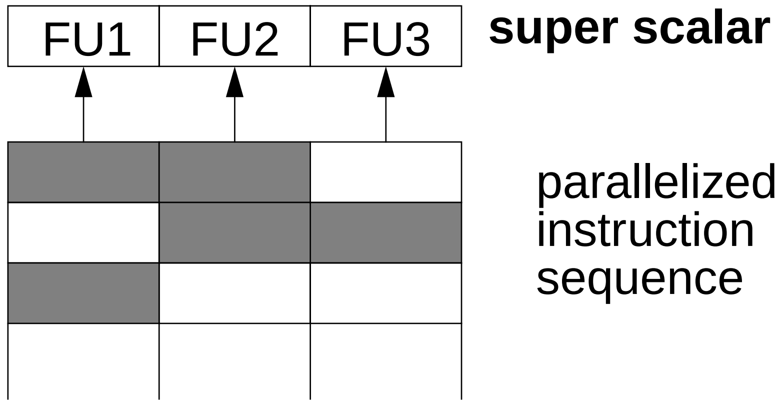
5 Code Parallelization

Processor with **instruction level parallelism (ILP)**
executes several instructions in parallel.

Classes of processors and parallelism:
VLIW, super scalar
Pipelined processors
Data parallel processors

Parallel functional units, VLIW super scalar



Compiler **analyzes sequential programs** to
exhibit potential parallelism
on instruction level;

model **dependences between computations**

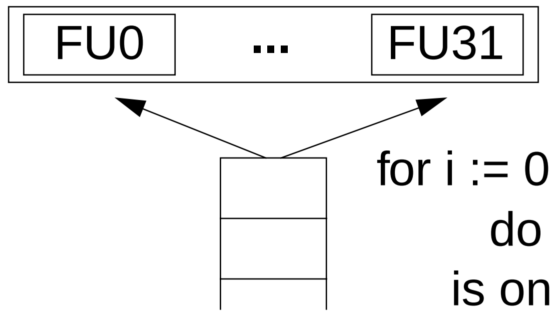
Pipeline processor

sequential code scheduled for pipelining

Compiler arranges instructions for
shortest execution time:
instruction scheduling

Compiler **analyzes loops**
to execute them in parallel
loop transformation
array transformation

Data parallel processor, SIMD



for i := 0 to 31
do c[i] := a[i] + b [i];
is one instruction!

5.1 Instruction Scheduling
Data Dependence Graph

Exhibit potential **fine-grained parallelism** among operations.
Sequential code is over-specified!

Data dependence graph (DDG) for a basic block:

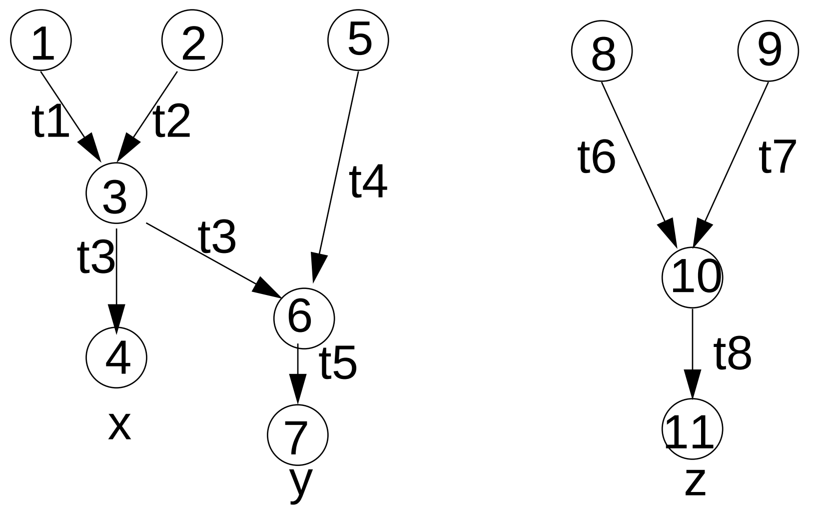
  Node: operation;

  Edge a -> b: operation b uses the result of operation a

Example for a basic block:

1:	t1	:= a
2:	t2	:= b
3:	t3	:= t1 + t2
4:	x	:= t3
5:	t4	:= c
6:	t5	:= t3 + t4
7:	y	:= t5
8:	t6	:= d
9:	t7	:= e
10:	t8	:= t6 + t7
11:	z	:= t8

data dependence graph

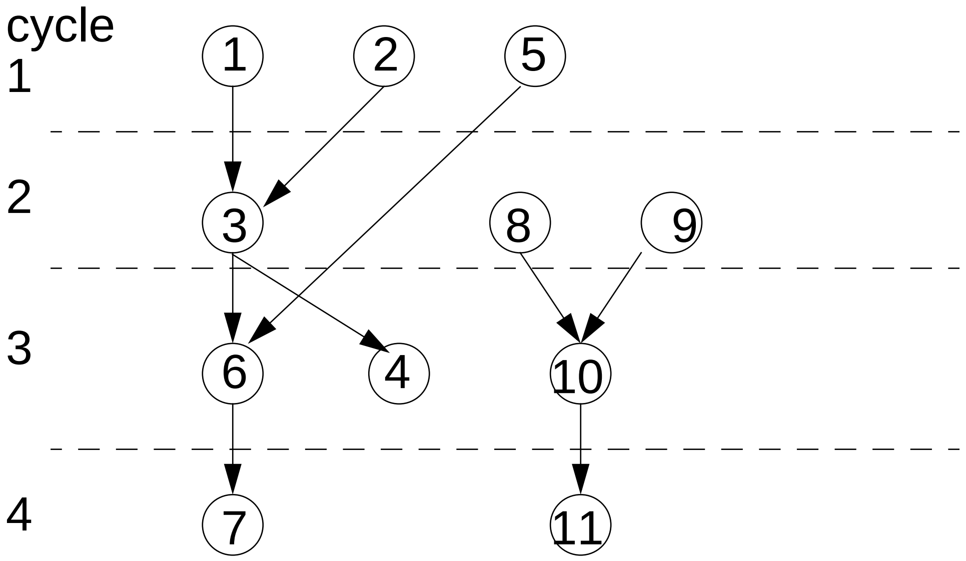


ti are symbolic registers, store intermediate results, obey single assignment rule

List Scheduling

Input: data dependence graph
Output: a schedule of **at most k operations per cycle**, such that all **dependences point forward;** DDG arranged in levels

Algorithm: A **ready list** contains all operations that are **not yet scheduled**, but whose **predecessors are scheduled**
Iterate: **select** from the ready list up to k operations for the next cycle (heuristic), **update** the ready list



- Algorithm is **optimal** only for **trees.**
- **Heuristic**: Keep ready list sorted by distance to an end node, e. g.

  (1 2 5) (8 9 3) (6 10 4) (7 11)

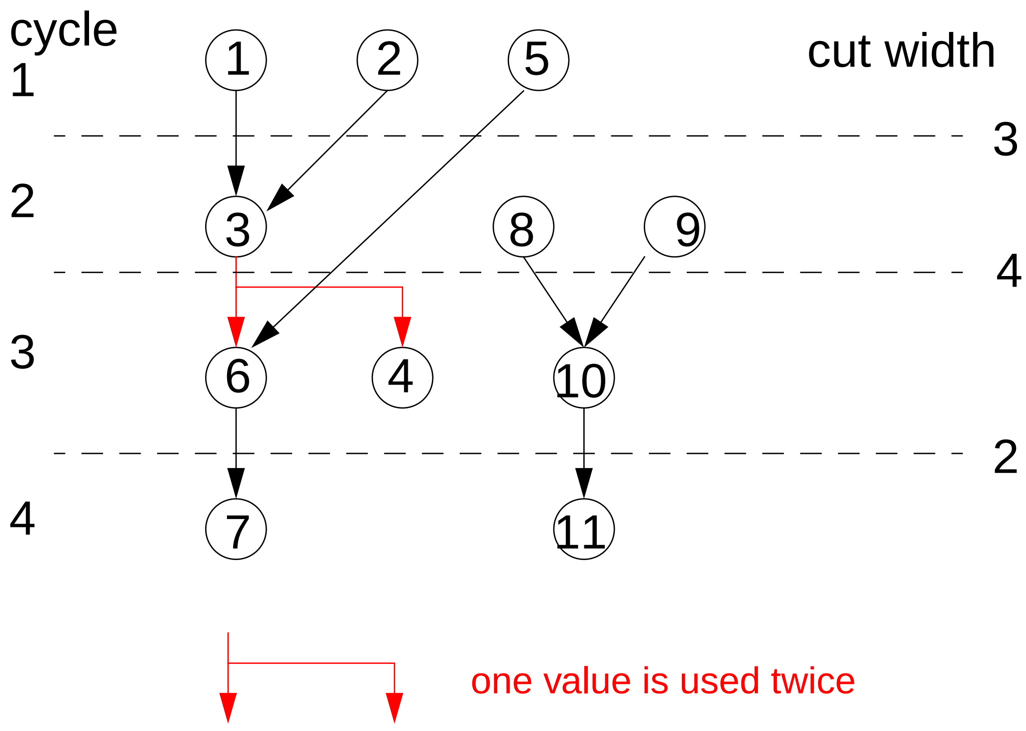
  without this heuristic:
  (1 8 9) (2 5 10) (3 11) (6 4) (7)

  ( ) operations in one cycle

Critical paths determine minimal schedule length: e. g. 1 -> 3 -> 6 -> 7

Variants and Restrictions for List Scheduling

- Allocate **as soon as possible**, ASAP (C-5.3); as **late** as possible, ALAP
- Operations have **unit execution time** (C-5.3); **different execution times:** selection avoids conflicts with already allocated operations
- Operations only on **specific functional units** (e. g. 2 int FUs, 2 float FUs)
- **Resource restrictions** between operations, e. g. <= 1 load or store per cycle



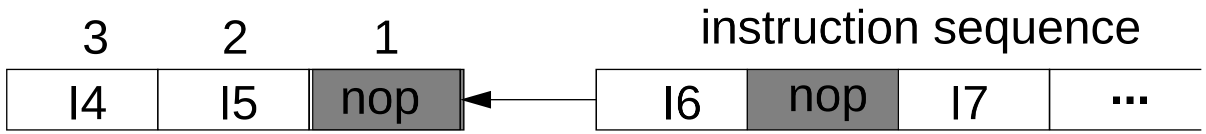
Scheduled DDG models **number of needed registers**:

- arc represents the use of an intermediate result
- **cut width** through a level gives the number of **registers needed**

The tighter the schedule the more registers are needed (*register pressure*).

Instruction Scheduling for Pipelining

Instruction pipeline with 3 stages:



Dependent instructions may not follow one another immediately.

without scheduling:

1:	t1	:= a
2:	t2	:= b
	nop	
3:	t3	:= t1 + t2
	nop	
4:	x	:= t3
5:	t4	:= c
	nop	
6:	t5	:= t3 + t4
	nop	
7:	y	:= t5
8:	t6	:= d
9:	t7	:= e
	nop	
10:	t8	:= t6 + t7
	nop	
11:	z	:= t8

Schedule rearranges the operation sequence, to minimize the number of delays:

1:	t1	:= a	
2:	t2	:= b	
5:	t4	:= c	
3:	t3	:= t1 + t2	**with**
8:	t6	:= d	**scheduling**
9:	t7	:= e	
6:	t5	:= t3 + t4	**no delays**
10:	t8	:= t6 + t7	
4:	x	:= t3	
7:	y	:= t5	
11:	z	:= t8	

Instruction Scheduling Algorithm for Pipelining

Algorithm: modified list scheduling:

Select from the ready list such that the selected operation

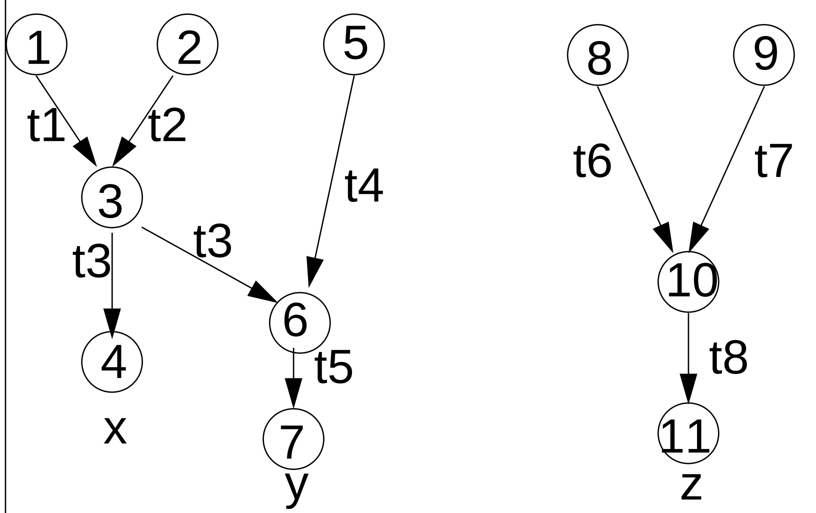
- has a sufficient **distance to all predecessors** in DDG
- has **many successors** (heuristic)
- has a **long path to the end** node (heuristic)

Insert an empty operation if none is selectable.

Ready list with additional information:

opr.	1	2	5	8	9	3	6	4	10	7	11
succ #	1	1	1	1	1	2	1	0	1	0	0
to end	3	3	2	2	2	2	1	1	1	0	0
sched. cycle	1	2	3	5	6	4	7	9	8	10	11

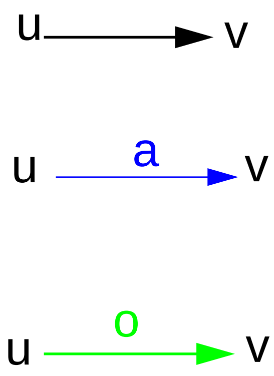
data dependence graph



cycle			
1	1:	t1	:= a
2	2:	t2	:= b
3	5:	t4	:= c
4	3:	t3	:= t1 + t2
5	8:	t6	:= d
6	9:	t7	:= e
7	6:	t5	:= t3 + t4
8	10:	t8	:= t6 + t7
9	4:	x	:= t3
10	7:	y	:= t5
11	11:	z	:= t8

with scheduling

Reused registers: anti- and output-dependences

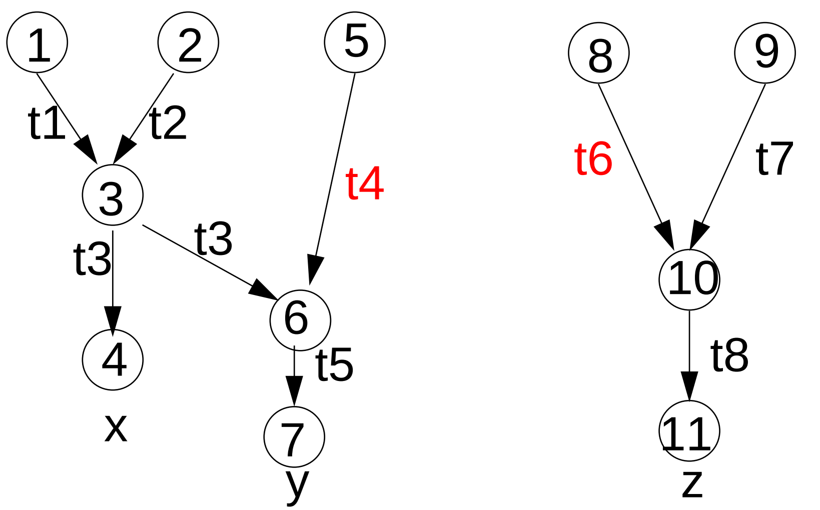


flow-dependence:
u writes before v uses

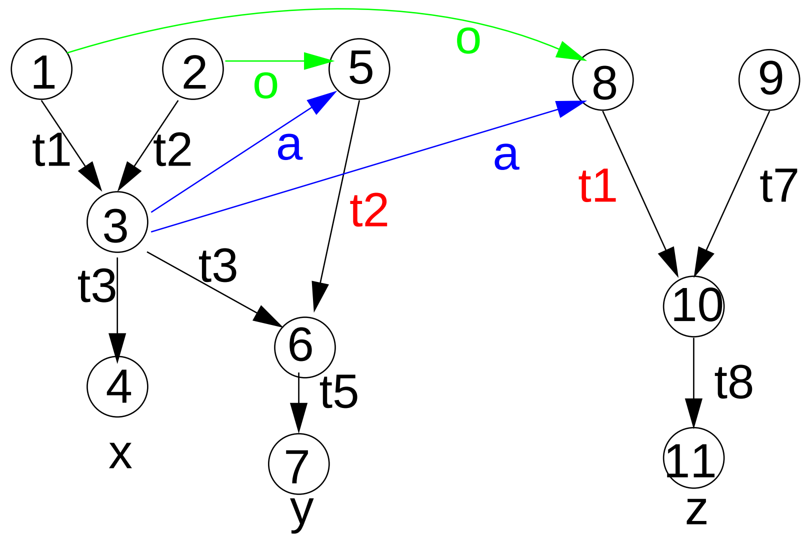
anti-dependence:
u uses a value
before v overwrites it

output-dependence:
u writes before v overwrites

DDG with symbolic registers ti
flow-dependences only



DDG with reused registers ti
flow, anti-, and output-dependences

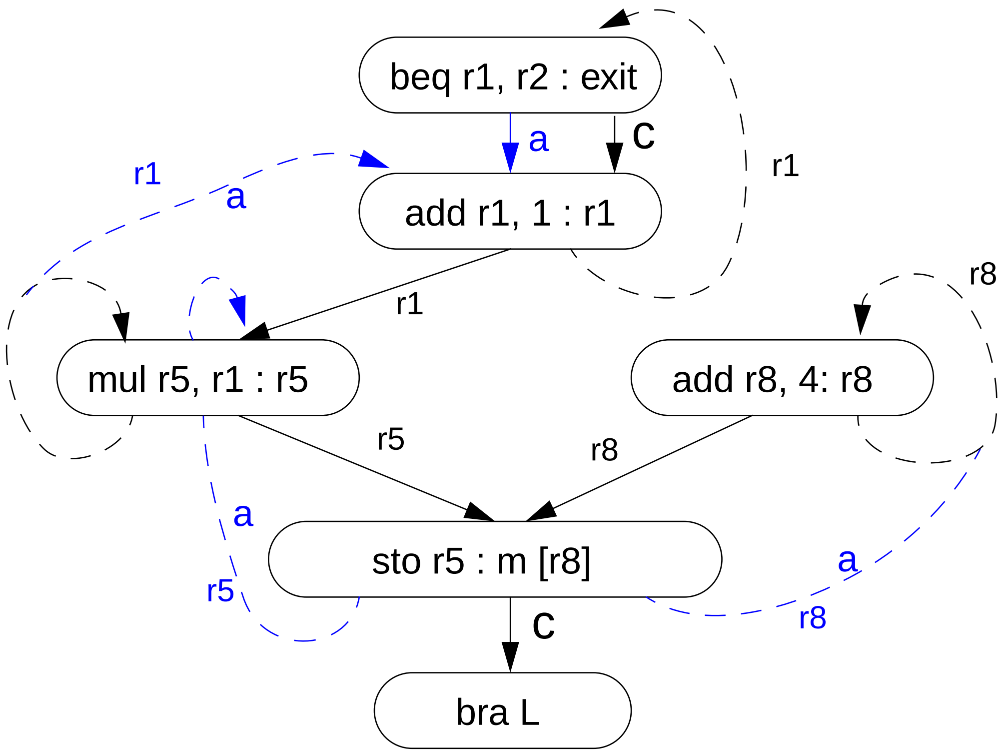


DDG with Loop Carried Dependences

Factorial computation:

program:	**seq. machine code:**
i = 0; f = 1;	
while (i != n)	L: beq r1, r2 : exit
{ i = i + 1;	add r1, 1 : r1
f = f * i;	mul r5, r1 : r5
	add r8, 4 : r8
m[i] = f;	sto r5 : m[r8]
}	bra L

Data dependence graph:



u ——► v **flow-dependence**: u writes before v uses

u - - - ► v **flow-dependence** into subsequent iteration

u - - a - ► v **anti-dependence**: u uses a value before v overwrites it

u - - o - ► v **output-dependence**: u writes before v overwrites

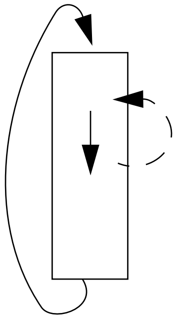
u —c—► v **control-dependence**: u has to be executed before v (u or v may branch)

Loop unrolling

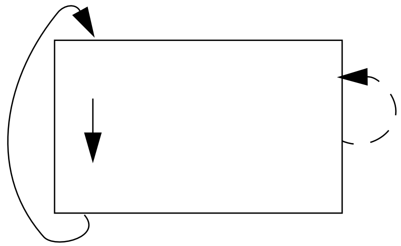
Loop unrolling: A technique for parallelization of loops.

A single loop body does not exhibit enough parallelism => sparse schedule.
Schedule the code (copies) of several adjacent iterations together
=> more compact schedule

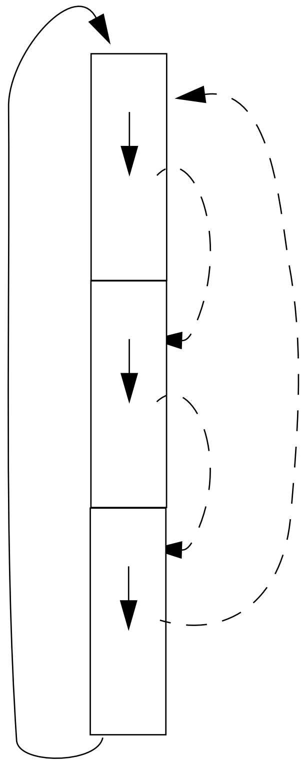
sequential loop



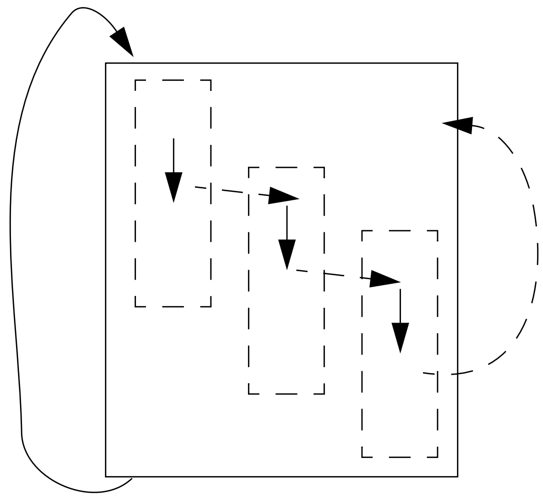
parallel schedule for single body



unrolled loop (3 times)



parallel schedule for unrolled loop



Prologue and epilogue needed to take care of iteration numbers that are not multiples of the unroll factor

Software Pipelining

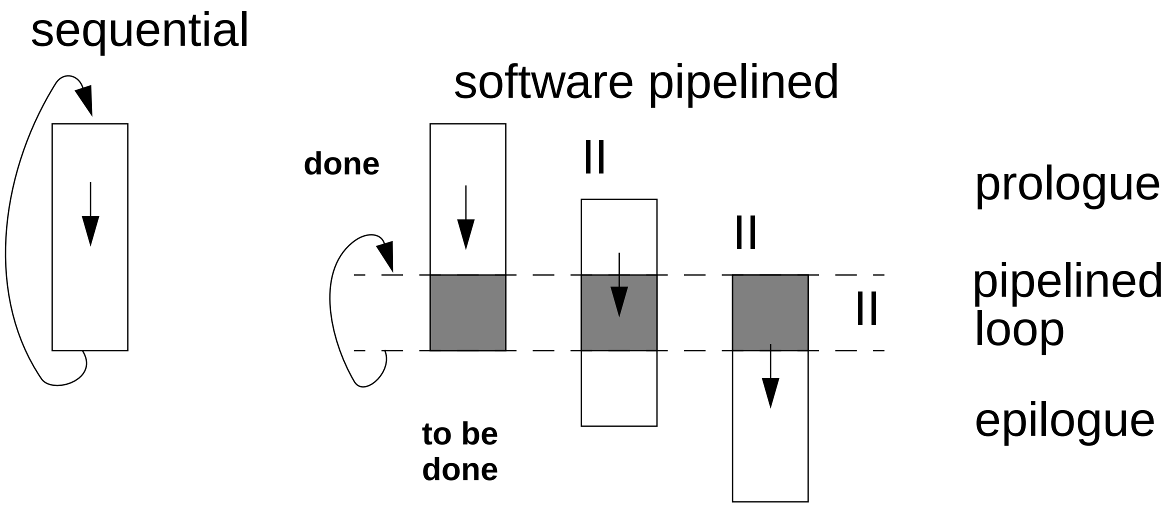
Software Pipelining: A technique for parallelization of loops.

A single loop body does not exhibit enough parallelism => sparse schedule.
Overlap the execution of several adjacent iterations => compact schedule

The **pipelined loop body**

has **each operation** of the original sequential body,
they belong to **several iterations**,
they are **tightly scheduled,**
its length is the **initiation interval II**,
is **shorter** than the original body.

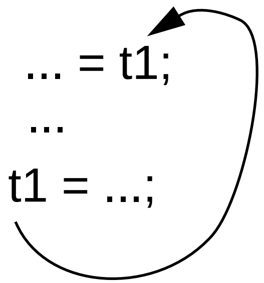
Prologue, epilogue: initiation and finalization code



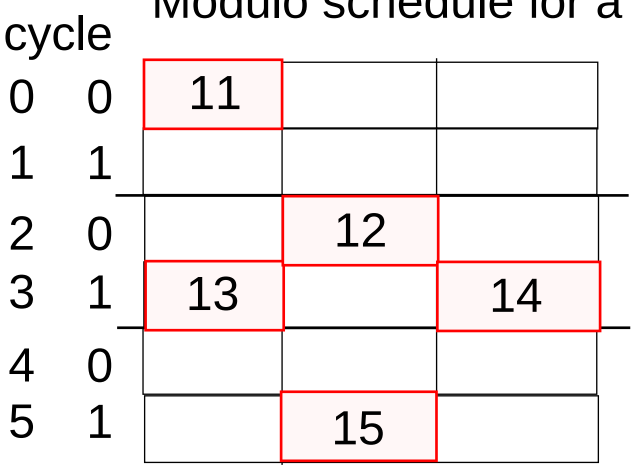
Transform Loops by Software Pipelining

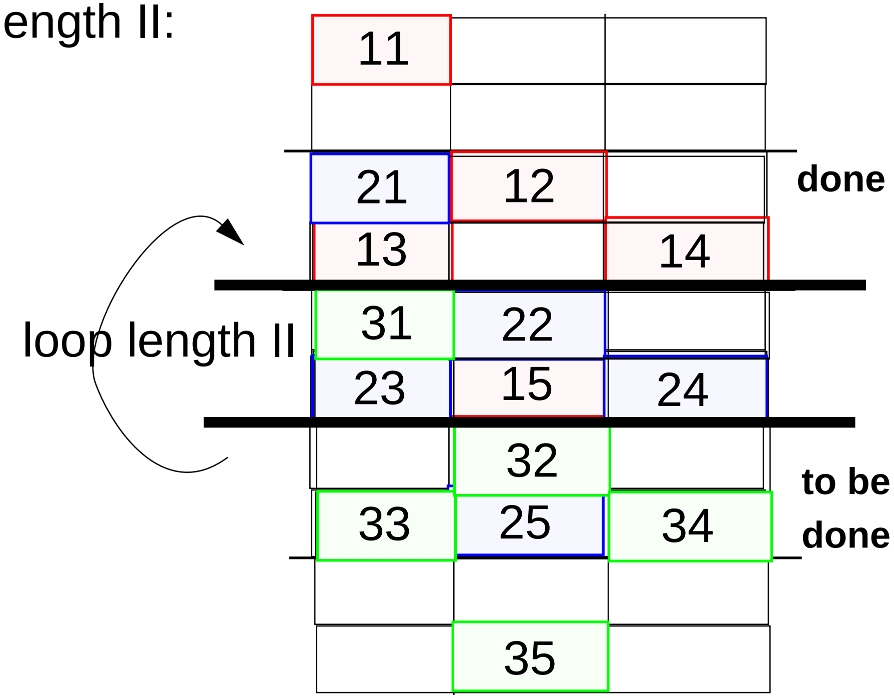
Technique:

1. **Data dependence graph** for the loop body, include **loop carried dependences**.
2. Chose a **small initiation interval II** - not smaller than #instructions / #FUs
3. Make a **„Modulo Schedule“** s for the loop body: Two instructions can not be scheduled on the same FU, i_1 in cycle c_1 and i_2 in cycle c_2, if c_1 mod II = c_2 mod II
4. If (3) does not succeed without conflict, increase II and repeat from 3
5. Allocate the instructions of s in the new loop of length II: i_j scheduled in cycle c_j is allocated to c_j mod II
6. Construct prologue and epilogue.



Modulo schedule for a loop body





Result of Software Pipelining

t	t_m		ADD	MUL	MEM	CTR
0	0	L:				beq r1, r2:exit
1	1		add r1, 1: r1			
2	0		add r8, 4 : r8	mul r5, r1 : r5		
3	1			... mul		
4	0				sto r5 : m r8	
5	1				... sto	
6	0					
7	1					bra L

4 dedicated FUs schedule of the loop body for II = 2

mul and sto need 2 cycles

add and sto in t_m=0, sto reads r8 before add writes it

bra not in cycle 6, it collides with beq: t_m=0

t	t_m		ADD	MUL	MEM	CTR
0	0					beq r1;r2:exit
1	1		add r1, 1 : r1			
2	0		add r8, 4 : r8	mul r5, r1 : r5		beq r1; r2 : ex
3	1		add r1, 1 : r1	... mul		
4	0	L:	add r8, 4 : r8	mul r5, r1 : r5	sto r5 : m r8	beq r1; r2 : ex
5	1		add r1, 1 : r1	... mul	... sto	bra L
6	1	ex:		... mul	... sto	
7	0				sto r5 : m r8	
8	1				... sto	
9	0					bra exit

prologue

software pipline with II = 2

epilogue

5.2 / 6. Data Parallelism: Loop Parallelization

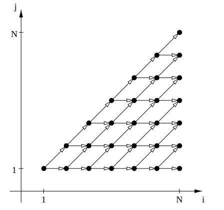
Regular loops on orthogonal data structures - parallelized for **data parallel** processors

Development steps (automated by compilers):

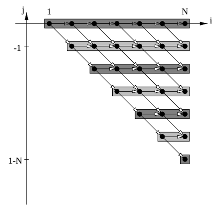
- **nested loops** operating on **arrays**, sequential execution of iteration space

```
DECLARE B[0..N,0..N+1]

FOR I := 1 ..N
    FOR J := 1 .. I
        B[I,J] :=
            B[I-1,J]+B[I-1,J-1]
    END FOR
END FOR
```

- analyze **data dependences** data-flow: definition and use of array elements

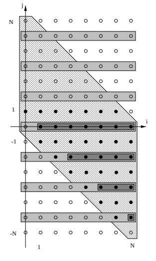


- **transform loops** keep data dependences forward in time



- **parallelize inner loop(s)** map to field or vector of processors

- **map arrays to processors** such that many accesses are local, transform index spaces



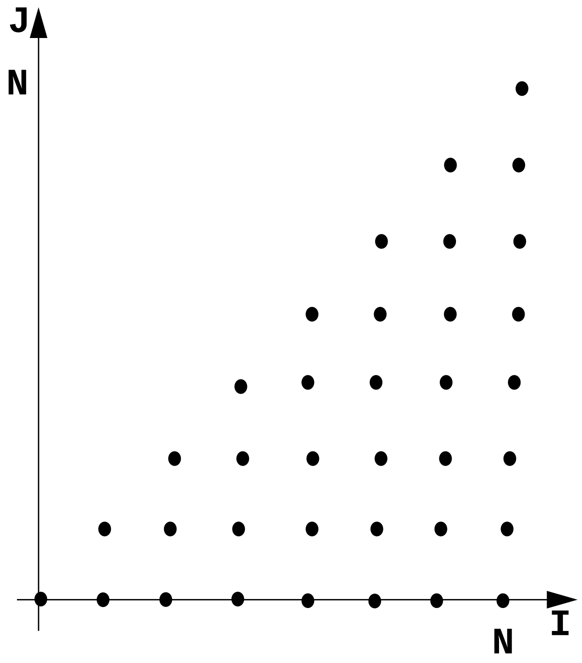
Iteration space of loop nests

Iteration space of a loop nest of depth n:

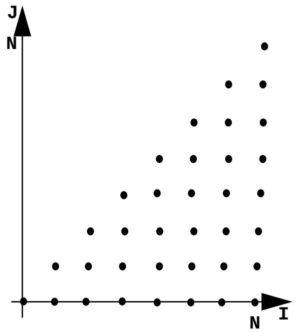
- **n-dimensional space of integral points** (polytope)
- each point (i_1, ..., i_n) represents an execution of the innermost loop body
- loop bounds are in general not known before run-time
- iteration need not have orthogonal borders
- iteration is elaborated sequentially

example:
computation of Pascal’s triangle

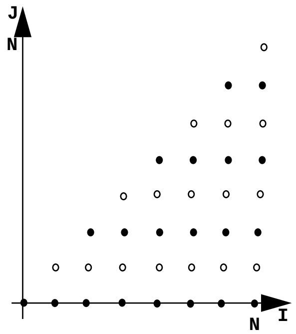
```
DECLARE B[-1..N,-1..N]

FOR I := 0 .. N
   FOR J := 0 .. I
      B[I,J] :=
         B[I-1,J]+B[I-1,J-1]
   END FOR
END FOR
```



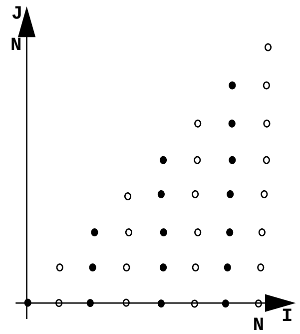
Examples for Iteration spaces of loop nests



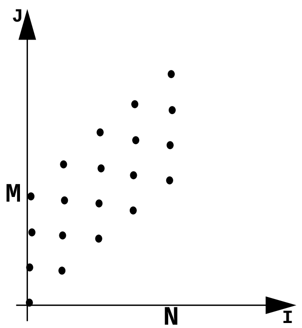

```
FOR I := 0 .. N
   FOR J := 0 .. I
```



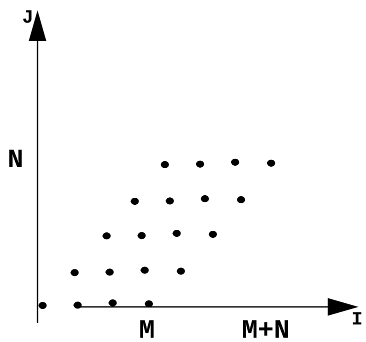

```
FOR I := 0 .. N
   FOR J := 0..I BY 2
```




```
FOR I := 0..N BY 2
   FOR J := 0 .. I
```




```
FOR I := 0 .. N
   FOR J := I..I+M

M = 3, N = 4
```

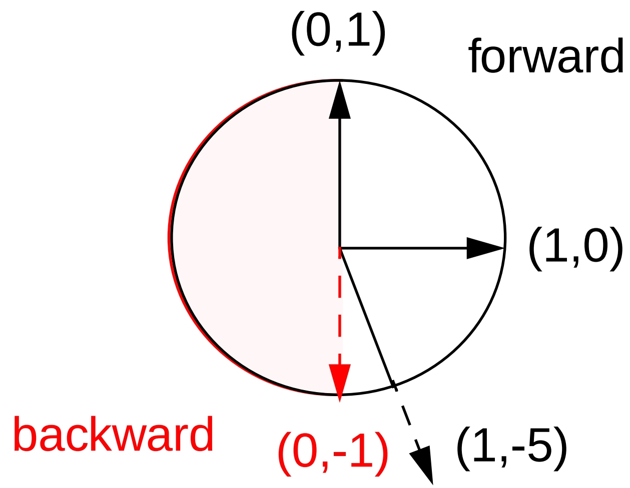



```
FOR I := 0 .. M+N
   FOR J := max(0, I-M)..
            min (I, N)
```

Data Dependences in Iteration Spaces

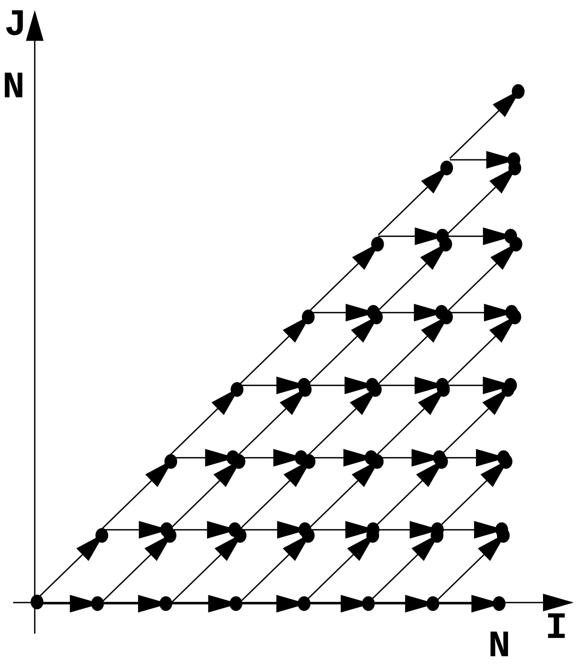
Data dependence from iteration point i1 to i2:

- Iteration **i**1 computes a value that is used in iteration **i2** (flow dependence)
- relative **dependence vector**
  d = **i2 - i1** = $(i2_1 - i1_1, ..., i2_n - i1_n)$
  holds for all iteration points except at the border
- Flow-dependences can **not be directed against the execution order**, can not point backward in time: each dependence vector must be **lexicographically positive**, i. e. **d** = $(0, ..., 0, d_i, ...)$, $d_i > 0$



Example:
Computation of Pascal´s triangle

```
DECLARE B[-1..N,-1..N]

FOR I := 0 .. N
   FOR J := 0 .. I
      B[I,J] :=
         B[I-1,J]+B[I-1,J-1]
   END FOR
END FOR
```



Loop Transformation

The **iteration space** of a loop nest is transformed to **new coordinates**. Goals:

- **execute innermost loop(s) in parallel**
- improve **locality** of data accesses;
  in space: use storage of executing processor,
  in time: reuse values stored in cache
- **systolic** computation and communication scheme

Data dependences must **point forward in time**, i.e.
lexicographically positive and
not within parallel dimensions

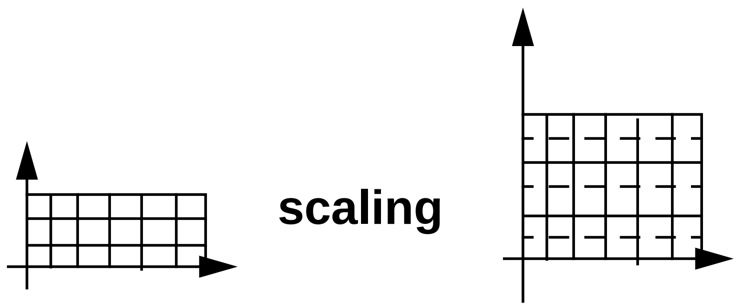
linear basic transformations:

- **Skewing**: add iteration count of an outer loop to that of an inner one
- **Reversal**: flip execution order for one dimension
- **Permutation**: exchange two loops of the loop nest

SRP transformations (next slides)

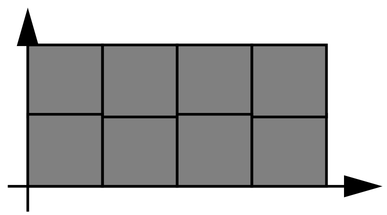
non-linear transformations, e. g.

- **Scaling**: **stretch the iteration space** in one dimension, causes gaps
- **Tiling**: introduce **additional inner loops** that **cover tiles** of fixed size



scaling

tiling

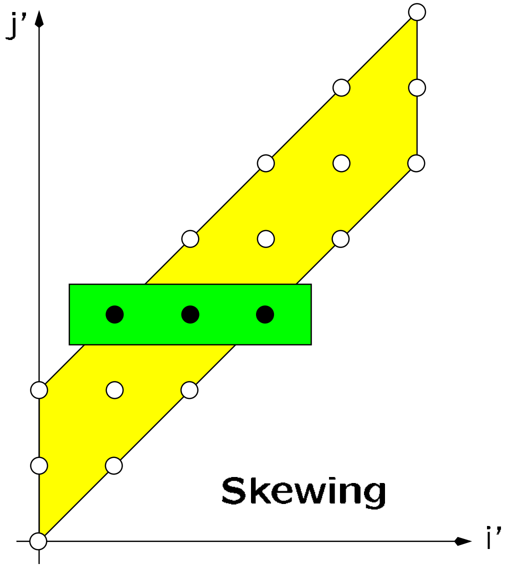


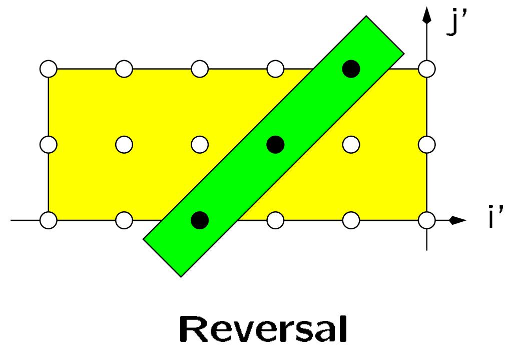
Transformations of

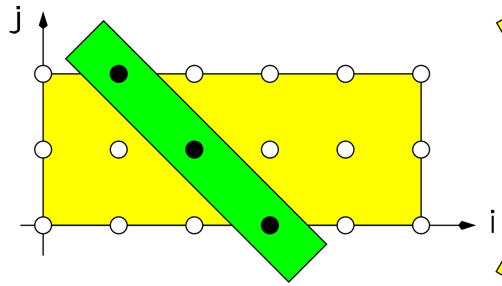
data

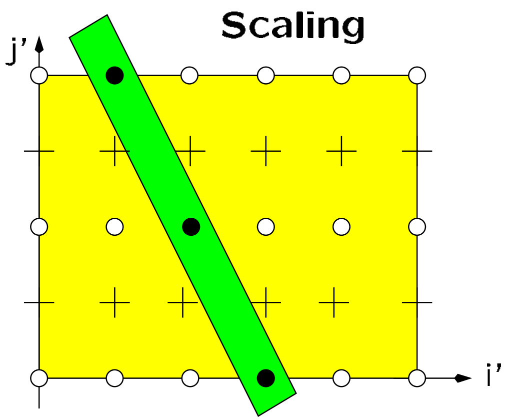
```
REAL B(1:n, 0:m)
```

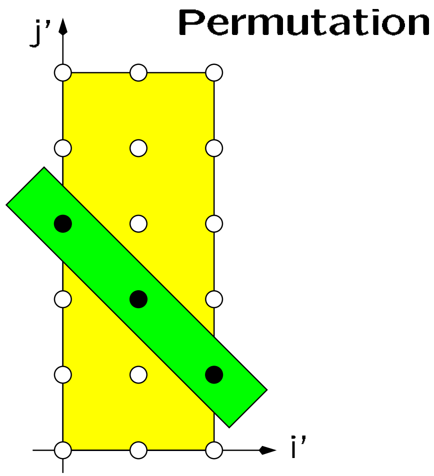
convex polytope












```
DO i = 0, m-1
   DO j = 0, k-1
      ...
   END
END
```

loop nests

Transformations defined by matrices

Transformation matrices: systematic transformation, check dependence vectors

Reversal

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

Skewing

$$\begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ f*i+j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

Permutation

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

Reversal

Iteration count of one loop is negated, that dimension is enumerated backward

general transformation matrix

$$\begin{pmatrix} 1 & & & & & 0 \\ & \ddots & & & & \\ & & 1 & & & \\ & & & \mathbf{-1} & & \\ & & & & 1 & \\ 0 & & & & & \ddots \\ & & & & & & 1 \end{pmatrix}$$

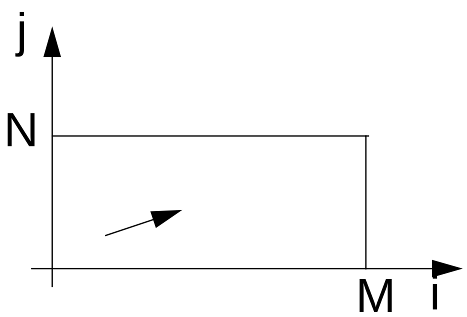
2-dimensional:

loop variables: old, new

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix} = \begin{pmatrix} ir \\ jr \end{pmatrix}$$

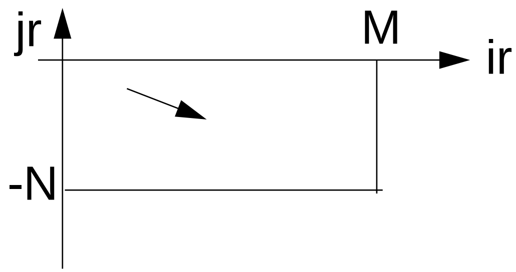
```
for i = 0 to M
   for j = 0 to N
      ...
```

```
for ir = 0 to M
   for jr = -N to 0
      ...
```



original

transformed



Skewing

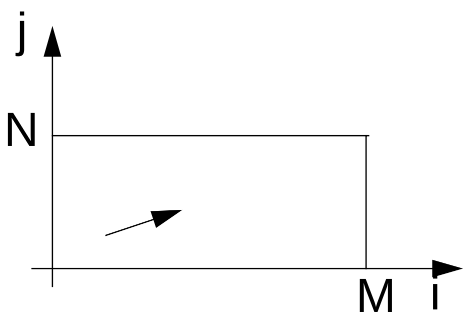
The **iteration count** of an outer loop is **added to the count of an inner loop;** iteration space is shifted; **execution order** of iteration points **remains unchanged**

general transformation matrix:

$$\begin{pmatrix} 1 & & & & & & 0 \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & \mathbf{f} & 1 & & & \\ & & & & 1 & & \\ 0 & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

```
for i = 0 to M
   for j = 0 to N
      ...
```

original



2-dimensional:

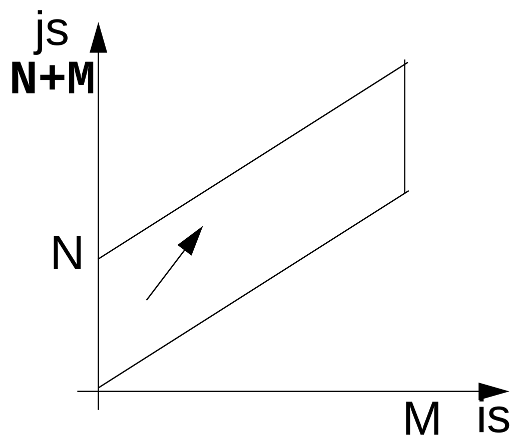
loop variables

old new

$$\begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ f*i+j \end{pmatrix} = \begin{pmatrix} is \\ js \end{pmatrix}$$

```
for is = 0 to M
   for js = f*is to N+f*is
      ...
```

transformed



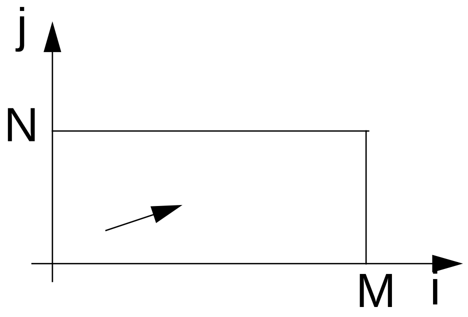
Permutation

Two loops of the loop nest are interchanged; the iteration space is flipped;
the **execution order** of iteration points **changes;** new dependence vectors must be legal.

general transformation matrix:

$$\begin{array}{c} \\ i \\ j \\ \\ \\ \end{array}\begin{pmatrix} 1 & & & & \\ & \mathbf{0} & \mathbf{1} & & 0 \\ & \mathbf{1} & \mathbf{0} & & \\ 0 & & & \ddots & \\ & & & & 1 \end{pmatrix}$$
$$\quad\quad\quad i \quad j$$

```
for i = 0 to M
   for j = 0 to N
      ...
```



original

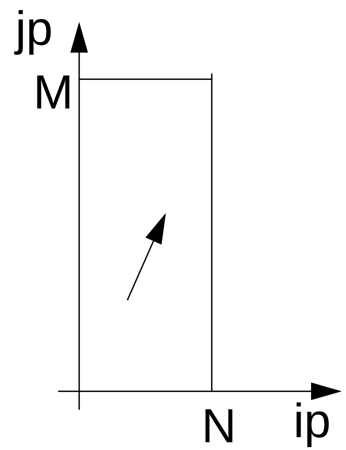
2-dimensional:

loop variables
old new

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} ip \\ jp \end{pmatrix}$$

```
for ip = 0 to N
   for jp = 0 to M
      ...
```

transformed



Use of Transformation Matrices

- Transformation matrix **T** defines **new iteration counts** in terms of the old ones: **T * i = i´**

e. g. Reversal

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

- Transformation matrix **T** transforms old **dependence vectors** into new ones: **T * d = d´**

e. g.

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

- inverse Transformation matrix $\mathbf{T}^{-1}$ defines **old iteration counts** in terms of new ones, for transformation of index expressions in the loop body: $\mathbf{T^{-1} * i' = i}$

e. g.

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} i' \\ -j' \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix}$$

- **concatenation of transformations** first $\mathbf{T_1}$ then $\mathbf{T_2}$: $\mathbf{T_2 * T_1} = \mathbf{T}$

e. g.

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

Inequalities Describe Loop Bounds

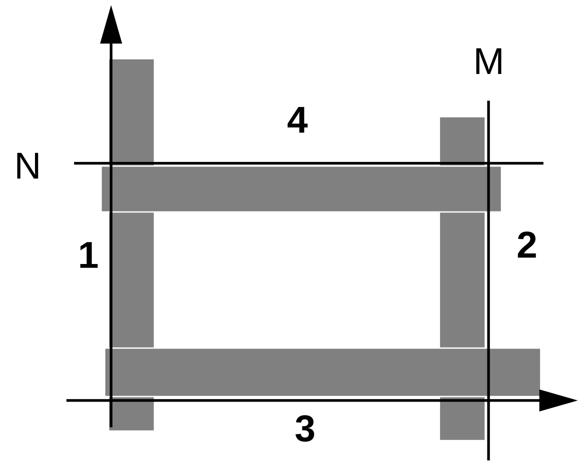
The bounds of a loop nest are described by a **set of linear inequalities**.
Each **inequality separates the space** in „inside and outside of the iteration space“:

$$\mathbf{B * i \leq c}$$

example 1

$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}$$

1 **-i** ≤ 0
2 **i** $\leq M$
3 **-j** ≤ 0
4 **j** $\leq N$

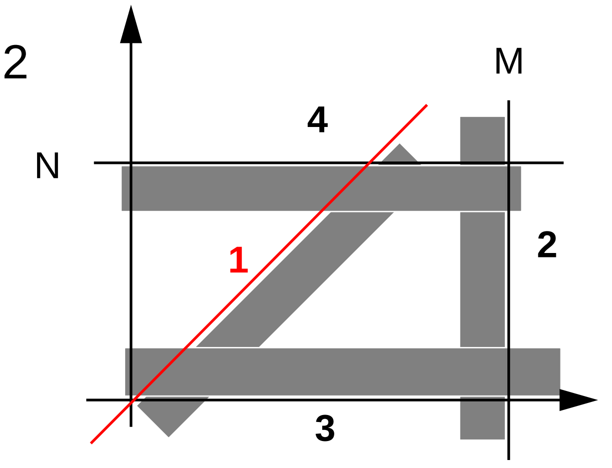


example 2

$$\begin{pmatrix} -1 & 1 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}$$

1 **-i +j** ≤ 0
2 **i** $\leq M$
3 **-j** ≤ 0
4 **j** $\leq N$

transformed



1, 4: $j \leq \min(i, N)$
3: $0 \leq j$
1+ 3: $0 \leq i$
2: $i \leq M$

positive factors represent **upper** bounds
negative factors represent **lower** bounds

Transformation of Loop Bounds

The inverse of a transformation matrix $\mathbf{T^{-1}}$ transforms a set of inequalities: $\mathbf{B * T^{-1} i' \leq c}$

skewing $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ inverse $\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$

$$\underset{B}{\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix}} * \underset{T^{-1}}{\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}} = \underset{B * T^{-1}}{\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ -1 & 1 \end{pmatrix}}$$

example 1
new bounds:

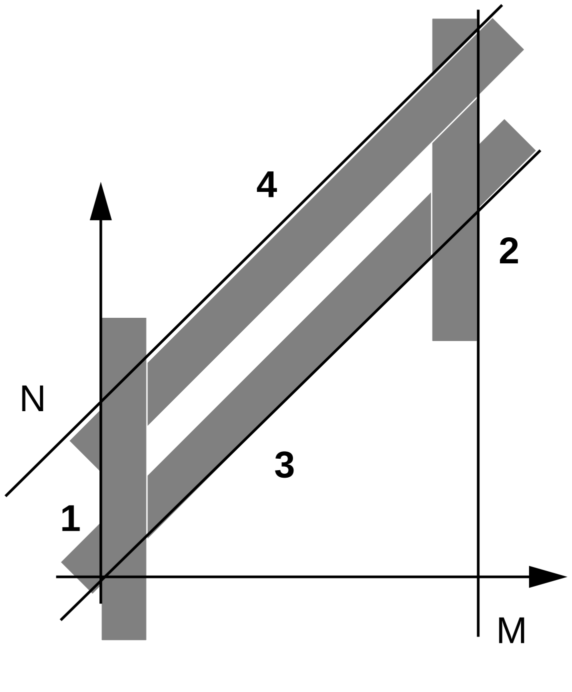
$$\underset{B * T^{-1}}{\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ -1 & 1 \end{pmatrix}} * \underset{i'}{\begin{pmatrix} i' \\ j' \end{pmatrix}} \leq \underset{c}{\begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}}$$

1 $-i' \leq 0$

2 $i' \leq M$

3 $i' - j' \leq 0$

4 $-i' + j' \leq N$



Example for Transformation and Parallelization of a Loop

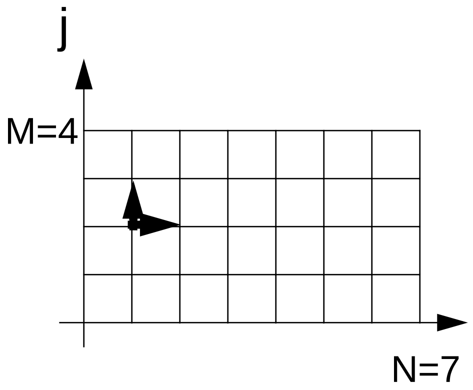
```
for i = 0 to N
   for j = 0 to M
      a[i, j] = (a[i, j-1] + a[i-1, j]) / 2;
```

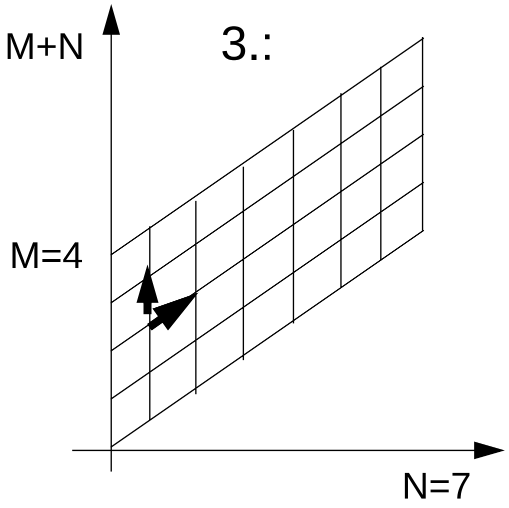
Parallelize the above loop.

1. Draw the iteration space.
2. Compute the dependence vectors and draw examples of them into the iteration space. Why can the inner loop not be executed in parallel?
3. Apply a skewing transformation and draw the iteration space.
4. Apply a permutation transformation and draw the iteration space. Explain why the inner loop now can be executed in parallel.
5. Compute the matrix of the composed transformation and use it to transform the dependence vectors.
6. Compute the inverse of the transformation matrix and use it to transform the index expressions.
7. Specify the loop bounds by inequalities and transform them by the inverse of the transformation matrix.
8. Write the complete loops with new loop variables ip and jp and new loop bounds.

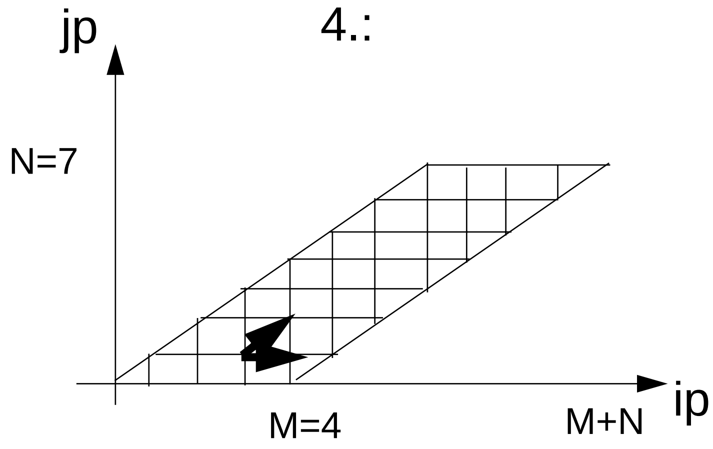
Solution of the Transformation and Parallelization Example

1., 2.:





4.:



5.:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

6.: Inverse

$$\begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$$

7. Bounds:

orig.: B $\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix}$ c $\begin{pmatrix} 0 \\ N \\ 0 \\ M \end{pmatrix}$

new: $B * T^{-1}$ $\begin{pmatrix} 0 & -1 \\ 0 & 1 \\ -1 & 1 \\ 1 & -1 \end{pmatrix}$

1 **-jp** ≤ 0
2 **jp** $\le N$
3 **-ip+jp** ≤ 0
4 **ip - jp** $\le M$

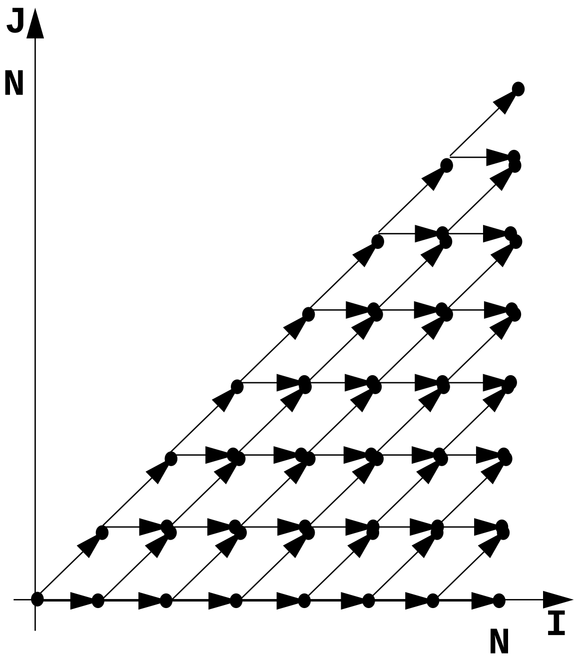
1, 3 => 0 $\le$ **ip**
2, 4 => ip $\le$ **M+N**
1, 4 => max (0, ip-M) $\le$ **jp**
2, 3 => jp $\le$ **min (ip, N)**

8.
```
for ip = 0 to M+N
   for jp = max (0, ip-M) to min (ip, N)
      a[jp, ip-jp] = (a[jp, ip-jp-1] + a[jp-1, ip-jp]) / 2;
```

Transformation and Parallelization

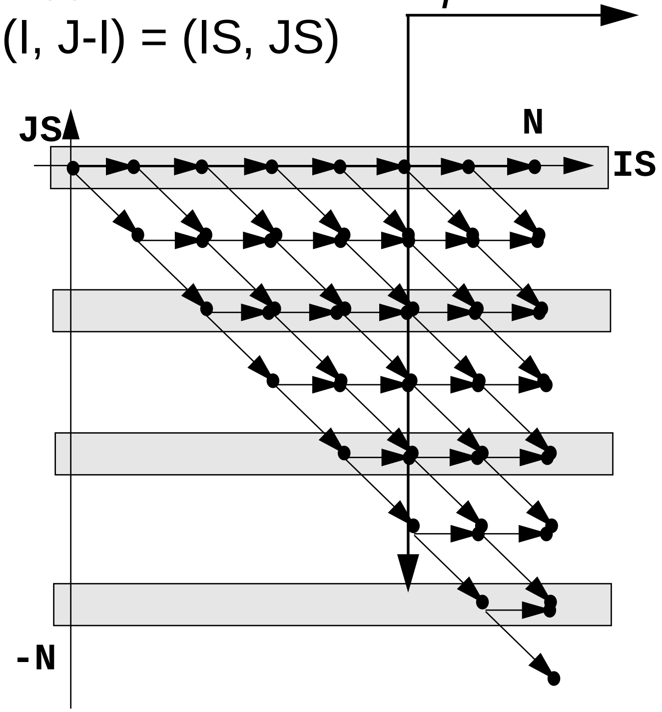
Iteration space

original



transformed
(I, J) -> (I, J-I) = (IS, JS)

sequential time IS



parallel processor map
JS mod 2

```
DECLARE B[-1..N,-1..N]

FOR I := 0 .. N
   FOR J := 0 .. I
      B[I,J] :=
         B[I-1,J]+B[I-1,J-1]
   END FOR
END FOR
```

```
DECLARE B[-1..N,-1..N]

FOR IS := 0.. N
   FOR JS := -IS .. 0
      B[IS,JS+IS] :=
         B[IS-1,JS+IS]+B[IS-1,JS-1+IS]
   END FOR
END FOR
```

Data Mapping

Goal:

Distribute array elements over processors, such that as many **accesses as possible are local.**

Index space of an array:

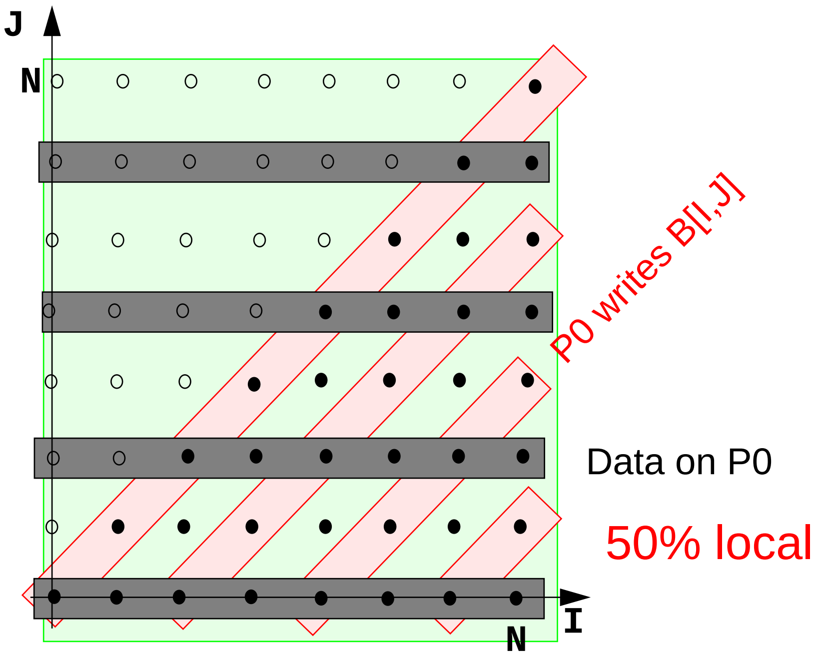
n-dimensional space of integral index points (polytope)

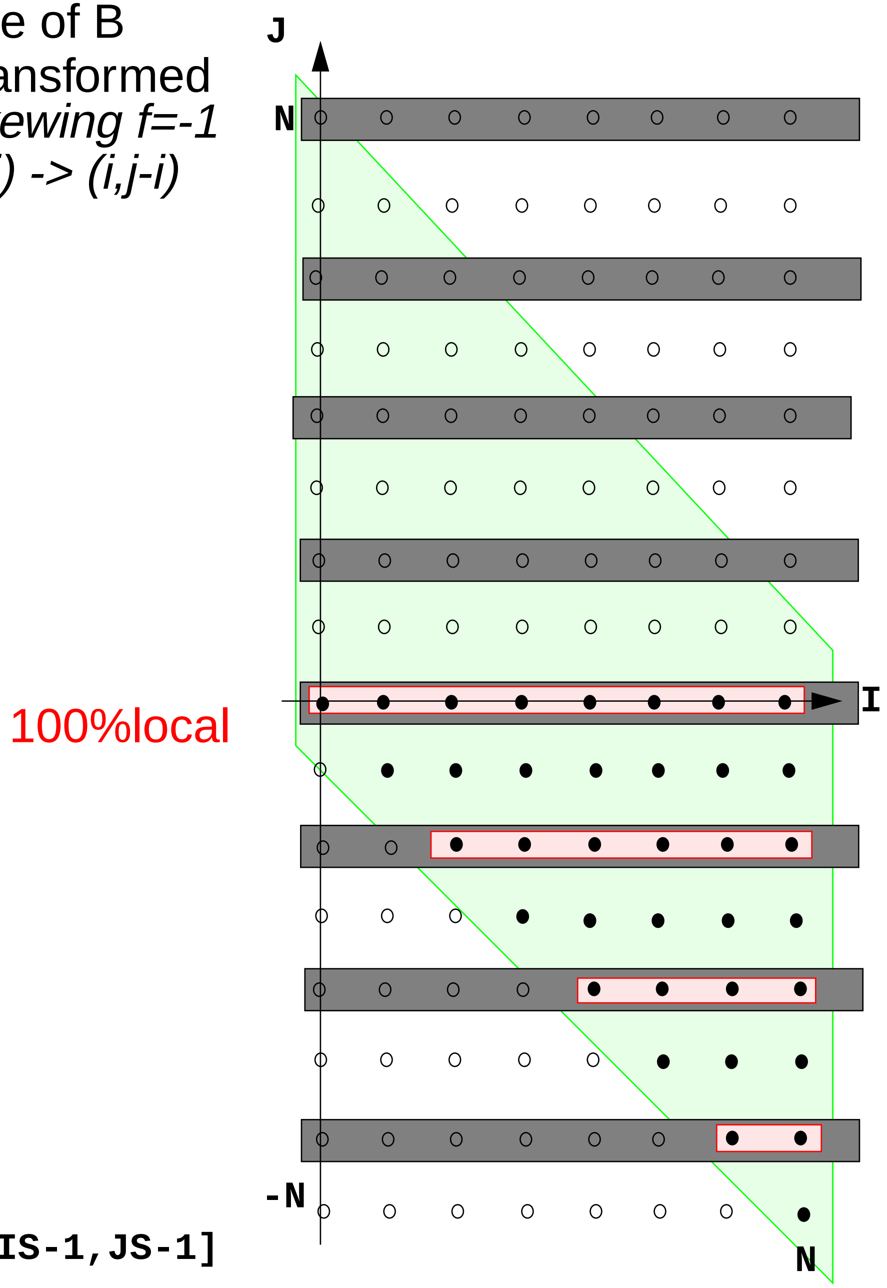
- **same properties as iteration space**
- same mathematical model
- same **transformations** are applicable (Skewing, Reversal, Permutation, ...)
- **no restrictions** by data dependences

Data distribution for parallel loops

index space of B

original | transformed
skewing f=-1
(i,j) -> (i,j-i)






```
DECLARE B[-1..N,-1..N]

FOR IS := 0.. N
    FOR JS := -IS .. 0
        B[IS,JS+IS] :=
            B[IS-1,JS+IS]+B[IS-1,JS-1+IS]
    END FOR
END FOR
```

```
DECLARE B[-1..N,-N..N]
...
    B[IS,JS] :=
        B[IS-1,JS-1]+B[IS-1,JS-1]
```