

Solving Linear Equations, FFT and Sorting on Distributed-Memory Architectures

SC3260/5260 High-Performance Computing

Hongyang Sun

(hongyang.sun@vanderbilt.edu)

Vanderbilt University

Spring 2020



Solving System of Linear Equations

- Solving $Ax = b$ as a benchmark to rank world's top supercomputers (<https://www.top500.org/>).
- We will discuss the classical two-step approach:
 - 1) Gaussian Elimination;
 - 2) Back Substitution.
- We assume a 2D block data distribution on $\sqrt{P} \times \sqrt{P}$ procs.
 - Proc P_{ij} holds matrix block A_{ij}
 - Diagonal processor P_{ii} holds vector block b_i and computes x_i

P_{00} A_{00}, b_0 $\rightarrow x_0$	P_{01} A_{01}	P_{02} A_{02}
P_{10} A_{10}	P_{11} A_{11}, b_1 $\rightarrow x_1$	P_{12} A_{12}
P_{20} A_{20}	P_{21} A_{21}	P_{22} A_{22}, b_2 $\rightarrow x_2$

A Simple Example

$$\blacksquare \begin{cases} 2x_0 + 4x_1 + 6x_2 = 8 \\ 3x_0 + 9x_1 + 15x_2 = 21 \\ 5x_0 + 12x_1 + 7x_2 = 2 \end{cases} \quad \rightarrow \quad \begin{bmatrix} 2 & 4 & 6 \\ 3 & 9 & 15 \\ 5 & 12 & 7 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 8 \\ 21 \\ 2 \end{bmatrix}$$

System of linear equations

Matrix form $Ax = b$

A Simple Example

■ **Step 1: Gaussian Elimination**

- Reduce $Ax = b$ to $Ux = y$ (*row echelon form*), where U is an upper triangular matrix (with 1 on diagonal).
- Achieved by dividing a row by a non-zero number and subtracting a row by a multiple of another row.

$$\begin{array}{ccc} \left[\begin{array}{ccc|c} 2 & 4 & 6 & 8 \\ 3 & 9 & 15 & 21 \\ 5 & 12 & 7 & 2 \end{array} \right] & \rightarrow & \left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 2 \end{array} \right] \\ A & & U \quad y \end{array}$$

A lower triangular matrix L records intermediate steps of this process

$$\begin{array}{ccc} & \searrow & \\ & \left[\begin{array}{ccc} 2 & 0 & 0 \\ 3 & 3 & 0 \\ 5 & 2 & -12 \end{array} \right] & \\ & L & \end{array}$$

A Simple Example

■ **Step 2: Back Substitution**

- Solve for x in $Ux = y$ in reverse order from x_{n-1} to x_0

$$\begin{array}{cc} \left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 2 \end{array} \right] & \Rightarrow & \begin{cases} x_0 + 2x_1 + 3x_2 = 4 \\ x_1 + 2x_2 = 3 \\ x_2 = 2 \end{cases} & \Rightarrow & \begin{cases} x_0 = 0 \\ x_1 = -1 \\ x_2 = 2 \end{cases} \\ U & & & & y \end{array}$$

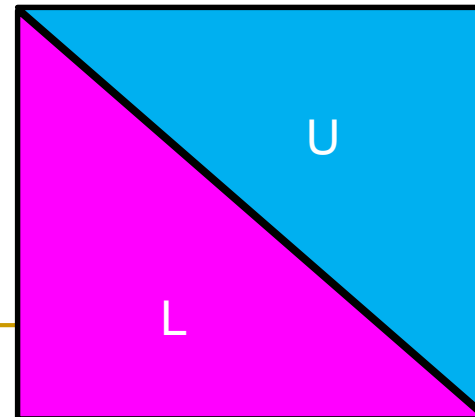
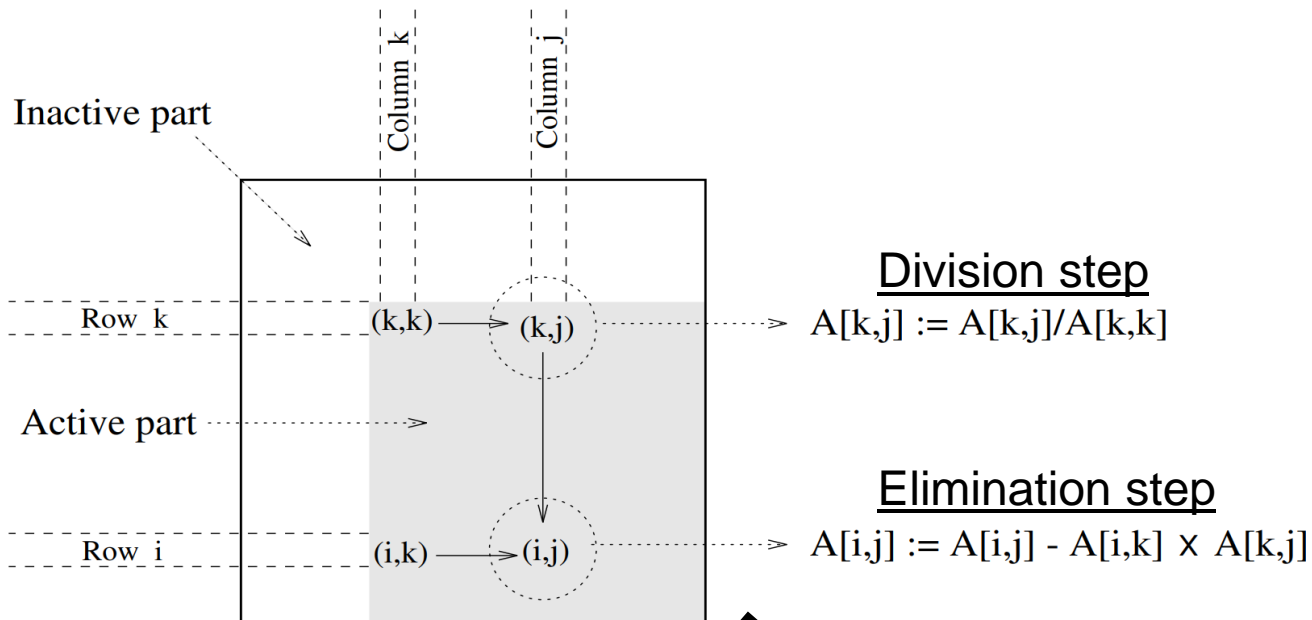
For an $n \times n$ matrix A , Step 1 (Gaussian Elimination) is more expensive taking $O(n^3)$ time, and Step 2 (Back Substitution) takes only $O(n^2)$ time.

Pseudocodes of Two Steps

```
1.  procedure GAUSSIAN_ELIMINATION ( $A, b, y$ )
2.  begin
3.    for  $k := 0$  to  $n - 1$  do           /* Outer loop */
4.      begin
5.        for  $j := k + 1$  to  $n - 1$  do
6.           $A[k, j] := A[k, j] / A[k, k];$  /* Division step */
7.           $y[k] := b[k] / A[k, k];$ 
8.           $A[k, k] := 1;$ 
9.          for  $i := k + 1$  to  $n - 1$  do
10.           begin
11.             for  $j := k + 1$  to  $n - 1$  do
12.                $A[i, j] := A[i, j] - A[i, k] \times A[k, j];$  /* Elimination step */
13.                $b[i] := b[i] - A[i, k] \times y[k];$ 
14.                $A[i, k] := 0;$ 
15.             endfor;           /* Line 9 */
16.           endfor;           /* Line 3 */
17.        end GAUSSIAN_ELIMINATION
```

```
1.  procedure BACK_SUBSTITUTION ( $U, x, y$ )
2.  begin
3.    for  $k := n - 1$  downto  $0$  do /* Main loop */
4.      begin
5.         $x[k] := y[k];$ 
6.        for  $i := k - 1$  downto  $0$  do
7.           $y[i] := y[i] - x[k] \times U[i, k];$ 
8.        endfor;
9.      end BACK_SUBSTITUTION
```

Illustration of Gaussian Elimination



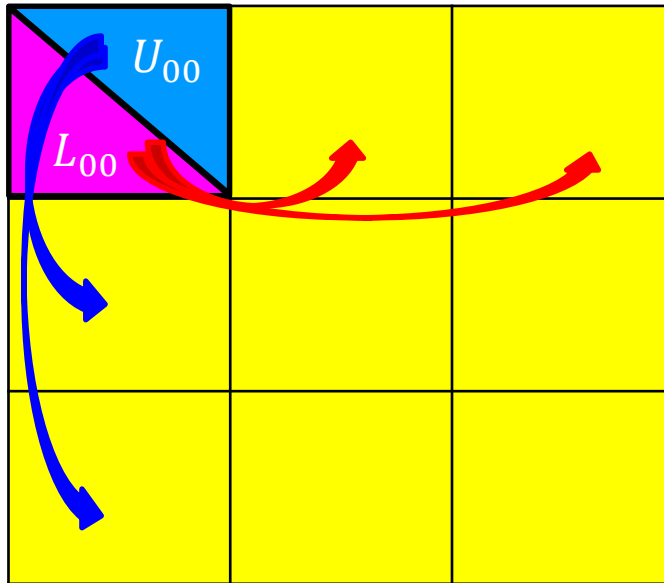
Gaussian Elimination in 2D Blocks

- For step $k = 0, 1, \dots, \sqrt{P} - 1$:
 - a) Processor P_{kk} performs local division and elimination on matrix block A_{kk} (resulting in L_{kk} and U_{kk});
 - b) Processor P_{kk} broadcasts L_{kk} to all processors P_{kj} ($j > k$), and U_{kk} to all processors P_{ik} ($i > k$);
 - c) Processors P_{kj} and P_{ik} ($i, j > k$) perform local division and elimination (resulting in U_{kj} and L_{ik} , respectively);
 - d) Processors P_{kj} broadcasts U_{kj} down to other processors in same column, and processors P_{ik} broadcasts L_{ik} to other processors to the right in same row;
 - e) Processors P_{ij} ($i, j > k$) perform local division and elimination (resulting in modified block \widetilde{A}_{ij}).

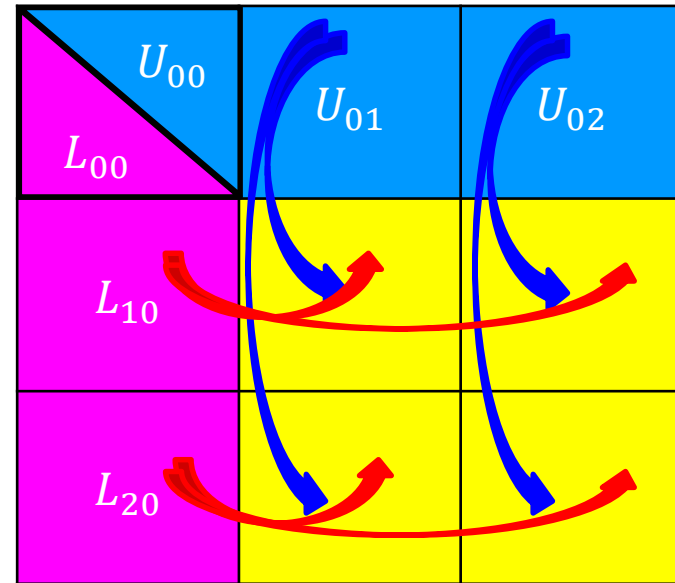
Next slide illustrates one step for $P = 9$ processors 8

Gaussian Elimination in 2D Blocks

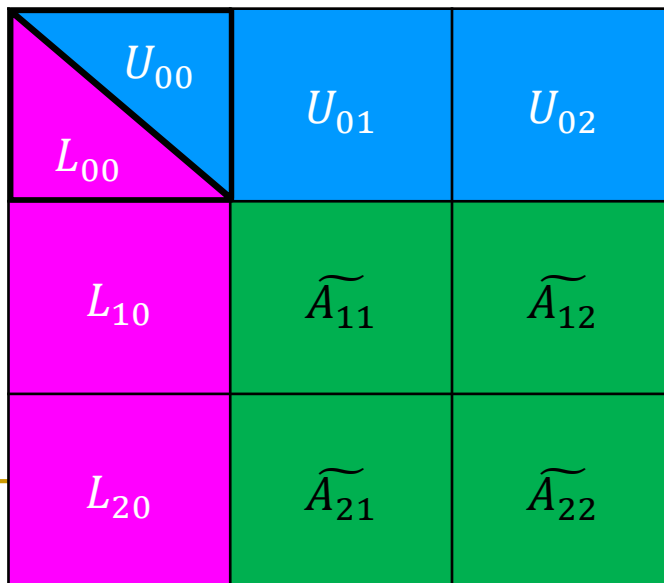
Step $k = 0$
(a)(b)



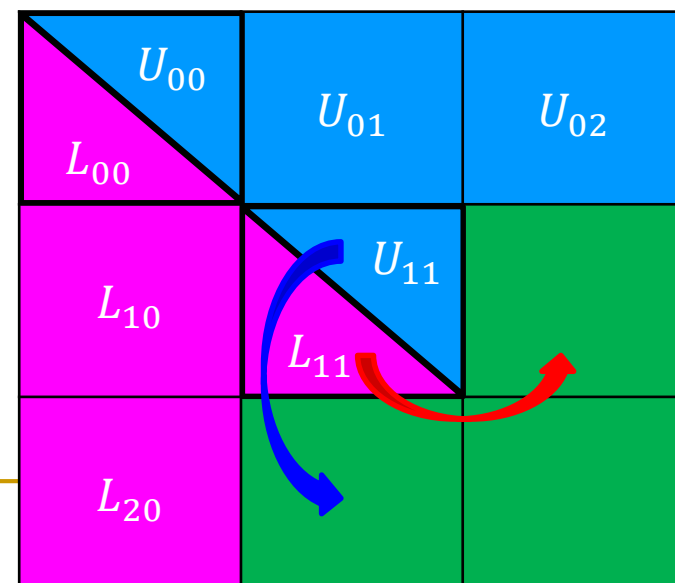
Step $k = 0$
(c)(d)



Step $k = 0$
(e)



Step $k = 1$
(a)(b)

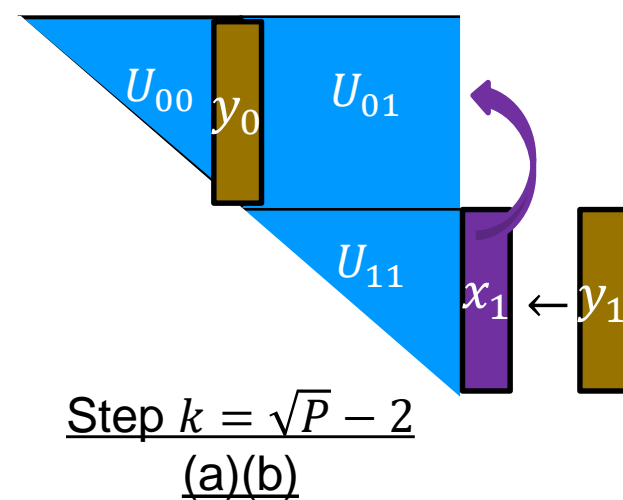
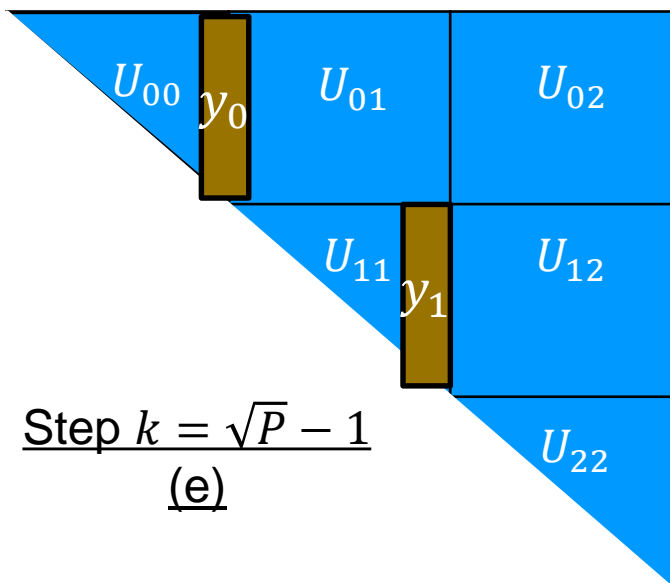
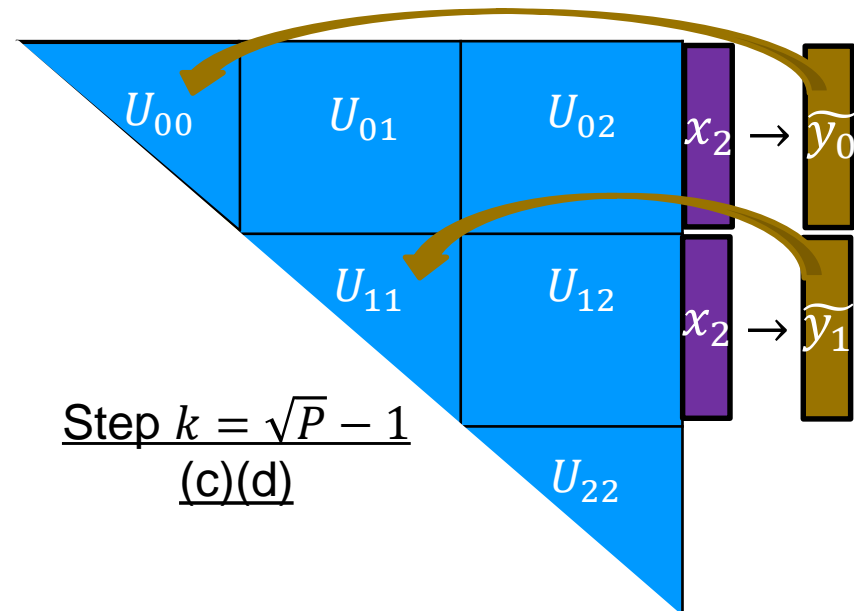
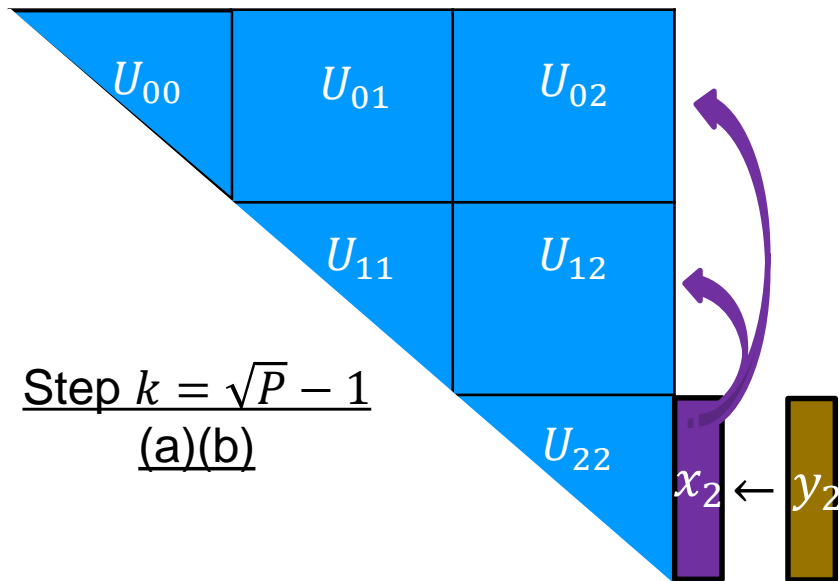


Back Substitution in 2D Blocks

- For step $k = \sqrt{P} - 1, \sqrt{P} - 2, \dots, 0$:
 - a) Processor P_{kk} performs local back substitution on matrix block U_{kk} and y_k (resulting in x_k);
 - b) Processor P_{kk} broadcasts x_k to all processors P_{jk} ($j < k$);
 - c) Processors P_{jk} ($j < k$) computes $\tilde{y}_j \leftarrow U_{jk} \times x_k$;
 - d) Processors P_{jk} ($j < k$) sends \tilde{y}_j to processors $P_{j,j}$;
 - e) Processors $P_{j,j}$ ($j < k$) updates $y_j \leftarrow y_j - \tilde{y}_j$.

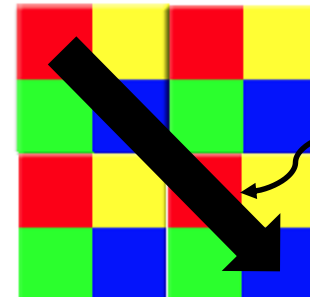
Next slide illustrates one step for $P = 9$ processors

Back Substitution in 2D Blocks

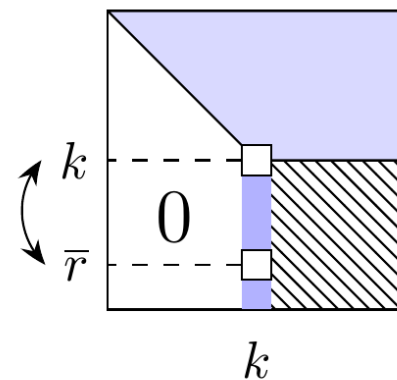


Solving $Ax = b$ in Practice

- **2D Block-cyclic** distribution is used (for better load balance);
- **Partial pivoting** is used during Gaussian Elimination (for numerical stability);
- **LU factorization** of matrix A (for reusing left side).



Order of computation for Gaussian Elimination



Partial pivoting by row permutation

These and other optimizations are implemented in the software package High-Performance Linpack (HPL) (<http://www.netlib.org/benchmark/hpl/>).

Fast Fourier Transform (FFT)

- **Discrete Fourier Transform (DFT):** converts a discrete signal of N samples from time domain $(x_0, x_1, \dots, x_{N-1})$ to frequency domain $(y_0, y_1, \dots, y_{N-1})$, or vice versa:

$$y_j = \sum_{k=0}^{N-1} w_N^{jk} \cdot x_k, \forall j = 0, 1, \dots, N-1$$

where $w_N = e^{-\frac{2\pi i}{N}}$ is the N -th root of unity, called the *twiddle factor*.

- **Fast Fourier Transform (FFT):** any algorithm that computes DFT in $O(n \log n)$ time instead of the naïve $O(n^2)$ complexity.

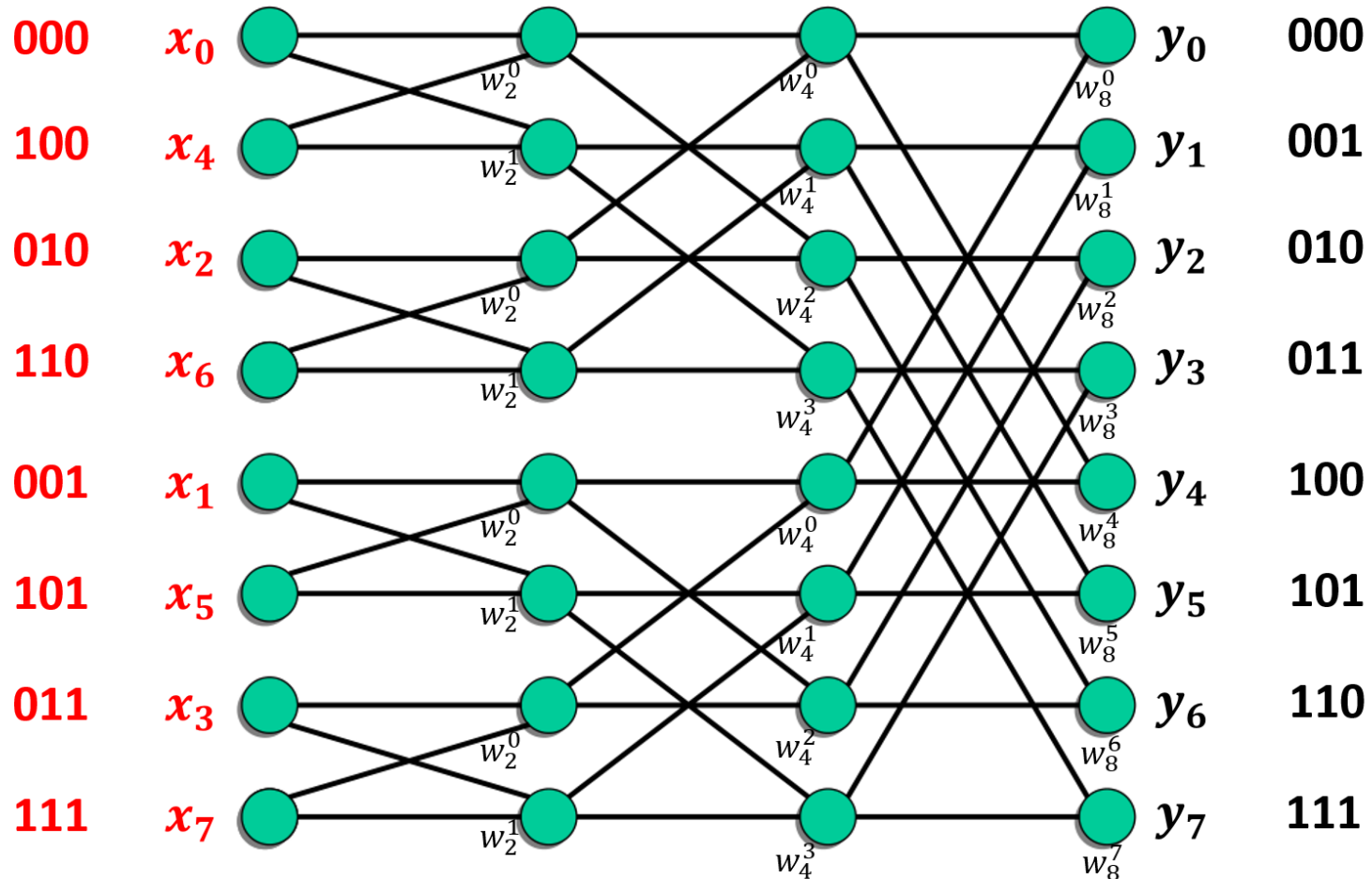
Fast Fourier Transform (FFT)

- FFT has numerous applications in science and engineering, and is considered to be one of the most important algorithms in the 20th century!
- We consider the classical *radix-2* algorithm and study two parallel implementations:
 - 1) **Binary-Exchange Algorithm;**
 - 2) **Transpose Algorithm.**
- For simplicity, we assume that both N and P (number of processors) are powers-of-2, i.e., $N = 2^r$, $P = 2^d$ and $d \leq r$.

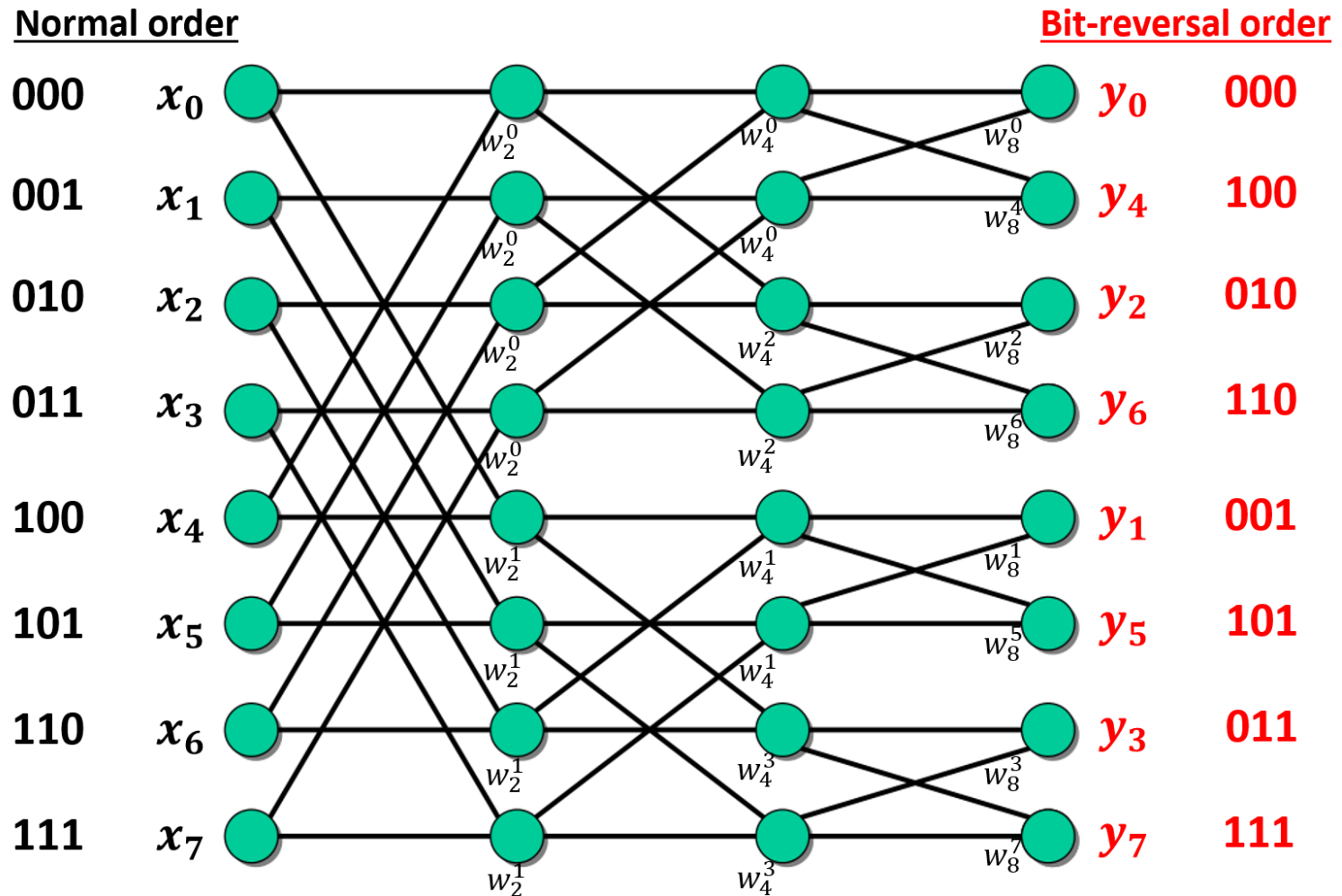
Radix-2 FFT (Decimation in Time)

Bit-reversal order

Normal order



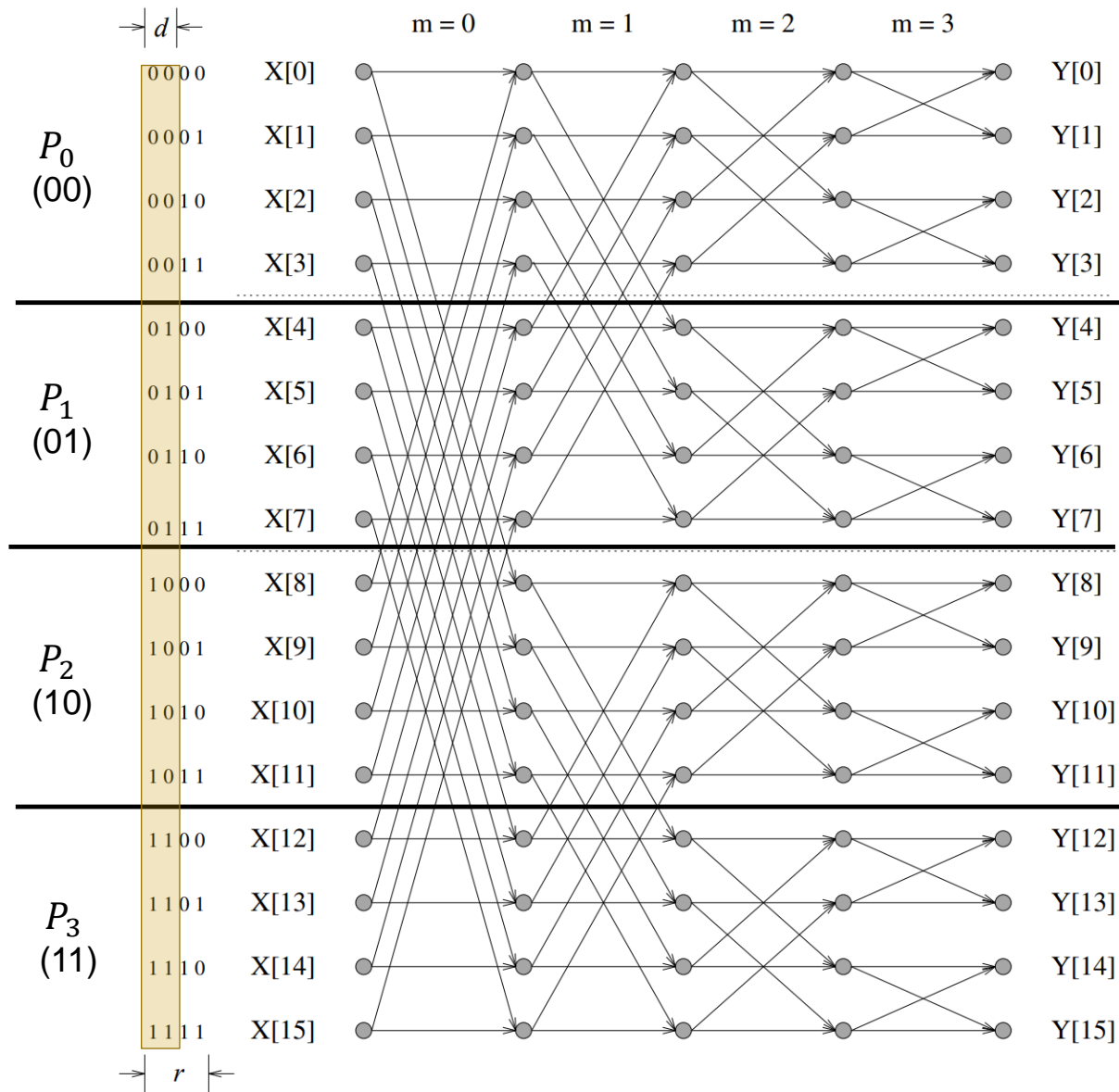
Radix-2 FFT (Decimation in Frequency)



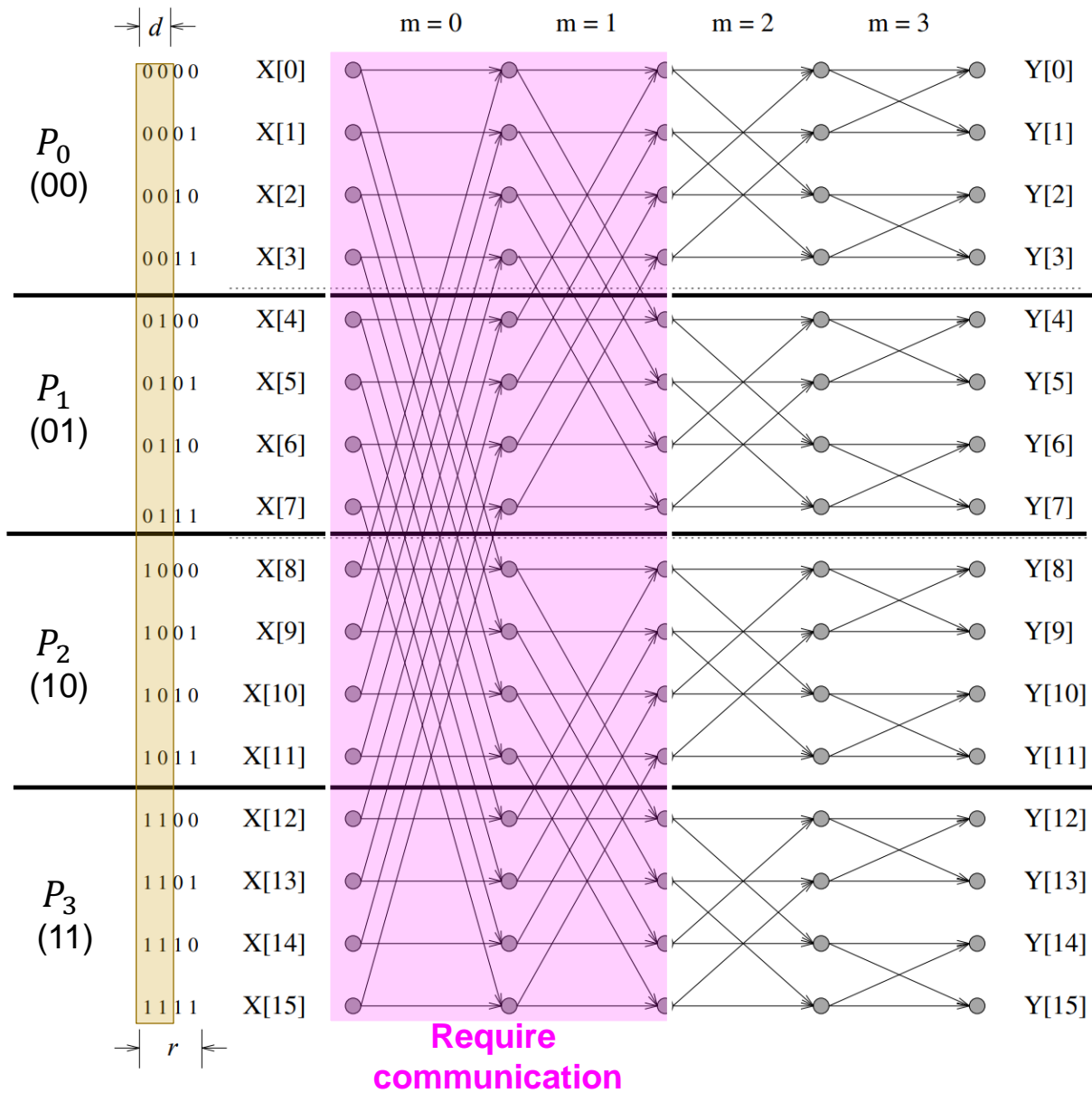
(1) Binary-Exchange Algorithm

- We assume **1D block** data distribution for the DIF implementation of the radix-2 algorithm:
 - Each processor holds N/P consecutive input values in x and responsible for computing the corresponding output values in y .
- The algorithm works in two phases:
 - Phase 1 (w/ communication): step $m = 0, 1, \dots, d - 1$:
 - a) Each processor P_i exchanges N/P numbers with processor $P_{i'}$, where P_i and $P_{i'}$ only differ in the m -th bit (from left);
 - b) Each processor P_i does N/P local complex computations.
 - Phase 2 (local computation): step $m = d, \dots, r - 1$:
 - a) Each processor P_i does N/P local complex computations.

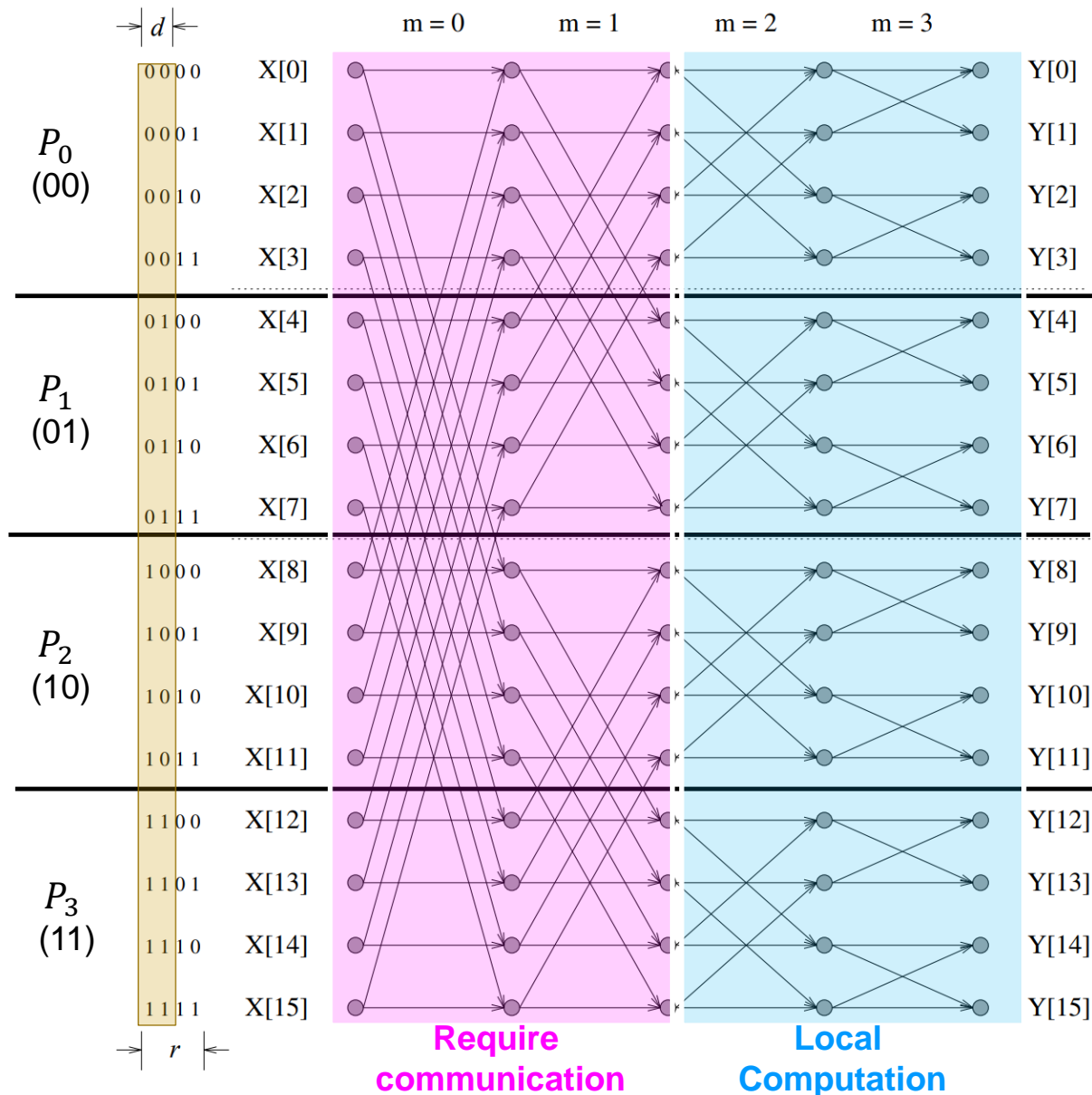
(1) Binary-Exchange Algorithm



(1) Binary-Exchange Algorithm



(1) Binary-Exchange Algorithm



(2) Transpose Algorithm

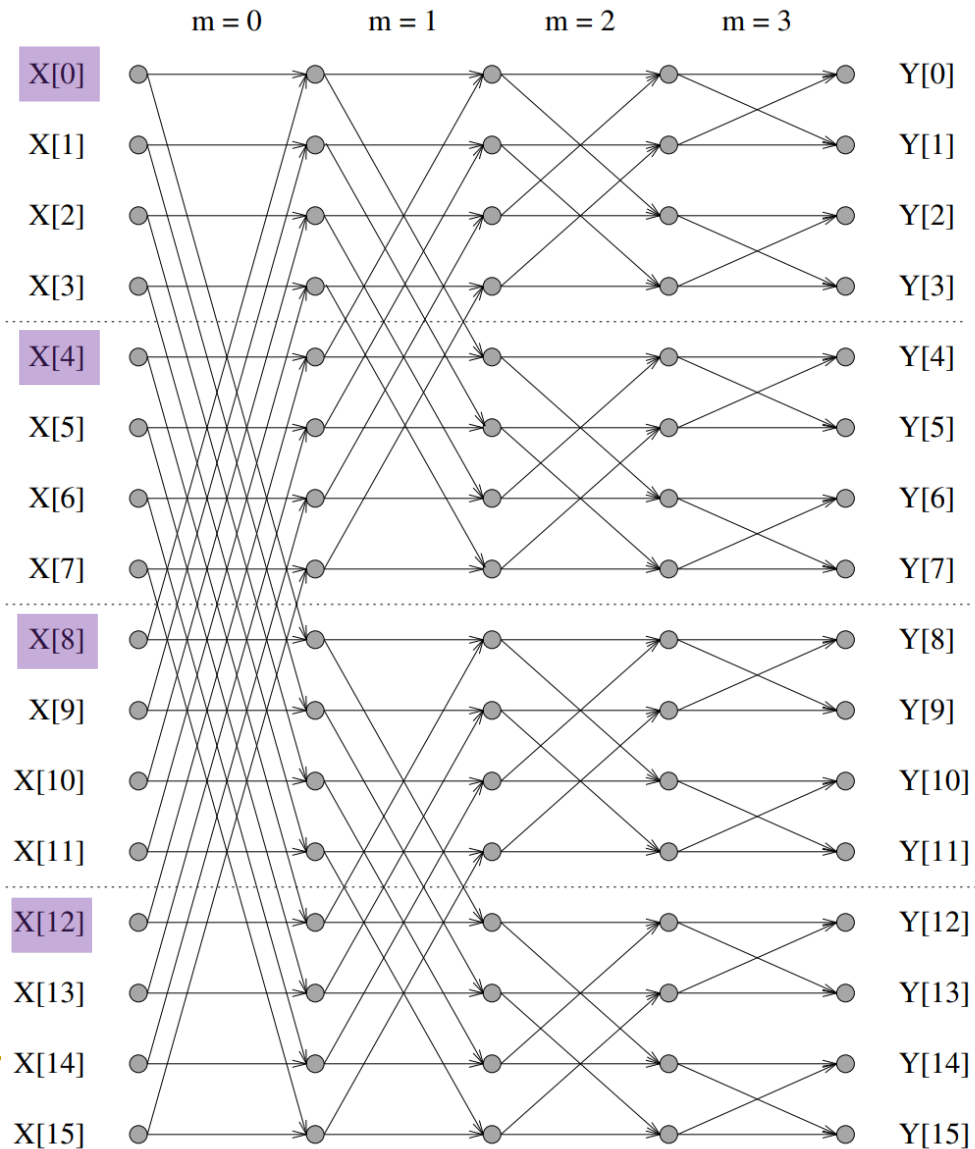
- We assume:

- The input data (vector x) is arranged in a $\sqrt{N} \times \sqrt{N}$ matrix (in row-major order);
- The matrix is distributed in **1D block-cyclic (column)** manner to the processors.

- The algorithm works in three steps:

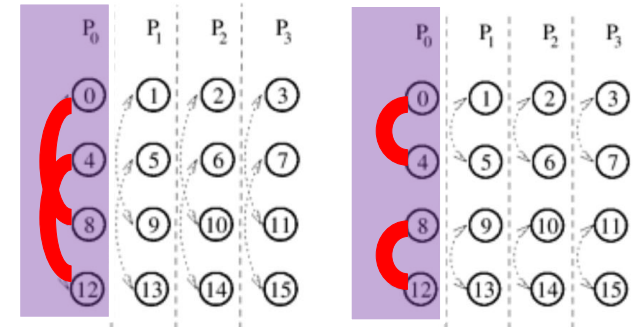
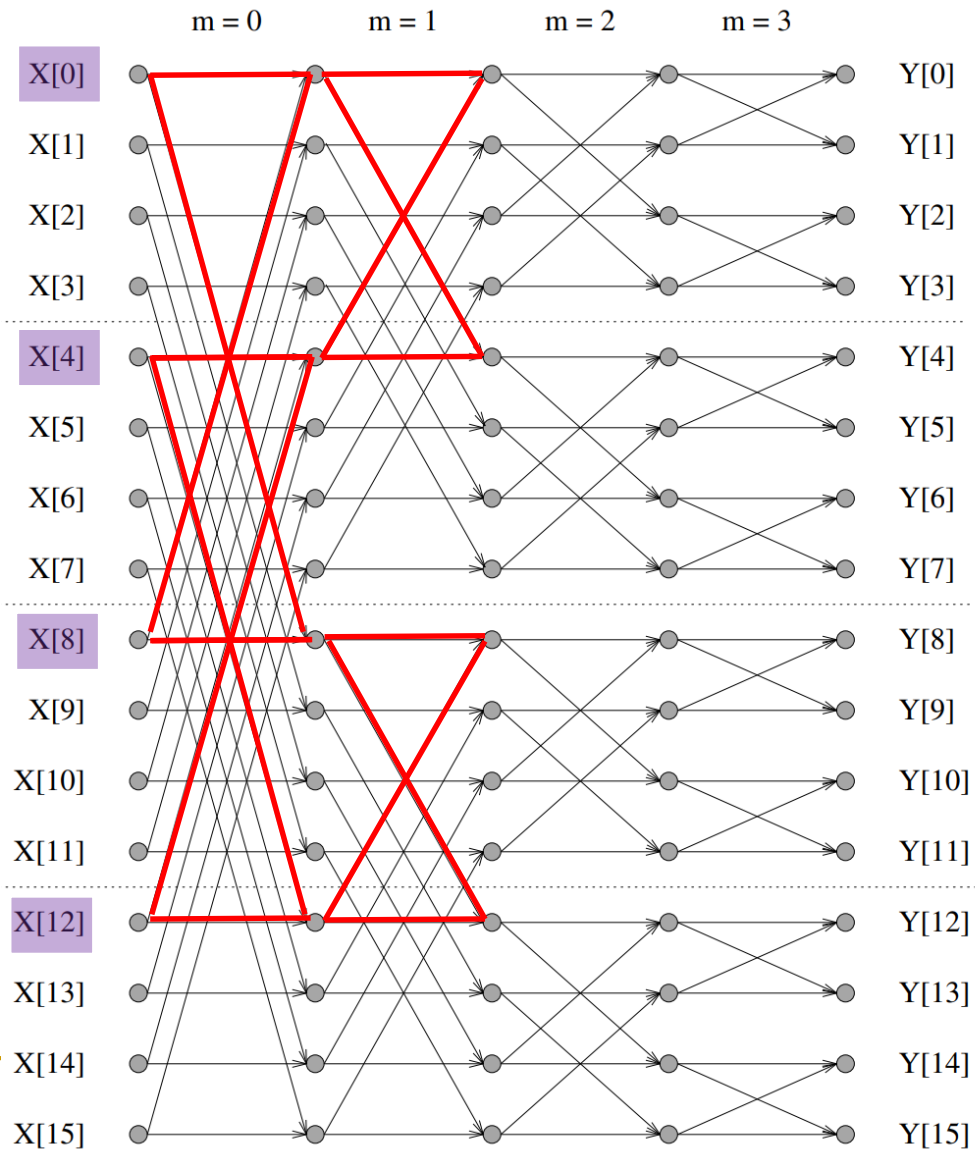
- 1) Each processor does \sqrt{N}/P local FFTs for each column of the matrix it holds;
- 2) Perform a global matrix transpose through an all-to-all personalized communication (MPI_Alltoall);
- 3) Each processor again does \sqrt{N}/P local FFTs for each column of the new matrix it holds.

(2) Transpose Algorithm



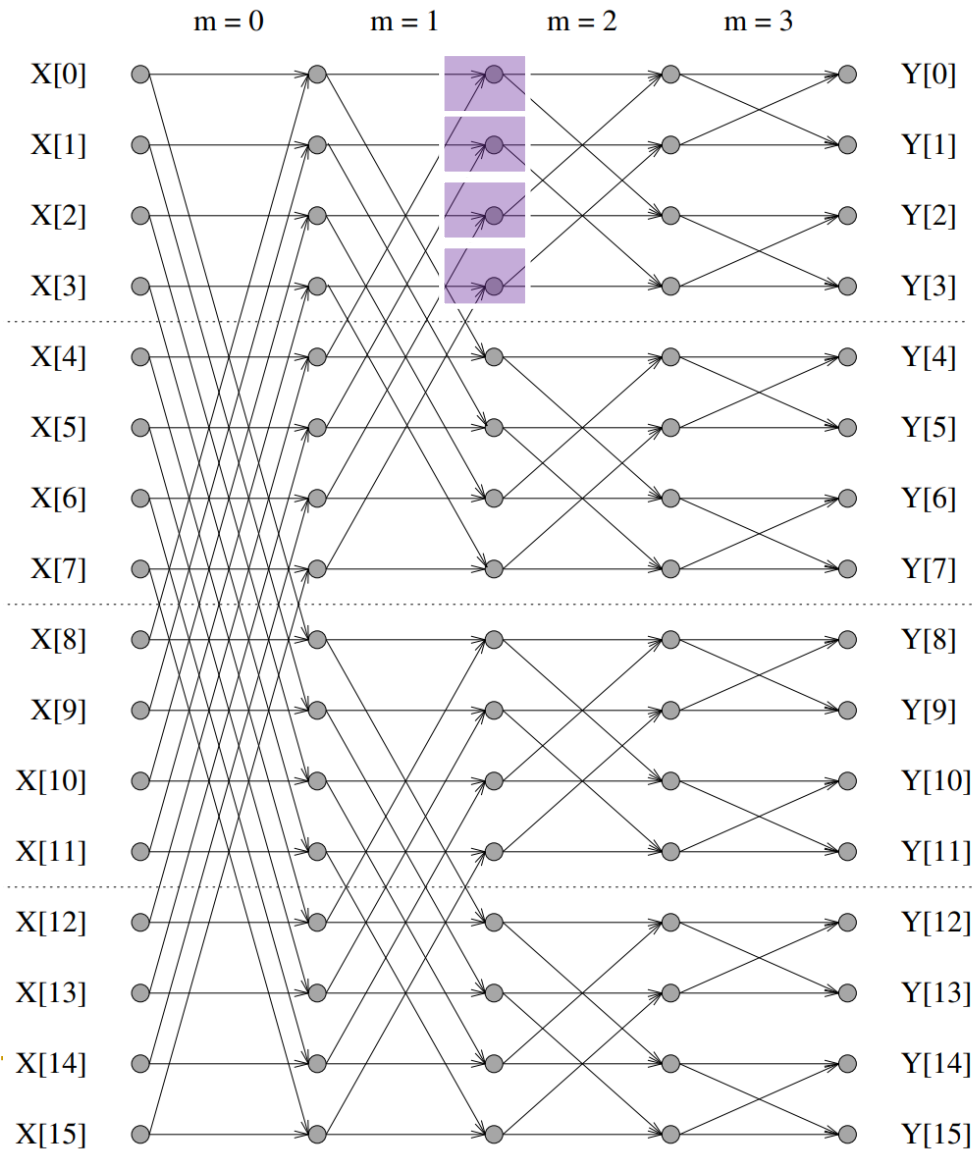
P_0	P_1	P_2	P_3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(2) Transpose Algorithm



Step 1: Local FFT for each column (on same processor)

(2) Transpose Algorithm

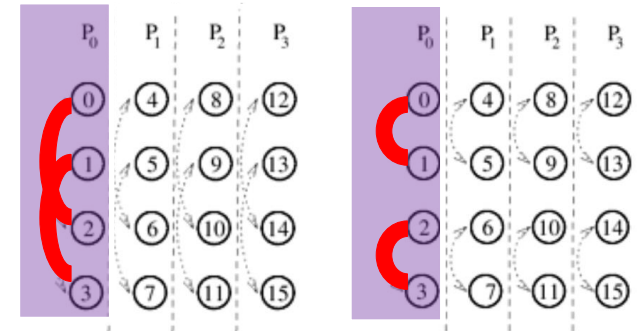
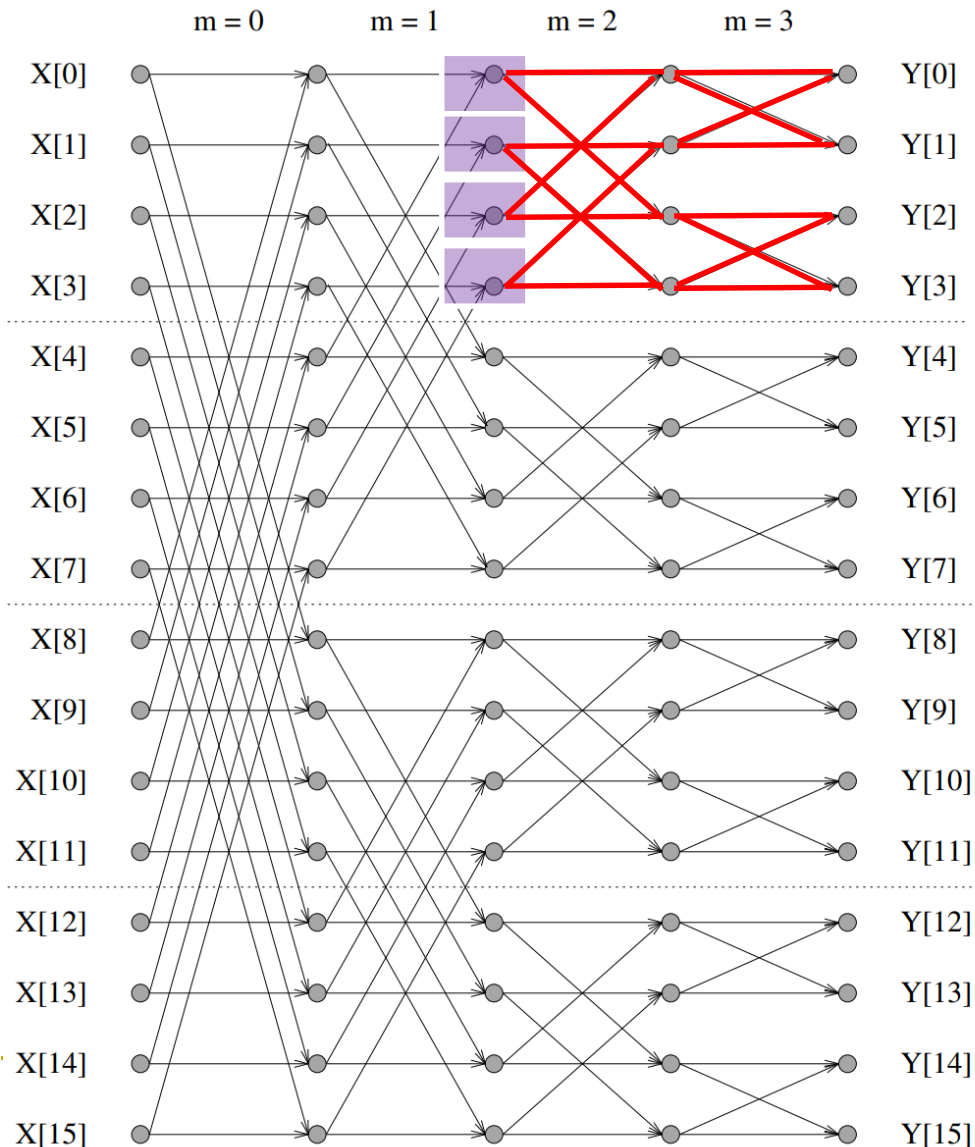


P_0	P_1	P_2	P_3
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Step 1: Local FFT for each column (on same processor)

Step 2: Transpose matrix (all-to-all personalized comm)

(2) Transpose Algorithm



Step 1: Local FFT for each column (on same processor)

Step 2: Transpose matrix (all-to-all personalized comm)

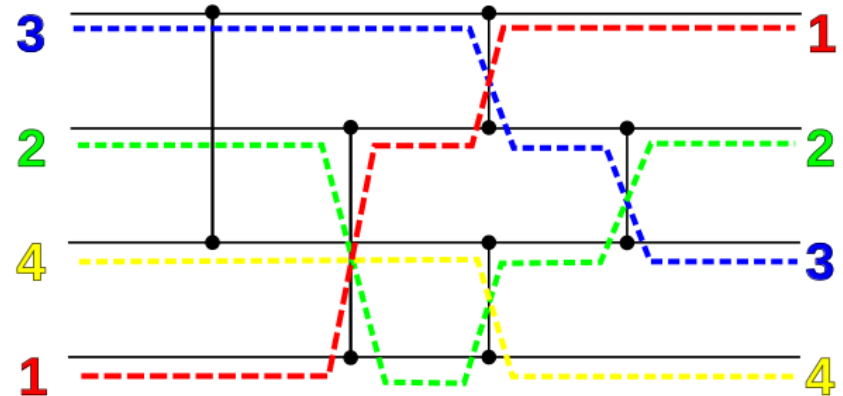
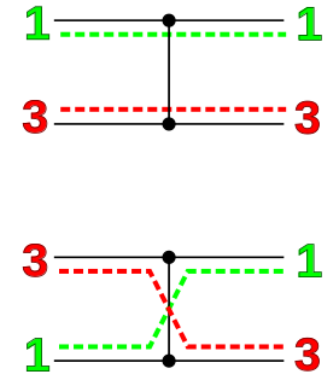
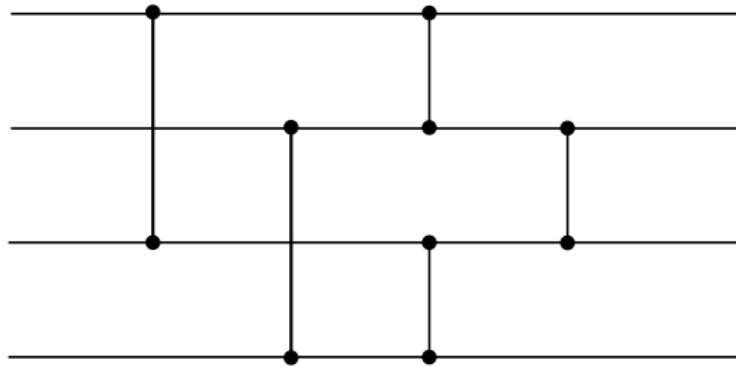
Step 3: Local FFT for each column (on same processor)

Sorting Algorithms

- A **sorting algorithm** arranges a list of n numbers or items in ascending or descending order --- a very important procedure in scientific computing.
- We have seen previously **quicksort** and **mergesort** in shared-memory architectures.
- In this class, we discuss three other sorting algorithms and their parallel implementations:
 - 1) **Bitonic Sort;**
 - 2) **Brick Sort;**
 - 3) **Shear Sort.**

Sorting Networks

- A **sorting network** sorts a fixed number of input values using a network of **comparators** connected by **wires**.

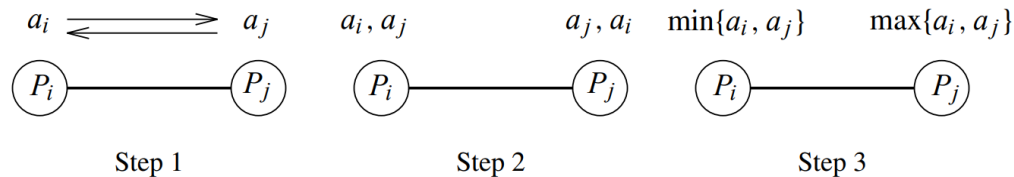


Why is this sorting network able to sort any four input values in ascending order?

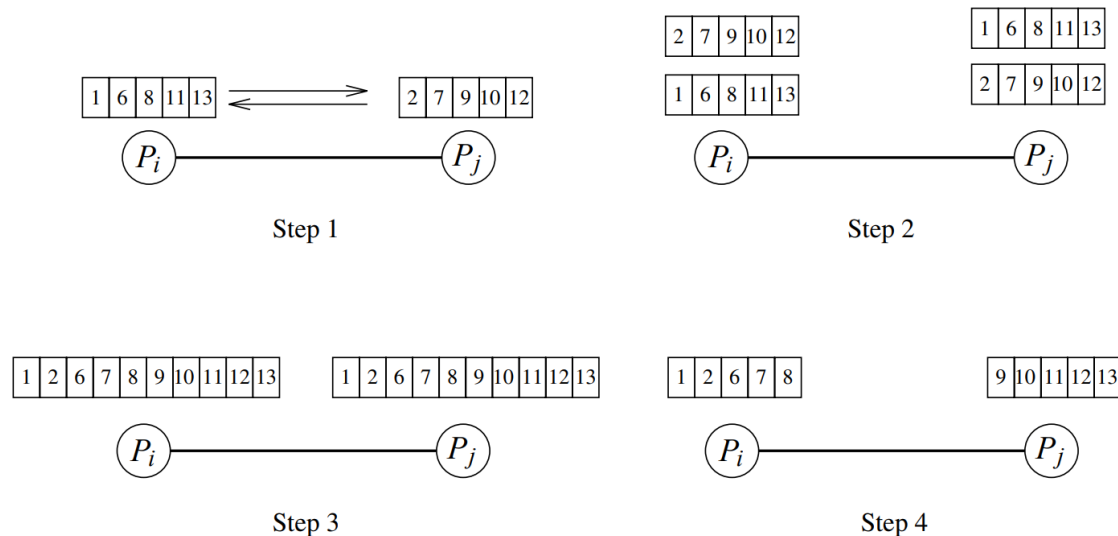
Figure source: https://en.wikipedia.org/wiki/Sorting_network

Basic Comparison Operations in Parallel

■ Compare-Exchange (one element per process)



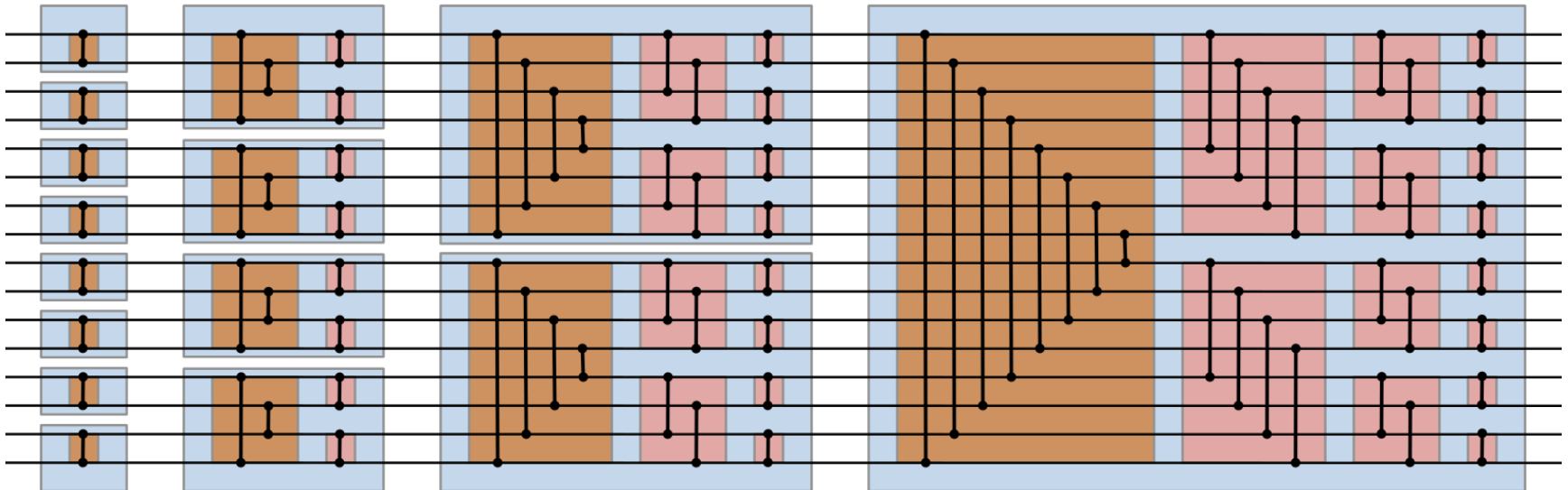
■ Compare-Split (multiple elements per process)



(1) Bitonic Sort

■ Bitonic Sorting Network

- Constructed recursively using **divide-and-conquer** for any number of inputs n that is a power of 2 (*read the reference materials on details of its construction*).



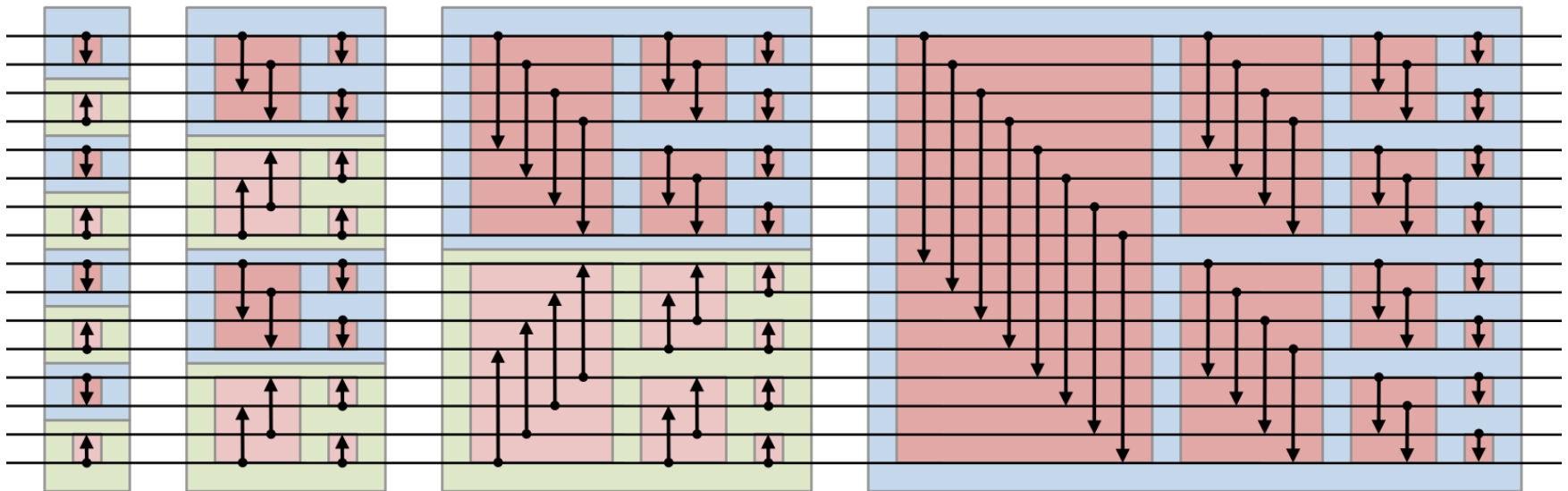
Standard form: *the larger number of a comparator always goes down.*

Figure source: https://en.wikipedia.org/wiki/Bitonic_sorter

(1) Bitonic Sort

■ Bitonic Sorting Network

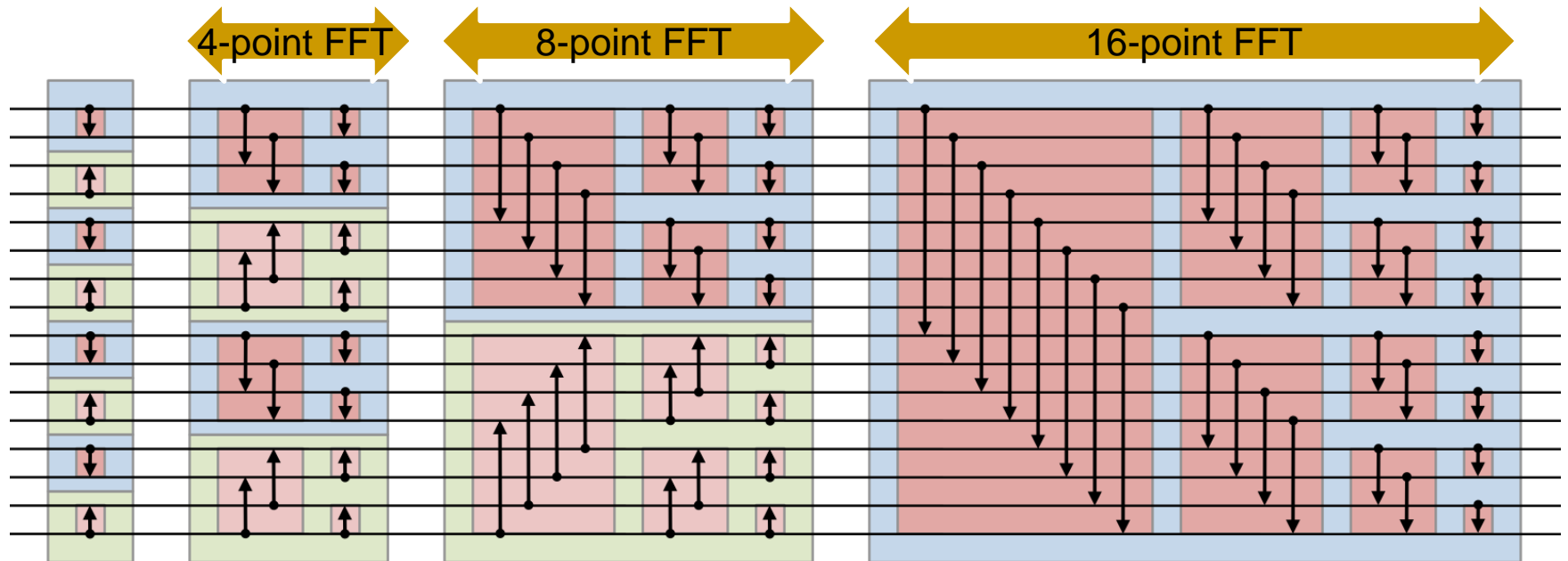
- Constructed recursively using **divide-and-conquer** for any number of inputs n that is a power of 2 (*read the reference materials on details of its construction*).



Non-standard form: the larger number of a comparator goes to point of arrow.

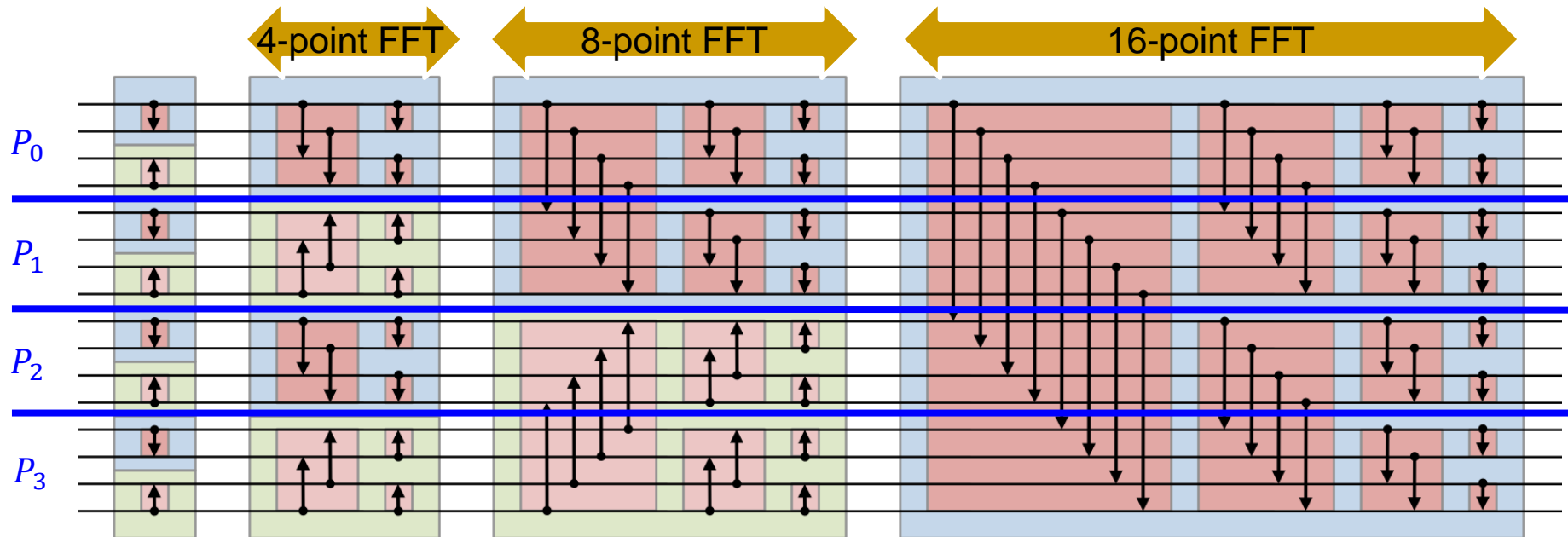
(1) Bitonic Sort in Parallel

Observe that a bitonic sorting network (non-standard) has multiple stages of computations with **FFT-like structures**.



(1) Bitonic Sort in Parallel

Observe that a bitonic sorting network (non-standard) has multiple stages of computations with **FFT-like structures**.



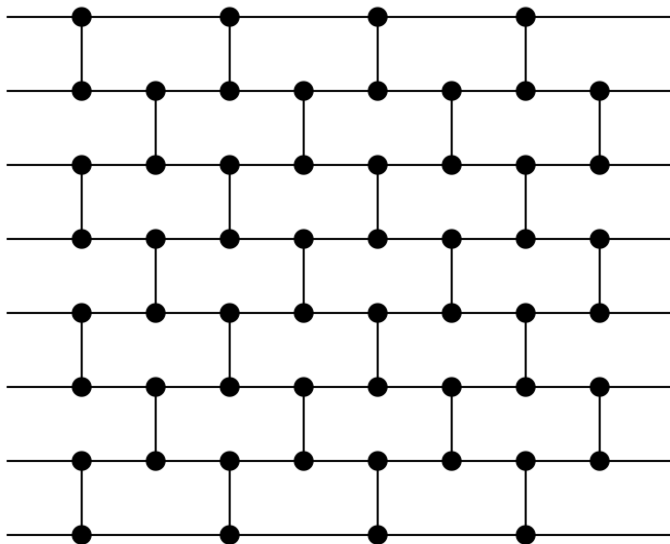
Parallel algorithms similar to Binary-Exchange for FFT can be implemented for Bitonic Sort.

- *1D Block distribution* of data, and comparisons across two processors can be done via *compare-exchange* operations.

(2) Brick Sort

■ Odd-Even Sorting Network

- Constructed by n levels of comparators connected in a “brick-like” pattern, with alternating odd and even sorting steps (it’s also called *Odd-Even Transposition Sort*).



Unsorted array: 2, 1, 4, 9, 5, 3, 6, 10

Step 1(odd): 2 1 4 9 5 3 6 10

Step 2(even): 1 2 4 9 3 5 6 10

Step 3(odd): 1 2 4 3 9 5 6 10

Step 4(even): 1 2 3 4 5 9 6 10

Step 5(odd): 1 2 3 4 5 6 9 10

Step 6(even): 1 2 3 4 5 6 9 10

Step 7(odd): 1 2 3 4 5 6 9 10

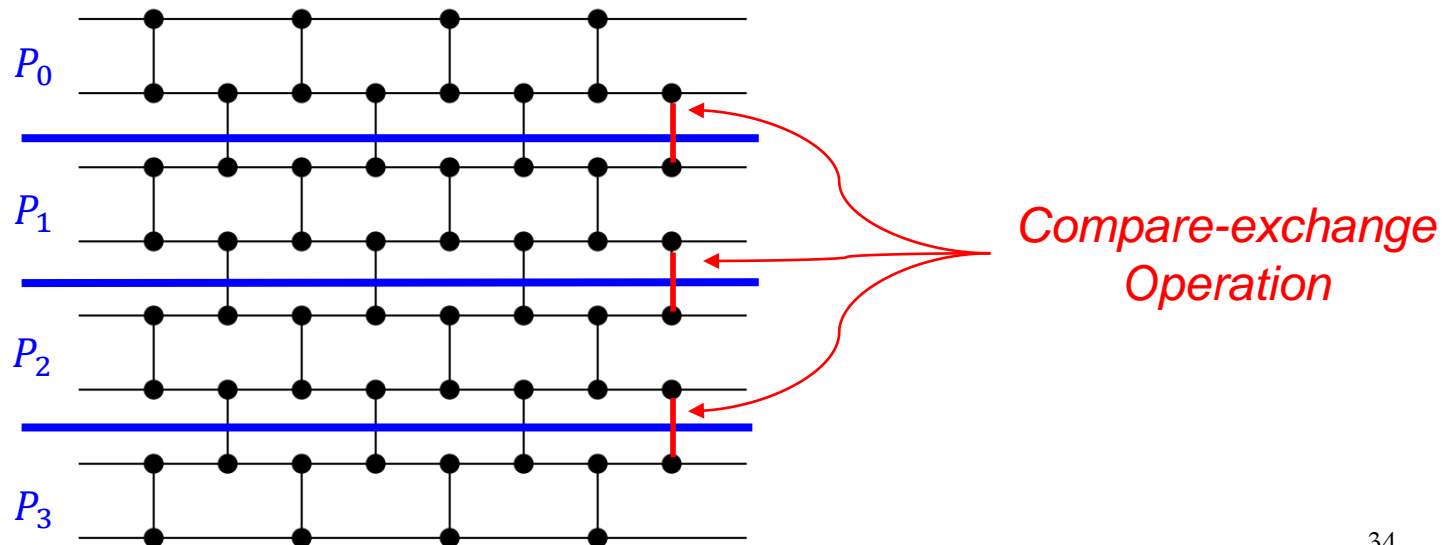
Step 8(even): 1 2 3 4 5 6 9 10

Sorted array: 1, 2, 3, 4, 5, 6, 9, 10

(2) Brick Sort in Parallel

■ Parallel Implementation 1:

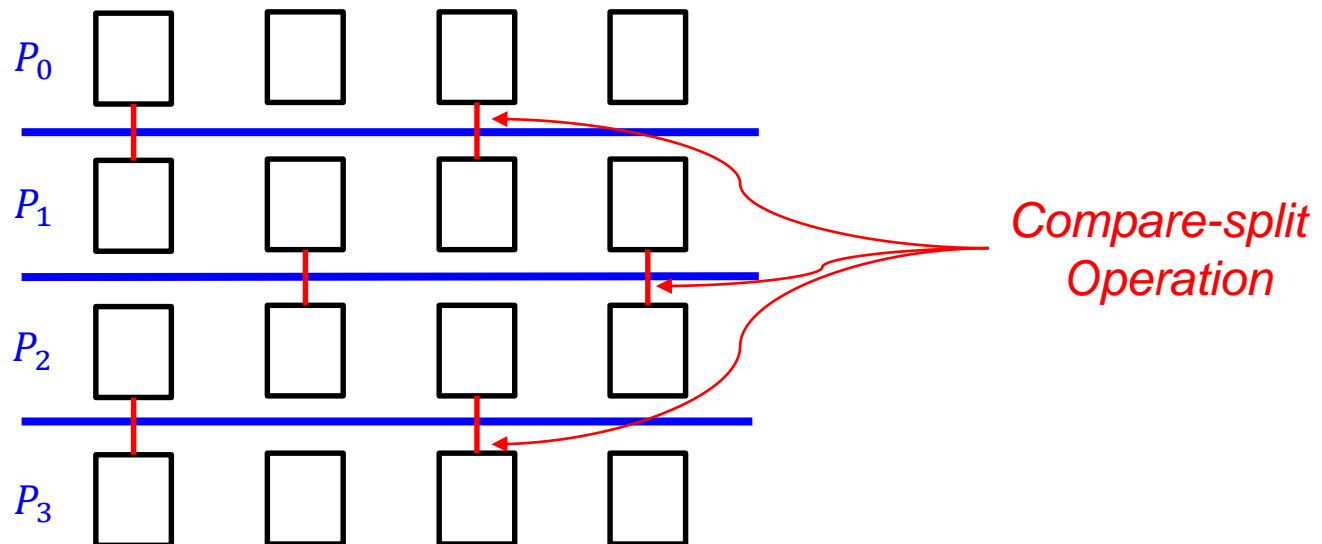
- ❑ 1D Block distribution;
- ❑ Follow exactly the same pattern of an n -step odd-even sorting network;
- ❑ Only boundary elements of a processor need communication (done via *compare-exchange*).



(2) Brick Sort in Parallel

■ Parallel Implementation 2:

- 1D Block distribution;
- Follow the same pattern again, but treating blocks of elements together $\rightarrow P$ steps instead of n ;
- Entire block of elements of a processor need to be communicated and compared (done via *compare-split*).



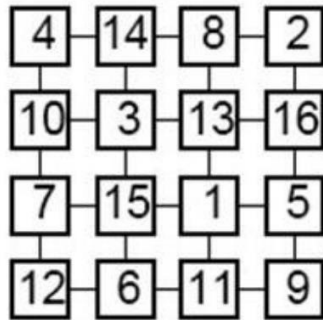
(3) Shear Sort

- Algorithm works in 2D:
 - A list of n numbers initially arranged in a $\sqrt{n} \times \sqrt{n}$ matrix;
 - For step $k = 1, 2, \dots, \lg n + 1$:
 - If k is odd, sort odd rows in *ascending* order and sort even rows in *descending* order;
 - If k is even, sort all columns in *ascending* order.
 - The final list will be sorted in **2D snake-like** order.

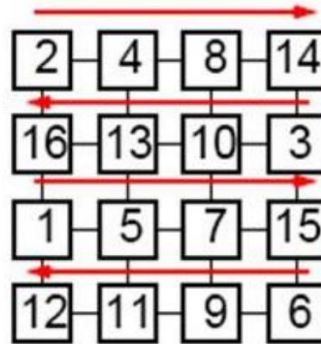
4	14	8	2
10	3	13	16
7	15	1	5
12	6	11	9

Original placement
of numbers

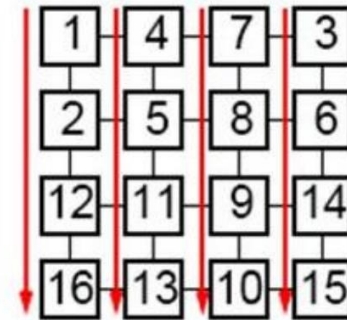
Shear Sort



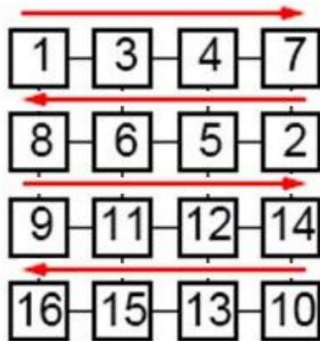
(a) Original placement of numbers



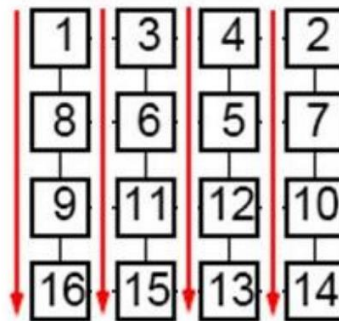
(b) Phase 1 — Row sort



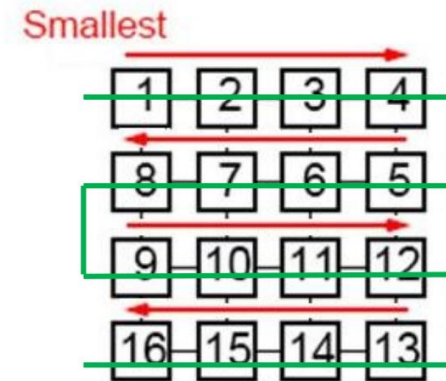
(c) Phase 2 — Column sort



(d) Phase 3 — Row sort



(e) Phase 4 — Column sort

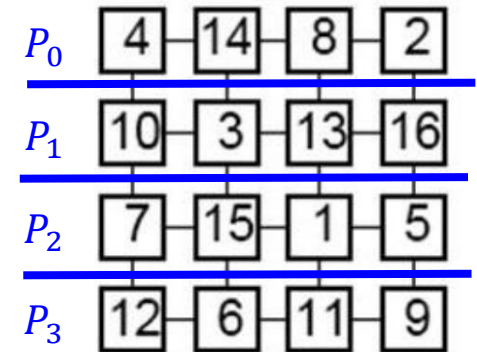


(f) Final phase — Row sort

Shear Sort in Parallel

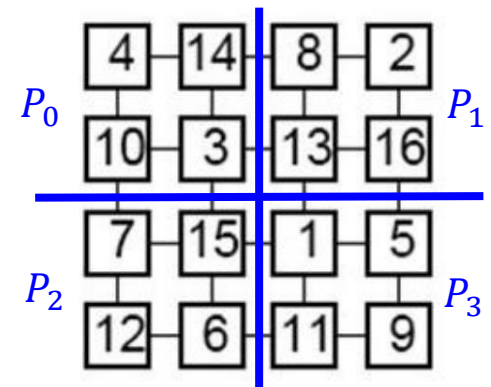
■ 1D Block Implementation:

- Row-sorting can be done *locally* on each processor;
- Column-sorting can be done across processors with *Brick Sort* for each column.



■ 2D Block Implementation:

- Both row-sorting and column-sorting can be done with *Brick Sort* across multiple processor groups (rows and columns).



References

- A. Grama, A. Gupta, G. Karypis, V. Kumar. Introduction to Parallel Computing (2nd Edition). *Addison-Wesley Professional*.
- H. Casanova, A. Legrand, Y. Robert. Parallel Algorithms. *Chapman & Hall/CRC*.
- B. Barney. Introduction to Parallel Computing Tutorial, Lawrence Livermore National Laboratory.
https://computing.llnl.gov/tutorials/parallel_comp/