# Dependence Analysis



## Motivating question

- Can the loops on the right be run in parallel?
  - *i.e.*, can different processors run different iterations in parallel?
- What needs to be true for a loop to be parallelizable?
  - Iterations cannot interfere with each other
  - No *dependence* between iterations

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}

for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i] + b[i - 1];
}
```



## Dependences

- A *flow dependence* occurs when one iteration writes a location that a *later* iteration reads

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

```
i = 1      i = 2      i = 3      i = 4      i = 5

W(a[1])    W(a[2])    W(a[3])    W(a[4])    W(a[5])
R(b[1])    R(b[2])    R(b[3])    R(b[4])    R(b[5])
W(c[1])    W(c[2])    W(c[3])    W(c[4])    W(c[5])
R(a[0])    R(a[1])    R(a[2])    R(a[3])    R(a[4])
```



## Running a loop in parallel

- If there is a dependence in a loop, we cannot guarantee that the loop will run correctly in parallel
  - What if the iterations run out of order?
    - Might read from a location before the correct value was written to it
  - What if the iterations do not run in lock-step?
    - Same problem!



## Other kinds of dependence

- *Anti dependence* – When an iteration *reads* a location that a later iteration *writes* (why is this a problem?)

```
for (i = 1; i < N; i++) {
  a[i - 1] = b[i];
  c[i] = a[i];
}
```

- *Output dependence* – When an iteration *writes* a location that a later iteration *writes* (why is this a problem?)

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  a[i + 1] = c[i];
}
```



## Data dependence concepts

- Dependence *source* is the earlier statement (the statement at the tail of the dependence arrow)
- Dependence *sink* is the later statement (the statement at the head of the dependence arrow)

```
i = 1      i = 2      i = 3      i = 4      i = 5

W(a[1])    W(a[2])    W(a[3])    W(a[4])    W(a[5])
R(b[1])    R(b[2])    R(b[3])    R(b[4])    R(b[5])
W(c[1])    W(c[2])    W(c[3])    W(c[4])    W(c[5])
R(a[0])    R(a[1])    R(a[2])    R(a[3])    R(a[4])
```

- Dependences can only go forward in time: always from an earlier iteration to a later iteration.

# Using dependences

- If there are no dependences, we can parallelize a loop
  - None of the iterations interfere with each other
- Can also use dependence information to drive other optimizations
  - Loop interchange
  - Loop fusion
  - (We will discuss these later)
- Two questions:
  - How do we represent dependences in loops?
  - How do we determine if there are dependences?



# Representing dependences

- Focus on flow dependences for now
- Dependences in straight line code are easy to represent:
  - One statement writes a location (variable, array location, etc.) and another reads that same location
  - Can figure this out using reaching definitions
- What do we do about loops?
  - We often care about dependences between the same statement in different iterations of the loop!

```
for (i = 1; i < N; i++) {
  a[i + 1] = a[i] + 2
}
```



# Iteration space graphs

- Represent each *dynamic* instance of a loop as a point in a graph
- Draw arrows from one point to another to represent dependences

```
for (i = 0; i < N; i++) {
  a[i + 2] = a[i]
}
```



# Iteration space graphs

- Represent each *dynamic* instance of a loop as a point in a graph
- Draw arrows from one point to another to represent dependences

```
for (i = 0; i < N; i++) {
  a[i + 2] = a[i]
}
```

- Step 1: Create nodes, 1 for each iteration
  - Note: not 1 for each array location!

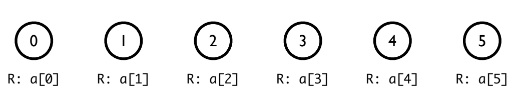



# Iteration space graphs

- Represent each *dynamic* instance of a loop as a point in a graph
- Draw arrows from one point to another to represent dependences

```
for (i = 0; i < N; i++) {
  a[i + 2] = a[i]
}
```

- Step 2: Determine which array elements are read and written in each iteration



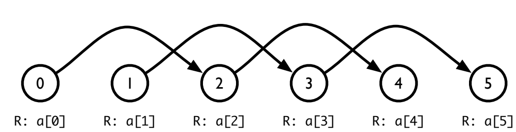



# Iteration space graphs

- Represent each *dynamic* instance of a loop as a point in a graph
- Draw arrows from one point to another to represent dependences

```
for (i = 0; i < N; i++) {
  a[i + 2] = a[i]
}
```

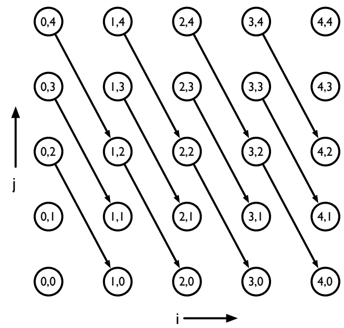
- Step 3: Draw arrows to represent dependences



# 2-D iteration space graphs

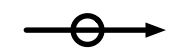
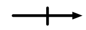
- Can do the same thing for doubly-nested loops
  - 2 loop counters

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j-2] = a[i][j] + 1
```






# Iteration space graphs

- Can also represent output and anti dependences
  - Use different kinds of arrows for clarity. *E.g.*
  -  for output
  -  for anti
- Crucial problem: Iteration space graphs are potentially infinite representations!
  - Can we represent dependences in a more compact way?



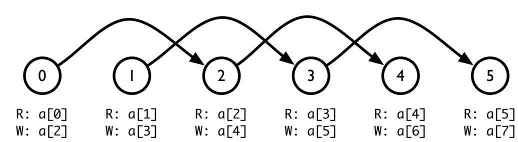
# Distance and direction vectors

- Compiler researchers have devised *compressed* representations of dependences
  - Capture the same dependences as an iteration space graph
  - May lose *precision* (show more dependences than the loop actually has)
- Two types
  - Distance vectors: captures the "shape" of dependences, but not the particular source and sink
  - Direction vectors: captures the "direction" of dependences, but not the particular shape



# Distance vector

- Represent each dependence arrow in an iteration space graph as a vector
  - Captures the "shape" of the dependence, but loses where the dependence originates

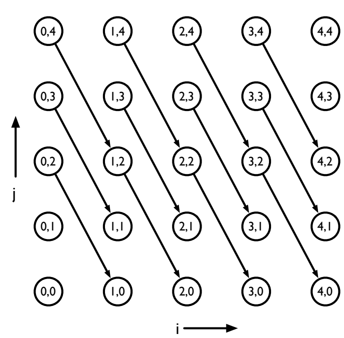



- Distance vector for this iteration space: (2)
  - Each dependence is 2 iterations forward



# 2-D distance vectors

- Distance vector for this graph:
  - (1, -2)
  - +1 in the i direction, -2 in the j direction
- Crucial point about distance vectors: they are always "positive"
  - First non-zero entry has to be positive
  - Dependences can't go backwards in time

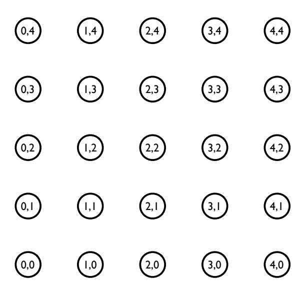





# More complex example

- Can have multiple distance vectors

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j-2] = a[i][j] +
                a[i-1][j-2]
```

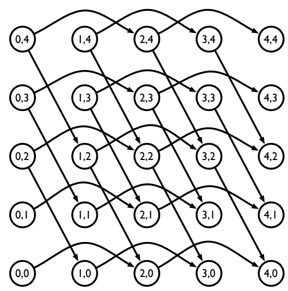


# More complex example

- Can have multiple distance vectors

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j-2] = a[i][j] +
               a[i-1][j-2]
```

- Distance vectors
  - (1, -2)
  - (2, 0)
- Important point: order of vectors depends on order of loops, not use in arrays

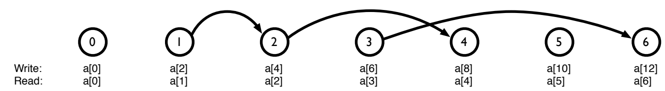





# Problems with distance vectors

- The preceding examples show how distance vectors can summarize all the dependences in a loop nest using just a small number of distance vectors
- Can't always summarize as easily
- Running example:

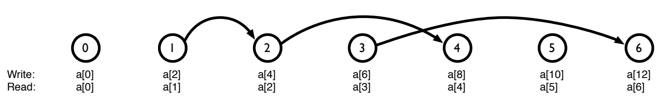
```
for (i = 0; i < N; i++)
  a[2*i] = a[i];
```






# Loss of precision

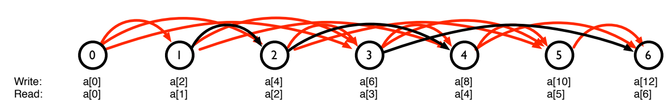
- What are the distance vectors for this code?
  - (1), (2), (3), (4) ...
- Note: we have information about the length of each vector, but not about the source of each vector
  - What happens if we try to reconstruct the iteration space graph?






# Loss of precision

- What are the distance vectors for this code?
  - (1), (2), (3), (4) ...
- Note: we have information about the length of each vector, but not about the source of each vector
  - What happens if we try to reconstruct the iteration space graph?






# Direction vectors

- The whole point of distance vectors is that we want to be able to succinctly capture the dependences in a loop nest
  - But in the previous example, not only did we add a lot of extra information, we still had an infinite number of distance vectors
- Idea: summarize distance vectors, and save only the *direction* the dependence was in
  - (2, -1) → (+, –)
  - (0, 1) → (0, +)
  - (0, -2) → (0, –)
    - (can't happen; dependences have to be positive)
  - Notation: sometimes use '<' and '>' instead of '+' and '–'



# Why use direction vectors?

- Direction vectors lose a lot of information, but do capture some useful information
  - Whether there is a dependence (anything other than a '0' means there is a dependence)
  - Which dimension and direction the dependence is in
- Many times, the only information we need to determine if an optimization is legal is captured by direction vectors
  - Loop parallelization
  - Loop interchange

# Loop parallelization



# Loop-carried dependence

- The key concept for parallelization is the *loop carried dependence*
  - A dependence that crosses loop iterations
- If there is a loop carried dependence, then that loop *cannot* be parallelized
  - Some iterations of the loop depend on other iterations of the same loop



# Examples

```
for (i = 0; i < N; i++)
  a[2*i] = a[i];
```

Later iterations of i loop depend on earlier iterations

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j-2] = a[i][j] + 1
```

Later iterations of both i and j loops depend on earlier iterations

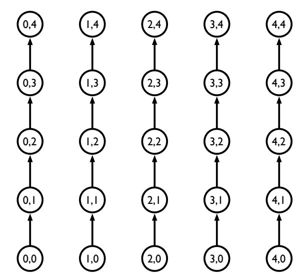


# Some subtleties

- Dependences might only be carried over one loop!

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i][j+1] = a[i][j] + 1
```

- Can parallelize i loop, but not j loop



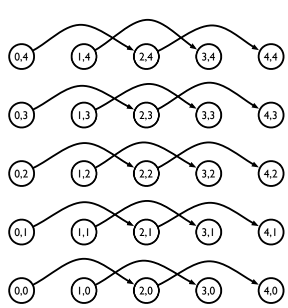



# Some subtleties

- Dependences might only be carried over one loop!

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j] = a[i-1][j] + 1
```

- Can parallelize j loop, but not i loop






# Direction vectors

- So how do direction vectors help?
  - If there is a non-zero entry for a loop dimension, that means that there is a loop carried dependence over that dimension
  - If an entry is zero, then that loop can be parallelized!
- May be able to parallelize inner loop even if entry is not zero, but you have to carefully structure parallel execution

# Data Dependence Tests



# Problem formulation

- Given the loop nest:

```
for (i = 0; i < N; i++)
  a[f(i)] = ...
  ... = a[g(i)]
```

- A dependence exists if there exist an *integer* i and an i' such that:
  - f(i) = g(i')
  - $0 \leq i, i' < N$
  - If i < i', write happens before read (flow dependence)
  - If i > i', write happens after read (anti dependence)



# Loop normalization

- Loops that skip iterations can always be *normalized* to loops that don't, so we only need to consider loops that have unit strides
  - Note: this is essentially of the reverse of linear test replacement

```
for (i = L; i < U; i += S)
  ... a[i] ...
```

$\downarrow$

```
for (i = 0; i < (U - L)/S; i += 1)
  ... a[S*i + L] ...
```



# Diophantine equations

- An equation whose coefficients and solutions are all integers is called a *Diophantine equation*
- Our question:

  f(i) = a*i + b      g(i) = c*i + d

  Does f(i) = g(i') have a solution?
- $f(i) = g(i') \Rightarrow ai + b = ci' + d \Rightarrow a_1*i + a_2*i' = a_3$



# Solutions to Diophantine eqns

- An equation $a_1*i + a_2*i' = a_3$ has a solution *iff* $gcd(a_1, a_2)$ evenly divides $a_3$
- Examples
  - 15*i + 6*j - 9*k = 12 has a solution (gcd = 3)
  - 2*i + 7*j = 3 has a solution (gcd = 1)
  - 9*i + 6*j = 10 has no solution (gcd = 3)



# Why does this work?

- Suppose g is the gcd(a, b) in a*i + b*j = c
- Can rewrite equation as

  g*(a'*i + b'*j) = c

  a' * i + b' * j = c/g
- a' and b' are integers, and relatively prime (gcd = 1) so by choosing i and j correctly, can produce *any* integer, but *only* integers
- Equation has a solution provided c/g is an integer

## Finding the GCD

- Finding GCD with Euclid's algorithm
  - Repeat

    a = a mod b

    swap a and b

    until b is 0 (resulting a is the gcd)
- Why? If g divides a and b, then g divides a mod b

gcd(27, 15): a = 27, b = 15
a = 27 mod 15 = 12
a = 15 mod 12 = 3
a = 12 mod 3 = 0
gcd = 3



## Downsides to GCD test

- If f(i) = g(i') *fails* the GCD test, then there is no i, i' that can produce a dependence → loop has no dependences
- If f(i) = g(i'), there *might* be a dependence, but might not
  - i and i' that satisfy equation might fall outside bounds
  - Loop may be parallelizable, but cannot tell
- Unfortunately, most loops have gcd(a, b) = 1, which divides everything
- Other optimizations (loop interchange) can tolerate dependences in certain situations



## Other dependence tests

- GCD test: doesn't account for loop bounds, does not provide useful information in many cases
- Banerjee test (Utpal Banerjee): accurate test, takes directions and loop bounds into account
- Omega test (William Pugh): even more accurate test, precise but can be very slow
- Range test (Blume and Eigenmann): works for non-linear subscripts
- Compilers tend to perform simple tests and only perform more complex tests if they cannot prove non-existence of dependence



# Other loop optimizations



## Loop interchange

- We've seen this one before
- Interchange doubly-nested loop to
  - Improve locality
  - Improve parallelism
    - Move parallel loop to outer loop (coarse grained parallelism)



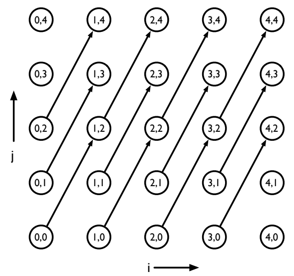
## Loop interchange legality

- We noted that loop interchange is not always legal, because it reorders a computation
- Can we use dependences to determine legality?

# Loop interchange dependences

- Consider interchanging the following loop, with the dependence graph to the right:

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j+2] = a[i][j] + 1
```



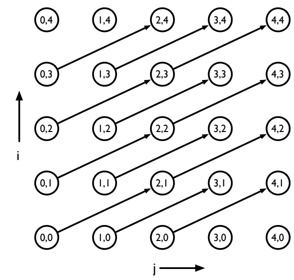

- Distance vector (1, 2)
- Direction vector (+, +)



# Loop interchange dependences

- Consider interchanging the following loop, with the dependence graph to the right:

```
for (j = 0; j < N; j++)
 for (i = 0; i < N; i++)
  a[i+1][j+2] = a[i][j] + 1
```




- Distance vector (2, 1)
- Direction vector (+, +)
- Distance vector gets swapped!



# Loop interchange legality

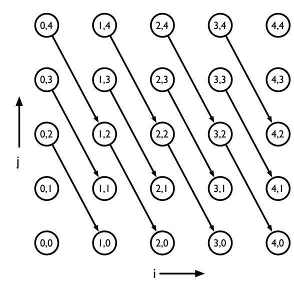
- Interchanging two loops swaps the order of their entries in distance/direction vectors
  - (0, +) → (+, 0)
  - (+, 0) → (0, +)
- But remember, we can't have backwards dependences
  - (+, –) → (–, +)
  - Illegal dependence → Loop interchange not legal!



# Loop interchange dependences

- Example of illegal interchange:

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j-2] = a[i][j] + 1
```

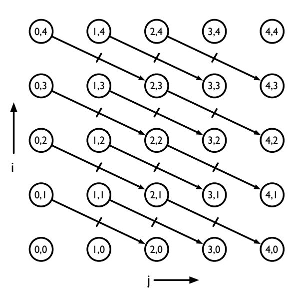





# Loop interchange dependences

- Example of illegal interchange:

```
for (j = 0; j < N; j++)
 for (i = 0; i < N; i++)
  a[i+1][j-2] = a[i][j] + 1
```




- Flow dependences turned into anti-dependences
  - Result of computation will change!



# Loop fusion/distribution

- Loop fusion: combining two loops into a single loop
  - Improves locality, parallelism
- Loop distribution: splitting a single loop into two loops
  - Can increase parallelism (turn a non-parallelizable loop into a parallelizable loop)
- Legal as long as optimization maintains dependences
  - Every dependence in the original loop should have a dependence in the optimized loop
  - Optimized loop should not introduce new dependences

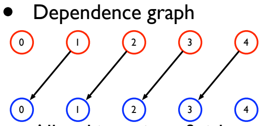
# Fusion/distribution example

- Code 1:

```
for (i = 0; i < N; i++)
  a[i - 1] = b[i]

for (j = 0; j < N; j++)
  c[j] = a[j]
```

- Dependence graph

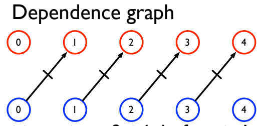



- All red iterations finish before blue iterations → flow dependence

- Code 2:

```
for (i = 0; i < N; i++)
  a[i - 1] = b[i]
  c[i] = a[i]
```

- Dependence graph




- i iterations finish before i+1 iterations → flow dependence now an anti dependence!



# Fusion/distribution utility

```
for (i = 0; i < N; i++)
  a[i] = a[i - 1]

for (j = 0; j < N; j++)
  b[j] = a[j]
```

Fusion →

← Distribution

```
for (i = 0; i < N; i++)
  a[i] = a[i - 1]
  b[i] = a[i]
```

- Fusion and distribution both legal
- Right code has better locality, but cannot be parallelized due to loop carried dependences
- Left code has worse locality, but blue loop can be parallelized