Matrix-Multiplication Algorithms on Distributed-Memory Architectures

SC3260/5260 High-Performance Computing

Hongyang Sun
(hongyang.sun@vanderbilt.edu)

Vanderbilt University

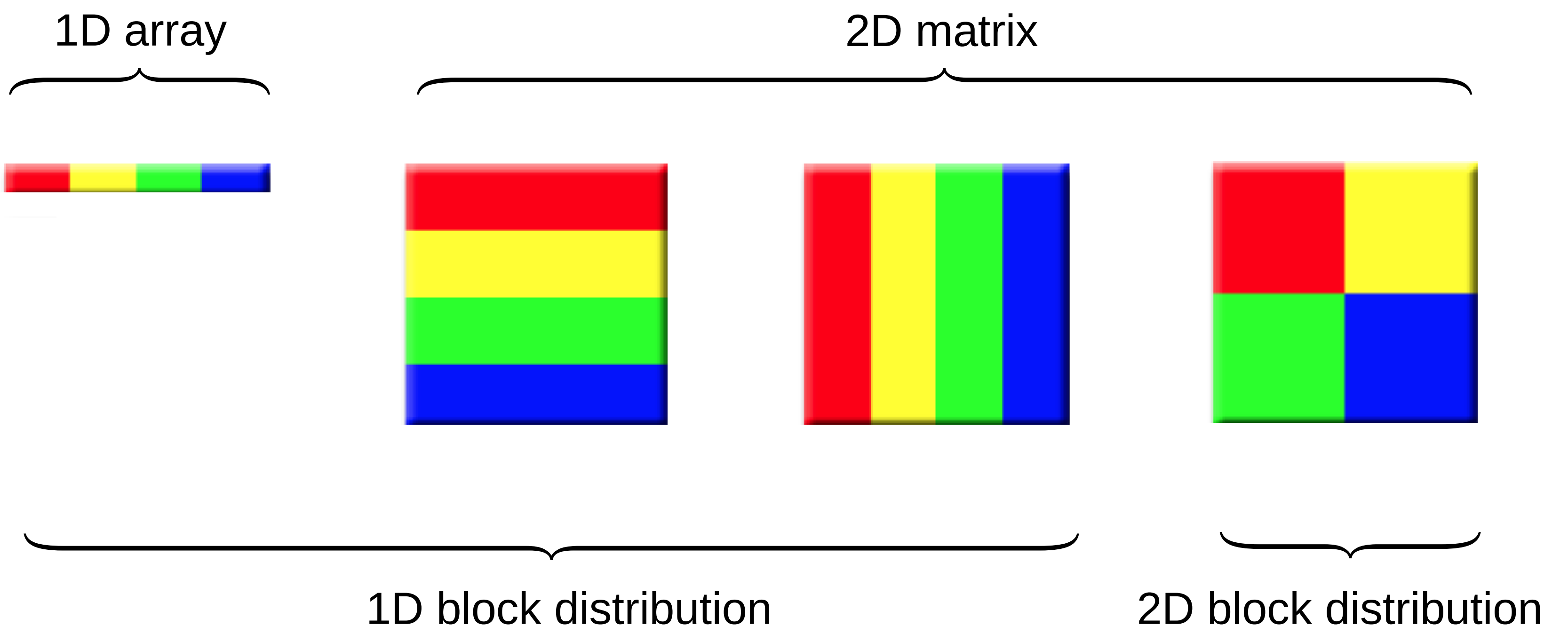
Spring 2020

Data Distribution

- With distributed memory, we assume the initial data is distributed among different processors, and communicated via message passing.
- There are different ways to distribute regular data (e.g., 1D array, dense matrix):
  - Block distribution;
  - Cyclic distribution;
  - Block-cyclic distribution.

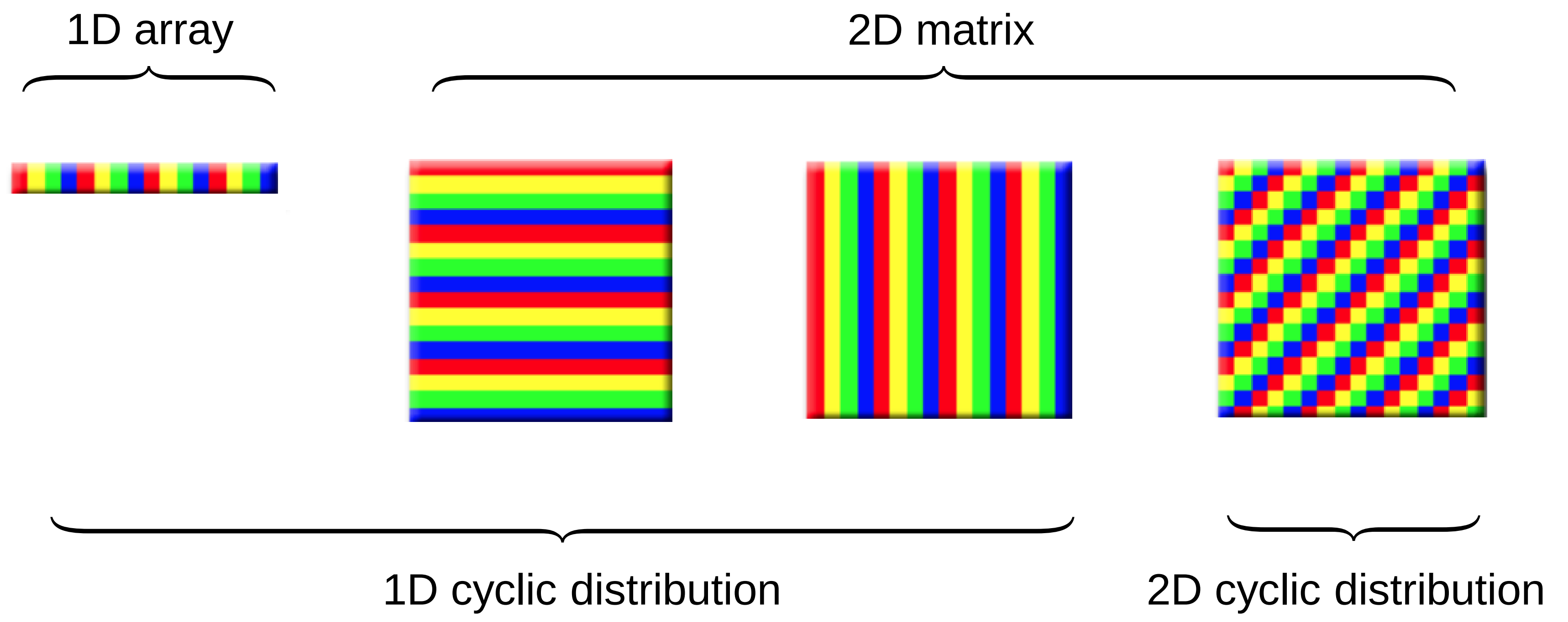
In this class, we will study distributed-memory algorithms that use block distribution, but block-cyclic distribution is more often used in practice.

Block Distribution



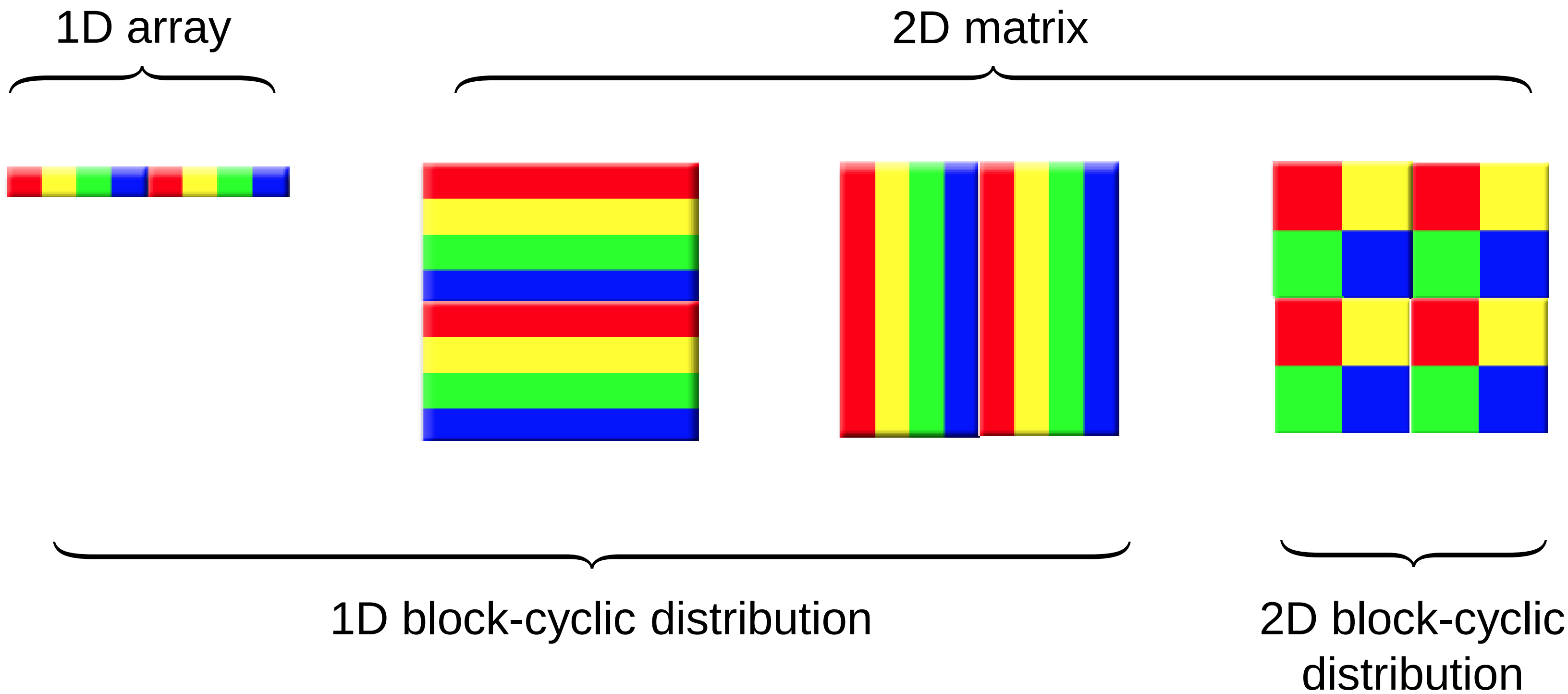
- Block distribution conserves locality of data for each process, and a higher-dimensional (e.g., 2D) distribution allows to use more processes.

Cyclic Distribution



- Cyclic distribution achieves better load balancing for certain computations (e.g., LU factorization), but has poor data locality for the processes.

Block-Cyclic Distribution



- Block-cyclic distribution strikes a balance between data locality and load balancing. It is more often used in practice by numerical libraries.

Matrix Multiplication

- We will discuss the following algorithms for multiplying two $n \times n$ matrices ($C = A \times B$) on P processors using 2D block distribution:
  1) Inner-product algorithm;
  2) Outer-product algorithm;
  3) Cannon's algorithm;
  4) Fox's algorithm;
  5) Snyder's algorithm.

Homework 3 *provides the inner-product algorithm and asks you to implement another algorithm and compare their performance on ACCRE.*

2D Block Distribution

- We assume:
  - All P processors are logically arranged in a 2D mesh of $\sqrt{P} \times \sqrt{P}$ processors, and n is divisible by $\sqrt{P}$.
  - Processor P_{ij} initially holds matrix blocks A_{ij} and B_{ij}, and is responsible for computing block C_{ij}. All blocks are of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$.
  - The figure on the right illustrates the initial data distribution for $P = 9$ processors.

P_{00} $(A_{00}, B_{00}) \rightarrow C_{00}$	P_{01} $(A_{01}, B_{01}) \rightarrow C_{01}$	P_{02} $(A_{02}, B_{02}) \rightarrow C_{02}$
P_{10} $(A_{10}, B_{10}) \rightarrow C_{10}$	P_{11} $(A_{11}, B_{11}) \rightarrow C_{11}$	P_{12} $(A_{12}, B_{12}) \rightarrow C_{12}$
P_{20} $(A_{20}, B_{20}) \rightarrow C_{20}$	P_{21} $(A_{21}, B_{21}) \rightarrow C_{21}$	P_{22} $(A_{22}, B_{22}) \rightarrow C_{22}$

(1) Inner-Product Algorithm

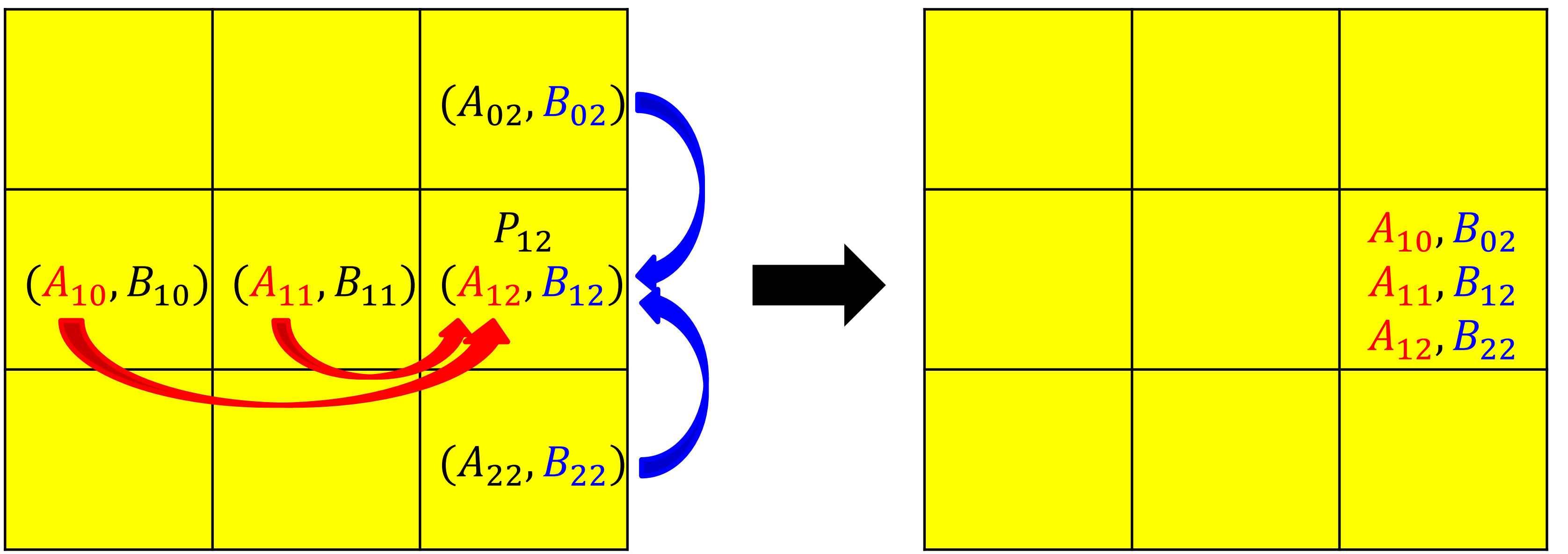
```
Inner-Product(A, B, C, P) {
   q = √P;
   for (i = 0; i < q; i++)
      for (j = 0; j < q; j++)
         for (k = 0; k < q; k++)
            // blocked matrix product
            C_ij += A_ik * B_kj;
}
```

- The above pseudocode shows the blocked version of the classical inner-product algorithm.
- **<u>Idea:</u>** *processor* P_{ij} *computes block* C_{ij} *locally by first collecting blocks* A_{ik} *and* B_{kj} *for all* $k = 0, 1, \ldots, \sqrt{P} - 1$.

(1) Inner-Product Algorithm

- The idea is illustrated below for processor P_{12}, which needs to computes:

$$C_{12} = A_{10} * B_{02} + A_{11} * B_{12} + A_{12} * B_{22}$$



(1) Inner-Product Algorithm

- Three steps of the algorithm:
  1) All-to-all broadcast (MPI_Allgather) of matrix A's blocks along each row;
  2) All-to-all broadcast (MPI_Allgather) of matrix B's blocks along each column;
  3) Each processor P_{ij} locally computes matrix block
     $C_{ij} = A_{i0} * B_{0j} + A_{i1} * B_{1j} + \cdots + A_{i\sqrt{P}} * B_{\sqrt{P}j}$
- <u>**Drawbacks:**</u> *each processor needs to hold $\sqrt{P}$ blocks of both matrices A and B, which may not fit in the local memory of the processor for very large matrix size n.*

(2) Outer-Product Algorithm

```
Outer-Product(A, B, C, P) {
   q = √P;
   for (k = 0; k < q; k++)
      for (i = 0; i < q; i++)
         for (j = 0; j < q; j++)
            // blocked matrix product
            C_ij += A_ik * B_kj;
}
```

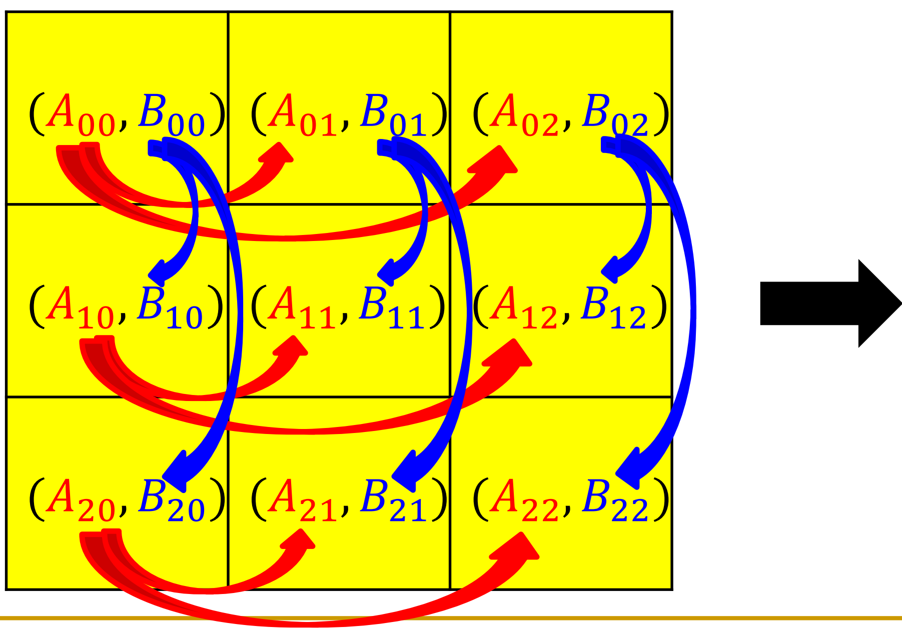
*The three "**for-loops**" can be arranged in any order without affecting correctness of result.*

- The above pseudocode shows the blocked version of the outer-product algorithm.
- **Idea:** *algorithm works in $\sqrt{P}$ steps; at the k-th step ($k = 0, 1, \ldots, \sqrt{P} - 1$), processor P_{ij} computes the k-th partial C_{ij} by collecting blocks A_{ik} and B_{kj}.*

(2) Outer-Product Algorithm

- The idea is illustrated below for step $k = 0$, where each processor P_{ij} needs to compute:

$$C_{ij} \mathrel{+}= A_{i0} * B_{0j}$$



A_{00}, B_{00} (A_{00}, B_{00})	A_{00}, B_{01} (A_{01}, B_{01})	A_{00}, B_{02} (A_{02}, B_{02})
$A_{10}\ B_{00}$ (A_{10}, B_{10})	A_{10}, B_{01} (A_{11}, B_{11})	A_{10}, B_{02} (A_{12}, B_{12})
A_{20}, B_{00} (A_{20}, B_{20})	A_{20}, B_{01} (A_{21}, B_{21})	A_{20}, B_{02} (A_{22}, B_{22})

(2) Outer-Product Algorithm

- Repeat step $k = 0, 1, \ldots, \sqrt{P} - 1$:
  1) One-to-all broadcast (MPI_Bcast) of matrix A's blocks in column k along each row;
  2) One-to-all broadcast (MPI_Bcast) of matrix B's blocks in row k along each column;
  3) Each processor P_{ij} locally updates matrix block
$$C_{ij} \mathrel{+}= A_{ik} * B_{kj}$$
- *Compared to the inner-product algorithm, the outer-product algorithm has much reduced memory requirement; each processor needs only hold 2 blocks of matrices A and B.*

(3) Cannon's Algorithm

- **Idea:** *algorithm works in $\sqrt{P}$ steps:*
  - *At step $k = 0$, processor P_{ij} updates block*

    $$C_{ij} = A_{i,(i+j) mod \sqrt{P}} * B_{(i+j) mod \sqrt{P}, j}$$

    through an initial alignment of matrices A and B's blocks via circular shift.
  - *At step $k = 1, 2, \dots, \sqrt{P} - 1$, processor P_{ij} updates block*

    $$C_{ij} = A_{i,(i+j+k) mod \sqrt{P}} * B_{(i+j+k) mod \sqrt{P}, j}$$

    by circular shifting matrices A and B's blocks by one position and computing the product.

(3) Cannon's Algorithm

- For step $k = 0$:
  1) Circular shift i-th row of matrix A's blocks i positions to the left (i.e., all diagonal blocks of A to first column);
  2) Circular shift i-th column of matrix B's blocks i positions up (i.e., all diagonal blocks of B to first row);
  3) Each processor P_{ij} locally updates matrix block C_{ij}
- For step $k = 1, 2, \ldots, \sqrt{P} - 1$:
  1) Circular shift each row of A's blocks one position left;
  2) Circular shift each column of B's blocks one position up;
  3) Each processor P_{ij} locally updates matrix block C_{ij}

Next slide illustrates all steps for $P = 9$ processors.

(3) Cannon's Algorithm

Initial data

(A_{00}, B_{00})	(A_{01}, B_{01})	(A_{02}, B_{02})
(A_{10}, B_{10})	(A_{11}, B_{11})	(A_{12}, B_{12})
(A_{20}, B_{20})	(A_{21}, B_{21})	(A_{22}, B_{22})

Step 0

(A_{00}, B_{00})	(A_{01}, B_{11})	(A_{02}, B_{22})
(A_{11}, B_{10})	(A_{12}, B_{21})	(A_{10}, B_{02})
(A_{22}, B_{20})	(A_{20}, B_{01})	(A_{21}, B_{12})

Step 1

(A_{01}, B_{10})	(A_{02}, B_{21})	(A_{00}, B_{02})
(A_{12}, B_{20})	(A_{10}, B_{01})	(A_{11}, B_{12})
(A_{20}, B_{00})	(A_{21}, B_{11})	(A_{22}, B_{22})

Step 2

(A_{02}, B_{20})	(A_{00}, B_{01})	(A_{01}, B_{12})
(A_{10}, B_{00})	(A_{11}, B_{11})	(A_{12}, B_{22})
(A_{21}, B_{10})	(A_{22}, B_{21})	(A_{20}, B_{02})

(3) Cannon's Algorithm

- Each processor needs only hold one block of matrix A and one block of matrix B at all times.
- Also note the order in which the partial results of each block C_{ij} is computed over the steps. For example,
  - $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20}$
  - $C_{11} = A_{12} * B_{21} + A_{10} * B_{01} + A_{11} * B_{11}$
  - $C_{22} = A_{21} * B_{12} + A_{22} * B_{22} + A_{20} * B_{02}$

(4) Fox's Algorithm

- **Idea:** *algorithm works in $\sqrt{P}$ steps:*
  - *At step $k = 0$, processor P_{ij} updates block*

    $$C_{ij} = A_{i,i} * B_{i,j}$$

    with the diagonal block of matrix A (i.e., $A_{i,i}$) being broadcasted in each row i.
  - *At step $k = 1, 2, \dots, \sqrt{P} - 1$, processor P_{ij} updates block*

    $$C_{ij} = A_{i,(i+k) mod \sqrt{P}} * B_{(i+k) mod \sqrt{P},j}$$

    with block $A_{i,(i+k) mod \sqrt{P}}$ broadcasted in each row i and circular shifting B by one position.

(4) Fox's Algorithm

- For step $k = 0$:
  1) One-to-all broadcast (MPI_Bcast) of block $A_{i,i}$ within each row i;
  2) Each processor P_{ij} locally updates matrix block C_{ij}
- For step $k = 1, 2, \dots, \sqrt{P} - 1$:
  1) One-to-all broadcast (MPI_Bcast) of block $A_{i,(i+k) mod \sqrt{P}}$ within each row i;
  2) Circular shift each column of B's blocks one position up;
  3) Each processor P_{ij} locally updates matrix block C_{ij}

Next slide illustrates all steps for $P = 9$ processors.

(4) Fox's Algorithm

Initial data

(A_{00}, B_{00})	(A_{01}, B_{01})	(A_{02}, B_{02})
(A_{10}, B_{10})	(A_{11}, B_{11})	(A_{12}, B_{12})
(A_{20}, B_{20})	(A_{21}, B_{21})	(A_{22}, B_{22})

Step 0

A_{00} (A_{00}, B_{00})	A_{00} (A_{01}, B_{01})	A_{00} (A_{02}, B_{02})
A_{11} (A_{10}, B_{10})	A_{11} (A_{11}, B_{11})	A_{11} (A_{12}, B_{12})
A_{22} (A_{20}, B_{20})	A_{22} (A_{21}, B_{21})	A_{22} (A_{22}, B_{22})

Step 1

A_{01} (A_{00}, B_{10})	A_{01} (A_{01}, B_{11})	A_{01} (A_{02}, B_{12})
A_{12} (A_{10}, B_{20})	A_{12} (A_{11}, B_{21})	A_{12} (A_{12}, B_{22})
A_{20} (A_{20}, B_{00})	A_{20} (A_{21}, B_{01})	A_{20} (A_{22}, B_{02})

Step 2

A_{02} (A_{00}, B_{20})	A_{02} (A_{01}, B_{21})	A_{02} (A_{02}, B_{22})
A_{10} (A_{10}, B_{00})	A_{10} (A_{11}, B_{01})	A_{10} (A_{12}, B_{02})
A_{21} (A_{20}, B_{10})	A_{21} (A_{21}, B_{11})	A_{21} (A_{22}, B_{12})

(4) Fox's Algorithm

- Each processor needs to hold two blocks of matrix A and one block of matrix B at all times.
- Also note the order in which the partial results of each block C_{ij} is computed over the steps. For example,
  - $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20}$
  - $C_{11} = A_{11} * B_{11} + A_{12} * B_{21} + A_{10} * B_{01}$
  - $C_{22} = A_{22} * B_{22} + A_{20} * B_{02} + A_{21} * B_{12}$

(5) Snyder's Algorithm

- **Idea:** *first transpose matrix B's blocks, and then compute each diagonal blocks of matrix C at a time by shifting matrix B's blocks by one position and performing a row-wide reduction.*
- Note that the transpose is with regard to the blocks of the whole matrix B, not for the values within each block.
- In this algorithm, each processor also needs only hold one block of matrix A and one block of matrix B at all times.

(5) Snyder's Algorithm

- Transpose matrix B's blocks;
- For step $k = 0, 1, \ldots, \sqrt{P} - 1$:
  1) Each processor P_{ij} computes the partial matrix block $C_{i,(i+k) mod \sqrt{P}}$
  2) All-to-one reduction (MPI_Reduce) of partial matrix blocks $C_{i,(i+k) mod \sqrt{P}}$ within each row i to processor $P_{i,(i+k) mod \sqrt{P}}$ to get the final $C_{i,(i+k) mod \sqrt{P}}$
  3) If $k < \sqrt{P} - 1$, then circular shift each block of B one position up.

Next slide illustrates all steps for $P = 9$ processors.

(5) Snyder's Algorithm

Initial data

(A_{00}, B_{00})	(A_{01}, B_{01})	(A_{02}, B_{02})
(A_{10}, B_{10})	(A_{11}, B_{11})	(A_{12}, B_{12})
(A_{20}, B_{20})	(A_{21}, B_{21})	(A_{22}, B_{22})

Step 0

$(A_{00}, B_{00}) \rightarrow C_{00}$	$(A_{01}, B_{10}) \rightarrow C_{00}$	$(A_{02}, B_{20}) \rightarrow C_{00}$
$(A_{10}, B_{01}) \rightarrow C_{11}$	$(A_{11}, B_{11}) \rightarrow C_{11}$	$(A_{12}, B_{21}) \rightarrow C_{11}$
$(A_{20}, B_{02}) \rightarrow C_{22}$	$(A_{21}, B_{12}) \rightarrow C_{22}$	$(A_{22}, B_{22}) \rightarrow C_{22}$

Step 1

$(A_{00}, B_{01}) \rightarrow C_{01}$	$(A_{01}, B_{11}) \rightarrow C_{01}$	$(A_{02}, B_{21}) \rightarrow C_{01}$
$(A_{10}, B_{02}) \rightarrow C_{12}$	$(A_{11}, B_{12}) \rightarrow C_{12}$	$(A_{12}, B_{22}) \rightarrow C_{12}$
$(A_{20}, B_{00}) \rightarrow C_{20}$	$(A_{21}, B_{10}) \rightarrow C_{20}$	$(A_{22}, B_{20}) \rightarrow C_{20}$

Step 2

$(A_{00}, B_{02}) \rightarrow C_{02}$	$(A_{01}, B_{12}) \rightarrow C_{02}$	$(A_{02}, B_{22}) \rightarrow C_{02}$
$(A_{10}, B_{00}) \rightarrow C_{10}$	$(A_{11}, B_{10}) \rightarrow C_{10}$	$(A_{12}, B_{20}) \rightarrow C_{10}$
$(A_{20}, B_{01}) \rightarrow C_{21}$	$(A_{21}, B_{11}) \rightarrow C_{21}$	$(A_{22}, B_{21}) \rightarrow C_{21}$

Some Remarks

- Among the discussed algorithms
  - ***Cannon's algorithm*** *incurs the lowest & optimal communication overhead, but it has certain limitations (i.e., P must be perfect square, and n divisible by $\sqrt{P}$).*
  - *Practical libraries (such as ScaLAPACK http://www.netlib.org/scalapack/) implement an* ***outer-product*** *version using 2D block-cyclic data distribution.*

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/