

# Compiling for Parallelism & Locality

---

## Last time

- SSA and its uses

## Today

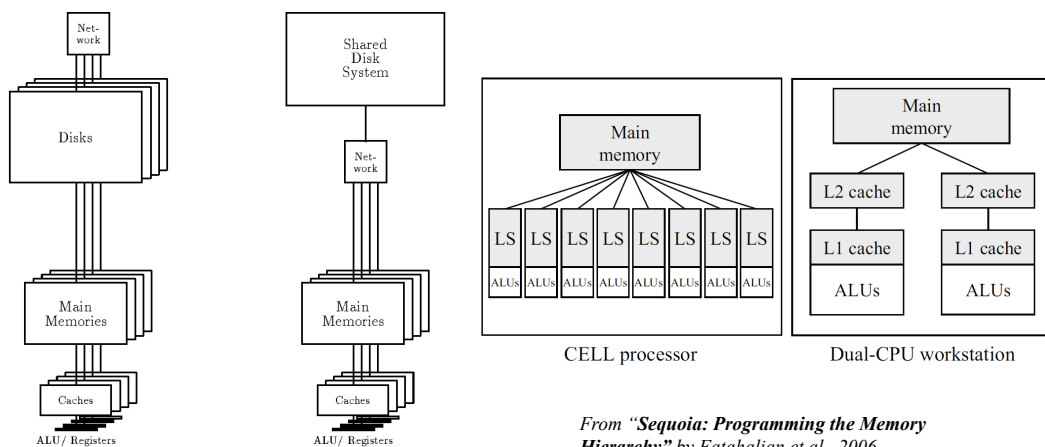
- Parallelism and locality
- Data dependences and loops

## The Problem: Mapping programs to architectures

---

**Goal:** keep each core as busy as possible.

**Challenge:** get the data to the core when it needs it



From "Modeling Parallel Computers as Memory Hierarchies" by B. Alpern and L. Carter and J. Ferrante, 1993.

From "Sequoia: Programming the Memory Hierarchy" by Fatahalian et al., 2006.

## Example 1: Loop Permutation for Improved Locality

Sample code: Assume Fortran's Column Major Order array layout

```

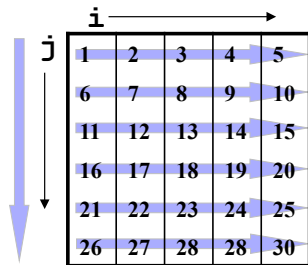
do j = 1,6
  do i = 1,5
    A(j,i) = A(j,i)+1
  enddo
enddo

```

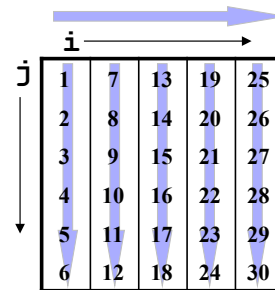
```

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

```



poor cache locality



good cache locality

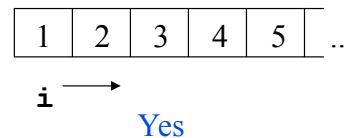
## Example 2: Parallelization

Can we parallelize the following loops?

```

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

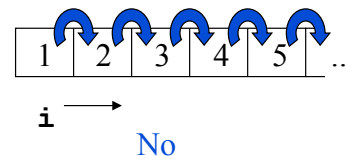
```



```

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

```



## Data Dependences

---

### Recall

- A data dependence defines ordering relationship two between statements
- In executing statements, data dependences must be respected to preserve correctness

### Example

$S_1$	$a := 5;$	?	$S_1$	$a := 5;$
$S_2$	$b := a + 1;$	≡	$S_3$	$a := 6;$
$S_3$	$a := 6;$		$S_2$	$b := a + 1;$

## Data Dependences and Loops

---

### How do we identify dependences in loops?

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

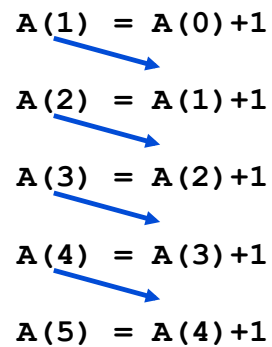
### Simple view

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

### Problems

- Impractical and often impossible
- 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
```



## Concepts needed for automating loop transformations

---

### Questions

- How do we determine if a transformation or parallelization is legal?
- What abstraction do we use for loops?
- How do we represent transformations and parallelization?
- How do we generate the transformed code?
- How do we determine when a transformation is going to be beneficial?

### Today

- Basic abstractions for loops and dependences and computing dependences

### Thursday

- Abstractions for loop transformations and determining their legality
- Code generation after performing a loop transformation

## 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

## Dependence Testing in General

---

### General code

```
do i1 = l1, h1
...
  do in = ln, hn
    A(f(i1, ..., in)) = ... A(g(i1, ..., in))
  enddo
...
enddo
```

There exists a dependence between iterations  $I=(i_1, \dots, i_n)$  and  $J=(j_1, \dots, j_n)$  when

- $f(I) = g(J)$
- $(l_1, \dots, l_n) < I, J < (h_1, \dots, h_n)$
- $I < J$  or  $J < I$ , where  $<$  is lexicographically less

## Algorithms for Solving the Dependence Problem

---

### Heuristics can say NO or MAYBE

- GCD test (Banerjee76, Towle76): determines whether integer solution is possible, no bounds checking
- Banerjee test (Banerjee 79): checks real bounds
- Independent-variables test (pg. 820): useful when inequalities are not coupled
- I-Test (Kong et al. 90): integer solution in real bounds
- Lambda test (Li et al. 90): all dimensions simultaneously
- Delta test (Goff et al. 91): pattern matches for efficiency
- Power test (Wolfe et al. 92): extended GCD and Fourier Motzkin combination

### Use some form of Fourier-Motzkin elimination for integers, exponential worst-case

- Parametric Integer Programming (Feautrier91)
- Omega test (Pugh92)

## Dependence Testing

---

Consider the following code...

```
do i = 1,5
  A(3*i+2) = A(2*i+1)+1
enddo
```

### Question

- How do we determine whether one array reference depends on another across iterations of an iteration space?

## Dependence Testing: Simple Case

---

Sample code

```
do i = 1,h
  A(a*i+c1) = ... A(a*i+c2)
enddo
```

Dependence?

- $a \cdot i_1 + c_1 = a \cdot i_2 + c_2$ , OR
- $a \cdot i_1 - a \cdot i_2 = c_2 - c_1$
- Solution may exist if  $a$  divides  $c_2 - c_1$

## GCD Test

---

### Idea

- Generalize test to linear functions of iterators/induction variables

### Code

```
do i = li, hi
  do j = lj, hj
    A(a1*i + a2*j + a0) = ... A(b1*i + b2*j + b0) ...
  enddo
enddo
```

### Again

- $a_1*i_1 - b_1*i_2 + a_2*j_1 - b_2*j_2 = b_0 - a_0$
- Solution exists if  $\text{gcd}(a_1, a_2, b_1, b_2)$  divides  $b_0 - a_0$

## Example

---

### Code

```
do i = li, hi
  do j = lj, hj
    A(4*i + 2*j + 1) = ... A(6*i + 2*j + 4) ...
  enddo
enddo
```

$$\text{gcd}(4, -6, 2, -2) = 2$$

Does 2 divide 4-1?

## Banerjee Test

---

```
for (i=L; i<=U; i++) {  
    x[a0 + a1*i] = ...  
    ... = x[b0 + b1*i]  
}
```

Does  $a_0 + a_1*i = b_0 + b_1*i'$  for some real  $i$  and  $i'$ ?

If so then  $(a_1*i - b_1*i') = (b_0 - a_0)$

Determine upper and lower bounds on  $(a_1*i - b_1*i')$

```
for (i=1; i<=5; i++) {  
    x[i+5] = x[i];  
}
```

upper bound =  $a_1*\max(i) - b_1*\min(i') = 4$

lower bound =  $a_1*\min(i) - b_1*\max(i') = -4$


$b_0 - a_0 =$

## Example 1: Loop Permutation (reprise)

---

### Sample code

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



### Why is this legal?

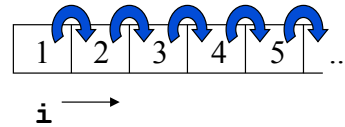
- No loop-carried dependences, so we can arbitrarily change order of iteration execution

## Example 2: Parallelization (reprise)

---

Why can't this loop be parallelized?

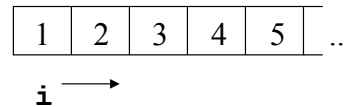
```
do i = 1,100
  A(i) = A(i-1)+1
enddo
```



Loop carried dependence

Why can this loop be parallelized?

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



No loop carried dependence,  
No solution to dependence problem

## Iteration Spaces

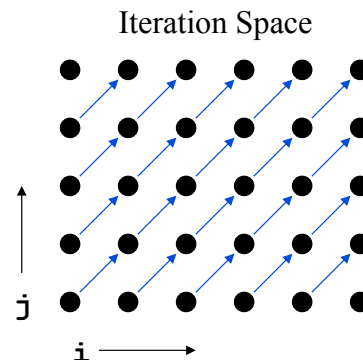
---

**Idea**

- Explicitly represent the iterations of a loop nest

**Example**

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



**Iteration Space**

- A set of tuples that represents the iterations of a loop
- Can visualize the dependences in an iteration space

## Distance Vectors

### Idea

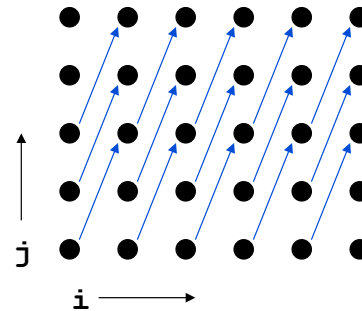
- Concisely describe dependence relationships between iterations of an iteration space
- For each dimension of an iteration space, the distance is the number of iterations between accesses to the same memory location

### Definition

- $v = i^T - j^S$

### Example

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



Distance Vector: (1,2)

outer loop

inner loop

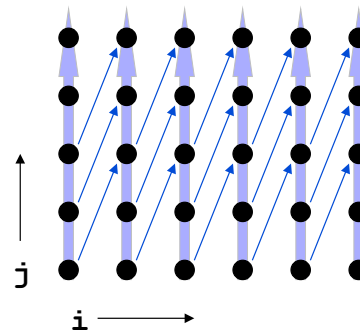
## Distance Vectors and Loop Transformations

### Idea

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

### Example

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



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

## Distance Vectors and Loop Transformations

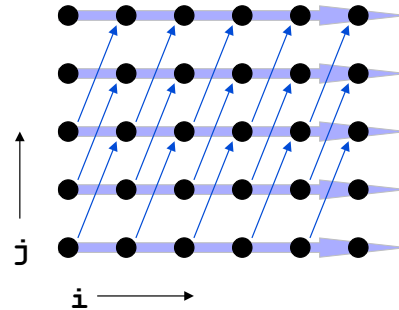
---

### Idea

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

### Example

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



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

- Yes

## Distance Vectors: Legality

---

### Definition

- A dependence vector,  $v$ , is **lexicographically nonnegative** when the left-most entry in  $v$  is positive or 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?

## 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  $s_2$

## Example

---

**Code**

```
do i = 1,h  
  A(2*i+2) = A(2*i-2)+1  
enddo
```

$i_1$

$i_2$

**Dependence?**

$$2*i_1 - 2*i_2 = -2 - 2 = -4$$

(yes, 2 divides -4)

**Kind of dependence?**

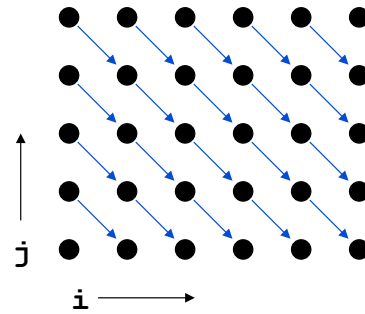
- Anti?  $i_2 + d = i_1 \Rightarrow d = -2$
- Flow?  $i_1 + d = i_2 \Rightarrow d = 2$

## Example

---

### Sample code

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



**Kind of dependence:** Flow

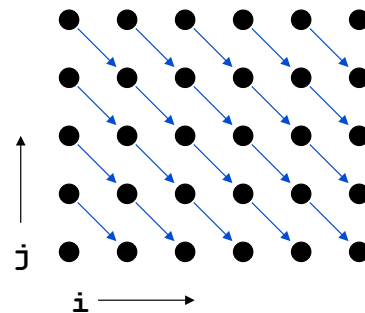
**Distance vector:** (1, -1)

## Exercise

---

### Sample code

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



**Kind of dependence:** Anti

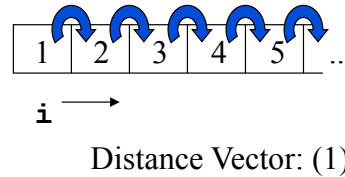
**Distance vector:** (1, -1)

## Example 2: Parallelization (reprise)

---

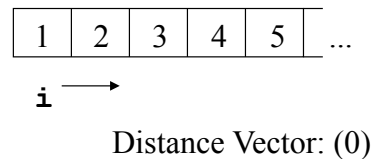
Why can't this loop be parallelized?

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



Why can this loop be parallelized?

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



## Protein String Matching Example

---

```
q = k_1
r = k_2
score[i,j] = 0 for the whole array

for i=1 to n1-1
  h[i,0] = p[i,0] = 0
  f[i,0] = -q
  for j=1 to n0-1
    f[i,j] = max(f[i,j-1],h[i,j-1]-q)-r
    EE[i,j] = max(EE[i-1,j],HH[i-1,j],-q)-r
    h[i,j] = p[i,j-1] + pam2[aa1[i],aa0[j]]
    h[i,j] = max( max(0,EE[i,j]), max(f[i,j],h[i,j]) )
    p[i,j] = HH[i-1,j]
    HH[i,j] = h[i,j]
    score[i,j] = max(score[i,j-1],h[i,j])
  endfor
endfor

return score[n1-1,n0-1]
```

## Loop-Carried Dependences

### Definition

- A dependence  $D=(d_1, \dots, d_n)$  is **carried** at loop level  $i$  if  $d_i$  is the first nonzero element of  $D$

### Example

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

**Distance vectors:** (0,1) for accesses to **A**  
(1,0) for accesses to **B**

### Loop-carried dependences

- The  $j$  loop carries dependence due to **A**
- The  $i$  loop carries dependence due to **B**

## Parallelization

### Idea

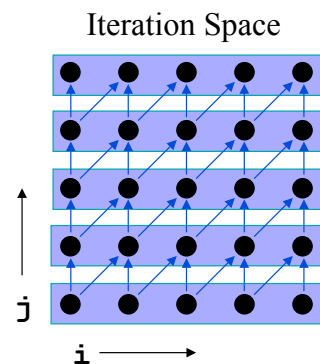
- Each iteration of a loop may be executed in parallel if that loop carries no dependences

### Example (different from last slide)

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

**Parallelize i loop?** Distance Vectors:

- (1,0) for **A** (flow)
- (1,1) for **B** (flow)



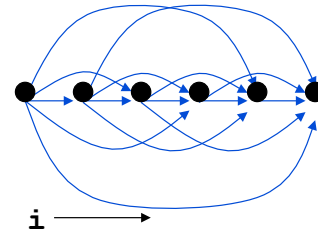
## Loop-Carried, Storage-Related Dependences

### Problem

- Loop-carried dependences inhibit parallelism
- Scalar references result in loop-carried dependences

### Example

```
do i = 1,6
  t = A(i) + B(i)
  C(i) = t + 1/t
enddo
```



Can this loop be parallelized? No.

What kind of dependences are these? Anti dependences.

Convention for these slides: Arrays start with upper case letters, scalars do not

## Direction Vector

### Definition

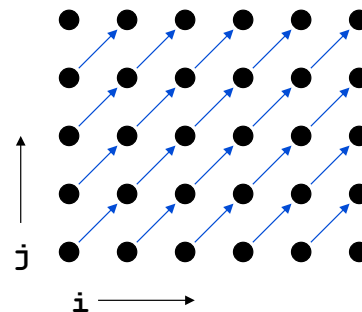
- A direction vector serves the same purpose as a distance vector when less precision is required or available
- Element  $i$  of a direction vector is  $<$ ,  $>$ , or  $=$  based on whether the source of the dependence precedes, follows or is in the same iteration as the target in loop  $i$

### Example

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

Direction vector: ( $<$ , $<$ )

Distance vector: (1,1)



## Removing False Dependences with Scalar Expansion

---

### Idea

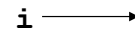
- Eliminate false dependences by introducing extra storage

### Example

```
do i = 1,6
  T(i) = A(i) + B(i)
  C(i) = T(i) + 1/T(i)
enddo
```



Can *this* loop be parallelized?



### Disadvantages?

## Scalar Expansion Details

---

### Restrictions

- The loop must be a **countable** loop  
*i.e.* The loop trip count must be independent of the body of the loop
- The expanded scalar must have no **upward exposed uses** in the loop

```
do i = 1,6
  print(t)
  t = A(i) + B(i)
  C(i) = t + 1/t
enddo
```

- Nested loops may require much more storage
- When the scalar is live after the loop, we must move the correct array value into the scalar

## Concepts

---

### **Improve performance by ...**

- improving data locality
- parallelizing the computation

### **Data Dependence Testing**

- general formulation of the problem
- GCD test and Banerjee test

### **Data Dependences**

- iteration space
- distance vectors and direction vectors
- loop carried

### **Transformation legality**

- must respect data dependences

## Next Time

---

### **Reading**

- Ch 11.4-11.6

### **Lecture**

- Abstractions for loop transformations and checking their legality
- Code generation after transformations have been performed