Compiler Optimisation

8 – Dependence Analysis

Hugh Leather
IF 1.18a
hleather@inf.ed.ac.uk

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2019

Introduction

This lecture:

- Parallelism
- Types of dependence flow, anti and output
- Distance and direction vectors
- Classification of loop based data dependences
- Dependence tests: gcd, Banerjee and Omega

References

- R. Allen and K Kennedy Optimizing compilers for modern architectures: A dependence based approach Morgan Kaufmann 2001. Main reference for this section of lecture notes (CMA)
- Michael Wolfe High Performance Compilers for Parallel Computing Addison – Wesley 1996.
- H. Zima and B. Chapman. Supercompilers for Parallel and Vector Computers. ACM Press Frontier Series 1990
- Today : The Omega Test: a fast and practical integer programming algorithm for dependence analysis Supercomputing 1992

Parallelism
Programming parallel computers

- Schools of thought:
  1. User specifies low-level parallelism and mapping
  2. User specifies parallelism (e.g. skeletons) - system tunes mapping
  3. Compiler finds parallelism in sequential code

- Popular approach is to break the transformation process into stages
- Transform to maximise parallelism i.e minimise critical path of program execution graph
- Map parallelism to minimise "significant" machine costs i.e. communication/ non-local access etc.

Parallelism

Different forms of parallelism

Statement parallelism

```
a = b + c
d = e + f
```

Operation parallelism

```
a = (b + c) * (e + f)
```

Function parallelism

```
f(a) = if (a <= 1)
  then return 0
  else return
f(a-1)+f(a-2)
```

Loop parallelism

```
for(i = 1 to n)
  A[i] = b[i] + c
```

Parallelism

Loop parallelism / Array parallelism

Original loop

```
for(i = 1 to n)
    A[i] = b[i] + c
```

Parallel loop

```
parfor(i = 1 to n)
    A[i] = b[i] + c
```

- All iterations of the iterator i can be performed independently
- Independence implies parallelism
- Loop parallelism $O(n)$ potential parallelism
  Compare statement and operation parallelism - $O(1)$.
- Recursive parallelism rich but dynamic. Exploited in functional computational models

Parallelism

Parallelism and data dependence

```
for(i = 1 to n)
    A[i] = b[i] + c
```

Each iteration independent – completely parallel

```
for(i = 1 to n)
    A[i+1] = A[i] + c
```

Each iteration dependent on previous – completely serial

Note: iterations NOT array elements

Need to apply transformations and know when it is safe to do so

Reordering transformation

A reordering transformation is any program transformation that only changes the execution order of statements without adding or deleting statements

A reordering transformation that preserves every dependence, preserves the meaning of the program

Parallelising loop iterations allows random interleaving (reordering) of statements in loop body

Data dependence
Types of data dependence

Relationship between reads and writes to memory has critical impact on parallelism

3 types of data dependence

Flow (True)
RAW hazard

S_1: a =
S_2: = a

Denoted $S_2\ \delta\ S_1$

Anti
WAR hazard

S_1: = a
S_2: a =

Denoted $S_2\ \delta^{-1}\ S_1$

Output
WAW hazard

S_1: a =
S_2: a =

Denoted $S_2\ \delta^{0}\ S_1$

Only data flow dependences are true dependences. Anti and output can be removed by renaming

Dependence

- Data-flow analysis can be used to define data dependences on a per block level for scalars but fails in presence of arrays
- Need finer grained analysis – determine if statements' array usage access same memory location and type of dependence

Dependence
Iteration vectors

Consider two loops:

```
   | for(i = 1 to n)
S  |    A[i+1] = A[i] + b[i]
```

```
   | for(i = 1 to n)
S  |    A[i+2] = A[i] + b[i]
```

- In both cases, statement S depends on itself
- However, there is a significant difference
- Need formalism to describe and distinguish such dependences

Dependence
Iteration vectors

Iteration number

Each iteration in a loop has an **iteration number** which is the value of the loop index at that iteration

Normalised iteration number

For iteration number i in loop with bounds L, U, and stride S, the **normalised iteration number** is[a]

$$(I - L + S)/S$$

Convenient to normalise

[a] This definition is one-based

Dependence
Iteration vectors

Iteration vectors extend this notion to loop nests

Iteration vector

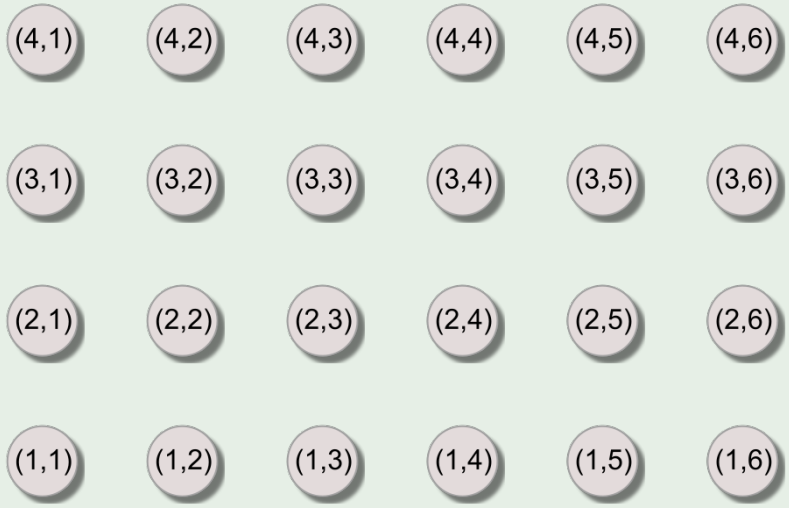
Iteration vector $\mathcal{I}$ of iteration is the vector of integers containing iteration numbers for loops in order of nesting level

```
    | for(i = 1 to 4)
    |   for(j = 1 to 6)
 S  |     some-statement
```

Iteration vector, $(2, 1)$ of S is when $i = 2$ and $j = 1$

Dependence
Iteration vectors

Iteration vectors for simple loop




```
     | for(i = 1 to 4)
     |   for(j = 1 to 6)
S    |     some-statement
```

Dependence
Iteration vector ordering

Iteration vectors ordered by execution order

For normalised vectors this is lexicographical ordering

Lexicographical ordering

For two iteration vectors, $\mathcal{I}$ and $\mathcal{J}$,
$\mathcal{I} < \mathcal{J}$ iff

1. $\mathcal{I}[1 : n-1] < \mathcal{J}[1 : n-1]$, or
2. $\mathcal{I}[1 : n-1] = \mathcal{J}[1 : n-1]$ and $\mathcal{I}_n < \mathcal{J}_n$

I.e. compare $<$ by first element, if $=$ compare $<$ next element, etc.

Why normalised?

Dependence
Iteration vector ordering

Iteration vectors ordered by execution order
For normalised vectors this is lexicographical ordering

Lexicographical ordering

For two iteration vectors, $\mathcal{I}$ and $\mathcal{J}$,
$\mathcal{I} < \mathcal{J}$ iff

1. $\mathcal{I}[1 : n-1] < \mathcal{J}[1 : n-1]$, or
2. $\mathcal{I}[1 : n-1] = \mathcal{J}[1 : n-1]$ and $\mathcal{I}_n < \mathcal{J}_n$

I.e. compare $<$ by first element, if $=$ compare $<$ next element, etc.

Why normalised?
Consider induction variable going backwards

Dependence
Iteration vector ordering

Lexicographical ordering

S

```
for(i = 1 to 4)
  for(j = 1 to 6)
    some-statement
```

Dependence

Dependencies between iterations

Loop-independent dependency

If statement S_2 depends on S_1 and S_1, S_2 execute in same iteration

Loop-carried dependency

If statement S_2 depends on S_1 and S_1, S_2 execute in different iterations

Dependence
Dependencies between iterations

Dependence distance

If dependence is between iterations $\mathcal{I}_{write}$ and $\mathcal{I}_{read}$, then distance is $\mathcal{I}_{read} - \mathcal{I}_{write}$

Distance example

Write `A[10,11]` at iteration $(5, 5)$. Read `A[10,11]` at $(5, 6)$. Distance is?

Dependence
Dependencies between iterations

Dependence distance

If dependence is between iterations $\mathcal{I}_{write}$ and $\mathcal{I}_{read}$,
then distance is $\mathcal{I}_{read} - \mathcal{I}_{write}$

Distance example

Write `A[10,11]` at iteration $(5, 5)$. Read `A[10,11]` at $(5, 6)$.
Distance is $(5, 6) - (5, 5) = (0, 1)$

Dependence

Dependencies between iterations

- If dependence distances all same, then say loop has that dependence distance
- But, loop may have many different dependence distances
- Direction vector summarises directions
- If first non '=' element is '<' then indicates flow dependence[1]

Dependence direction

Direction vector summary of distance dimensions
i.e. per dimension

- $<$ All +ve
- $>$ All -ve
- $=$ All 0
- $*$ Mixed

Direction example

Given distances:

```
(0, 1,-1,-1)
(0, 2,-2, 0)
(0, 3,-3, 1)
```

Direction is: ?

[1] (Why?)

Dependence

Dependencies between iterations

- If dependence distances all same, then say loop has that dependence distance
- But, loop may have many different dependence distances
- Direction vector summarises directions
- If first non '=' element is '<' then indicates flow dependence[1]

Dependence direction

Direction vector summary of distance dimensions
i.e. per dimension

- $<$ All +ve
- $>$ All -ve
- $=$ All 0
- $*$ Mixed

Direction example

Given distances:

```
(0, 1,-1,-1)
(0, 2,-2, 0)
(0, 3,-3, 1)
```

Direction is:

```
(=, <, >, *)
```

[1] (Why?)

Dependence

Dependencies between iterations

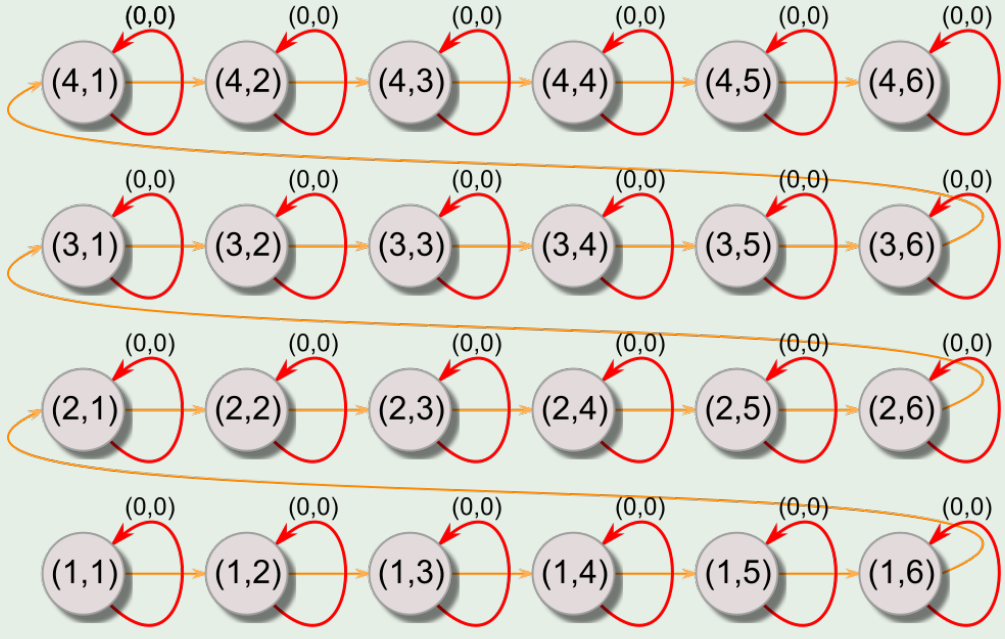
Where are dependences, distances, directions here?


```
      | for(i = 1 to 4)
      |   for(j = 1 to 6)
S2    |     A[ i, j ] =
S2    |               = A[ i, j ] + 1
```

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?



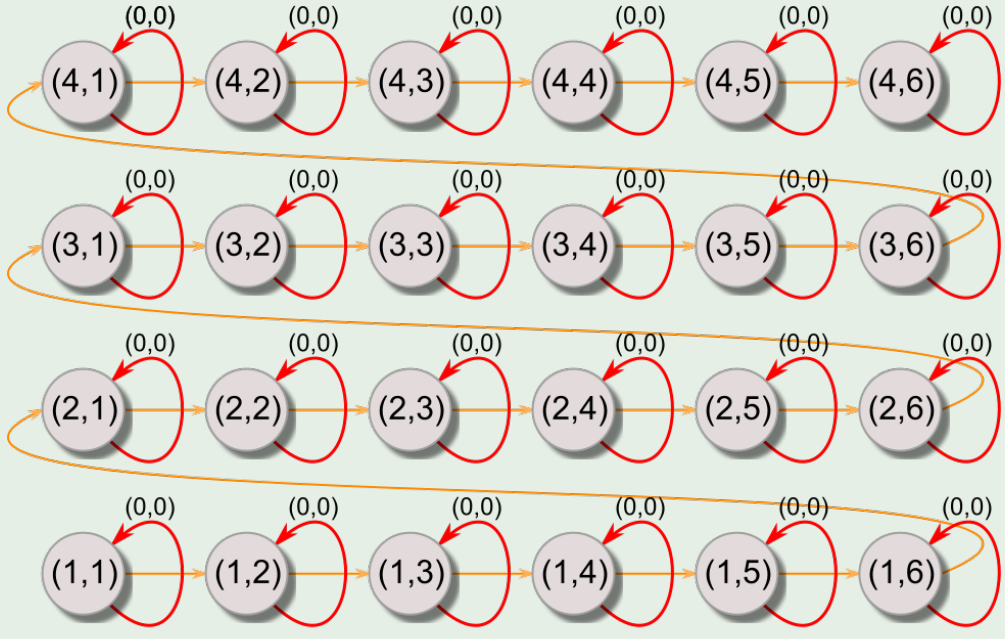
Direction
(=,=)

```
      | for(i = 1 to 4)
      |   for(j = 1 to 6)
S2    |     A[ i, j ] =
S2    |               = A[ i, j ] + 1
```

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?



Direction
(=,=)

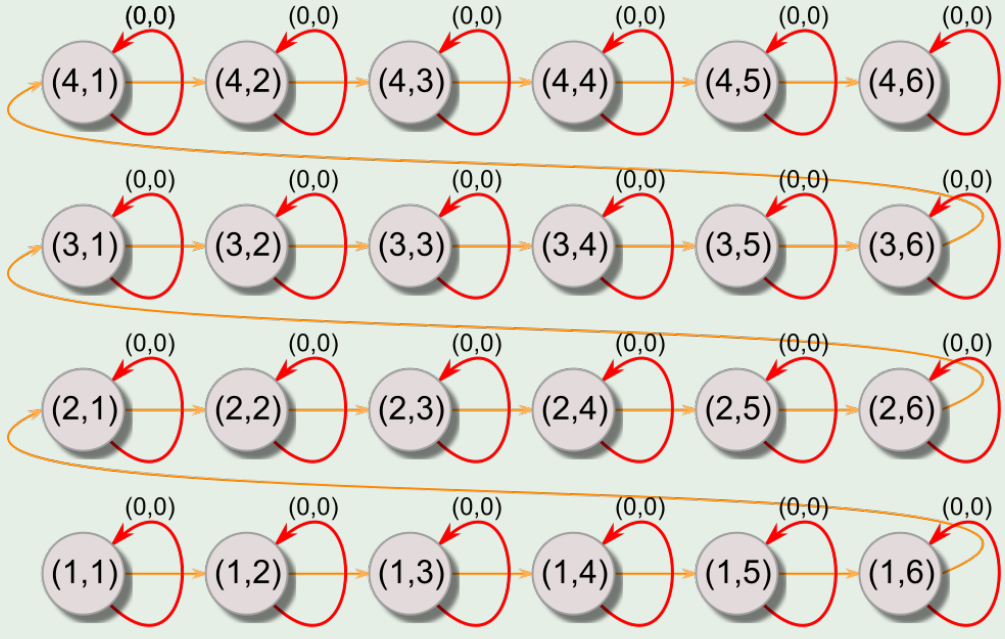
```
S | for(i = 1 to 4)
  |   for(j = 1 to 6)
  |     A[ i, j ] = A[ i, j ] + 1
```

Statement can depend on itself

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?



Direction
(=,=)

S

```
for(i = 1 to 4)
  for(j = 1 to 6)
    A[ j+10, i ] = A[ j+10, i ] + 1
```

Iterations, not array elements!

Dependence

Dependencies between iterations

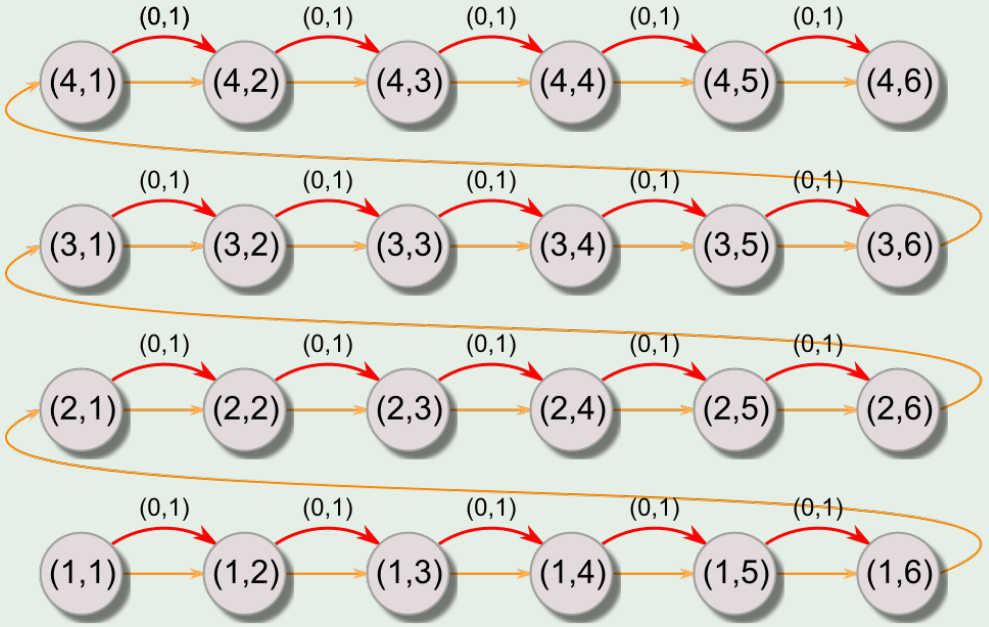
Where are dependences, distances, directions here?


```
   | for(i = 1 to 4)
   |   for(j = 1 to 6)
S  |     A[ i, j ] = A[ i, j - 1 ] + 1
```

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?



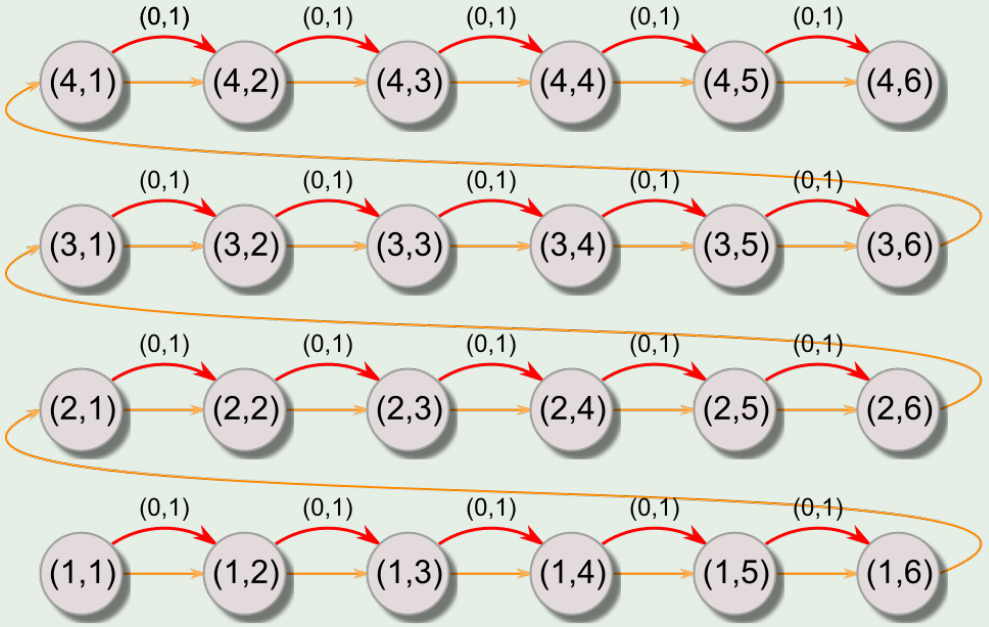
Direction
(=,<)

```
S | for(i = 1 to 4)
  |   for(j = 1 to 6)
  |     A[ i, j ] = A[ i, j - 1 ] + 1
```

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?



Direction
(=,<)

S

```
for(i = 1 to 4)
  for(j = 1 to 6)
    A[ i, j + 1 ] = A[ i, j ] + 1
```

Clearly the same thing

Dependence

Dependencies between iterations

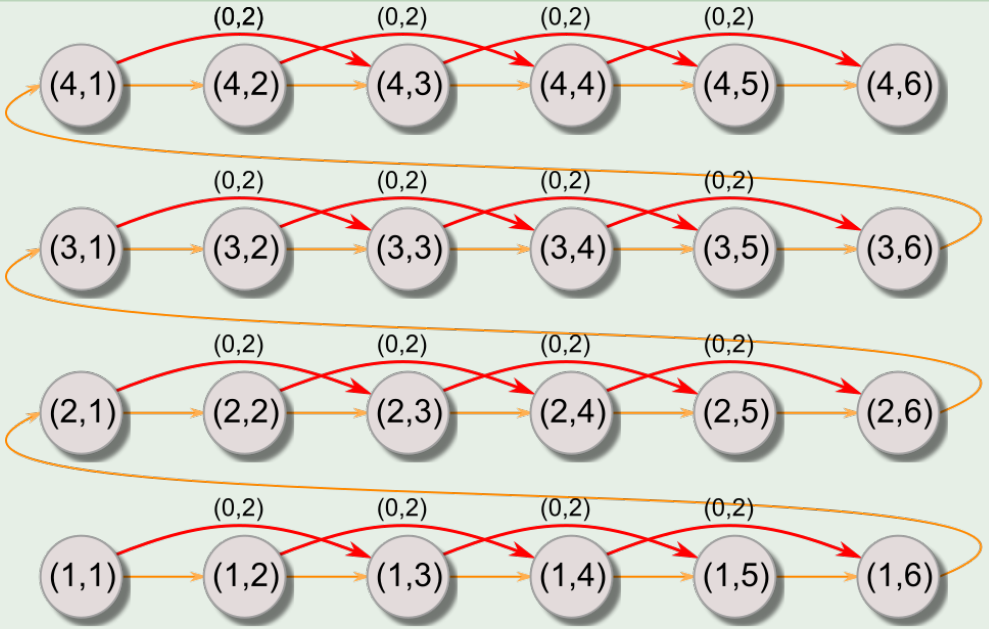
Where are dependences, distances, directions here?


```
   | for(i = 1 to 4)
   |   for(j = 1 to 6)
S  |     A[ i, j ] = A[ i, j - 2 ] + 1
```

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?



Direction
(=,<)

```
   | for(i = 1 to 4)
   |   for(j = 1 to 6)
S  |     A[ i, j ] = A[ i, j - 2 ] + 1
```

Dependence

Dependencies between iterations

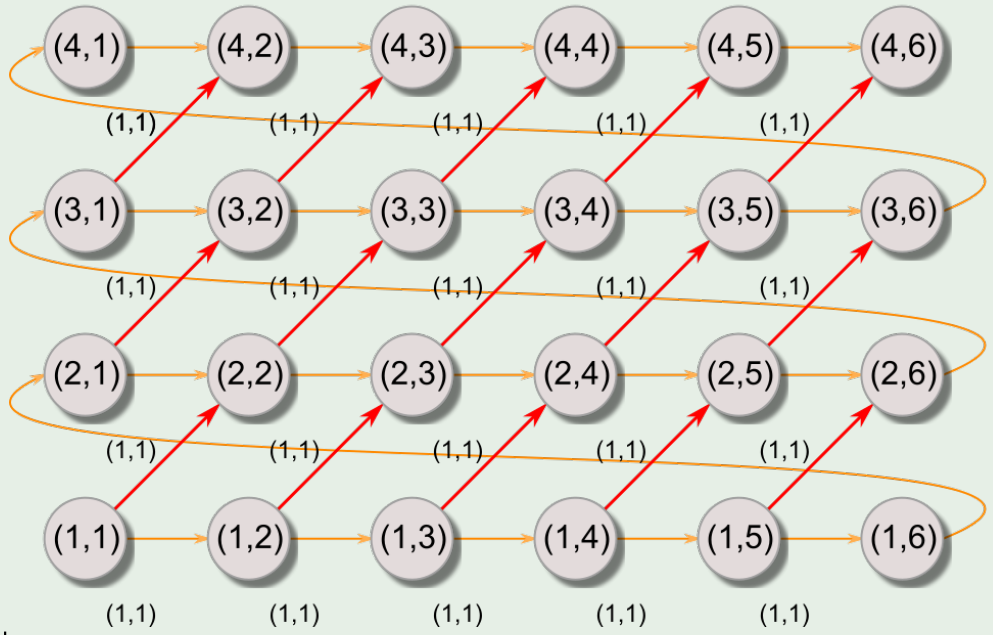
Where are dependences, distances, directions here?


```
   | for(i = 1 to 4)
   |   for(j = 1 to 6)
S  |     A[ i, j ] = A[ i - 1, j - 1 ] + 1
```

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?



Direction
(<,<)

```
S | for(i = 1 to 4)
  |   for(j = 1 to 6)
  |     A[ i, j ] = A[ i - 1, j - 1 ] + 1
```

Dependence

Dependencies between iterations

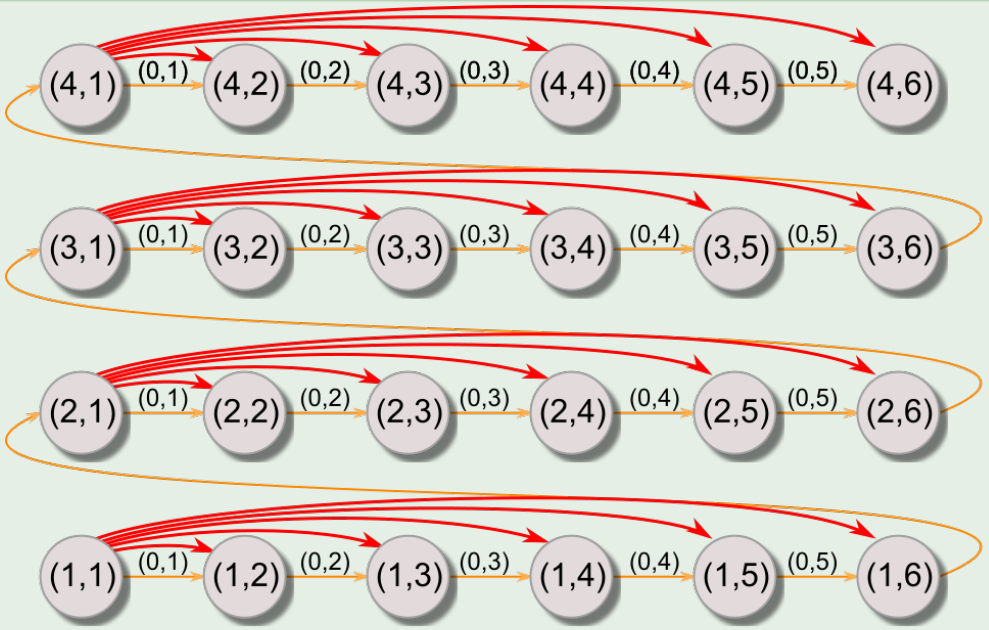
Where are dependences, distances, directions here?


```
   | for(i = 1 to 4)
   |   for(j = 1 to 6)
 S |     A[ i, j ] = A[ i, 1 ] + j
```

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?



Direction
(=,<)

```
   | for(i = 1 to 4)
   |   for(j = 1 to 6)
S  |     A[ i, j ] = A[ i, 1 ] + j
```

Dependence

Dependencies between iterations

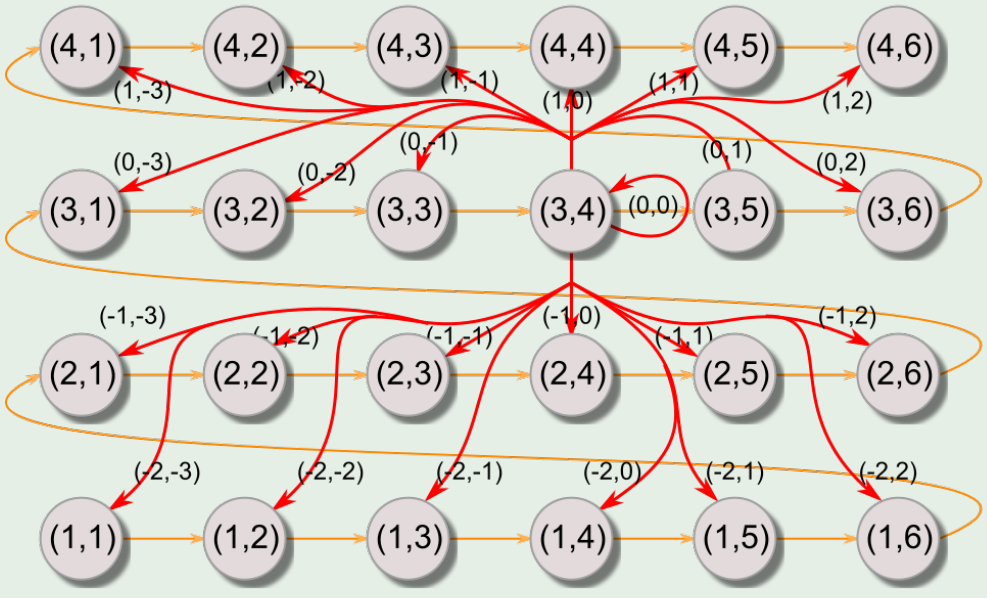
Where are dependences, distances, directions here?


```
   | for(i = 1 to 4)
   |   for(j = 1 to 6)
S  |     A[ i, j ] = A[ 3, 4 ] + i * j
```

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?



Direction
(*,*)

S

```
for(i = 1 to 4)
  for(j = 1 to 6)
    A[ i, j ] = A[ 3, 4 ] + i * j
```

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?


```
   | for(i = 1 to 4)
   |   for(j = 1 to 6)
S  |     A[ i, j ] = A[ 0, 0 ]
```

Dependence

Dependencies between iterations

Where are dependences, distances, directions here?


```
   | for(i = 1 to 4)
   |   for(j = 1 to 6)
S  |     A[ i, j ] = A[ 0, 0 ]
```

There are none!

Dependence
Solving the dependence problem

- Question: is there dependence between array write in S_1 and read in S_2?

[2] After *Diophantus of Alexandria* c. 210AD

[3] This is Hilberts tenth problem – set in 1900, proven in 1970

[4] Consider $n \geq 2, \forall a, b, c > 0; a^n + b^n - c^n \neq 0$ *(Fermat's last theorem)*

[5] Integer linear programming is NP-complete

Dependence
Solving the dependence problem

- Question: is there dependence between array write in S_1 and read in S_2?
- Assume write in iteration $\mathcal{I}_w$, read in $\mathcal{I}_r$

[2] After *Diophantus of Alexandria* c. 210AD

[3] This is Hilberts tenth problem – set in 1900, proven in 1970

[4] Consider $n \geq 2, \forall a, b, c > 0; a^n + b^n - c^n \neq 0$ *(Fermat's last theorem)*

[5] Integer linear programming is NP-complete

Dependence
Solving the dependence problem

- Question: is there dependence between array write in S_1 and read in S_2?
- Assume write in iteration $\mathcal{I}_w$, read in $\mathcal{I}_r$
- Assume write of $A[f_w(\mathcal{I}_w)]$, read of $A[f_r(\mathcal{I}_r)]$, with f_w and f_r as polynomials

[2] After *Diophantus of Alexandria* c. 210AD

[3] This is Hilberts tenth problem – set in 1900, proven in 1970

[4] Consider $n \geq 2, \forall a, b, c > 0; a^n + b^n - c^n \neq 0$ *(Fermat's last theorem)*

[5] Integer linear programming is NP-complete

Dependence
Solving the dependence problem

- Question: is there dependence between array write in S_1 and read in S_2?
- Assume write in iteration $\mathcal{I}_w$, read in $\mathcal{I}_r$
- Assume write of $A[f_w(\mathcal{I}_w)]$, read of $A[f_r(\mathcal{I}_r)]$, with f_w and f_r as polynomials
- Solve $f_w(\mathcal{I}_w) - f_r(\mathcal{I}_r) = 0$ for integer solutions (inside iteration space)

[2] After *Diophantus of Alexandria* c. 210AD

[3] This is Hilberts tenth problem – set in 1900, proven in 1970

[4] Consider $n \geq 2, \forall a, b, c > 0; a^n + b^n - c^n \neq 0$ *(Fermat's last theorem)*

[5] Integer linear programming is NP-complete

Dependence
Solving the dependence problem

- Question: is there dependence between array write in S_1 and read in S_2?
- Assume write in iteration $\mathcal{I}_w$, read in $\mathcal{I}_r$
- Assume write of $A[f_w(\mathcal{I}_w)]$, read of $A[f_r(\mathcal{I}_r)]$, with f_w and f_r as polynomials
- Solve $f_w(\mathcal{I}_w) - f_r(\mathcal{I}_r) = 0$ for integer solutions (inside iteration space)
- This is diophantine equation[2]

[2] After *Diophantus of Alexandria* c. 210AD
[3] This is Hilberts tenth problem – set in 1900, proven in 1970
[4] Consider $n \geq 2, \forall a, b, c > 0; a^n + b^n - c^n \neq 0$ *(Fermat's last theorem)*
[5] Integer linear programming is NP-complete

Dependence
Solving the dependence problem

- Question: is there dependence between array write in S_1 and read in S_2?
- Assume write in iteration $\mathcal{I}_w$, read in $\mathcal{I}_r$
- Assume write of $A[f_w(\mathcal{I}_w)]$, read of $A[f_r(\mathcal{I}_r)]$, with f_w and f_r as polynomials
- Solve $f_w(\mathcal{I}_w) - f_r(\mathcal{I}_r) = 0$ for integer solutions (inside iteration space)
- This is diophantine equation[2]
- Undecidable in general[3,4]

[2] After *Diophantus of Alexandria* c. 210AD

[3] This is Hilberts tenth problem – set in 1900, proven in 1970

[4] Consider $n \geq 2, \forall a, b, c > 0; a^n + b^n - c^n \neq 0$ *(Fermat's last theorem)*

[5] Integer linear programming is NP-complete

Dependence
Solving the dependence problem

- Question: is there dependence between array write in S_1 and read in S_2?
- Assume write in iteration $\mathcal{I}_w$, read in $\mathcal{I}_r$
- Assume write of $A[f_w(\mathcal{I}_w)]$, read of $A[f_r(\mathcal{I}_r)]$, with f_w and f_r as polynomials
- Solve $f_w(\mathcal{I}_w) - f_r(\mathcal{I}_r) = 0$ for integer solutions (inside iteration space)
- This is diophantine equation[2]
- Undecidable in general[3,4]
- Limit to linear diophantine equations with constraints[5]

  $a_n x_1 + a_{n-1} x_{n-1} + \ldots + a_1 x_1 + a_0 = 0$

[2] After *Diophantus of Alexandria* c. 210AD

[3] This is Hilberts tenth problem – set in 1900, proven in 1970

[4] Consider $n \geq 2, \forall a, b, c > 0; a^n + b^n - c^n \neq 0$ *(Fermat's last theorem)*

[5] Integer linear programming is NP-complete

Dependence

Solving the dependence problem

Example

```
for(i = 1 to 100)
  for(j = i to 100)
    A[i, j+1] = A[i, j]
```

Let $\mathcal{I}_w = (i_w, j_w)$ and $\mathcal{I}_r = (i_r, j_r)$

Let $f_w(i_w, j_w) =$

Dependence

Solving the dependence problem

Example

```
for(i = 1 to 100)
  for(j = i to 100)
    A[i, j+1] = A[i, j]
```

Let $\mathcal{I}_w = (i_w, j_w)$ and $\mathcal{I}_r = (i_r, j_r)$

Let $f_w(i_w, j_w) = (i_w, j_w + 1)$ and $f_r(i_r, j_r) =$

Dependence

Solving the dependence problem

Example

```
for(i = 1 to 100)
  for(j = i to 100)
    A[i, j+1] = A[i, j]
```

Let $\mathcal{I}_w = (i_w, j_w)$ and $\mathcal{I}_r = (i_r, j_r)$

Let $f_w(i_w, j_w) = (i_w, j_w + 1)$ and $f_r(i_r, j_r) = (i_r, j_r)$

First constrain induction variables

Dependence

Solving the dependence problem

Example

```
for(i = 1 to 100)
  for(j = i to 100)
    A[i, j+1] = A[i, j]
```

Let $\mathcal{I}_w = (i_w, j_w)$ and $\mathcal{I}_r = (i_r, j_r)$

Let $f_w(i_w, j_w) = (i_w, j_w + 1)$ and $f_r(i_r, j_r) = (i_r, j_r)$

First constrain induction variables

$$1 \leq i_w \leq 100,$$

Dependence

Solving the dependence problem

Example

```
for(i = 1 to 100)
  for(j = i to 100)
    A[i, j+1] = A[i, j]
```

Let $\mathcal{I}_w = (i_w, j_w)$ and $\mathcal{I}_r = (i_r, j_r)$

Let $f_w(i_w, j_w) = (i_w, j_w + 1)$ and $f_r(i_r, j_r) = (i_r, j_r)$

First constrain induction variables

$$1 \leq i_w \leq 100,\ i_w \leq j_w \leq 100,$$

Dependence

Solving the dependence problem

Example

```
for(i = 1 to 100)
  for(j = i to 100)
    A[i, j+1] = A[i, j]
```

Let $\mathcal{I}_w = (i_w, j_w)$ and $\mathcal{I}_r = (i_r, j_r)$

Let $f_w(i_w, j_w) = (i_w, j_w + 1)$ and $f_r(i_r, j_r) = (i_r, j_r)$

First constrain induction variables

$$1 \leq i_w \leq 100,\ i_w \leq j_w \leq 100,$$
$$1 \leq i_r \leq 100,\ i_r \leq j_r \leq 100$$

Dependence

Solving the dependence problem

Example

```
for(i = 1 to 100)
  for(j = i to 100)
    A[i, j+1] = A[i, j]
```

Let $\mathcal{I}_w = (i_w, j_w)$ and $\mathcal{I}_r = (i_r, j_r)$

Let $f_w(i_w, j_w) = (i_w, j_w + 1)$ and $f_r(i_r, j_r) = (i_r, j_r)$

First constrain induction variables

$$1 \leq i_w \leq 100,\ i_w \leq j_w \leq 100,$$
$$1 \leq i_r \leq 100,\ i_r \leq j_r \leq 100$$

Constraints so $f_w(i_w, j_w) = f_r(i_r, i_r)$

Dependence

Solving the dependence problem

Example

```
for(i = 1 to 100)
  for(j = i to 100)
    A[i, j+1] = A[i, j]
```

Let $\mathcal{I}_w = (i_w, j_w)$ and $\mathcal{I}_r = (i_r, j_r)$

Let $f_w(i_w, j_w) = (i_w, j_w + 1)$ and $f_r(i_r, j_r) = (i_r, j_r)$

First constrain induction variables

$$1 \leq i_w \leq 100,\ i_w \leq j_w \leq 100,$$
$$1 \leq i_r \leq 100,\ i_r \leq j_r \leq 100$$

Constraints so $f_w(i_w, j_w) = f_r(i_r, i_r)$

$$i_w = i_r, j_w + 1 = j_r$$

Dependence

Solving the dependence problem

Example

```
for(i = 1 to 100)
  for(j = i to 100)
    A[i, j+1] = A[i, j]
```

Let $\mathcal{I}_w = (i_w, j_w)$ and $\mathcal{I}_r = (i_r, j_r)$

Let $f_w(i_w, j_w) = (i_w, j_w + 1)$ and $f_r(i_r, j_r) = (i_r, j_r)$

First constrain induction variables

$$1 \leq i_w \leq 100,\ i_w \leq j_w \leq 100,$$
$$1 \leq i_r \leq 100,\ i_r \leq j_r \leq 100$$

Constraints so $f_w(i_w, j_w) = f_r(i_r, i_r)$

$$i_w = i_r, j_w + 1 = j_r$$

Are there integer solutions for i_w, j_w, i_r, j_r?

Dependence

Solving the dependence problem

Example

```
for(i = 1 to 100)
  for(j = i to 100)
    A[i, j+1] = A[i, j]
```

Let $\mathcal{I}_w = (i_w, j_w)$ and $\mathcal{I}_r = (i_r, j_r)$

Let $f_w(i_w, j_w) = (i_w, j_w + 1)$ and $f_r(i_r, j_r) = (i_r, j_r)$

First constrain induction variables

$$1 \leq i_w \leq 100,\ i_w \leq j_w \leq 100,$$
$$1 \leq i_r \leq 100,\ i_r \leq j_r \leq 100$$

Constraints so $f_w(i_w, j_w) = f_r(i_r, i_r)$

$$i_w = i_r, j_w + 1 = j_r$$

Are there integer solutions for i_w, j_w, i_r, j_r?Yes, lots:

$$1 \leq i_w = i_r \leq j_w = (j_r - 1) \leq 99$$

Dependence

Hierarchical computation of dependence directions in loops

```
for(i = 1 to n)
    A[f(i)] =
            = A[g(i)]
```

- Test for any dependence from iteration $\mathcal{I}_w$ to $\mathcal{I}_r$: $1 \leq \mathcal{I}_w \leq n, 1 \leq \mathcal{I}_r \leq n \wedge f(\mathcal{I}_w) = g(\mathcal{I}_r)$
- Use this test to test any direction [*]
- If solutions add additional constraints:

  $<$ direction : add $\mathcal{I}_w < \mathcal{I}_r$,

  $=$ direction : add $\mathcal{I}_w = \mathcal{I}_r$
- Extend for multi loops, [*, *] then [$<$, *], [$=$, *] etc - hierarchical testing
- If L is loop depth, requires $O(3^L))$ tests per array access pair!

Dependence
Conservative testing

- Full problem is NP-complete; use some quick and dirty tests
- Apply test
- If fail to accurately solve dependence, try more tests
- Still fail? Assume dependency
- Never dangerous, may be sub-optimal
- Works correctly for vast majority of code

Dependence

Classification for simplification : Kennedy approach

- Test for each subscript in turn, if any subscript has no dependence - then no solution

ZIV, SIV, MIV

- Subscript is pair of expressions at same dimension

  ZIV if it contains no index - e.g. $\langle 2, 10 \rangle$

  SIV if it contains only one index - e.g. $\langle i, i+2 \rangle$

  MIV if it contains more than one index - e.g. $\langle i+j, j \rangle$

Example classification

```
A[5, i+1, j] = A[10, i, k] + c
```

- Subscript in 1st dim

Dependence

Classification for simplification : Kennedy approach

- Test for each subscript in turn, if any subscript has no dependence - then no solution

ZIV, SIV, MIV

- Subscript is pair of expressions at same dimension

  ZIV if it contains no index - e.g. $\langle 2, 10 \rangle$
  SIV if it contains only one index - e.g. $\langle i, i+2 \rangle$
  MIV if it contains more than one index - e.g. $\langle i+j, j \rangle$

Example classification

```
A[5, i+1, j] = A[10, i, k] + c
```

- Subscript in 1st dim contains zero index variables (ZIV)
- Subscript in 2nd dim

Dependence

Classification for simplification : Kennedy approach

- Test for each subscript in turn, if any subscript has no dependence - then no solution

ZIV, SIV, MIV

- Subscript is pair of expressions at same dimension

  ZIV if it contains no index - e.g. $\langle 2, 10 \rangle$

  SIV if it contains only one index - e.g. $\langle i, i+2 \rangle$

  MIV if it contains more than one index - e.g. $\langle i+j, j \rangle$

Example classification

```
A[5, i+1, j] = A[10, i, k] + c
```

- Subscript in 1st dim contains zero index variables (ZIV)
- Subscript in 2nd dim contains single (i) index variables (SIV)
- Subscript in 3rd dim

Dependence

Classification for simplification : Kennedy approach

- Test for each subscript in turn, if any subscript has no dependence - then no solution

ZIV, SIV, MIV

- Subscript is pair of expressions at same dimension

  ZIV if it contains no index - e.g. $\langle 2, 10 \rangle$

  SIV if it contains only one index - e.g. $\langle i, i+2 \rangle$

  MIV if it contains more than one index - e.g. $\langle i+j, j \rangle$

Example classification

```
A[5, i+1, j] = A[10, i, k] + c
```

- Subscript in 1st dim contains zero index variables (ZIV)
- Subscript in 2nd dim contains single (i) index variables (SIV)
- Subscript in 3rd dim contains multi (j,k) index variables (MIV)

Dependence
Separability

Separability

- Indices in *separable* subscripts do not occur in other subscripts
- If two different subscripts contain same index they are *coupled*
- Separable subscripts and coupled groups handled independently

Example separability

```
a(i+1, j) = a(k, j) + c
```

First subscript is

Dependence
Separability

Separability

- Indices in *separable* subscripts do not occur in other subscripts
- If two different subscripts contain same index they are *coupled*
- Separable subscripts and coupled groups handled independently

Example separability

```
a(i+1, j) = a(k, j) + c
```

First subscript is separable.
Second subscript is

Dependence
Separability

Separability

- Indices in *separable* subscripts do not occur in other subscripts
- If two different subscripts contain same index they are *coupled*
- Separable subscripts and coupled groups handled independently

Example separability

```
a(i+1, j) = a(k, j) + c
```

First subscript is separable.
Second subscript is separable.

```
a(i, j, j) = a(i, j, k) + c
```

Second subscript is

Dependence
Separability

Separability

- Indices in *separable* subscripts do not occur in other subscripts
- If two different subscripts contain same index they are *coupled*
- Separable subscripts and coupled groups handled independently

Example separability

```
a(i+1, j) = a(k, j) + c
```

First subscript is separable.
Second subscript is separable.

```
a(i, j, j) = a(i, j, k) + c
```

Second subscript is coupled.
Third subscript is

Dependence
Separability

Separability

- Indices in *separable* subscripts do not occur in other subscripts
- If two different subscripts contain same index they are *coupled*
- Separable subscripts and coupled groups handled independently

Example separability

```
a(i+1, j) = a(k, j) + c
```

First subscript is separable.
Second subscript is separable.

```
a(i, j, j) = a(i, j, k) + c
```

Second subscript is coupled.
Third subscript is coupled.

Dependence
ZIV test

ZIV

A[..., c_w, ...] = A[..., c_r, ...] ...

If c_w and c_r are constants or loop invariant and $c_w \neq c_r$
No dependence

ZIV example

```
A[5, j+1, 10, k] = A[i, j, 12, k-1] + c
```

Third subscript has ZIV, and $10 \neq 12$
No dependence

Dependence
Strong SIV test

Strong SIV test

Constraint is: `A[ai+c_w] = A[ai+c_r] ...`

$$ai_w + c_w = ai_r + c_r \quad \Rightarrow$$

$$i_w - i_r = (c_w - c_r)/a$$

Has solution is $(c_w - c_r)/a$ is integer and is in range.

Strong SIV example

```
for(i = 1 to 10)
    A[i+1] = A[i] + c
```

Subscript is Strong SIV, $a = 1, c_w = 1, c_r = 0$

$$(c_w - c_r)/a = 1 \in [1, 10]$$

Dependence

Dependence

General SIV test or greatest common divisor

General SIV test

Constraint is: `A[a_w i+c_w] = A[a_r i+c_r] ...`

$$a_w i_w + c_w = a_r i_r + c_r$$

If $gcd(a_w, a_r)$ does not divides $c_w - c_r$ then no solution.
Else possibly many solutions.

General SIV example

```
for(i = 1 to 10)
    A[2i+1] = A[4i] + c
```

Subscript is Strong SIV, $a_w = 2, a_r = 4, c_w = 1, c_r = 0$

$$(c_w - c_r)/gcd(a_w, a_r) = 0.5$$

No dependence

Dependence
Banerjee test

- GCD test does not consider range - only if integer solution possible somewhere
- Banerjee test for existence of real valued solution in range
- If no real solution in range, then no integer one either

Banerjee test

$$\texttt{A[a}_w\texttt{i+c}_w\texttt{]} = \texttt{A[a}_r\texttt{i+c}_r\texttt{]} \ \ldots$$

Constraint is:

$$a_w i_w + c_w = a_r i_r + c_r \quad \Rightarrow$$

$$h(i_w, i_r) = a_w i_w - a_r i_r + c_w - c_r = 0$$

True by intermediate value theorem, if $max(h) \geq 0 \wedge min(h) \leq 0$

Banerjee test

```
for(i = 1 to 100)
    A[2*i+3] = A[i+7]
```

- We have $2i_w + 3 = i_r + 7, h = 2i_w i_r 4$ and $1 \leq i_w \leq i_r \leq 100$
- $max(h) = (2 * 100 - 1 - 4) = 195, min(h) = (2 * 1 - 100 - 4) = -102$
- $max(h) = 195 \geq 0 \geq min(h) = 102$ – Hence solution
- Simple example can be extended. Technical difficulties with complex iteration spaces
- Performed sub-script at a time, Used for MIV

Dependence
Pugh's Omega Test[6]

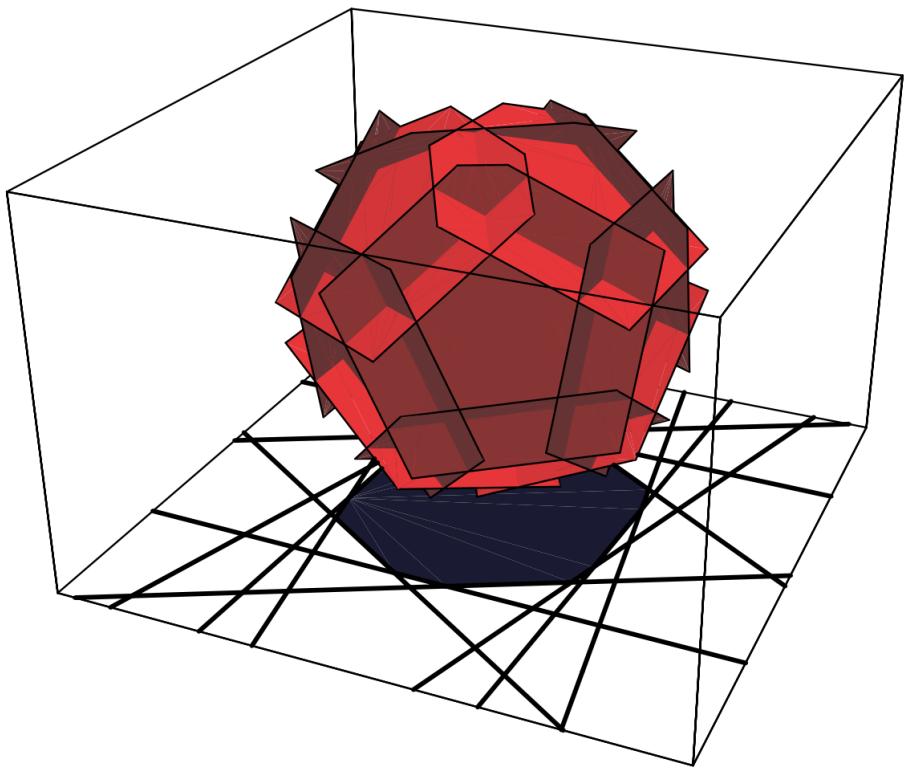
- Exact solutions using integer linear programming
- Fast enough for most real programs
  - Worst case exponential time
  - Commonly low order polynomial
  - Can directly yield direction and distance
- Algorithm
  - Express constraints
  - Simplify constraints
  - Do Fourier-Motzkin elimination

[6] Read the paper!

Dependence
Pugh's Omega Test

Geometric interpretation of constraints

- Constraints define n dimensional, revalued volume
- If empty no possible integer solutions
- Project volume to one fewer dimensions - giving real shadow
- If no integer points in shadow, no integer points in volume



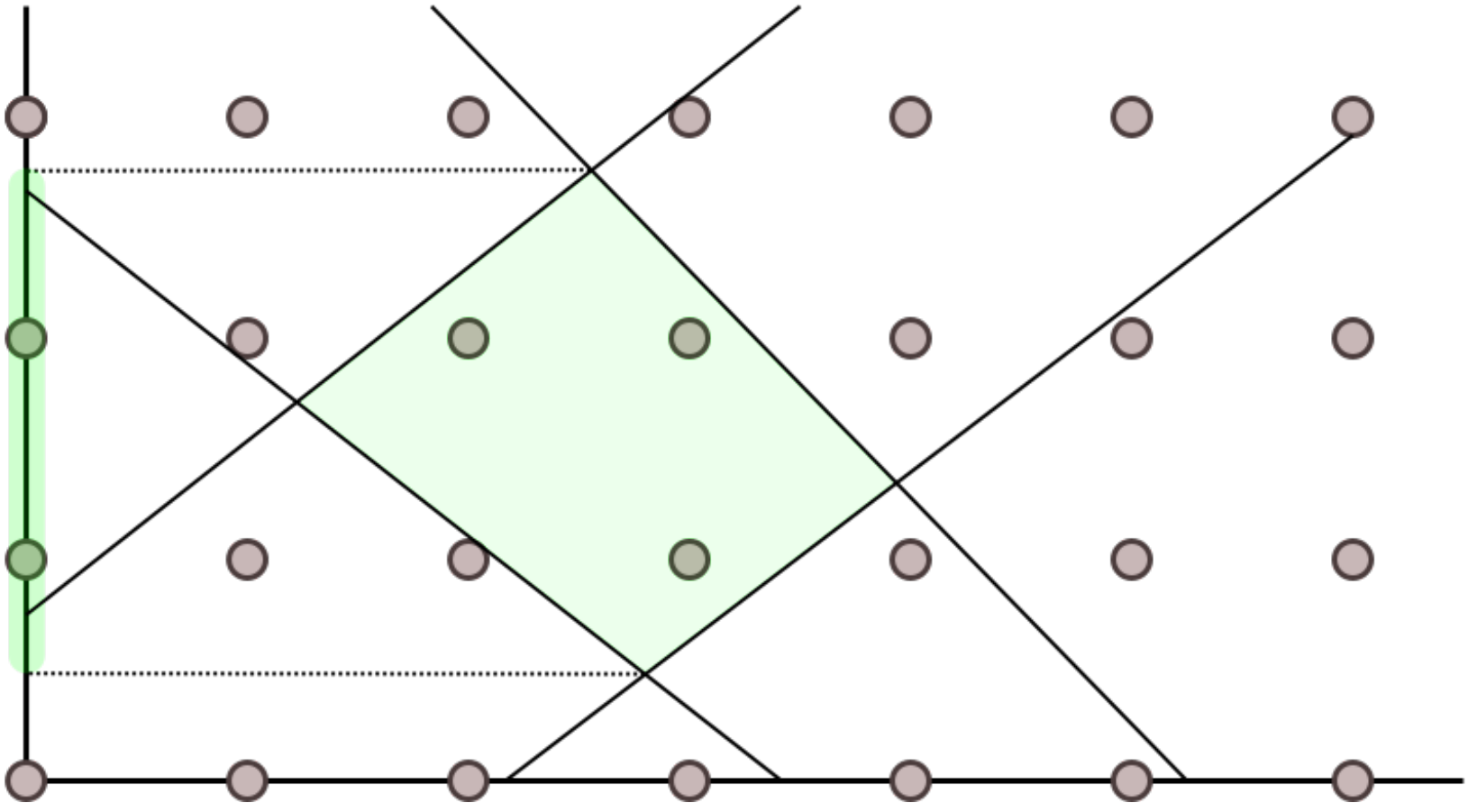
Dependence
Pugh's Omega Test

Integer points in real shadow do not imply integer points in volume

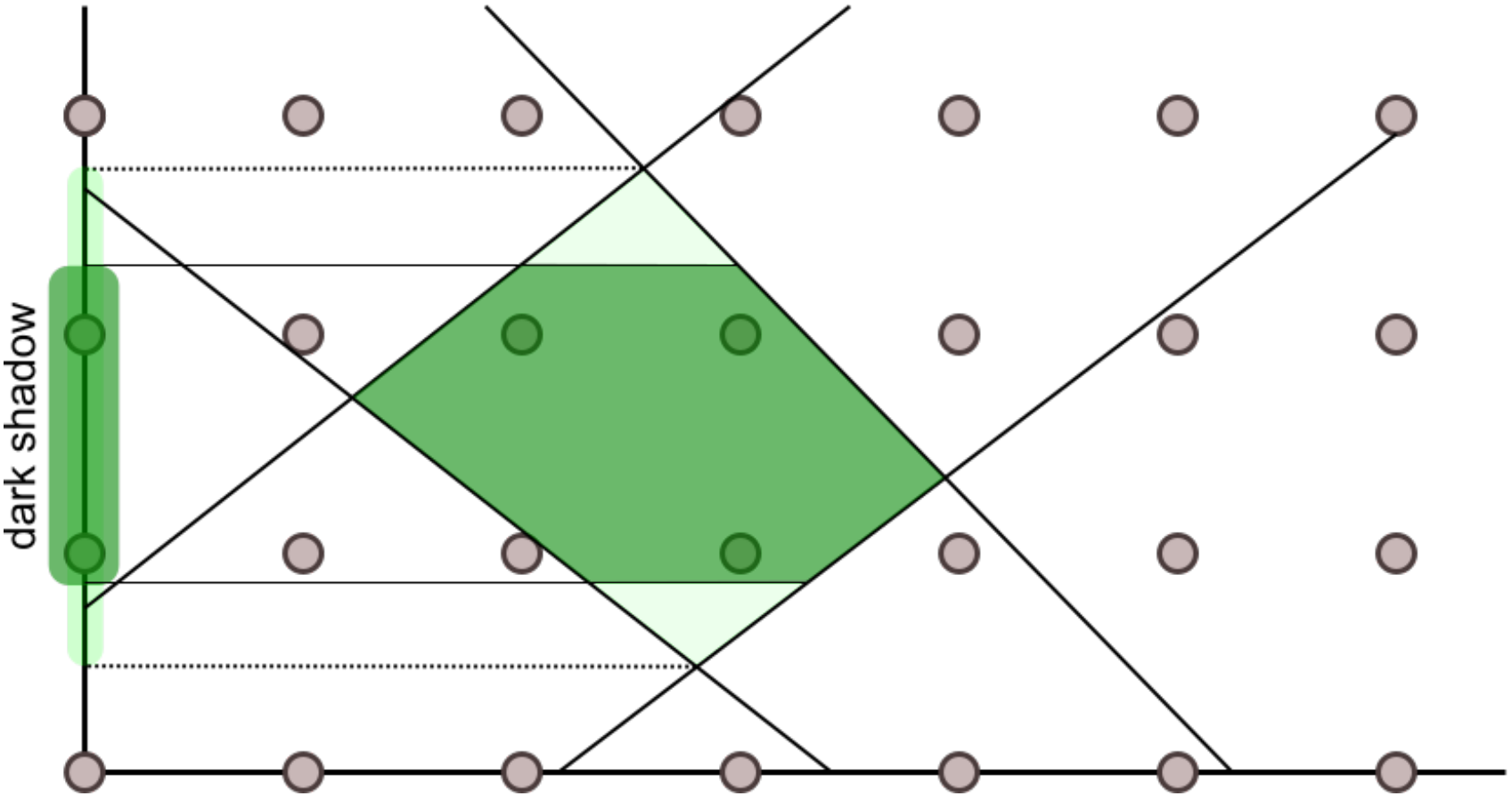
- Define *'dark shadow'* where if dark shadow contains integer point then real volume must
- E.g.[7] shadow wherever real volume is thicker than 1
- No integer points in dark shadow does not imply no integer points in real volume

[7] Read paper for what they do

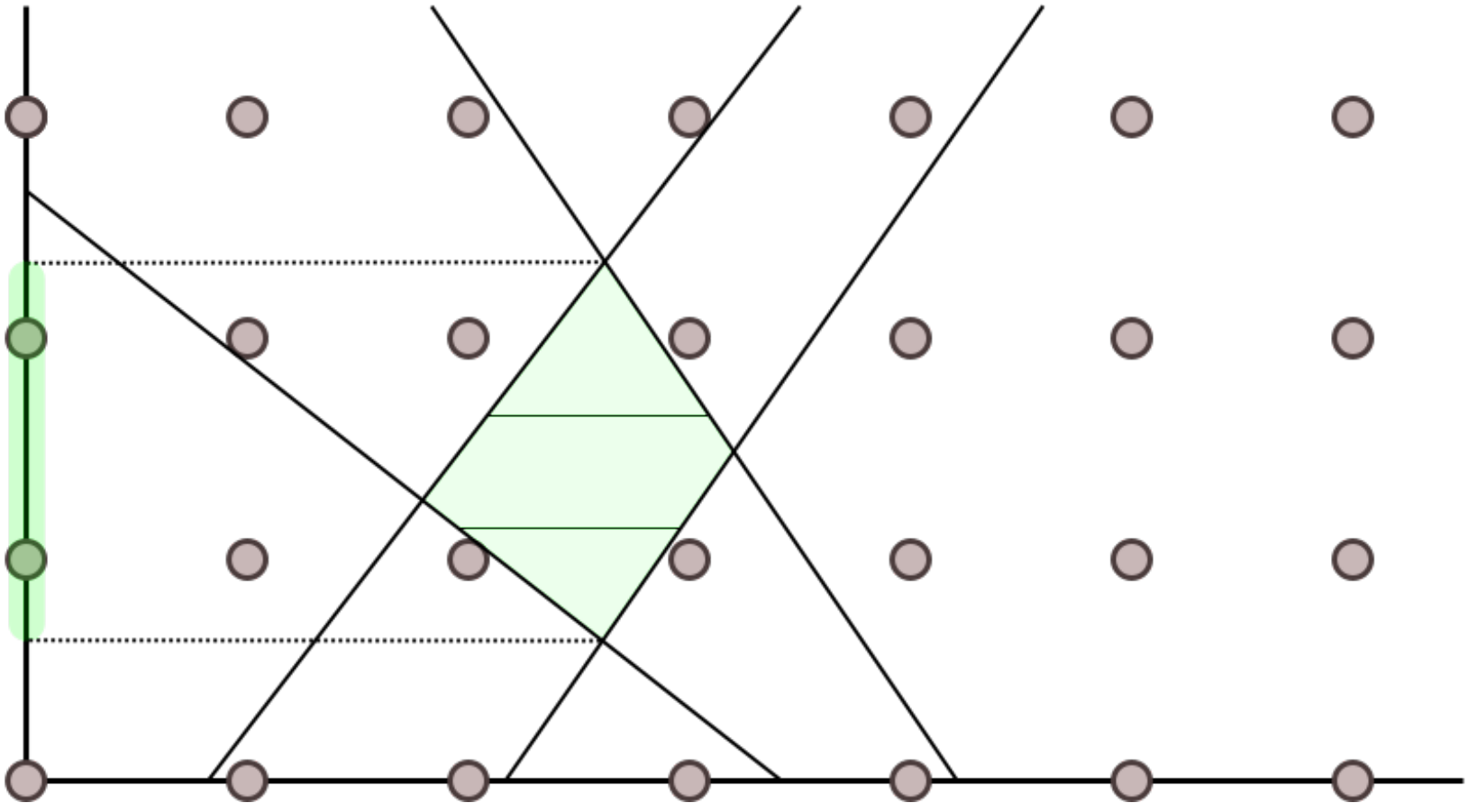
Dependence
Pugh's Omega Test



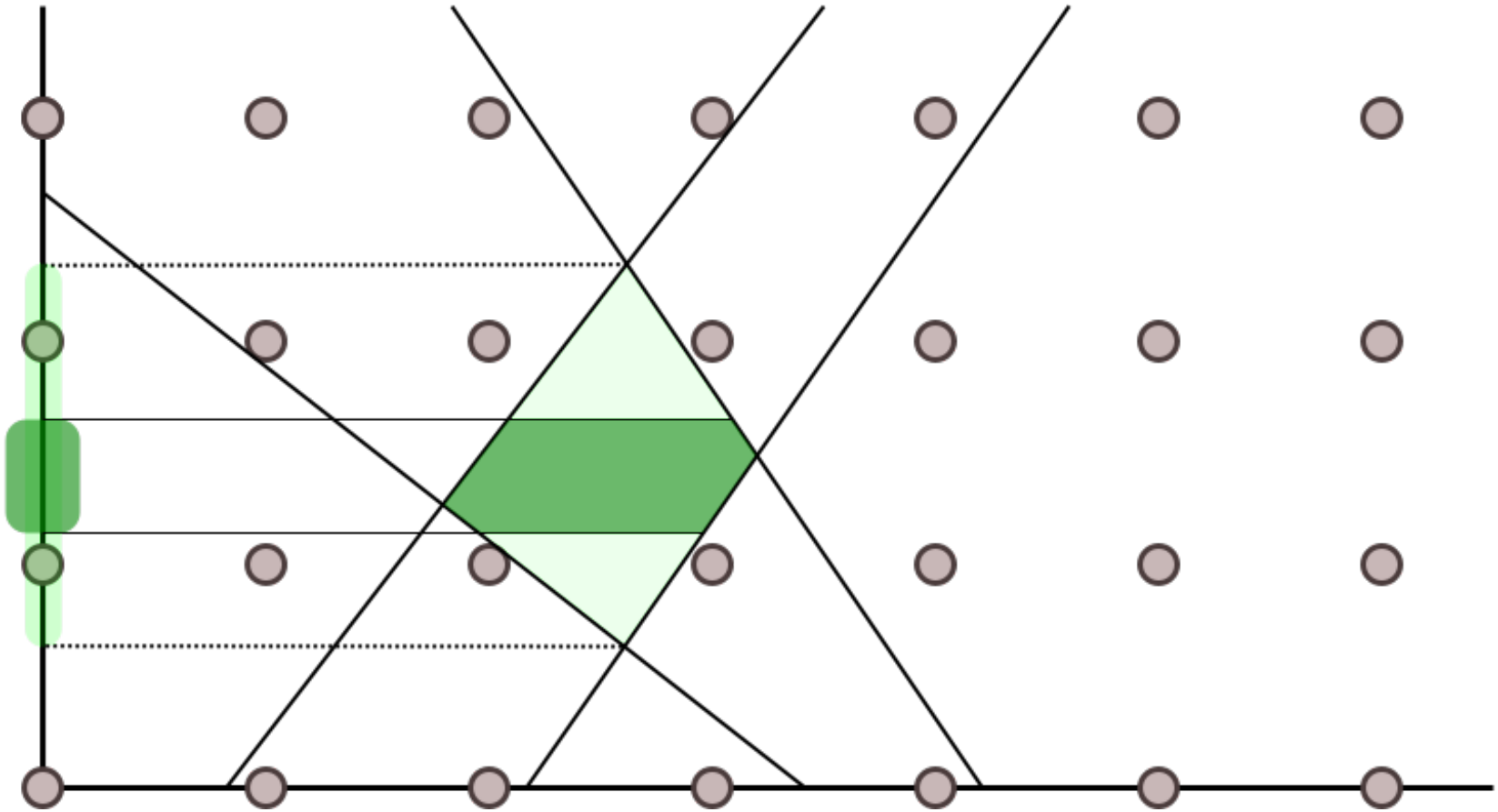
Dependence
Pugh's Omega Test



Dependence
Pugh's Omega Test



Dependence
Pugh's Omega Test



Summary

- Parallelism
- Types of dependence flow, anti and output
- Distance and direction vectors
- Classification of loop based data dependences
- Dependence tests: gcd, Banerjee and Omega

PPar CDT Advert