

The Substeps of Global Loop Transformations in the Polyhedral Model

Sven Verdoolaege

Overview

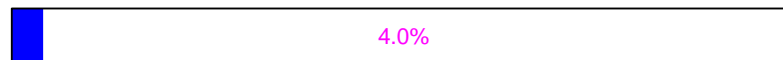
- Introduction
- Overview
- Subdivision
- Dependence analysis
- Translation
- Linear placement
- Example
- Related work
- Conclusions

Overview

- Introduction
- Overview
- Subdivision
- Dependence analysis
- Translation
- Linear placement
- Example
- Related work
- Conclusions

Loop transformations

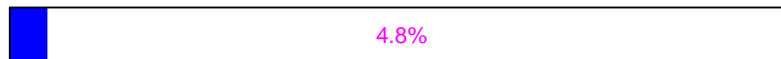
● What ?



Loop transformations

- What ?

A loop transformation changes the execution order of the iterations of one or more loops.

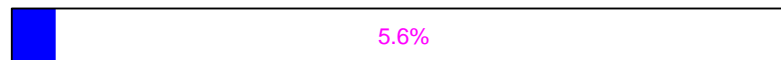


Loop transformations

- What ?

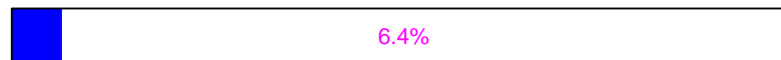
A loop transformation changes the execution order of the iterations of one or more loops.

- Why ?



Loop transformations

- What ?
A loop transformation changes the execution order of the iterations of one or more loops.
- Why ?
 - Parallelism
 - Locality
 - Memory size

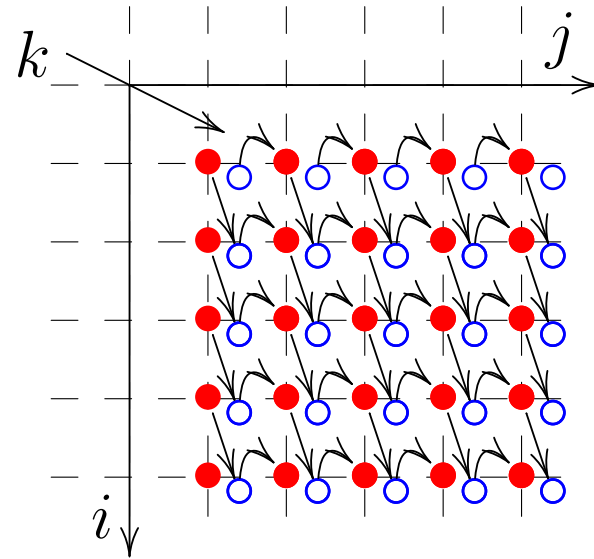


Parallelism example

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

Parallelism example

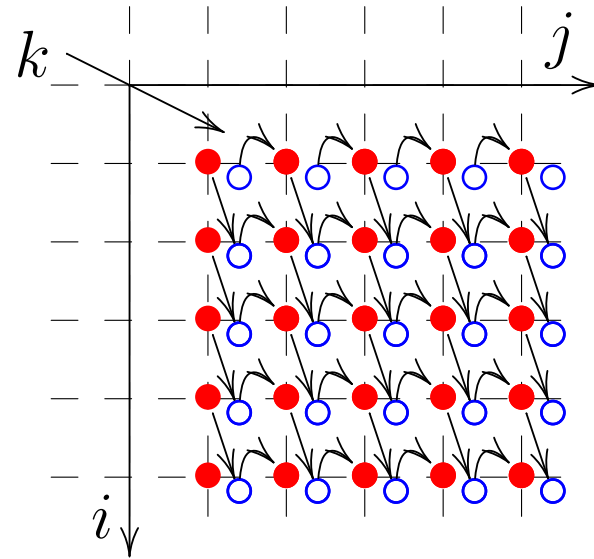
```
for (i = 1; i <= N; ++i)
  for (j = 1; j <= N; ++j)
    a[i][j] = f(b[i][j-1]);
    b[i][j] = g(a[i-1][j]);
```



Parallelism example

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

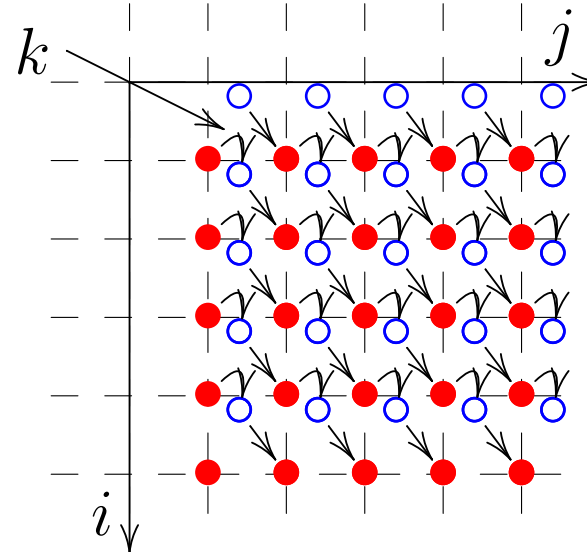
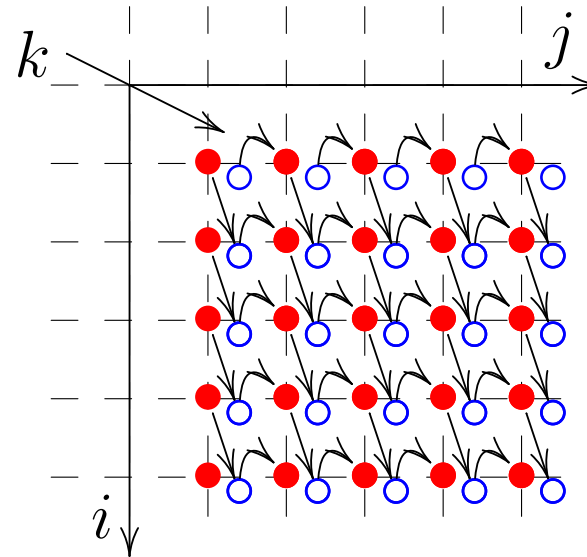
```
for (j = 1; j <= N; ++j)
  b[1][j] = g(a[0][j]);
for (i = 1; i <= N-1; ++i)
  for (j = 1; j <= N; ++j) {
    a[i][j] = f(b[i][j-1]);
    b[i+1][j] = g(a[i][j]);
  }
for (j = 1; j <= N; ++j)
  a[N][j] = f(b[N][j-1]);
```



Parallelism example

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

```
for (j = 1; j <= N; ++j)
  b[1][j] = g(a[0][j]);
for (i = 1; i <= N-1; ++i)
  for (j = 1; j <= N; ++j) {
    a[i][j] = f(b[i][j-1]);
    b[i+1][j] = g(a[i][j]);
  }
for (j = 1; j <= N; ++j)
  a[N][j] = f(b[N][j-1]);
```

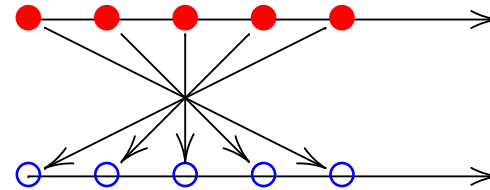


Locality example

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

Locality example

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



Locality example

```
for (i = 0; i <= N; ++i)
```

```
  a[i] = ...
```

```
for (i = 0; i <= N; ++i)
```

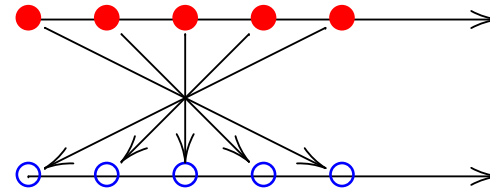
```
  b[i] = f(a[N-i])
```

```
for (i = 0; i <= N; ++i)
```

```
  a[i] = ...
```

```
for (i = 0; i <= N; ++i)
```

```
  b[N-i] = f(a[i])
```



Locality example

```
for (i = 0; i <= N; ++i)
```

```
  a[i] = ...
```

```
for (i = 0; i <= N; ++i)
```

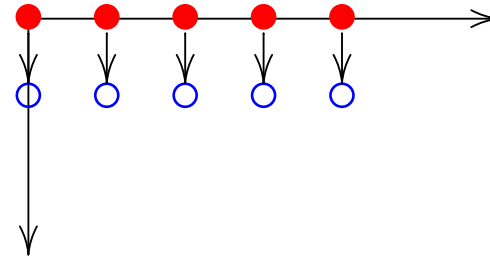
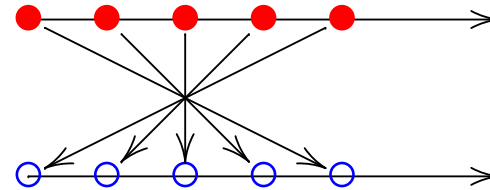
```
  b[i] = f(a[N-i])
```

```
for (i = 0; i <= N; ++i)
```

```
  a[i] = ...
```

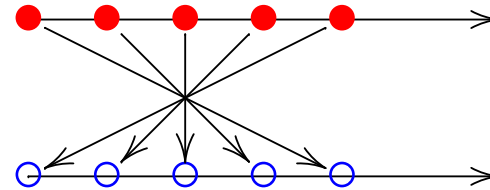
```
for (i = 0; i <= N; ++i)
```

```
  b[N-i] = f(a[i])
```

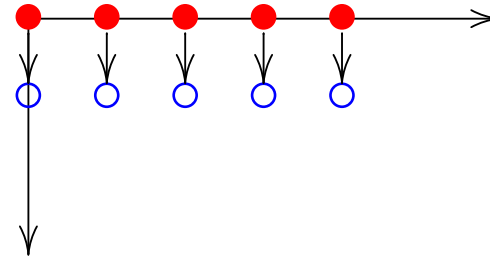


Locality example

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



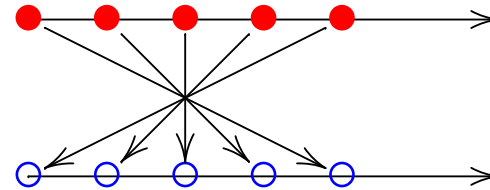
```
for (i = 0; i <= N; ++i)
  a[i] = ...
for (i = 0; i <= N; ++i)
  b[N-i] = f(a[i])
```



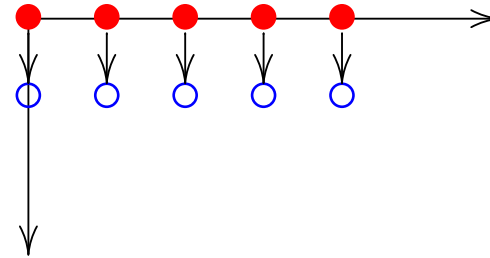
```
for (i = 0; i <= N; ++i) {
  a[i] = ...
  b[N-i] = f(a[i])
}
```

Locality example

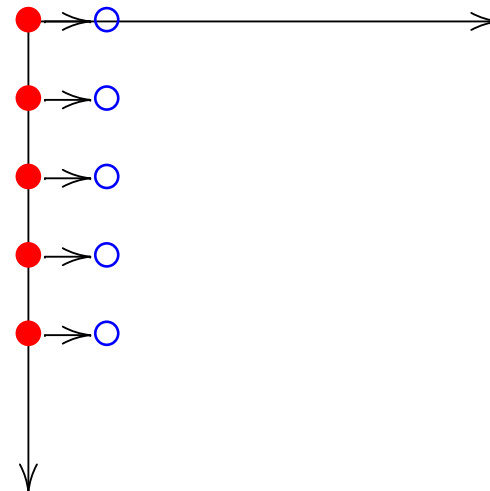
```
for (i = 0; i <= N; ++i)
  a[i] = ...
for (i = 0; i <= N; ++i)
  b[i] = f(a[N-i])
```



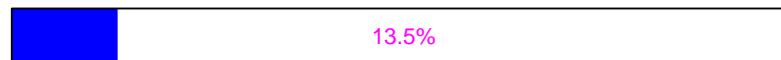
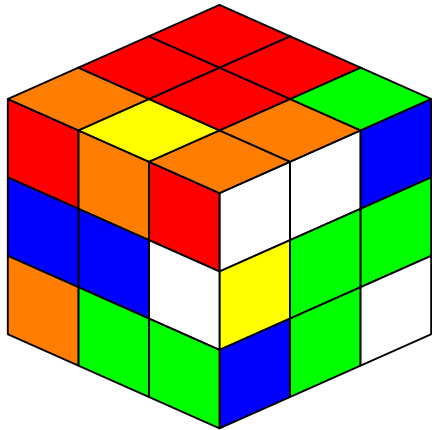
```
for (i = 0; i <= N; ++i)
  a[i] = ...
for (i = 0; i <= N; ++i)
  b[N-i] = f(a[i])
```



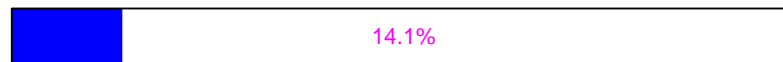
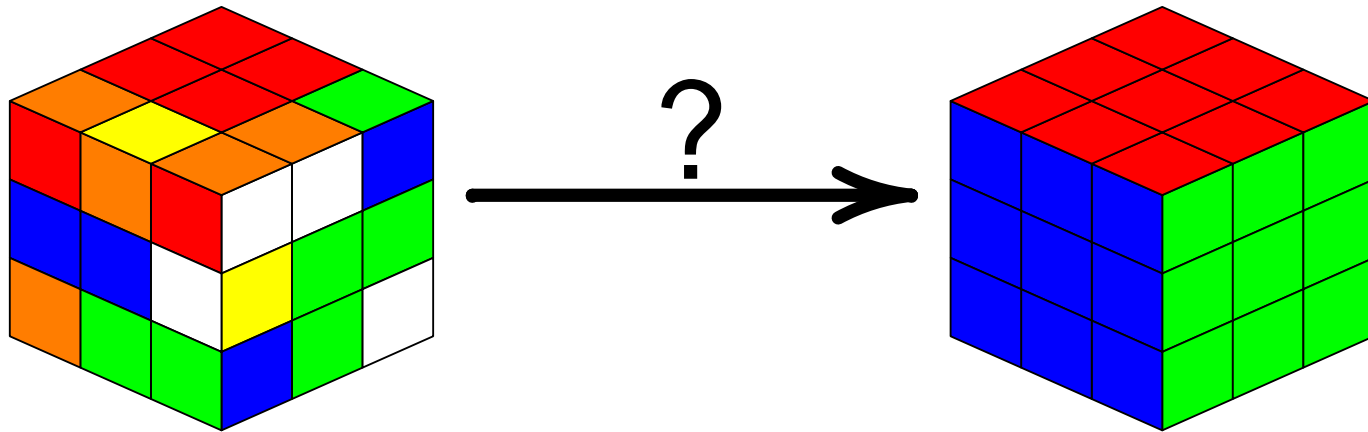
```
for (i = 0; i <= N; ++i) {
  a[i] = ...
  b[N-i] = f(a[i])
}
```



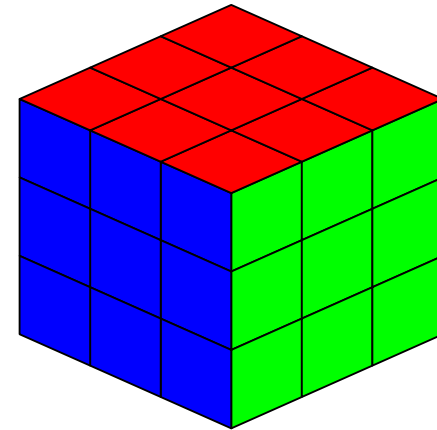
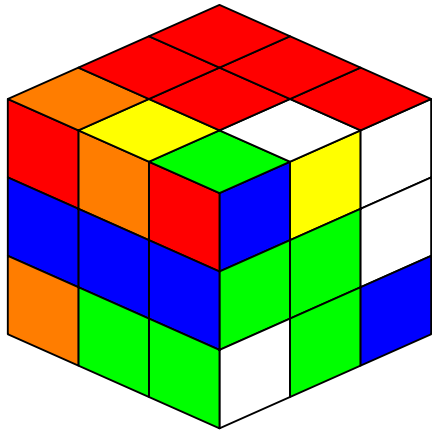
Rubik's cube



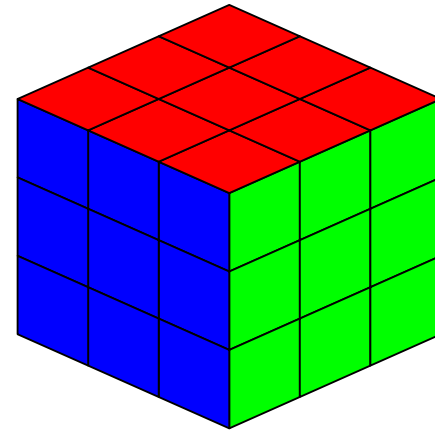
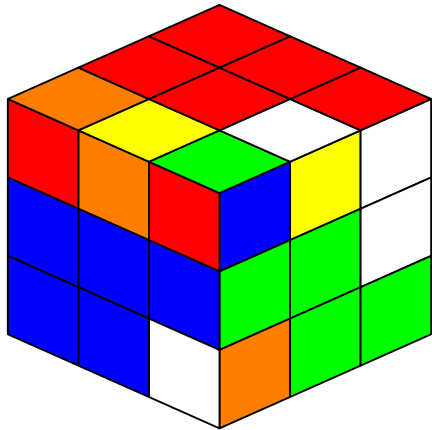
Rubik's cube



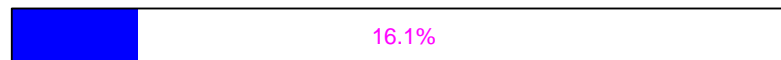
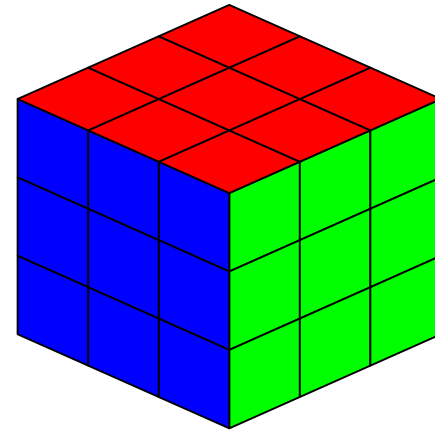
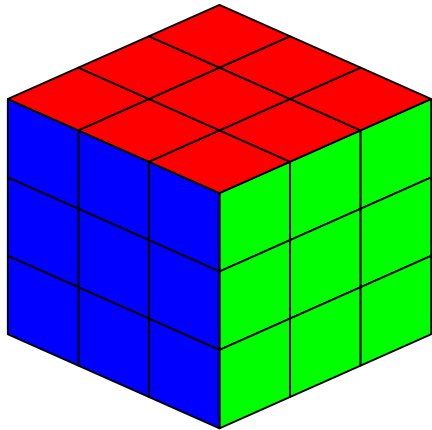
Rubik's cube



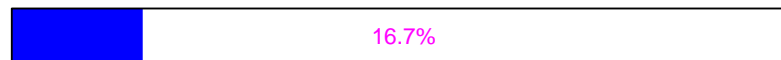
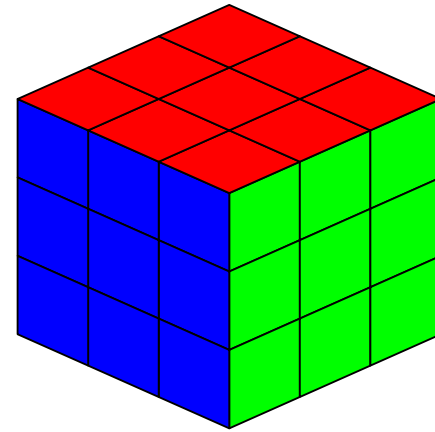
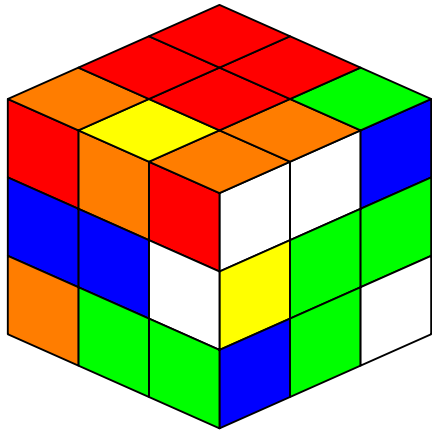
Rubik's cube



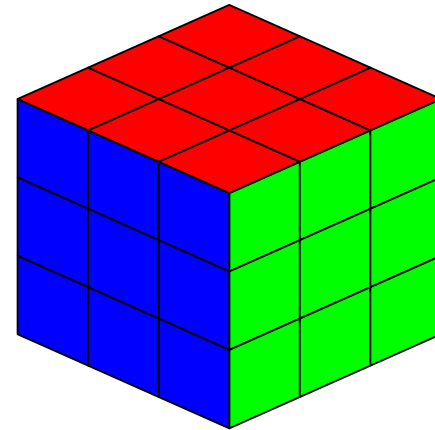
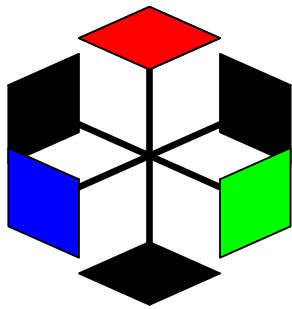
Rubik's cube



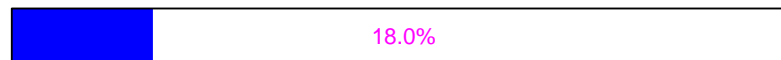
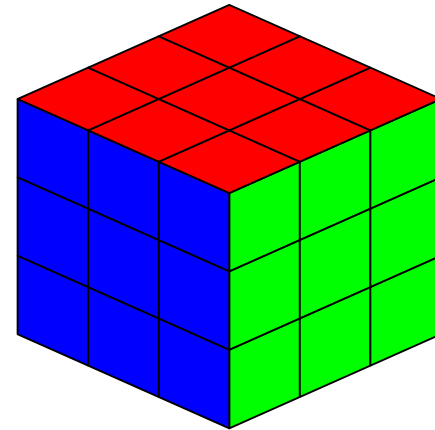
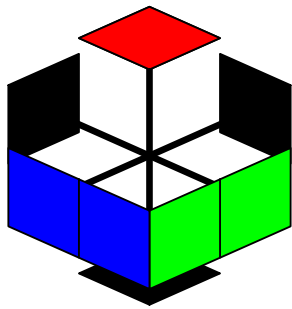
Rubik's cube



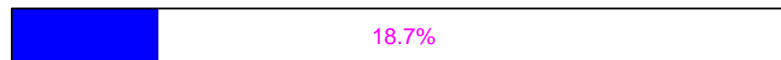
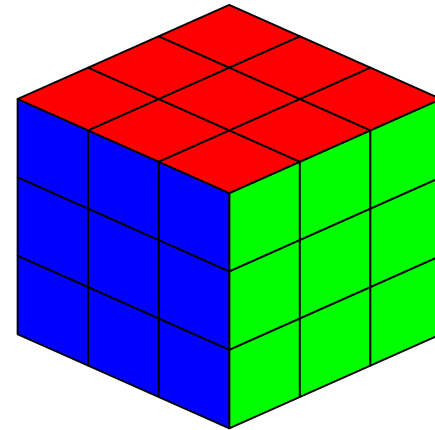
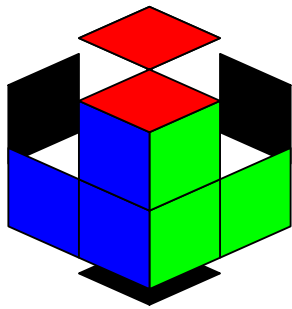
Rubik's cube



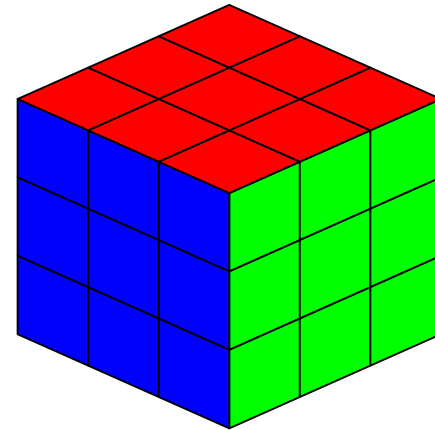
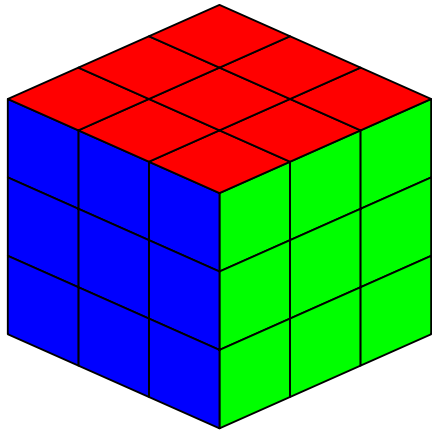
Rubik's cube



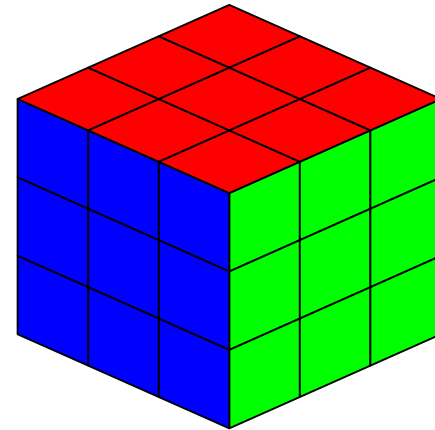
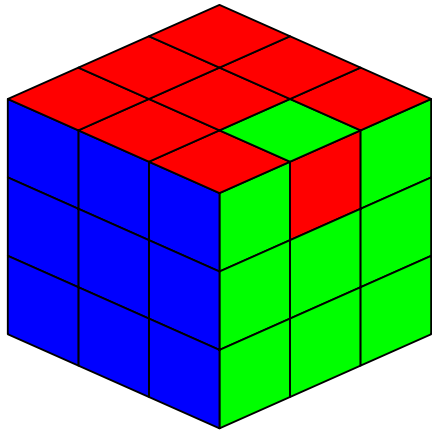
Rubik's cube



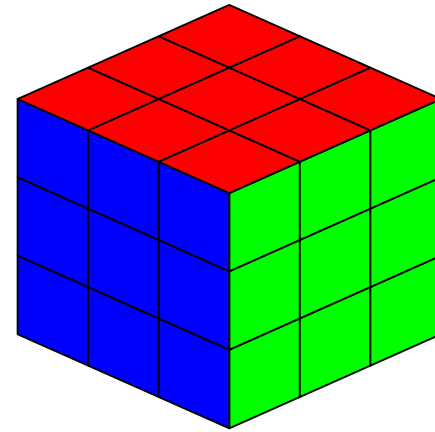
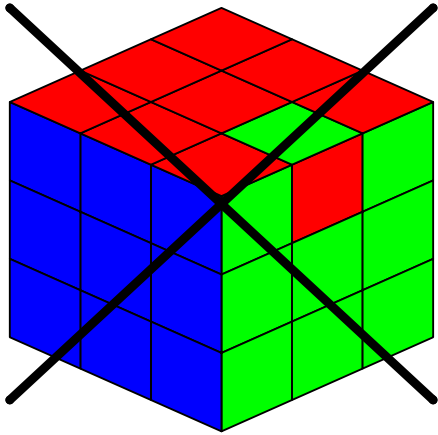
Rubik's cube



Rubik's cube



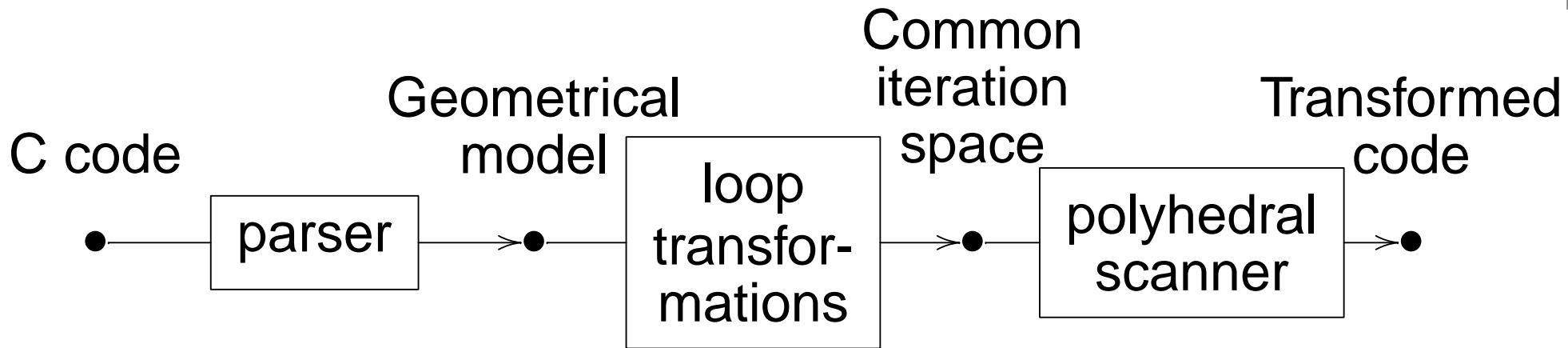
Rubik's cube



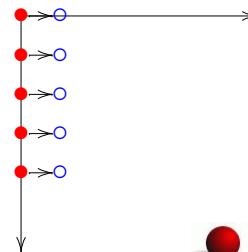
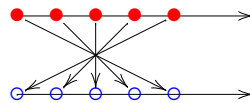
Overview

- Introduction
- Overview
- Subdivision
- Dependence analysis
- Translation
- Linear placement
- Example
- Related work
- Conclusions

Loop transformations overview



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



```
for (i = 0; i <= N; ++i) {
  a[i] = ...
  b[N-i] = f(a[i])
}
```

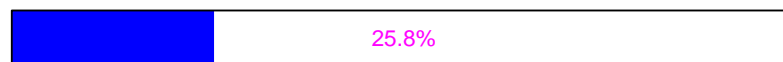
● PER

● LooPo

● w2p

● LoopGen

● CLooG



Model Extraction

- Input: C code (Silage, ...)
- Output: Iteration domains (polytopes) + dependences

Model Extraction

- Input: C code (Silage, ...)
- Output: Iteration domains (polytopes) + dependences

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

Model Extraction

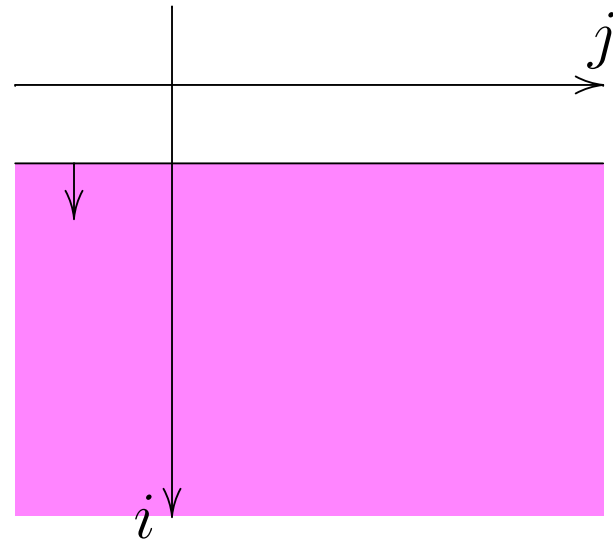
- Input: C code (Silage, ...)
- Output: Iteration domains (polytopes) + dependences

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

Iteration domain:

$$P = \{[i, j] \mid i \geq 1$$

}



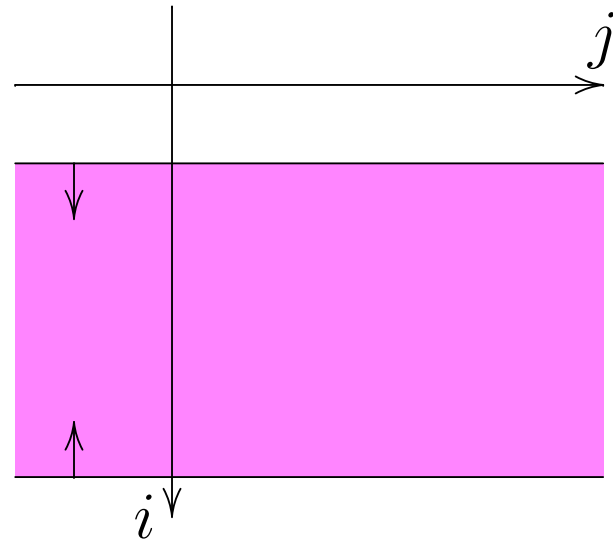
Model Extraction

- Input: C code (Silage, ...)
- Output: Iteration domains (polytopes) + dependences

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

Iteration domain:

$$P = \{[i, j] \mid \begin{array}{l} i \geq 1 \\ \wedge \quad i \leq N \end{array} \}$$



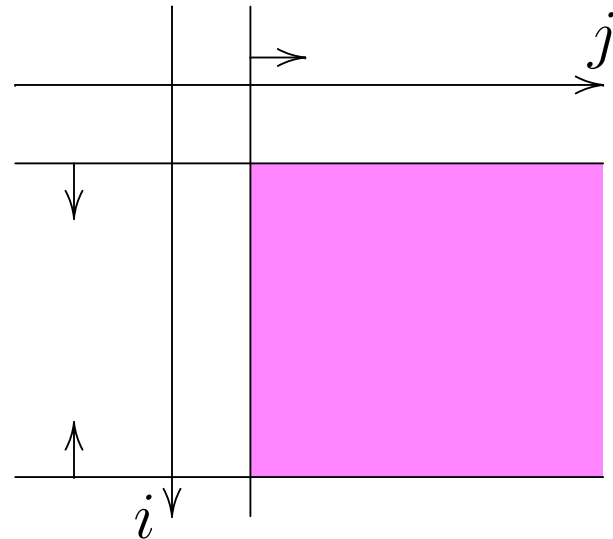
Model Extraction

- Input: C code (Silage, ...)
- Output: Iteration domains (polytopes) + dependences

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

Iteration domain:

$$P = \{[i, j] \mid \begin{array}{l} i \geq 1 \\ \wedge i \leq N \\ \wedge j \geq 1 \end{array} \}$$



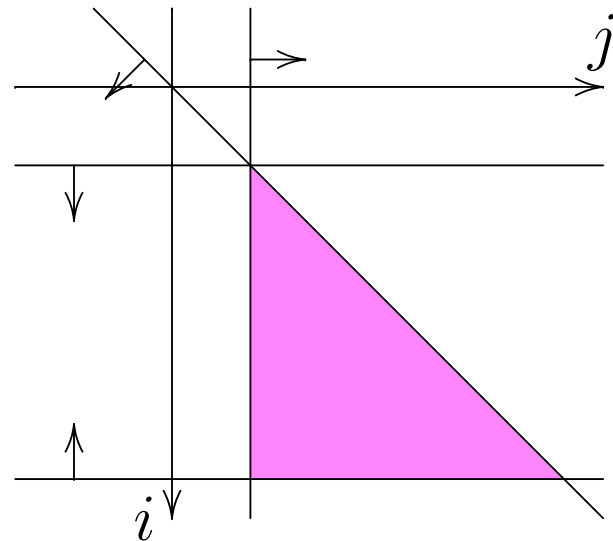
Model Extraction

- Input: C code (Silage, ...)
- Output: Iteration domains (polytopes) + dependences

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

Iteration domain:

$$P = \{[i, j] \mid \begin{aligned} & i \geq 1 \\ & \wedge i \leq N \\ & \wedge j \geq 1 \\ & \wedge j \leq i \end{aligned} \}$$



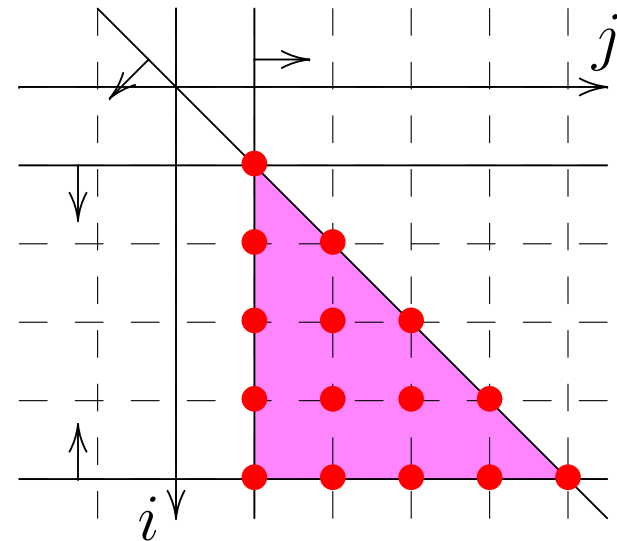
Model Extraction

- Input: C code (Silage, ...)
- Output: Iteration domains (polytopes) + dependences

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

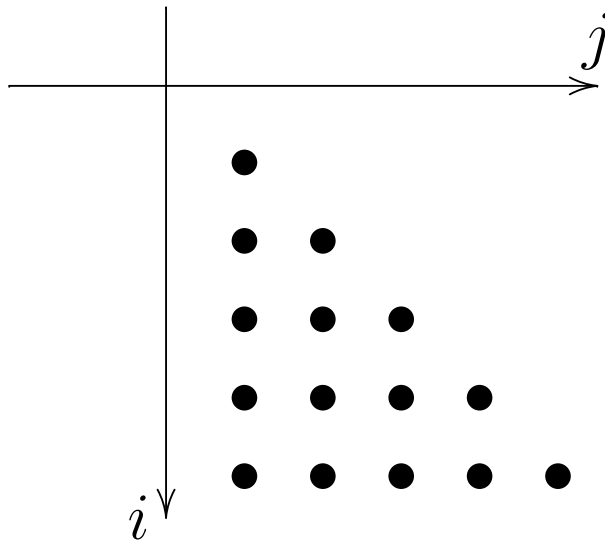
Iteration domain:

$$P = \{[i, j] \mid \begin{array}{l} i \geq 1 \\ \wedge i \leq N \\ \wedge j \geq 1 \\ \wedge j \leq i \end{array}\}$$



Scanning Polyhedra

- Input: Iteration domains (polytopes) in a common iteration space
- Output: C code
- Ordering: lexicographical
⇒ one loop per dimension



⇒

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

Overview

- Introduction
- Overview
- Subdivision
- Dependence analysis
- Translation
- Linear placement
- Example
- Related work
- Conclusions

Affine-by-statement Scheduling

The iteration domain of each statement is transformed through an affine transformation

$$\forall \vec{i} \in P_X : \mathcal{A}_X(\vec{i}) = A_X \vec{i} + \vec{a}_X$$

$\mathcal{A}_X(P_X)$ is a polytope

$$t_X(\vec{i}) < t_Y(\vec{j}) \iff \mathcal{A}_X(\vec{i}) \prec \mathcal{A}_Y(\vec{j})$$

Affine-by-statement Scheduling

The iteration domain of each statement is transformed through an affine transformation

$$\forall \vec{i} \in P_X : \mathcal{A}_X(\vec{i}) = A_X \vec{i} + \vec{a}_X$$

$\mathcal{A}_X(P_X)$ is a polytope

$$t_X(\vec{i}) < t_Y(\vec{j}) \iff \mathcal{A}_X(\vec{i}) \prec \mathcal{A}_Y(\vec{j})$$

Linear transformation: transforms the polytope

Affine-by-statement Scheduling

The iteration domain of each statement is transformed through an affine transformation

$$\forall \vec{i} \in P_X : \mathcal{A}_X(\vec{i}) = A_X \vec{i} + \vec{a}_X$$

$\mathcal{A}_X(P_X)$ is a polytope

$$t_X(\vec{i}) < t_Y(\vec{j}) \iff \mathcal{A}_X(\vec{i}) \prec \mathcal{A}_Y(\vec{j})$$

Linear transformation: transforms the polytope

Offset: translates the polytope

Affine-by-statement Scheduling

The iteration domain of each statement is transformed through an affine transformation

$$\forall \vec{i} \in P_X : \mathcal{A}_X(\vec{i}) = A_X \vec{i} + \vec{a}_X$$

$\mathcal{A}_X(P_X)$ is a polytope

$$t_X(\vec{i}) < t_Y(\vec{j}) \iff \mathcal{A}_X(\vec{i}) \prec \mathcal{A}_Y(\vec{j})$$

Linear transformation: transforms the polytope

Offset: translates the polytope

If A_X is unimodular then $\# \mathcal{A}_X(P_X) = \# P_X$

Alternative approach

Affine-by-statement “mapping” + global ordering vector $\vec{\pi}$
(used previously at IMEC)

$$t_X(\vec{i}) < t_Y(\vec{j}) \iff \vec{\pi}^T \mathcal{A}_X(\vec{i}) < \vec{\pi}^T \mathcal{A}_Y(\vec{j})$$

Alternative approach

Affine-by-statement “mapping” + global ordering vector $\vec{\pi}$
(used previously at IMEC)

$$t_X(\vec{i}) < t_Y(\vec{j}) \iff \vec{\pi}^T \mathcal{A}_X(\vec{i}) \prec \vec{\pi}^T \mathcal{A}_Y(\vec{j})$$

Ordering is equivalent to an extra linear transformation:

$$\vec{\pi}^T \vec{a} < \vec{\pi}^T \vec{b} \Rightarrow U_{\vec{\pi}} \vec{a} \prec U_{\vec{\pi}} \vec{b}$$

$$U_{\vec{\pi}}(A_X \vec{i} + \vec{a}_X) = A_X'' \vec{i} + \vec{a}_X''$$

$U_{\vec{\pi}}$: Unimodular extension of $\vec{\pi}^T$

Alternative approach

Affine-by-statement “mapping” + global ordering vector $\vec{\pi}$
(used previously at IMEC)

$$t_X(\vec{i}) < t_Y(\vec{j}) \iff \vec{\pi}^T \mathcal{A}_X(\vec{i}) \prec \vec{\pi}^T \mathcal{A}_Y(\vec{j})$$

Ordering is equivalent to an extra linear transformation:

$$\vec{\pi}^T \vec{a} < \vec{\pi}^T \vec{b} \Rightarrow U_{\vec{\pi}} \vec{a} \prec U_{\vec{\pi}} \vec{b}$$

$$U_{\vec{\pi}}(A_X \vec{i} + \vec{a}_X) = A_X'' \vec{i} + \vec{a}_X''$$

$U_{\vec{\pi}}$: Unimodular extension of $\vec{\pi}^T$

\Rightarrow more complicated without a clear gain

Subdivision

- Dependence Analysis
(part of model extraction)
- Linear transformation step
⇒ Determine all A_X
- Translation step
⇒ Determine all \vec{a}_X

⇒ Linear transformation step needs to ensure that a valid translation still exists

Overview

- Introduction
- Overview
- Subdivision
- Dependence analysis
- Translation
- Linear placement
- Example
- Related work
- Conclusions

Dependence Relation

The dependence relation contains all pairs of iterations that depend on each other.

For each pair of statements A and B containing references to the same array: $A[J_A(\vec{i})]$ and $A[J_B(\vec{i})]$

$$\delta_{A,B} = \{(\vec{x}, \vec{y}) \in (I_A, I_B) \mid \vec{x} \prec \vec{y} \wedge J_A(\vec{x}) = J_B(\vec{y})\}$$

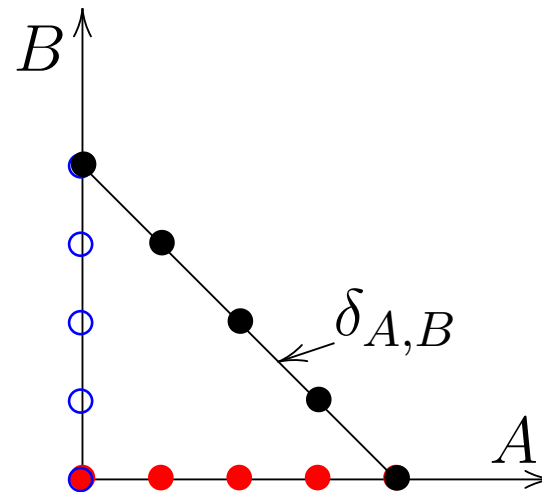
Dependence Relation

The dependence relation contains all pairs of iterations that depend on each other.

For each pair of statements A and B containing references to the same array: $A[J_A(\vec{i})]$ and $A[J_B(\vec{i})]$

$$\delta_{A,B} = \{(\vec{x}, \vec{y}) \in (I_A, I_B) \mid \vec{x} \prec \vec{y} \wedge J_A(\vec{x}) = J_B(\vec{y})\}$$

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



Dependence Polytope

The dependence polytope is the convex hull of all distance vectors.

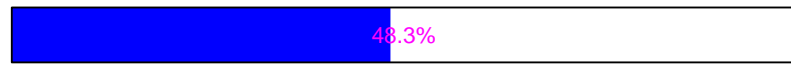
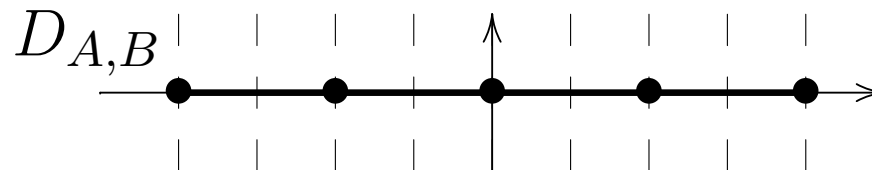
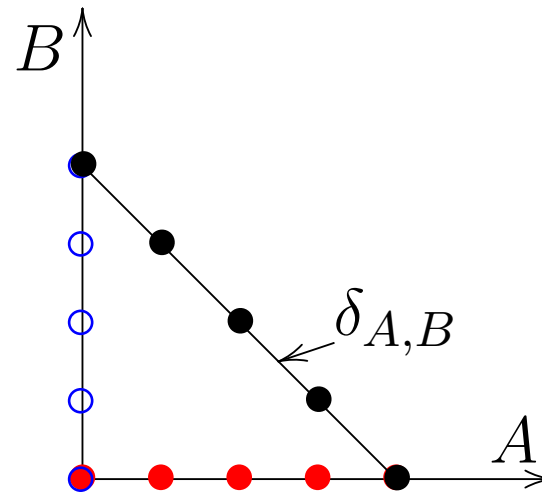
$$D_{A,B} = \text{conv} \{ \vec{y} - \vec{x} \mid \vec{x} \delta_{A,B} \vec{y} \}$$

Dependence Polytope

The dependence polytope is the convex hull of all distance vectors.

$$D_{A,B} = \text{conv} \{ \vec{y} - \vec{x} \mid \vec{x} \delta_{A,B} \vec{y} \}$$

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



Valid Transformation

Each iteration of a statement should be executed after all the statement iterations on which it depends:

$$\mathcal{A}_X(\vec{i}) \prec \mathcal{A}_Y(\vec{j}) \quad \text{if } \vec{i} \delta_{X,Y} \vec{j}$$

Valid Transformation

Each iteration of a statement should be executed after all the statement iterations on which it depends:

$$\mathcal{A}_X(\vec{i}) \prec \mathcal{A}_Y(\vec{j}) \quad \text{if } \vec{i} \delta_{X,Y} \vec{j}$$

Dependence relation after transformation:

$$\delta'_{X,Y} = \left\{ (\mathcal{A}_X(\vec{i}), \mathcal{A}_Y(\vec{j})) \mid \vec{i} \delta_{X,Y} \vec{j} \right\}$$

Valid Transformation

Each iteration of a statement should be executed after all the statement iterations on which it depends:

$$\mathcal{A}_X(\vec{i}) \prec \mathcal{A}_Y(\vec{j}) \quad \text{if } \vec{i} \delta_{X,Y} \vec{j}$$

Dependence relation after transformation:

$$\delta'_{X,Y} = \left\{ (\mathcal{A}_X(\vec{i}), \mathcal{A}_Y(\vec{j})) \mid \vec{i} \delta_{X,Y} \vec{j} \right\}$$

Validity constraint:

$$D'_{X,Y} \succ \vec{0}$$

Overview

- Introduction
- Overview
- Subdivision
- Dependence analysis
- Translation
- Linear placement
- Example
- Related work
- Conclusions

Incremental Translation

Translation = loop fusion + loop shifting

1. Initialize G .
2. If G contains a single node, stop.
3. Select two nodes p_1 and p_2 in G .
4. Select an offset $\vec{\alpha}_{p_1, p_2} = \vec{a}_{p_2} - \vec{a}_{p_1}$ for p_2 relative to p_1 .
5. Combine p_1 and p_2 into one node.
6. Goto 2.

Incremental Translation

Translation = loop fusion + loop shifting

1. Initialize G .
2. If G contains a single node, stop.
3. Select two nodes p_1 and p_2 in G .
4. Select an offset $\vec{\alpha}_{p_1, p_2} = \vec{a}_{p_2} - \vec{a}_{p_1}$ for p_2 relative to p_1 .
5. Combine p_1 and p_2 into one node.
6. Goto 2.

How to select relative offset ?

Incremental Translation

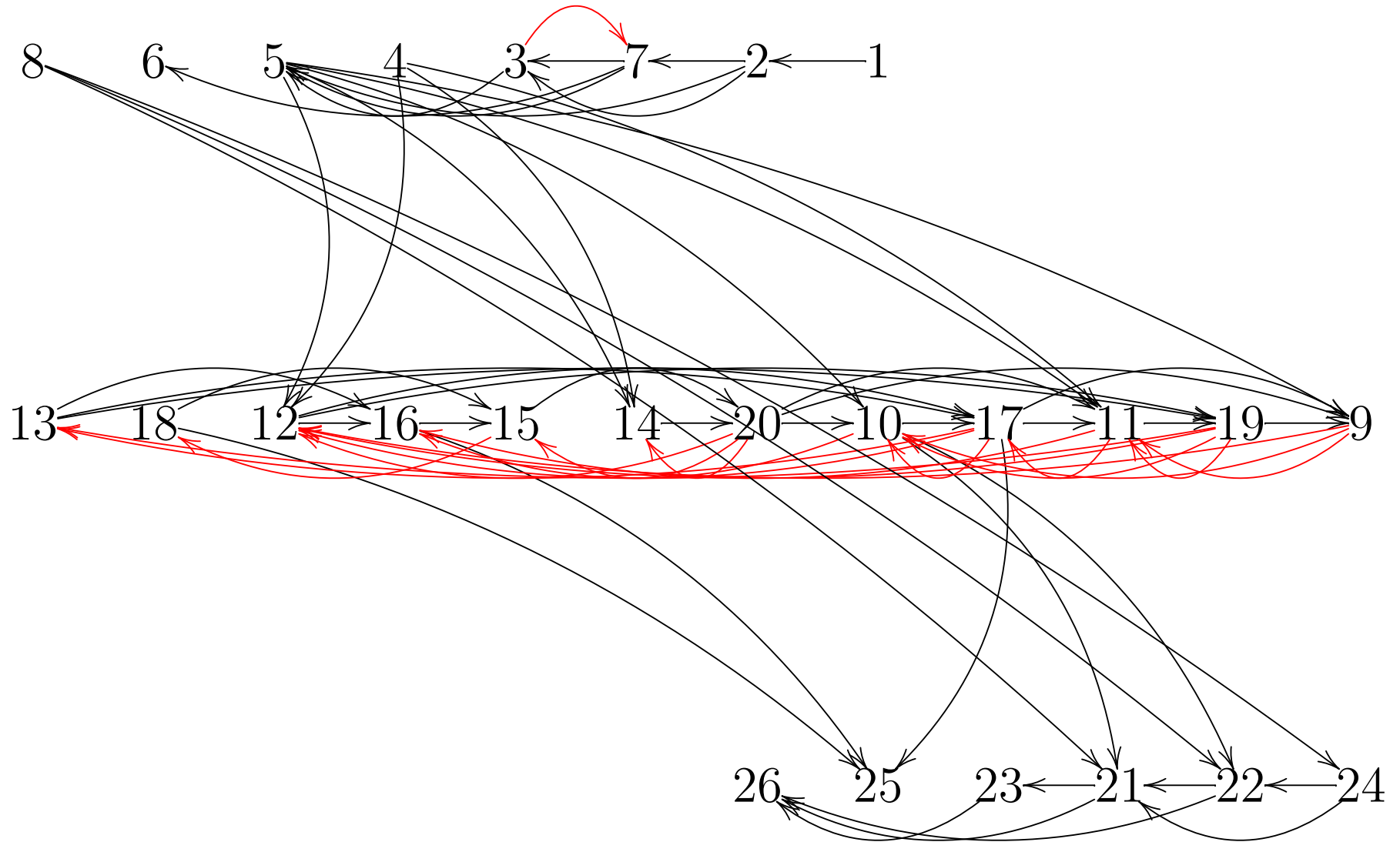
Translation = loop fusion + loop shifting

1. Initialize G .
2. If G contains a single node, stop.
3. Select two nodes p_1 and p_2 in G .
4. Select an offset $\vec{\alpha}_{p_1,p_2} = \vec{a}_{p_2} - \vec{a}_{p_1}$ for p_2 relative to p_1 .
5. Combine p_1 and p_2 into one node.
6. Goto 2.

How to select relative offset ?

⇒ Easy if dependence graph is acyclic

USVD Dependence Graph



Cycles and Valid Relative Offsets

Let π be a cycle in the dependence graph

$$\pi = (p_1, p_2, \dots, p_{n-1}, p_n = p_1) \quad \vec{d}_{i,j} \in D_{i,j}$$

$$\vec{d}_{1,2} + \vec{a}_2 - \vec{a}_1 + \vec{d}_{2,3} + \vec{a}_3 - \vec{a}_2 + \dots + \vec{d}_{n-1,1} + \vec{a}_1 - \vec{a}_{n-1} \succ \vec{0}$$

Cycles and Valid Relative Offsets

Let π be a cycle in the dependence graph

$$\pi = (p_1, p_2, \dots, p_{n-1}, p_n = p_1) \quad \vec{d}_{i,j} \in D_{i,j}$$

$$\vec{d}_{1,2} + \cancel{\vec{a}_2} - \cancel{\vec{a}_1} + \vec{d}_{2,3} + \cancel{\vec{a}_3} - \cancel{\vec{a}_2} + \dots + \vec{d}_{n-1,1} + \cancel{\vec{a}_1} - \cancel{\vec{a}_{n-1}} \succ \vec{0}$$

Cycles and Valid Relative Offsets

Let π be a cycle in the dependence graph

$$\pi = (p_1, p_2, \dots, p_{n-1}, p_n = p_1) \quad \vec{d}_{i,j} \in D_{i,j}$$

$$\vec{d}_{1,2} + \vec{a}_2 - \vec{a}_1 + \vec{d}_{2,3} + \vec{a}_3 - \vec{a}_2 + \dots + \vec{d}_{n-1,1} + \vec{a}_1 - \vec{a}_{n-1} \succ \vec{0}$$

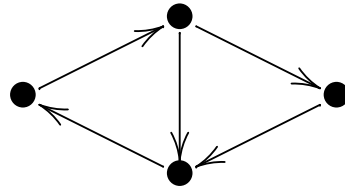
$$\mathcal{V}_{j,j} = \left(\sum_{i=1}^{n-1} D_{i,i+1} \right) \succ \vec{0}$$

Feasibility criterion:

$$\vec{0} \prec \min_j \mathcal{V}_{j,j}$$

New Cycles

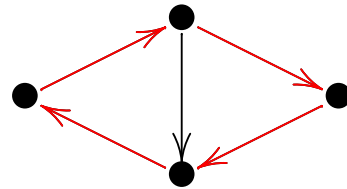
Combining two nodes creates new cycles



2 cycles

New Cycles

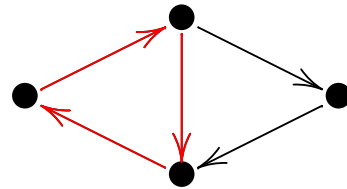
Combining two nodes creates new cycles



2 cycles

New Cycles

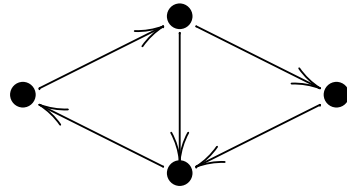
Combining two nodes creates new cycles



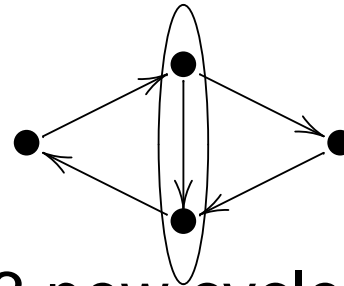
2 cycles

New Cycles

Combining two nodes creates new cycles



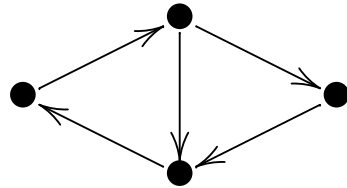
2 cycles



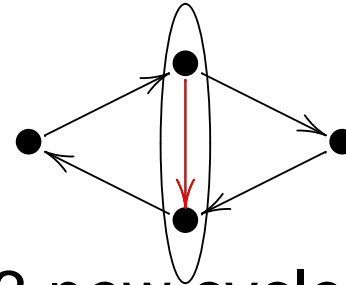
3 new cycles

New Cycles

Combining two nodes creates new cycles



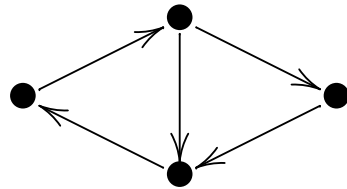
2 cycles



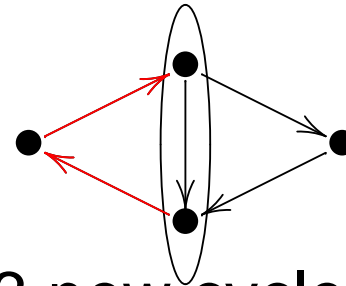
3 new cycles

New Cycles

Combining two nodes creates new cycles



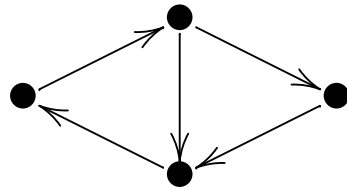
2 cycles



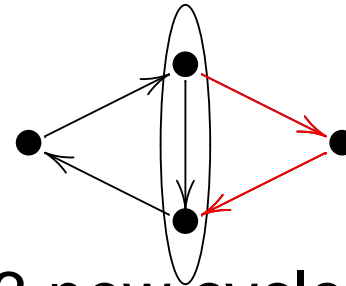
3 new cycles

New Cycles

Combining two nodes creates new cycles



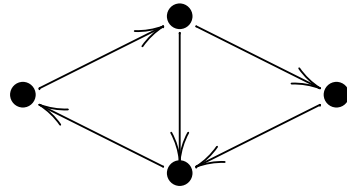
2 cycles



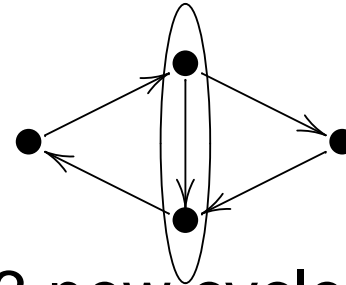
3 new cycles

New Cycles

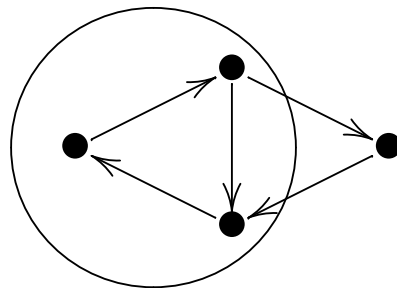
Combining two nodes creates new cycles



2 cycles

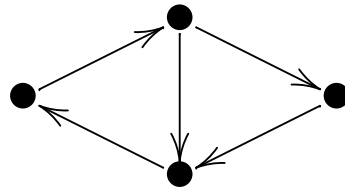


3 new cycles

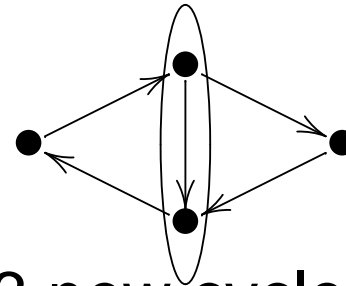


New Cycles

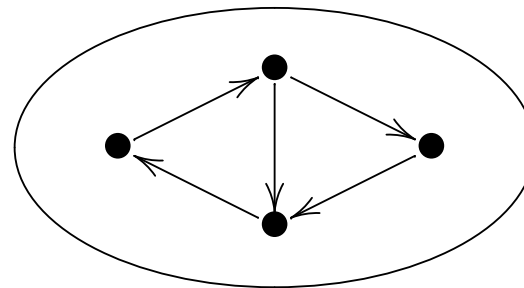
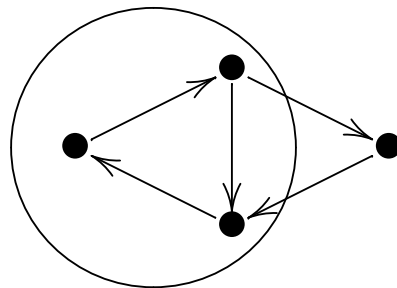
Combining two nodes creates new cycles



2 cycles



3 new cycles



New Cycles

When combining p_1 and p_2

$$\vec{d}_{1,2} + \vec{a}_2 - \vec{a}_1 \succ \vec{0} \quad \forall \vec{d}_{1,2} \in D_{1,2}$$

Minimal distance vectors:

$$\vec{d}_{i,j} = \min_{\prec} \mathcal{V}_{i,j}$$

$$\vec{d}_{1,2} + \vec{a}_{1,2} \succcurlyeq \vec{0} \quad \vec{d}_{2,1} - \vec{a}_{1,2} \succcurlyeq \vec{0}$$

Constraint on relative offset:

$$-\vec{d}_{1,2} \preccurlyeq \vec{a}_{1,2} \preccurlyeq \vec{d}_{2,1}$$

Overview

- Introduction
- Overview
- Subdivision
- Dependence analysis
- Translation
- **Linear placement**
- Example
- Related work
- Conclusions

Linear Transformation

Legality Constraint:

$$\vec{0} \prec \min_j \bigcup_j \mathcal{V}_{j,j}$$

⇒ For each SCC a constraint of the form:

$$[A_1 \quad A_2 \quad A_3 \quad \cdots \quad A_n] P \succ \vec{0}$$

⇒ Top rows of A_i should correspond to linear equalities in P

Overview

- Introduction
- Overview
- Subdivision
- Dependence analysis
- Translation
- Linear placement
- Example
- Related work
- Conclusions

Cost functions

- Linear placement:
 - regularity
 - ⇒ minimize dimension of dependence polytope
 - locality over (indirect) self-dependences
 - ⇒ reuse in inner loop(s)
- Translation:
 - locality over other dependences
 - ⇒ minimize distance (lexicographically)
 - memory requirement
 - data reuse

Illustrative example

Initial specification:

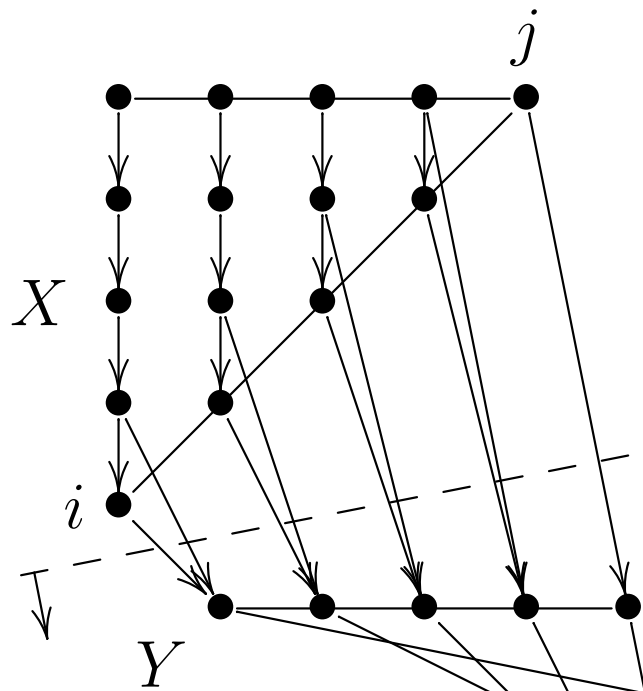
```
for (i = 1; i <= N; ++i)
    for (j = 1; j <= N-i+1; ++j)
        a[i][j] = in[i][j] + a[i-1][j];

for (p = 1; p <= N; ++p)
    b[p][1] = f( a[N-p+1][p], a[N-p][p] );

for (k = 1; k <= N; ++k)
    for (l = 1; l <= k; ++l)
        b[k][l+1] = g( b[k][l] );
```

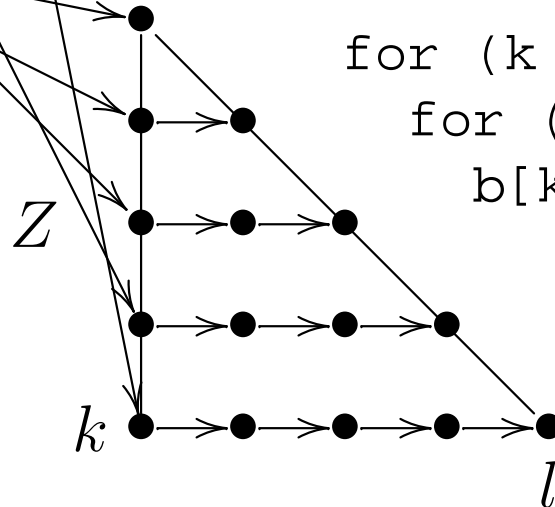
Buffer space required: $2N$

Model Extraction



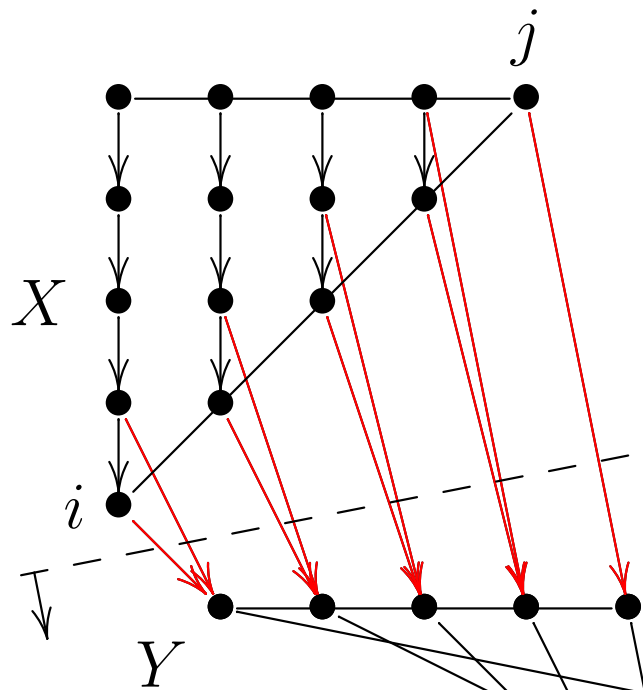
```
for (i = 1; i <= N; ++i)
  for (j = 1; j <= N-i+1; ++j)
    a[i][j] = in[i][j] + a[i-1][j];
```

```
for (q = 1; q <= 1; ++q)
  for (p = 1; p <= N; ++p)
    b[p][1] = f(a[N-p+1][p], a[N-p][p]);
```



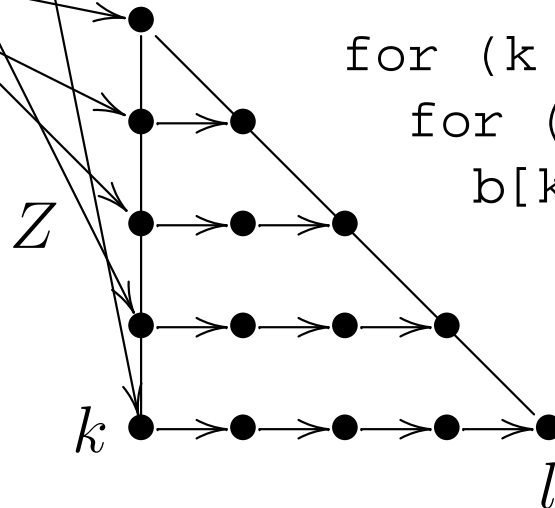
```
for (k = 1; k <= N; ++k)
  for (l = 1; l <= k; ++l)
    b[k][l+1] = g(b[k][l]);
```

Model Extraction



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

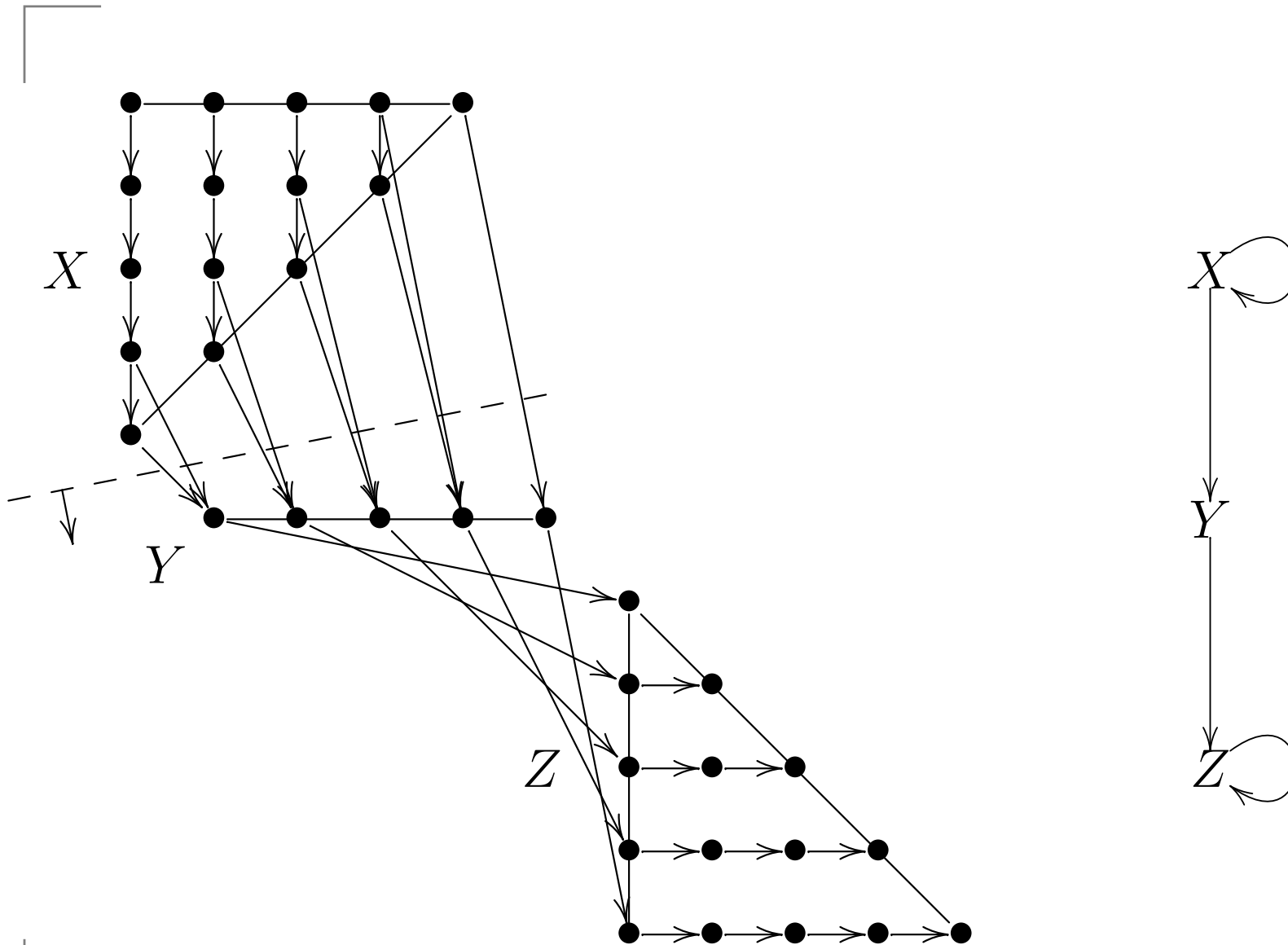
```
for (q = 1; q <= 1; ++q)
  for (p = 1; p <= N; ++p)
    b[p][1] = f(a[N-p+1][p], a[N-p][p]);
```



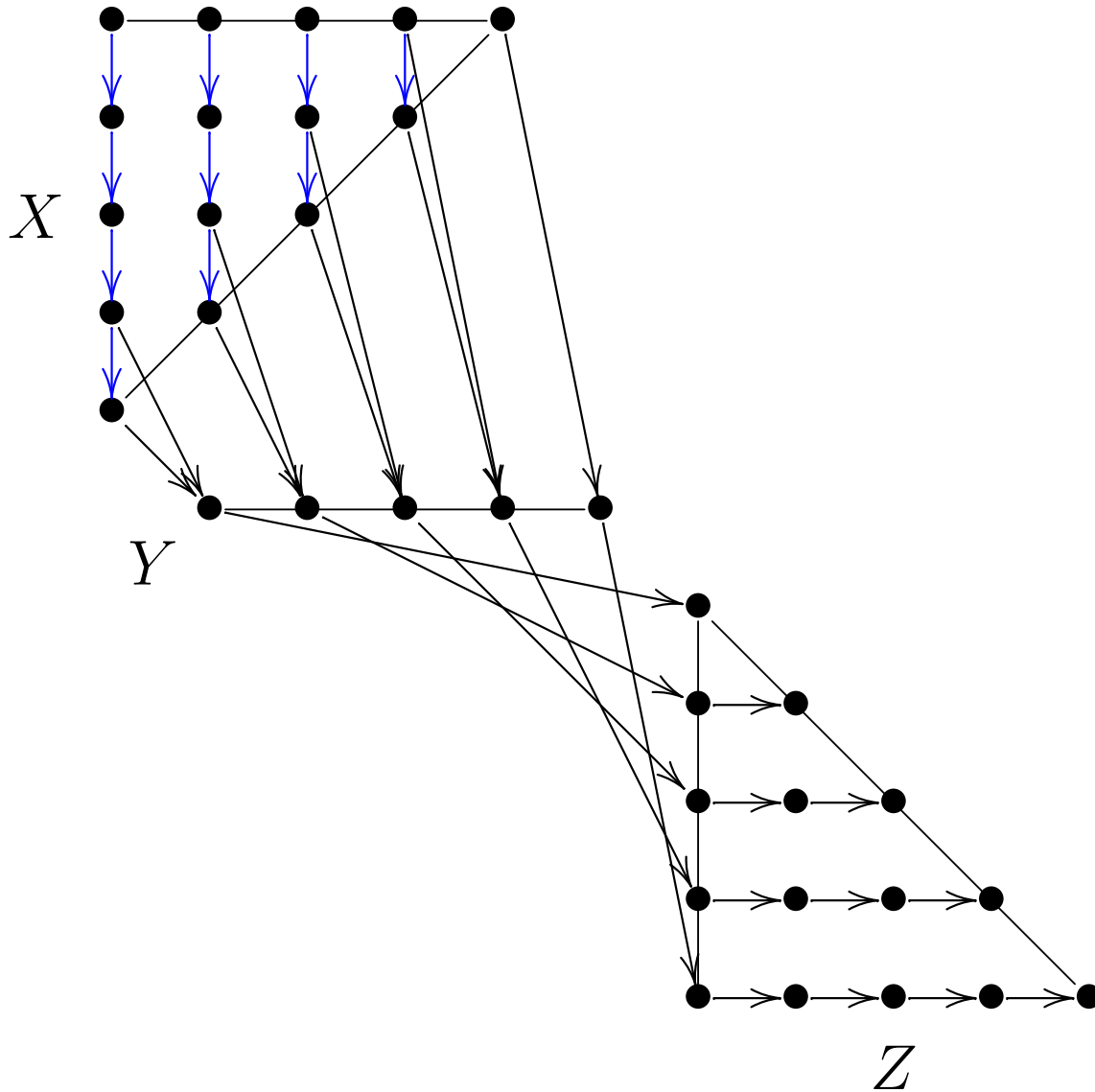
```
for (k = 1; k <= N; ++k)
  for (l = 1; l <= k; ++l)
    b[k][l+1] = g(b[k][l]);
```



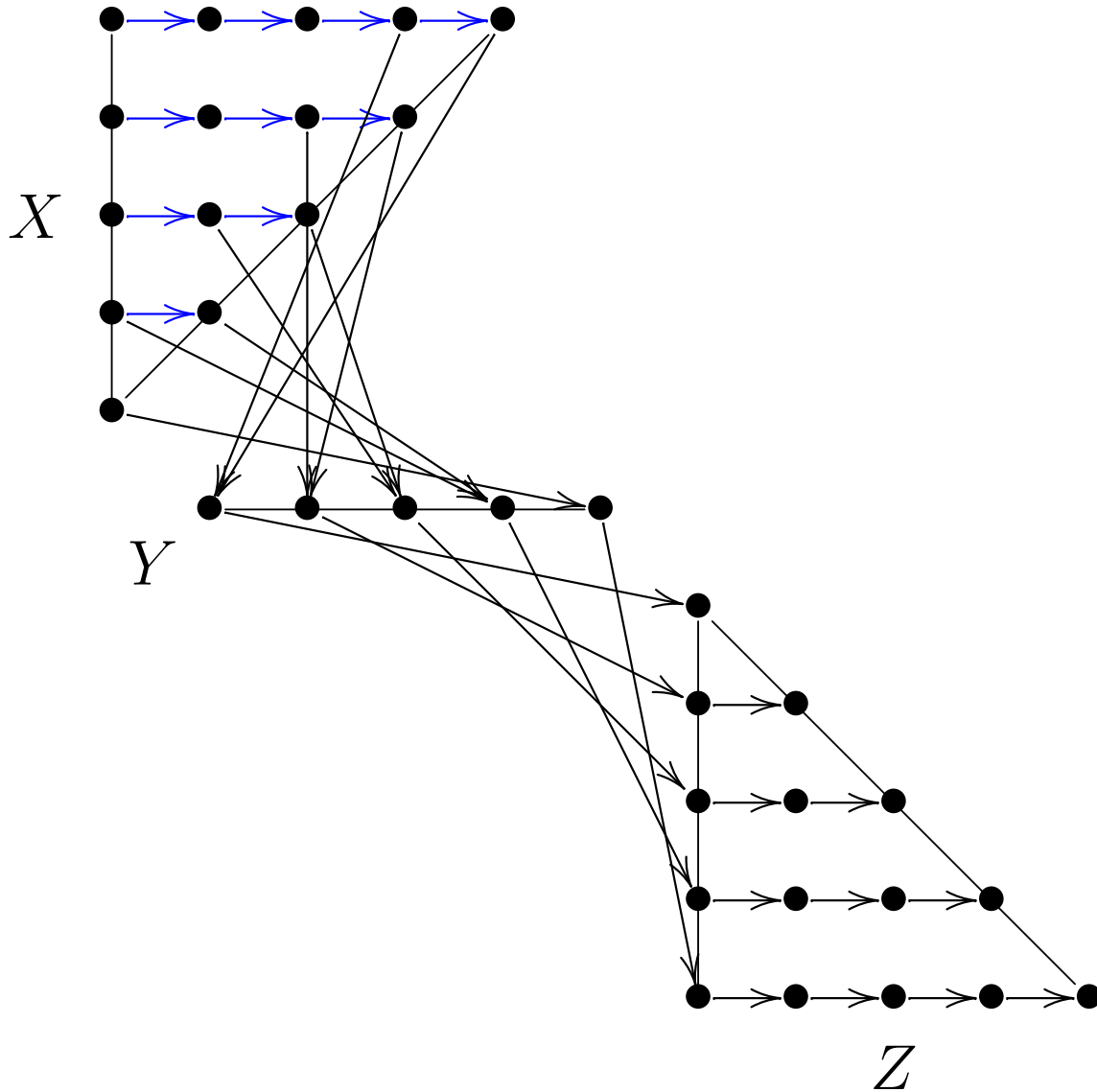
Model Extraction



Linear transformation

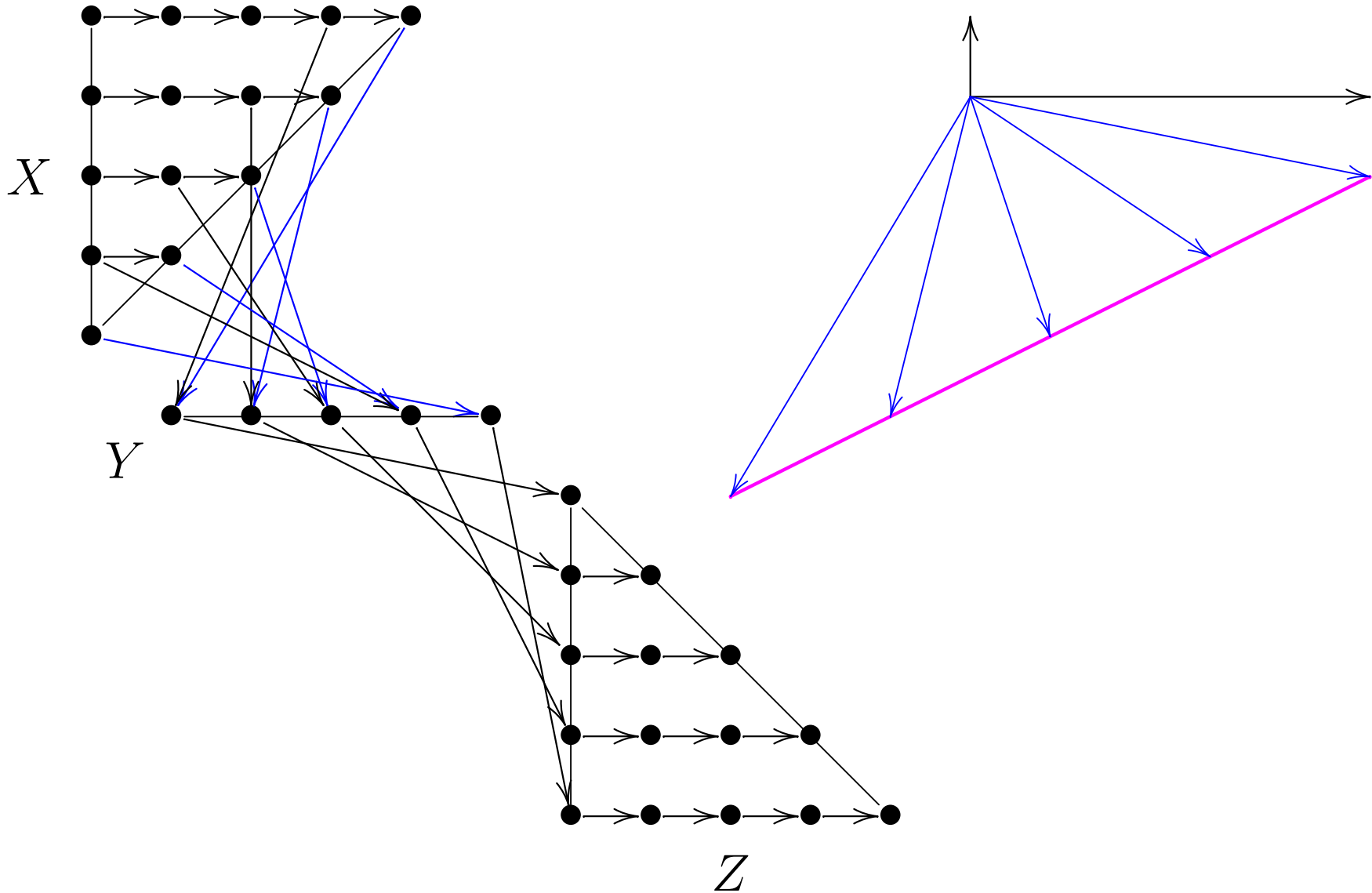


Linear transformation

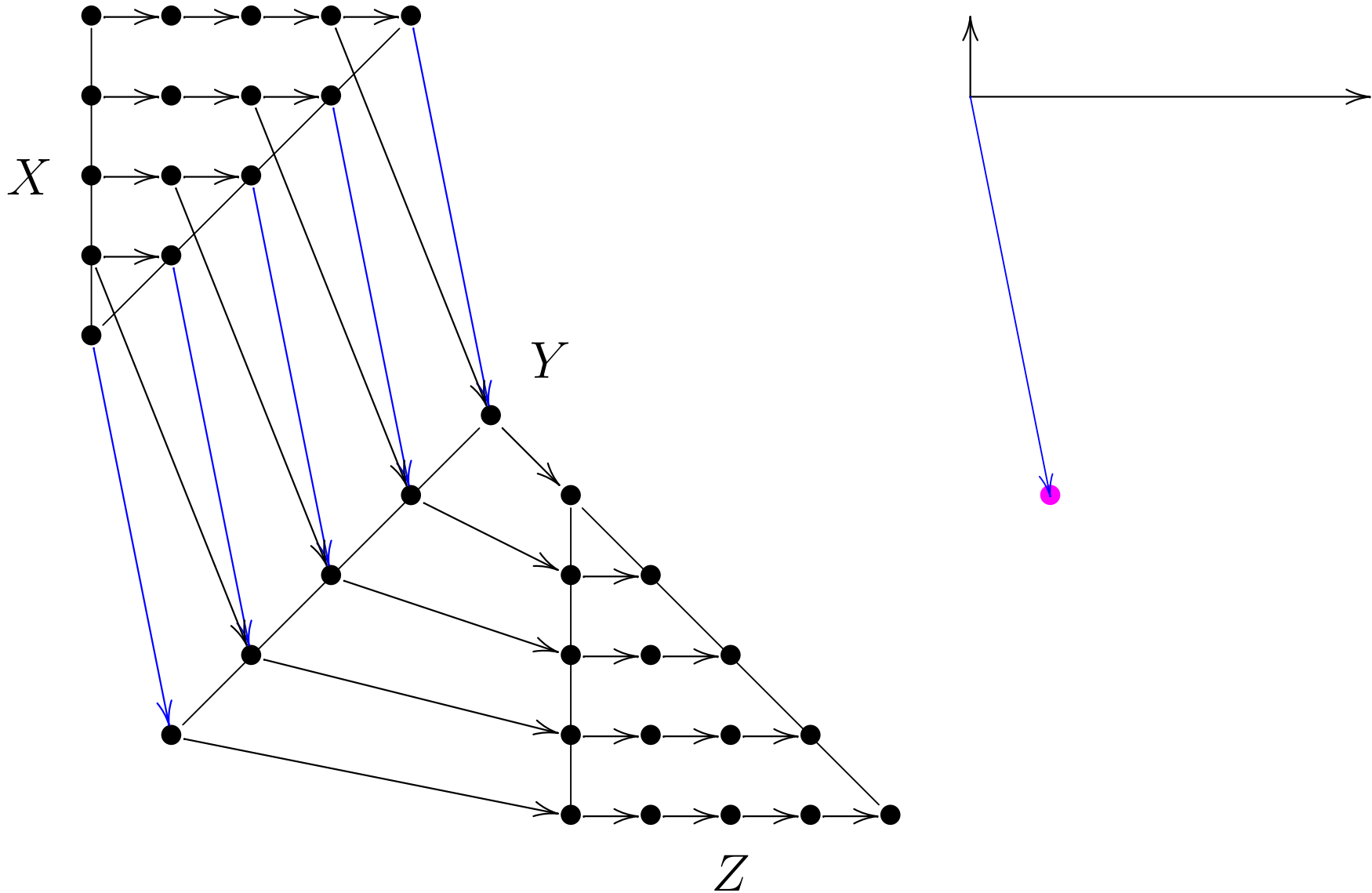


81.5%

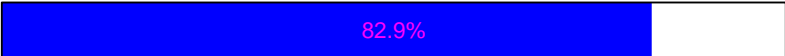
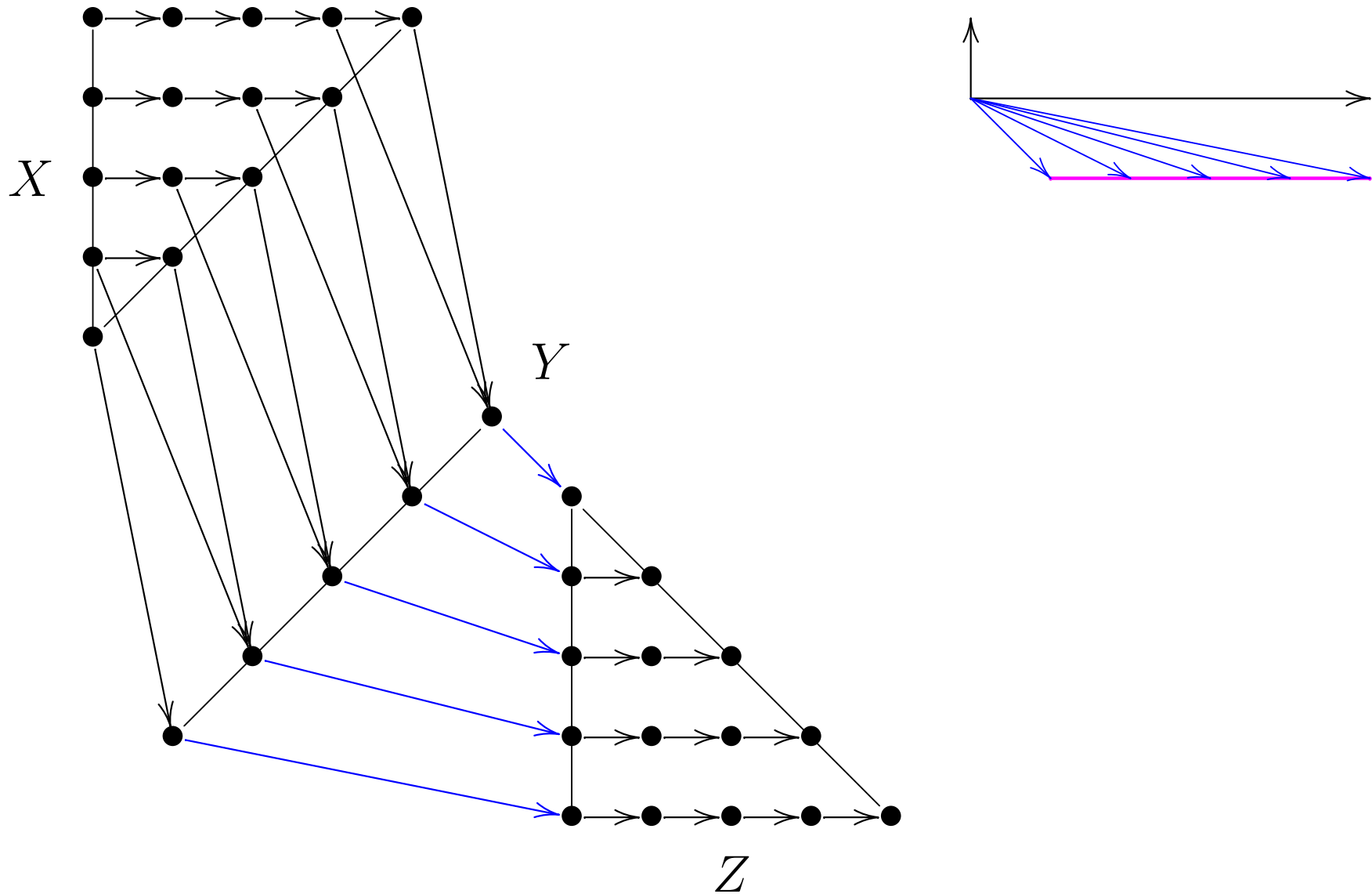
Linear transformation



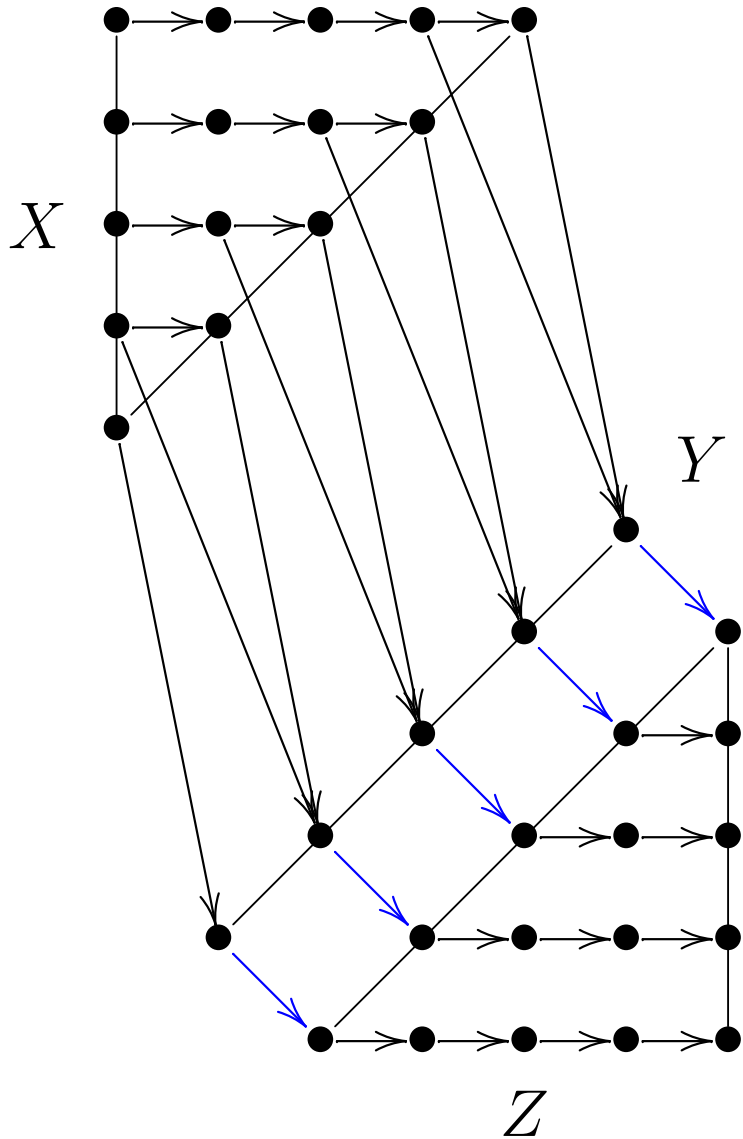
Linear transformation



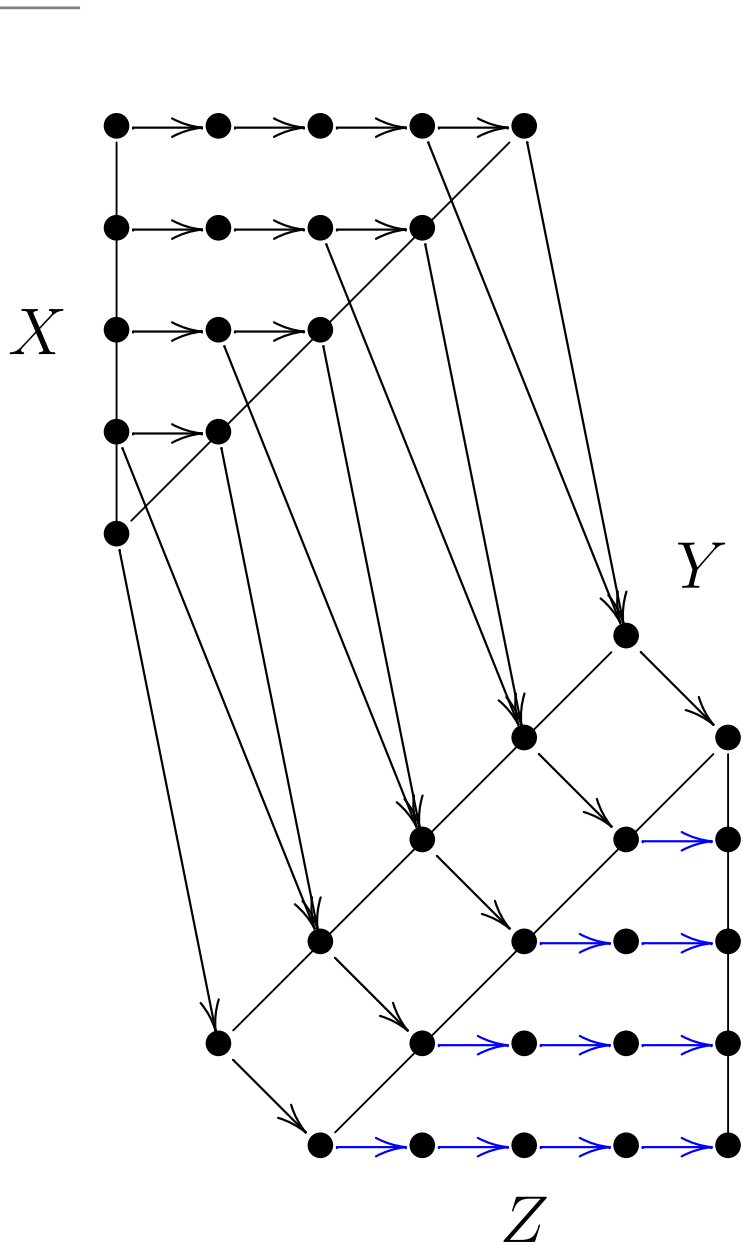
Linear transformation



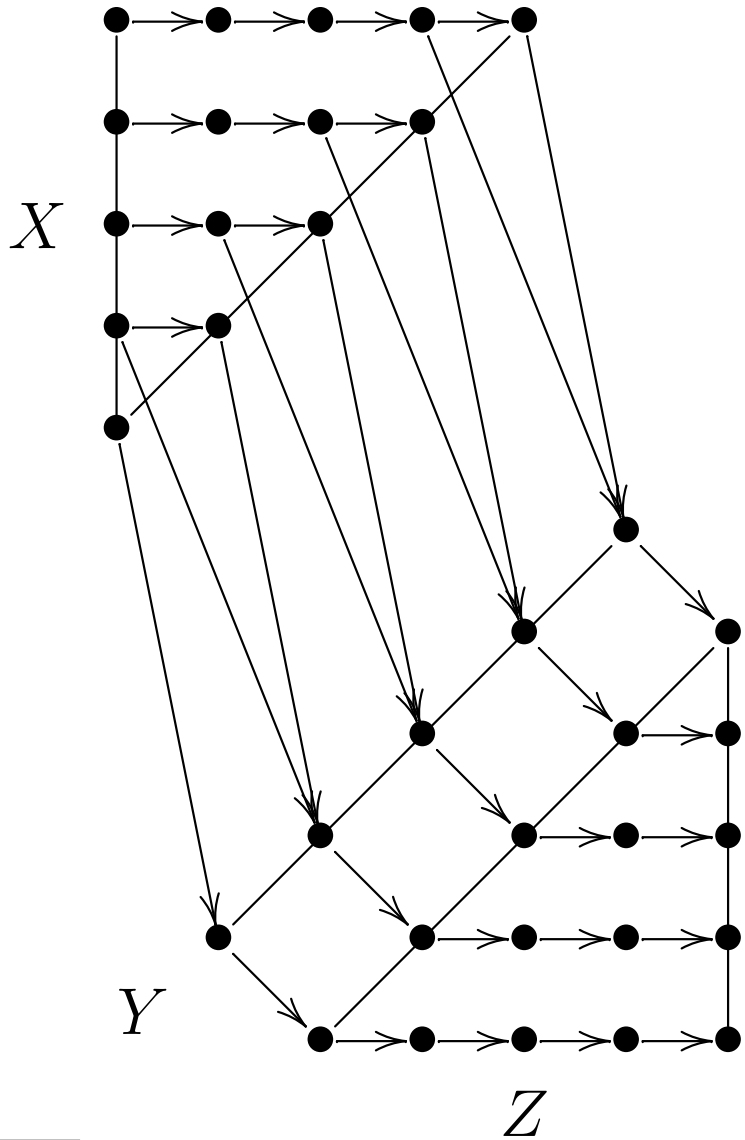
Linear transformation



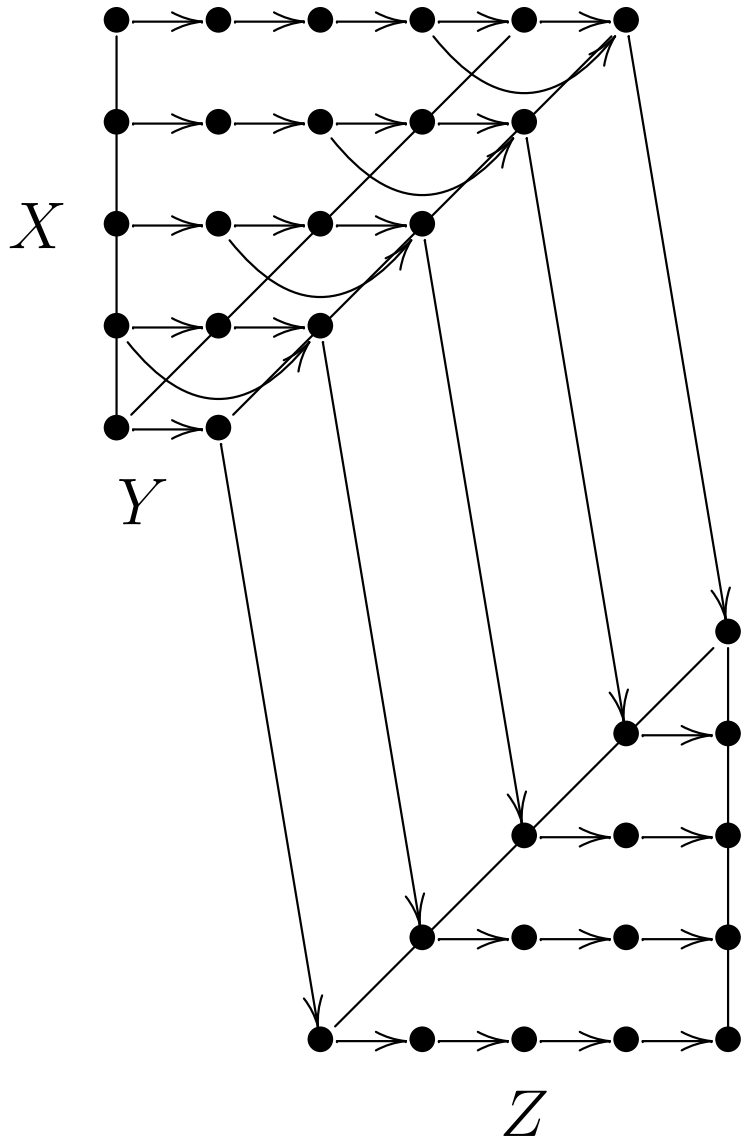
Linear transformation



Translation

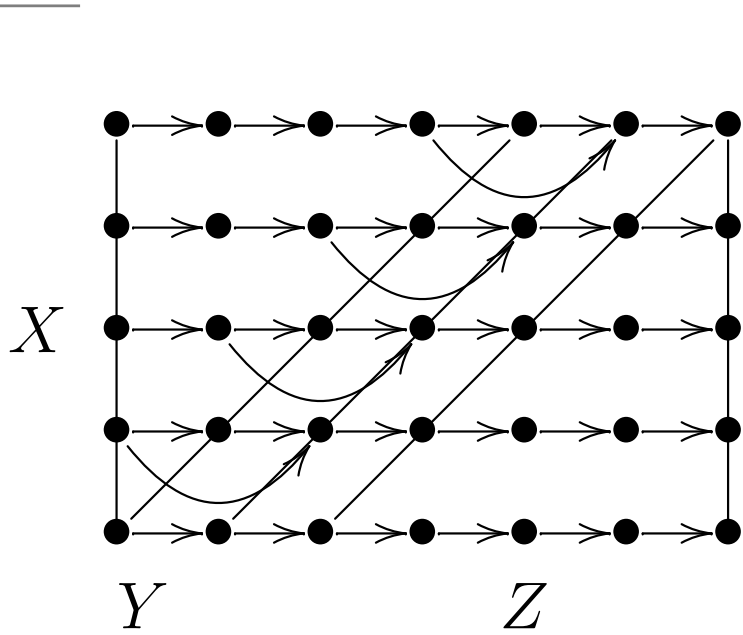


Translation

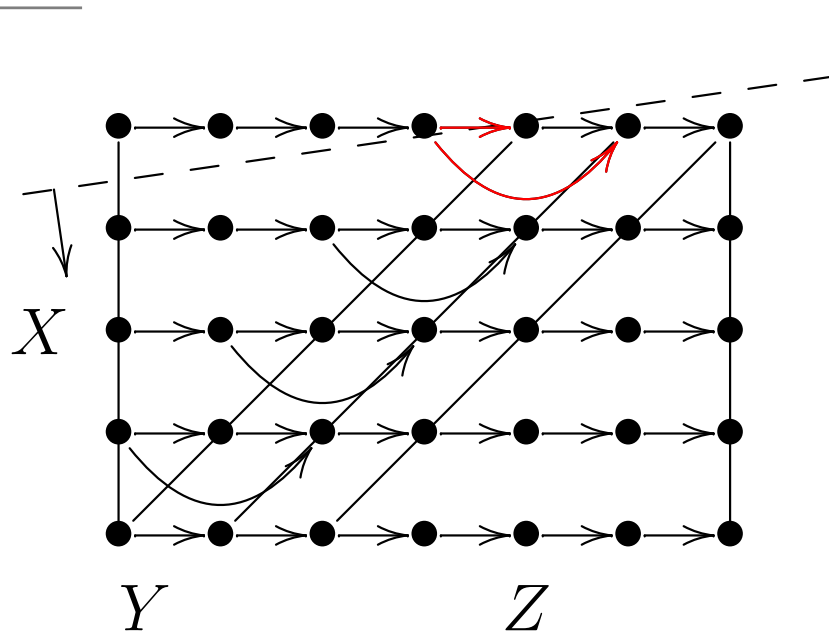


85.1%

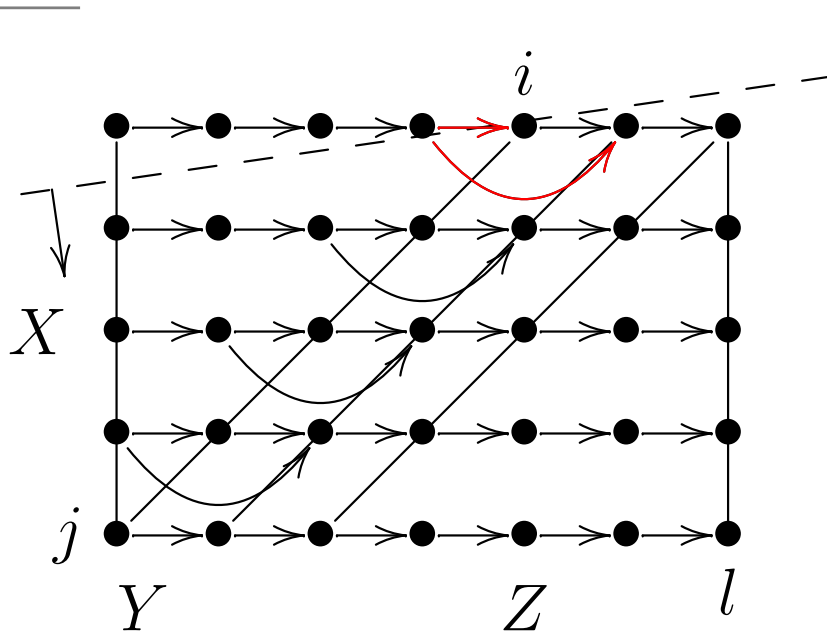
Translation



Translation



Translation



```
for (j=1; j<=N; ++j) {  
  for (i=1; i<=N-j+1; ++i)  
    a[i][j] = in[i][j] + a[i-1][j];  
  b[j][1] = f(a[N-j+1][j], a[N-j][j]);  
  for (l=1; l<=j; ++l)  
    b[j][l+1] = g(b[j][l]);  
}
```

Buffer space required: 2

Overview

- Introduction
- Overview
- Subdivision
- Dependence analysis
- Translation
- Linear placement
- Example
- [Related work](#)
- Conclusions

Related Work: affine-by-statement

- General framework: Kelly and Pugh (1994)
- Optimizing parallelism
 - Darte and Robert (1995)
 - Feautrier (1992) \Rightarrow minimal latency
 - Darte and Vivien (1997) (shifted linear)
 - Lim and Lam (1997) \Rightarrow minimal synchronization
- Optimizing locality
 - Lim et al. (2001)

Main differences:

- Two separate steps
- Use of distance vectors

Related Work: linear phase

● Self-reuse

- Wolf and Lam (1991)
- Kandemir et al. (2001)

Apply a single linear transformation to a single loop nest

● Regularity

- Olsen and Gao (1992)
 - Limited linear transformations per statement
 - Requires perfectly nested loop nests
- van Swaaij (1992)
- Danckaert (2001)
 - Performs an additional ordering step that can be avoided

Related Work: fusion + bumping

- Manjikian and Abdelrahman (1997)
 - Only bumping for legality
 - Acyclic dependence graph
- Fraboulet et al. (1999), Song et al. (2001)
 - Memory requirement
 - (Near) uniform dependences
 - One-shot (min-flow)
- Darte and Huard (2002)
 - Array contraction
 - One-shot (ILP formulation)

All are essentially one-dimensional

Overview

- Introduction
- Overview
- Subdivision
- Dependence analysis
- Translation
- Linear placement
- Example
- Related work
- **Conclusions**

Conclusions

- Much related work
- Two step approach
 - Linear transformation
 - Translation (loop fusion + loop bumping)
- Implementation
 - Dependence analysis
 - Translation step
- Future work:
 - Linear transformation algorithm
 - More/improved cost functions
 - Trade-offs

Unimodular extension

Unimodular extension of $\vec{\pi}^T$:

$$U_{\vec{\pi}} = \begin{bmatrix} \vec{\pi}^T \\ U' \end{bmatrix} \quad |\Delta(U_{\vec{\pi}})| = 1$$

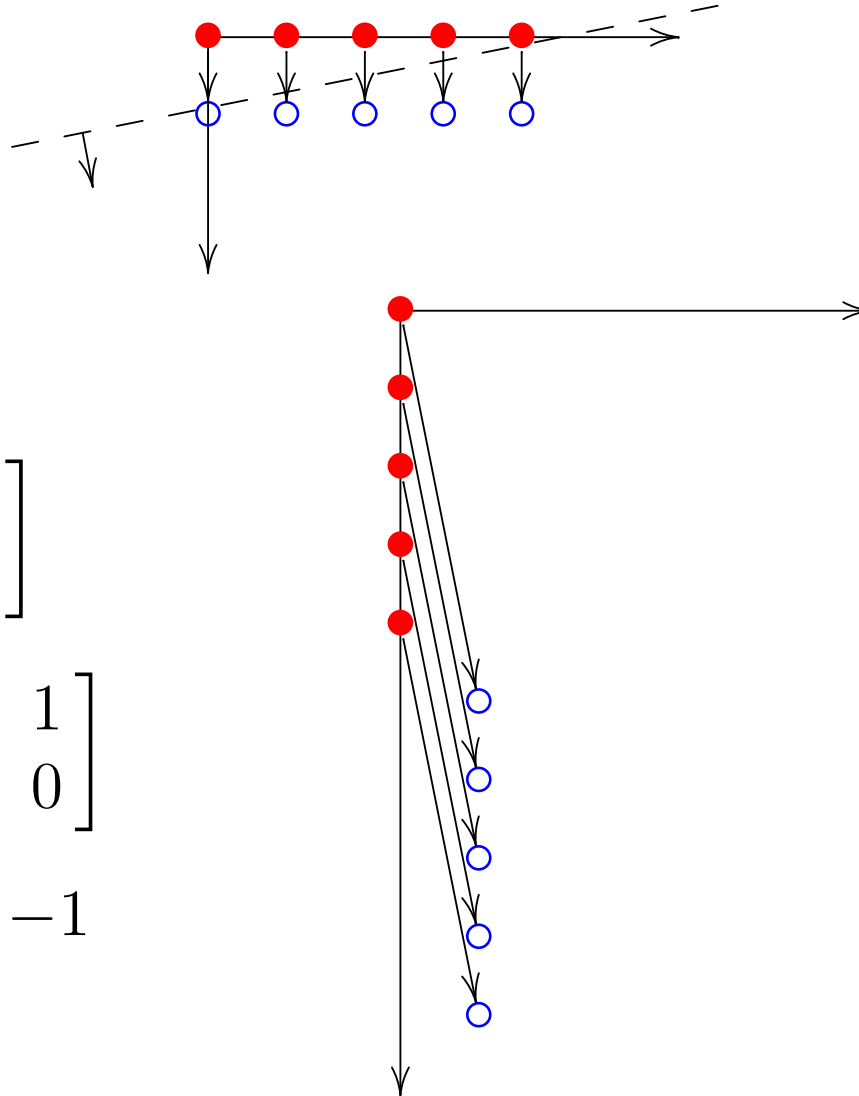
Exists if $\gcd(\vec{\pi}) = 1$; not unique

$$\vec{\pi}^T \vec{a} < \vec{\pi}^T \vec{b} \Rightarrow U_{\vec{\pi}} \vec{a} \prec U_{\vec{\pi}} \vec{b}$$

$$\begin{aligned} & U_{\vec{\pi}}(A_X \vec{i} + \vec{a}_X) \\ &= U_{\vec{\pi}} A_X \vec{i} + U_{\vec{\pi}} \vec{a}_X \\ &= A_X'' \vec{i} + \vec{a}_X'' \end{aligned}$$

\Rightarrow ordering step \equiv linear transformation of common iteration space

Unimodular extension

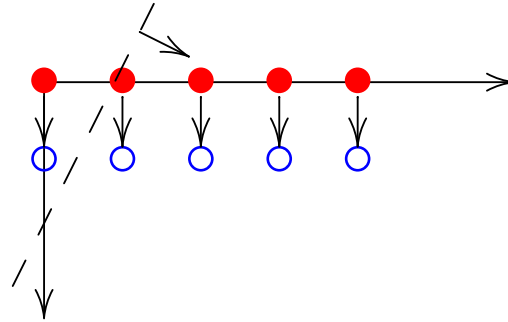


$$\vec{\pi} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$

$$U_{\vec{\pi}} = \begin{bmatrix} 5 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\Delta(U_{\vec{\pi}}) = -1$$

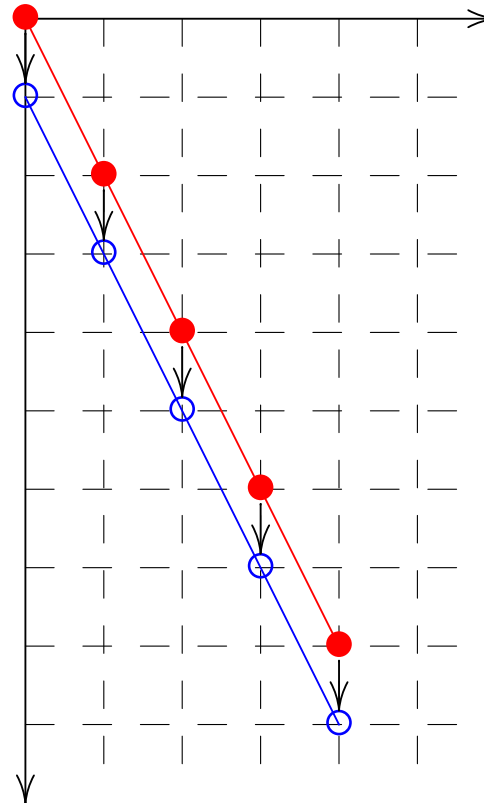
Unimodular extension



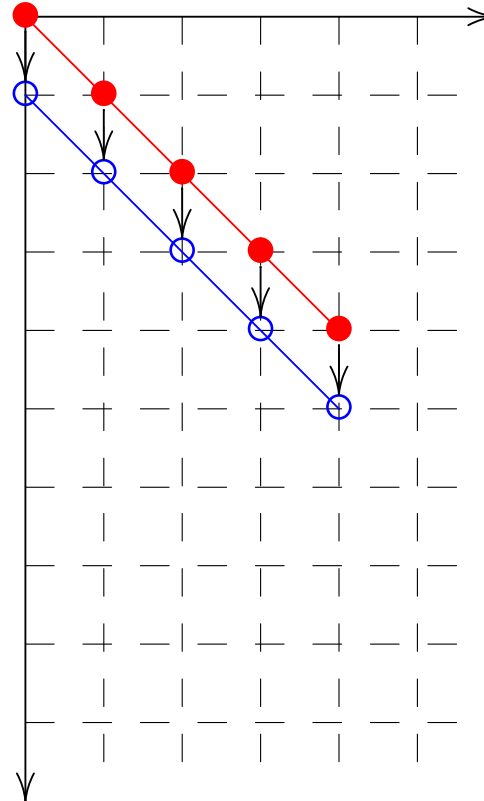
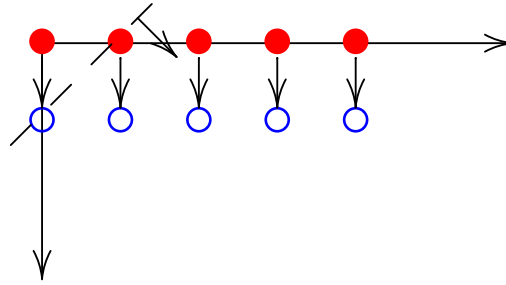
$$\vec{\pi} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$U_{\vec{\pi}} = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$$

$$\Delta(U_{\vec{\pi}}) = 1$$



Unimodular extension



$$\vec{\pi} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$U_{\vec{\pi}} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

$$\Delta(U_{\vec{\pi}}) = 1$$

Why an extra ordering step ?

- Advantages:
 - The complexity is reduced. In each phase, a cost function can be used which is more simple than a combined cost function.
 - It is a new approach compared to existing work. The latter has not led to an automated loop transformation methodology.

(Dixit Koen Danckaert)

Why an extra ordering step ?

- Disadvantages:
 - All transformations can be performed in many different ways, by different combinations of placement and ordering. Future work should investigate whether at least part of this redundancy can be removed.
 - Only heuristic (*i.e., non time-related*) cost functions are possible during the placement, such that potentially a placement could be chosen which does not allow a good ordering (and final solution) anymore.

(Dixit Koen Danckaert)

Finding good mappings

- Greedy search
 - very fast
 - only for trivial cases
- Constraint Satisfaction Problem
 - fast
 - (currently) only for 0D dependences
- Backtracking search
 - can find non truly-optimal placements
 - limited set of mappings
 - set too large for dimensions > 3

Backtracking search

Initialization

- Generate set of linear mappings

$$\mathbb{A} = \{0, 1\}^{n \times n}$$

- Find implicit equalities C_i
- Eliminate redundant mappings

Polytope i : $A_1 R_i A_2 \iff A_1 - A_2 = M \cdot C_i$

$$\mathbb{A}_i = \mathbb{A} / R_i$$

- Calculate optimal dimension

Backtracking search

For each dependency (D_i) between a pair of polytopes:

$$r_{d,D_i} \leq c_i$$

1. $\forall i : c_i = n$ (n : dimension)
2. find solution among \mathbb{A}_i satisfying constraints
3. if insolvable or optimal solution found, stop
4. tighten one constraint
5. goto 2

References

- Danckaert, K. (2001). *Loop transformations for data transfer and storage reduction on multiprocessor systems*. Ph. D. thesis, KU Leuven.
- Darte, A. and G. Huard (2002). New results on array contraction. In *ASAP 2002*, pp. 359–370.
- Darte, A. and Y. Robert (1995). Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. *Journal of Parallel and Distributed Computing* 29(1), 43–59.
- Darte, A. and F. Vivien (1997). Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming* 25(6), 447–496.
- Feautrier, P. (1992, October). Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *International Journal of Parallel Programming* 21(5), 313–348.
- Fraboulet, A., G. Huard, and A. Mignotte (1999, November). Loop Alignment for Memory Accesses Optimization. In *Twelfth International Symposium on System Synthesis Proceedings (ISSS'99)*, pp. 71–77. IEEE Computer Society Press.
- Kandemir, M. T., J. Ramanujam, A. N. Choudhary, and P. Banerjee (2001). A layout-conscious iteration space transformation technique. *IEEE Transactions on Computers* 50(12), 1321–1335.
- Kelly, W. and W. Pugh (1994). Finding legal reordering transformations using mappings. In *Languages and Compilers for Parallel Computing*, pp. 107–124.
- Lim, A. W. and M. S. Lam (1997). Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the Twenty-fourth Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, pp. 201–214. ACM Press.
- Lim, A. W., S.-W. Liao, and M. S. Lam (2001). Blocking and array contraction across arbitrarily nested loops using affine partitioning. *ACM SIGPLAN Notices* 36(7), 103–112.
- Manjikian, N. and T. S. Abdelrahman (1997). Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems* 8(2), 193–209.
- Olsen, R. and G. Gao (1992, April). Collective analysis and transformation of loop clusters. Technical Report ACAPS Technical Memo 24, McGill University.
- Song, Y., R. Xu, C. Wang, and Z. Li (2001). Data locality enhancement by memory reduction. In *International Conference on Supercomputing*, pp. 50–64.
- van Swaaij, M. (1992). *Data flow geometry: exploiting regularity in system-level synthesis*. Ph. D. thesis, KU Leuven.
- Wolf, M. E. and M. S. Lam (1991, June). A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pp. 30–44.