## 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

## Lecture Parallel Programming WS 2014/2015 / Slide 38

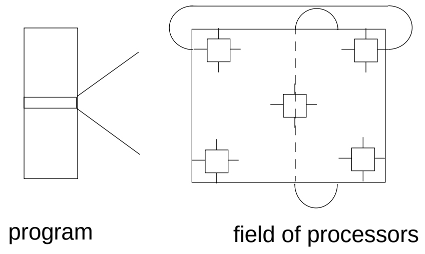
**Objectives:**
Overview over notions of data parallelism

**In the lecture:**
Explain the notions

## 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



## Lecture Parallel Programming WS 2014/2015 / Slide 39

**Objectives:**
Architecture of a SIMD computer

**In the lecture:**
Explanation of the properties

## 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:

$$\boxed{C} = \boxed{A} + \boxed{B}$$

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

## Lecture Parallel Programming WS 2014/2015 / Slide 40

**Objectives:**

idea of loop parallelization

**In the lecture:**

- explain the example,
- show the reasons for the simplicity of the parallelization

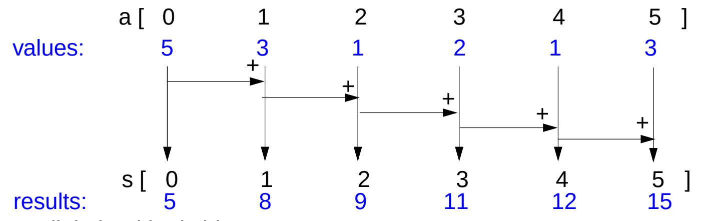
**Questions:**

- Give examples for array operations that can be parallelized with similar ease.

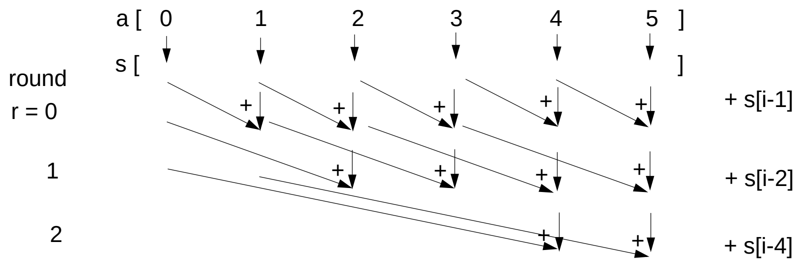
## Example prefix sums

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:



## Lecture Parallel Programming WS 2014/2015 / Slide 41

**Objectives:**

Understand the parallel computation of prefix sums

**In the lecture:**

Explain

- the task,
- the algorithmic idea,
- how to exploit associativity,
- computations in rounds,
- duplication of distance

**Questions:**

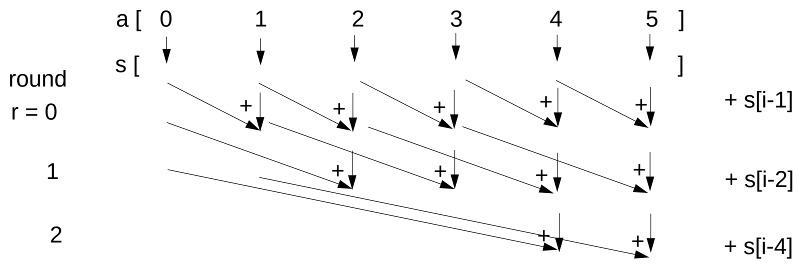
- What is the formula for the number of steps in the sequential and in the parallel case?

## 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

## Lecture Parallel Programming WS 2014/2015 / Slide 41a

**Objectives:**

Proof the parallel computation of prefix sums

**In the lecture:**

Explain

- the proof

## 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

## Lecture Parallel Programming WS 2014/2015 / Slide 42

**Objectives:**

Point out the methods

**In the lecture:**

- Explain the methods for the prefix sums.
- Point out other applications of these methods.

## 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)

## Lecture Parallel Programming WS 2014/2015 / Slide 43

**Objectives:**

Understand the concept of barriers

**In the lecture:**

Explain

- the barrier rule,
- the relation to the prefix sums,
- applications.

## 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
}  }  }
```

## Lecture Parallel Programming WS 2014/2015 / Slide 44

**Objectives:**

Understand the monitor implementation

**In the lecture:**

Explain

- the implementation,
- why waiting in a loop is not necessary.

**Questions:**

- Why does this central solution cause a bottleneck?

## Distributed tree barrier

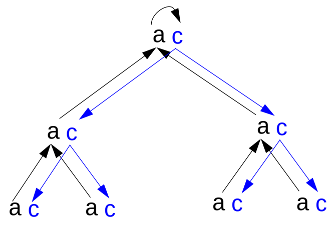
Barrier synchronization of the worker processes organized as a **binary tree**.
Bottleneck of central synchronization is avoided.

**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



## Lecture Parallel Programming WS 2014/2015 / Slide 45

**Objectives:**
Understand the tree barrier

**In the lecture:**
Explain

- the principle of 2 phases,
- the advantage of the distributed solution,

## 2 Rules for Synchronization Using Flags

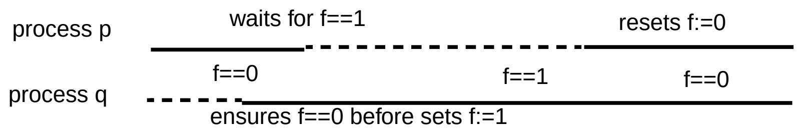
Flag for synchronization between exactly 2 processes

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.



## Lecture Parallel Programming WS 2014/2015 / Slide 45a

**Objectives:**
Understand flag synchronization

**In the lecture:**
Explain

- the general flag rules.

**Assignments:**

- Design a Java class for flag synchronization between 2 processes. Ensure that the flag rules are obeyed.

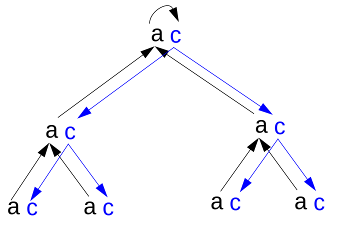
## Distributed tree barrier implementation

**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
```

## Lecture Parallel Programming WS 2014/2015 / Slide 45b

**Objectives:**

Understand the tree barrier

**In the lecture:**

Explain

- the different code for the 3 kinds of nodes,

**Assignments:**

- Write the code for the 3 kinds of nodes using objects of the flag class.

## Symmetric, distributed barrier (dissemination)

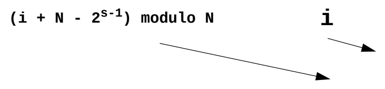
Processes **synchronize pairwise** in **rounds with doubled distances**.

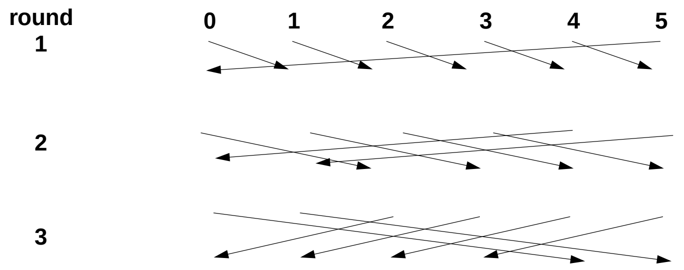
N processes are synchronized after r rounds if N <= $2^r$

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.

## Lecture Parallel Programming WS 2014/2015 / Slide 46

**Objectives:**

Understand the dissemination barrier

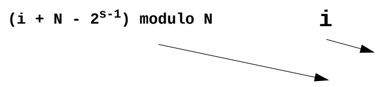
**In the lecture:**

- Symmetric code for arbitrary many processes.
- Arc i to j in the diagram means j waits for arrival of i.
- show the synchronization for pairs.
- No cyclic waiting, because the arrival is indicated first, then the partner is waited for.
- After the last round all processes are synchronized, because for all processes p a binary tree exists s.t. p is its root, all processes are in that tree, the arcs are waiting pairs from the diagram forming pathes from the leaves to the root..

**Questions:**

- Write the synchronization code.
- Show one of the binary trees.

## 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
```

## Lecture Parallel Programming WS 2014/2015 / Slide 46a

**Objectives:**

Understand the dissemination barrier

**In the lecture:**

- Processes have to wait before they set AND before they reset the flag.
- Symmetric code for arbitrary many processes.

**Questions:**

- Write the synchronization code.
- Show one of the binary trees.

## 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
} } }
```

## Lecture Parallel Programming WS 2014/2015 / Slide 47

**Objectives:**

Examples for synchonization points

**In the lecture:**

Explain

- the invariant,
- the access of `s[procNo]`,
- the reasons for the 3 synchronization points.

**Questions:**

- Explain the reasons for the 3 synchronization points.

## 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                      parallel loop in lock step
      s[i] := a[i];
   END;

   dist := 1;

   WHILE dist < N                                  parallel loop in lock step
      FORALL i: [0..N-1] IN SYNC
         IF (i-dist) >= 0 THEN
            t[i] := s[i - dist];                          implicit barrier
            s[i] := s[i] + t[i];                     for each statement
         END
      END;
      dist := dist * 2;
   END
END
```

### Lecture Parallel Programming WS 2014/2015 / Slide 48

**Objectives:**

Implicit barriers

**In the lecture:**

- Explain the language constructs.
- If expressions were evaluated in lock step, too, the array `t` could be omitted.
- The MasPar SIMD machine would be programmed similarly.

**Questions:**

- Explain the execution if values were not saved in `t[i]`.

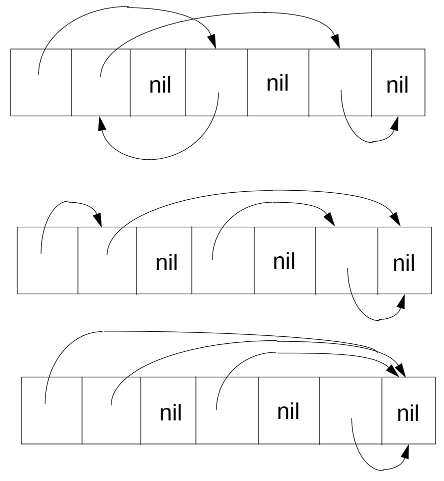
## Finding list ends: data parallel approach

| | |
|---|---|
| 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



### Lecture Parallel Programming WS 2014/2015 / Slide 49

**Objectives:**

Data parallelism not only for arrays!

**In the lecture:**

Explain

- parallel scanning of lists,
- doubling distances for lists,
- `last[last[i]]`,
- that it is only useful if the ends of many long lists are searched.

**Questions:**

- Which role plays the distance `d` here?

## 5.2 / 6. Data Parallelism: Loop Parallelization

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

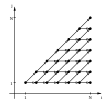
Development steps (automated by compilers):

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

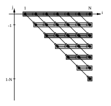
```
DECLARE B[0..N,0..N+1]

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

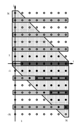
- analyze **data dependences**
  data-flow: definition and use of array elements



- **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



## Lecture Parallel Programming WS 2014/2015 / Slide 50

**Objectives:**

Overview

**In the lecture:**

Explain

- Application area: scientific computations
- goals: execute inner loops in parallel with efficient data access
- transformation steps

## Iteration space of loop nests

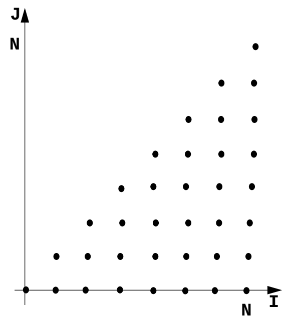
**Iteration space** of a loop nest of depth n:

- **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
```



## Lecture Parallel Programming WS 2014/2015 / Slide 51

**Objectives:**

Understand the notion of iteration space

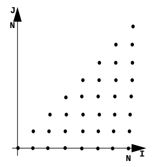
**In the lecture:**

- Explain the iteration space of the example.
- Show the order of elaboration of the iteration space.
- If the step size is greater than 1 the iteration space has gaps - the polytope is not convex.

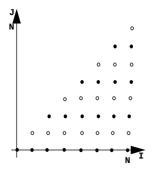
**Questions:**

- Draw an iteration space that has step size 3 in one dimension.

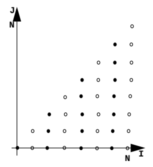
## Examples for Iteration spaces of loop nests



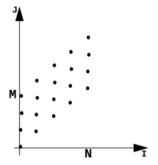
```
FOR I := 0 .. N
    FOR J := 0 .. I
```



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

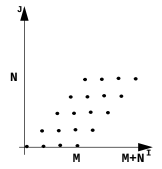


```
FOR I := 0..N BY 2
    FOR J := 0 .. I
```



```
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)
```

## Lecture Parallel Programming WS 2014/2015 / Slide 51a

**Objectives:**

Relate loop nests to iteration spaces

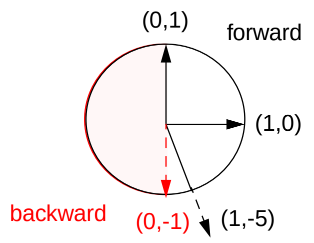
**In the lecture:**

- Explain the iteration spaces of the examples

## Data Dependences in Iteration Spaces

**Data dependence from iteration point i1 to i2**:

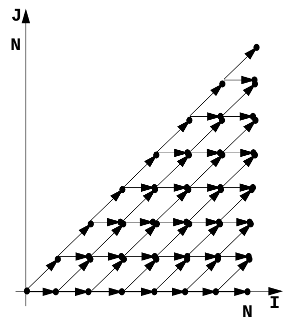
- Iteration **i**1 computes a value that is used in iteration **i2** (flow dependence)
- relative **dependence vector**
  **d** = **i2 - i1** = ($i2_1$ - $i1_1$, ..., $i2_n$ - $i1_n$)
  holds for all iteration points except at the border
- Flow-dependences can **not be directed against the execution 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
```



## Lecture Parallel Programming WS 2014/2015 / Slide 52

**Objectives:**

Understand dependences in loops

**In the lecture:**

Explain:

- Vector representation of dependences,
- examples,
- admissable directions graphically

**Questions:**

- Show different dependence vectors and array accesses in a loop body which cause dependences of given vectors.

## 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**

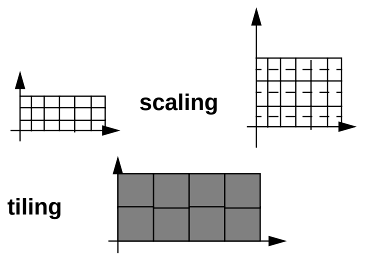
**linear basic transformations:**

- **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



### Lecture Parallel Programming WS 2014/2015 / Slide 53

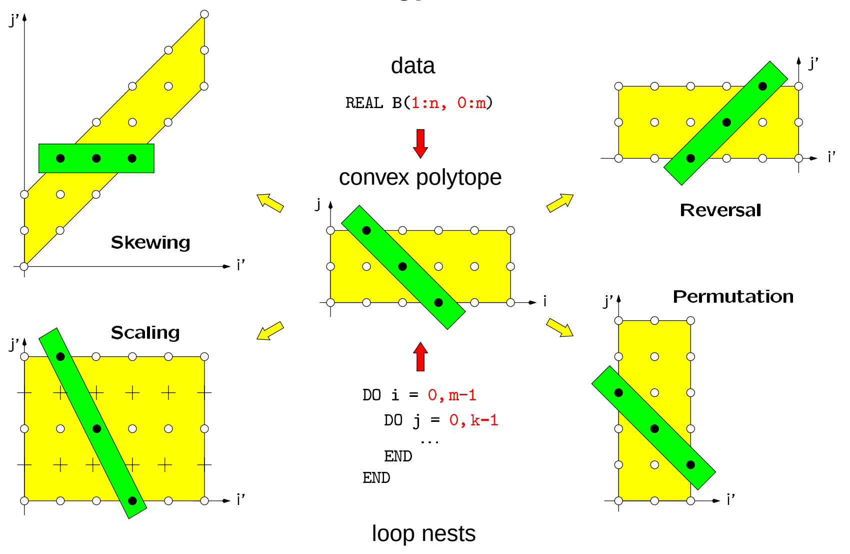
**Objectives:**

Overview

**In the lecture:**

- Explain the goals.
- Show admissable directions of dependences.
- Show diagrams for the transformations.

## Transformations of



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

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

### Lecture Parallel Programming WS 2014/2015 / Slide 54

**Objectives:**

Visualize the transformations

**In the lecture:**

- Give concrete loops for the diagrams.
- Show how the dependence vectors are transformed.
- Skewing and scaling do not change the order of execution; hence, they are always applicable.

**Questions:**

- Give dependence vectors for each transformation, which are still valid after the transformation.

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

## Lecture Parallel Programming WS 2014/2015 / Slide 55

**Objectives:**

Understand the matrix representation

**In the lecture:**

- Explain the principle.
- Map concrete iteration points.
- Map dependence vectors.
- Show combinations of transformations.

**Questions:**

- Give more examples for skewing transformations.

## Reversal

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

**general transformation matrix**

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

**2-dimensional:**

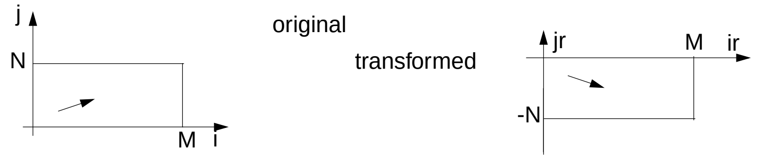
loop variables

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
      ...
```



## Lecture Parallel Programming WS 2014/2015 / Slide 55a

**Objectives:**

Understand reversal transformation

**In the lecture:**

- Explain the effect of reversal transformation.
- Explain the notation of the transformation matrix.
- There may be no dependences in the direction of the reversed loop - they would point backward after the transformation.

**Questions:**

- Show an example where reversal enables loop fusion.

## 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:**

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

**2-dimensional:**

loop variables
old new

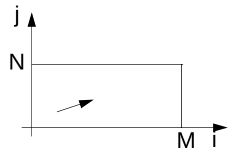
$$\begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ f*i+j \end{pmatrix} = \begin{pmatrix} is \\ js \end{pmatrix}$$

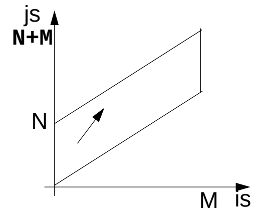
```
for i = 0 to M
   for j = 0 to N
      ...
```

original

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

transformed





### Lecture Parallel Programming WS 2014/2015 / Slide 55b

**Objectives:**

Understand skewing transformation

**In the lecture:**

- Explain the effect of a skewing transformation.
- Skewing is always applicable.
- Skewing can enable loop permutation

**Questions:**

- Show an example where skewing enables loop permutation.

## 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

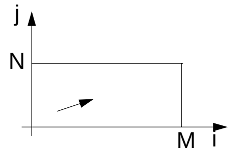
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} ip \\ jp \end{pmatrix}$$

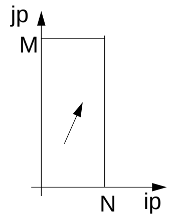
```
for i = 0 to M
   for j = 0 to N
      ...
```

original

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

transformed





### Lecture Parallel Programming WS 2014/2015 / Slide 55c

**Objectives:**

Understand loop permutation

**In the lecture:**

- Explain the effect of loop permutation.
- Show effect on dependence vectors.
- Permutation often yields a parallelizable innermost loop.

**Questions:**

- Show an example where permutation yields a parallelizable innermost loop.

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

## Lecture Parallel Programming WS 2014/2015 / Slide 56

**Objectives:**

Learn to Use the matrices

**In the lecture:**

- Explain the 4 uses with examples.
- Transform a loop completely.

**Questions:**

- Why do the dependence vectors change under a transformation, although the dependence between array elements remains unchanged?

## Inequalities Describe Loop Bounds

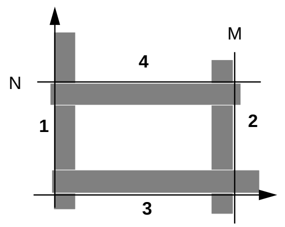
The bounds of a loop nest are described by a **set of linear inequalities**.
Each **inequality separates the space** in „inside and outside of the iteration space“:

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

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

example 1

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

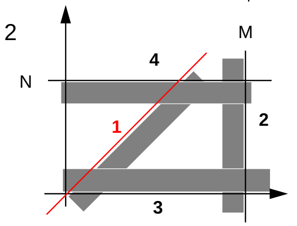


$$\begin{pmatrix} -1 & 1 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} \le \begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}$$

example 2

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

transformed



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

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

## Lecture Parallel Programming WS 2014/2015 / Slide 56a

**Objectives:**

Understand representation of bounds

**In the lecture:**

- Explain matrix notation.
- Explain graphic interpretation.
- There can be arbitrary many inequalities.

**Questions:**

- Give the representations of other iteration spaces.

## Transformation of Loop Bounds

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

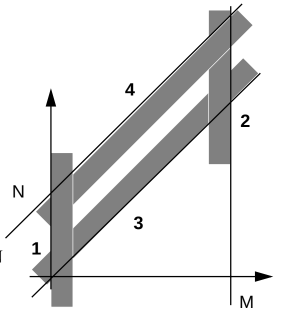
skewing $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ inverse $\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$

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

example 1
new bounds:

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

**1** $-i' \le 0$
**2** $i' \le M$
**3** $i' - j' \le 0$
**4** $-i' + j' \le N$



## Lecture Parallel Programming WS 2014/2015 / Slide 56b

**Objectives:**

Understand the transformation of bounds

**In the lecture:**

- Explain how the inequalities are transformed

**Questions:**

- Compute further transformations of bounds.

## 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.

## Lecture Parallel Programming WS 2014/2015 / Slide 56c

**Objectives:**

Exercise the method for an example

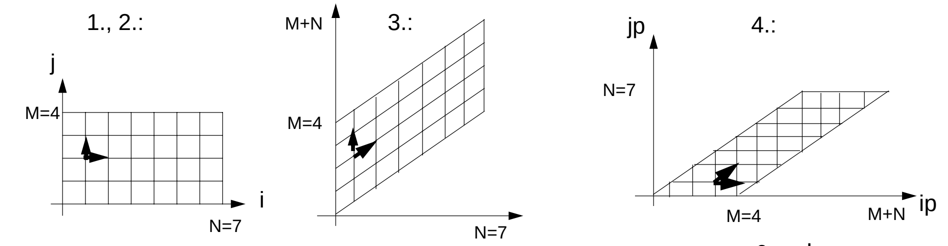
**In the lecture:**

- Explain the steps of the transformation.
- Solution on C-5.22

**Questions:**

- Are there other transformations that lead to a parallel inner loop?

## Solution of the Transformation and Parallelization Example



5.: $\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}1\\0\end{pmatrix}$ $\quad\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}1\\1\end{pmatrix}$

6.: Inverse $\begin{pmatrix}0 & 1\\1 & -1\end{pmatrix}$

7. Bounds:

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

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

## Lecture Parallel Programming WS 2014/2015 / Slide 56d

**Objectives:**

Solution for C-60

**In the lecture:**

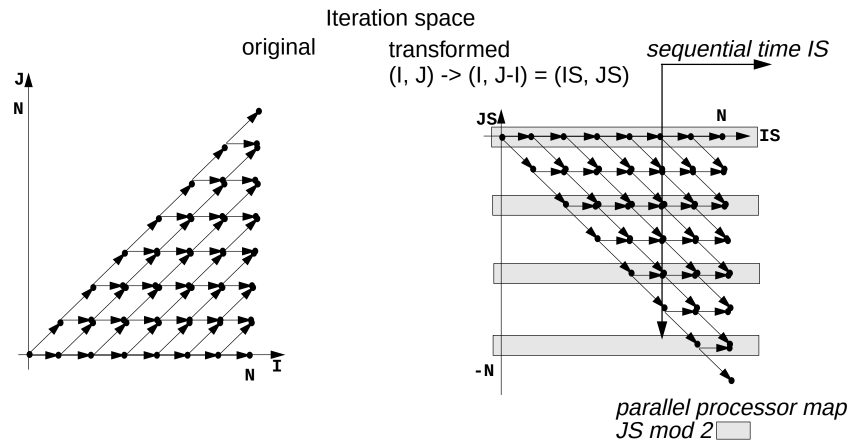
Explain

- the bounds of the iteration spaces,
- the dependence vectors,
- the transformation matrix and its inverse,
- the conditions for being parallelizable,
- the transformation of the index expressions
- the transformation of the loop bounds.

**Questions:**

- Describe the transformation steps.

## 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
```

## Lecture Parallel Programming WS 2014/2015 / Slide 57

**Objectives:**

Example for parallelization

**In the lecture:**

- Explain skewing transformation: f = -1
- Inner loop in parallel.
- Explain the time and processor mapping.
- mod 2 folds the arbitrary large loop dimension on a fixed number of 2 processors.

**Questions:**

- Give the matrix of this transformation.
- Use it to compute the dependence vectors, the index expressions, and the loop bounds.

## 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

## Lecture Parallel Programming WS 2014/2015 / Slide 58

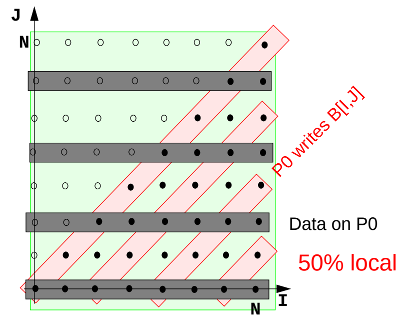
**Objectives:**
Reuse model of iteration spaces

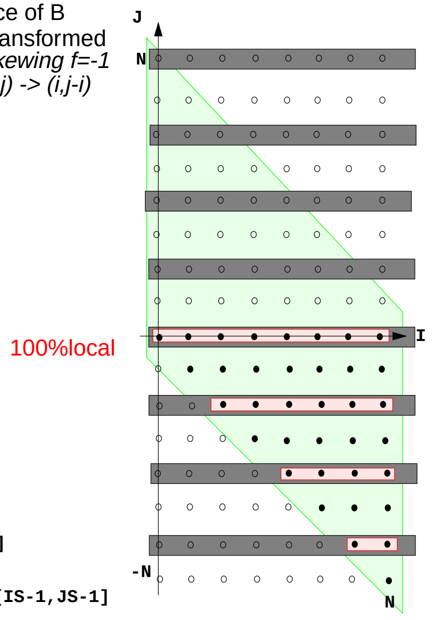
**In the lecture:**
Explain, using examples of index spaces

**Questions:**
- Draw an index space for each of the 3 transformations.

## 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-1+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]
```

## Lecture Parallel Programming WS 2014/2015 / Slide 59

**Objectives:**
The gain of an index transformation

**In the lecture:**
Explain
- local and non-local accesses,
- the index transformation,
- the gain of locality,
- unused memory because of skewing.

**Questions:**
- How do you compute the index transformation using a transformation matrix?