Introduction to Parallel Processing

東京大学情報基盤中心　准教授　片桐孝洋

Takahiro Katagiri, Associate Professor,
Information Technology Center, The University of Tokyo

台大数学科学中心　科学計算冬季学校

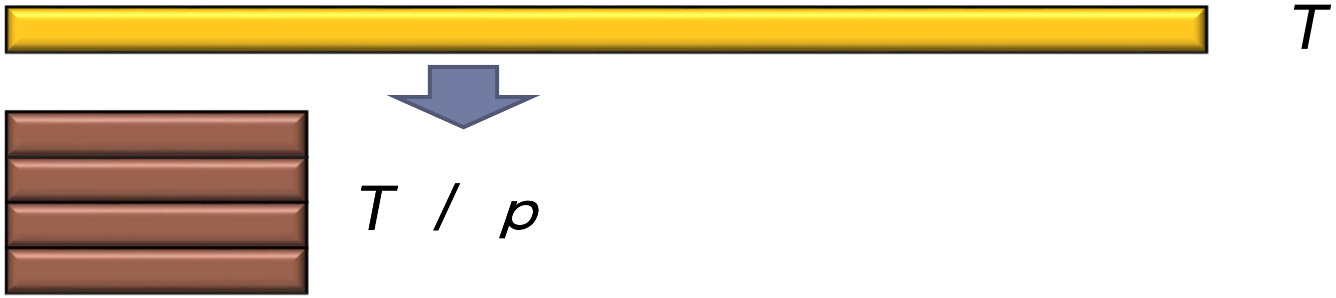
Agenda

1. Basics of Parallel Programming
2. Metrics of Performance Evaluation
3. Data Distribution Methods

Basics of Parallel Programming

What is Parallel Programming?

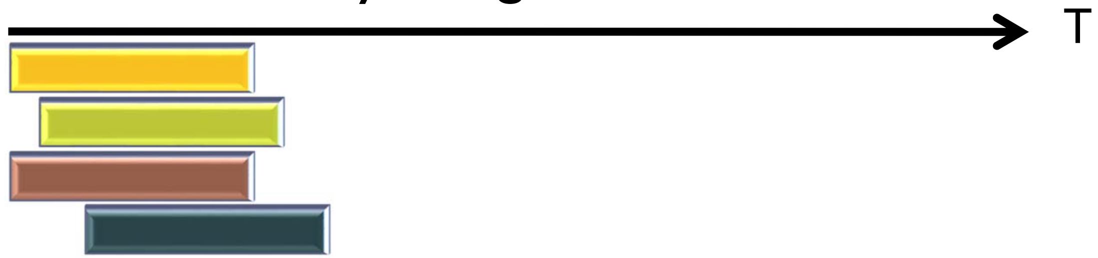
- Making T / p execution time for sequential programming (execution time T) with p machines.



- It seems very easy.
- However, it depends on target process (algorithms).
  - Part of sequential that cannot be parallelized.
  - Communication overheads:
    - Communication set up latency.
    - Data transfer time.

Parallel and Concurrent

- Parallel
  - Physically parallel (time independent)
  - There are many things in a time.



- Concurrent
  - Theoretical parallel (time dependent)
  - There is one thing in a time (with a processor).

  - Time division multiplexing, Pseudo Parallelization.
  - Process scheduling by OS (Round-robin Scheduling)

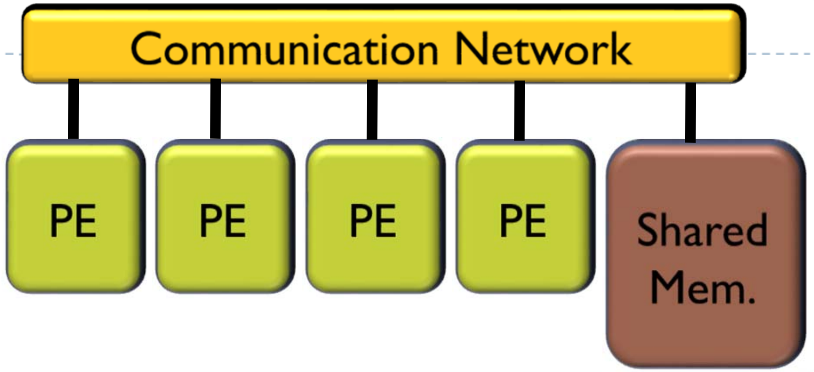
Classification of Parallel Computers

- Classification by Prof. Michael J. Flynn (Stanford U.) (1966)

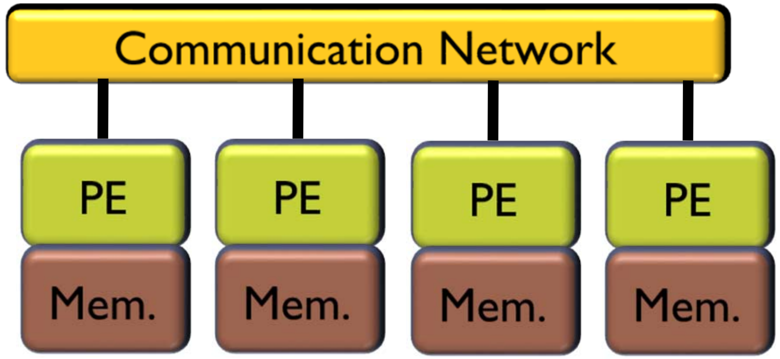
- SISD, Single Instruction Single Data Stream
- SIMD, Single Instruction Multiple Data Stream
- MISD, Multiple Instruction Single Data Stream
- MIMD, Multiple Instruction Multiple Data Stream

Classification of Parallel Computers by Memory Types

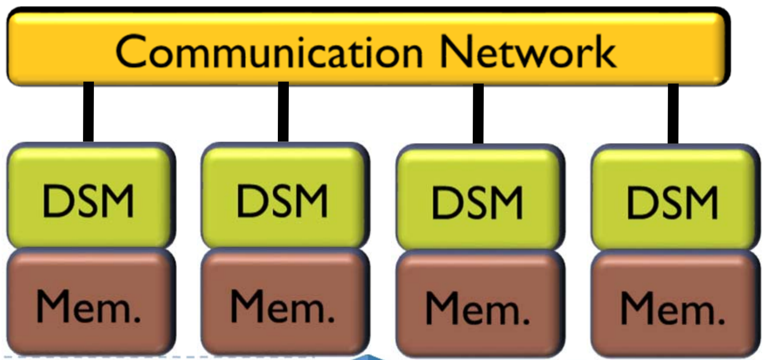
1. Shared Memory Type
(SMP,
Symmetric Multiprocessor)



2. Distributed Memory Type
(Message Passing)

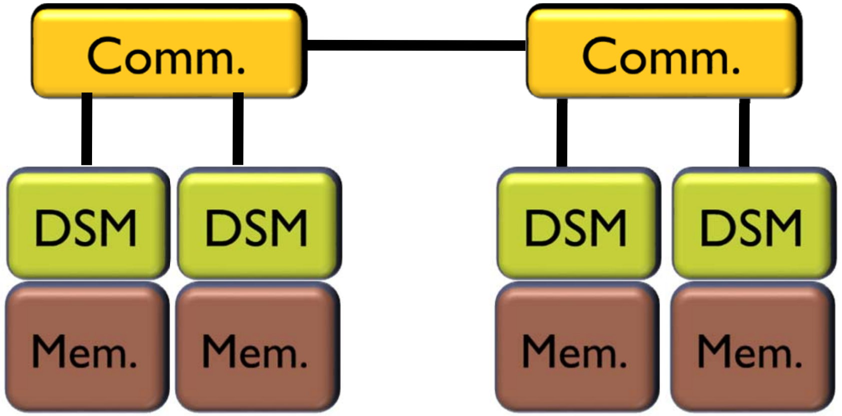


3. Distributed Shared Memory Type
(DSM)



Classification of Parallel Computers by Memory Types

4. Shared and Unsymmetric Memory Type (ccNUMA, Cache Coherent Non-Uniform Memory Access)

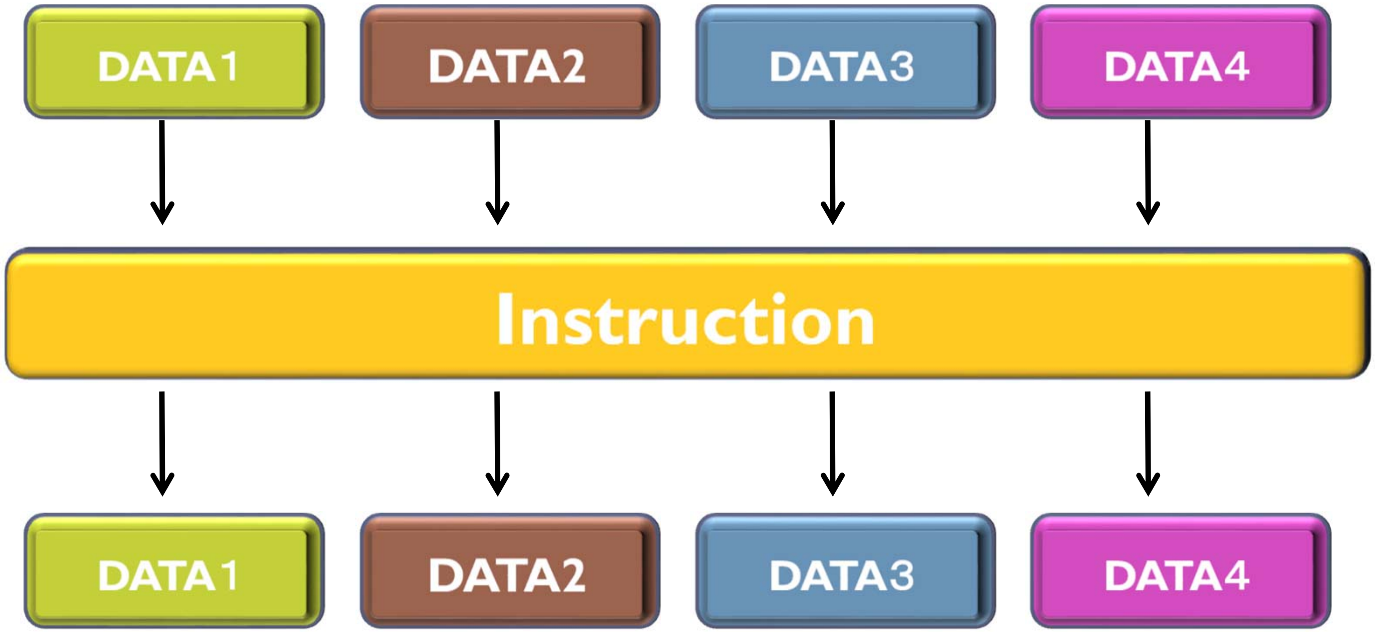


Relationships between Classification of Parallel Computers and MPI

- Target of MPI is distributed memory parallel computers.
  - MPI defines communications between distributed memories.
- MPI can apply shared memory parallel computers.
  - MPI can perform process communication in shared memory.
- Programming model with MPI is SIMD.
  - Program with MPI is only one (= an instruction), but there are several data in the program (such as arrays).

Models of Parallel Programming

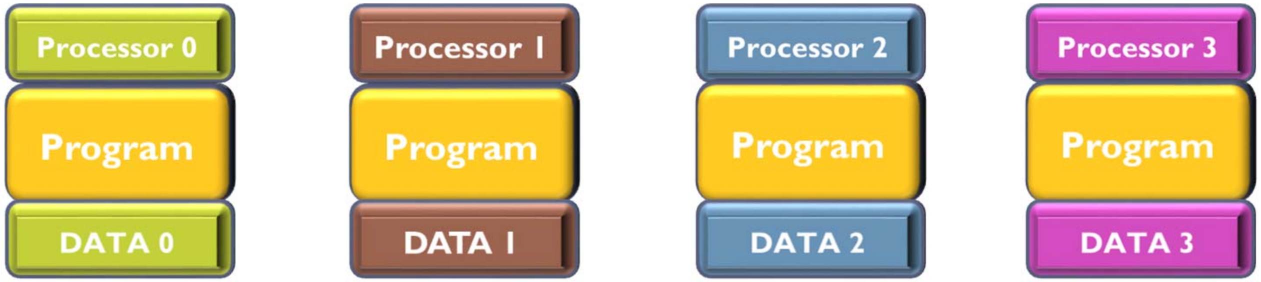
- Behavers of actual programming are MIMD.
- But SIMD is basic model when we program.
  - It is impossible to understand complex behavers.



Models of Parallel Programming

- Parallel Programming Model in MIMD

1. SPMD（Single Program Multiple Data）
   - A common program is copied to all processors when starting parallel processing.
   - Model of MPI (version 1)



2. Master / Worker（Master / Slave）
   - One process（A Master）creates / deletes multiple processes（Workers).

Kinds of Parallel Programming

- Multi Processes
  - MPI（Message Passing Interface）
  - HPF（High Performance Fortran）
    - Fortran Compiler with Automatic Parallelization.
    - Programmer describes data distribution explicitly.
- Multi Threads
  - Pthread (POSIX threads)
  - Solaris Thread (Sun Solaris OS)
  - NT thread (Windows NT, After Windows95)
    - Fork and Join are explicitly described for threads.
  - Java
    - Language specification defines threads.
  - OpenMP
    - Programmer describes lines of parallelization.

Difference between process and threads.
•Take care of shared memory or not.
•Distributed Memory
> Process
•Shared Memory
> Thread

Multi processes and Multi threads can be used simultaneously.
> Hybrid MPI / OpenMP executions.

Example of Parallel Processing (1)

- Data parallelism
  - Parallelization to do data distribution.
  - Data operation (= instruction) is same.
  - Example of data parallelism: Matrix-Matrix Multiplication

As same as SIMD

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1*9+2*6+3*3 & 1*8+2*5+3*2 & 1*7+2*4+3*1 \\ 4*9+5*6+6*3 & 4*8+5*5+6*2 & 4*7+5*4+6*1 \\ 7*9+8*6+9*3 & 7*8+8*5+9*2 & 7*7+8*4+9*1 \end{pmatrix}$$

- Parallelization

Shared with all CPUs.

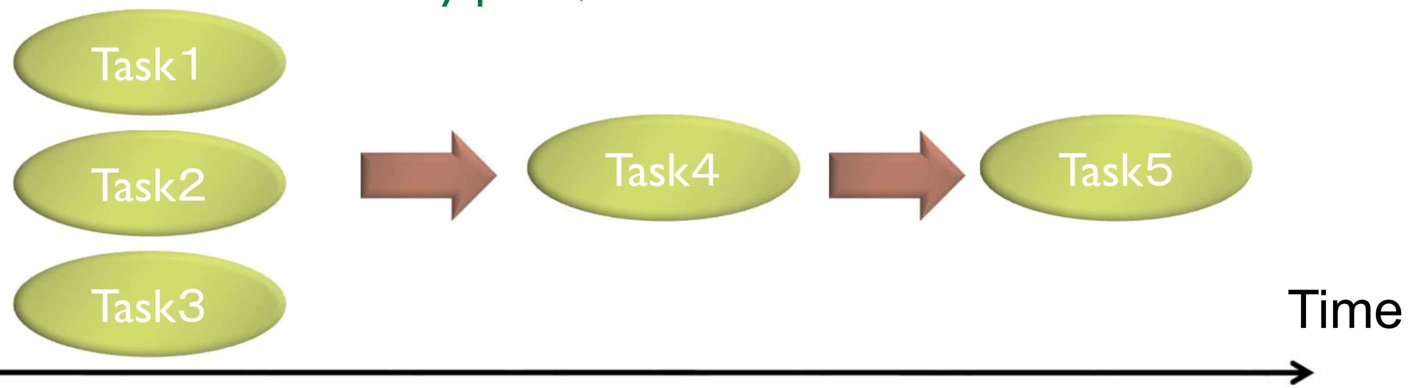
CPU0	$1 \; 2 \; 3$				$1*9+2*6+3*3 \quad 1*8+2*5+3*2 \quad 1*7+2*4+3*1$
CPU1	$4 \; 5 \; 6$	$\begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$	=		$4*9+5*6+6*3 \quad 4*8+5*5+6*2 \quad 4*7+5*4+6*1$
CPU2	$7 \; 8 \; 9$				$7*9+8*6+9*3 \quad 7*8+8*5+9*2 \quad 7*7+8*4+9*1$

Parallel Computation: allocated data is different; but computations are same.

Example of Parallel Processing (2)

- Task Parallelism
  - Parallelization by division of tasks (jobs)
  - Operations of data (=Instructions) may be different.
  - Example of task parallelism : Making Curry.
    - Task1 : Cutting vegetables.
    - Task2 : Cutting meat.
    - Task3 : Boling water.
    - Task4 : Boiling vegetables and meat.
    - Task5 : Stew with curry paste,

- Paralle-lization

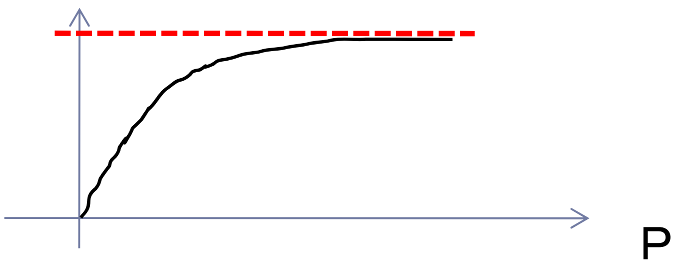


Metrics of Performance Evaluation

Metrics of parallelization

Metrics of Parallelization -Speedup ratio

- Speedup ratio
  - Formula : $S_P = T_S / T_P \quad (0 \le S_p)$
  - T_S : Time for sequential. T_P : Execution with *P* machines.
  - If we obtain $S_P = P$ with *P* machines, it is ideal speedup.
  - If we obtain $S_P > P$ with *P* machines, it is super-linear speedup.
    - Main reason is localizing data access, and ratio of cache hit increases. This causes high efficiency of computation compared to sequential execution.
- Effectiveness of parallelization
  - Formula : $E_P = S_P / P \times 100 \; (0 \le E_p) \; [\%]$
- Saturation performance
  - Limitation of speedup.



Amdahl's law

- Let K be time of sequential computation. Let α be ratio of parallelization in the sequential part.
- The speedup ratio can be calculated as:

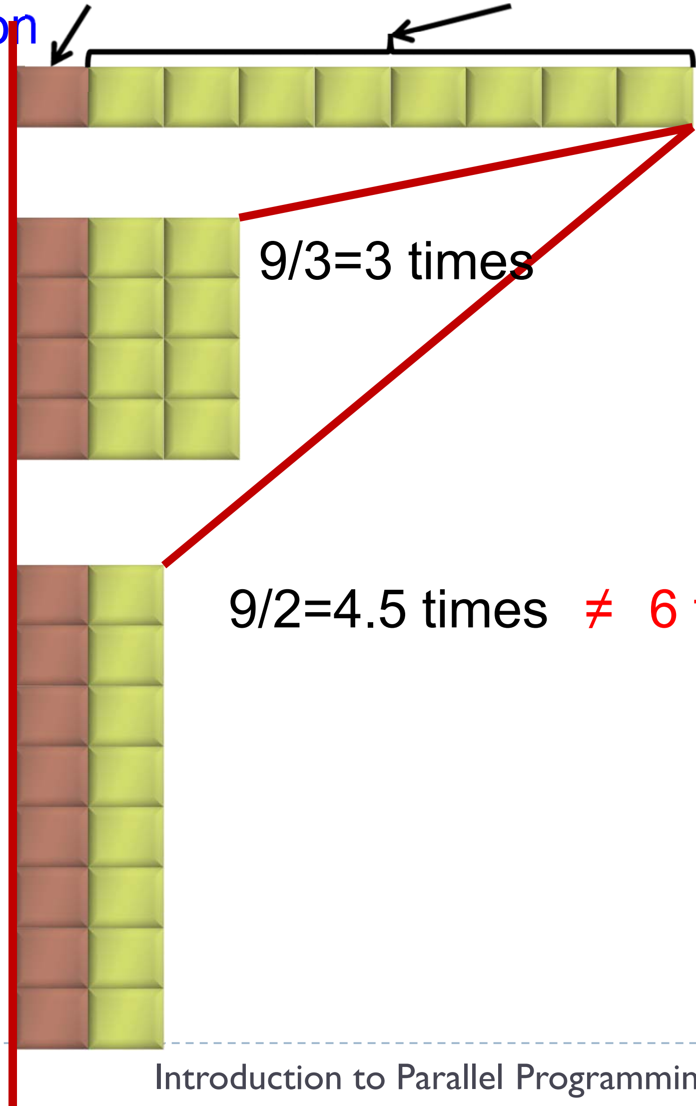
$$S_P = K/(K\alpha/P + K(1-\alpha))$$
$$= 1/(\alpha/P + (1-\alpha)) = 1/(\alpha(1/P - 1) + 1)$$

- (Amdahl's law) With the above formula, we use processors without limitation, such as $(P \to \infty)$, the limitation of speedup ratio is: $1/(1-\alpha)$
  - This indicates that if we can parallelize 90% of total part, and without limitation of number of processors, the maximum speedup is only: 1/(1-0.9) = 10 Times!

> To establish high performance, efforts of higher efficiency of parallelization is crucial.

Amdahl's law : An example

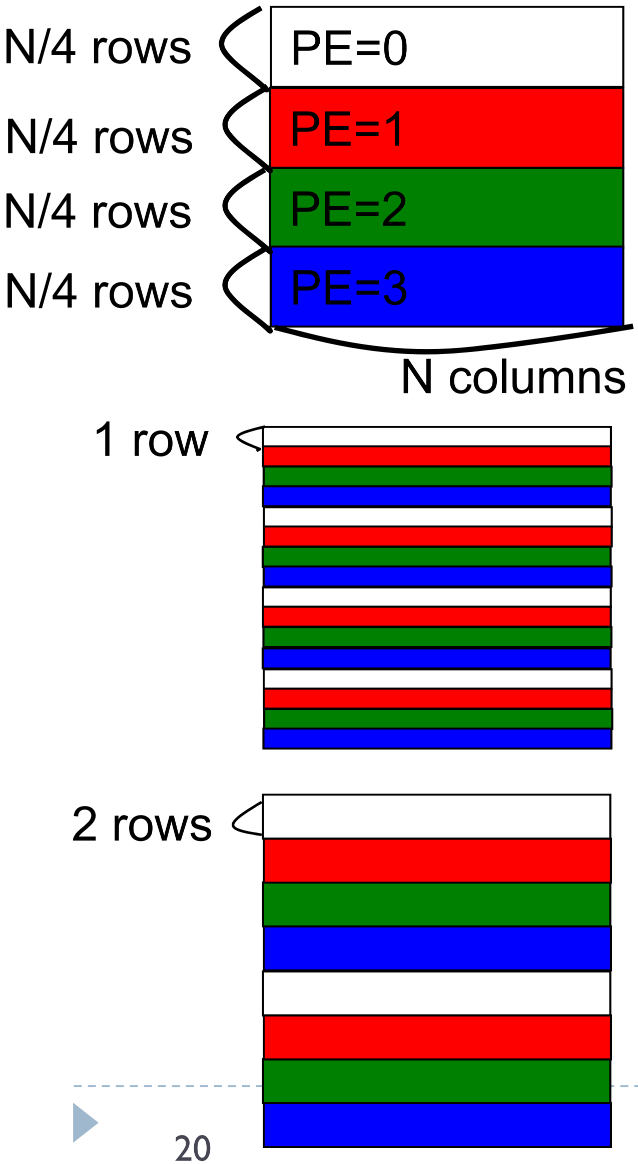
- Sequential Execution
- Parallel Execution (4 parallelisms)
- Parallel Execution (8 parallelisms)



Basic Computations

- "Data structure" is important in sequential processing.
- "Data distribution" is important in parallel processing!
  1. To improve "load balancing" between processes.
     - "Load Balancing" : One of basic operations for parallel processing.
     - Adjustment of grain of parallelism.
  2. To improve "amount of required memory" between processes.
  3. To reduce communication time after computations.
  4. To improve "data access pattern" each process.
     (= It is as same as data structure in sequential processing,.
- Data distribution methods
  - < Dimension Level> : One Dimensional Distribution, Two Dimensional Distribution.
  - < Distribution Level> : Block Distribution, Cyclic Distribution.

One Dimensional Distribution



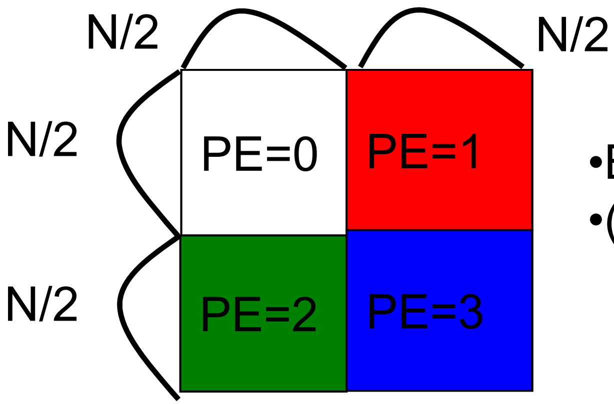
- (row wise) Block Distribution
- (Block, *) Distribution

- (row wise) Cyclic Distribution
- (Cyclic, *) Distribution

- (row wise) Block-cyclic Distribution
- (Cyclic(2), *) Distribution

"2" in this case: <Block Length>

Two Dimensional Distribution



- Block-Block Distribution
- (Block, Block) Distribution

- Cyclic-Cyclic Distribution
- (Cyclic, Cyclic) Distribution

0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3

- 2 Dimensional Block-Cyclic Distribution
- (Cyclic(2), Cyclic(2)) Distribution

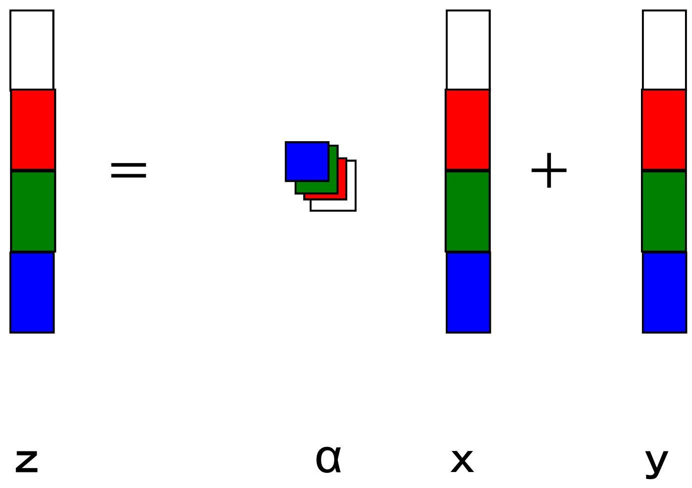
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3

Computation with vectors

- In the following computation:

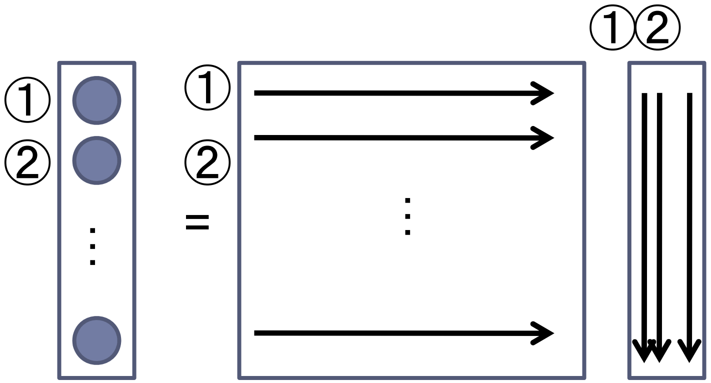
$$z = ax + y$$

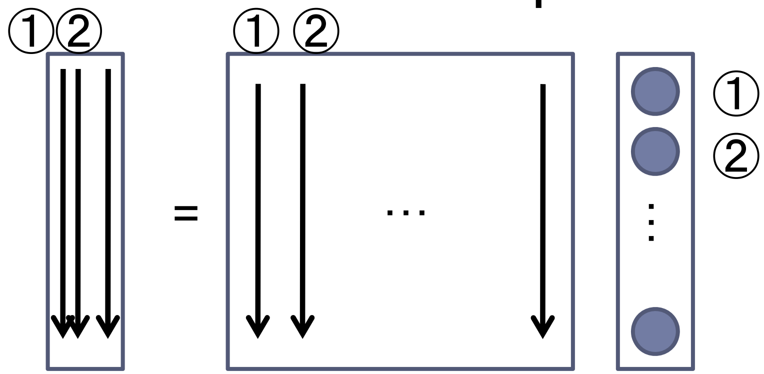
  - , where α is a scalar, and z, x, and y are vectors.
- This can be parallelized with arbitrary distributions.
  - The scalar α is shared with all PEs.
  - While amount of memory for vectors is $O(n)$, but that of memory for scalar is only $O(1)$. →The amount of memory for scalar can be ignored.
  - Computation Complexity: $O(N/P)$
  - It is easy, but not interesting.



Matrix-vector Multiplication

- <Row wise> and <Column wise> computations.
  - Combination between <Data distributions> and <Computations>.






```
for(i=0;i<n;i++){
    y[i]=0.0;
    for(j=0;j<n;j++){
        y[i] += a[i][j]*x[j];
    }
}
```

```
for(j=0; j<n; j++) y[j]=0.0;
for(j=0; j<n; j++) {
   for (i=0; i<n; i++) {
      y[i] += a[i][j]*x[j];
   }
}
```

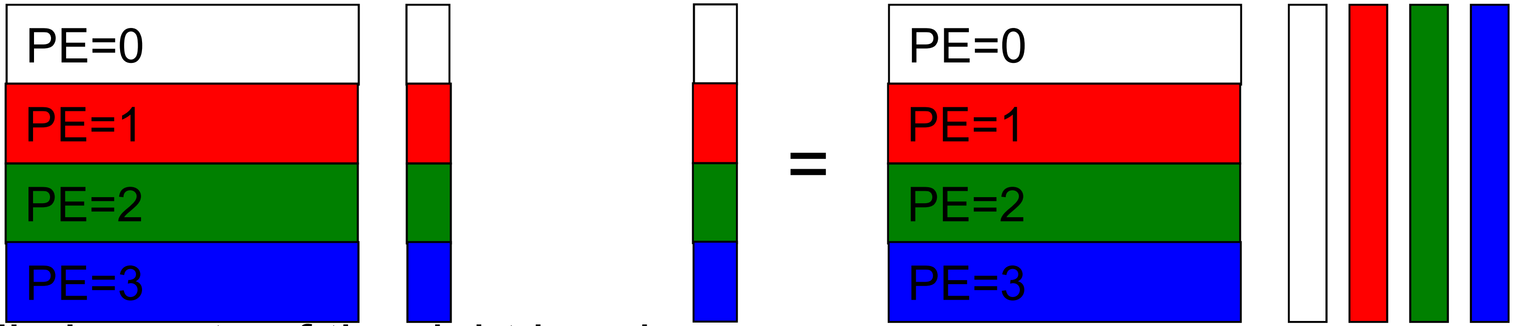
<Row wise>:Natural implementations. For C language.

<Column wise>: For Fortran language.

Matrix-vector Multiplication

Case of <Row wise> Computation

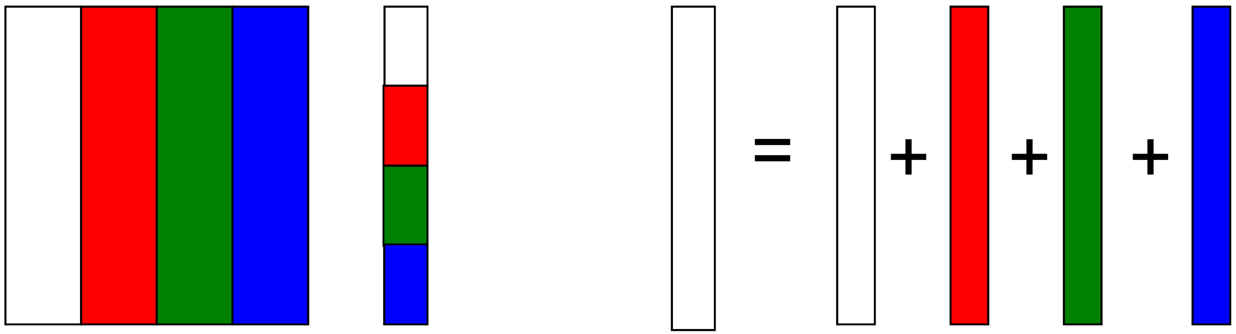
<Row wise> Distribution : Good for row wise computation.



Gather all elements of the right hand vector with MPI_Allgather between all PEs

Local matrix-vector multiplication in each PE.

<Colum wise> Distribution : Good for case that has whole elements of vectors .



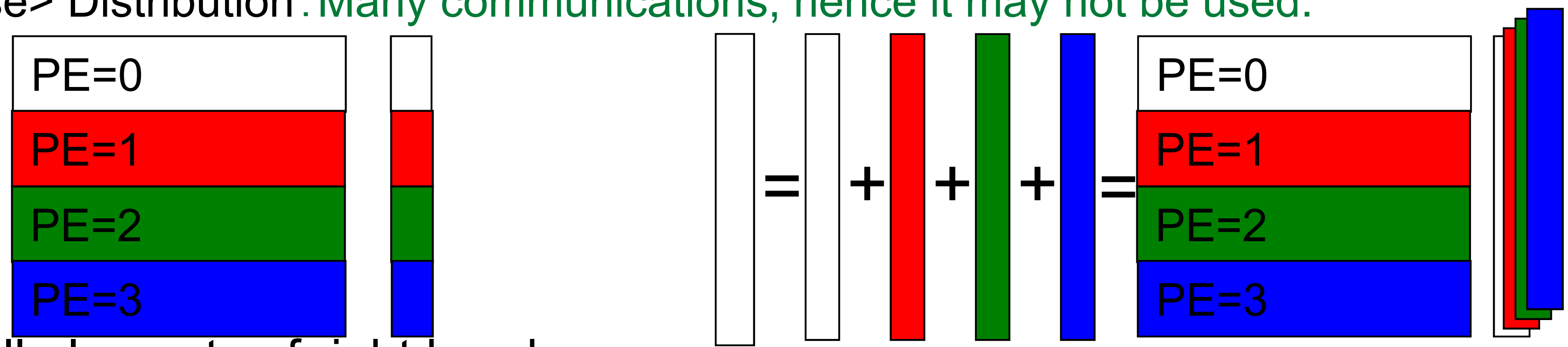
Local matrix-vector multiplication in each PE.

Summation with MPI_Reduce.
*all elements of vector are gathered in a PE.

Matrix-vector Multiplication

Case of <Colum wise> computation

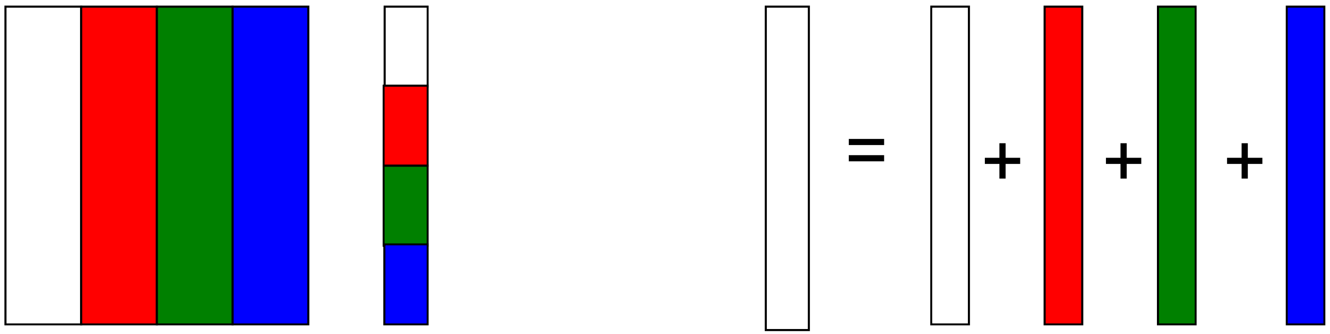
<Row wise> Distribution : Many communications, hence it may not be used.



Gather all elements of right hand vector with MPI_Allgather between all PEs

Summation with MPI_Reduce.

<Colum wise> Distribution : Good for row wise distribution.



Local matrix-vector multiplication in each PE.

Summation with MPI_Reduce.
*all elements of vector are gathered in a PE.