## Compiling for Parallelism & Locality

**Last time**

– Predication and speculation

**Today**

– Data dependences and loops

– Parallelism and locality

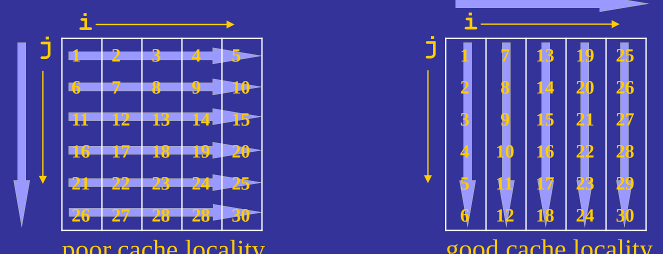


## Example 1: Loop Permutation for Improved Locality

**What do we notice about the following code?**

– Assume Fortran’s Column Major Order array layout

```
do j = 1,6                    do i = 1,5
   do i = 1,5                    do j = 1,6
      A(j,i) = A(j,i)+1             A(j,i) = A(j,i)+1
   enddo                         enddo
enddo                         enddo
```




poor cache locality good cache locality

## Example 2: Parallelization

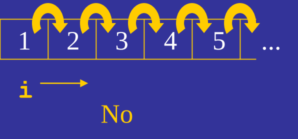
**Can we parallelize the following loop?**

– Can we execute the different iterations at the same time?

```
do i = 1,100
   A(i) = A(i)+1
enddo
```


**How about this loop?**

```
do i = 1,100
   A(i) = A(i-1)+1
enddo
```






## Data Dependences

**Recall**

– A data dependence defines an ordering relationship between two statements

– In executing statements, data dependences must be respected to preserve correctness

**Example**

| | | | | |
|---|---|---|---|---|
| $s_1$ | `a := 5;` | | $s_1$ | `a := 5;` |
| $s_2$ | `b := a + 1;` | $\stackrel{?}{\equiv}$ | $s_3$ | `a := 6;` |
| $s_3$ | `a := 6;` | | $s_2$ | `b := a + 1;` |

## Data Dependences and Loops

**How do we identify dependences in loops?**

```
do i = 1,5
   A(i) = A(i-1)+1
enddo
```

**Simple view**

– Imagine that all loops are fully unrolled
– Examine data dependences as before

**Problems?**

– Impractical
– Lose loop structure

```
A(1) = A(0)+1
A(2) = A(1)+1
A(3) = A(2)+1
A(4) = A(3)+1
A(5) = A(4)+1
```



## Dependence Analysis for Loops

**Big picture**

– To improve data locality and parallelism we often focus on loops
– To transform loops, we must understand data dependences in loops
– Since we can’t represent all iterations of a loop, we need some abstractions
– The basic question: does a transformation preserve all dependences?

**Today**

– Basic abstractions and machinery

**Next class**

– Its application

## Recall Data Dependence Terminology

**We say statement $s_2$ depends on $s_1$**

- **True (flow) dependence**: $s_1$ writes memory that $s_2$ later reads
- **Anti-dependence**: $s_1$ reads memory that $s_2$ later writes
- **Output dependences**: $s_1$ writes memory that $s_2$ later writes
- **Input dependences**: $s_1$ reads memory that $s_2$ later reads

**Notation:** $s_1 \, \delta \, s_2$

- $s_1$ is called the **source** of the dependence
- $s_2$ is called the **sink** or **target**
- $s_1$ must be executed before $s_2$

April 27, 2015 Compiling for Parallelism and Locality 7

## Dependences and Loops

**Loop-independent dependences**

```
do i = 1,100
   A(i) = B(i)+1
   C(i) = A(i)*2
enddo
```

Dependences within the same loop iteration

**Loop-carried dependences**

```
do i = 1,100
   A(i) = B(i)+1
   C(i) = A(i-1)*2
enddo
```

Dependences that cross loop iterations– these depend on the loop structure

April 27, 2015 Compiling for Parallelism and Locality 8

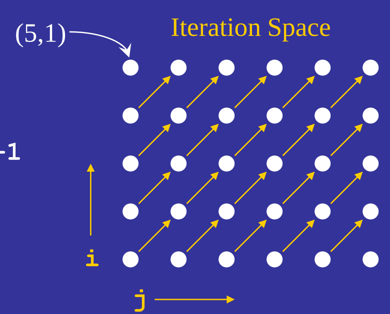
## Iteration Spaces

**Idea**

– Explicitly represent the iterations of a loop nest

**Example**

```
do i = 1,5
   do j = 2,6
      A(j,i) = A(j-1,i-1)+1
   enddo
enddo
```




**Iteration Space**

– A set of tuples that represents the iterations of a loop

– Can use an iteration space to visualize the loop's dependences



## Distance Vectors

**Idea**

– Concisely describe dependence relationships among iterations of an iteration space

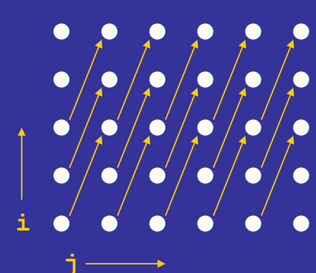
– For each dimension of an iteration space, the distance is the number of iterations between accesses to the same memory location

**Definition**

– $\mathbf{v} = \mathbf{i}^T - \mathbf{i}^S$

**Example**

```
do i = 1,5
   do j = 2,6
      A(j,i) = A(j-1,i-2)+1
   enddo
enddo
```




outer loop (i)

**Distance Vector:** (2,1)

inner loop (j)

## Distance Vectors Example

**Example**

```
do i = 1,5
   do j = 2,6
      A(j,i) = A(j-1,i-2)+1
   enddo
enddo
```

When does `A(j-1,i-2)` get written?

| Iteration | Write | Read |
|---|---|---|
| $i^T$ `(i,j)` | `A(j,i)` | `A(j-1,i-2)` |
| $i^S$ `(i-2,j-1)` | `A(j-1,i-2)` | . . . |

RAW dependence

**Which iteration comes first?**

`(i-2,j-1)`

$v = i^T - i^S =$ `(i,j) – (i-2,j-1) = (2,1)`



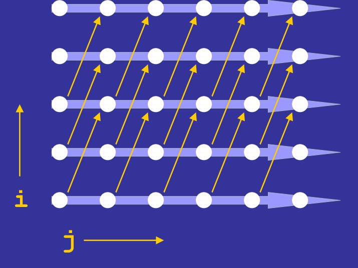
## Distance Vectors and Loop Transformations

**Idea**

– Any transformation we perform on the loop must respect the dependences

**Example**

```
do i = 1,5
   do j = 2,6
      A(j,i) = A(j-1,i-2)+1
   enddo
enddo
```




**Can we permute the `i` and `j` loops?**

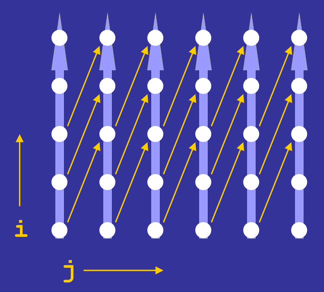
## Distance Vectors and Loop Transformations (cont)

**Idea**

– Any transformation we perform on the loop must respect the dependences

**Example**

```
do j = 2,6
   do i = 1,5
      A(j,i) = A(j-1,i-2)+1
   enddo
enddo
```




**Can we permute the `i` and `j` loops?**

– Yes

April 27, 2015 Compiling for Parallelism and Locality 13

## Example 2

**Code**

```
do i = 1,5
   do j = 2,6
      A(j,i) = A(j-1,i+1)+1
   enddo
enddo
```

**When does `A(j-1,i+1)` get written?**

| Iteration | Write | Read |
|---|---|---|
| $i^S$ (i,j) | A(j,i) | A(j-1,i+1) |
| $i^T$ (i+1,j-1) | A(j-1,i+1) | . . . WAR dependence |

**Which iteration comes first?**

(i,j)

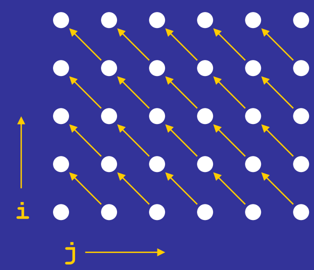
$v = i^T - i^S =$ (i+1,j-1) – (i,j) = (1,-1)

April 27, 2015 Compiling for Parallelism and Locality 14

## Example 2 (cont)

**Sample code**

```
do i = 1,5
   do j = 2,6
      A(j,i) = A(j-1,i+1)+1
   enddo
enddo
```




**Kind of dependence:** Anti

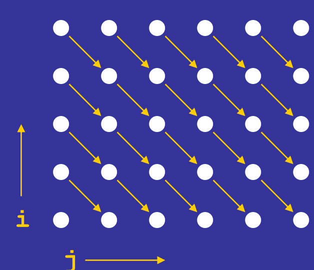
**Distance vector:** (1, –1)



## Exercise: Consider Loop Permutation

**Sample code**

```
do j = 2,6
   do i = 1,5
      A(j,i) = A(j-1,i+1)+1
   enddo
enddo
```




**Kind of dependence:** Flow

**Distance vector:** (1, -1)

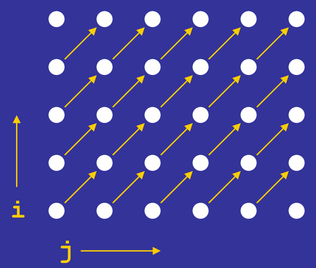
## Direction Vector

**Definition**

- A direction vector serves the same purpose as a distance vector when less precision is required or available
- Element *i* of a direction vector is <, >, or = based on whether the source of the dependence precedes, follows, or is in the same iteration as the target in loop *i*

**Example**

```
do i = 1,5
   do j = 2,6
      A(j,i) = A(j-1,i-1)+1
   enddo
enddo
```




**Direction vector:** (<,<)

**Distance vector:** (1,1)



## Distance Vectors: Legality

**Definition**

- A dependence vector, *v*, is **lexicographically nonnegative** when the left-most non-zero entry in *v* is positive or when all elements of *v* are zero
  Yes: (0,0,0), (0,1), (0,2,-2)
  No: (-1), (0,-2), (0,-1,1)
- A dependence vector is **legal** when it is lexicographically nonnegative (assuming that indices increase as we iterate)

**Why are lexicographically negative distance vectors illegal?**

**What are legal direction vectors?**