The polyhedral model

Dillon Huff

Can we reverse this loop?

```
for i in [1, 4]:
  S: A[i] = A[i - 1]
```

Do these loops have the same behavior?

```
for i in [1, 4]:
  S: A[i] = A[i - 1]
```

```
for i in [4, 1]:
  S: A[i] = A[i - 1]
```

Lets look at the program traces

```
for i in [1, 4]:
  S: A[i] = A[i - 1]
```

S[1]
S[2]
S[3]
S[4]

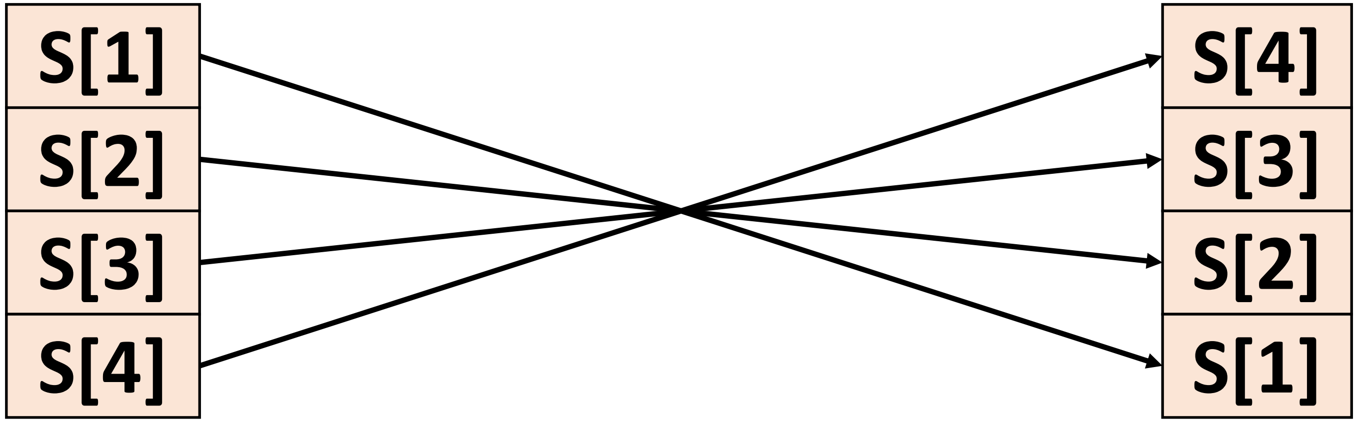
```
for i in [4, 1]:
  S: A[i] = A[i - 1]
```

S[4]
S[3]
S[2]
S[1]

The sets of statements are the same in each one

for i in [1, 4]:
  S: A[i] = A[i - 1]

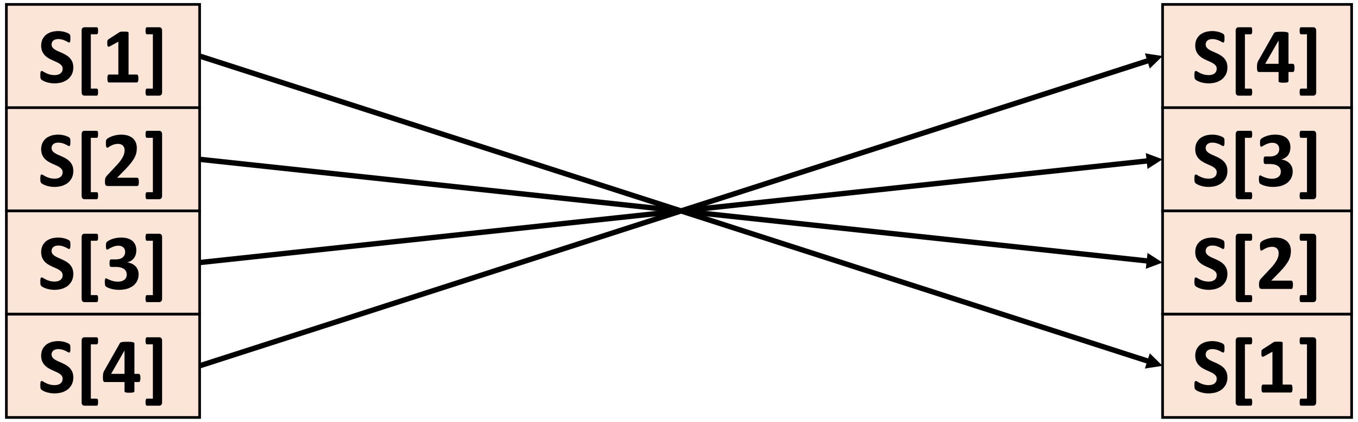
for i in [4, 1]:
  S: A[i] = A[i - 1]



Only the order changes

for i in [1, 4]:
  S: A[i] = A[i - 1]

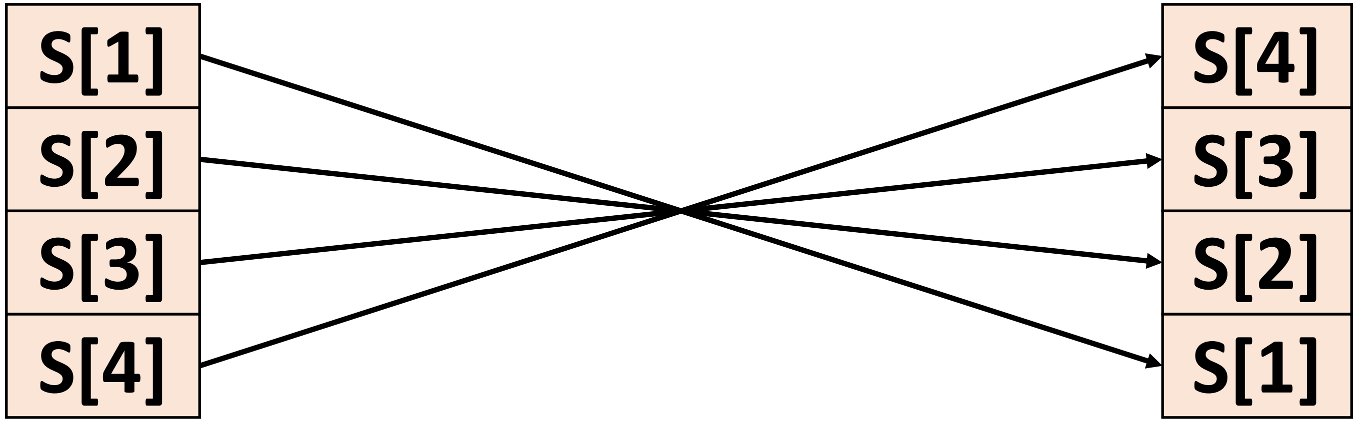
for i in [4, 1]:
  S: A[i] = A[i - 1]



So when we change the order, does anything go wrong?

```
for i in [1, 4]:
  S: A[i] = A[i - 1]
```

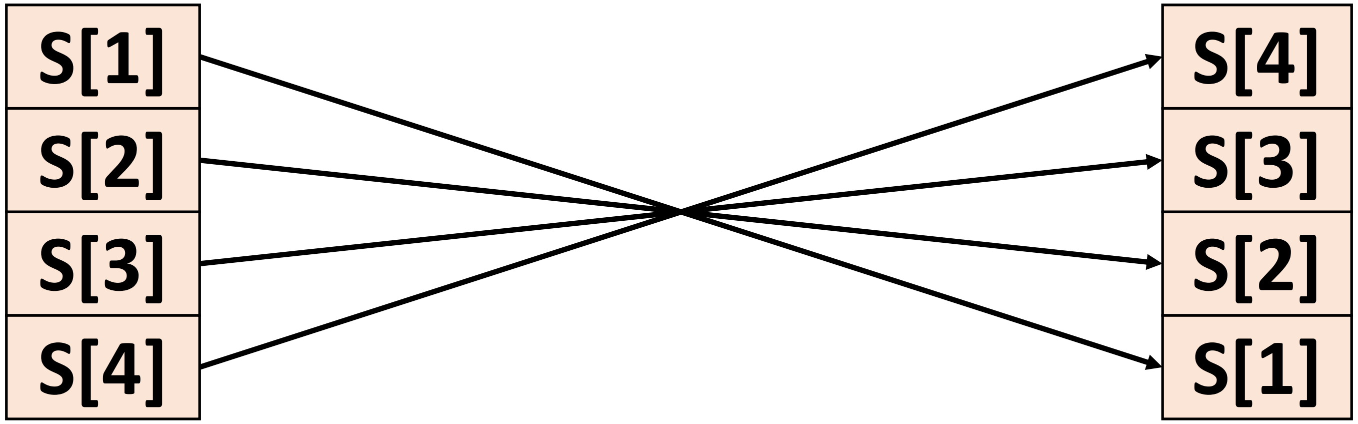
```
for i in [4, 1]:
  S: A[i] = A[i - 1]
```



So when we change the order, does the program's behavior change?

for i in [1, 4]:
  S: A[i] = A[i - 1]

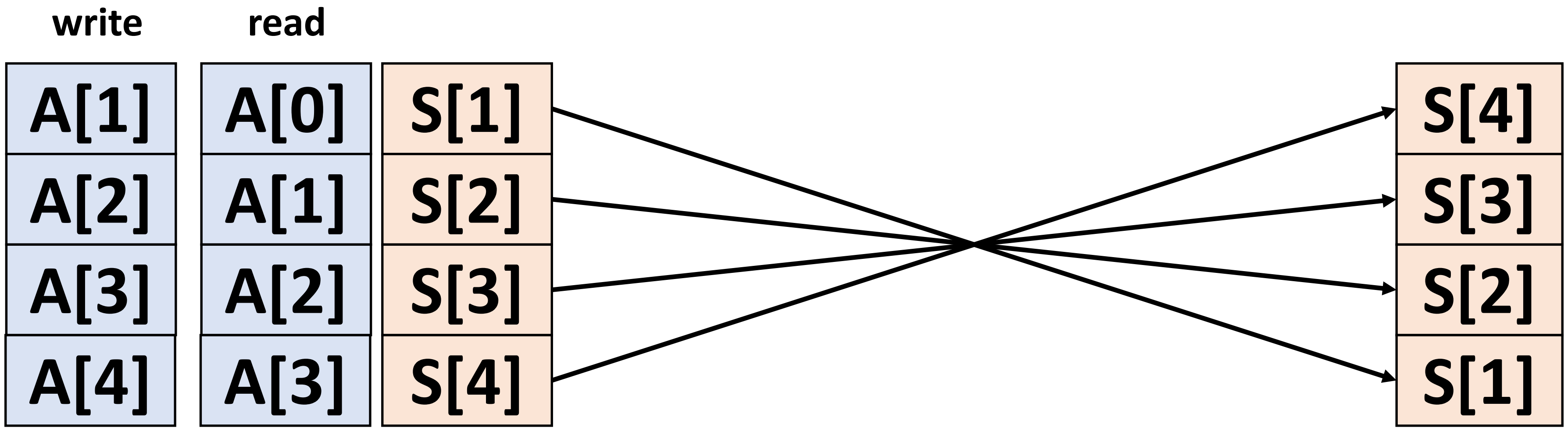
for i in [4, 1]:
  S: A[i] = A[i - 1]



So when we change the order, are any dependencies violated?

for i in [1, 4]:
  S: A[i] = A[i - 1]

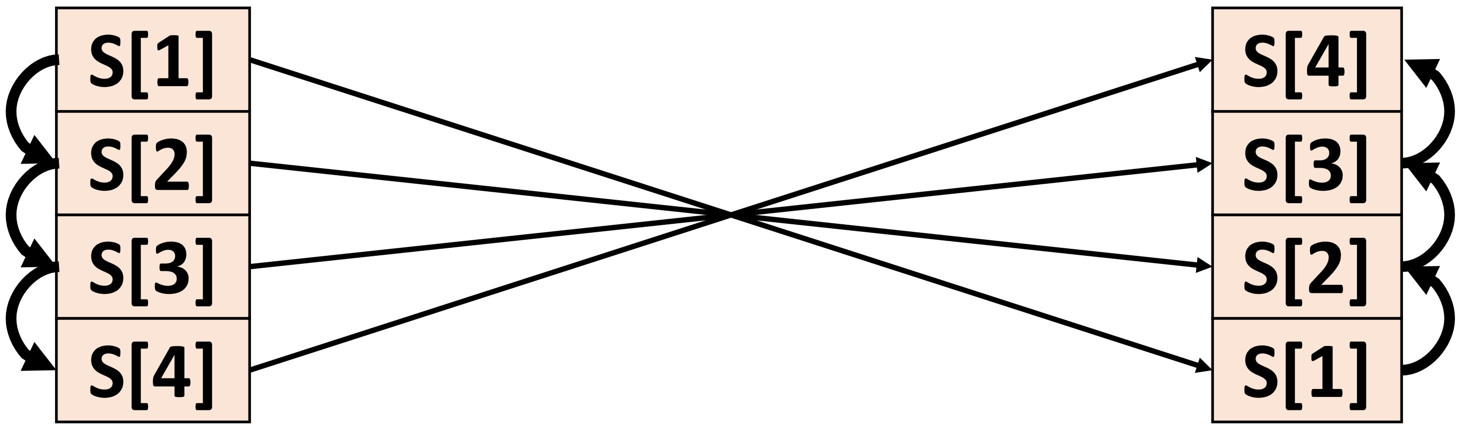
for i in [4, 1]:
  S: A[i] = A[i - 1]



Yes

for i in [1, 4]:
  S: A[i] = A[i - 1]

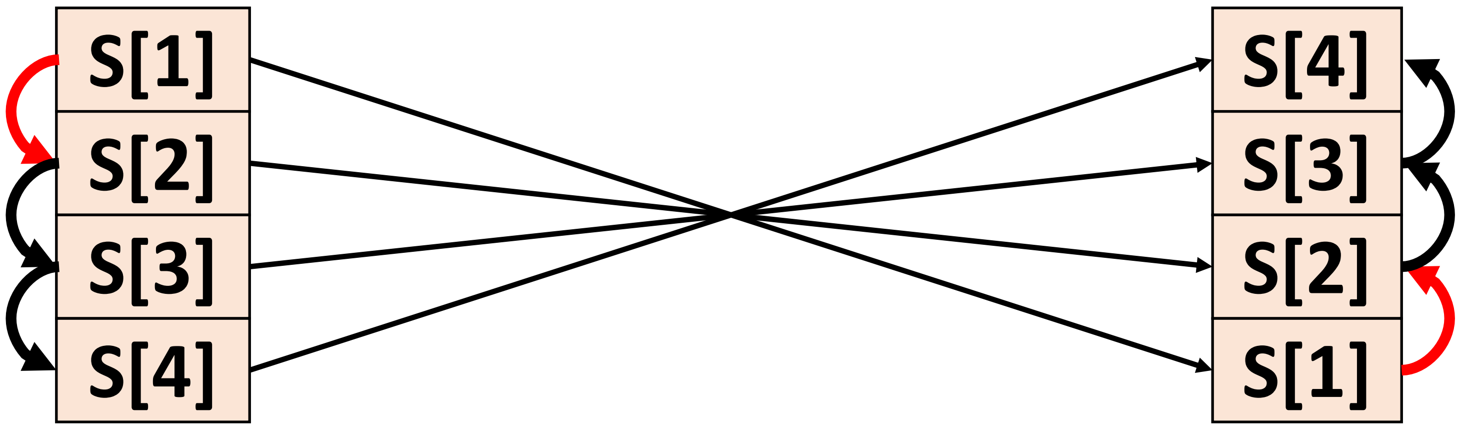
for i in [4, 1]:
  S: A[i] = A[i - 1]



For example

for i in [1, 4]:
  S: A[i] = A[i - 1]

for i in [4, 1]:
  S: A[i] = A[i - 1]



Now lets formalize this analysis a little more

We are given an original program

```
for i in [1, 4]:
  S: A[i] = A[i - 1]
```

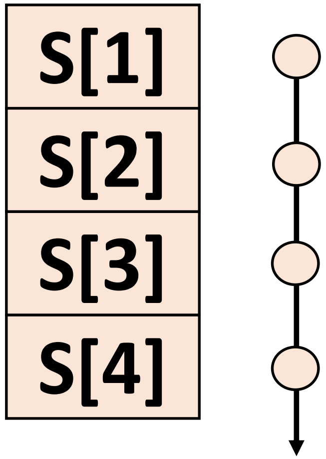
And a candidate target program

```
for i in [1, 4]:
  S: A[i] = A[i - 1]
```

```
for i in [4, 1]:
  S: A[i] = A[i - 1]
```

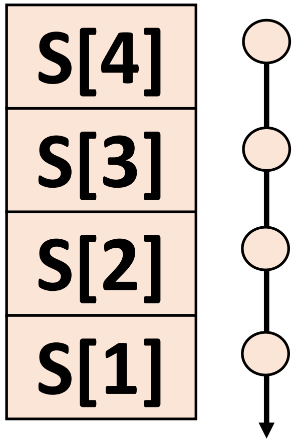
Both of them define execution traces that contain the same set of statements

for i in [1, 4]:
  S: A[i] = A[i - 1]



{ S[i] | 1 <= i <= 4 }

for i in [4, 1]:
  S: A[i] = A[i - 1]



{ S[i] | 1 <= i <= 4 }

The control logic defines the order in which statements are executed

for i in [1, 4]:

S: A[i] = A[i - 1]

S[1]
S[2]
S[3]
S[4]

{ S[i] | 1 <= i <= 4 }

{ S[i] -> i }

for i in [4, 1]:

S: A[i] = A[i - 1]

S[4]
S[3]
S[2]
S[1]

{ S[i] | 1 <= i <= 4 }

{ S[i] -> 5 - i }

The schedule and memory access pattern of the original program define a set of data dependencies

```
for i in [1, 4]:
  S: A[i] = A[i - 1]
```

S[1]
S[2]
S[3]
S[4]

{ S[i] | 1 <= i <= 4 }

{ S[i] -> i }

```
for i in [4, 1]:
  S: A[i] = A[i - 1]
```

S[4]
S[3]
S[2]
S[1]

{ S[i] | 1 <= i <= 4 }

{ S[i] -> 5 - i }

{ S[i] -> S[i + 1] | 1 <= i <= 3 }

Construct the set of all violated dependencies in the new schedule

The set of all violated dependencies is the intersection of:

The set of all pairs (a, b) where a sends data to b

The set of all pairs of (a, b) where a comes before b in the new schedule:

{ (S[i], S[i + 1]) | 1 <= i <= 3 && Sched(i) >= Sched(i + 1) }

{ (S[i], S[i + 1]) | 1 <= i <= 3 && 5 – i >= 5 – (i + 1) }

And then check if it is empty

{ (S[i], S[i + 1]) | 1 <= i <= 3 && Sched(i) >= Sched(i + 1)}

{ (S[i], S[i + 1]) | 1 <= i <= 3 && 5 – i >= 5 – (i + 1) }

This emptiness check can be done with integer linear programming

{ (S[i], S[i + 1]) | 1 <= i <= 3 && 5 – i >= 5 – (i + 1) } is empty

if and only if the system of linear inequalities:

1 <= i <= 3
5 – i >= 5 – (i + 1)

has no solution

This emptiness check can be done with integer linear programming

{ (S[i], S[i + 1]) | 1 <= i <= 3 && 5 – i >= 5 – (i + 1) } is empty

if and only if the system of linear inequalities:

1 <= 1 <= 3
5 – 1 >= 5 – (1 + 1)

has no solution

This emptiness check can be done with integer linear programming

{ (S[i], S[i + 1]) | 1 <= i <= 3 && 5 – i >= 5 – (i + 1) } is empty

if and only if the system of linear inequalities:

1 <= 1 <= 3
4 >= 5 – 2

has no solution

This emptiness check can be done with integer linear programming

{ (S[i], S[i + 1]) | 1 <= i <= 3 && 5 – i >= 5 – (i + 1) } is empty

if and only if the system of linear inequalities:

1 <= 1 <= 3

4 >= 3

Integer linear programming

Solves linear integer equations and inequalities and optimizes linear objective functions

$3*x + 4*y + 7 >= 0$

$-3*x - 3 <= 0$

$z + 2 + x = 0$

This is an ILP problem

find integers x, y, and z such that:

3*x + 4*y + 7 >= 0

-3*x – 3 <= 0

z + 2 + x = 0

This is **NOT** an ILP problem

find integers x, y, and z such that:

3*x + 4*y + 7 + **sin(x)** >= 0

-3*x – 3 <= 0

z + 2 + x = 0

This is an ILP problem

find integers x, y, and z that minimize: x + y + z

subject to:

3*x + 4*y + 7 >= 0

-3*x – 3 <= 0

z + 2 + x = 0

This is **NOT** an ILP problem

find integers x, y, and z that minimize: x + y + z

subject to:

3*x*y + 4*y + 7 >= 0

-3*x – 3 <= 0

z + 2 + x = 0

This is **NOT** an ILP problem

find integers x, y, and z that minimize: y + z

subject to:

x + 4*y + 7 >= 0

-3*x – 3 <= 0

z + 2 + x = 0

forall x >= 0. x + y <= 1

This is an ILP problem

find integers x, y, and z that minimize: y + z

subject to:

x + 4*y + 7 >= 0

-3*x – 3 <= 0

z + 2 + x = 0

x + y <= 1

Integer linear programming (ILP)

NP-complete (so it is very hard in theory)

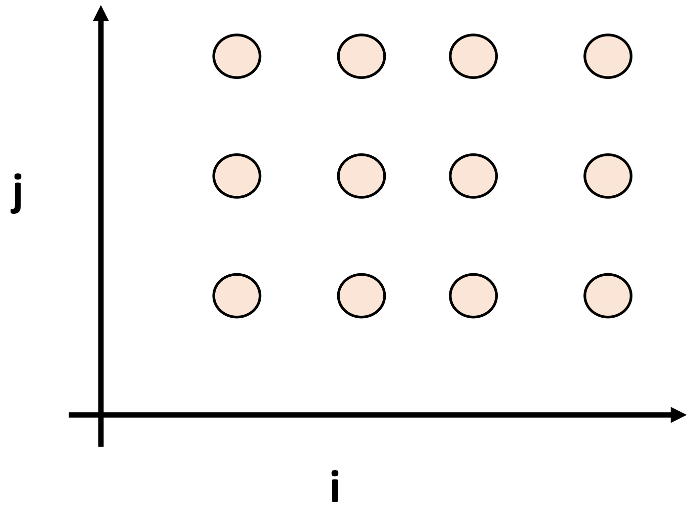
Often tractable in practice for problems with hundreds of variables

This includes more than you might think

- You can express propositional logic, division and remainder (by a constant), min, max, absolute value, comparisons, and many other things

What about multiple dimensions?

```
for i in [1, 4]:
  for j in [1, 3]:
    S: A[i][j] = A[i – 1][j + 1]
```



We need schedules with multiple dimensions

```
for i in [1, 4]:
  for j in [1, 3]:
    S: A[i][j] = A[i – 1][j + 1]
```

S[i, j] -> [i, j]

But how are these schedules ordered?

```
for i in [1, 4]:
  for j in [1, 3]:
    S: A[i][j] = A[i – 1][j + 1]
```

[i, j] > [i + 3 j - 1] ???

Lexicographically

```
for i in [1, 4]:
  for j in [1, 3]:
    S: A[i][j] = A[i – 1][j + 1]
```

[i, j] >> [i + 3 j - 1]

<->

(i > i + 3) v ((i = i + 3) ^ (j > j – 1))

<->

False

Lexicographic order is like the time on a clock

- [1, 0] >> [0, 9] for the same reason that 1 minute and zero seconds is a larger amount of time than 0 minutes and 9 seconds

- [a, b] >> [c, d] if and only if: a > c or (a = c and b > d)

- Requires more calls to an ILP solver to check emptiness

Checking if loop interchange is possible

```
for i in [1, 4]:
  for j in [1, 3]:
    S: A[i][j] = A[i – 1][j + 1]
```

S[i, j] -> [i, j]

```
for j in [1, 3]:
  for i in [1, 4]:
    S: A[i][j] = A[i – 1][j + 1]
```

S[i, j] -> [j, i]

Checking if loop interchange is possible

```
for i in [1, 4]:
  for j in [1, 3]:
    S: A[i][j] = A[i – 1][j + 1]
```

```
for j in [1, 3]:
  for i in [1, 4]:
    S: A[i][j] = A[i – 1][j + 1]
```

S[i, j] -> [i, j]

S[i, j] -> [j, i]

[i’, j’] << [j, i] &&
i' = 1 + i && j' = -1 + j
1 <= i <= 4 &&
1 <= j <= 3

Checking if loop interchange is possible

```
for i in [1, 4]:
  for j in [1, 3]:
    S: A[i][j] = A[i – 1][j + 1]
```

S[i, j] -> [i, j]

```
for j in [1, 3]:
  for i in [1, 4]:
    S: A[i][j] = A[i – 1][j + 1]
```

S[i, j] -> [j, i]

[i’, j’] << [j, i] &&
i' = 1 + i && j' = -1 + j
1 <= i <= 4 &&
1 <= j <= 3
This is SAT, so the transformation is illegal

Ok, so we can check if a program transformation is legal…

But what if we want to find a program transformation from scratch?

Two ways to use the polyhedral model

- **Analysis:** Check legality of a transform: extract initial schedule, and data dependencies, construct the final schedule you want, and then check if the final schedule breaks any dependencies

- **Scheduling:** Extract initial schedule, and data dependencies. Set up an objective function that captures what you want, and constraints that guarantee that all dependencies are satisfied. Then solve the resulting ILP

Optimize this program for locality

```
for i in [0, 5]:
  P: A[i] = input[i] + 1

for j in [0, 5]:
  C: B[j] = A[j] * 2
```

The optimization problem

optimize: some function that models how much locality there is in the new schedules

subject to:

a bunch of constraints on the new schedules that guarantee that the dependencies in the original program are respected

The new schedules will be affine functions of the original loop index variables…

```
for i in [0, 5]:
  P: A[i] = input[1] + 1

for j in [0, 5]:
  C: B[j] = A[j] * 2
```

SP(i) = sp*i + dp

SC(j) = sc*j + dc

Our optimization problem needs to pick values for the schedule parameters

```
for i in [0, 5]:
  P: A[i] = input[1] + 1

for j in [0, 5]:
  C: B[j] = A[j] * 2
```

SP(i) = **sp***i + **dp**

SC(j) = **sc***j + **dc**

The optimization problem

optimize: some function that captures the locality of the schedules

subject to:

constraints on sp, dp, sc, dc that guarantee that the dependencies in the original program are respected

The optimization problem

optimize: some function that captures the locality of the schedules

subject to:

forall i, j such that P(i) sends data to C(j). SP(i) <= SC(j)

The optimization problem

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. SP(i) <= SC(j)

The optimization problem

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. sp*i + dp <= sc*j + dc

But this is totally intractable!

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. sp*i + dp <= sc*j + dc

We have non-linear constraints…

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. sp*i + dp <= sc*j + dc

And they are universally quantified…

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. sp*i + dp <= sc*j + dc

We can resolve both of these problems with a theorem called the affine form of Farkas lemma

forall x in { x | Ax + b >= 0 } . s^Tx + d >= 0

<->

exists p_0, p >= 0. forall x. s^Tx + d = p^T(Ax + b) + p_0

How the @&#!& does that help?!

forall x in { x | Ax + b >= 0 } . s^Tx + d >= 0

<->

exists p_0, p >= 0. forall x. s^Tx + d = p^T(Ax + b) + p_0

Another way to say this…

forall x in { x | Ax + b >= 0 } . $s^T x + d >= 0$

<->

exists p_0, p >= 0. forall x. $s^T x + d = p^T(Ax + b) + p_0$

An affine form is non-negative over a polyhedron if and only if it can be written as a non-negative combination of the constraints that form the polyhedron

Lets look at a smaller example:

forall x >= 0 . a*x >= 0

A small example

forall x >= 0 . a*x >= 0

<-> **farkas lemma**

exists p0, p1 >= 0 . forall x . a * x = p1 * x + p0

A small example

forall x >= 0 . a*x >= 0

<-> **farkas lemma**

exists p0, p1 >= 0 . forall x . a * x = p1 * x + p0

<-> **isolate the universally quantified “x”**

exists p0, p1 >= 0 . forall x . (a – p1) * x - p0 = 0

A small example

forall x >= 0 . a*x >= 0

<-> **farkas lemma**

exists p0, p1 >= 0 . forall x . a * x = p1 * x + p0

<-> **isolate the universally quantified “x”**

exists p0, p1 >= 0 . forall x . (a – p1) * x - p0 = 0

<-> **simplify using standard linear algebra**

exists p0, p1 >= 0 . (a – p1) = 0 && p0 = 0

<->

SAT(p0 >= 0 && p1 >= 0 && a = p1 && p0 = 0)

Back to our more realistic example…

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. sp*i + dp <= sc*j + dc

Lets re-organize to isolate the quantified variables…

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. sp*i + dp – sc*j – dc <= 0

Lets re-organize to isolate the quantified variables…

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. -sp*i - dp + sc*j + dc >= 0

The inequality is actually an affine form (a dot product of 2 vectors plus a constant) with respect to i and j

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. -sp*i + sc*j + (dc –dp) >= 0

And the domain of the quantifier is a polyhedron

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. -sp*i + sc*j + (dc –dp) >= 0

And the domain of the quantifier is a polyhedron

optimize: some function that captures the locality of the schedules

subject to:

forall 0 <= i && i <= 5 && 0 <= j && j <= 5 && i <= j & j <= i.

-sp*i + sc*j + (dc –dp) >= 0

Lets normalize the domain constraints

optimize: some function that captures the locality of the schedules

subject to:

forall i >= 0 && -i >= -5 && j >= 0 && -j >= -5 && j – i >= 0 & i - j >= 0.

-sp*i + sc*j + (dc –dp) >= 0

This Farkas lemma trick helps with the objective function too!

optimize: **some function that captures the locality of the schedules**

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j. sp*i + dp <= sc*j + dc

Create a variable that represents a bound on the time between producers and consumers

minimize: w

subject to:

```
forall 0 <= i <= 5 && 0 <= j <= 5 && i = j.
  sp*i + dp <= sc*j + dc

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j.
  sp*i + dp – sc*j – dc <= w
```

In general we might want…

minimize: wp + wn

subject to:

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j.
sp*i + dp <= sc*j + dc

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j.
sp*i + dp – sc*j – dc <= w

w = wp – wn
wp, wn >= 0

We can use the same strategy to push dependencies further apart to create parallelism

maximize: e

subject to:

```
forall 0 <= i <= 5 && 0 <= j <= 5 && i = j.
  sp*i + dp <= sc*j + dc

forall 0 <= i <= 5 && 0 <= j <= 5 && i = j.
  e <= sc*j + dc - sp*i - dp

0 <= e <= UPPER_BOUND_ON_DISTANCE
```

Multiple dimensions can be handled iteratively

```
Dependence_Graph = Initial_DD(prog)
Schedule_Vectors = []
while (!empty(Dependence_Graph))
  Next_Schedule_Levels = Solve_ILP(DependenceGraph, objective)
  Schedule_vectors.append(Next_Schedule_Levels)
  Dependence_Graph = Remove_Carried_Deps(Next_Schedule_Levels)
```

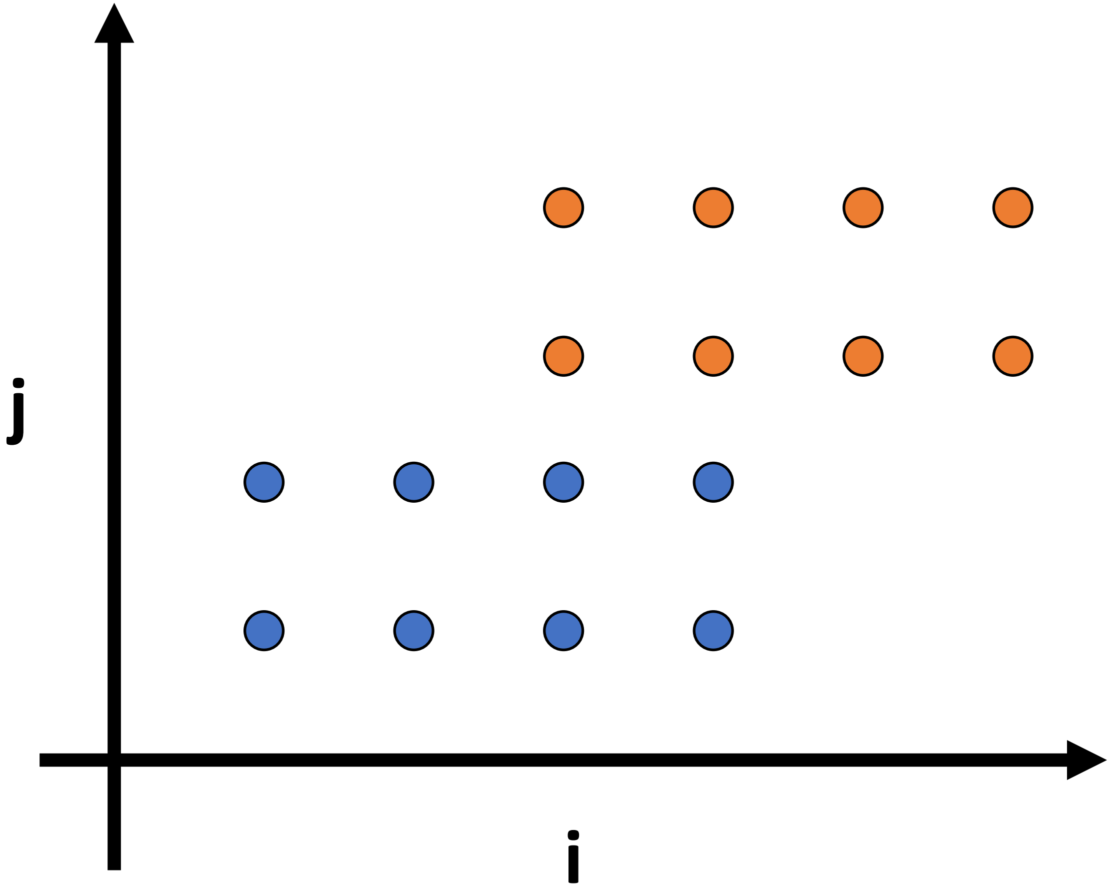
So how do we turn polyhedral schedules back into for loops?

Two rectangular iteration domains…

Output polyhedra

A = { [i, j] | 1 <= i <= 4 and 1 <= j <= 2 }

B = { [i, j] | 3 <= i <= 6 and 3 <= j <= 4 }

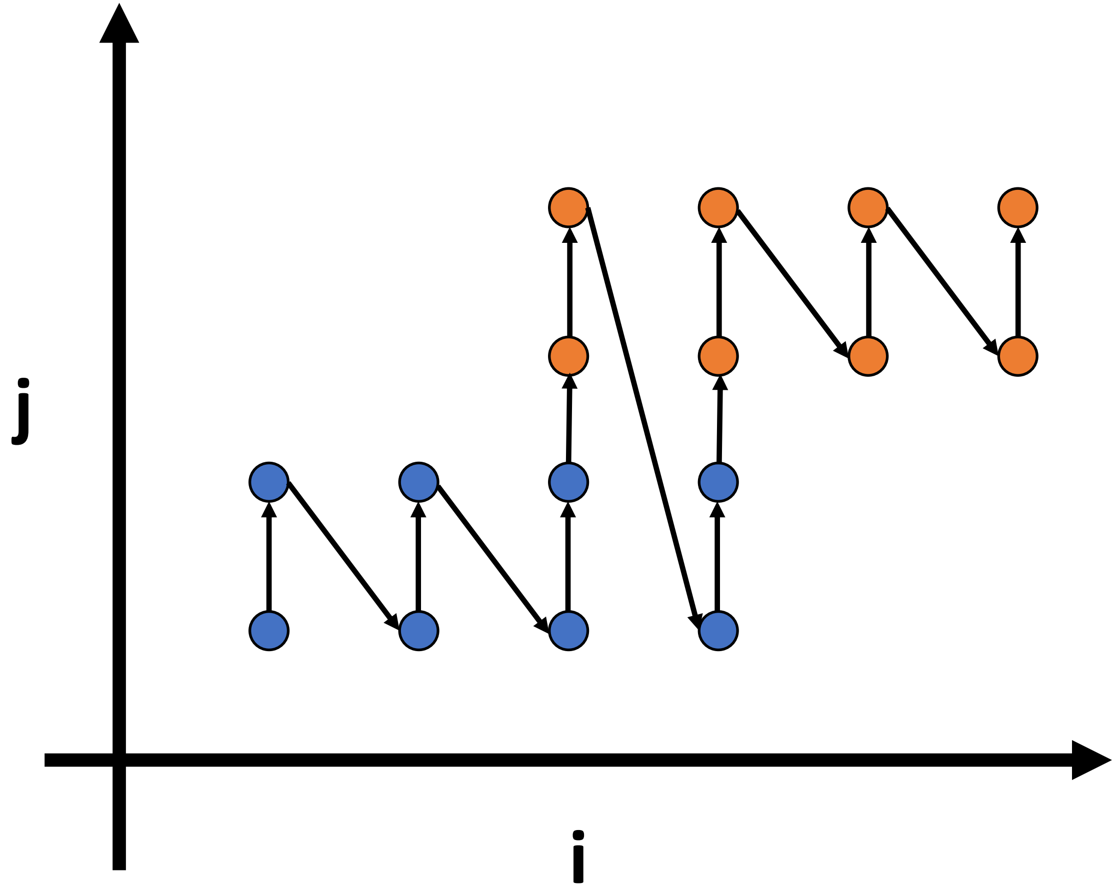


How do we create loops for these points in lexicographic order?

Output polyhedra

A = { [i, j] | 1 <= i <= 4 and 1 <= j <= 2 }

B = { [i, j] | 3 <= i <= 6 and 3 <= j <= 4 }

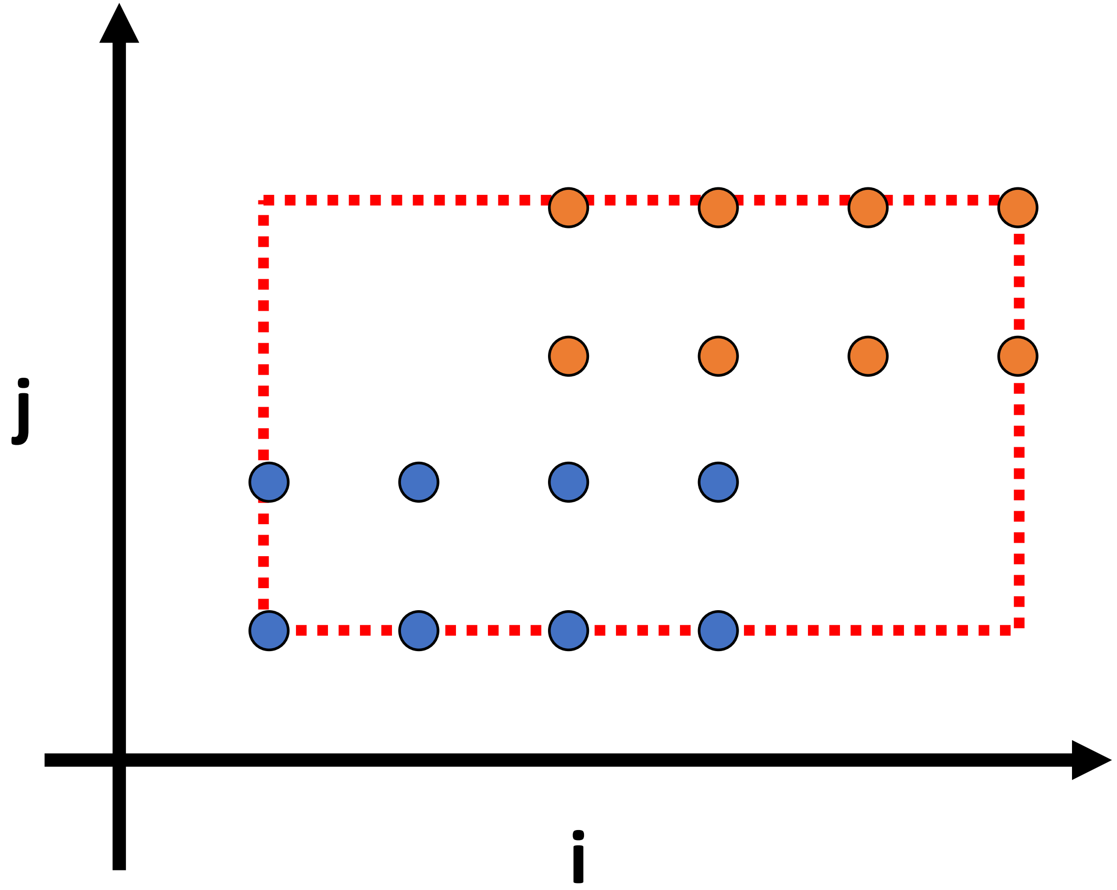


Easy: Compute a hull of all statements, iterate over it with a perfect loop nest, and use the polyhedra as guards

Output polyhedra

A = { [i, j] | 1 <= i <= 4 and 1 <= j <= 2 }

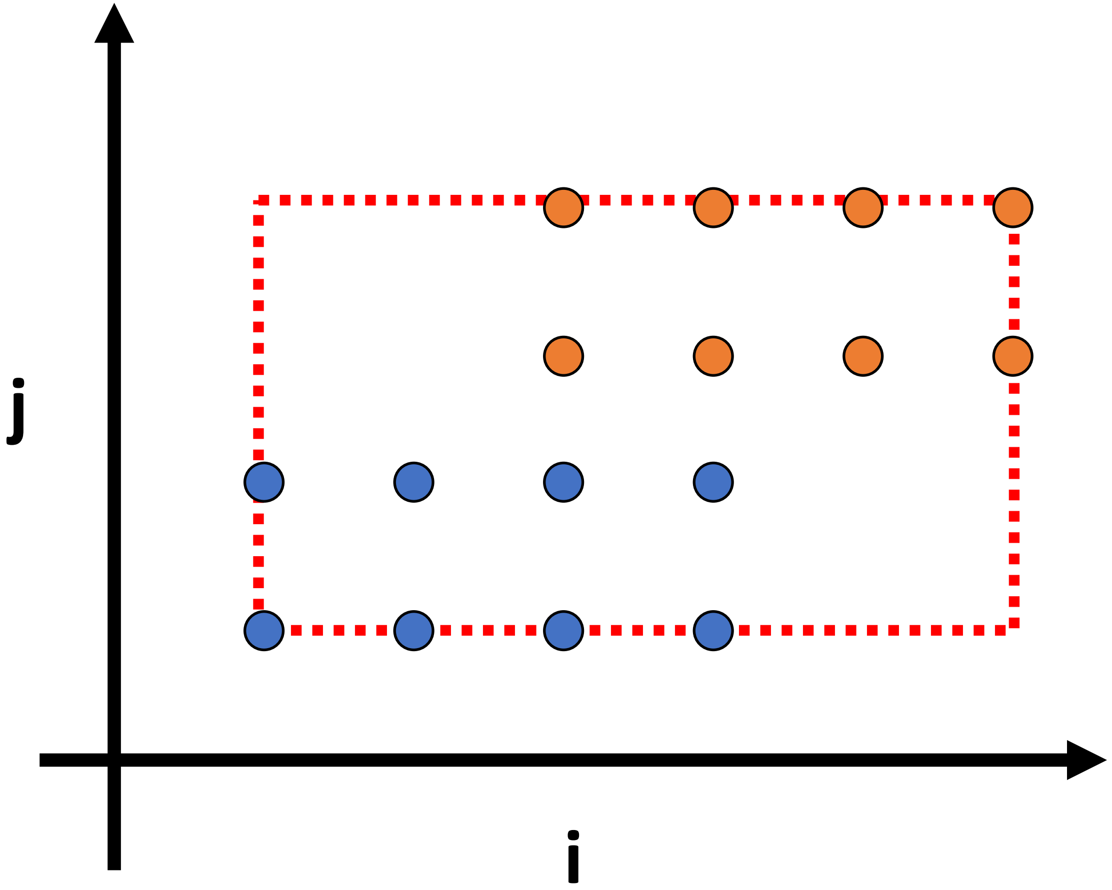
B = { [i, j] | 3 <= i <= 6 and 3 <= j <= 4 }



Easy: Compute a hull of all statements, iterate over it with a perfect loop nest, and use the polyhedra as guards

Scanning loops

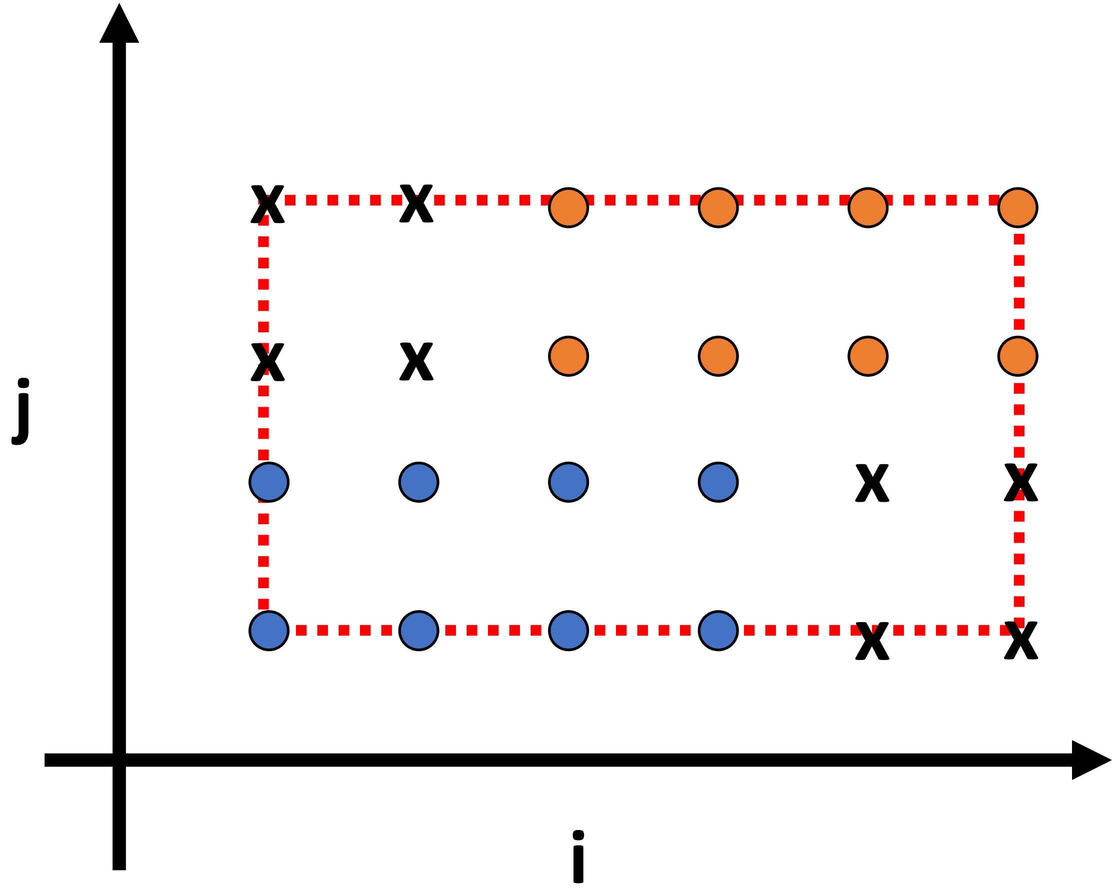
```
for (int i = 1; i <= 6; i++)
  for (int j = 1; j <= 4; j++) {
    if (1 <= i && i <= 4 && 1 <= j && j <= 2)
      A(i, j)
    if (3 <= i && i <= 6 && 3 <= j && j <= 4)
      B(i, j)
  }
```



Easy but inefficient: Compute a hull of all statements, iterate over it with a perfect loop nest, and use the polyhedra as guards

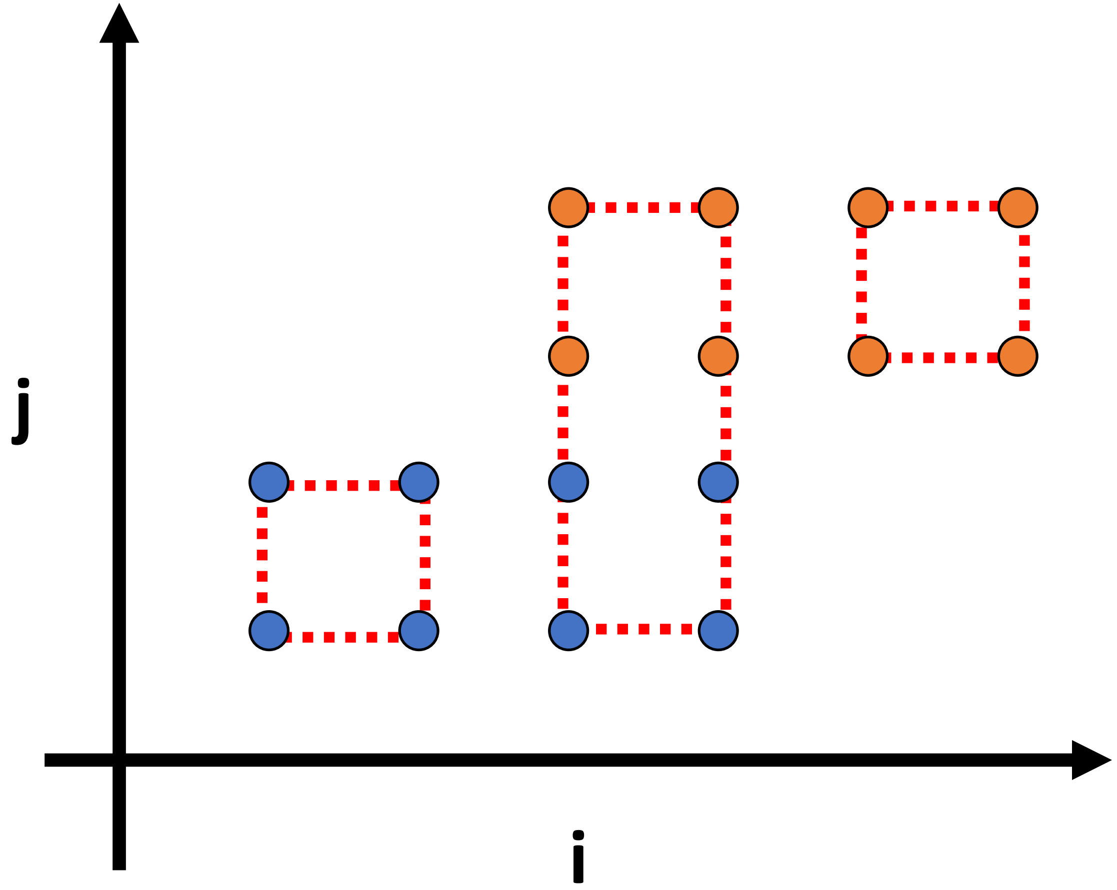
Scanning loops

```
for (int i = 1; i <= 6; i++)
  for (int j = 1; j <= 4; j++) {
    if (1 <= i && i <= 4 && 1 <= j && j <= 2)
      A(i, j)
    if (3 <= i && i <= 6 && 3 <= j && j <= 4)
      B(i, j)
  }
```



Harder but more efficient: Use projection to isolate regions with the same statements

```
for (int i = 1; i <= 2; i++)
  for (int j = 1; j <= 2; j++)
    A(i, j)
for (int i = 3; i <= 4; i++)
  for (int j = 1; j <= 2; j++) {
    if (1 <= j && j <= 2)
      A(i, j)
    if (3 <= j && j <= 4)
      B(i, j)
  }
for (int i = 5; i <= 6; i++)
  for (int j = 3; j <= 4; j++)
    B(i, j)
```



There are many other code generation tricks…

- **But:** Polyhedral code generation still doesn’t work that well

It’s a powerful tool, but only in a narrow domain…

- Modest size programs
- With (quasi)affine address expressions and bounds
- Code generation is tricky
- Counting is even harder (Barvinok)
- Ravi Mullapudi: “Polyhedral analysis is great *for analysis*”
- Standard tool for polyhedral analysis: ISL by Sven Verdooleage (generic) Polly (LLVM)