Linear Algebra
on a distributed environment

Jing Liu
TDB & LMB, Uppsala University

Programming of Parallel Computers,Jan, 2015

UPPSALA UNIVERSITET

Dep. of Information Techology

Topics in LA

Scalar
Vector
Matrix

- Solve linear equations
- Matrices operations
  - Matrices addition/ multiplication / transformation
  - Eigenvalue/ Eigenvector
  - Transpose, projection …
- Vector space
- …

UPPSALA UNIVERSITET

Dep. of Information Techology

Loops

- LA operations are often basic building blocks in scientific applications
- Three basic types of loops
  - Perfectly parallel loops
  - Reduction loops
  - Recursive loops
  - Combination of different loops

Perfectly parallel loops

- Example $Z_m = \lambda X_m + Y_m$

```
for ( i = 0; i < m; i ++ ){
      Z[i] = λ * X[i] + Y[i];
}
```

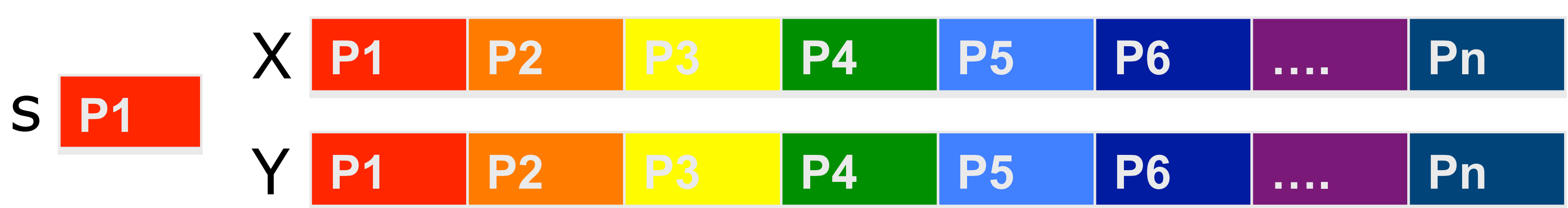
X,Y,Z

- MPI_Scatter and MPI_Gather

Reduction loops

- Limited parallelism
- Example: Dot production $s = X \bullet Y^T$

```
for ( i = 0; i < m; i ++ ) {
    s += X[i] + Y[i];
}
```



- [MPI_Reduce, MPI_Allreduce]

UPPSALA UNIVERSITET

Dep. of Information Techology

Recursive loops

- Each iteration depends on the previous one
- Hardly parallelize, “serial” loop
- Example

```
for ( i = 1; i < m; i ++) {
    X[ i ] = X[ i ] + X[ i - 1 ];
}
```

UPPSALA UNIVERSITET

Dep. of Information Techology

Nested loops

- Often the order of loops can be interchanged ➔ for maximal parallelism, choose the perfectly-parallel loops as outmost, and parallelize over it.
- Example: Matrix-Vector multiplication

```
for ( i = 0;  i < m; i++){
    for ( j = 0; j< m ; j++){
        Y[i] += A[i][j] * X[j];
    }
}
```

Nested loops – Alt 1

- Row-wise partition

- All processors have a copy of X, one piece of A and Y.

Nested loop – Alt. 2

- Block algorithm with 1D partition

$$
\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} * \begin{bmatrix} \mathbf{X_0} \\ \mathbf{X_1} \\ \mathbf{X_2} \\ \mathbf{X_3} \end{bmatrix}
$$

- Step 1: Compute Y[i] = A[i][i] * X[i] in process i, and then shift X[i] circular one step up.
- Step 2: Compute again, in which j=(i+1) mod p, shift X circular one step up.
- Repeat, in total (p-1) step

UPPSALA UNIVERSITET

Dep. of Information Techology

Nested loop – Alt. 2 cont.

- Non-blocking communication to shift X, before computation. MPI_Isend, MPI_Irecv, MPI_wait
- Which one is more efficient?
  - Alt. 2 is more memory efficient.
  - CPU efficient is all depends on the problem size, computer systems, implementations of MPI_functions, etc.

UPPSALA UNIVERSITET

Dep. of Information Techology

Nested loop – Alt. 3

- Block algorithm – 2D partition
- Processor block $\sqrt{p} * \sqrt{p}$,
- Step 1: Divide A_{mn} to $\sqrt{p} * \sqrt{p}$ blocks, X to $\sqrt{p}$ parts
- Step 2: Processor P_{ij} get block A_{ij} and X_j, and hold $Y_i^{(j)} = \mathbf{0}$
- Step 3: P_{ij} computes $Y_i^{(j)} = A_{ij} * X_j$
- Step 4: Accumulate Y_i in each row.

UPPSALA UNIVERSITET

Dep. of Information Techology

Nested loop – Alt. 3

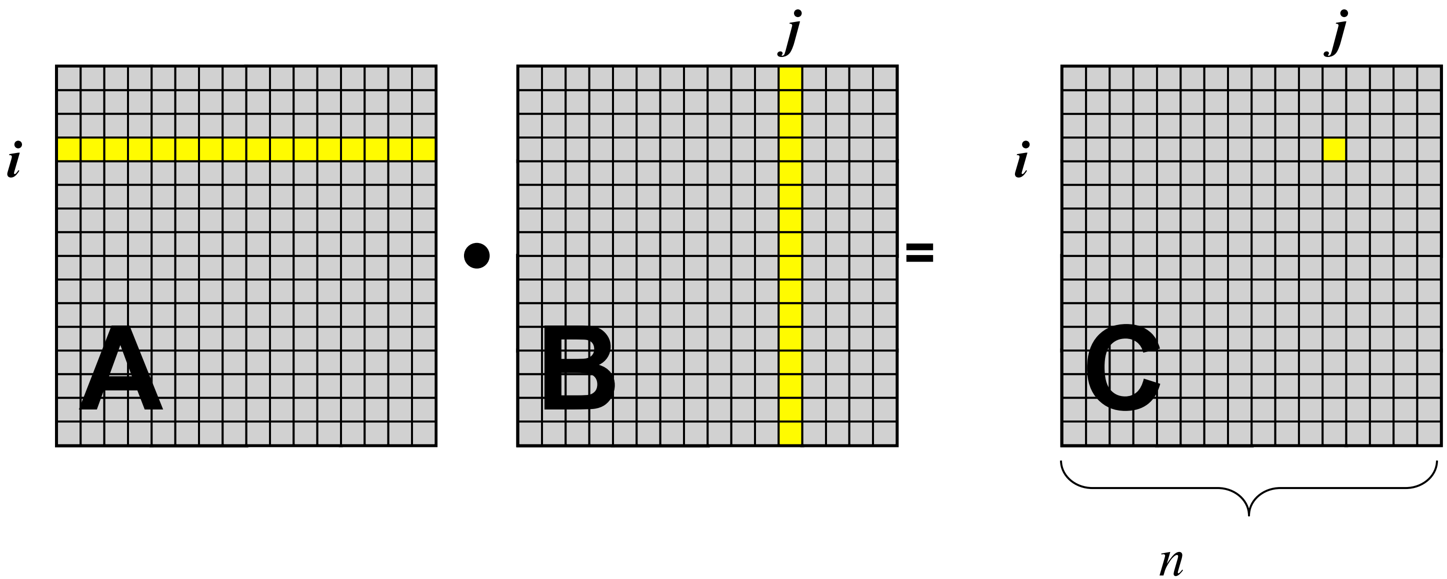
Y_0	$\xleftarrow{\Sigma_j}$	$Y_0^0 = A_{00} * X_0$	$Y_0^1 = A_{01} * X_1$	$Y_0^2 = A_{02} * X_2$	$Y_0^3 = A_{03} * X_3$
Y_1	$\xleftarrow{\Sigma_j}$	$Y_1^0 = A_{01} * X_0$	…	…	…
Y_2	$\xleftarrow{\Sigma_j}$	$Y_2^0 = A_{02} * X_0$	…	…	…
Y_3	$\xleftarrow{\Sigma_j}$	$Y_3^0 = A_{03} * X_0$	$Y_3^1 = A_{13} * X_1$	$Y_3^2 = A_{23} * X_2$	$Y_3^3 = A_{33} * X_3$

- Efficient for large matrices.
- Scalability? 2D > 1D. For many processors, 1D partition strips become so thin and communications increases faster.

Dep. of Information Techology

UPPSALA UNIVERSITET

Matrix-Matrix Multi.



$$C(i,j) = \Sigma_k A(i,k)\, B(k,j)$$

UPPSALA UNIVERSITET

Dep. of Information Techology

More nested Loops

- Example : Matrix-Matrix Multiplication
- i and j are perfectly parallel loops, k is reduction loop

```
for ( _ = 0;  _ < m; _++){
      for ( _ = 0; _< m ; _++){
           for (  _ = 0 ; _< m; _ ++){
                       C[i][j] += A[i][k] * B[k][j];
           }
      }
}
```

Matrix-Matrix Multi.

- 1D partitioning – choose j as the outmost loop → partition data column wise

$$
\left[\begin{array}{c|c|c|c} C_0 & C_1 & C_2 & C_3 \end{array}\right] = \left[\begin{array}{c|c|c|c} A_0 & A_1 & A_2 & A_3 \end{array}\right] * \left[\begin{array}{c|c|c|c} B_{00} & B_{01} & B_{02} & B_{03} \\ \hline B_{10} & \cdots & & \\ \hline B_{20} & & \cdots & \\ \hline B_{30} & & & B_{33} \end{array}\right]
$$

- ➔ $C_0 = A_0 * B_{00} + A_1 * B_{10} + A_2 * B_{20} + A_3 * B_{30}$
- …

UPPSALA UNIVERSITET

Dep. of Information Techology

C = A*B, 1D partition

- A is needed in every processor.
- Alt. 1 : Every processor has completed A,
  - ➔ Not scalable (memory?!)
- Alt. 2: Shift A around.
  - ➔ Similar idea to matrix-vector alt. 2.
  - ➔ For many processors, the stripes (block-columns) become thin and comm. overhead becomes large.

UPPSALA UNIVERSITET
Dep. of Information Techology

C = A*B, 2D partition

- Choose both i and j outmost.
- √p * √p blocks, each processor gets one block of each matrix.
- In processor P_{ij}, compute $C_{ij} = \sum_{k=0}^{\sqrt{p}-1} A_{ik} * B_{kj}$

  ➔ P_{ij} need all blocks A_{ik} in block row i, and B_{kj} in block column j

➔ Communications needed.

UPPSALA UNIVERSITET

Dep. of Information Techology

C = A*B, 2D partition, Alt. 1

- Simple and naïve method.
- Simply distribute A in each block row, and distribute of B in each block column, using MPI_functions

  ➔ limited scalability due to memory.

  ➔ Bad performance if data don't fit in catch

UPPSALA UNIVERSITET

Dep. of Information Techology

C = A*B, 2D partition, Alt. 2 Cannon's Algorithm (1969)

- Shift and compute. M*M mesh (√p * √p blocks processors, data).
- Phase 1: shift
  - Shift the i th block row of A i steps cyclically to the left.
  - Shift the j th block column of B j steps cyclically upwards

A_{00}	A_{01}	A_{02}	A_{03}
A_{11}	A_{12}	A_{13}	A_{10}
A_{22}	A_{23}	A_{20}	A_{21}
A_{33}	A_{30}	A_{31}	A_{32}

B_{00}	B_{11}	B_{22}	B_{33}
B_{10}	B_{21}	B_{32}	B_{03}
B_{20}	B_{31}	B_{02}	B_{13}
B_{30}	B_{01}	B_{12}	B_{23}

C = A*B, 2D partition, Alt. 2 Cannon's Algorithm Cont.

UPPSALA UNIVERSITET

Dep. of Information Techology

- Phase 2: Compute and shift
- For each iteration do:
  - Compute $C_{ij} = A_{ik} * B_{kj}$ in each processor P_{ij}, where k = (i+j+l) mod M, where l is the number of iterations (start from 0).
  - Shift A one step left, B one step upwards
- In total, M-1 steps. We can do shift with non-blocking communication, and compute while sending.
- Read more on-line Cannon's algorithm.

C = A*B, 2D partition, Alt. 3 Fox's Algorithm

- In total M-1 step.
- For each step k (k = 0,1,…, M-1)
  - Broadcast block n of A within each block row i (n = (i+k) mod M)
  - Multiply the broadcasted block with B-block in each processor ($C_{ij} += A_{in} * B_{nj}$)
  - Shift blocks of B, one step upwards.

C = A*B, 2D partition

- Both Cannon's and fox's algorithm is scalable.
- Which is more efficient?
  - Depends on problem size, computer system, efficiency of MPI, etc
- Read more about efficient on Fox and Cannon.

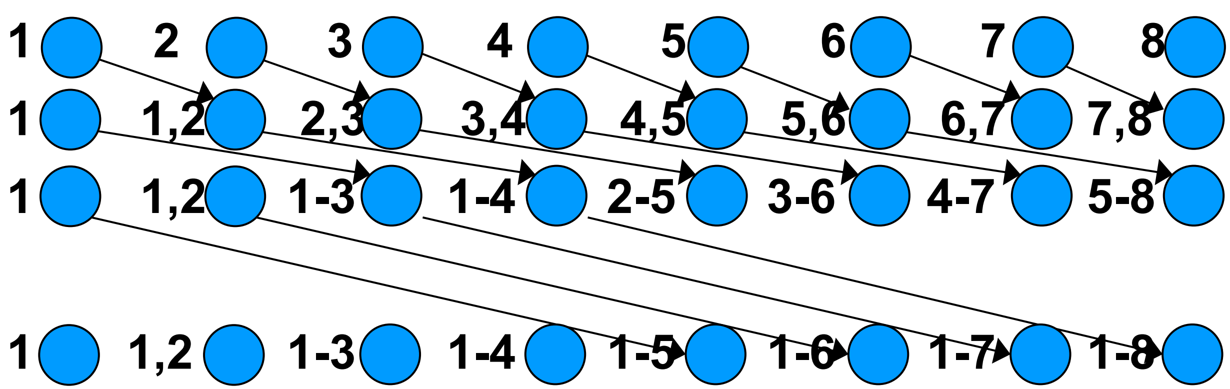
UPPSALA UNIVERSITET

Dep. of Information Techology

Advanced Topic: Recursive loop

- Example:

```
for ( i=1; i<n; i++){
    X[i] += X[i-1];
}
```



UPPSALA UNIVERSITET

Dep. of Information Techology

Assignment 1

- Dense matrix-matrix multiplication.
- Fortran/C/C++ and MPI.
- Two parameters:
  - The number of process
  - The size of matrices
- Randomly generate A and B
- Distribute data
- Implement Fox's algorithm
- Collect data and output.

UPPSALA UNIVERSITET

Dep. of Information Techology

Assignment 1, cont.

- Data generation (at rank 0): srand(), rand() / CALL RANDOM_SEED(), CALL RANDOM_NUMBER()
- Data distribution: use MPI_Type_vector, MPI_Cart_rank, MPI_Isend, MPI_Recv
- Data Collection: MPI_Probe, MPI_Cart_coords, MPI_Recv, MPI_wait

UPPSALA UNIVERSITET

Assignment 1, cont.

- C structure / C++ class is helpful to make a nicer code.
  - Name space works for large project.
- Good coding style makes your code more understandable and maintainable.
- Write comments in your code to help yourself and others.
- Demo code at https://github.com/JinLi971/MPI_DEMO

Dep. of Information Techology

UPPSALA UNIVERSITET

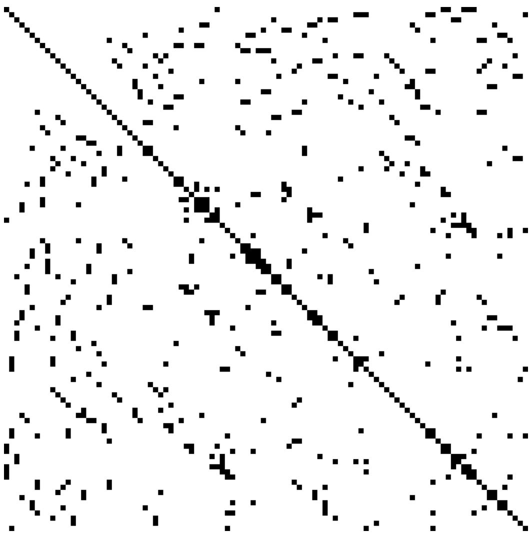
More Advanced Topic: BLAS

- CPUs:
  - Armadillo : Matlab style, C++ coding .
  - CBLAS :GNU supported.
  - Support: AMD -> ACML; IBM -> ESSL; Apple -> Accelerate framework; HP -> MLIB; SUN -> Sun Performance Library, Intel Math Kernel Library
- GPUs: NVIDIA -> CuBLAS

  OPENCL : third part support.

More Advanced Topic: Sparse Matrix

- A sparse matrix is a matrix populated primarily with zeros.
- Save sparse matrix:
  - Dictionary of keys
  - List of lists
  - Coordinate list
  - Yale format
  - Etc.



More Advanced Topic: Application using LA

- PageRank: imaging incredible large matrix
- Modern Digital imaging.
  - Video tracking: Xbox Kinect
- Genetics
- Cryptography
- Economic
- More …

More Advanced Topic: Application using LA

- Schedule & auto tuning
  - Test cases and pre-determined
  - Dynamically schedule
- Kernel and Convolution
  - Performs in parallel computers, edges of each blocks need to fix, according to the size of the kernel