

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

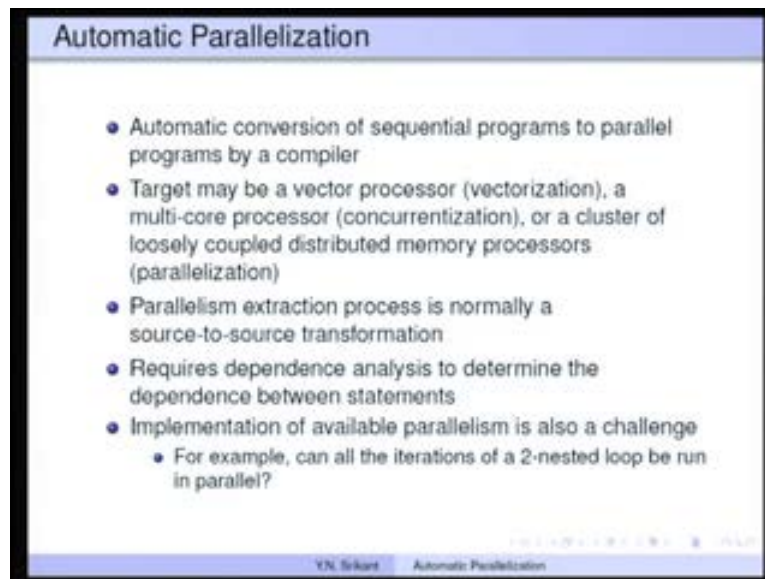
Module No. # 14

Lecture No. # 35

Automatic Parallelization-Part 2

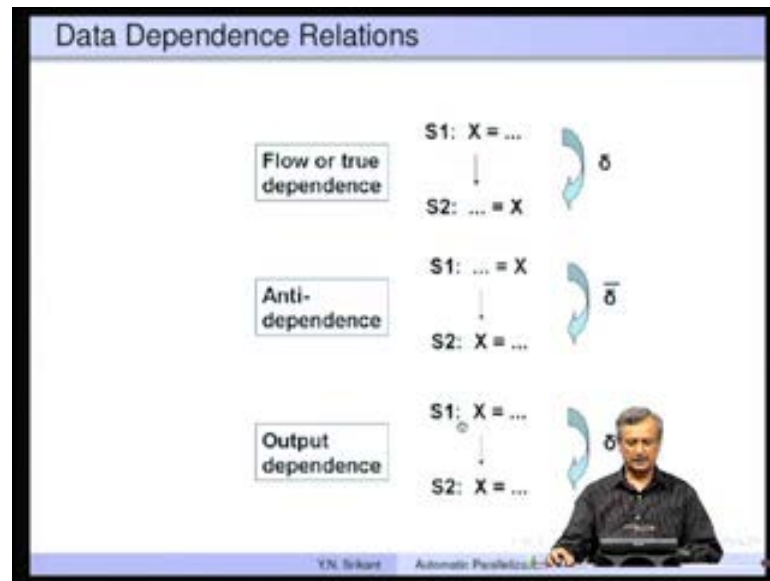
Welcome to part 2 of the lecture on automatic parallelization. Last time, we looked at an introduction to automatic parallelization. For example, we understood what exactly is automatic parallelization. It is the conversion of sequential programs to parallel programs by a compiler without any manual intervention. So the target could be any one of the parallel processors such as vector processor, multi core processor or a multiprocessor on a network itself.

(Refer Slide Time: 00:23)



The parallelism extraction is normally a source-to-source translation. The transformations are always made on the source simply because an important phase called dependence analysis really requires the expressions in the array subscripts.

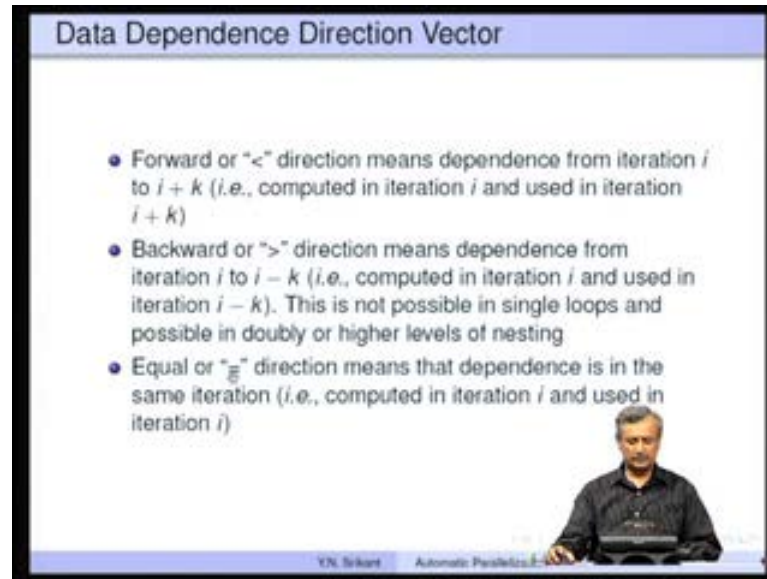
(Refer Slide Time: 01:32)



Implementation of available parallelism is also a challenge. For example, can we implement two dimensional parallelism; that is, if both the loops of a nested double loop can be run in parallel, can be really exploit so much parallelism in practice. We also got introduced to data dependence relation such as flow dependence, anti-dependence and output dependence. Flow dependence is a definition in S1 is used by S2 without any changes to x.

Anti-dependence says, there is a use of x in the statement S1 followed by a definition of x again without any changes to x in between. Output dependence is there is a definition of x in S1 followed by another definition of x in S2 without any other changes to x in between.

(Refer Slide Time: 02:14)



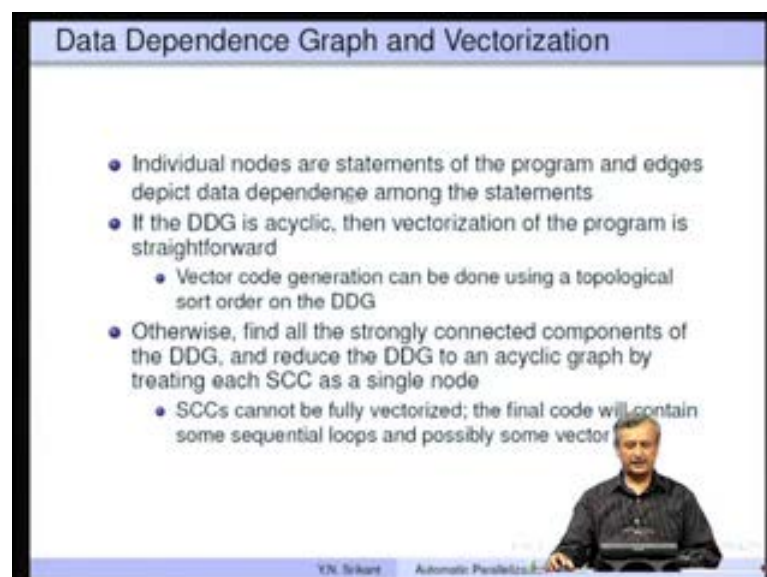
Data Dependence Direction Vector

- Forward or "<" direction means dependence from iteration i to $i + k$ (i.e., computed in iteration i and used in iteration $i + k$)
- Backward or ">" direction means dependence from iteration i to $i - k$ (i.e., computed in iteration i and used in iteration $i - k$). This is not possible in single loops and possible in doubly or higher levels of nesting
- Equal or "≡" direction means that dependence is in the same iteration (i.e., computed in iteration i and used in iteration i)

Y.N. Srikant Automatic Parallelization

So, the concept of data dependence direction vector was also introduced. There are 3 types of vectors components: one is the forward or less than direction which means, that we compute in iteration i and use the value in iteration i plus k . The backward or greater than direction vector means, we compute in iteration i and used it in iteration i minus k . This is not possible in single loops but possible only in double or higher levels of nesting. We will see examples of this as we go on today. There is equal to direction vector which means that the dependence is in the same iteration computed in iteration i and used in iteration i .

(Refer Slide Time: 02:59)



Data Dependence Graph and Vectorization

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

Y.N. Srikant Automatic Parallelization

So, the concept of data dependence graph is very important for parallelization in general. The nodes are the statements of the program and edges are the data dependences between the statements. So, if the data dependence graph is acyclic then vectorization is straight forward, you simply go through a topological sort order on the DDG and a met vector code. Otherwise, we need to find strongly conflict components, reduce the DDG to an acyclic graph by treating each of the strongly connected components as single nodes and then SCCs cannot be fully vectorized, so we need to run parts of that code as sequential code and other parts as vector code.

(Refer Slide Time: 03:45)

Vectorization Example 3.1

```

for i = 1 to 100 do {
S1: X(i) = Y(i) + 10
  for j = 1 to 100 do {
S2: B(j) = A(j,N)
    for k = 1 to 100 do {
S3: A(j+1, k) = B(j) + C(j, k)
    }
S4: Y(i+j) = A(j+1, N)
  }
}

```

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

Here is an example to say, what exactly we mean by SCCs and so on. Here is a program; here is its data dependence graph. So it has many strongly connected components, so this is the biggest strongly connected component. So, we put all that together and say this is a SCC and then this is an acyclic graph now. So, we can vectorize S1 straight away and S2, S3, S4 needs to have some sequential code in it.

(Refer Slide Time: 04:13)

Vectorization Example 3.2

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

S4: $Y(i+1:i+100) = A(2:101, N)$
S1: $X(1:100) = Y(1:100) + N$

Level 2 DDG for the composite node S2S3S4

YN Srikant Automatic Parallelization

(Refer Slide Time: 04:45)

Vectorization Example 3.3

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

S4: $Y(i+1:i+100) = A(2:101, N)$
S1: $X(1:100) = Y(1:100) + N$

Level 3 DDG for the composite node S2S3

YN Srikant Automatic Parallelization

So now for example, we run I in sequential mode and then if you look at the level 2 DDG for S2, S3, S4 this again has a strongly connected component, whereas S4 and S2 S3 forms the DDG at the next level. So with in this, we generate code for S4 in a tree mode and S2, S3 will be generated at the next level. So, this is the final DDG at the S2 S3 level. So, we see that we really cannot do too much, there is a dependence between S2 and S3; so we generate code for S2 in sequential mode whereas, S3 can be vectorized to some extent. So, this is a summary of what we did in the last lecture.

(Refer Slide Time: 05:08)

Data Dependence Direction Vector

- Data dependence relations are augmented with a direction of data dependence which is expressed as a direction vector
- There is one direction vector component for each loop in a nest of loops
- The *data dependence direction vector* (or *direction vector*) is $\Psi = (\Psi_1, \Psi_2, \dots, \Psi_d)$, where $\Psi_k \in \{<, =, >, \leq, \geq, \neq, *\}$
- We say $S_v \delta_{\Psi_1, \dots, \Psi_d} S_w$ (or $S_v \delta_{\Psi} S_w$), when
 - there exist particular instances of S_v and S_w , say, $S_v[i_1, \dots, i_d]$ and $S_w[j_1, \dots, j_d]$, such that $S_v[i_1, \dots, i_d] \delta S_w[j_1, \dots, j_d]$, and
 - $\theta(i_k) \Psi_k \theta(j_k)$, for $1 \leq k \leq d$
- $\theta(i_k) < \theta(j_k)$ only when iteration i_k is executed before iteration j_k
- $\theta(i_k) = \theta(j_k)$ only when $i_k = j_k$
- $\theta(i_k) > \theta(j_k)$ only when iteration i_k is executed after iteration j_k

YN Triker Automate Personalization

So today, we look at formal definition of data dependence direction vector and understand its implications. Data dependence relations are augmented with a direction of data dependence which is expressed as a direction vector, informally I have directly shown you some examples. The important thing to notice is there is one direction vector component for each loop in a nest of loops. So, this direction vector component has nothing to do with the number of subscripts in an array or something like that; again one direction vector component for each loop. So, if it is a loop nest of depth 3, so with i followed by another loop inside it with j as the index and followed by another loop inside the j loop called the k loop. Then, we would have three direction vector components in any direction vector associated with these array subscripts.

So, what exactly is the data dependence direction vector are simply called as direction vector. So, this is a vector consisting of d components, so d is supposed to be the depth of nesting of the loop and each of this $\psi_1 \psi_2$ etcetera that is ψ_k is one of these seven members less than, equal to and greater than. These are the three members which we already know (Refer Slide Time: 06:48). Less than equal to is a combination of less and equal. Greater than equal to is combination of greater than and equal. Not equal to implies everything but equal and star implies any one of the direction vector components but, we do not know which one. So, we say that statement S_v is $\delta_{\psi_1 \dots \psi_d} S_w$. That is, $S_v \delta S_w$ with the direction vector components ψ_1 to ψ_d .

So, we write this as $S_v \Delta \psi S_w$, when there must be particular instances of S_v and S_w . So, we see S_v and S_w are inside the loops and loop indices keep on changing a $S_1, 2, 3, 4$ etcetera. So at any point in time, if we look at the iteration in these values, we would have i_1 to i_d ready, so i_1 for at loop 1 i_2 for loop 2 etcetera.

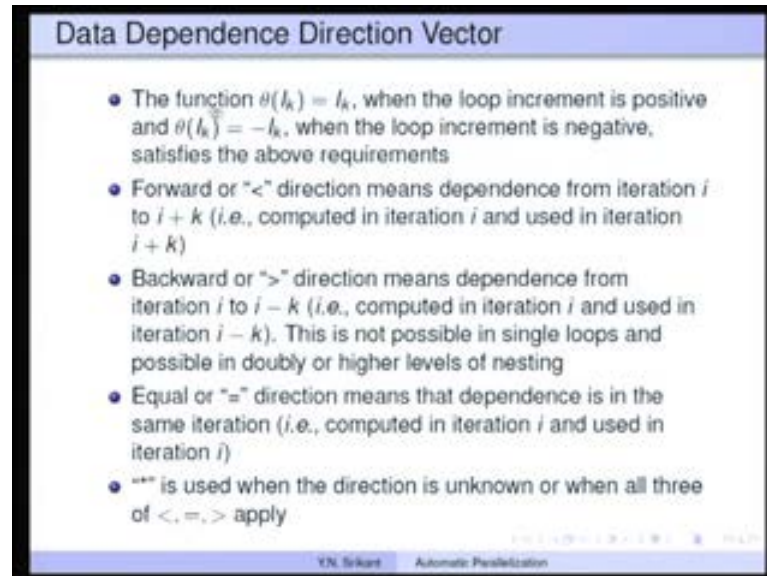
So you freeze the loops in time, you would have i_1 to i_d . Similarly, you may have another set of values at some other point in time j_1 to j_d . These two sets of values should be such that there is a dependence between S_v i_1 to i_d and S_w j_1 to j_d . So, S_v of i_1 to i_d is the instance of S_v which is running when the counter values are i_1 to i_d ; S_w of j_1 to j_d is the statement S_w running when the counter values are j_1 to j_d (Refer Slide Time: 08:36).

So, this says there must be a dependence between these two. So that is, whatever is computed in S_v of i_1 to i_d must be used in S_w of j_1 to j_d . Then, it also has a relationship between the iteration values θ of i_k ψ_k θ of j_k . In other words, the k th component of the direction vector size that is, i_k is related to θ of i_k and θ of j_k . So for example, if ψ_k is equal to 1 then θ of i_k equal to θ of j_k and if ψ_k is less than 1 then it would be θ of i_k less than θ of j_k and so on and so forth for all values between 1 and d , the k values between 1 and d (Refer Slide Time: 09:29).

So why this θ , if we actually use only the positive increments. So then, we can simply say $i_k \psi_k j_k$ and the θ part is not needed at all. So forward running the loop, so θ of i_k less than θ of j_k only when iteration i_k is executed before iteration j_k . So this before, there could be interpreted in two ways for example, when the loop is running with increment plus 1 or positive increment then, i_k is really less than j_k but, if the loop is running with a negative increment that is for i equal to 100 down to 1 (Refer Slide Time: 10:25).

So, then the increment would be minus 1 or minus 2 etcetera. So in such a case i_k less than j_k in absolute value terms will not hold. So, we will have to really look at minus i_k and minus j_k ; minus i_k less than minus j_k will definitely hold. So, θ of i_k equal to θ of j_k only when i_k equal to j_k and θ of i_k is greater than θ of j_k only when iteration i_k is executed after iteration j_k .

(Refer Slide Time: 10:54)

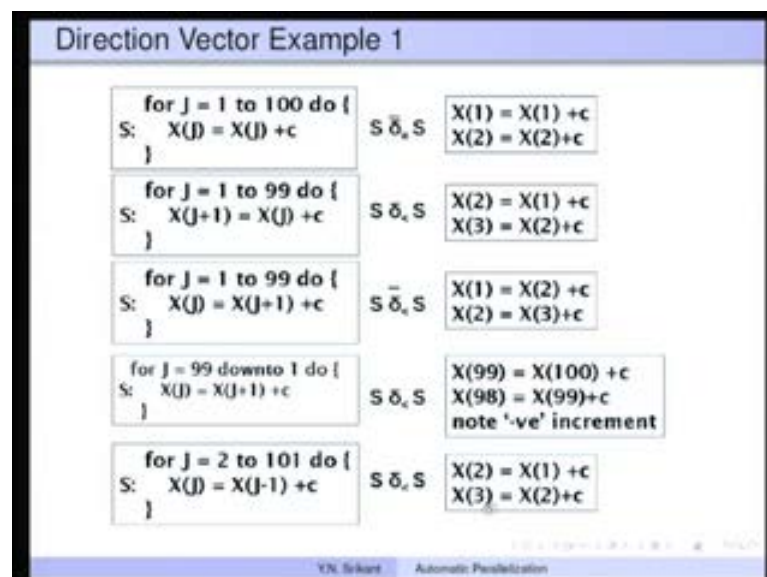


Data Dependence Direction Vector

- The function $\theta(I_k) = I_k$, when the loop increment is positive and $\theta(I_k) = -I_k$, when the loop increment is negative, satisfies the above requirements
- Forward or "<" direction means dependence from iteration i to $i + k$ (i.e., computed in iteration i and used in iteration $i + k$)
- Backward or ">" direction means dependence from iteration i to $i - k$ (i.e., computed in iteration i and used in iteration $i - k$). This is not possible in single loops and possible in doubly or higher levels of nesting
- Equal or "=" direction means that dependence is in the same iteration (i.e., computed in iteration i and used in iteration i)
- "*" is used when the direction is unknown or when all three of <, =, > apply

So, the function theta I_k is really just I_k when the loop increment is positive, this is what I was just mentioning. Theta I_k equal to minus I_k , when the loop increment is negative and this definition of theta I_k satisfies our requirements. When we are running with a positive increment, we simply use I_k less than j_k or I_k equal to j_k etcetera. When we are running with a negative increment, we use minus I_k less than minus j_k etcetera.

(Refer Slide Time: 11:43)



Direction Vector Example 1

<pre>for J = 1 to 100 do { S: X(J) = X(J) + c }</pre>	S δ _c S	<pre>X(1) = X(1) + c X(2) = X(2) + c</pre>
<pre>for J = 1 to 99 do { S: X(J+1) = X(J) + c }</pre>	S δ _c S	<pre>X(2) = X(1) + c X(3) = X(2) + c</pre>
<pre>for J = 1 to 99 do { S: X(J) = X(J+1) + c }</pre>	S δ _c S	<pre>X(1) = X(2) + c X(2) = X(3) + c</pre>
<pre>for J = 99 downto 1 do { S: X(J) = X(J+1) + c }</pre>	S δ _c S	<pre>X(99) = X(100) + c X(98) = X(99) + c note '-ve' increment</pre>
<pre>for J = 2 to 101 do { S: X(J) = X(J-1) + c }</pre>	S δ _c S	<pre>X(2) = X(1) + c X(3) = X(2) + c</pre>

So this I already mentioned, forward or less than direction means, computed in iteration i and used in iteration i plus k and so on and so forth. So let us go on to the next slide.

Here are now lots of examples to understand what we mean by these iteration - direction vector components. Let us take a simple loop, J equal to 1 to 100 do S is X J equal to X J plus c . So on this side, we have really unrolled the loop giving values for J , so X 1 equal to X 1 plus c , X 2 equal to X 2 plus c , X 3 equal to X 3 plus c etcetera. Now, if you look at this, the X 1 is read and then X 1 is assigned but, they are all in the same iteration J equal to 1.

Similarly, X 2 is read and then assigned to X 2 adding c to it and this is done in the same iteration J equal to 2 and so on and so forth. Therefore, the relationship between this index value on the right hand side and this index value is just the equality relationship, so i_k equal to j_k so 1 equal to 1, 2 equal to 2 etcetera (Refer Slide Time: 13:05).

The type of dependence is anti-dependence because we are reading and then writing into the same location. So the dependence here is S delta equal to bar, delta bar equal to S - S delta bar equal to S . The dependence is within the same iteration.

Let us take the next loop, J equal to 1 to 99 do X of J plus 1 equal to X of J plus c , so again just enroll the loop J equal to 1 will get X 2 equal to X 1 plus c , J equal to 2 will get X 3 equal to X 2 plus c . Now, similarly X 4 equal to X 3 plus c and so on and so forth.

Now see, whatever was computed in J equal to 1 that is X of 2, is used in J equal to 2 as X of 2 again. We compute in iteration k and then use it in iteration k plus 1. This is a forward relationship, so we have I_k and j_k , I_k is less than j_k ; I_k can be treated as some j value in this case, I_k plus 1 is j_k - I_k is less than j_k (Refer Slide Time: 14:27).

Now, because of this relationship I_k less than j_k we have the dependences S delta less than S . This is a flow dependence because X 2 is computed here and X 2 is used in the next iteration j equal to 2. Let us take another example J equal to 1 to 99 again X J equal to X J plus 1 plus c ; X J plus 1 is on the left hand side here now it is on the right hand side. So we get by unrolling the loop for J equal to 1, we get X 1 equal to X 2 plus c and for J equal to 2, we get X 2 equal to X 3 plus c . Now, the dependence flow is anti-dependence X 2 is read here and then assigned. It is read in J equal to 1 and then assigned in J equal to 2.

So, again we have less than as the direction vector component I_k less than j_k and the dependence is anti. So, $S \delta \bar{S} < S$ will be the actual dependence. Here is a loop with a negative increment, for J equal to 99 down to 1 do $X J$ equal to $X J$ plus 1 plus c . So, the same loop but we are running backwards from 99 to 1. So let us unroll the loop; so this is a loop which runs from 99 to 1 with a negative increment of minus 1 $X 99$ is $X 100$ plus c , $X 98$ is $X 99$ plus c , $X 97$ would be $X 98$ plus c and so on and so forth.

Now in j equal to 1, we compute $X 99$ and then in J equal to 2, we use $X 99$. So, because of the loop running in a different direction from 99 to 1 the dependence in this example has really turned to a flow dependence - you compute $X 99$ and use $X 99$ - and the direction of dependence is still less than because, we compute in J equal to 1 and use it in J equal to 2, 1 less than 2.

So, $s \delta S < S$ will be the dependence in this case. The last case here for J equal to 2 to 101 do $X J$ equal to $X J$ minus 1 plus c , again unrolling the loops you get $X 2$ equal to $X 1$ plus c $X 3$ equal to $X 2$ plus c and $X 4$ equal to $X 3$ plus c and so on. Again it is easy to see that here is a flow dependence computing J equal to 1 and use in J equal to 2, so you have $S \delta < S$.

(Refer Slide Time: 17:17)

So, another slightly more complicated example with nested loops. You have I equal to 1 to 5 do J equal to 1 to 4 do, $S1$ is A of $I J$ equal to B of $I J$ plus C of $I J$, $S2$ is B of I comma J plus 1 equal to $A I, J$ plus $B I, J$.

The dependence graph is shown here; it shows a dependence from S2 to S1 as delta equal to comma less than. There is a dependence from S1 to S2 with delta equal to comma equal to and there is an dependence from S2 to S2 with delta of equal to comma less than. So, let us see how these arise.

Again we resort to the familiar technique of unrolling the loop, I equal to 1 and J equal to 1 to begin with. So, A_{11} is equal to B_{11} plus C_{11} , so that is S1 and S2 will be A_{12} B_{12} equal to A_{11} plus B_{11} . So in this case, please observe that A_{11} here is computed and A_{11} is used. So this is in the same iterations with values I equal to 1 and J equal to 1, so whatever is computed by S1 is used by S2.

So, the dependencies is from S1 to S2 - delta the flow dependence and the values of I and J which are in which the computation takes place and the usage takes place are the same. So, I value is the same so delta of equal to J value is also the same so another equal to.

So, in the same iteration of I and J we compute and use. So this becomes delta of equal to comma equal to. So that is the dependence from S1 to S2. Next let us increment J; J equal to 2, so A_{12} is B_{12} plus C_{12} and B_{13} is A_{12} plus B_{12} . Now B_{12} was computed in J I equal to 1 and J equal to 1 and B_{12} is used in I equal to 1 and J equal to 2. So I value remains the same but J value has increased, so 1 less than 2 and I value is the same, so S2 delta S1 computed use so it is a flow dependence but the first component is equal to that is same value of I and less than because J equal to 1 is less than J equal to 2 1 less than 2. So this is the S2 delta equal to comma less than S1, so this is the dependence that we have shown here.

So, let us look at the third dependence J equal to 3. We have A_{13} equal to B_{13} plus C_{13} and B_{14} equal to A_{13} plus B_{13} . So B_{13} was computed here in J equal to 2 and then B_{13} is used in J equal to 3. So it was computed in S2 in J equal to 2, I equal to 1 and j equal to 2 and computed in S2 again in I equal to 1 and J equal to 3. So here - this is what we are looking at - this is the computation part and this is the usage part this usage is this is this computes first and in some later iteration this uses it.

So, between these two there is a flow dependence S2 delta S2; it happens in the same iteration of I, so the first component is equal to and between J equal to 2 and J equal to 3 2 less than C, so that the second component is less than. So we have S2 delta equal to

comma less than S2. So this is the third arc that we have written here, so these are the three dependences in this example along with the direction vector components.

(Refer Slide Time: 21:34)

The slide, titled "Direction Vector Example 3", illustrates the relationship between two statements, S1 and S2, within a nested loop structure. The main code is:

```

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

```

Two specific states are shown:

- Top State:** Labeled $S1 \delta_{(i,j)} S2$. It shows the state where $i=1, J=2$ for S1 (executing $A(2,2) = \dots$) and $i=2, J=1$ for S2 (executing $\dots = A(2,2)$).
- Bottom State:** Labeled $S2 \delta_{(i,j)} S1$. It shows the state where $i=1, J=2$ for S2 (executing $A(2,2) = \dots$) and $i=2, J=1$ for S1 (executing $\dots = A(2,2)$).

The slide also features a small inset image of a man in the bottom right corner and a footer with the text "YN Sikant Automata Peralihan".

So the next example, I promised to give you an example with the greater than direction vector in the last class, so I will do it now. Look at this for I equal to 1 to N do and then the nested inside that is for J equal to 1 to N do, S1 is A of i plus 1, J equal to something and S2 is something equal to A of I, J plus 1, so let us unroll the loops.

I equal to 1 and J equal to 2, so you get A of 2 and 2, so A 2 2 equal to something. Then for considering S2 look at a different values of I and J, I equal to 2 and J equal to 1. So S2 will be again A 2 2 because this is I and J plus 1 so 2 and 2.

So, observe that there is a dependence from this instance of S1 that is A 2 2 equal to something to this instance of S2 is again equal to A 2 2 with different values of I and J. The relationship between I values is 1 less than 2, so the first component is less than. The relationship between the J values is too greater than. So the second component is greater than so S1 delta S2 with the 2 direction at the components less than or greater than.

What we need to observe here is, when the I value is 1, the J loop completes; it runs with J equal to 1, 2, 3, 4 etcetera up to N. In the next iteration of I that is I equal to 2 another set of J values the J loop completes again - now they are executed. So, in that particular instance there is a J equal to 1 in which A 2 2 is occurring, so it is perfectly ok because

the J loop for I equal to 1 gets executed before the J loop for I equal to 2. So there is no violation of any execution order here and everything is legal. Whereas with a single loop, we cannot do this. We cannot have some value computed and then you use it in a previous iteration, we cannot go back.

Here, we are really not going back, we are going forward; it is just that the relationship between the J values is different but they occur in two different instances of the I loop, I equal to 1 and I equal to 2. So everything is going forward, no invalid loop executions here.

One more example with S2 delta less than comma greater than S1. So similar loop I equal to 1 to N J equal to N S1 is equal to A of I, J plus 1 and S2 is A of I plus 1, J. So, we have just swapped the left and right hand sides of S1 and S2.

Again take A I equal to 1 J equal to 2 so you get A 2 2 equal to something that is S2 and take I equal to 2 and J equal to 1 you get S1 which is A 2 2. So again the relationship between the I value is 1 less than 2, so the first component is less than and the second component is 2 greater than 1, so it is again greater than. So again it is just the same story, I equal to 1 the entire J loop completes. So when it is J equal to 2 you compute A 2 2 with I equal to 2 another set of J values is going to execute, so in that J equal to 1 you are going to use the previous computed value of A 2 2. So again the execution is going forward and there is no violation of any aspect.

(Refer Slide Time: 25:26)

Direction Vector Example 4

```

for i = 1 to 100 do {
  for j = 1 to 100 do {
    for k = 1 to 100 do {
      S1: X(i, j+1, k) = A(i, j, k) + 10
    }
    for l = 1 to 50 do {
      S2: A(i+1, j, l) = X(i, j, l) + 5
    }
  }
}

```

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

YN, Srikant Automatic Parallelization

So this is an example which we saw before, let me quickly run through this. There are 2 dependences S1 to S2 with delta of equal to comma less than and another one from S2 to S1 with delta of less than and equal to.

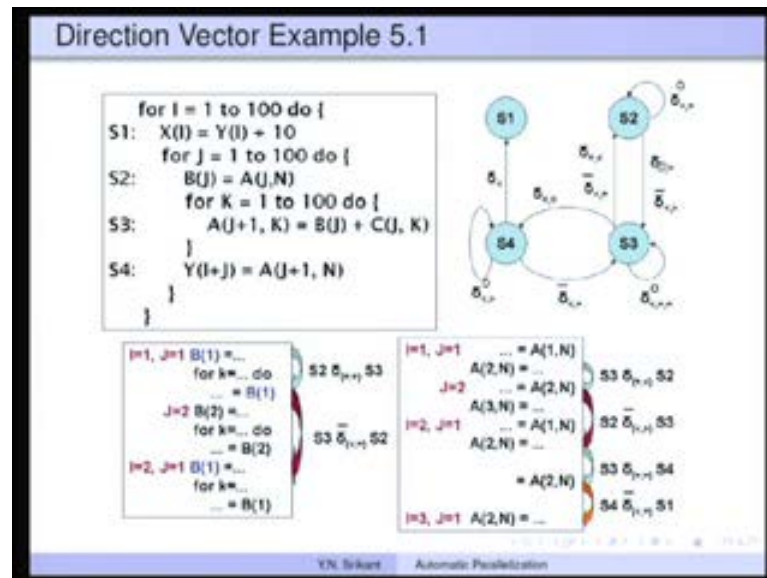
So, again when we unroll the loops with I equal to 1 J equal to 1 I equal to 2 J equal to 2 J equal to 3 etcetera, these are the dependences. So X 1 2 K, some value of K will be all right any value will do here between 1 and 50 that is all and the same value of L can be taken in this case. So X of 1 comma 2 comma K is computed in I equal to 1 and J equal to 1 and it is used in I equal to 1 and J equal to 2 X of 1, 2, L.

So, arbitrarily you can say K equal to 25 and L equal to 25 so then these two become identical 1 to 25 and 1 to 25. So, whatever is computed here is used here let us look at the relationships between I and J in this case it is the same iteration of I. So the first component would be equal to here. It is a different iteration of J with 1 less than 2 computed in 1 used in 2 so the second component would be less than. So another instance here again X of 1 3 K and X of 1 3 L.

Now for this other dependence so this was the dependence from S1 to S2. What about the dependence S2 to S1? So A of 2, 1, L is used as A of 2, 1, K again put some arbitrary values of L and K, which are within these bounds and which are equal then, you can see that is A to 1 L is computed in I equal to 1 and J equal to 1 and it is used in I equal to 2 and J equal to 1. So observe that this J and in I equal to 1 and I equal to 2 or two different instances of J. They are two different loops which are executing in with different values of I.

But, if we just look at the values of J as before they are equal. So there is dependence from here to here; it is first component I equal to 1 and I equal to 2 - 1 is less than 2 - so it is less than 1 is less than 2. The second component has the same value of A therefore the second component is equal to. So there are two other instances here so that is the dependence from S2 to S1.

(Refer Slide Time: 28:13)



Finally, this is a very complicated example which we saw before that needs justification of all the dependences. So, let us look at all the dependences here. Let us finish off the dependences between S2 and S3. So there are many dependences here; so there is a self loop on S2 and S3. Then there are two dependences S3 to S2 and another two dependences from S2 to S3.

So first of all, if we look at S2 delta equal to equal to S3 that is here S2 delta S3 with equal to equal to. So, you can get this by I equal to 1 J equal to 1, you have B of 1 equal to something there is a K loop inside here and then there is a B here which is independent of K that is again B of 1 J only. So B 1 is computed here and B 1 is used here, so it is a same value of I and J.

So since this B 1 is outside k, we do not have k component in the direction vector. Values of I and J are equal so equal to equal to S3. The next one is B 1 is read here and then B 1 is computed here in I equal to 2 and J equal to 1, so from here to here. So that gives rights to other S3 delta bar S2 - so S3 delta bar S2. The I value has changed so 1 less than 2 so less than, J value remains the same so it is equal that is here.

Then, we come to S3 equal to less than S2. So again S3 to S2, first equal to and less than this particular dependence. Now, you have I equal to 1 and J equal to 1 so this is A equal to 1 comma N so that is here equal to this particular thing and then A 2 comma N is here this is S3.

So, from this $A[2, N]$ when $J = 2$, we have $A[2, N]$; that is, S_2 this particular component (Refer Slide Time: 30:46). So, we compute here and then in the next iteration of J , we use it in this so that is, what is shown here $J = 1$. You compute $A[2, N]$ and $J = 2$ - you use $A[2, N]$. So the computation happens in S_3 , usage happens in S_2 in the next iteration of J .

So, that gives rise to $S_3 \delta S_2$. I value is the same, so first component is equal to. J value 1 less than 2, so second component is less than, so that is also taken care of. Similarly, you have A to N in $J = 2$ so that is usage here in S_2 and then you have A to N which is computed, so that is S_3 . There is also an anti-dependence A to N here and A to N here. These two instances that is, usage here and computation here and for that you have different values of I ; $I = 1$ and $I = 2$, so less than and different values of J as well; this is $J = 2$ and this is $J = 1$, so greater than (Refer Slide Time: 31:52).

So, $S_2 \delta S_3$ and this is an anti-dependence with less than and greater than is here. Then, this takes care of all the four dependences here in this case. Now, let see between S_3 and S_4 . You have A to N computed that is in S_3 for some value of J ; we have started from $I = 2$ and $J = 1$ to just reduce the amount of space that is consumed for the picture. Then you have $A[2, N]$ in S_4 . So this and this component, this particular expression.

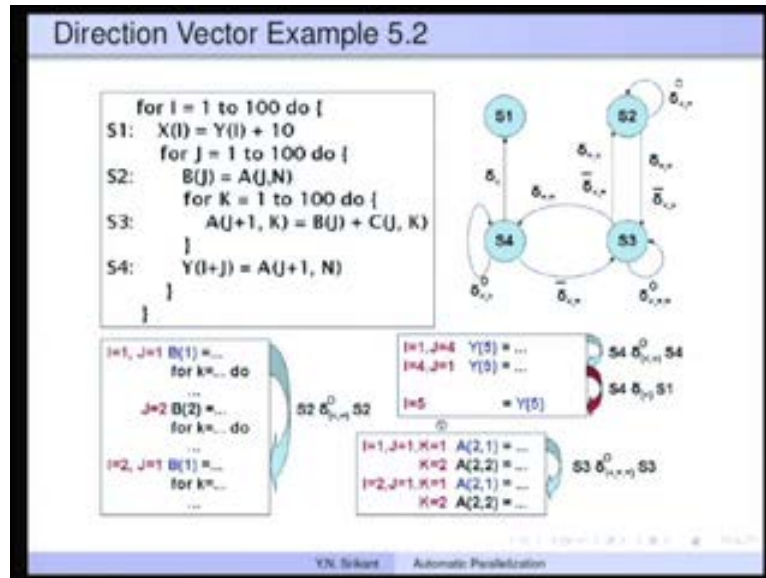
So, you compute here and you use here between these two and the value of I and J are identical. So, it is a flow dependence between this and this with equal to comma equal to so that is taken care of.

Next, between S_4 and S_1 there is another dependence with a delta bar. This is S_4 , this is S_4 and S_1 - S_4 and S_3 - it should have been S_3 not S_1 so S_4 to S_3 with a delta bar. We actually use $A[2, N]$ here with $I = 2$ and $J = 1$ that is the usage that we are talking about here. Then, we have $I = 3$ and $J = 1$ that would be A to N equal to that is S_3 again this S_3 and this is S_4 .

So, from S_4 we are actually using something and which is computed in S_3 , a little we have used something that is a read the old value and then the new value is computed in S_3 , so this would be S_3 so $S_4 \delta S_3$ with less than or equal to, so that would be S_4 to S_3 this particular component. So, here is a usage followed by a definition so that

would be an anti-dependence I value is 2 and three so this is less than J value is 1 and 1 so it is equal.

(Refer Slide Time: 34:26)



So, these are the couple of these 4 plus these 2 that we have taken care of and now there are more here. So, between B 1 here and B 1 here - this is B 1 - S2, S2 to S2 is the other dependence that we need to take care of, it is an output dependence. So I equal to 1, J equal to 1 and then I equal to 2 J equal to 1. So, we have 1 less than 2, so less than and equal J equal to 1 and J equal to 1 it is less equal to so it is an output dependence from S2 to S2 so that is the loop that we have taken care of here.

Then, we have S4 to S4 and S4 to S1; S4 to S4 this one and S4 to S1. So this is interesting, this is the one that really causes the dependences from S4 to S4. So if I equal to 1 and J equal to 4 we have Y 5 equal to and obviously this is just sum so I equal to 4 and A equal to one also we get Y 5 equal to so that is an output dependence between these two.

I value is less than, 1 less than 4, J value 4 greater than 1 so the second component is greater than. Then we have between these two S4 and S1, so S1 is here. So we are really computing Y of I plus J here and using Y I here. So with I equal to 5 this would become Y 5 and Y 5 was computed here. So between these two we again have a dependence so this is 4 and this is 5, so less than S1- S4 delta less than S1.

So, this how the dependence arises. The last part is here $S_3 \delta_0$, S_3 with less than equal to and equal to, so this loop. There are 3, so this the inside nested loop in all the three loops it has been nested. Now, you compute A to 1 equal to and use A to 1 equal to again with the value I equal to 1, J equal to 1, k equal to 1, I equal to 2, J equal to 1 and K equal to 1; that is, this is this, this particular thing (Refer Slide Time: 36:41).

So, the values are identical A to 1 and A to 1, so this is an output dependence and I value is 1 less than 2 so the first component is less than. J value and k value are equal, so other 2 components are equal to. So this is a very large example to show the various dependences in this particular program and how they arrive. So typically, the moral of the story is computation of dependences cannot be done by a machine by unrolling loops like this.

(Refer Slide Time: 37:23)

Execution Order Dependence and Direction Vector

- $S_v \ominus S_w$ if S_v can be executed before S_w (in the normal execution of the program)
- $S_v \delta_\psi S_w$ only if $S_v \ominus_\psi S_w$
- *i.e.*, \ominus may hold but δ may not hold
- Example:

$S_1: a=b+c$ $S_2: a=c+d$ $S_3: e=a+f$	$S_1 \ominus S_2$, $S_2 \ominus S_3$, and $S_1 \ominus S_3$ are all true, but $S_1 \delta S_2$ and $S_1 \delta S_3$ are false; only $S_2 \delta S_3$ is true
--	--
- Hence execution ordering is weaker
- Execution order direction vector is similar to the data dependence direction vector (similar definition)
- Not all direction vectors are possible
- We will now consider legal exec order d.v. by looking at the syntax of constructs

⏪ ⏩ ⏴ ⏵ 🔍 🔄

YN Srikant Automatic Parallelization

We need to actually have mathematical test which determine whether the dependence or not and that is precisely, what we want to study a little later. Before this, we have to look at the data dependence vector is what we saw. Let us look at what is known as execution order dependence and execution order dependence direction vector. Why are these needed? These are going to be useful later, so we will consider these as legal direction vectors and these can be computed by looking at the syntax of constructs.

So, when we compute the data dependence direction vectors this will be of great use in reducing the computation. So, the execution order dependence is denoted by the big

theta. So $S_v \theta S_w$, if S_v can be executed before S_w in the normal execution of the program a very simple definition this is just execution order. So, it is clear if $S_v \delta S_w$ only if $S_v \theta S_w$ in other words, theta may hold but delta may not hold, so let us see how?

Here is S_1 , $a = b + c$; then S_2 , $a = c + d$, assigning another value of a . S_3 equal to $a + f$, so S_1 occurs before S_2 and S_3 , so $S_1 \theta S_2$ and $S_1 \theta S_3$ both hold. Similarly, $S_2 \theta S_3$ also holds, because the order of execution is first S_1 then S_2 and then finally S_3 .

So, all these three are true $S_1 \theta S_2$, $S_2 \theta S_3$, $S_1 \theta S_3$ but, then S_2 redefines a value of a ; a was assigned a value $b + c$ here but, again it redefines the value as $c + d$. So the value of a which is used in S_3 is from S_2 and not from S_1 there is no dependence between S_1 and S_3 as far as the delta is concerned, data does not flow from S_1 to S_3 only it flows from S_2 and S_3 . So, $S_2 \delta S_3$ is true but, we do not have $S_1 \delta S_3$; we do not have $S_1 \delta S_2$ none of these are true but, $S_1 \delta S_2$ is true is definitely true but, $S_1 \delta S_3$ is not true, because nothing flows from S_1 to S_3 .

So, hence execution ordering is weaker than delta data flow, the data dependence. The reason is many more dependences are possible under theta but few are dependences are possible under delta. Execution order direction vector is very similar to the data dependence direction vector. We already defined the data dependence direction vector; we want to compute similar vector for this and not all direction vectors are legal in the case of execution order dependence and the legal ones are possibly we can guess them, compute them by using the syntax of the constructs let us look at some examples.

(Refer Slide Time: 40:45)

The slide is titled "Single Loop Legal Direction Vectors - 1". It contains two bullet points and two code snippets. The first bullet point states that $S1 \theta_{(\leq)} S2$, $S2 \theta_{(<)} S1$, $S1 \theta_{(<)} S1$, and $S2 \theta_{(<)} S2$ are all possible. The second bullet point notes that $S2 \theta_{(=)} S1$ is not possible because S2 comes after S1 in lexical ordering. The first code snippet shows a loop: `for i = L to U do { S1: ... S2: ... }`. The second code snippet shows the unrolled execution order: `i = 1 S1 S2 i = 2 S1 S2`. A small video inset in the bottom right shows a man speaking.

Why are we looking at such examples? The point is legal direction vector as for as the execution order can be constructed using the syntax of the program. Here we are looking at the most important constructs of any program and by looking at similar constructs we can say, the compiler can construct these direction vectors and dependences.

So here is a single loop, I equal to L to U lower bound and upper bound of loop, positive increment S1 S2, it could even be negative increment that really does not matter as far as execution order goes. So let us look at I equal to 1, so S1 and S2 we have unrolled the loop, I equal to 2 again we have unrolled the loop S1 and S2. So any possible execution order among this gives rise to S1 S2 executes first in I equal to 1 and then followed by S1 and S2 in I equal to 2.

So, theta is possible between S1 and S2, this S1 and this S1, this S1 and this S2, this S2 and this S1 finally this S2 and this S2, so that is what is said here. S1 theta less than or equal to S2, so why S1 in I equal to 1 is related to S2 both in I equal to 1 so that is taken care of by the equal to component and in the I equal to 2 component 1 less than 2, so this and this are also related by this execution also related by this execution order. Similarly S2 theta less than S1, S2 is here; S1 is here so, I equal to 1 and I equal to 2, 1 less than 2 so that is theta less than S1. S1 theta less than S1 is also possible, so I equal to 1, I equal to 2 this is the S1 we are considering S1 and S1, so this is true.

Similarly, $S2 \theta_{<} S1$ is also possible. This and this, $I = 1$ $I = 2$ we are comparing these two. But note that $S2 \theta_{=} S1$ is not possible $S2 \theta_{=} S1$ because we cannot reverse the execution order of these two. We can execute $S2$ first and then this $S1$ but that will be in $I = 2$ not in the same $I = 1$.

(Refer Slide Time: 43:25)

Single Loop Legal Direction Vectors - 2

- $S1 \theta_{(=)} S2$ and $S2 \theta_{(=)} S1$ cannot happen
- $S1 \theta_{(<)} S2$, $S2 \theta_{(<)} S1$, $S1 \theta_{(<)} S1$, and $S2 \theta_{(<)} S2$ are all possible

```

for i = L to U do {
    if (...) then
S1: ...
    else
S2: ...
    endif
}

```

```

i = 1
S1
i = 2
S2
i = 3
S2
i = 4
S1

```

S1 and S2 may be in any order, but both S1 and S2 cannot occur together in any iteration

So, these are the only legal direction vectors possible with a single loop. What about a loop with an if then condition, so in this case the same loop either $S1$ or $S2$ will be executed but not both in the same iteration. In different iterations they can be in any order but in the same iteration either $S1$ or $S2$ executes. So $I = 1$, $S1$ let us say executes $I = 2$, perhaps $S2$ executes, $I = 3$ again possibly $S2$ $I = 4$ possibly $S1$.

So again, we look at the ordering here and then guess the theta. So this $S1$ is here, $\theta_{=} S2$ $S1 \theta_{=} S2$ is not possible, because $S1$ and $S2$ cannot execute in the same iteration at all. $S2 \theta_{=} S1$ is also not possible, because of the same reason. $S1 \theta_{<} S2$ is possible, $S1$ is here $S2$ is here so $I = 1$ $I = 2$ $1 < 2$. $S2 \theta_{<} S1$ is possible, so $S2$ is here $S1$ is here so $2 < 4$, $2 < 4$.

$S1 \theta_{<} S1$ is possible $S1$ here and $S1$ here so $1 < 4$. $S2 \theta_{<} S2$ is also possible $S2$ here $S2$ here $2 < 3$. So these are all possible legal direction vector as far as this single loop is concerned.

(Refer Slide Time: 44:53)

Multi-Loop Legal Direction Vectors - 1

Loop 1

- $S1 \theta_{(=, \leq)} S2$, $S2 \theta_{(=, <)} S1$, $S1 \theta_{(<, =)} S2$, $S2 \theta_{(<, =)} S1$, $S1 \theta_{(<, >)} S1$, and $S2 \theta_{(<, >)} S2$ are all possible
- $S2 \theta_{(=, >)} S1$ and $S1 \theta_{(=, >)} S2$ are not possible

```

Loop 1
for I = LI to UI do {
  for J = LJ to UJ do {
    S1: ...
    S2: ...
  }
}
  
```

```

I = 1
  J = 1  S1
        S2
  J = 2  S1
        S2
I = 2
  J = 1  S1
        S2
  J = 2  S1
        S2
  
```

YN Srikant Automate Parallelism

What about multiple loops. So here is I going from L I to U I, J going from L J to U J. So we have again S1 and S2. Let us unroll the loops for J equal to 1 S1 S2, J equal to 2 again S1 S2. Similarly, I equal to 2, J equal to 1 and J equal to 2 so any possibility theta between these is valid. So, what is not possible let us look at that it is easier. S2 theta equal to equal to S1. So, S2 is here in the same iteration of I and J we cannot reverse S2 and S1. Therefore S2 theta S1 with equal to equal to direction vector is not possible. Similarly, S1 theta equal to greater than S2 is not possible; this greater than implies S1 theta greater than theta S2 is not possible with equal to because I cannot execute S1 in later iteration than S2 but with the same value of I that is not possible; whereas the others are all possible, let us take 1 or 2 of them - S1 theta less than equal to S2. So S1 theta equal to equal to S2 is possible, so S1 is here same value of S1 S2, I equal to 1 J equal to 1. So here is S1 and here is S2, so that is possible no problem at all. If we want to relate it to another value of S2 with J value greater that is here, so this S1 and this S2 I equal to 1 J equal to 1 J equal to 2. So J 1 less than 2 whereas I value is the same so this S1 this S2 are related by the less than relationship here.

So, S2 theta equal to comma less than S1. So, S2 is here and the same value as I, so S1 is here with the different value of J, so 1 less than 2 you rise to this and so on and so forth. So these are all the possible direction vectors as far as a multiple loop is concerned one can argue similarly with the others as well.

(Refer Slide Time: 47:19)

Multi-Loop Legal Direction Vectors - 2

Loop 2

- $S1 \theta_{(=, <)} S2$, $S1 \theta_{(<, =)} S2$, $S2 \theta_{(=, <)} S1$, and $S2 \theta_{(<, =)} S1$ are all possible
- $S2 \theta_{(=, =)} S1$ and $S1 \theta_{(=, =)} S2$ are not possible

Loop 2

```
for I = LI to UI do {
  for J = LJ to UJ do {
    if (...) then
S1: ...
    else
S2: ...
    end if
  }
}
```

```
I = 1
  J = 1  S1
  J = 2  S2
I = 2
  J = 1  S2
  J = 2  S1
I = 3
  J = 1  S1
  J = 2  S2
```

YN Srikant Automatic Parallelization

What happens if we have an if then else inside multiple loop. Again, the only change is S1 and S2 cannot execute at the same time, either S1 or S2 will execute in any iteration of J. Again, the trace indicates various possibilities. Again as usual, let us see what is not possible. S2 theta equal to equal to S1 is not possible, well for the same simple reason that S2 and S1 cannot even execute in the same iteration. So, S1 theta equal to equal to S2 is not possible.

Let us see, how S2 theta equal to comma less than S1 is possible. We have to look at the same value of I, but different values of J. We are looking at S2 here and S1 here, so this is same value of I and different values of J. This gives you S2 theta equal to comma less than S1, so these two give you that.

What about S1 theta equal to comma less than S2? That is given by this (Refer Slide Time: 48:32) S1, S2 same value of I and different values of J. Similarly you can get, S2 theta less than, less than S1. So, pick any S2 here and pick any other S1 in another one S2 is here and S1 is here, so this is equal to these two **are this is** gives you I equal to 2 and I equal to 3 component is less than J equal to 1 and J equal to 1 second component is equal to. If you wanted less than, you have to pick some other value of I where J equal to 3 or something like that has S1. That is how, the legal direction vectors are all computed using the syntax of the loop.

(Refer Slide Time: 49:20)

The slide is titled "Data Dependence Equation". It contains the following text:

```
Given a program segment such as:  
for  $I_1 = L_1$  to  $U_1$  by  $N_1$  do {  
  ...  
  for  $I_d = L_d$  to  $U_d$  by  $N_d$  do {  
     $S_v$ : ...  $X(\dots f(I_1, \dots, I_d), \dots)$  ...  
     $S_w$ : ...  $X(\dots g(I_1, \dots, I_d), \dots)$  ...  
  }  
  ...  
}
```

In the bottom right corner of the slide, there is a small video inset showing a man with grey hair, wearing a dark shirt, sitting at a desk with a laptop. The video title at the bottom of the inset reads "YN Srikant Automate Paralleliz".

Now, we come to the data dependence equation. Suppose, we are given a program segment such as I_1 , it is a nested loop S_v and S_w are the two statements which are nested inside a d depth loop.

First loop is I_1 equal to L_1 to U_1 by N_1 . Second would be I_2 equal to L_2 to U_2 by N_2 etcetera; I_d would be L_d to U_d by N_d . S_v the exact statement itself does not matter to us, we are only interested in the two array expressions in S_v and S_w because the data dependence has to be computed between any pair of array expressions. So, X and X same array that is very important for us, first one has a subscript f of I_1 to I_d - a function of the loop indices I_1 to I_d and if the same subscript S_w has X of g I_1 to I_d .

Again is a function of I_1 to I_d but, a different function g . This is the generalized program segment that is given to us and we want to check whether, there is dependence from S given S_v to S_w or from S_w to S_v .

(Refer Slide Time: 50:48)

Data Dependence Equation

- Suppose that $\vec{i} = (i_1, \dots, i_d)$, and $f(\vec{i})$ and $g(\vec{j})$ are given by

$$f(\vec{i}) = A_0 + \sum_{k=1}^d A_k i_k$$
$$g(\vec{j}) = B_0 + \sum_{k=1}^d B_k j_k$$

- We try to find solutions \vec{i} and \vec{j} for \vec{i} that satisfy the dependence equation

$$f(\vec{i}) = g(\vec{j})$$

such that the DV is also satisfied

$$\theta(i_k) \leq \psi_k \leq \theta(j_k)$$

YN Srikant Automate Parallelism in #k

So, the formulation is follows. Suppose, \vec{i} is i_1 to i_d . Now, \vec{i} can be written as A_0 plus sigma of k equal to 1 to d $A_k i_k$, why? Recall that we want to impose certain restriction on the array subscripts. The dependence testing that is dependence analysis cannot be done on arbitrary subscripts. We do not have the mathematical techniques to do that. We always restrict them to some form of equations and the easiest and most widely used form is the linear dependence equation. The subscripts are the linear functions of the loop indices.

So, A_0 is a constant $A_k i_k$ is a linear dependence on i_k , so $A_k i_k$. We would have A_0 plus $A_1 i_1$ plus $A_2 i_2$ $A_3 i_3$ plus $A_d i_d$, so that is going to be f of \vec{i} . Similarly, g of \vec{j} would be some other constant B_0 not plus $B_1 i_1$ plus $B_2 i_2$ etcetera $B_d i_d$. We would have these two as the linear functions of i_1 to i_d , which are the subscripts and we want to check whether f of \vec{i} equal to g of \vec{j} for some value of \vec{i} and \vec{j} . What are \vec{i} and \vec{j} ? They are the values of various loop indices, so i_1 to i_d .

Freeze the loop at any point as we have seen, do we get some values of \vec{i} and \vec{j} for this \vec{i} , which make f of \vec{i} equal to g of \vec{j} . In this, we want to make sure that they have exactly the same value. If that can happen, then the direction vector is should also be satisfied $\theta(i_k) \leq \psi_k \leq \theta(j_k)$ so this could be less than greater than etcetera.

We want to find some loop instance values i_1, i_2, i_3, i_d . Similarly, another set of loop instance values j_1, j_2, j_3, j_d for same set of loops again, such that f of \vec{i} is same as g

of \bar{j} . So if this happens, then that is going to be a conflict when we access an array. In other words, we are going to produce a value in some particular configuration of loop indices and use that value in some other set of loop indices. If you try to run these two sentences - statements in parallel - then the dependences would be lost and hence that would be some incorrect computation.

(Refer Slide Time: 53:49)

Data Dependence Equation

- If we use a normalized index I_k^n instead of I_k , where

$$I_k = I_k^n N_k + L_k$$
- I_k^n satisfies the inequality $0 \leq I_k^n \leq (U_k - L_k)/N_k$ and has increment one
- The dependence equations now become

$$f^n(\bar{I}^n) = A_0 + \sum_{k=1}^d A_k N_k I_k^n + \sum_{k=1}^d A_k L_k$$

$$g^n(\bar{I}^n) = B_0 + \sum_{k=1}^d B_k N_k I_k^n + \sum_{k=1}^d B_k L_k$$
- Finding solutions \bar{i}^n and \bar{j}^n for \bar{I}^n to the normalized equations is equivalent to finding solutions to the original equation

YN Srikant Automatic Parallelization

You could also use a normalized index in other words L lower bound is 1, increment is 1 etcetera. So, if we use a normalized index then I_k would be $I_k N$ which is a normalized index N_k plus L_k . We already saw some transformation to do this. Then the dependence equations can be read written using normalized index also.

We will assume that, all our indices are normalized, so we do not want to do normalization; this is just a transformation which is done by the compiler. Finding solutions - normalized solutions \bar{i}^n and \bar{j}^n for \bar{I}^n to the normalized equation is equivalent to finding solutions to the original equation.

(Refer Slide Time: 54:39)

The GCD Test - 1

- The dependence equation
$$A_1x_1 + \dots + A_nx_n - B_1y_1 - \dots - B_ny_n = B_0 - A_0$$
has a solution *if and only if* $GCD(A_1, A_2, \dots, A_d, B_1, B_2, \dots, B_d)$ divides $B_0 - A_0$
- The GCD test is quick but not very effective in practice
- The GCD test indicates dependence whenever the dependence equation has a solution anywhere, not necessarily within the region imposed by the loop bounds

YN Srikant Automatic Parallelization

Let us very quickly look at the GCD test. The GCD test determines whether the dependence equation that is A_1x_1 plus A_2x_2 etcetra A_nx_n minus B_1y_1 minus B_2y_2 etcetra minus B_ny_n is equal to B_0 minus A_0 , that is this equation (Refer Slide Time: 55:33).

(Refer Slide Time: 55:01)

Data Dependence Equation

- Suppose that $\vec{l} = (l_1, \dots, l_d)$, and $f(\vec{l})$ and $g(\vec{l})$ are given by
$$f(\vec{l}) = A_0 + \sum_{k=1}^d A_k l_k$$
$$g(\vec{l}) = B_0 + \sum_{k=1}^d B_k l_k$$
- We try to find solutions \vec{i} and \vec{j} for \vec{l} that satisfy the dependence equation
$$f(\vec{i}) = g(\vec{j})$$
such that the DV is also satisfied
$$\theta(\vec{i}_k) \leq \psi_k \leq \theta(\vec{j}_k)$$

YN Srikant Automatic Parallelization

So, $f(\vec{i})$ equal to $g(\vec{j})$, so $f(\vec{i})$ equal to $g(\vec{l})$. So this A_0 plus sigma $A_k I_k$ is equal to B_0 plus sigma $B_k I_k$ expanded will really give you this particular equation (Refer Slide Time: 55:17).

(Refer Slide Time: 55:17)

The GCD Test - 1

- The dependence equation
$$A_1x_1 + \dots + A_nx_n - B_1y_1 - \dots - B_ny_n = B_0 - A_0$$
has a solution *if and only if*
 $GCD(A_1, A_2, \dots, A_d, B_1, B_2, \dots, B_d)$ divides $B_0 - A_0$
- The GCD test is quick but not very effective in practice
- The GCD test indicates dependence whenever the dependence equation has a solution anywhere, not necessarily within the region imposed by the loop bounds

YN Srikant Automatic Penetration

We are assuming N loop depth so this is the expanded version. This has a solution if and only if the GCD of all the coefficients $A_1, A_2, A_d; B_1, B_2, B_d$ divides this constant B_0 minus A_0 . This is a very well known theorem and this is called as a Diophantine equation and this is easy to apply. The GCD test is extremely quick but, it is not very effective in practice that is the problem.

We are going to see examples in the next class. The GCD test really indicates dependence, whenever the dependence equation has a solution anywhere but not necessarily in the region imposed by the loop bound. What happens is GCD test will say there is a solution, the loop may run from 1 to 10 but the solution may be given when i equal to 20 or something like that so this is not relevant to us. GCD test simply tells you that there is a solution but it does not worry about the loop bounds. This is the difficulty, we are going to see some examples and continue with other powerful test in the next class. Thank you.