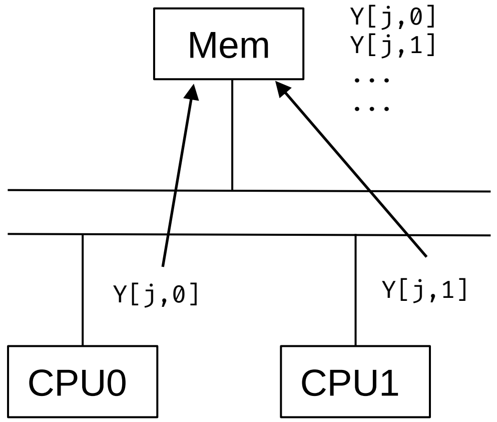
COL874: Advanced Compiler Techniques

Module 161-165 (Pipelining and Blocking)
By: Harsh Yadav

Introduction to Pipelining (Mod161)

- It’s kind of a different parallelization scheme for affine loop nest.
  - Different from synchronization free or sync-based (barriers).
- Probably heard about 5-stage pipeline of instruction execution at hardware level in other courses.
  - Inst. Fetch - Decode - Execute - Memory - Writeback.
- We will use similar concept on the software side (at compiler level).
- Let’s see an example...

```
for (i = 0; i <= M; i++) {
    for(j = 0; j <= N; j++) {
        X[i] = X[i] + Y[j][i];
    }
}
```



- The outer loop is parallelizable but…
- If y is laid out in row-major form?
  - Each processor memory footprint is high -> Cache misses.
  - Accesses a different cache line on each inner loop iteration.
  - In real case, L1 cache not used.
- Cache miss could be very expensive then parallelization:
  - All cache misses from different processors will go to memory -> increasing load on memory.
  - Depends on machine.

```
for (i = 0; i <= M; i++) {
   for(j = 0; j <= N; j++) {
      X[i] = X[i] + Y[j][i];
   }
}
```

Benefits

Better Cache Usage for each CPU.

Spatial locality is good.

Is it possible that…

CPU$_j$ accesses Y[j,0], Y[j,1], Y[j,2],.....

And possibly X[0], X[1], X[2],......

Problem?

Data Dependency Alert!!

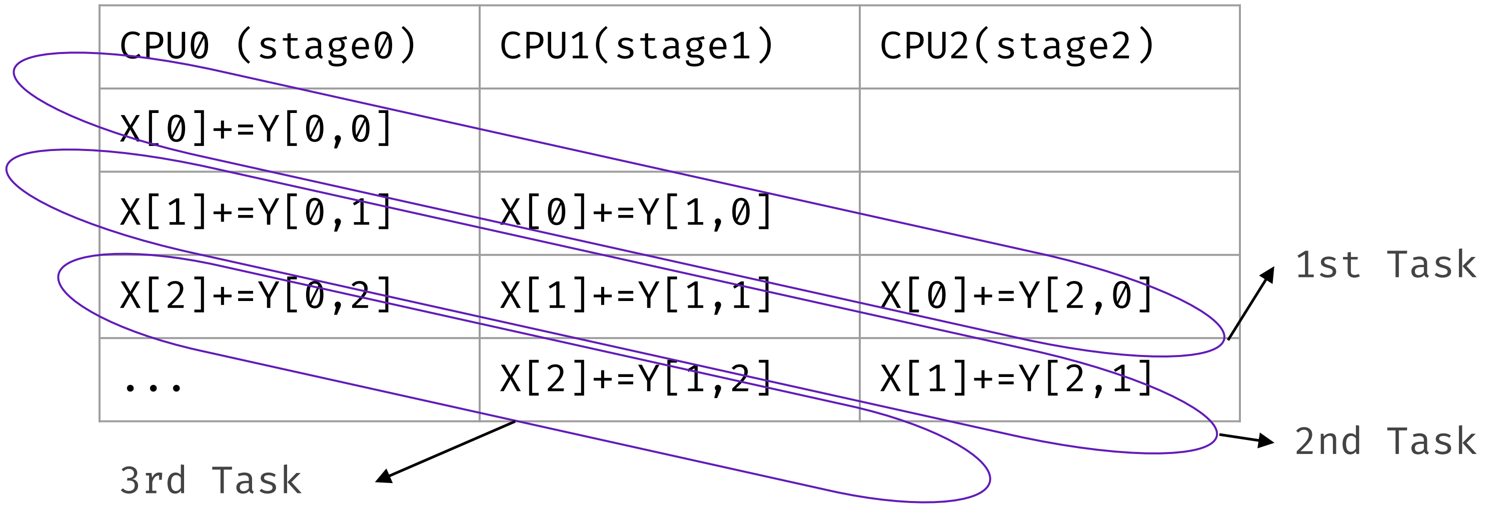
Each processor will write to X[0], X[1],...

But after CPU0 written to X[0] it doesn't care. So we can pipeline the accesses.

```
for (i = 0; i <= M; i++) {
    for(j = 0; j <= N; j++) {
        X[i] = X[i] + Y[j][i];
    }
}
```

O(n) Synchronisation
(Somewhat Loose,
everything doesn’t have to
happen in lock step.)

One Task | N stages
(Each Column is a Stage).



CPU0 (stage0)	CPU1(stage1)	CPU2(stage2)
X[0]+=Y[0,0]		
X[1]+=Y[0,1]	X[0]+=Y[1,0]	
X[2]+=Y[0,2]	X[1]+=Y[1,1]	X[0]+=Y[2,0]
...	X[2]+=Y[1,2]	X[1]+=Y[2,1]

```
for (i = 0; i <= M; i++) {
```

Fetch - Decode - Execute - Mem - WB

```
}
```

It’s like each CPU is specialised for one particular job.

Pipelining requires a loop depth of >= 2.

- Then iterations of *outer loop* can be counted as *tasks.*
- Iteration of the *inner loop* are counted as *stages.*
- Each processor will be specialized for the particular stage.

SOR Example(Mod162)

```
for (i = 0; i <= M; i++) {
    for(j = 0; j <= N; j++) {
        X[j+1] = (X[j] + X[j+1] + X[j+2])/3;
    }
}
```

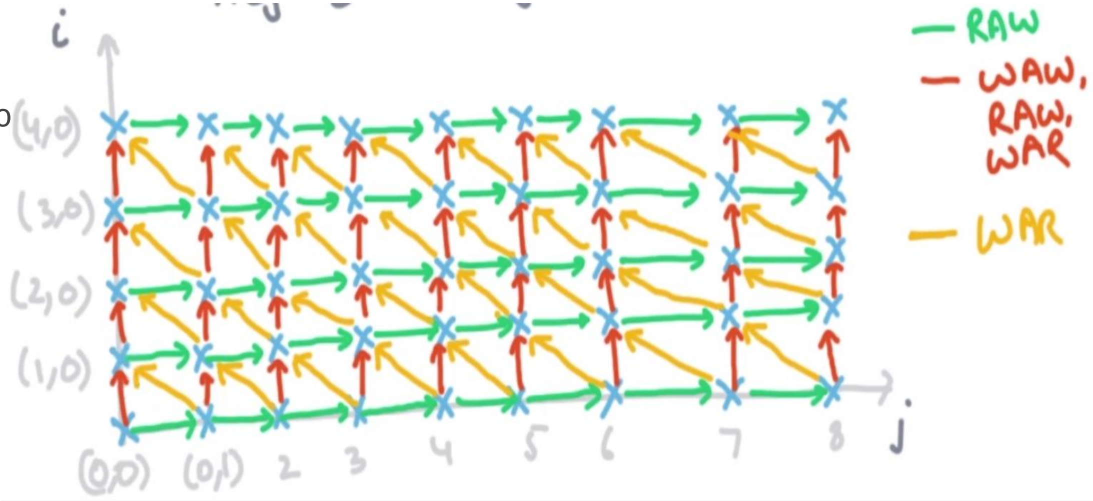
SOR (Successive Over Relaxation)

- It's kind of relaxing the value using the neighbours.
- N ∝ Size of X.
- M ∝ Number of relaxations.

Data Dependencies...
Green one: at jth itr write to X[j+1]
at j+1th itr read to X[j+1]

Red one: Between outer iterations because they are writing and reading to same elements.

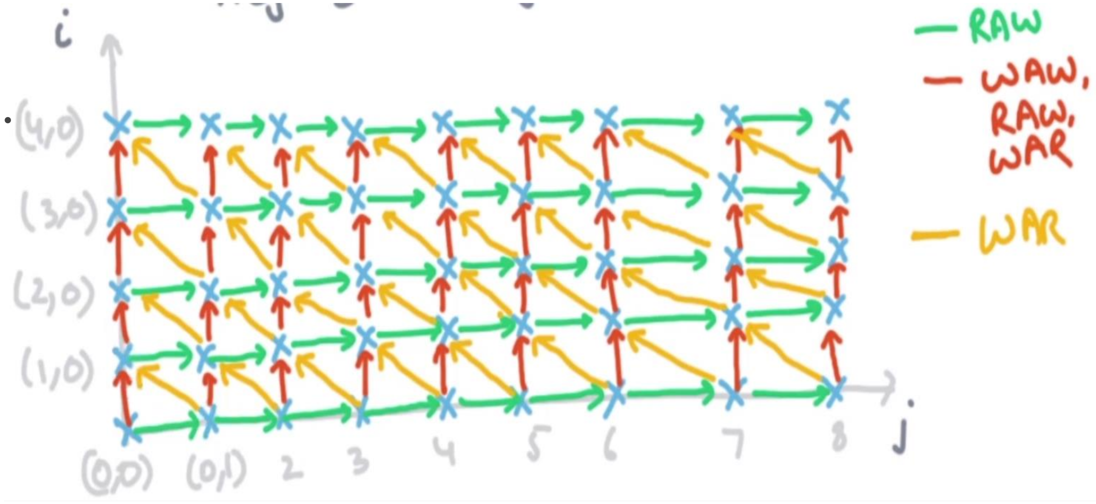
Yellow one: (0,1) read X[1].
(1,0) write X[1].
(0,1) < (1,0) so WAR



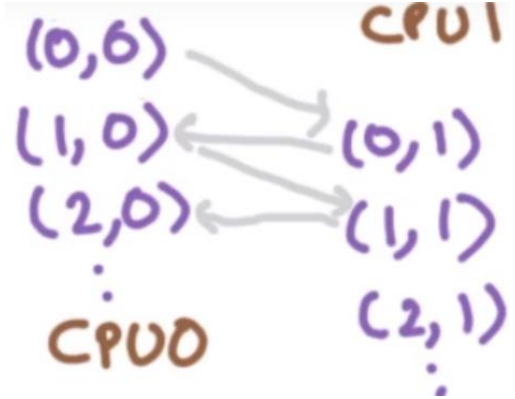
Sync-Free Parallelism?
Not possible. Whole graph is connected. Any two points in the graph are dependent.

```
for (i = 0; i <= M; i++) {
    for(j = 0; j <= N; j++) {
        X[j+1] = (X[j] + X[j+1] + X[j+2])/3;
    }
}
```

Is it Pipelinable?
- I want it to divide in tasks, task have stages and stages of first task have one way dependency to stages of second task and so on.
- Wavefront Phenomena.
- Not bidirectional dependencies

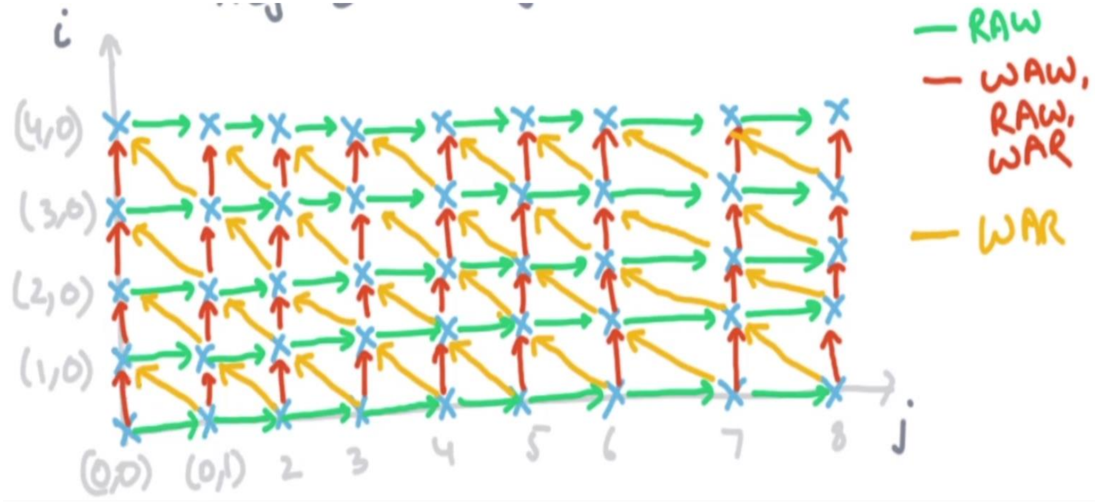


Lets try out the simple way



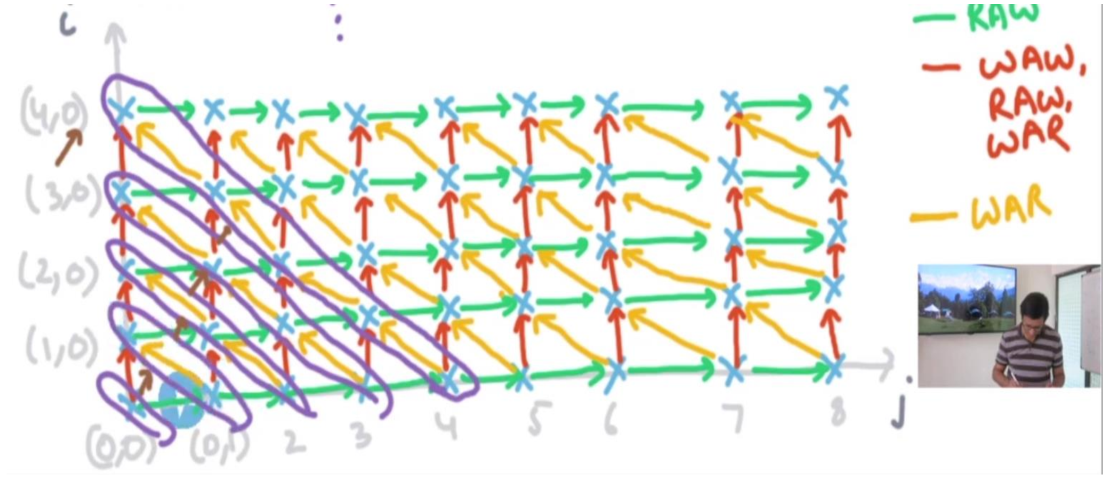
```
for (i = 0; i <= M; i++) {
    for(j = 0; j <= N; j++) {
        X[j+1] = (X[j] + X[j+1] + X[j+2])/3;
    }
}
```

Above will not work because there are dependency between stages.



```
for (i = 0; i <= M; i++) {
    for(j = 0; j <= N; j++) {
        X[j+1] = (X[j] + X[j+1] + X[j+2])/3;
    }
}
```

We can try doing it diagonally



Fully Permutable Loop(Mod163)

- How do distinguish between pipelinable and non-pipelinable loop?
- A loop is **fully permutable** if it can be permuted arbitrarily without changing the meaning of the program.

```
for (i = ...) {
   for(j = ...) {
     for(k = ...)
   }
}
```

⟷

```
for (j = ...) {
   for(i = ...) {
     for(k = ...)
   }
}
```

⟷

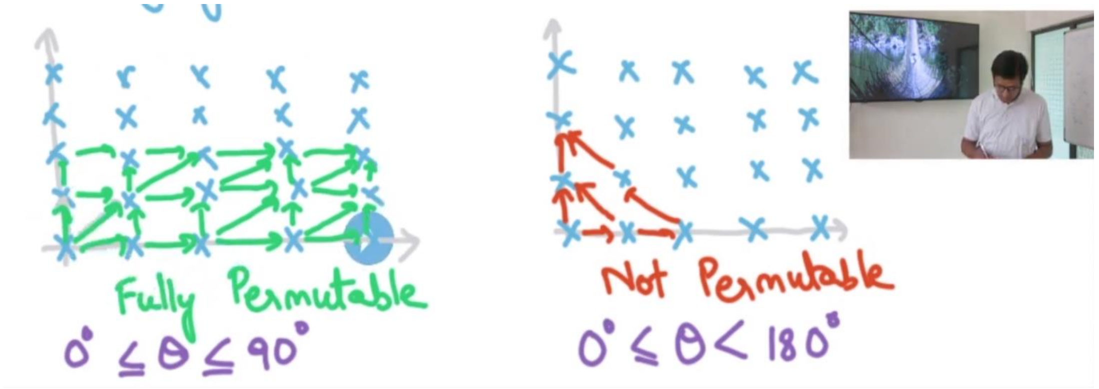
```
for (k = ...) {
   for(j = ...) {
     for(i = ...)
   }
}
```

⟷

$\forall\ \underline{i}_1, \underline{i}_2$

$(\underline{i}_1 < \underline{i}_2)$ && $(p(\underline{i}_1) > p(\underline{i}_2))$ => No data dependence between $\underline{i}_1$ and $\underline{i}_2$.

A loop is **fully permutable** if it can be permuted arbitrarily without changing the meaning of the program.



Fully Permutable
$0^\circ \le \theta \le 90^\circ$
Not Permutable
$0^\circ \le \theta < 180^\circ$

In fully permutable,
See (0,1) and (2,0) they have no data dependency. So the order of execution can be changed.

Data dependencies are from strictly lower (i,j) to strictly larger values of (i,j).
If all the edges are acute, then loop is permutable.

Change axis in SOR

```
for (i = 0; i <= M; i++) {
   for(j = 0; j <= N; j++) {
       X[j+1] = (X[j] + X[j+1]
                 + X[j+2])/3;
   }
}
```

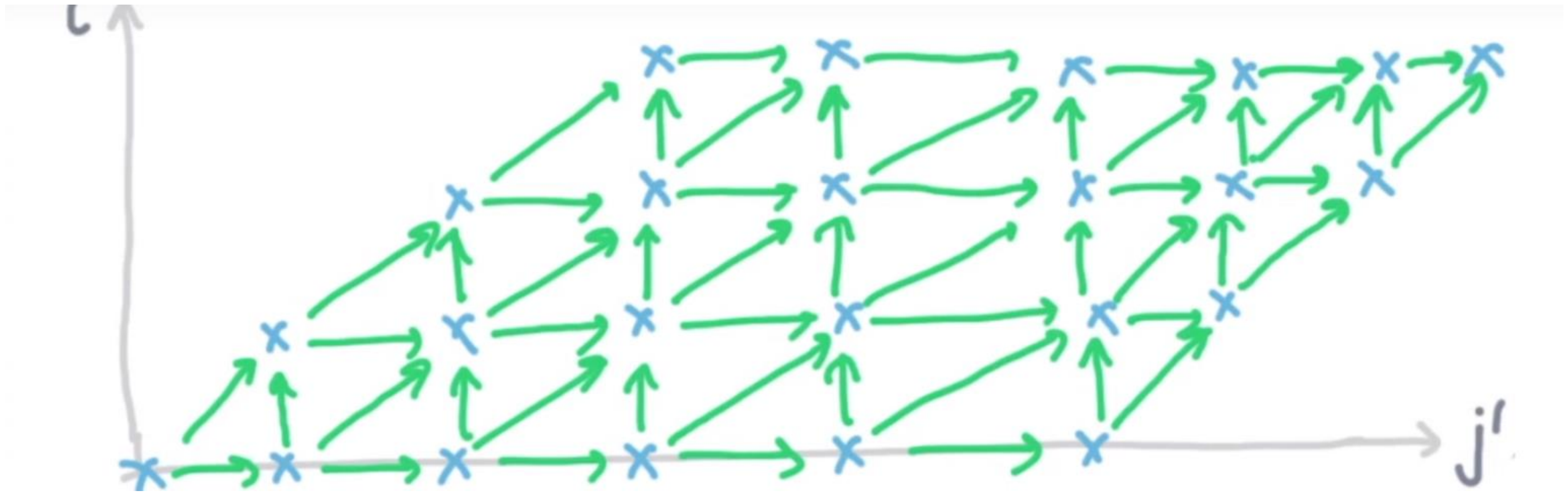
```
for (i = 0; i <= M; i++) {
   for (jd = i; jd <= i+N; jd++) {
       X[jd-i+1] = (X[jd-i] + X[jd-i+1]
                   + X[jd-i+2])/3;
   }
}
```

$$\begin{bmatrix} i \\ jd \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

Use fourier motzkin method to find the bounds

Now you can see all the dependencies are now in the acute angle.

```
for (i = 0; i <= M; i++) {
    for (jd = i; jd <= i+N; jd++) {
        X[jd-i+1] = (X[jd-i] + X[jd-i+1]
                             + X[jd-i+2])/3;
    }
}
```

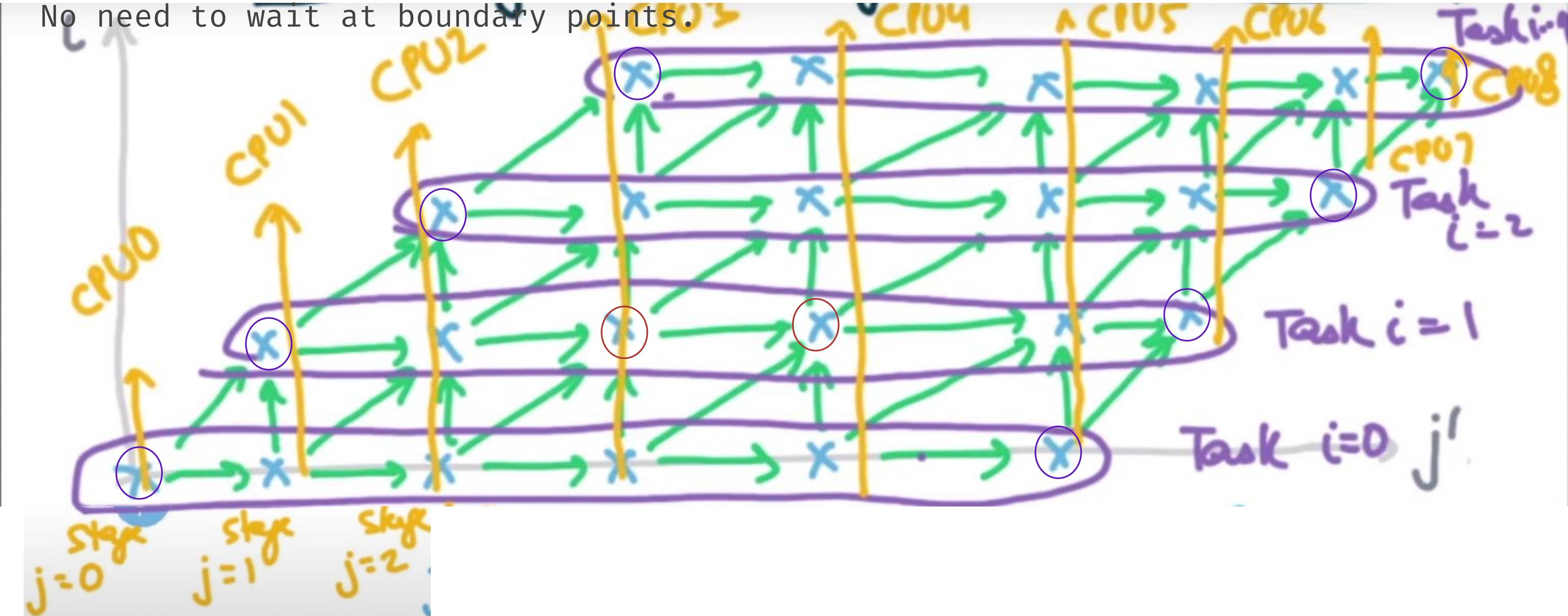


For each task i:
CPU j waits for CPU j-1 to signal before exec

O(N) synchronization. N tasks.

No need to wait at boundary points.

```
for (i = 0; i <= M; i++) {
    for (jd = i; jd <= i+N; jd++) {
        X[jd-i+1] = (X[jd-i] + X[jd-i+1]
                    + X[jd-i+2])/3;
    }
}
```



Pipeline Code Generation (Mod164)

- A loop with k outermost fully permutable loops can be structured as a pipeline with O(k-1) dimensions with O(n) synchronisation.
  - n : number of iterations of the inner loop.
  - Ex: SOR example k=2.

```
for (i = 0; i <= M; i++) {
   for(j = 0; j <= N; j++) {
       X[j+1] = (X[j] + X[j+1] +
X[j+2])/3;
   }
}
```

Pipelining Fully Permutable Loop

Ignoring boundary condition, a processor p can execute stage ith of a task only after processor p-1 executed i-1th stage of that task.

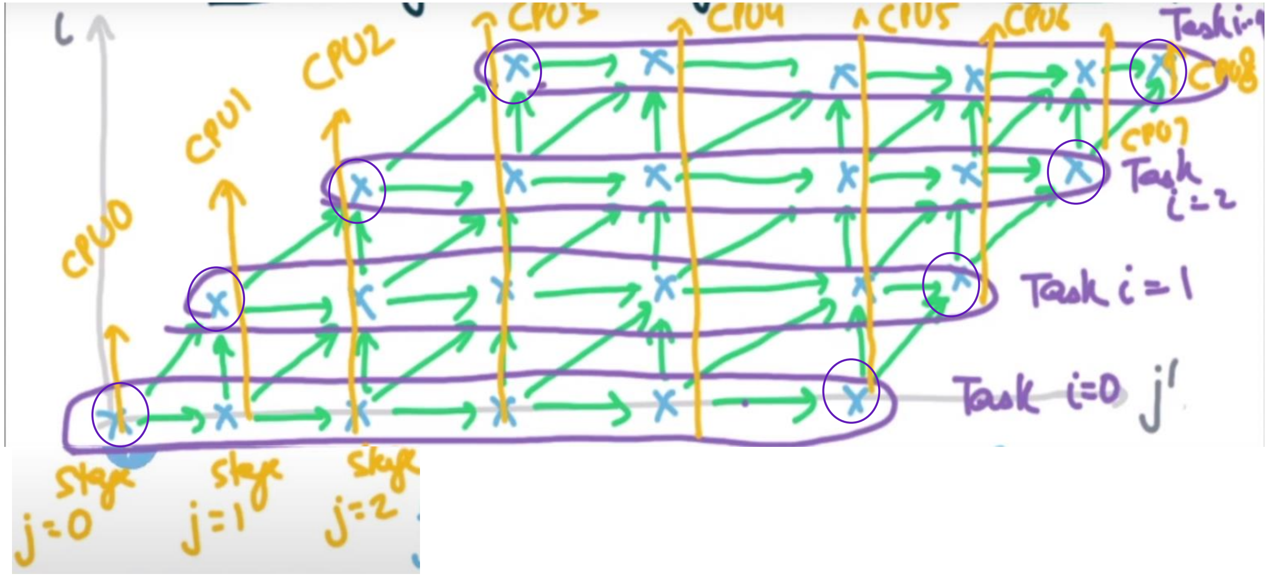
To do this we use **Condition Variables:**
On every iteration, processor p executes wait(p-1) *before the body* and signal(p+1) *after the body*.

Total Number of CPUs = M+N+1

Each CPU$_j$ is specialize for stage j. So lets generate the code for the CPU$_j$

CPU j ∈ [0, M+N]
i >= 0 and i >= j-N
i <= M and i <= j

```
for (i = 0; i <= M; i++) {
   for (jd = i; jd <= i+N; jd++) {
       X[jd-i+1] = (X[jd-i] + X[jd-i+1]
                   + X[jd-i+2])/3;
   }
}
```



Total Number of CPUs = M+N+1

Each CPU$_j$ is specialize for stage j. So lets generate the code for the CPU$_j$

CPU j ∈ [0, M+N]
i >= 0 and i >= j-N
i <= M and i <= j

```
for (i = 0; i <= M; i++) {
  for (jd = i; jd <= i+N; jd++) {
     X[jd-i+1] = (X[jd-i] + X[jd-i+1]
                   + X[jd-i+2])/3;
  }
}
```

```
for (i = max(0, j-N); i <= min(j, M); i++) {
   if(j > i) wait(j-1);     // If node is left boundary node or not.
   X[j-i+1] = (X[j-i] + X[j-i+1] + X[j-i+2])/3;
   if(j < i+N) signal(j+1);// If node is right boundary node or not.
}
```

Pipelining Vs Barrier

- Barrier has a *lock-step semantics.*
  - Faster Cpus need to wait more for slower threads.
- Pipeline has greater level of asynchronous behavior.
  - Wait/signal
  - Relaxed wavefront
  - Eg. Keep faster cpus can be called first for the execution and slower cpus can be called later.
  - One cpu can be slower but the other cpus may not have to wait for it.

Blocking (Mod165)

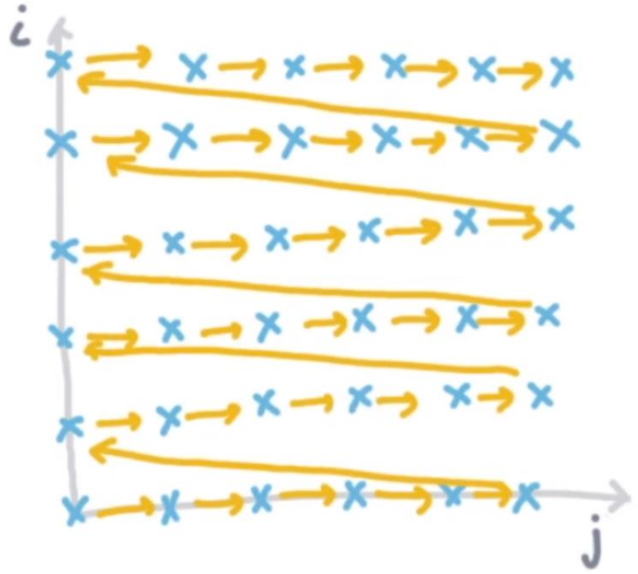
```
for (i = 0; i <= n; i++)
   for (j = 0;  j <= n; j++)
            S;
```

$\longrightarrow$

```
for (ii = 0; ii <= n; i+=b)
   for (jj = 0;  jj <= n; j+=b)
      for (i = ii*b; i <= min(n, (ii+1)*b); i++)
         for (j = jj*b;  j <= min(n, (jj+1)*b); j++)
            s;
```

-> Assume no Data Dependencies here.

-> Yellow lines are execution



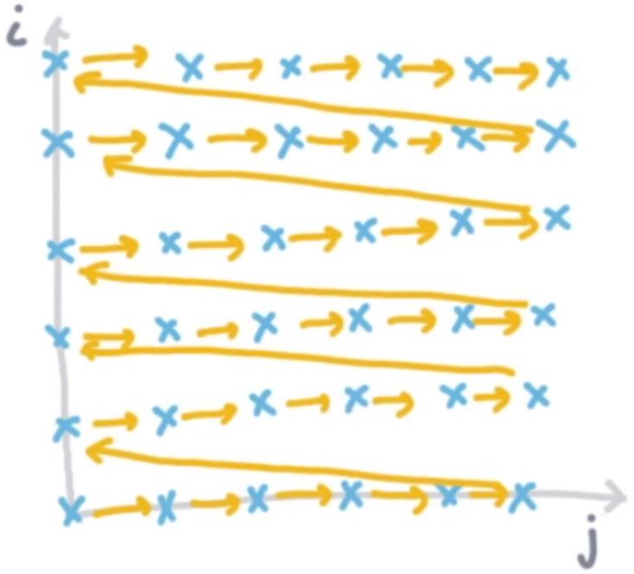
- Blocking is changing the execution order.
- When is it okay to do this? Why its good?
- Why part already discussed. *Matrix Mul.*
- Choosing b for greater cache locality. Gives both the spatial and temporal locality.
- How compiler can do it?
- **What are the conditions on s such that we can do blocking?**

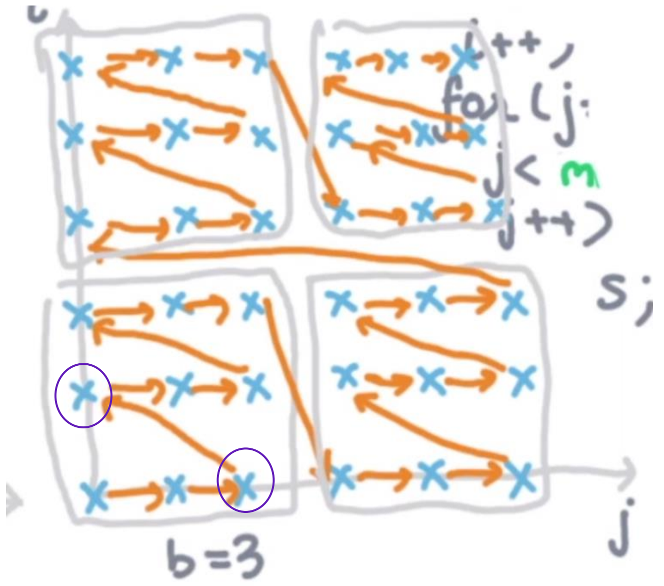
```
for (i = 0; i <= n; i++)
   for (j = 0;  j <= n; j++)
            S;
```

```
for (ii = 0; ii <= n; i+=b)
   for (jj = 0;  jj <= n; j+=b)
      for (i = ii*b; i <= min(n, (ii+1)*b); i++)
         for (j = jj*b;  j <= min(n, (jj+1)*b); j++)
            s;
```

-> Assume no Data Dependencies here.

-> Yellow lines are execution



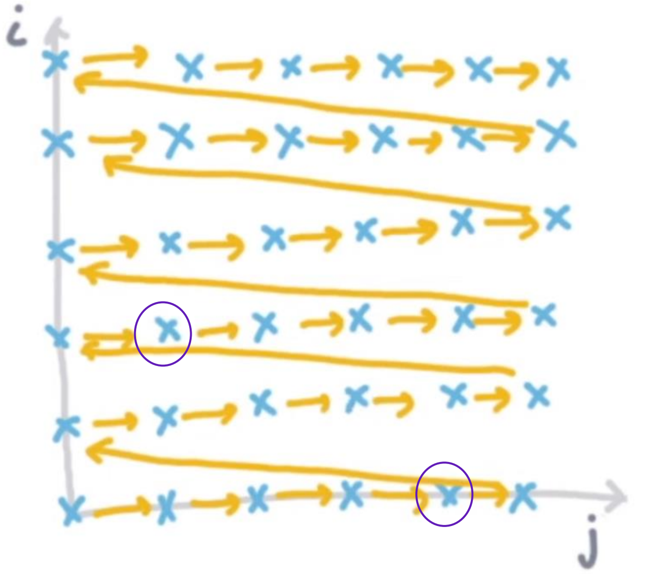


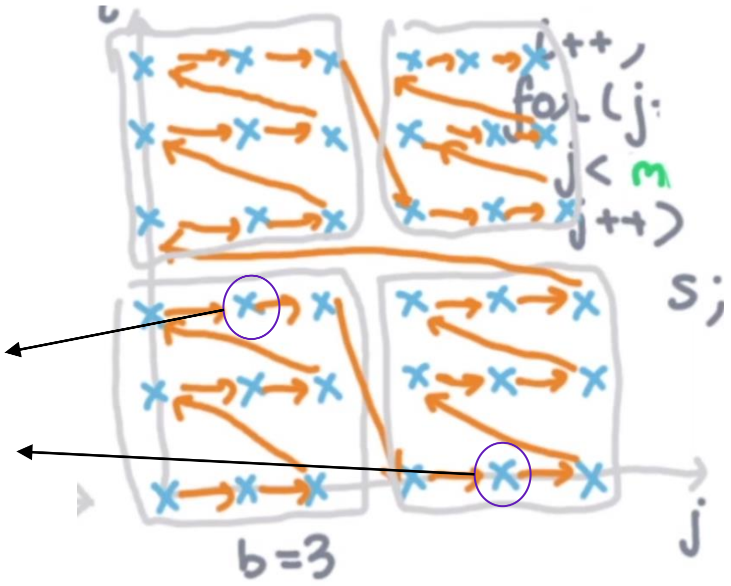
When is it possible to do blocking?
Lets see this

s: A[B[i,j]] = f(i,j)

We can’t do blocking here because there is a higher chance that there are dependencies (let’s say (2,1), (0,4)) and reordering them will give different results.

```
for (ii = 0; ii <= n; i+=b)
    for (jj = 0;  jj <= n; j+=b)
        for (i = ii*b; i <= min(n, (ii+1)*b); i++)
            for (j = jj*b;  j <= min(n, (jj+1)*b); j++)
                s;
```





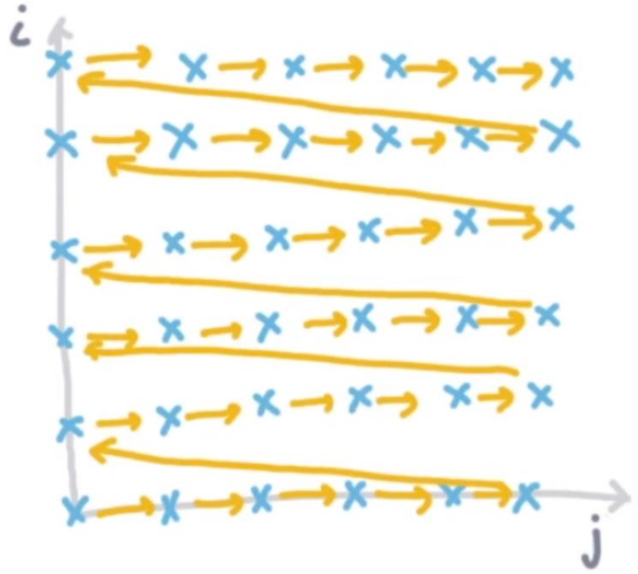
Relative order changed

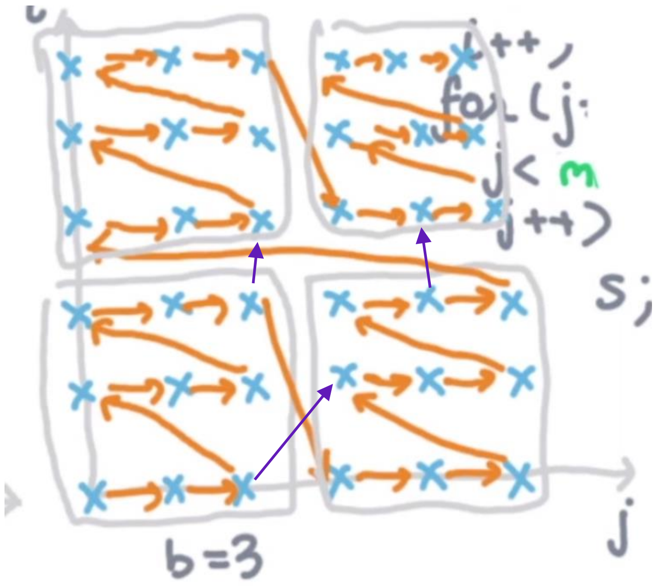
When is it possible to do blocking?
If order has changed, then there should not be data dependence between them.

Relative order remain same for 0 <= θ <= 90.

So all data dependencies should be at acute angle only.

Ex: check exec order in figure.



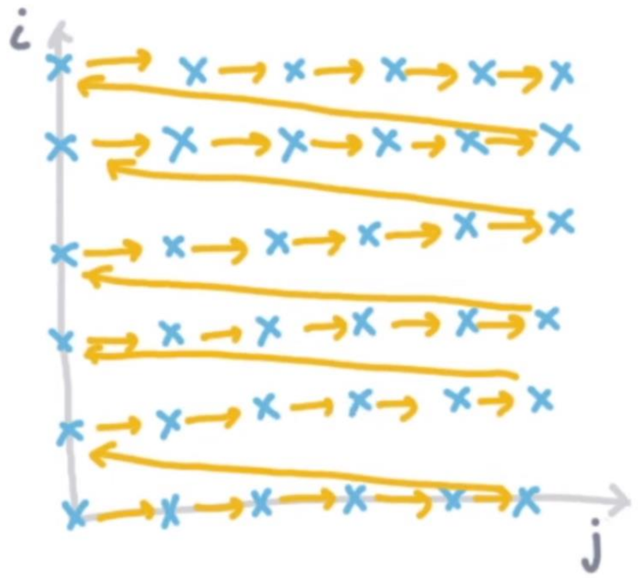


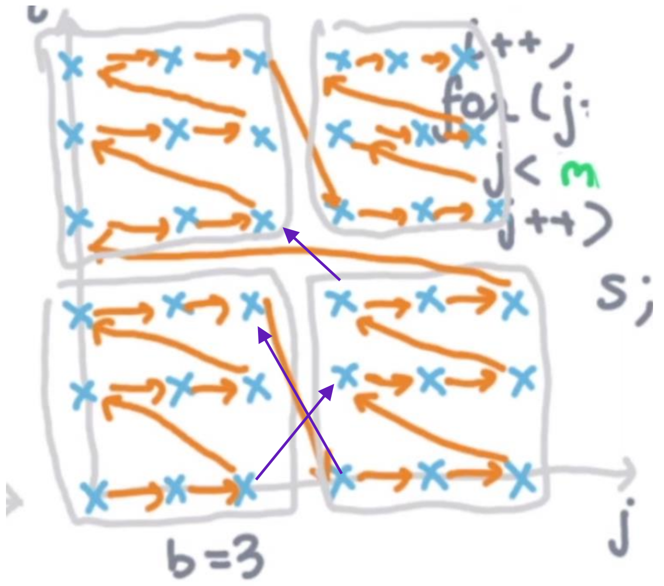
When is it possible to do blocking?
If order has changed, then there should not be data dependence between them.

Sufficient condition: whenever θ > 90 between two itr points, then there is no data dependency.

Relative order may change for 90 < θ < 180.

Relative order change ⇒ θ > 90.
Whenever θ > 90 ⇏ Relative order change.





It is always possible to block fully permutable loops.

- They share same condition.
- In matrix multiplication example we did 3-level blocking.
    - Data dependence coming from C.
        - If $(i_1 == i_2)$ && $(j_1 == j_2)$
    - Fully permutable? Yes. Check every order you will see. Commutative property of addition.

```
for (i = 0; i <= n; i++) {
    for (j = 0;  j <= n; j++) {
        for (k = 0;  j <= n; j++) {
            C[i,j] += A[i,k] * B[k,j];
        }
    }
}
```

Thank You

Harsh Yadav

(These slides are the summary of Module 161-165 of Compiler Design. For more details check Youtube Playlist on compilerai channel).