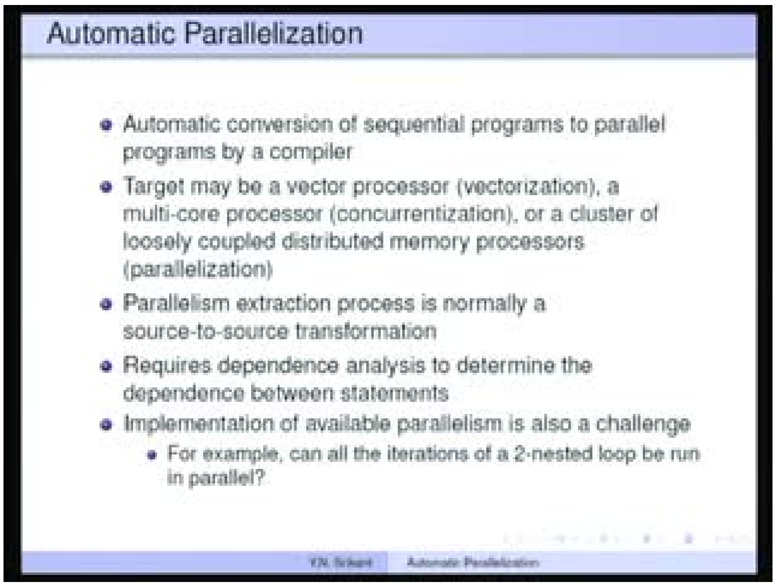
Compiler Design

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

Module No. # 14
Lecture No. # 34
Automatic Parallelization

Welcome to the lecture on automatic parallelization, typically we look at conversion of sequential programs to parallel programs by a compiler this is a process of automatic parallelization.

(Refer Slide Time: 00:19)



Automatic Parallelization

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

Y.N. Srikant Automatic Parallelization

First of all, we must understand why automatic parallelization is essential, you see the parallel programming of vector processors or multi core processors or multi processors is an extremely difficult task; for example, even with 4 core processors, we can have 4 threads. You can even have 10 threads, if enough speed of the processor is high. Managing all the 10 threads, their communication and sharing of data structures etcetera, is definitely not easy for a programmer.

Specially now, programmers cannot write concurrent programs without difficulty that is the reason why automatic parallelization which converts sequential programs to parallel programs, which actually guarantees correctness of the program- parallel program and that has become popular.

Research in this direction has been going on from 1969 by David Cook and others. It actually reached some saturation point somewhere in the early 90s, but with the advent of multicore processors it has really picked up vigorously all over again. The target for a parallel automatic parallelizer may be a vector processor in which it is called vectorization; it could be a multicore processor and we call this process as concurrentization; it could be a cluster of loosely coupled distributed memory processors, so we call this processor parallelization in general.

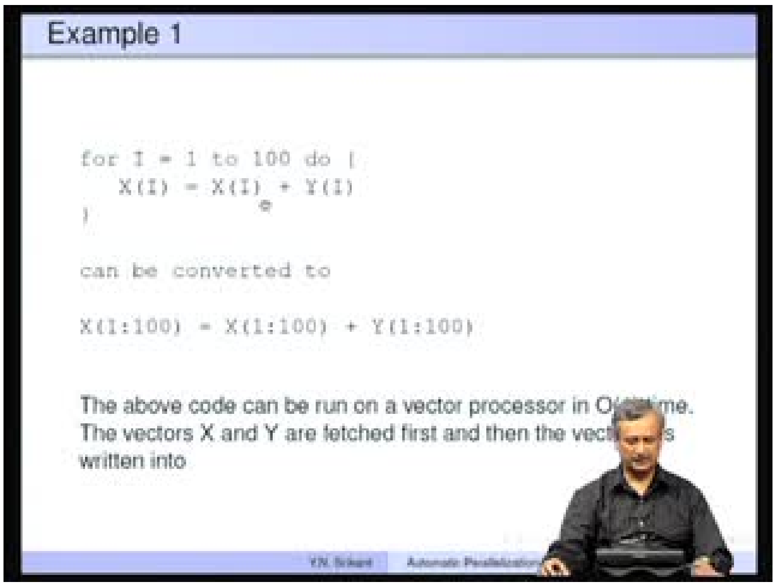
Parallelism extraction process is normally a source to source transformation, there are reasons for this. Possibly, we could do this on binary but the effectiveness is much lower. What happens is, when we look at sequential program the loops with array access are the most important, the others scalars and non-loop parts of the program are important, but they do not really give you too much of speed up even when it runs on a parallel processor.

When you have arrays, the basic question that we must ask during parallelization is, when we have two array accesses are these two accesses to the same location of the array. If it is a read access from both of them then there is no problem, but if one of them writes and the other reads or both of them writes to the same location then they cannot do so in parallel, they must be asked to do in the program order. This is the basic question that is asked and this is as during a phase called dependence analysis.

To answer this question we require the expression of the array subscript itself, when we use an array we have an index expression, so we need the expression in order to perform this dependence analysis. That is the reason why we would like the parallelism transformation sequential to parallel program transformation to be a source to source transformation. This requires dependence analysis to determine dependence between statements, which is what I mentioned just now, the implementation of available parallelism is also a challenge.

If you have a single loop, let us say it runs from 1 to 1 million, there are 4 processor's cores available, you could simply say 2.5 lakh iterations per thread would run on each processor and there would be 4 threads running. So, within the threads the 2.5 lakh iterations would run in sequential mode, but what if there is a doubly nested loop? Let us say both the loops can run in parallel, in such a case, can we actually benefit from parallelizing both the loops, one nested inside another. Does the processor have enough parallelism, enough number of processors to give you benefit in such a case? This is the question that is asked when we look at implementation of available parallelism.

(Refer Slide Time: 05:25)



Example 1

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

can be converted to

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

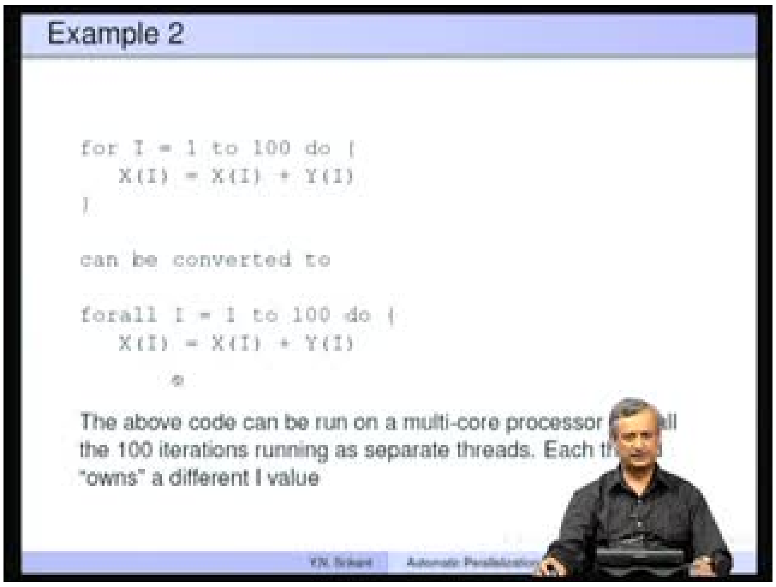
The above code can be run on a vector processor in O(1) time. The vectors X and Y are fetched first and then the vector is written into

Y.N. Srikant Automatic Parallelization

Let us start with some simple examples, some of these we saw during introduction to optimization. Here is a for loop, for I equal 1 to 100 do X I equal to X I plus Y I, this code can be very easily converted to a vector processor code X of 1 colon 100 equal to X of 1 colon 100 plus Y of 1 colon 100. The code can run on a vector processor in constant amount of time - O 1 amount of time. The reason is we assumed that there is enough number of vector registers available, let us say 100 of them. The X and Y components are fetched in parallel, then they are added and then they are written into the X vector register, this is how it goes. In the entire read process, 100 items of X and Y will happen in constant amount of time, the addition takes constant amount of time and the writing also takes constant amount of time.

This vectorization is possible because there is no dependence between these iterations; for example, this writes X1 equal to X1 plus Y1, the second iterations writes X2 equal to X2 plus Y2. Each iteration writes and reads different X and Y items, so there is no dependence from one iteration to another.

(Refer Slide Time: 06:57)



Example 2

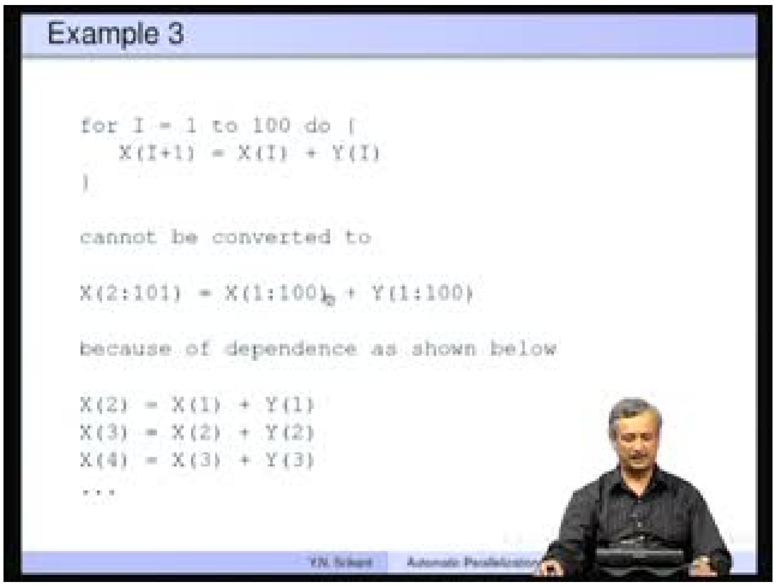
```
for I = 1 to 100 do {
   X(I) = X(I) + Y(I)
}

can be converted to

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

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

Y.N. Srikant   Automatic Parallelization

The same example, the program can also run on a multi core processor. So, what we write is for all I equal to 1 to 100 do X I equal to Y I X I plus Y I. What happens is there will be 100 iterations running as separate threads assuming that there are 100 cores possible. Each thread actually owns a different I value, so for I equal to 1 there is a thread, for I equal to 2 there is another thread and so on. Each thread is responsible for executing one of the iterations, so X I equal to X I plus Y I. Since, the I values are all different, again each thread writes and reads from a different X location and Y location that is how it runs on a multi core processor, weather it is useful to run 100 threads or not is a matter of detail. In practical situations we may not be using so many threads that we will see later.

(Refer Slide Time: 08:04)



Example 3

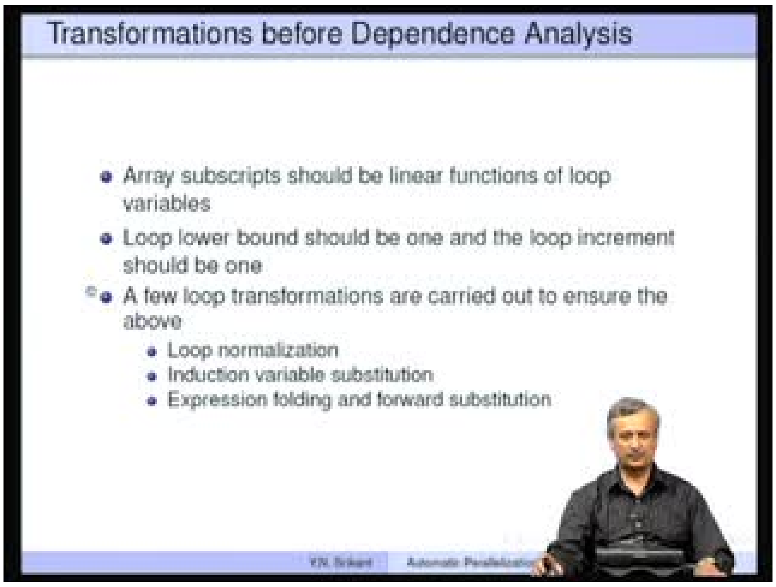
```
for I = 1 to 100 do {
   X(I+1) = X(I) + Y(I)
}

cannot be converted to

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

because of dependence as shown below

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

Y.N. Srikant Automatic Parallelization

In the next example something that cannot be parallelized, so you have for I equal to 1 to 100 do X of I plus 1 equal to X I plus Y I. If you simply right these as vector code X of 2 colon 101 equal to X of 1 colon 100 plus Y of 1 colon 100 this would be wrong, the reason is the dependence informally. Let us see what this dependence is, so let us substitute I equal to 1, 2, 3 etcetera and see what happens to these statements. With I equal to 1, the statement becomes X2 equal to X1 plus Y1; with I equal to 2, it becomes X3 equal to X2 plus Y2; with I equal to 3, it becomes X4 equal to X3 plus Y3 etcetera. You can see that with I equal to 1, we write into X of 2; in I equal to 2, we read from X of 2. In other words, what is written in the first iteration is read in the second iteration. Similarly, what is written into in the second iteration that is X of 3 is read in the third iteration X of 3 again.

There is a dependence from one iteration to the next iteration, what is written in this iteration is read in the next iteration. Because of this dependence we cannot run this code in vector mode, because the vector statement which is given here implies that all the X values are at first and then added to the Y values and placed in X, which would be incorrect, the dependence is not actually satisfied. This is a very simple example of code that cannot be parallelized.

(Refer Slide Time: 09:52)



Transformations before Dependence Analysis

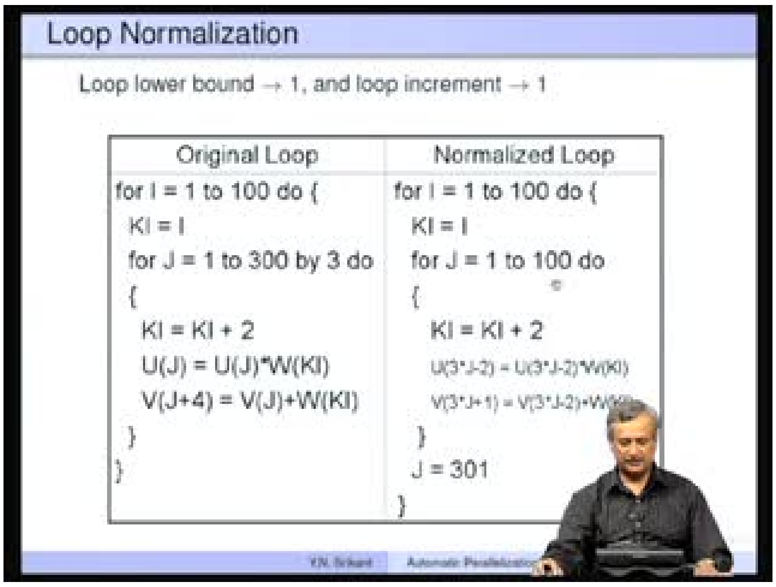
- Array subscripts should be linear functions of loop variables
- Loop lower bound should be one and the loop increment should be one
- A few loop transformations are carried out to ensure the above
  - Loop normalization
  - Induction variable substitution
  - Expression folding and forward substitution

Y.N. Srikant Automatic Parallelization

There are some assumptions on the programs in the form of array subscripts, things like that must be satisfied for performing dependence analysis, which is one of the most important steps in parallelization. For example, array subscripts should be linear functions of loop variables. I already mentioned that array is the most important in parallelization, so the form of the subscripts, the index expressions in the array access should be linear functions of loop variables. In other words, I cannot have I square plus J square or I star J and things like that, I must have K star I plus m star J etcetera, where K and m are constants, so linear functions of loop variables only.

Loop lower bound should be 1 and loop increment should be 1, so this is to make the analysis simpler and this is easy to actually make sure of as we will see very soon. A few transformations on loops are carried out to ensure the above conditions. You cannot really make a non-linear function of linear variables into a linear function, but some normalization is possible, so loop normalization is possible, induction variable substitution is possible, expression folding and forward substitution are possible, we will see what these are.

(Refer Slide Time: 11:27)



Loop Normalization

Loop lower bound → 1, and loop increment → 1

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

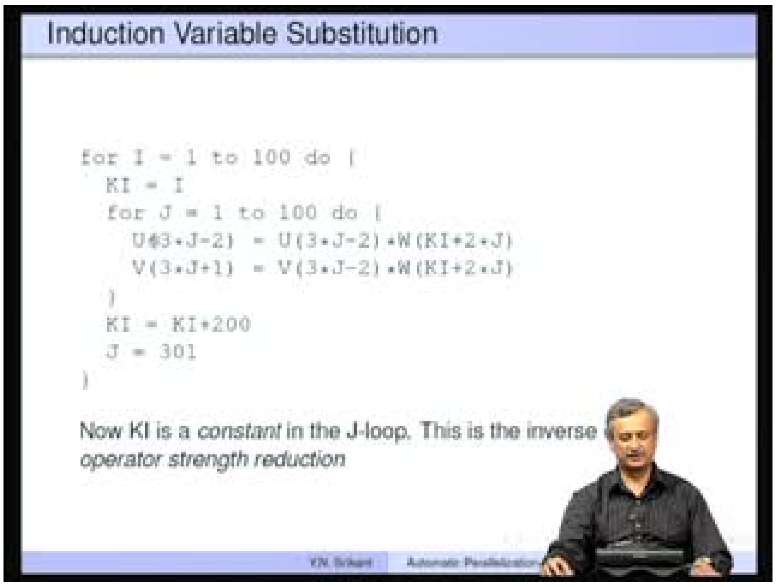
Y.N. Srikant Automatic Paralleliza

Let us take a simple example, the original loop is like this I equal to 1 to 100 do KI equal to I for J equal to 1 to 300 by 3 do and there are some statements inside. The first and for most transformation is we want to make the loop lower bound 1 and the loop increment as 1. In both loops, the loop lower bound is indeed 1, so this is I equal to 1 and this is J equal to 1, so that is satisfied already, but the loop increment is not 1 in both the loops. In this loop it is 1 but in this loop it is 3, so we want to change the loop to look as in this picture I equal to 1 to 100 do KI equal to I, J equal to 1 to 100 do, so we want to change the increment to 1.

Obviously, when the increment is changed to 1, the expressions which use J inside will have to undergo some change. Here, if you just expand J equal to 1, 2, 3 etcetera we would have had U1 equal to U1 star WKI, U2 equal to U2 star WKI, etcetera. Sorry then not J equal to 2 J equal to 1 and then plus 3 J equal to 4, so U of 4 equal to U of 4 plus into W of KI etcetera. We must jump J by 3, so the way of doing it is multiply J by 3 and then subtract 2. We have U of 3 star J minus 2 equal to U of 3 star J minus 2 star W of KI. If you have J equal to 1 this would be 3 minus 2 which is 1. If you have equal to 2 this becomes 3 into 2, 6 minus 2, 4 etcetera. The array subscripts are going to take the same values as before, but the loop has been changed to run from 1 to 100.

Similarly, this is V of J plus 4, so we would have V of 5, V of 8, etcetera. This 3 star J plus 1 is similar, J equal to 1 this becomes V of 4, J equal to 2 this would be 3 into 2 6 plus 1 etcetera. We will actually have 7 here and 7 here, then you know 7 plus 3 10, so 3 into 3 plus 1 is 10 etcetera, so the loop indices are changed and the subscripts have been changed in order to satisfy this loop increment equal to 1.

(Refer Slide Time: 14:17)



Induction Variable Substitution

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

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

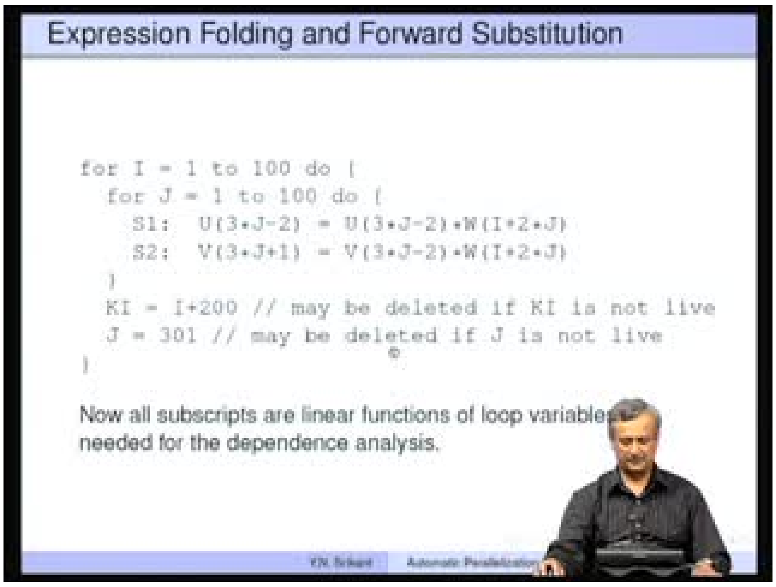
The next preprocessing on the loop is called induction variable substitution, so recall we had induction variable elimination and along with it we had strength reduction. Here, we are going to do something similar but in inverse fashion, let us see what we are doing. Recall this, keep this in your mind we have normalized loop, we changed the loop indices, now the loop runs from 1 to 100 as usual then KI equal to I and J equal to 1 to 100 that was the normalized loop. U equal to 3 star J minus 2 etcetera this was as before, V of 3 star J plus 1 etcetera this was also as before, then we have KI equal to KI plus 200 and J equal to 301.

What has changed? The change is in the W part not in the U or V part, the reason is here is a variable KI which is used in the W part; W of KI, W is accessed using the KI alright. Actually, KI is incremented by 2 in the J loop, so it is actually an induction variable dependent on J for the increment. It starts with I, then K equal to I, then I plus 2, I plus 3, I plus 4, I plus 5, etcetera in the J loop.

This fact it is actually made use of what we do, is we have KI equal to I and then instead of having KI equal to KI plus 2 in the loop we actually insert W of KI plus 2 star J, so every time J increments, this increment is by 2. We effectively do KI plus 2 right here, at the end we must give the value of KI that it would have got previously, so that would be here KI plus 200. That is how we have eliminated the addition to KI incrementing KI by 2 every time, so now KI is a constant in the J loop; this does not vary in the J loop that is the advantage.

This is the inverse of operator strength reduction that we perform during induction variable elimination; we have actually introduced 2 star J here. In fact, we had KI equal to KI plus 2, but we have introduced KI plus 2 star J here, so we are doing more work. But, we have eliminated the dependence on the incrementing of KI within this loop; KI now is a constant within this loop.

(Refer Slide Time: 17:20)



Expression Folding and Forward Substitution

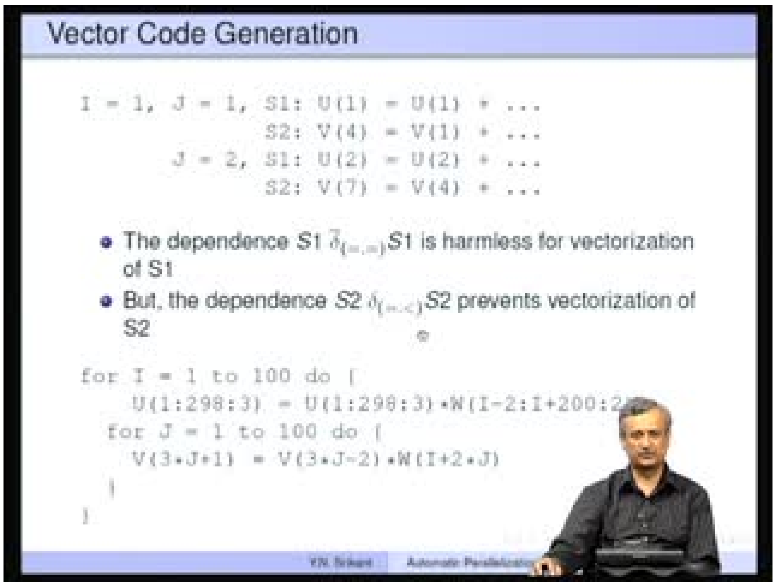
```
for I = 1 to 100 do {
  for J = 1 to 100 do {
    S1:  U(3*J-2) = U(3*J-2)*W(I+2*J)
    S2:  V(3*J+1) = V(3*J-2)*W(I+2*J)
  }
  KI = I+200 // may be deleted if KI is not live
  J = 301 // may be deleted if J is not live
}
```

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

So, with that done the next step is expression folding and forward substitution. See this, we had KI equal to I here, so we just do forward substitution and replace this KI by I, so that is what we do here. So W I plus 2 star J is what we get here, the rest here also I plus 2 star J, the rest of the program remains the same here, since KI was a constant we could replace it by I, so KI is I plus 200 and J is 301. Suppose KI and J are actually not used, later on they do not live after this point, then in that case we can delete these two statements, they will not be needed, otherwise we need to retain them.

Now, you can see that all subscripts are linear functions of loop variables as needed for the dependence analysis. If you had looked at the original loop, see this KI is something they are in the loop not normalized, once we normalize it this KI was still a problem, because it was some variable which was not linearly dependent on I and J. So, in various steps we actually converted KI - the dependence on KI to dependence on I and J. Thereby the subscripts in this particular loop are all now linear and they are ready for dependence analysis.

(Refer Slide Time: 18:57)



Vector Code Generation

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

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

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

Y.N. Srikant | Automatic Parallelization

Then, how does one generate vector code? Let us say, let us look at a glimpse of this process and then take up the details. You are given a loop like this, it has all the loops that are normalized, all the subscripts of arrays etcetera are all linear functions of the index variables, so then what we do? Suppose, you look at the previous loop, this particular loop expands it by giving values for I and J, so that is what I have done here.

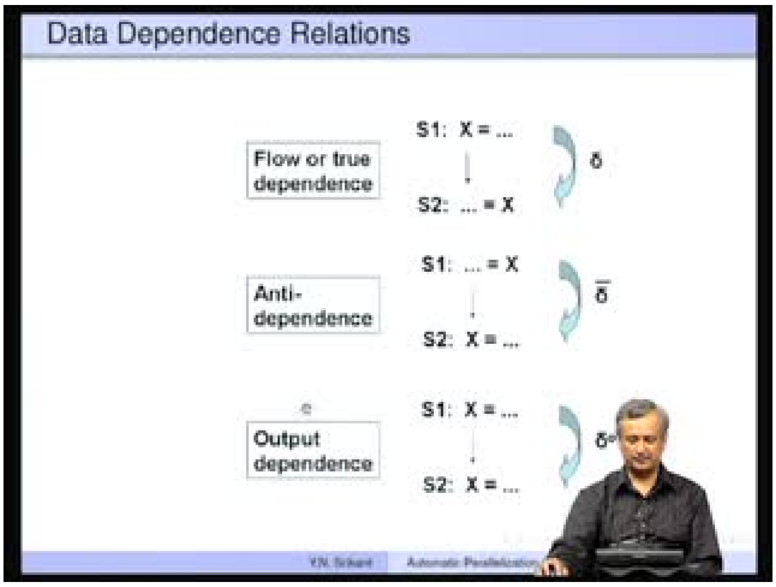
So I put I equal to 1 and then J equal to 1, the statement S1 is this statement. You can observe that 3 star J minus 2 3 star J minus 2 I plus 2 star J, this is identical, this and this are identical. Similarly, 3 star J plus 1, 3 star J minus 2 and then I plus 2 star J, so just remember this. Now, S1 becomes U1 equal to c1 U1 plus something, S2 becomes V4 equal to V1 plus something. When we change J to 2 S1 becomes U2 equal to U2 plus something and S2 becomes V7 equal to V4 plus something.

You can observe that this particular U1 is read and then written into; this is called as anti-dependence, so this type of anti-dependence U2; then U2 here is harmless for vectorization. It does not really pose any problems, because all the vector registers are supposed to be read first and then written into, so this does not pose problems. But, the dependence S2 delta S2 prevents vectorization of S2. So, S2 is here and then it is writing into V4, then S2 in J equal to 2 is actually reading from V4; that means form this sentence S2 to this sentence S2 there is a dependence that dependence is denoted as delta, it is write and then read, it is called as a flow dependence. If you observe the I index, J index and their relationships this is happening in the same value of I, so there is a small subscript called equal to here. Then it is happening in different values of J, writing is happening in J equal to 1 and reading is happening in J equal to 2 with 1 less than 2, so the second subscript is less than, so that is the relationship between the two subscripts that we have indicated here. This will become very clear, this is called as a direction vector and this will become very clear much later.

So, the dependence and direction vector are the two items that we need to compute very accurately for vectorization and parallelization, because of this dependence vectorization of S2 is not possible, vectorization of S1 is definitely possible. We do partial vectorization for I equal to 1 to 100 do the statement S1 is vectorized, U of 1 to 298 colon 3, U colon 1 to 298 colon 3 star W and etcetera.

The S1 statement has been completely vectorized, it runs in vector mode, whereas for J loop it contains the statement S2 which cannot be vectorized, so the J loop and I loop runs sequentially on this particular S2. This is the vector code generation process, so essentially we compute the dependences and the direction vectors. Then we are going to look at some conditions on the dependences in order to tell us whether vectorization is possible or not possible.

(Refer Slide Time: 23:08)


Data Dependence Relations

Flow or true dependence	S1: X = ... S2: ... = X	δ
Anti-dependence	S1: ... = X S2: X = ...	$\bar{\delta}$
Output dependence	S1: X = ... S2: X = ...	δ^o

Y.N. Srikant Automatic Parallelization

So, let us formally look at the dependence relations, this was an introduction that they give you to vectorization so far. There are three types of dependence which are very important, one is the flow or true dependence, the second is the anti-dependence and the third is the output dependence. In flow dependence the two statements S1 and S2 share a variable X, S1 writes into it and then S2 in the same execution order reads form it. Later this is called as flow dependence because the value flows from this particular definition of X to this use of X.

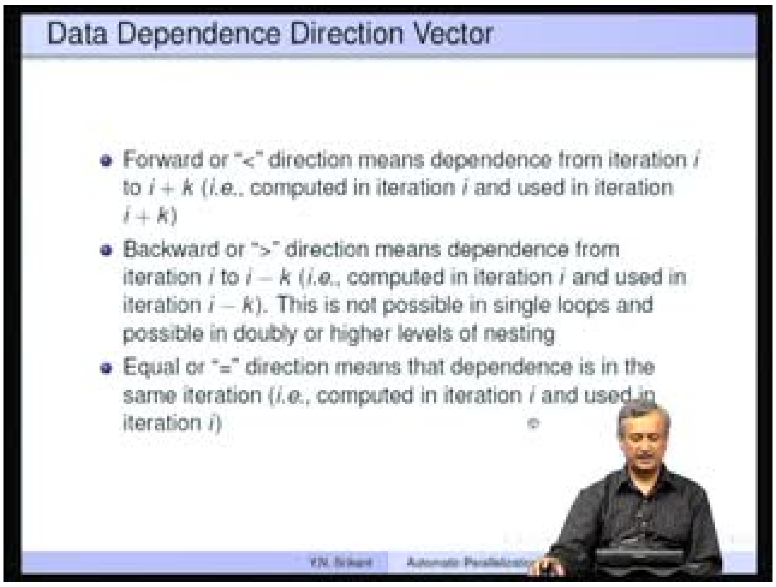
Anti-dependence is the other way, S1 reads from X and then S2 writes into X, so you first read and then write, so this is called as an anti-dependence; this is shown as delta bar. Output dependence is S1 writes into X and then S2 also writes into X, but then you cannot really change the order of S1 and S2, so this is called as output dependence. For example, if output dependence is violated and the output device is a printer then the printed output may appear in jumbled form.

Here we may actually end up, you know if you write first and then read you may get the wrong value, whereas if you read first and then write you will still get wrong value. In all the three forms of dependence we are really very seriously worried about the flow dependence, it is called as true dependence and it cannot be removed by any means. In the case of anti-dependence, if we really store the value of X in some temporary right from the beginning, then we can write into X any time. So, in that particular dependence we do not have to make S to wait until S1 is completed, if we have stored the value of X in some temporary variable we can dispense with that.

Instead of X, in that case, we are actually going to use the variable t in S1 and thereby get rid of the dependence. So, t would have stored the value of X right in the beginning of the group of statements, so anti dependence can be gotten rid of by using some temporary variables.

In the case of output dependence, it is not possible to get rid of it by using temporaries, but we can rename variables and get rid of it. For example, instead of using X and X in both places if we simply change this to X and Y two different variables, such renaming is always possible, we can get rid of output dependence. Since, we can get rid of anti and output dependences they are not treated on par with the flow or true dependence; flow or true dependence cannot be gotten rid of. Most of the time we worry about the flow dependence, but definitely if there is a need we will look at anti and output dependence as well.

(Refer Slide Time: 26:34)



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

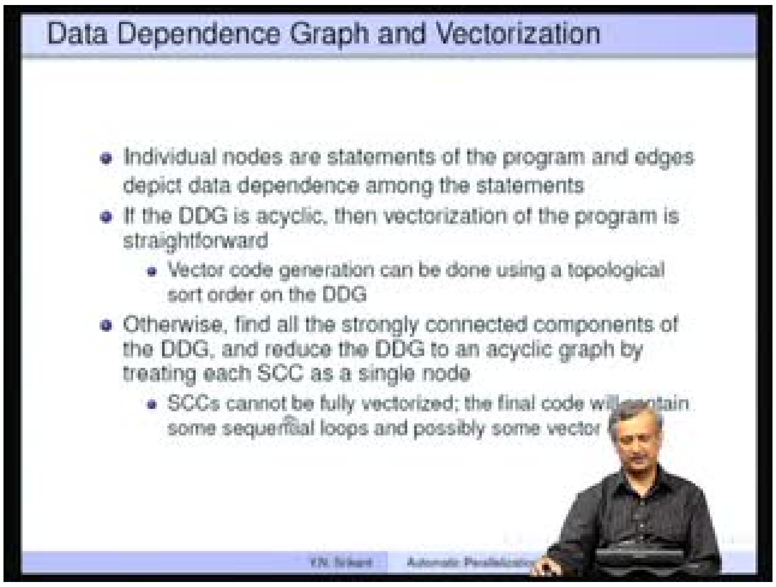
What exactly is this direction vector that I mentioned just now informally let us to understand? The direction vector and the exact computation of direction vectors, it will be taken up later. There are three types of direction vector components possible, one is the forward component, the other is the backward component, third is the equal to component.

If we have the symbol less than it is called as a forward direction vector component, it means that the dependence from iteration I to the iteration I plus K. In other words, we compute a quantity in iteration I say X equal to something and then use it in the iteration I plus K, K greater than say 0. So it is computed in iteration I and used in iteration I plus K that is why this is called as a forward dependence. This less than is a relationship between I and I plus K, I is less than I plus K that is why the symbol is less than but it is read as a forward dependence, the value is carried forward in the iteration space.

Backward dependence is denoted as greater than, so this direction means dependence is from iteration I to the iteration I minus K, in other words you compute in iteration I and you are using it in a previous iteration I minus K, so this sounds a bit weird. Definitely it is weird, it is not possible to have this type of a direction vector component in single loops, because loops run in 1 direction, you cannot come back and execute one of the previous iterations again. As we will see later in doubly nested or higher levels of nesting it is possible to have this type of a component in one of the inner loops. Outer loop cannot have greater than component, but one of the inner loops can definitely have the greater than component in its direction vector.

Equal to direction vector is denoted as equal and that means the dependence is in the same iteration, in other words computed in iteration I and used in iteration I.

(Refer Slide Time: 28:58)



Data Dependence Graph and Vectorization

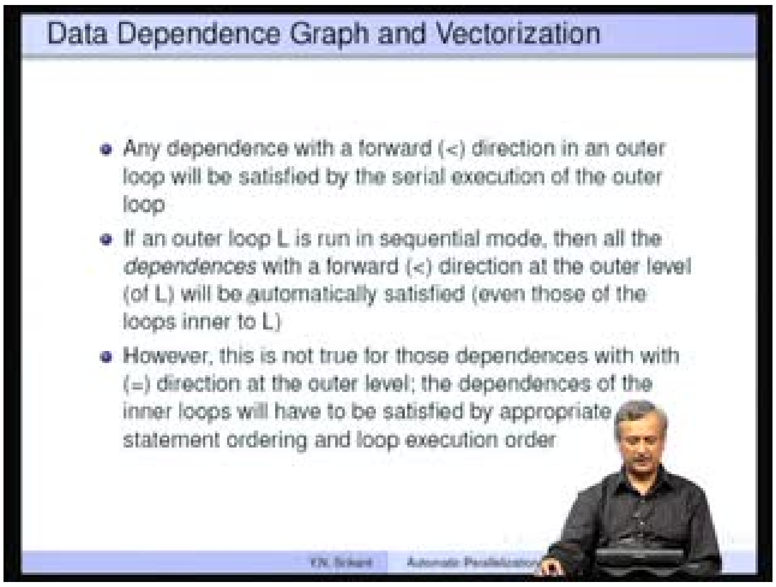
- Individual nodes are statements of the program and edges depict data dependence among the statements
- If the DDG is acyclic, then vectorization of the program is straightforward
  - Vector code generation can be done using a topological sort order on the DDG
- Otherwise, find all the strongly connected 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

So here, we had that example. For example, here you know U1 was here and U1 was here, we are actually reading something here and then writing into it again, so this is in the same iteration of I and also J, so that is why the two direction vectors are equal here. Whereas, wrote into V 4 in I equal to 1 and J equal to 1, read in I equal to 1 and J equal to 2 that is why the direction vector component is equal to and less than.

Now, what exactly is a data dependence graph and how is it related to vectorization? You take each statement, so we are considering statements we are not looking at binary or intermediate code here. Each statement is a node, so in the data dependence graph individual nodes are statements of the program and edges depict data dependence among the statements, so we are going to see some examples to understand all this better.

If the data dependence graph is acyclic then vectorization of the program is straight forward, so we will see how this happens. Vector code generation can be done using a topological sort order on the data dependence graph. If suppose, the data dependence graph is cyclic then there is a complication, so what we do is we find all the strongly connected components of the data dependence graph and then reduce the DDG to an acyclic graph by treating each strongly connected component as a single node. So the SCCs - the strongly connected component themselves cannot be fully vectorized, the final code will have some sequential part and some vector part. Again, we are going to see some examples of this.

(Refer Slide Time: 31:10)

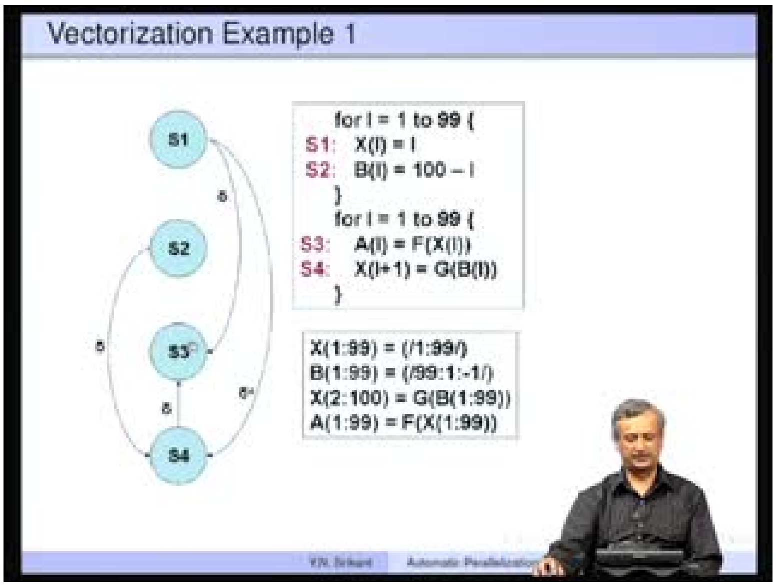


Data Dependence Graph and Vectorization

- Any dependence with a forward (<) direction in an outer loop will be satisfied by the serial execution of the outer loop
- If an outer loop L is run in sequential mode, then all the *dependences* with a forward (<) direction at the outer level (of L) will be automatically satisfied (even those of the loops inner to L)
- However, this is not true for those dependences with with (=) direction at the outer level; the dependences of the inner loops will have to be satisfied by appropriate statement ordering and loop execution order

Y.N. Srikant Automatic Parallelization

(Refer Slide Time: 31:18)



Vectorization Example 1

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

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

Y.N. Srikant | Automatic Parallelization

We will come back to this particular slide a little later, after looking at vectorization example. Here is the original program and here is the vectorized program, so you have for I equal to 1 to 99, S1 is X I equal to I, S2 is B I equal to 100 minus I, then you have another loop for I equal to 1 to 99, S3 is A I equal to F of X I, F is some function of X I and X of I plus 1 is G of B I, again G is some function of B of I.

Here is the data dependence graph for these four statements S1, S2, S3, S4 correspond to the four statements S1, S2, S3, S4, this shows that there is a dependence from S1 to S3, so let us understand it. This is S1, this is S3, so this says X I equal to I, so that means X1 equal to 1, X2 equal to 2, X3 equal to 3, etcetera; there is some initialization which has happened. Then in S3 A I equal to F of X I, F is some function some expression of X of I say X I plus 1, X I plus 2, etcetera, so that means we are reading X of I here and writing into X of I here. There is flow dependence from S1 to S3 that is what is shown here S1 to S3. Since these two are in two different loops, this is I loop and this is another independent loop, there is no direction vector component shown here, direction vector components are shown for accesses within the same loop.

Then we have other dependence from S1 to S4 that is output dependence. So, S1 is again X I equal to I, S4 is X I plus 1 equal to G of B I, so here we have X1 equal to 1, X2 equal to 2, etcetera, here we have X2 equal to something, X 3 equal to something, etcetera. So, we write into X in S1 and after the I loop here terminates the second loop, here terminates the second loop again and writes into locations of X; that means, there is output dependence from S1 to S4.

What about S2 and S4, there seems to be a dependence from S2 to S4 which is of the flow kind. S2 is B I equal to 100 minus I and S4 is X I plus 1 equal to G of B I. So that means, we write into B I here and read B I here, but these two are in two different loops. So, there is no direction vector component, just a dependence writing into B and reading from B, So S2 to S4 there is a flow dependence, which is shown as an arc.

It also shows an arc from S3 to S4 labeled as delta, so S3 to S4, so F of X I means you read from X I, X I plus 1 equal to means you write into X I, but you should observe here that this runs as the right hand side, runs as X1, X2, X3 etcetera, whereas the left hand side in S4 runs as X2, X3 and X4 etcetera. In other words, in I equal to 1 we write into X2, but we read from X1, in I equal to 2 we write into X3, but we read from X2, X2 was written with in the value I equal to iteration I equal to 1. So that means, whatever was written into in S4 is being read in S3, so there is a flow dependence from S4 to S3 which is of the form delta; not from S3 to S4, it is from S4 to S3 the picture is correct.

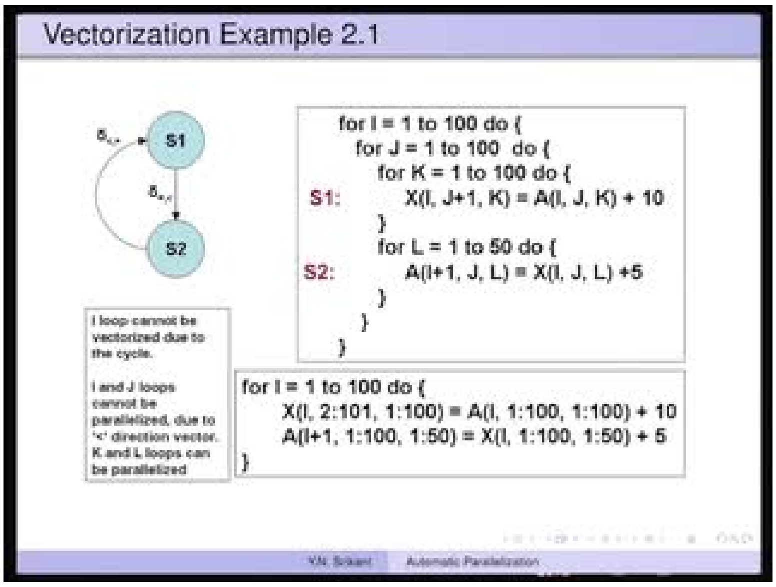
So this is how the dependences are in this particular dependence graph and there is no dependence from S1 to S2. You can see that it does not even share variables arrays, so there is no dependence. Now, it is easy to see that this particular graph is acyclic; you know there are no cycles in graph. So what we said is if the graph is acyclic then vector code generation can be done using a topological sort order on the data dependence graph.

If you do a topological sort order on this particular graph you can actually vectorize S1 and then you can vectorize S2, because there is no dependence from S1 to S2, then you vectorize S4 and finally you vectorize S3, why? S1 is easy to understand, there are no statements which actually S1 is dependent on, so X of 1 to 99 is just a constant 1 to 99 and this is corresponding to S1 here. Then S2 is also not dependent on any other statements, so we can vectorize S2 also straight away. This is S2 B I equal to 100 minus I and we have B of 1 to 99 equal to 99 colon 1 colon minus 1, so this is 100 minus I the constant.

Now, it is not possible to vectorize S3 before S4, because the dependence says whatever is computed in S4 is used in S3. But, we have finished S1 and S2, so we can actually vectorize S4 after them, but we could not have vectorized S4 before S1 and S2 because there are dependences to S4. So S4 is X of I plus 1 equal to G of B I, so X of 2 colon 100 is G of B colon G of B of 1 to 99.

Now the last statement which remains is S3, so that can be very easily vectorized, this is A I equal to F of X I, so A of 199 if F of X of 1 to 99. So we did a topological ordering of the nodes which gave us S1, S2, S4, S3 and we have emitted a vector code because this graph is acyclic.

(Refer Slide Time: 37:44)



Vectorization Example 2.1

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

I loop cannot be vectorized due to the cycle.

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

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

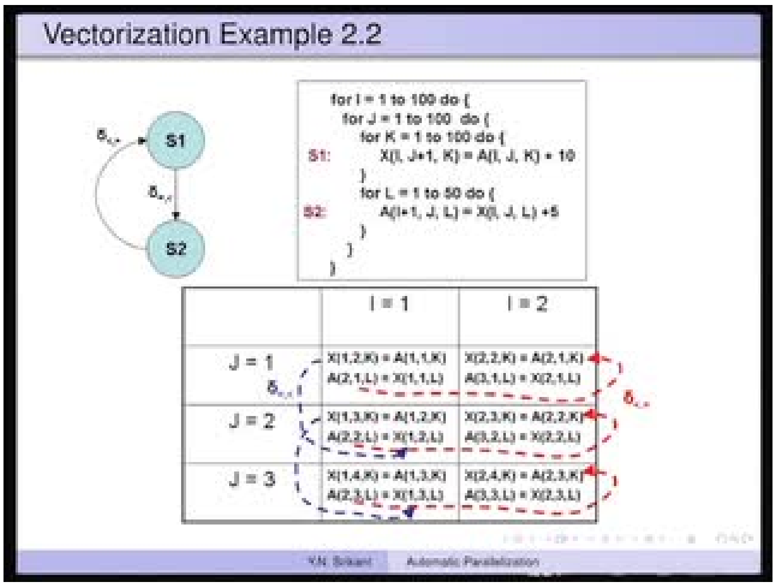
Y.N. Srikant   Automatic Parallelization

Let us look at a slightly more complicated example, this loop is nested, so for I equal to 1 to 100 do, then for J equal to 1 to 100 do, then inside that we have another K loop and a an L loop. So I and J loops are common to both these statements S1 and S2, but S1 is nested only inside K loop and S2 is nested only in the L loop. So K equal to 1 to 100 do X of I comma J plus 1 comma K is equal to A of I comma J comma K plus 10 and L equal to 1 to 50 do A of J I plus 1 comma J comma L is X of I comma J comma L J comma L plus 5.

If you look at this code, here is the dependence graph and it shows that there is a dependence from S1 to S2 which is of the flow kind that is delta, the dependence vector is equal to and less than. Similarly, there is a dependence from S2 to S1 which is of the flow kind that is delta and direction vector is less than and equal to. There is dependence from S1 to S2 rather two dependences from S1 to S2 and both are written in brackets, let me explain why.

As I told you K and L are actually two different depend you know loops, so the only dependence between them can be written as flow and T R output, there can be no direction vector components. That is why just to indicate that the loops are K and L, I have shown this as delta of K L. Delta bar of K L that corresponds to the dependences from S1 to S2. So here is the left hand side and here is the right hand side, so there is a write possibility from here to here and the read possibility from here to here, so that gives you both delta and delta bar for different values of K and L, so that is what this is.

(Refer Slide Time: 40:08)



Vectorization Example 2.2

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

	I = 1	I = 2

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

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

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

Y.N. Srikant Automatic Parallelization

Now, let me show you why the dependences occur, so we have I equal to 1, I equal to etcetera; here, J equal to 1, J equal to 2, J equal to 3, etcetera along this path. Along this column we have expanded the expressions here and substituted I equal to 1, J equal to 1 etcetera. So, the first statement S1 becomes X of 1 comma 2 comma K and here we have A of 1 comma 1 comma K, the second 1 becomes S2 is A of 2 comma 1 comma L which is X of 1 comma 1 comma L, I did not write 5 and 10, I just skipped it for brevity.

Similarly, for I equal to 2 you get X of 2 comma 2 comma K etcetera, so similarly for J equal to 2 I equal to 1 here are the enumerations. It is easy to see that this X 1 2 K is same in I equal to 1 and J equal to 2 X 1 2 L, so for different some values of K and L these are going to be the same and therefore, they will be a dependence from this to this. Similarly, X 1 3 K and X 1 3 L there is going to be a dependence from S1 to S2.

So that is really what is shown as delta equal to and less than, why is it equal to? It is in the same iteration of I, why is it less than? This is J equal to 1 and this is J equal to 2, 1 less than 2, this is J equal to 2 and this is J equal to 3, 2 less than 3 etcetera; delta of equal to and less than is here, so this is the dependence from S1 to S2.

What about the dependence from S2 to S1? So, S2 is A of 2 comma 1 comma L and in I equal to 2 and J equal to 1, you have A comma 2 A of 2 comma 1 comma K, so for some values of K and L these will be equal and therefore, whatever is written it to here in I equal to 1 and J equal to 1 is read in I equal to 1 I equal to 2 and J equal to 1. So, you can similarly see that A 2 2 L and A 2 2 K are identical, A 2 3 L and 2 3 K are also identical. This corresponds to the dependence delta of less than and equal to, why? The first component is I, so I equal to 1 and I equal to 2, so written into I equal to 1, read in I equal to 2, so that means 1 less than 2 that is why the first component is less than.

The second component is equal to because both the dependences are in the same value of J, J equal to 1 and J equal to 1, but I must point out that the J equal to 1 here and the J equal to 1 here are in two different values of I; that mean, S2 different instantiations of the same J loop, but that is not what we consider. We just look at the values of the J this is one of the limitations of the direction vector, it does not consider differenced instantiations of the inner loops but it suffices for our work.

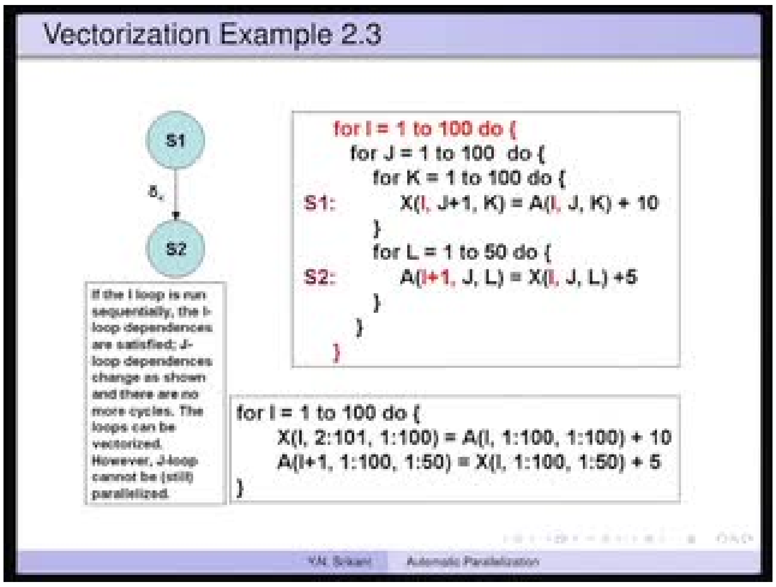
We have the two direction vectors written here, now this is where the problem arises, there is a cycle here right, whenever there is a cycle like this the rule that we are going to follow is given in this particular slide. So, any dependence with a forward direction in an outer loop will be satisfied 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 in the outer level of L will be automatically satisfied, even those of the loops inner to L all these will be automatically satisfied. But, the problem is if this is not true for those dependences with equal to direction at the outer level, the dependences of the inner loops will have to be satisfied by appropriate statement ordering and loop execution order.

So, dependence with less than direction vector in an outer loop will be satisfied by serial execution of the outer loop, this is what is important for us in this case. The I loop cannot be vectorized because of the cycle, so you can see that the cycle is in the I loop. Then once we actually run the I loop in sequential mode the dependences of the inners loops will automatically satisfied, this is what I said. I will show you examples of how this happens a little later, but here just believe that the J K L loops can be vectorized once the I loop is run in sequential mode. The reason is if we run I equal to 1, I equal to 2etcetera in sequential mode, all these executions happens first and then these executions happen. Because of that this dependence which runs from 1 to 2 of I value will be automatically satisfied right, we run this first and then this so this would have been written into first and then we written I equal 2, so this will can be read appropriately without any trouble.

What about this here, the I equal to 1 all these are going to run, so here is the value of I and here is the value of J. We are going to run all the iterations of I in sequential mode and all the iterations of J possibly in parallel mode, so that is the way it is. Let us assume that this works properly, I will show you examples of why this works properly a little later.

Assuming that this is correct J K loops can be vectorized, so I loop run in sequential mode then the J loop and the K loop can actually run in parallel mode. So, X of something equal to A of something plus 10 and A of something equal to X of something plus 5, so this is exactly how it happens.

(Refer Slide Time: 46:52)



Vectorization Example 2.3

$$\begin{tikzcd} S1 \arrow[d, "\delta_<"'] \\ S2 \end{tikzcd}$$

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

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

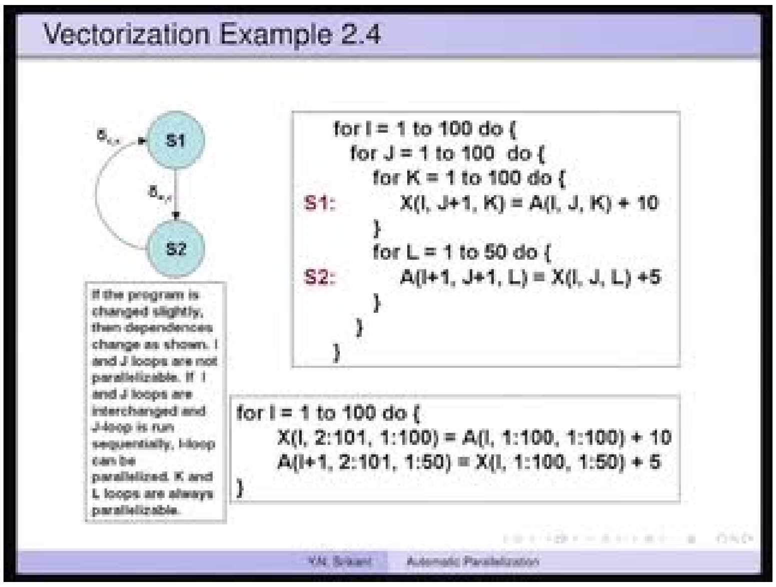
```
for I = 1 to 100 do {
     X(I, 2:101, 1:100) = A(I, 1:100, 1:100) + 10
     A(I+1, 1:100, 1:50) = X(I, 1:100, 1:50) + 5
}
```

Y.N. Srikant   Automatic Parallelization

Here is the reason why I promised that I will give you reasons why vectorization can happen in the J loop and the K loop. Here is the reason, let us assume that I loop run in sequential mode, then the dependence diagram changes as I have shown here. The dependences changes for example, the I dependences are all gone, so there was a dependence here from this to this S2 to S1 right on the less than part, so that is gone, so we take it away there is nothing at all. Whereas, we had dependence from S1 to S2 on the second 1, the equal to part is trivially satisfied, so there is no need to worry about the equal to component from S2 to S1, because we do not reorder statements, we just keep the statement ordering as S1 to S2, automatically the equal to part is satisfied.

Now, there will be one dependency which is less than from S1 to S2 that is here, S1 to S2 with delta. We have shown I in red that means, it is run in sequential mode, so the only dependences are from this to this and from this to this. You see that there are no cycles in this particular graph anymore, because of the absence of cycles it is possible to vectorize these loops that is the basic reason why it can be vectorized.

(Refer Slide Time: 48:36)



Vectorization Example 2.4

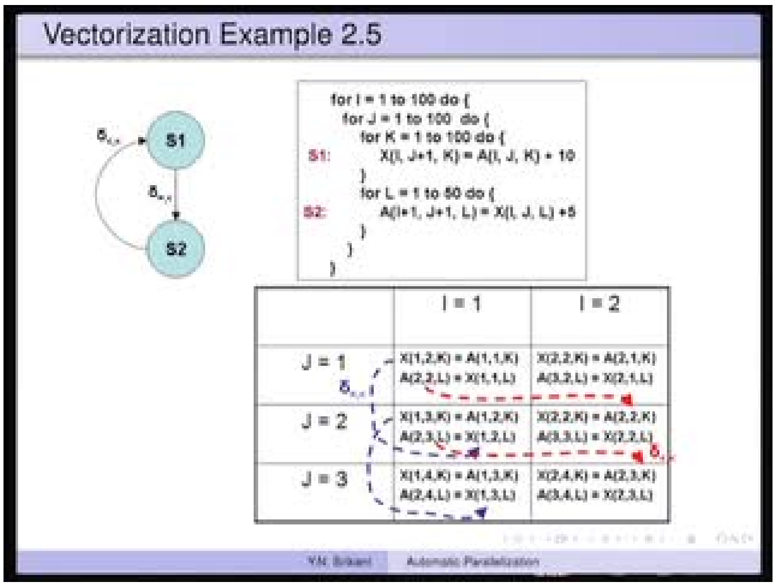
```
for I = 1 to 100 do {
  for J = 1 to 100  do {
    for K = 1 to 100 do {
S1:     X(I, J+1, K) = A(I, J, K) + 10
    }
    for L = 1 to 50 do {
S2:     A(I+1, J+1, L) = X(I, J, L) +5
    }
  }
}
```

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

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

Y.N. Srikant   Automatic Parallelization

(Refer Slide Time: 49:36)



Vectorization Example 2.5

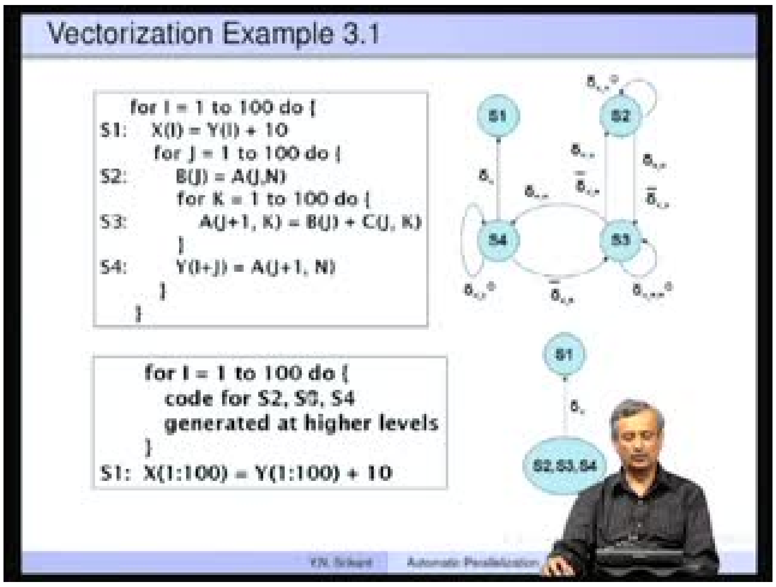
```
    for I = 1 to 100 do {
      for J = 1 to 100  do {
        for K = 1 to 100 do {
S1:        X(I, J+1, K) = A(I, J, K) + 10
        }
        for L = 1 to 50 do {
S2:        A(I+1, J+1, L) = X(I, J, L) +5
        }
      }
    }
```

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

Suppose the program is changed even little bit, the previous program had I J plus 1 K and I plus 1 J L, whereas here we have I plus 1 J plus 1 L, so that is the only change that we made, the dependences change. The previous one had dependence from S2 to S1, S1 to S2 and S1 to S2, here we have S2 to S1, S1 to S2 and S1 to S2 again, but the direction vector components have changed. Please observe that the previous direction vectors where less than or equal to and equal to less than, whereas here it is less than less than and equal to less than, so these are two different once.

Even now, I loop can run in sequential mode and the J loop can be vectorized, so this is the vector code that will be produced. Reasons for this will become a little clearer a little later. So here are the dependences for this particular changed program, so the dependences are from here to here and here to here, here to here and here to here that is why the dependences have changed to delta less than less than, so this part and delta equal to less than remains as it is, so this is the changed part.

(Refer Slide Time: 50:00)



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)
      }
    }
```

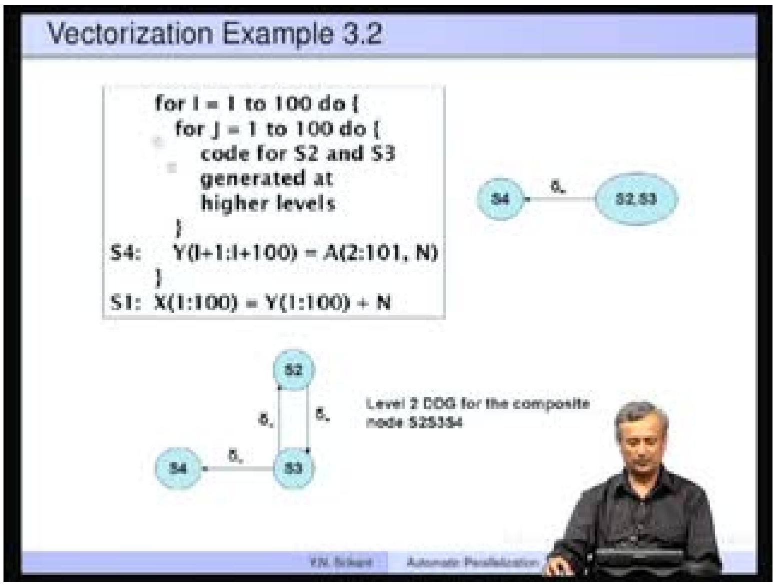
```
      for I = 1 to 100 do {
        code for S2, S3, S4
        generated at higher levels
      }
S1:  X(1:100) = Y(1:100) + 10
```

Then we take up another example which shows what happens if we actually have to generate code computing the strongly connected components. So here is a program S1, S2, S3, S4, there is an I loop in which there is a statement S1, then there is a J loop in which there is a statement S2, then there is a statement S3 which is inside the K loop and the finally S4 is one more which is inside the J loop. So the K loop has only S3, whereas J loop has S2 and S4 along with this K loop.

Dependences are complex so we will understand these dependences later, but please observe that there are number of cycles here. See S4 to S1 is a dependence, here is S4, this is S1, so Y of I plus J to Y I is the dependence from S4 to S1. There is a dependence on S4 itself that is because of this particular loop I, which is enclosing this, it is independent. This is Y of I plus J equal to A J plus 1 comma N, so when the I loop changes there is going to be a dependence from S4 to S4 itself.

Then we have dependence from S4 to S3 and S3 to S4, finally dependence from S3 to S2 and S2 to S3. One more, there are again dependences from S3 to S3 and S3 to S3; this is a very complex data dependence graph. For the present we are only interested in the loop structure of these data dependence graphs.

(Refer Slide Time: 52:50)



Vectorization Example 3.2

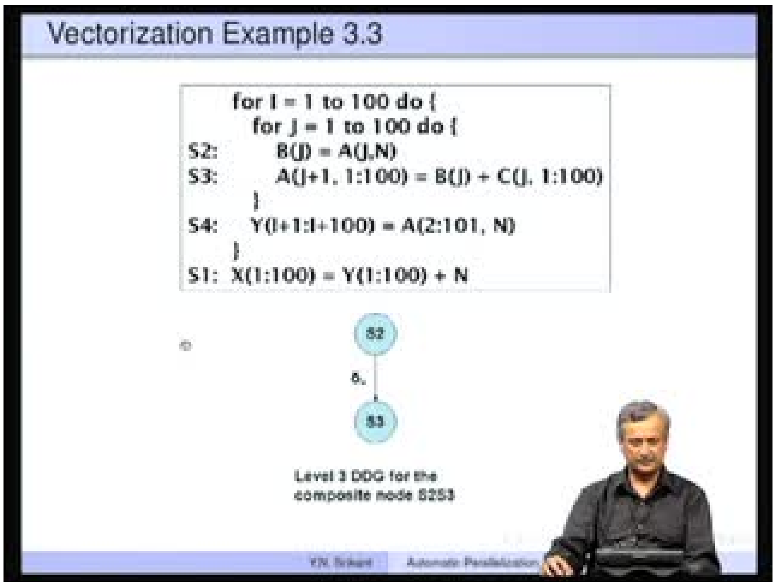
```
        for I = 1 to 100 do {
          for J = 1 to 100 do {
             code for S2 and S3
             generated at
             higher levels
          }
S4:       Y(I+1:I+100) = A(2:101, N)
        }
S1:  X(1:100) = Y(1:100) + N
```

Level 2 DDG for the composite node S2S3S4

Because of the loop structure at I level cannot really vectorize this code directly, let us assume that we run the I loop in sequential mode. There are strongly connected components, here 1 is here and another 1 is here, so both these together actually come into the same; there are 2 loops, so both these actually come into the same strongly connected component, S2, S3, S4 together forms an S C C because of the 2 loops here which are connected. So, S1 is not in the S C C, so we have S1 and then the S C C, S2, S3 S4 together, so we can generate vector code for S3, but S2, S3, S4 run with in the sequential I loop. Now what happens? Actually the level 2 diagram becomes like this, we had this and we are now expanding this ok.

So there is a loop from S2 to S3 but there is no other loop in the system, the outer loop has been removed, this loop has been removed, the I loop has been removed, only this particular loop at the lower level at the J level remains. So here with this loop this is a strongly connected components, so S2 and S3 are grouped together, S4 remains as it is. We can vectorize S4 within J for the J loop, but S2 and S3 have to be embedded within J. J has to run sequentially and S2 S3 are inside. So I run sequentially, J run sequentially, S2 and S3 are inside, S4 is vectorized for the J loop and S3 is vectorized for the I loop.

(Refer Slide Time: 53:50)



Vectorization Example 3.3

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

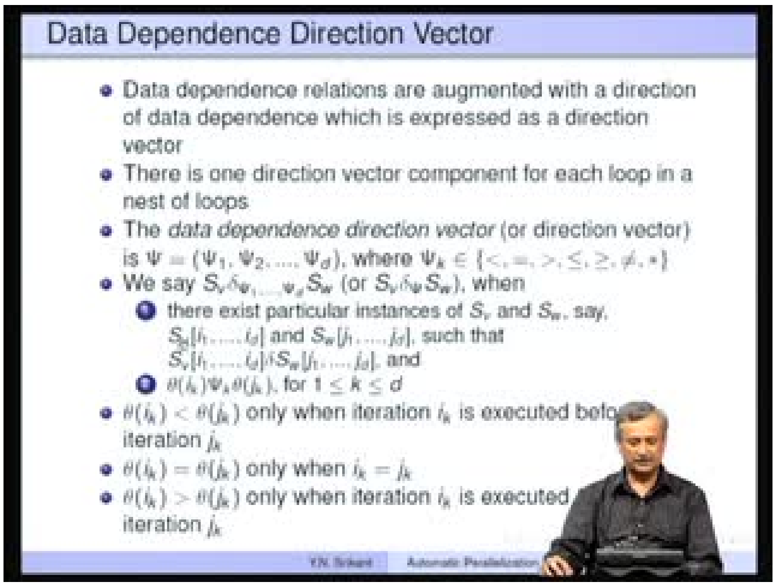
Level 3 DDG for the composite node S2S3

Then we have this particular data dependence graph level 3, S3 to S2 with single dependence, so this can be vectorized at the K level. What we really have done is vectorize S3 at the K level and S4 has already been vectorized. So please see that S4 has already been vectorized, S1 has also been vectorized, only S2 and S3 remain.

So we cannot do much for S2, it remains within J loop and it is not vectorisable as it is because it has nothing more to vectorize. Whereas, S3 is embedded within, look at the original program, it is embedded within the K loop which can be vectorized. So we have vectorized the K loop and here this is the vector code for the K loop. This is the final code that has been generated for this particular cyclic data dependence graph.

Hierarchically, we constructed the S C Cs then grouped the statements appropriately, generated code for the statements outside the group - the S C C and finally considered the S C C one by one. Then again we looked at the loops and strongly connected components within this data dependence graph and we again isolated the loop statements which are outside the strongly connected component generated code for it, this goes on. Finally, you may end up generating only sequential code, there is a loop even at the lowest level, but hopefully this does not happen in all data dependence graphs.

(Refer Slide Time: 55:41)



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 $\Psi = (\Psi_1, \Psi_2, \ldots, \Psi_d)$, where $\Psi_k \in \{<, =, >, \leq, \geq, \neq, *\}$
- We say $S_v \delta_{\Psi_1, \ldots, \Psi_d} S_w$ (or $S_v \delta_\Psi S_w$), when
  1. there exist particular instances of S_v and S_w, say, $S_v[i_1, \ldots, i_d]$ and $S_w[j_1, \ldots, j_d]$, such that $S_v[i_1, \ldots, i_d] \delta S_w[j_1, \ldots, j_d]$, and
  2. $\theta(i_k) \Psi_k \theta(j_k)$, for $1 \leq k \leq d$
- $\theta(i_k) < \theta(j_k)$ only when iteration i_k is executed befo
  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
  iteration j_k

Y.N. Srikant | Automatic Parallelization

So this is the end of today’s lecture and next time we will start discussion on data dependence direction vectors, thank you.