Transformation based parallel programming

Program parallelization techniques.

1. **Program Mapping**
   - Program partitioning (with task aggregation). Dependence analysis.
   - Scheduling & load balancing.
   - Code distribution.
2. **Data Mapping.**
   - Data partitioning.
   - Communication between processors.
   - Data distribution. Indexing of local data.

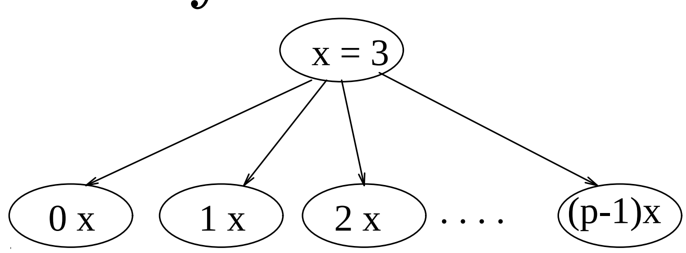
Program and data mapping should be **consistent**.

An Example

Sequential code:

```
x=3
For i = 0 to  p-1.
    y(i)= i*x;
Endfor
```

Dependence analysis:



Scheduling: Replicate $x = 3$ (instead of broadcasting).

0	1	2		p-1
x = 3	*x = 3*	*x = 3*	. . .	*x = 3*
0 x	*1x*	*2 x*		*(p-1)x*

SPMD Code:

```
int x,y,i;
x = 3;
i = mynode();
y = i * x;
```

Data and program distribution :

Sequential		Parallel (one node)
Data		
Array y $[0, 1, \ldots, p-1]$	$\Longrightarrow$	Element y
program		
`For i=0 to p-1`	$\Longrightarrow$	$y = i * x$
`    y(i) = i*x`		

Dependence Analysis

- For each task, define the input and output sets.

Example: S : A = C + B

IN(S) = {C,B}

OUT(S) = {A}.

- Given a program with two tasks: S_1, S_2.

  If changing execution order of S_1 and S_2 affects the result. $\Longrightarrow$ S_2 depends on S_1.

- **Type of dependence:**

  1. Flow dependence (true data dependence).
  2. Output dependence. Anti dependence.
     – Useful in a shared memory machine.
  3. Control dependence (e.g. if A then B).

- **Flow Dependence:** $\mathrm{OUT}(S_1) \cap \mathrm{IN}(S_2) \neq \phi$

  $S_1 :$ A $=$ x + B

  $S_2 :$ C $=$ A + 3

  S2 is dataflow-dependent on S1.

- **Output Dependence:** $\mathrm{OUT}(S_1) \cap \mathrm{OUT}(S_2) \neq \phi$.

  $S_1 :$ A $=$ 3

  $S_2 :$ A $=$ x

  S2 is output-dependent on S1.

- **Anti Dependence:** $\mathrm{IN}(S_1) \cap \mathrm{OUT}(S_2) \neq \phi$.

  $S_1 :$ B $=$ A + 3

  $S_2 :$ A $=$ x + 5

  S2 is anti-dependent on S1.

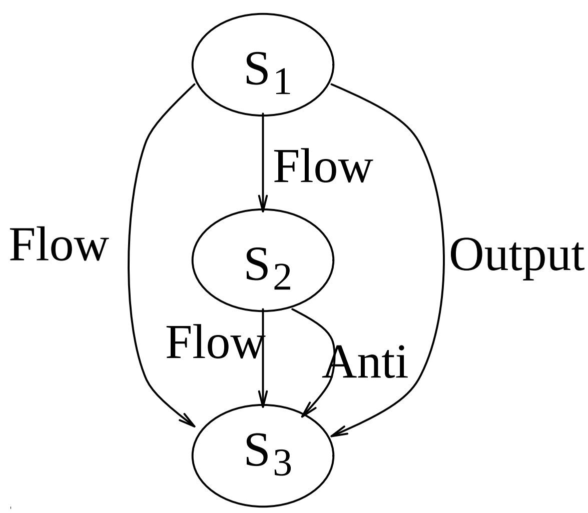
Coarse-grain dependence graph.

Tasks operate on data items of large sizes and perform a large chunk of computations.

Assume each function below only reads input parameters.

Ex:

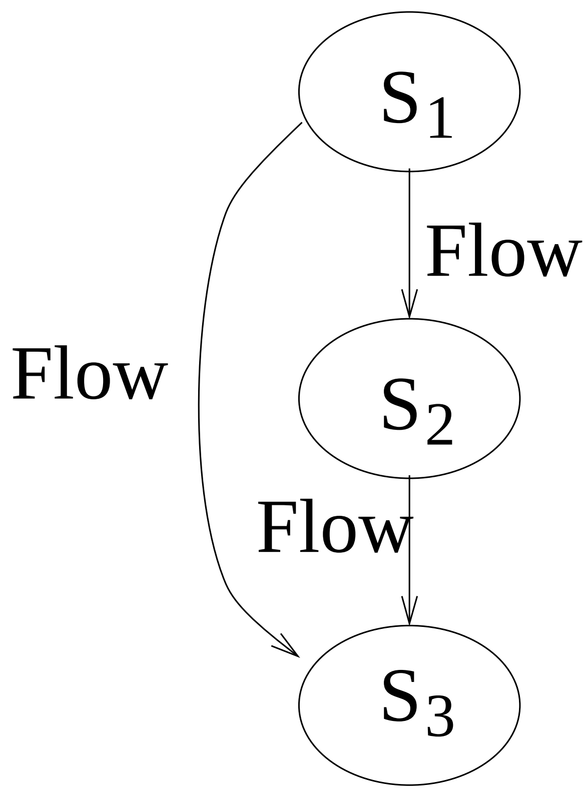
$$
\begin{aligned}
S_1: \quad & A &= \quad & f(X,B) \\
S_2: \quad & C &= \quad & g(A) \\
S_3: \quad & A &= \quad & h(A,C)
\end{aligned}
$$



Delete redundant dependence edges

The deletion should not affect the correctness.

An anti or output dependence edge can be deleted if it is subsumed by another dependence path.

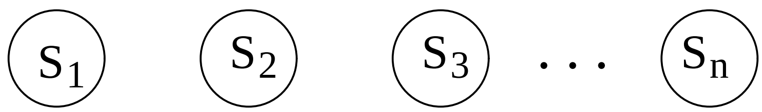


Loop Parallelism

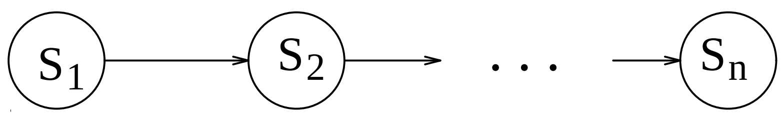
Iteration space – all iterations of a loop and data dependence between iteration statements.

1 D Loop:

```
For i = 1 to n
```
S_i : $a_i = b_i + c_i$

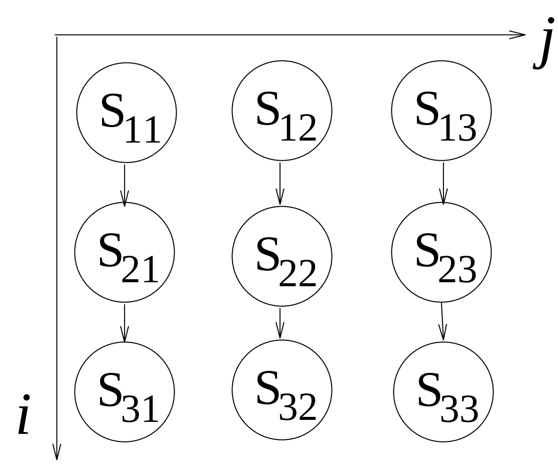



```
For i = 1 to n
```
S_i : $a_i = a_{i-1} - 1$



2 D Loop:

```
For i = 1 to n
  For j = 1 to n
```
S_{ij} : $x_{ij} = x_{i-1,j} + 1$



Program Partitioning

Purpose:

- Increase task granularity (task grain size).
- Reduce unnecessary communication.
- Ease the mapping of a large number of tasks to a small number of processors.

Actions: Group several tasks together as one task.

Loop partitioning techniques:

- Loop blocking/unrolling.
- Interior loop blocking.
- Loop interchange.

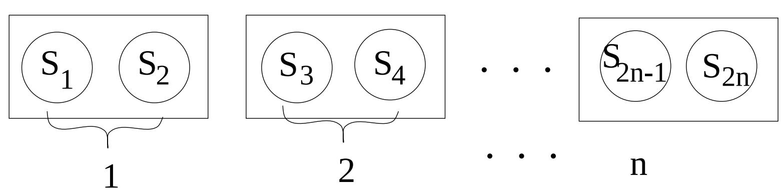
Loop blocking/unrolling

Given:

For i=1 to 2n

$$S_i : a_i = b_i + c_i$$

Block this loop by a factor of 2 or unroll this loop by a factor of 2.



After transformation:

$\Longrightarrow$ `For` $i = 1$ `to` n

`do` S_{2i-1}, S_{2i}

General 1D Loop Blocking

Given: For i = 1 to r*p

S_i : a(i) = b(i)+c(i)

Block this loop by a factor of r:

For j = 0 to p-1

```
For i = r*j+1 to r*j+r
    a(i) = b(i)+c(i)
```

SPMD code on p nodes.

```
me=mynode();
For i = r*me+1 to r*me+r
        a(i) = b(i)+c(i)
```

Interior Loop Partitioning

Block the interior loop and make it one task.

Example:

For $i = 1$ to 4

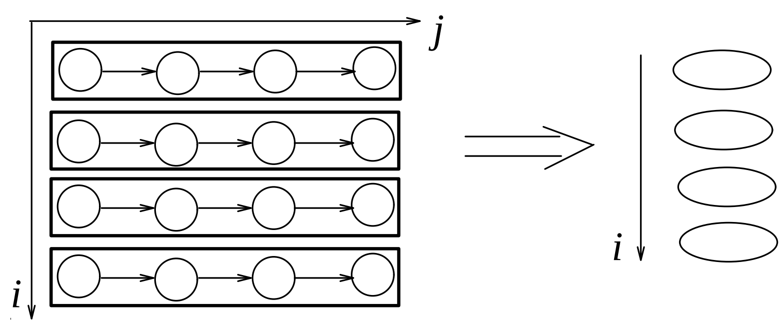
    For $j = 1$ to 4

        $x_{i,j} = x_{i,j-1} + 1$

After blocking:

For $i = 1$ to 4

$$\boxed{\begin{array}{l} \texttt{For}\quad j = 1 \texttt{ to } 4 \\ \quad x_{i,j} = x_{i,j-1} + 1 \end{array}}$$

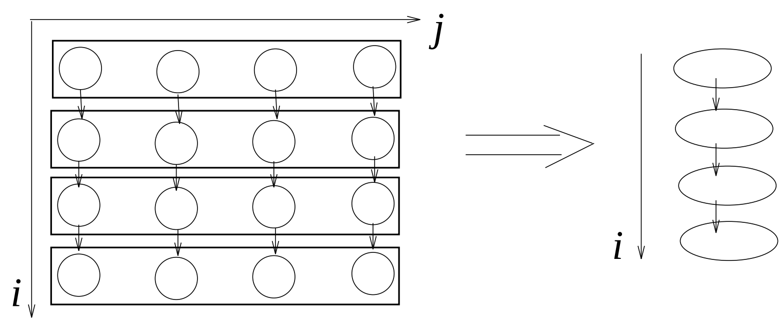


The above example preserves the parallelism.

Partitioning may reduce parallelism

For $i = 1$ to 4

$$\boxed{\begin{array}{l} \texttt{For} \quad j = 1 \texttt{ to } 4 \\ \qquad x_{i,j} = x_{i-1,j} + 1 \end{array}}$$



No inter-task parallelism!

Loop Interchange

Definition: A program transformation that changes the execution order of a loop program.

Actions: Swap the loop control statements.

Example:

For $i = 1$ to 4
    For $j = 1$ to 4
        $x_{i,j} = x_{i-1,j} + 1$

After loop interchange:

For $j = 1$ to 4
    For $i = 1$ to 4
        $x_{i,j} = x_{i-1,j} + 1$

Why loop interchange?

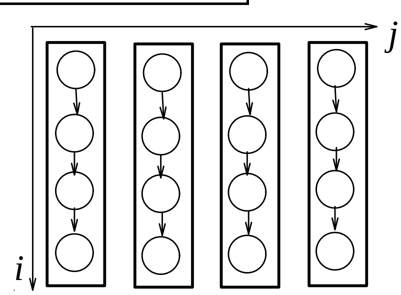
Usage: Help loop partitioning for better performance.

Example. *Interior loop blocking after interchange.*

For $j = 1$ to 4

For $i = 1$ to 4

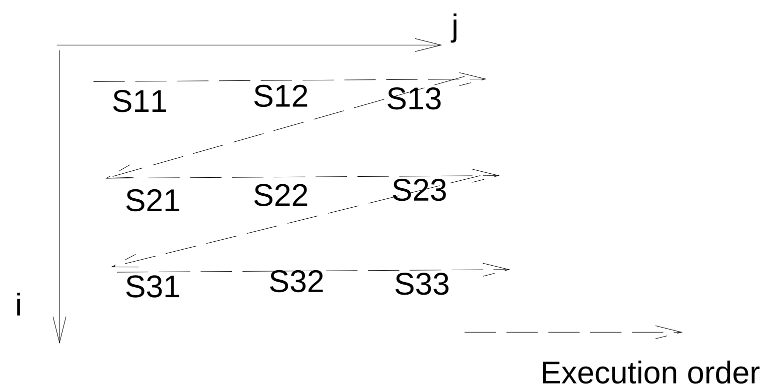
$$x_{ij} = x_{i-1j} + 1$$



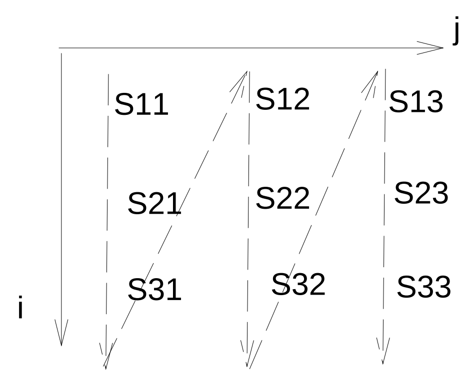
Execution order after loop interchange

Loop interchange alters the execution order.

```
For i = 1 to 3
  For j= 1 to 3
    S i,j :
```




```
For j= 1 to 3
    For i = 1 to 3
   S i,j :
```



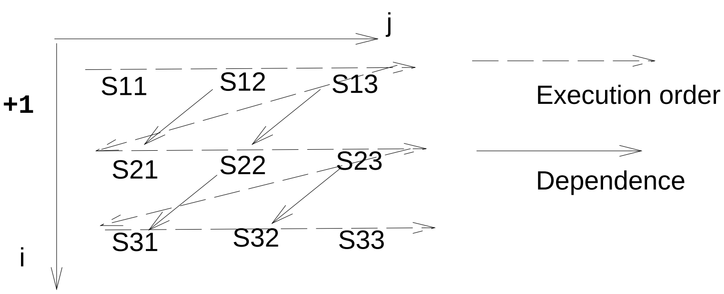
Not every loop interchange is legal

in the sequential code

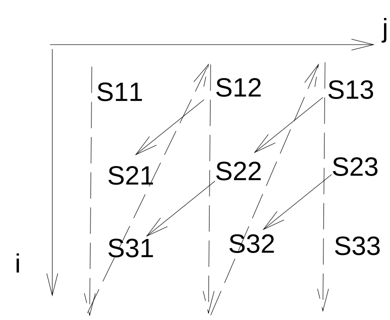
Loop interchange is not legal if the new execution order violates data dependence.

```
For i = 1 to 3
  For j= 1 to 3
    S_{i,j} :  X(i,j)=X(i-1,j+1)+1
```

Legal?



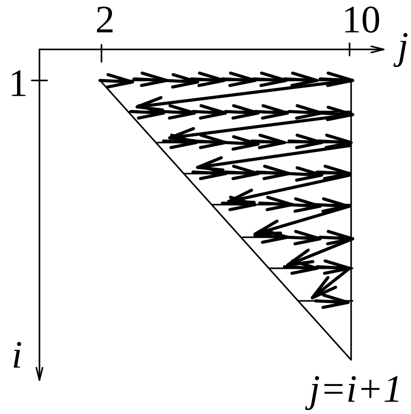

```
For j= 1 to 3
    For i = 1 to 3
      S_{i,j} : X(i,j)=X(i-1,j+1)+1
```

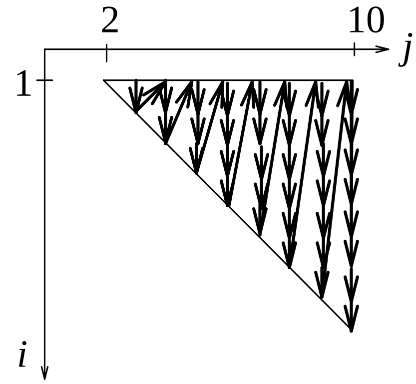


Parallel code execution needs to make sure data dependence is satisfied when loop interchange is used.

Interchanging triangular loops

```
For i=1 to 10        ⟹ For j=2 to 10
  For j=i+1 to 10          For i=1 to j-1
```





Transformation for loop interchange

How to derive the new bounds for i and j loops?

- **Step 1:** List all inequalities regarding i and j from the original code.

$$i \leq 10, \quad i \geq 1, \qquad j \leq 10, \quad j \geq i + 1.$$

- **Step 2:** Derive bounds for loop j.
  - Extract all inequalities regarding the upper bound of j.

    $$j \leq 10.$$

    The upper bound is 10.
  - Extract all inequalities regarding the lower bound of j.

    $$j \geq i + 1.$$

    The lower bound is 2 since i could be as low as 1.
- **Step 3:** Derive bounds for loop i when j

value is fixed (now loop i is an inner loop).

– Extract all inequalities regarding the upper bound of i.

$$i \leq 10, \quad i \leq j-1.$$

The upper bound is $\min(10, j-1)$.

– Extract all inequalities regarding the lower bound of i.

$$i \geq 1.$$

The lower bound is 1.

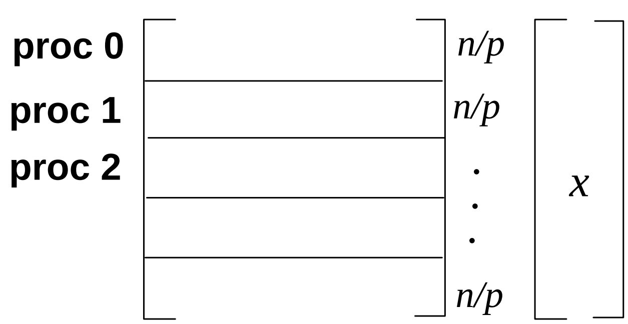
Data Partitioning and Distribution

Data structure is divided into *data units* and assigned to processor local memories.

Why?

- Not enough space for replication for solving large problems.
- Partition data among processors so that data accessing is localized for tasks.

Ex : $y = \mathtt{A}_{n\times n} \cdot x$



Distribute array A among p nodes. But replicate x to all processors.

Corresponding Task Mapping: $(r = n/p)$

P_0	P_1	$\cdots$
$\mathtt{A}_1 x$	$\mathtt{A}_{r+1} x$	
$\mathtt{A}_2 x$	$\mathtt{A}_{r+2} x$	$\cdots$
	$\cdots$	
$\mathtt{A}_r x$	$\mathtt{A}_{2r} x$	

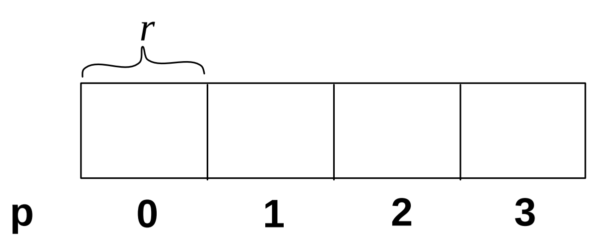
1D Data Mapping Methods

1D array $\longrightarrow$ **1D processors**.

- Assume that data items are counted from $0, 1, \cdots n-1$.
- Processors are numbered from 0 to $p-1$.

Mapping methods: Let $r = \lceil \frac{n}{p} \rceil$.

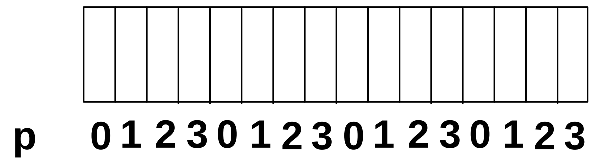
- **1D Block**



$$\text{Data} \quad \Longrightarrow \quad \text{Proc}$$

$$i \qquad\qquad \lfloor \frac{i}{r} \rfloor$$

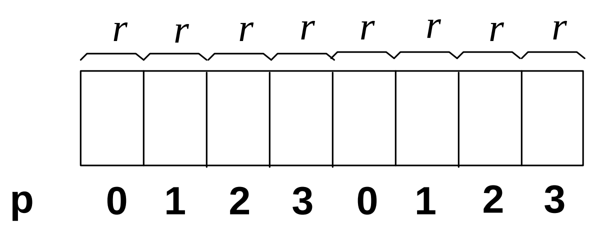
- **1D Cyclic**



$$\begin{array}{ccc} \text{Data} & \Longrightarrow & \text{Proc} \\ i & & i \texttt{ mod } p \end{array}$$

- **1D Block Cyclic.**

  First the array is divided into a set of units using block partitioning (block size b). Then these units are mapped in a cyclic manner to p processors.



$$\begin{array}{ccc} \text{Data} & \Longrightarrow & \text{Proc} \\ i & & \lfloor \frac{i}{b} \rfloor \bmod p \end{array}$$

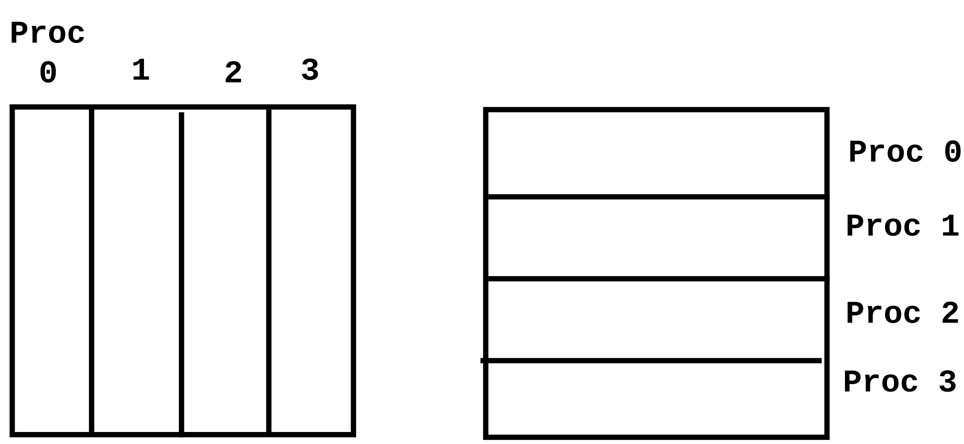
2D array $\longrightarrow$ 1D processors

2D data space is partitioned into a 1D space. Then partitioned data items are counted from $0, 1, \cdots n-1$.

Processors are numbered from 0 to $p-1$.

Methods:

- **Column-wise block.** (call it (*,block))
  Data $(i,j) \Rightarrow$ $Proc \lfloor \frac{j}{r} \rfloor$



- **Row-wise block.** (call it (block,*))
  Data $(i,j) \Rightarrow$ $Proc \lfloor \frac{i}{r} \rfloor$

- **Row cyclic.** (cyclic,*)

  Data $(i, j) \Rightarrow Proc\ i \bmod p$.

- **Others:** Column cyclic. Column block cyclic. Row block cyclic $\cdots$.

2D array $\longrightarrow$ 2D processors

Data elements are counted as (i, j) where $0 \leq i, j \leq \cdots n - 1$.

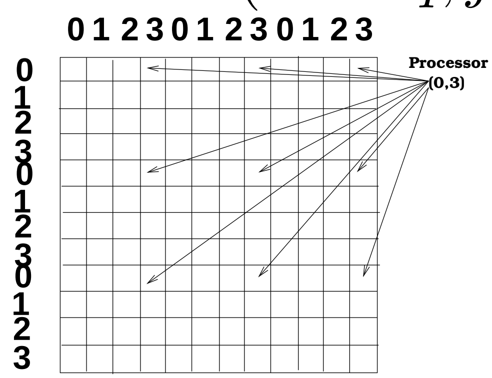
Processors are numbered as (s, t) where $0 \leq s, t \leq \cdots q - 1$ where $q = \sqrt{p}$. Let $r = \lceil \frac{n}{q} \rceil$.

- **(Block, Block)**

  Data $(i, j) \Rightarrow Proc\ (\lfloor \frac{i}{r} \rfloor, \lfloor \frac{j}{r} \rfloor)$

	0	1	2	3
0	Proc (0,0)	Proc (0,1)	Proc (0,2)	Proc (0,3)
1				
2				
3				

- **(Cyclic, Cyclic)**
  Data $(i, j) \Rightarrow$ Proc $(i \; \texttt{mod} \; q, j \; \texttt{mod} \; q,)$



- **Others:** (Block, Cyclic), (Cyclic, Block), (Block Cyclic, Block Cyclic).

Program & data mapping: Consistency

Criteria:

- Sufficient parallelism is provided by partitioning.
- Also the number of distinct units accessed by each task is minimized.

A simple mapping heuristic:

"Owner Computes Rule". If task x modifies data item, then processor that owns this data item executes x.

An Example of "Owner computes rule"

Sequential code:

```
For i = 0 to r*p-1
        S_i : a[i] = 3.
```

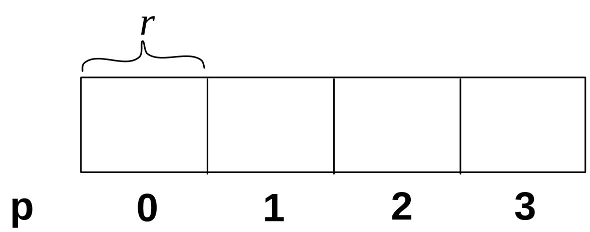
Data distribution:

Map data a(i) to node $proc\_map(i)$.
Data array $a(i)$ are distributed to processors such that if processor x executes $a(i) = 3$, then $a(i)$ is assigned to processor x.

SPMD code on p processors:

```
me=mynode();
For i =0 to rp-1
        if ( proc_map(i) == me) a[i] = 3.
```

SPMD code with 1D block mapping



$$\text{Data } i \quad \Longrightarrow \quad proc\_map(i) = \lfloor \frac{i}{r} \rfloor.$$

Data distribution:

Processor 0 owns data $a(0), a(1), \cdots, a(r-1)$.

Processor 1 owns data $a(r), a(r+1), \cdots, a(2r-1)$.

$\cdots$.

Code distribution:

me=mynode();

For i =0 to rp-1

  if ($proc\_map(i) == me$) a[i] = 3.

Comments: General, but with extra loop and branch overhead.

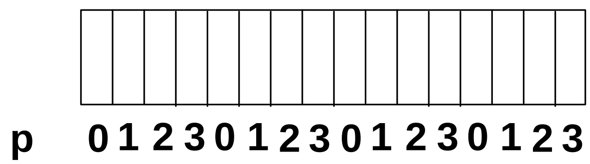
Optimization to remove loop and branch overhead : First, explicitly block the loop code by a factor of r.

```
For j = 0 to p-1
    For i = r*j to r*j+r-1
        a[i] = 3.
```

Optimized SPMD code on p processors:

```
me=mynode();
For i = r*me to r*me+r-1
        a[i] = 3.
```

SPMD code with 1D cyclic mapping



Mapping: $proc\_map(i) = i \bmod p$.

Data distribution:

Processor 0 owns data $a(0), a(p), a(2p), \cdots$.

Processor 1 owns data $a(1), a(p+1), a(2p+1), \cdots$.

Optimized SPMD code on p processors:

```
me=mynode();
For i = me to r*p-1 step p
      a[i] = 3.
```

Global Data Space vs. Local Address

Sequential program	$\Rightarrow$	Global data address
Distributed program	$\Rightarrow$	Local data address

Data indexing in

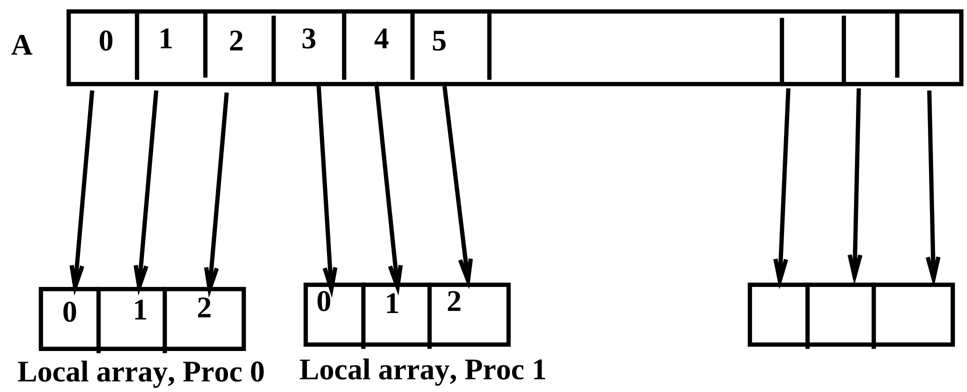
```
me=mynode();
For i =0 to rp-1
        if ( proc_map(i) == me) a[i] = 3.
```

Problem: "a(i)=3" uses "i" as the index function and the value of i is in a range between 0 to $rp - 1$. Each processor has to allocate the entire array!

Data localization: Allocate r units for each processor, translate the global index i to a local index which accesses the local memory only.

From global address to local address

Use 1D block mapping.



SPMD code.

int a[r]; /* Not entire array! */
me=mynode();
For i =0 to rp-1
    if ($proc\_map(i) == me$) a[local(i)] = 3.

Mapping Function for 1D Block:

$$Local(i) = i \bmod r.$$

Ex. p=2, r=3.

Proc 0	Proc 1
$0 \rightarrow 0$	$3 \rightarrow 0$
$1 \rightarrow 1$	$4 \rightarrow 1$
$2 \rightarrow 2$	$5 \rightarrow 2$

Mapping Function for 1D Cyclic:

$$Local(i) = \lfloor \frac{i}{p} \rfloor.$$

Ex. p=2.

proc 0	proc 1
$0 \rightarrow 0$	$1 \rightarrow 0$
$2 \rightarrow 1$	$3 \rightarrow 1$
$4 \rightarrow 2$	$5 \rightarrow 2$
$6 \rightarrow 3$	

Important Mapping Functions

Given: data item i.

- **1D Block**

  Processor ID:

  $$proc\_map(i) = \lfloor \frac{i}{r} \rfloor$$

  Local data address:

  $$Local(i) = i \bmod r$$

- **1D Cyclic**

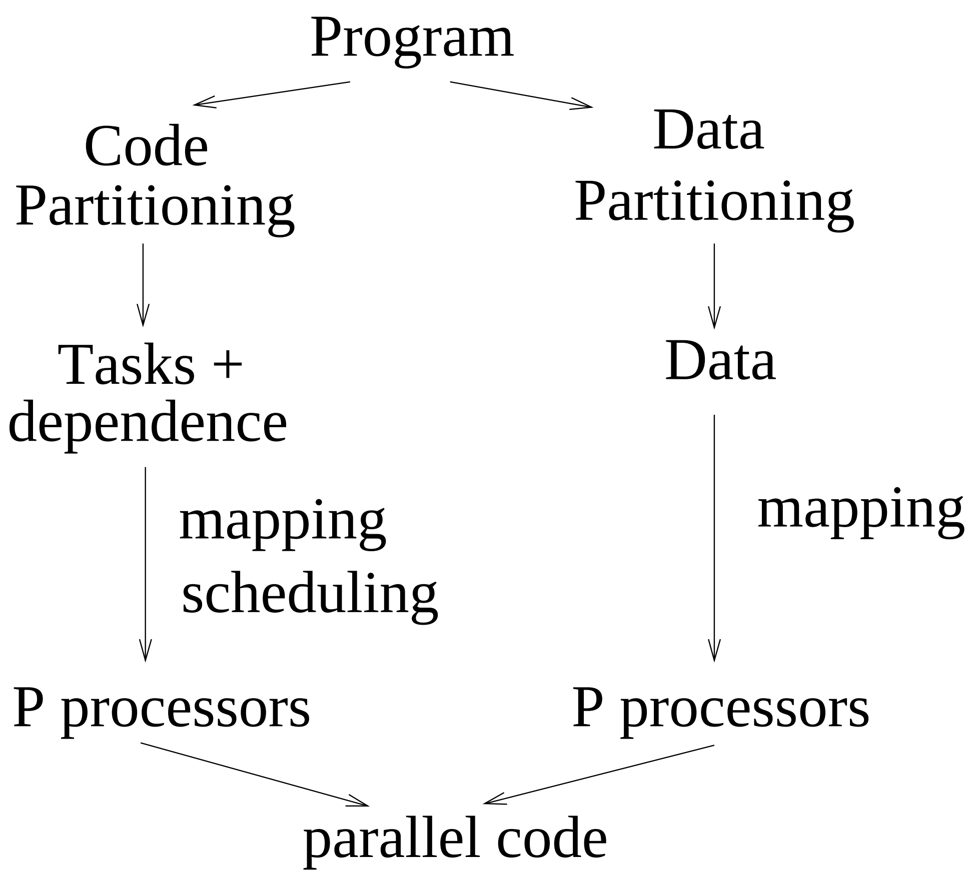
  Processor ID:

  $$proc\_map(i) = i \bmod p$$

  Local data address:

  $$Local(i) = \lfloor \frac{i}{p} \rfloor.$$

Program Parallelization



Techniques

- Cyclic/block partitioning
- Loop interchange, unrolling, blocking
- Dependence analysis
- Task scheduling
- Task mapping. Data mapping.
  (cyclic/ block mapping)
- Data indexing and communication.