5. Data Parallelism: Barriers

Many processes execute the **same operations at the same time on different data**; usually on elements of **regular data structures**: arrays, sequences, matrices, lists.

Data parallelism as an **architectural model of parallel computers**:
vector machines, e. g. Cray
SIMD machines (Single Instruction Multiple Data), e. g. Connection Machine, MasPar
GPUs (Graphical Processing Units); massively parallel processors on graphic cards

Data parallelism as a **programming model for parallel computers**:

- computations on **arrays in nested loops**
- analyze **data dependences** of computations, **transform** and **parallelize** loops
- iterative **computations in rounds**, synchronize with **Barriers**
- **systolic computations**: 2 phases are iterated: compute - shift data to neighbour processes

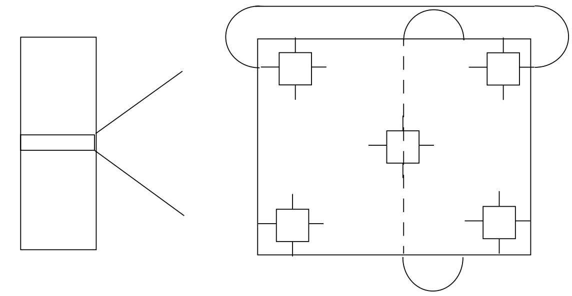
Applications mainly in **technical, scientific computing**, e. g.

- fluid mechanics
- image processing
- solving differential equations
- finite element method in design systems

Data parallelism as an architectural model

SIMD machine: Single Instruction Multiple Data

- very many processors, **massively parallel**
  e. g. 32 x 64 processor field
- **local memory** for each processor
- same instructions in **lock step**
- fast communication in **lock step**
- fixed topology, usually a **grid**
- machine types e. g. Connection Machine, MasPar



program

field of processors

Data parallelism as a programming model

- regular data structures (arrays, lists) are mapped onto a field of processors
- processes execute the same program on individual data in lock step
- communication with neighbours in the same direction in lock step

simple example matrix addition:

C = A + B

sequential:

```
for (i = 0; i < N; i++)
   for (j = 0; j < M; j++)
      c[i,j] = a [i,j] + b[i,j];
```

```
distribute A, B
c = a + b
collect C
```

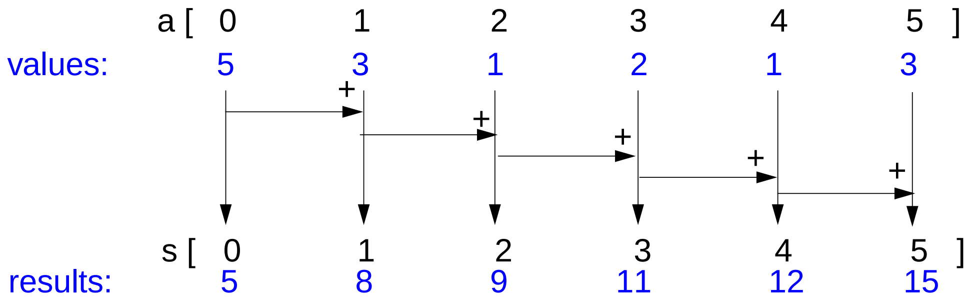
1 step!

- these can be parallelized directly, since there are no **data dependences**
- **data mapping** is trivial: array element [i,j] on process [i,j]
- **communication** is not needed
- no **algorithmic idea** is needed

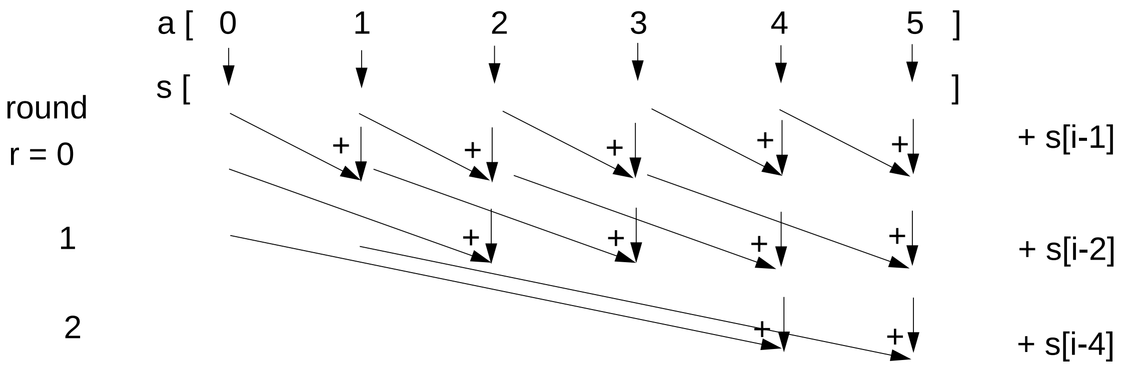
Example prefix sums

input: sequence a of numbers;
output: sequence s of sums of the prefixes of a

$$s[i] = \sum_{j=0}^{i} a[j]$$



parallel algorithmic idea:

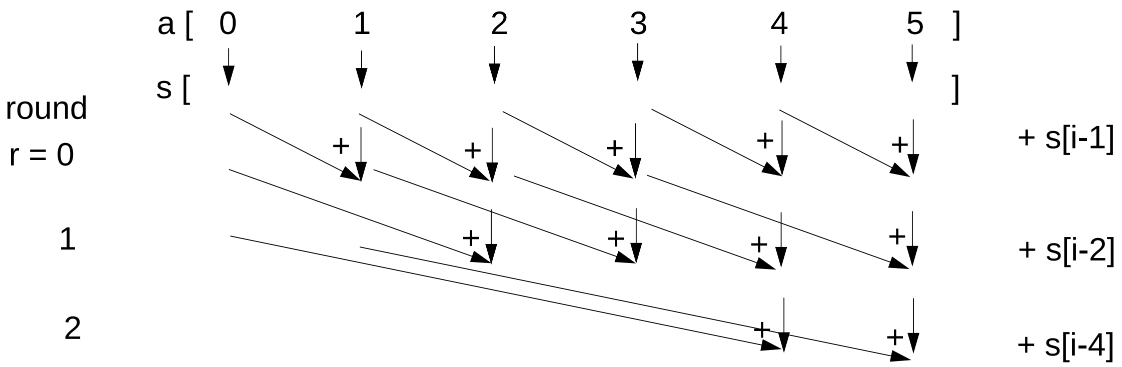


Example prefix sums (2)

input:sequence a of numbers;
output:sequence s of sums of the prefixes of a

$$s[i] = \sum_{j=0}^{i} a[j]$$

parallel algorithmic idea:



Proof for process p = 0 .. n - 1

Invariant SUM: s[p] = a[p-d+1] + ... + a[p] with d = 1, 2, ..., m <= n distance before next round

Induction begin: d = 1; s[p] = a[p] holds by initialization

induction step: computation s[p] = s[p - d] + s[p]
a[p-2d+1] + ... + a[p-d] + a[p-d+1] + ... + a[p]

substitution of 2d by d implies SUM

Prefix sums: applied methods

- computational scheme **reduction**:
  all array elements are comprised using a reduction operation (here: addition)
- iterative **computation in rounds**:
  in each round all processes perform a computation step
- **duplication of distance**:
  data is exchanged in each round with a neighbour at twice the distance as in the previous round
- **barrier** synchronization:
  processes may not enter the next round, before all processes have finished the previous one

Barriers

Several processes meet at a common point of synchronization

Rule: All processes must have reached the barrier (for the j-th time), before one of them leaves it (for the j-th time).

Applications:

- iterative computations, where iteration j uses results of iteration j-1
- separation of computational phases

Scheme:

```
public void run ()
{  do {  computeNewValues (i);
         b.barrier();
      }
   while (!converged);
}
```

Implementation techniques for barriers:

- central controller: monitor or coordination process
- worker processes coordinated as a tree
- worker processes symmetrically coordinated (butterfly barrier, dissemination barrier)

Barrier implemented by a monitor

Monitor stops a given number of processes and releases them together:

```
class BarrierMonitor
{  private int  processes        // number of processes to be synchronized
                arrived = 0;     // number of processes arrived at the barrier

   public BarrierMonitor (int procs)
   {  processes = procs; }

   synchronized public barrier ()
   {  arrived++;
      if (arrived < processes)
         try { wait(); } catch (InterruptedException e) {}
                                     // exception destroys barrier behaviour
      else
      {  arrived = 0;                // reset arrival count
         notifyAll();                // release the other processes
}  }  }
```

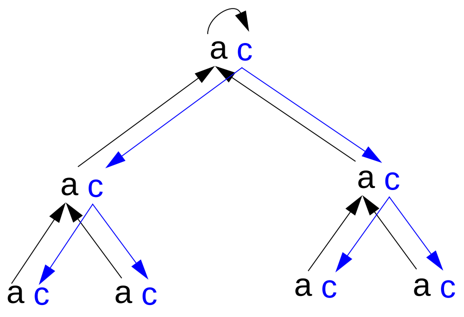
Distributed tree barrier

Barrier synchronization of the worker processes organized as a **binary tree**.
Bottleneck of central synchronization is avoided.

2 synchronization variables (flags) at each node:

arrived: all processes in a subtree have arrived, is propagated upward

continue: all processes in a subtree may continue, is propagated downward



disadvantage:
different code is needed for root, inner nodes, and for leafs

2 Rules for Synchronization Using Flags

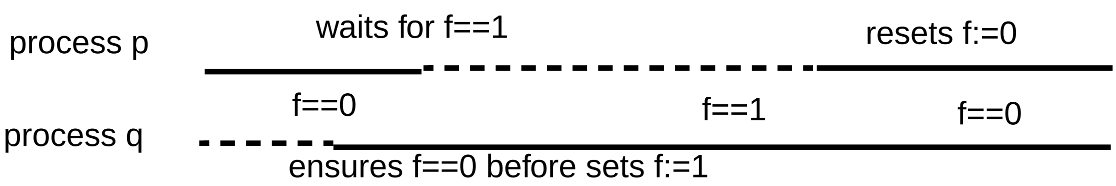
Flag for synchronization between exactly 2 processes

One process waits until the flag is set.
The other process sets the flag.

May be implemented by a monitor in Java.

Flag rules: 1. The process that waits for a flag resets it.
2. A flag that is set may not be set again before being reset.

Consequence: no state change will be lost.



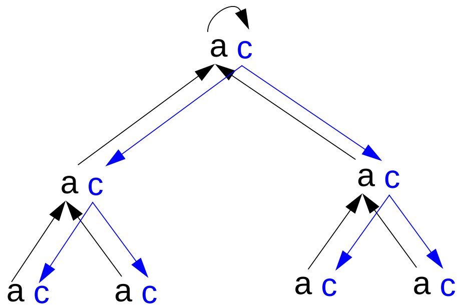
Distributed tree barrier implementation

2 synchronization variables (flags) at each node:

arrived: all processes in a subtree have arrived
propagated upward

continue: all processes in a subtree may continue
propagated downward

initially all flags are reset



code for an **inner** node:

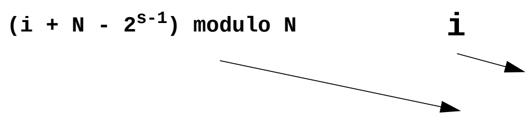
```
                                                    leaf      root
execute this.task();                                x         x
wait for left.arrived; reset left.arrived;                    x
wait for right.arrived; reset right.arrived;                  x
set this.arrived;                                   x
wait for this.continue; reset this.continue;        x
set left.continue;                                            x
set right.continue;                                           x
```

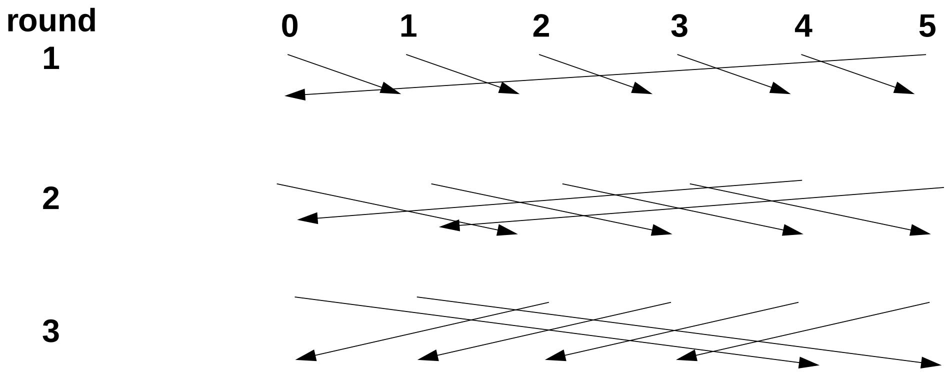

Symmetric, distributed barrier (dissemination)

Processes **synchronize pairwise** in **rounds with doubled distances**.

N processes are synchronized after r rounds if N <= 2^r

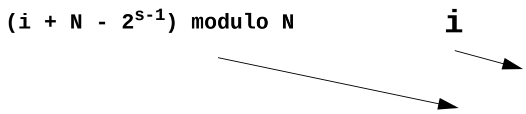
In round s
process i indicates its arrival and then waits
for the arrival of process (i + N - 2^{s-1}) modulo N:





After r rounds each process is synchronized with each other. Proof idea: For each process i each other process occurs in a tree of processes which have synchronized (in)directly with i.

Symmetric, distributed barrier: implementation



In round s

process i indicates its arrival and

waits for the arrival of process (i + N - 2^{s-1}) modulo N

Code for each process:

```
execute this.task();

// synchronize:
s = 0;
while (N > 2^s)
   s++;
   wait for f==0; set f=1;
   partner=p[(i + N - 2^(s-1)) modulo N];
   wait partner.f; reset partner.f=0
```


Prefix sums with barriers

```
class PrefixSum extends Thread
{  private int procNo;                         // number of process
   private BarrierMonitor bm;                     // barrier object

   public PrefixSum (int p, BarrierMonitor b)
   { procno = p; bm = b; }

   public void run ()
   {  int addIt, dist = 1;                                 // distance
                                               // global arrays a and s
      s[procNo] = a[procNo];                  // initialize result array
      bm.barrier();

      // invariant SUM: s[procNo] == a[procNo-dist+1]+...+a[procNo]
      while (dist < N)
      {  if (procNo - dist >= 0)
            addIt = s[procNo - dist];   // value before overwritten
         bm.barrier();
         if (procNo - dist >= 0)
            s[procNo] += addIt;
         bm.barrier();
         dist = dist * 2;                          // doubled distance
} } }
```

Prefix sums in a synchronous parallel programming model

Notation in Modula-2* with synchronous (and asynchronous) loops for parallel machines

```
VAR a, s, t: ARRAY [0..N-1] OF INTEGER;
VAR dist: CARDINAL;
BEGIN
   ...
   FORALL i: [0..N-1] IN SYNC
      s[i] := a[i];
   END;
```

parallel loop in lock step

```
   dist := 1;

   WHILE dist < N
      FORALL i: [0..N-1] IN SYNC
         IF (i-dist) >= 0 THEN
            t[i] := s[i - dist];
            s[i] := s[i] + t[i];
         END
      END;
      dist := dist * 2;
   END
END
```

parallel loop in lock step

implicit barrier
for each statement

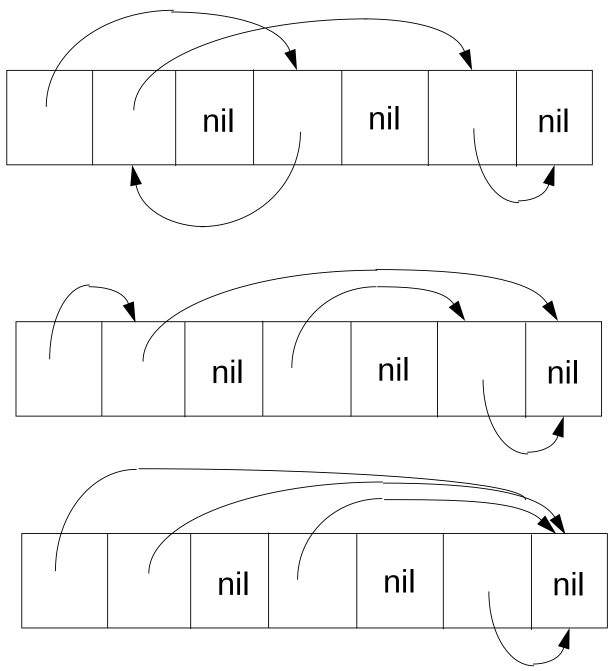
Finding list ends: data parallel approach

input: int array link stores lists; link[i] contains the index of the successor or nil

output: int array last; last[i] contains the index of the last element of list link[i]

method: **worker process** i computes last[i] = last[last[i]] in log N **rounds**

```
int d = 1;
last[i] = link[i];
barrier

while (d < N)
{  int newlast = nil;
   if ( last[i] != nil &&
        last[last[i]] != nil)
      newlast = last[last[i]];
   barrier
   if (newlast != nil)
      last[i] = newlast;
   barrier
   d = 2*d;
}
```



last[i] points to the end of those lists which are not longer than d

5.2 / 6. Data Parallelism: Loop Parallelization

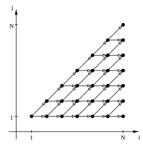
Regular loops on orthogonal data structures - parallelized for **data parallel** processors

Development steps (automated by compilers):

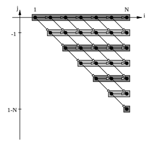
- **nested loops** operating on **arrays**,
  sequential execution of iteration space

```
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
```

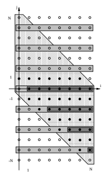
- analyze **data dependences**
  data-flow: definition and use of array elements



- **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



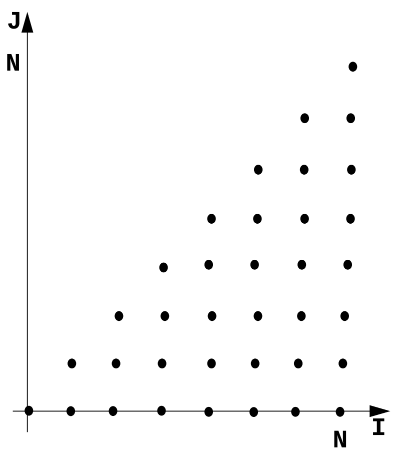
Iteration space of loop nests

Iteration space of a loop nest of depth n:

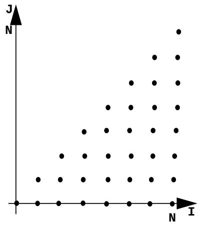
- **n-dimensional space of integral points** (polytope)
- each point (i_1, ..., i_n) represents an execution of the innermost loop body
- loop bounds are in general not known before run-time
- iteration need not have orthogonal borders
- iteration is elaborated sequentially

example:
computation of Pascal's triangle

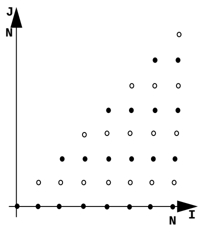
```
DECLARE B[-1..N,-1..N]

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



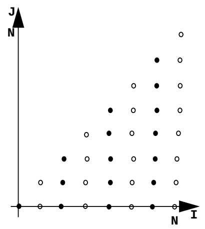
Examples for Iteration spaces of loop nests



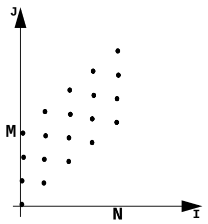

```
FOR I := 0 .. N
    FOR J := 0 .. I
```



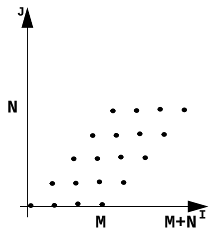

```
FOR I := 0 .. N
    FOR J := 0..I BY 2
```




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




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

M = 3, N = 4
```

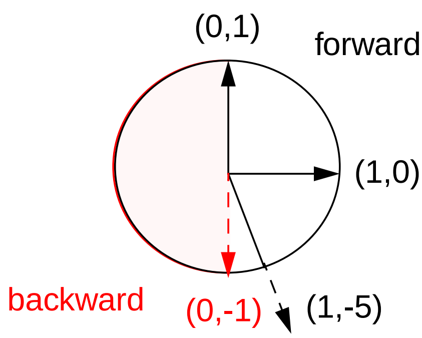



```
FOR I := 0 .. M+N
    FOR J := max(0, I-M)..
                min (I, N)
```


Data Dependences in Iteration Spaces

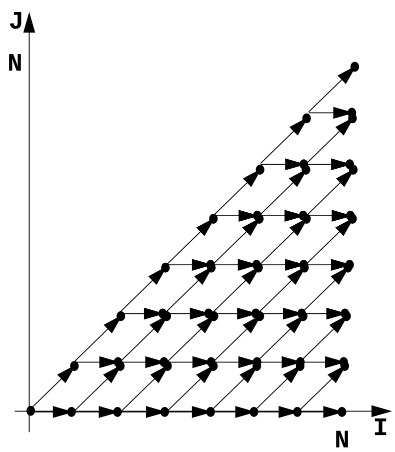
Data dependence from iteration point i1 to i2:

- 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

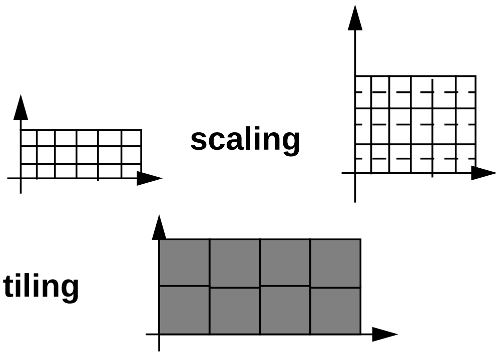
linear basic transformations:

- **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

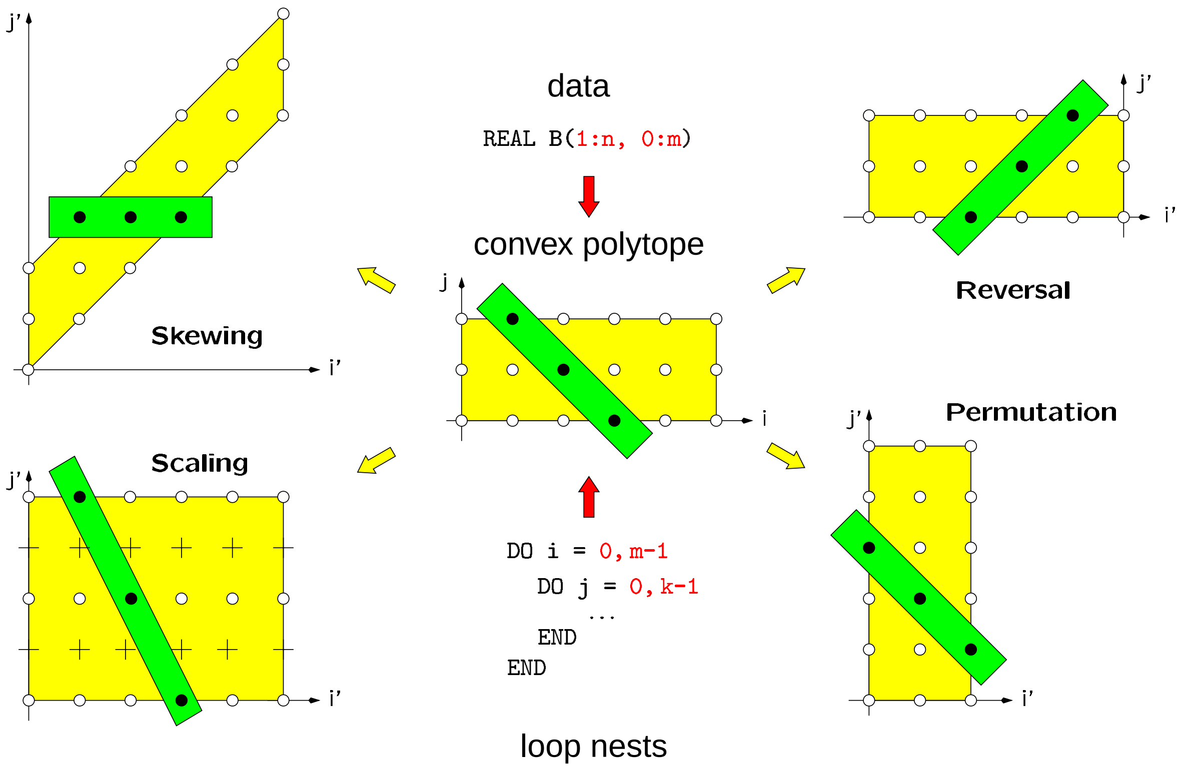
SRP transformations (next slides)

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



Transformations of



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

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

2-dimensional:

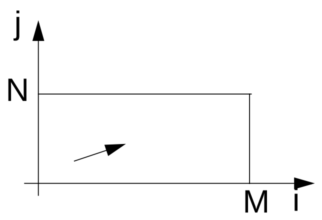
loop variables

old new

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

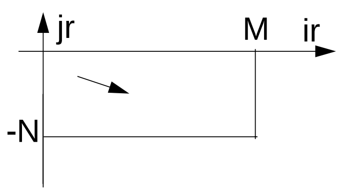
```
for i = 0 to M
   for j = 0 to N
      ...
```

```
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:

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

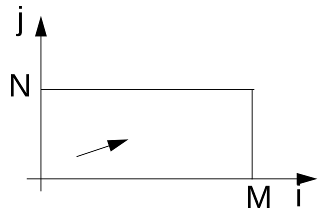
2-dimensional:

loop variables
old new

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

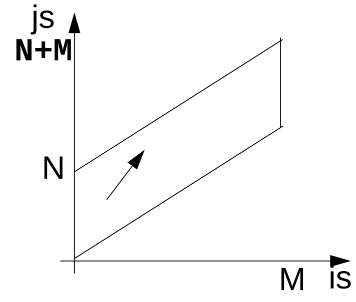
```
for i = 0 to M
   for j = 0 to N
      ...
```

original




```
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:

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

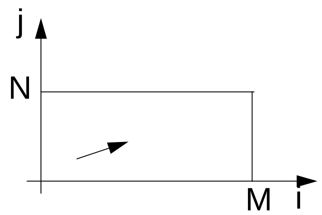
2-dimensional:

loop variables
old new

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

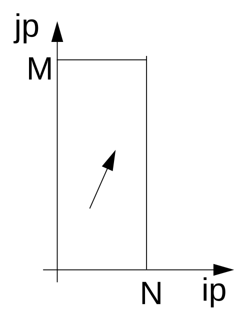
```
for i = 0 to M
   for j = 0 to N
      ...
```

original




```
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: $\mathbf{T}^{-1} * \mathbf{i'} = \mathbf{i}$

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 $\mathbf{T_2}$: $\mathbf{T_2} * \mathbf{T_1} = \mathbf{T}$

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

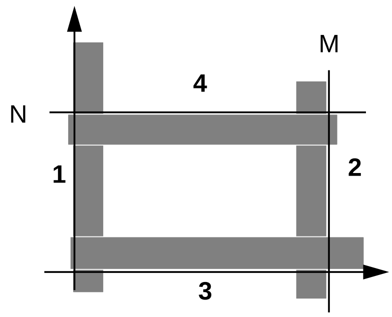
The bounds of a loop nest are described by a **set of linear inequalities**.
Each **inequality separates the space** in „inside and outside of the iteration space“:

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

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

example 1

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

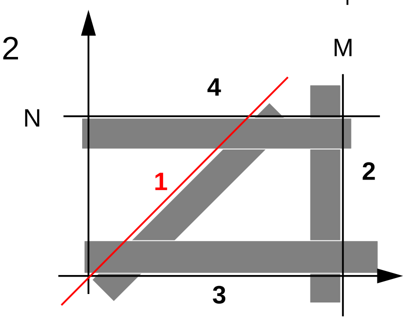


$$\begin{pmatrix} -1 & 1 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}$$

example 2

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

transformed



1, 4: $j \leq \min(i, N)$ **1+ 3:** $0 \leq i$
3: $0 \leq j$ **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: $\mathbf{B * T^{-1} i' \leq c}$

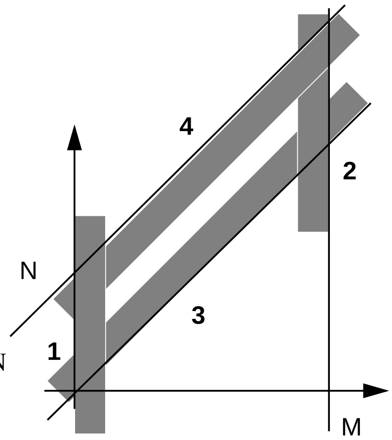
skewing $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ inverse $\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$

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

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 $-i' + j' \leq N$



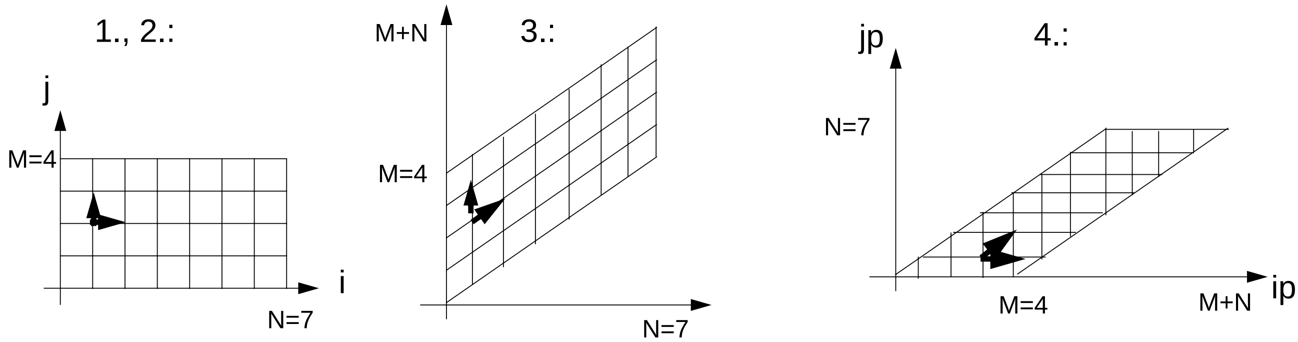
Example for Transformation and Parallelization of a Loop

```
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



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:

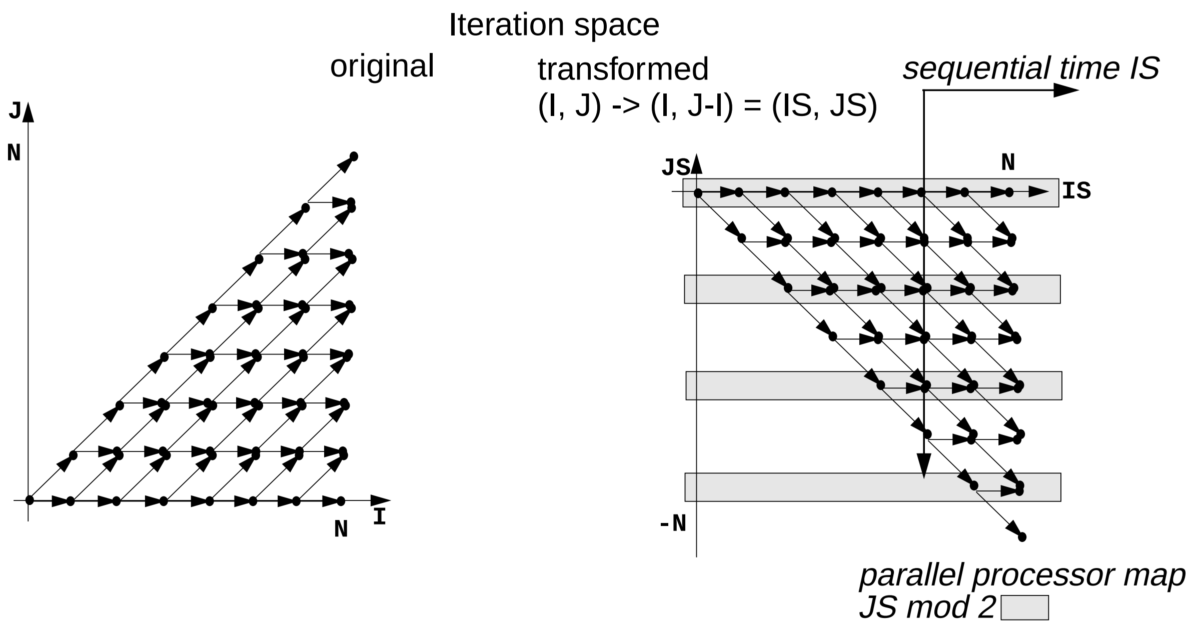
orig.: $\overset{B}{\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix}} \quad \overset{c}{\begin{pmatrix} 0 \\ N \\ 0 \\ M \end{pmatrix}}$ new: $\overset{B * T^{-1}}{\begin{pmatrix} 0 & -1 \\ 0 & 1 \\ -1 & 1 \\ 1 & -1 \end{pmatrix}}$

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




```
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

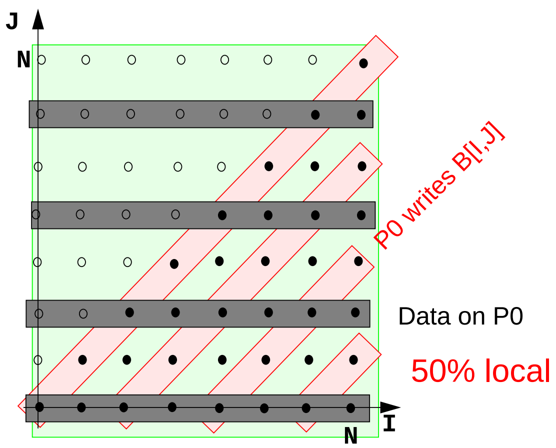
index space of B

original

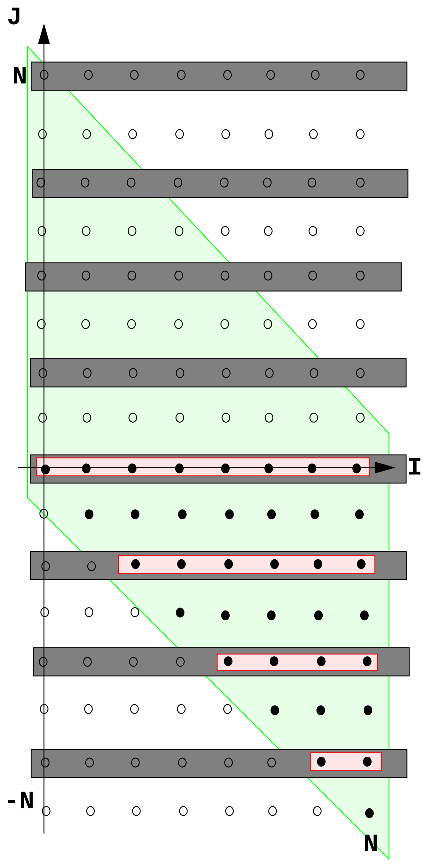
transformed

skewing f=-1

(i,j) -> (i,j-i)



100%local




```
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]
```