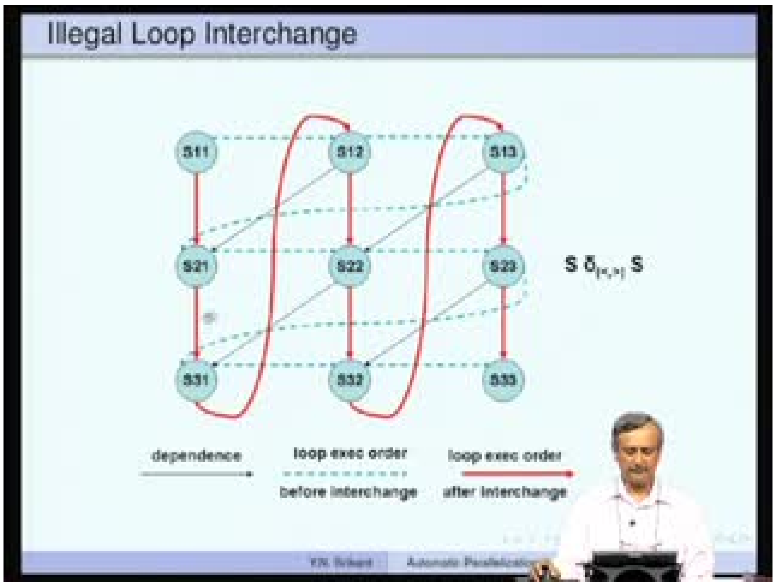

Legal Loop Interchange
S11
S12
S13
S21
S22
S23
S31
S32
S33
S δ(=,<) S
dependence
loop exec order before interchange
loop exec order after interchange
Y.N. Srikant
Automatic Parallelization

What is the intuition behind this loop interchange? So, if we have this is iteration space S 1 1, S 1 2, S 1 3, S 2 1, S 2 2, S 2 3 etcetera to begin with these are the dependences. So, 1, 2, S 3 1, S 3 2 etcetera and when we are running sequentially, these dotted lines indicate how we go.

So, we go from S 1 1 to S 1 2 to S 1 3 then, to S 2 1 to S 2 2 to S 2 3 etcetera. If you actually, interchange the loops we are going to run in this direction. So, the other index runs more frequently. So, this is a legal loop interchange. Why? In the original case, we had to compute S 1 1 first; and then S 1 2 and then S 1 3; then S 2 1, S 2 2, S 2 3 etcetera.

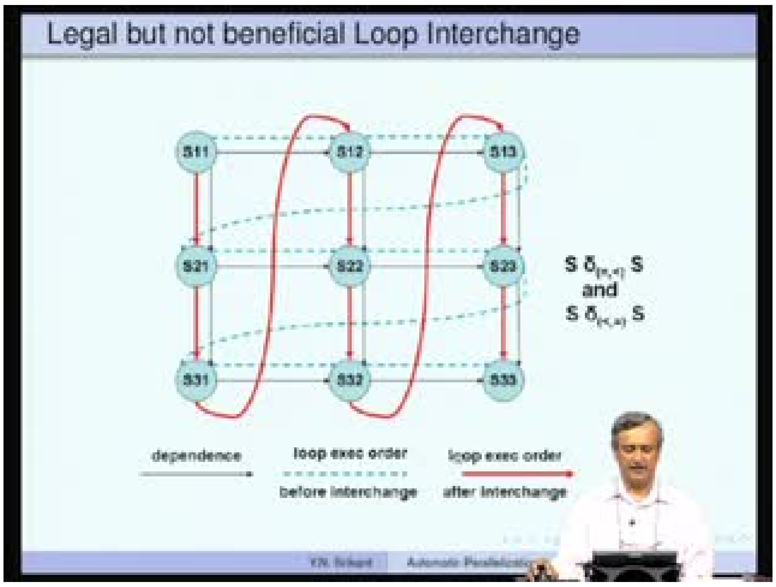
If you run in this order, there is no dependence between S 1 2 and S 2 1. So, no dependence is the dependences are all in this direction only. So, if we execute all these in and then come to these and so on. There is no violation of dependence so, loop interchange is legal.

(Refer Slide Time: 51:14)


Illegal Loop Interchange
S11
S12
S13
S21
S22
S23
S31
S32
S33
S δ(<,>) S
dependence
loop exec order before interchange
loop exec order after interchange
Y.N. Srikant
Automatic Parallelization

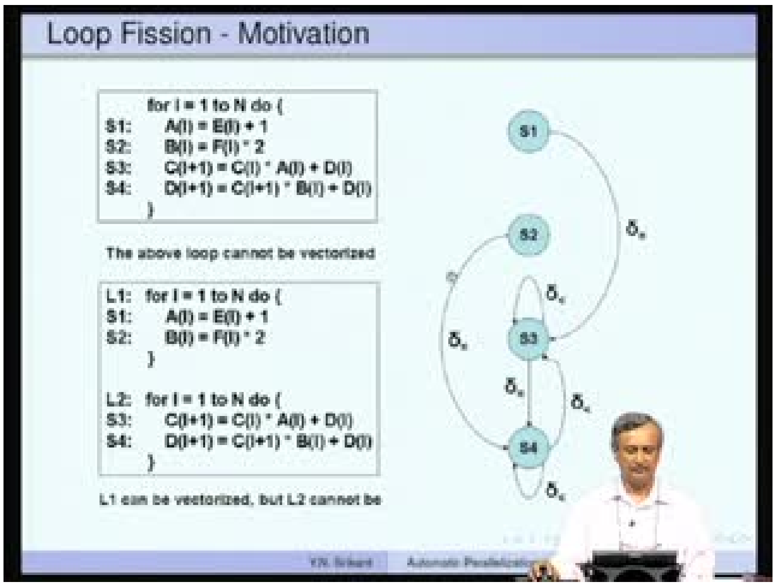
If you have a dependence from this to this then, instead of running in this direction, if you try running in this direction you will not be executing this before this so, there is a violation of dependence. So, this is an example of loop illegal loop interchange that is, you have less than and greater than. This is the kind of dependence you would have. So, you will have a dependence in this direction, which cannot permit loop interchange.

(Refer Slide Time: 51:41)


Legal but not beneficial Loop Interchange
S11
S12
S13
S21
S22
S23
S31
S32
S33
S δ(=,<) S
and
S δ(<,=) S
dependence
loop exec order before interchange
loop exec order after interchange
Y.N. Srikant
Automatic Parallelization

Loop interchange is not always beneficial. So, if we have loop dependences in both directions instead of just 1 then, loop interchange will still not help you. You cannot parallelize the loop whereas, in this case when we said loop interchange is permitted you could have run this entire thing the outer loop S delta equal to comma less than so, this is a way it was. So, we were running in sequential mode so, we could run this entire thing in parallel; this entire thing in parallel and this entire thing in parallel; the dependences would have been satisfied. So, that is what this really was.

(Refer Slide Time: 52:26)



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)
      }
```

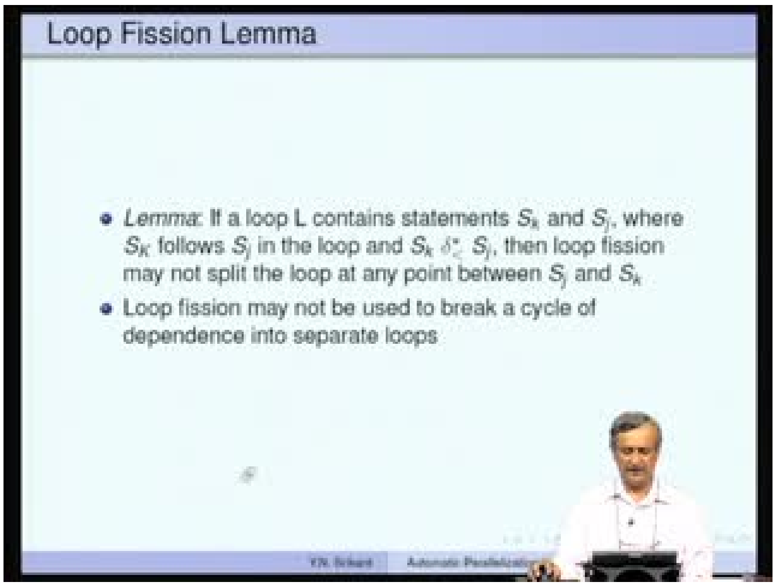
L1 can be vectorized, but L2 cannot be

S1
S2
S3
S4
δ_c
δ_c
δ_c
δ_c
δ_c
δ_c

Y.N. Srikant | Automatic Parallelization

So, loop fission is dividing a given loop into 2 parts. So, you have a single loop here. This loop cannot be vectorized as it is. Suppose, you divide it into 2 parts so, that no violation have dependences occur then, this loop can be vectorized, but this loop cannot be vectorized. But at least partially you are able to vectorize 1 part of the loop if not both loops.

(Refer Slide Time: 52:56)

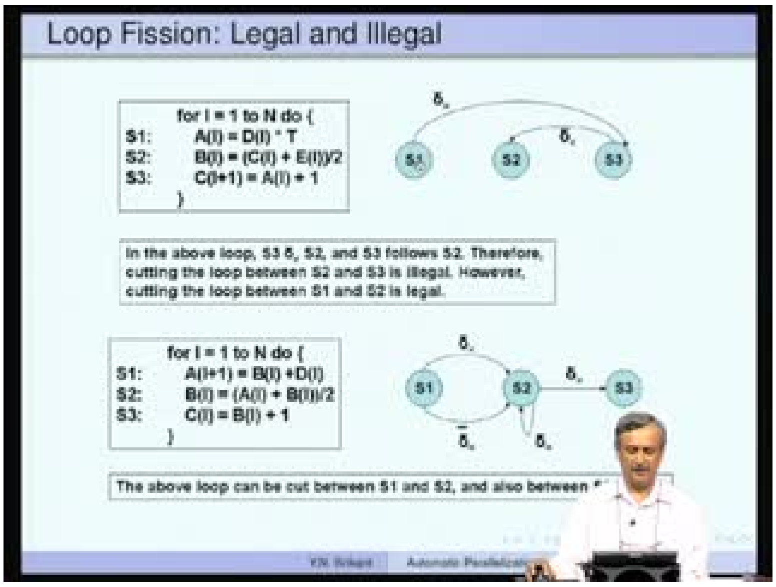


Loop Fission Lemma

- *Lemma*: If a loop L contains statements S_k and S_j, where S_K follows S_j in the loop and $S_k\ \delta^*_<\ S_j$, then loop fission may not split the loop at any point between S_j and S_k
- Loop fission may not be used to break a cycle of dependence into separate loops

So, what exactly is the condition for loop fission? If a loop contains statement S k and S, where S k follows S j in the loop so, in other words, S j comes first and then S k. There is a dependence S k to S j with delta less than. So, that means, there is a backward dependence. So, S j comes first and then S k, but the dependence is from S k to S j in the backward direction.

(Refer Slide Time: 53:29)



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
        }
```

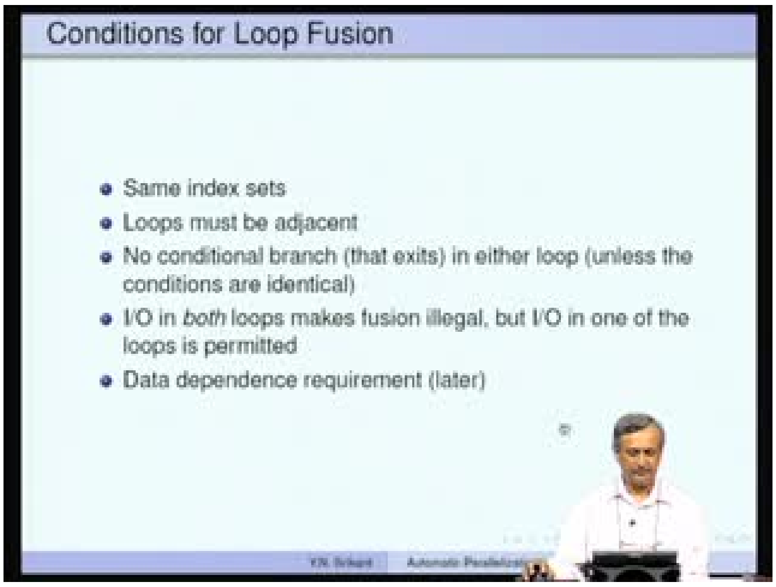
The above loop can be cut between S1 and S2, and also between

Then loop fission may not split the loop at any point between S j and S k. So, let us see what this really is? So, you have a loop like this we have the dependence from S 1 to S 3 and S 2 to S 3. Now, S 2 and S 3 are here, the dependence is from S 3 to S 2 with the delta less than. That is what this said. They said this is the way it was.

So, we have a dependence from here to here, but S 2 and S 3 are in sequence; S 3 comes after S 2. So, if you try to break the loop between these 2 points; there is a violation of the lemma and a loop fission is set to be illegal. How does that show here? Suppose, we divide the loop here at this point. Now, this A I and this A I are, but whatever is computed in S 3 as A I plus 1 will never be fed to S 2 as A I.

So, this entire thing will compute using the old values of C I; then the new values of C I would be computed, if you divide the loop between S 2 and S 3 which is illegal. Whereas, in this case, there are no such backward dependences between S 3 and S 2 there is no backward dependence; between S 1 and S 2 there is no backward dependence; so, the loop can be cut either here or here or in both places and it will still be a valid loop fission.

(Refer Slide Time: 54:49)

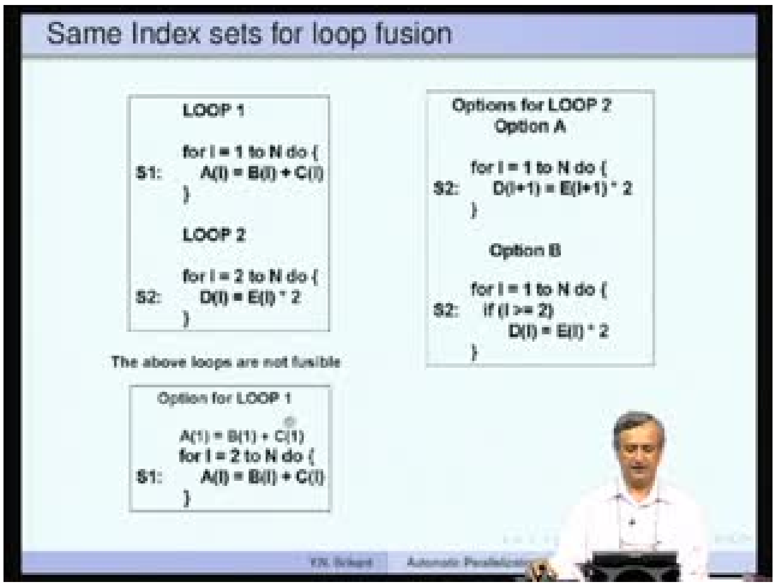


Conditions for Loop Fusion

- Same index sets
- Loops must be adjacent
- No conditional branch (that exits) in either loop (unless the conditions are identical)
- I/O in *both* loops makes fusion illegal, but I/O in one of the loops is permitted
- Data dependence requirement (later)

Y.N. Srikant
Automatic Paralleliza

(Refer Slide Time: 55:05)



Same Index sets for loop fusion

```
        LOOP 1

        for I = 1 to N do {
S1:         A(I) = B(I) + C(I)
        }

        LOOP 2

        for I = 2 to N do {
S2:         D(I) = E(I) * 2
        }
```

The above loops are not fusible

```
        Option for LOOP 1

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

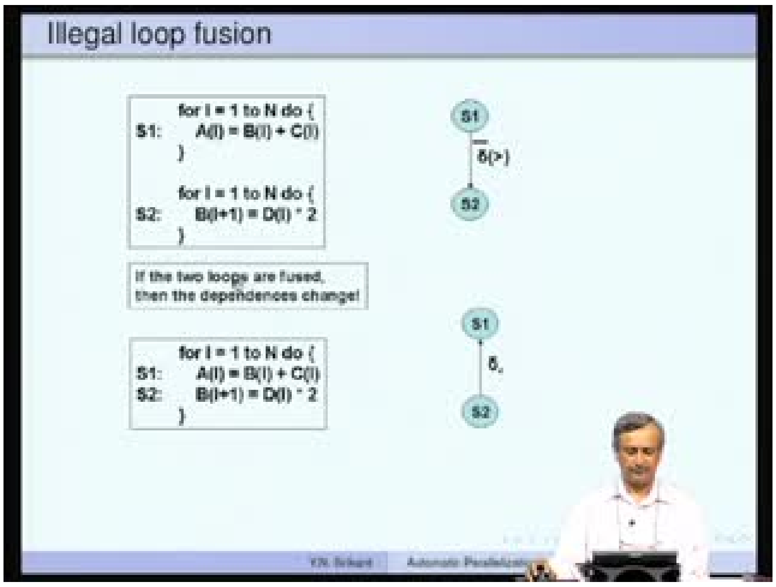
```
        Options for LOOP 2
        Option A

        for I = 1 to N do {
S2:         D(I+1) = E(I+1) * 2
        }

        Option B

        for I = 1 to N do {
S2:       if (I >= 2)
              D(I) = E(I) * 2
        }
```

Y.N. Srikant   Automatic Parallelization

The last one is the loop fusion. So, they must have the same index sets. Let loops must be adjacent. No conditional branches in either loop. I O only in one of the loops is permitted not in both and data dependence requirement must be taken care of. So, if you have loops like this and if the loops do not run with the same index set; this is runs from 1 to N; this runs from 2 to N; obviously, we cannot fuse these 2. But you can make sure that the loop index sets are the same. You can say for example, this can become if I greater than equal to then, D I equal to I star 2 and run it from 1 to N.

So, otherwise, you could change the loop 2 as I equal to 1 to N; D I plus 1 equal to A I plus 1 star 2; so, you change this loop expression here subscript expression. So, if we do that then, we can fuse the loop. For example, in this case, we simply peeled the loop A 1 equal to B 1 plus C 1 and ran the loop from I equal to 2 to N; so, this loop this loop.

(Refer Slide Time: 56:05)



Illegal loop fusion

```
     for I = 1 to N do {
S1:     A(I) = B(I) + C(I)
     }

     for I = 1 to N do {
S2:     B(I+1) = D(I) * 2
     }
```

If the two loops are fused, then the dependences change!

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

S1
$\bar{\delta}(>)$
S2
S1
$\delta_{<}$
S2

Now, this loop also runs from a 2 to N. So, it can be merged with this particular loop. So, illegal loop fusion is like this. Suppose, you have 2 loops S 1 and S 2 so, even if you merge these 2 loops, what really has happened is the dependence has changed. So, when we had the loop separate we computed A I here equal to B I plus C I and B I plus 1 equal to D I star 2 was computed here. So, this was actually an anti-dependence all the b's were used then the b's were computed. But once you fuse the loops the value computed here fed to S 1 so, the dependence has really changed.

So, because of this it is not always possible to really fuse the loops. You really have to look at the dependence from 1 loop to another. For example, there is a dependence from S 2, S 1 to S 2 it is an anti-dependence; this is B 1, B 2, B 3 etcetera and this is B 2, B 3, B 4 etcetera and because S 1 is completed first and then S 2 it becomes an anti-dependence whereas, when you fuse the loops it really becomes a flow dependence from S 2 to S 1. So, because of such reasons loop fusion is not always legal. So, this is just to give you a sampler of various loop transformations that occur in practice.

This is the end of the parallelization part of our course. So, in the next lecture we will begin with instruction scheduling etcetera. Thank you.