COMP 605: Introduction to Parallel Computing
Topic: MPI: Matrix-Matrix Multiplication

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 03/01/17
Updated: 03/06/17

Table of Contents

Matrix-Matrix Multiplication

There are two types of matrix multiplication operations:

- Hadamard (element-wise) multiplication $C = A. * B$
- Matrix-Matrix Multiplication

Hadamard (element-wise) Multiplication

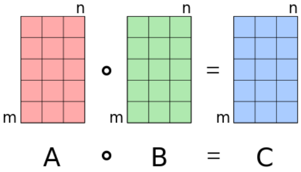
The Hadamard (or Schur) product is a binary operator that operates on 2 identically-shaped matrices and produces a third matrix of the same dimensions.

Definition: If $A = [a_{ij}]$ and $B = [b_{ij}]$ are $m \, x \, n$ matrices, then the Hadamard product of A and B is defined to be:

$$(A \circ B)_{ij} = (A)_{ij} \cdot (B)_{ij}$$

is an $m \, x \, n$ matrix $C = [c_{ij}]$ such that

$$c_{ij} = a_{ij} * b_{ij}$$



Notes: The Hadamard product is associative and distributive, and commutative; used in lossy compression algorithms such as JPEG Ref:

http://en.wikipedia.org/wiki/Hadamard_product_(matrices)

2D Matrix-Matrix Multiplication (Mat-Mat-Mult)

```
/* Serial_matrix_mult */
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++) {
      C[i][j] = 0.0;
      for (k = 0; k < n; k++)
         C[i][j] = C[i][j] + A[i][k]*B[k][j];
   printf(... )
}
```

Where:
A is an $[m \, x \, k]$ matrix
B is a $[k \, x \, n]$
C is a matrix with the dimensions $[m \, x \, n]$

2D Matrix-Matrix Multiplication (Mat-Mat-Mult)

Definition: Let A be an $[m \, x \, k]$ matrix, and B be a be an $[k \, x \, n]$, then C will be a matrix with the dimensions $[m \, x \, n]$.

Then $AB = \lfloor c_{ij} \rfloor$, and

$$c_{ij} = \sum_{t=1}^{k} a_{it} b_{tj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{k1} b_{kj}$$

$$= \begin{bmatrix} a_{00} \;\dots\; a_{0j} \;\dots\; a_{0,k-1} \\ \dots \\ \mathbf{a_{i0}} \;\dots\; a_{ij} \;\dots\; a_{i,k-1} \\ \dots \\ a_{m-1,0} \;\dots\; a_{m-1,j} \;\dots\; a_{m-1,k-1} \end{bmatrix} \bullet \begin{bmatrix} b_{00} \;\dots\; \mathbf{b_{0j}} \;\dots\; b_{0,n-1} \\ \dots \\ b_{i0} \;\dots\; \mathbf{b_{ij}} \;\dots\; b_{i,n-1} \\ \dots \\ b_{k-1,1} \;\dots\; \mathbf{b_{kj}} \;\dots\; b_{n-1,p-1} \end{bmatrix}$$

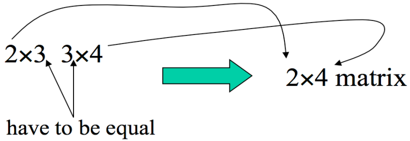
$$= \begin{bmatrix} c_{00} \;\dots\; c_{1j} \;\dots\; c_{1,n-1} \\ \dots \\ c_{i0} \;\dots\; \mathbf{c_{ij}} \;\dots\; c_{i,n-1} \\ \dots \\ c_{m-1,0} \;\dots\; c_{mj} \;\dots\; c_{m-1,n-1} \end{bmatrix}$$

$$c_{12} = a_{11} b_{12} + a_{12} b_{22} + a_{13} b_{32}$$

Matrix Inner Dimensions Must Match

To multiply two matrices, inner numbers must match:

Otherwise, not defined.



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \end{bmatrix}$$

2×3 3×4 2×4

Mat-Mat-Mult is associative $[(AB)C = A(BC)]$
Mat-Mat-Mult is not commutative $(AB \neq BA)$

Ref: `http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt`

Serial Matrix-Matrix Multiplication

Let A be a $m \times k$ matrix, and B be a $k \times n$ matrix,

$$AB = \left[c_{ij}\right]$$

$$c_{ij} = \sum_{t=1}^{k} a_{it} b_{tj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{ik}b_{kj}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \end{bmatrix}$$

$$a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} = c_{12}$$

Ref: `http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt`

Pacheco: serial_mat_mult.c

```
/* serial_mat_mult.c -- multiply two square matrices on a
 * single processor
 * Input:
 *     n: order of the matrices
 *     A,B: factor matrices
 * Output:
 *     C: product matrix
 * See Chap 7, pp. 111 & ff in PPMPI
 */
#include <stdio.h>
#define MAX_ORDER 10
typedef float MATRIX_T[MAX_ORDER][MAX_ORDER];
main() {
    int       n;
    MATRIX_T  A, B, C;

    void Read_matrix(char* prompt, MATRIX_T A, int n);
    void Serial_matrix_mult(MATRIX_T A, MATRIX_T B,
               MATRIX_T C, int n);
    void Print_matrix(char* title, MATRIX_T C, int n);

    printf("What's the order of the matrices?\n");
    scanf("%d", &n);

    Read_matrix("Enter A", A, n);
    Print_matrix("A = ", A, n);
    Read_matrix("Enter B", B, n);
    Print_matrix("B = ", B, n);
    Serial_matrix_mult(A, B, C, n);
    Print_matrix("Their product is", C, n);

}  /* main */
```

```
/*****************************************************************
/* MATRIX_T is a two-dimensional array of floats */
void Serial_matrix_mult(
        MATRIX_T   A   /* in  */,
        MATRIX_T   B   /* in  */,
        MATRIX_T   C   /* out */,
        int        n   /* in  */) {

    int i, j, k;

    void Print_matrix(char* title, MATRIX_T C, int n);

    Print_matrix("In Serial_matrix_mult A = ", A, n);
    Print_matrix("In Serial_matrix_mult B = ", B, n);

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
            C[i][j] = 0.0;
            for (k = 0; k < n; k++)
                C[i][j] = C[i][j] + A[i][k]*B[k][j];
            printf("i = %d, j = %d, c_ij = %f\n",
                        i, j, C[i][j]);
        }
}  /* Serial_matrix_mult */
```

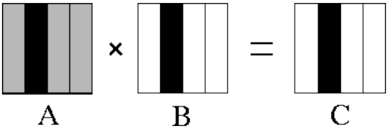
Parallel 2-D Matrix Multiplication Characteristics

- **Computationally independent**: each element computed in the result matrix C, c_{ij}, is, in principle, independent of all the other elements.
- **Data independence:** the number and type of operations to be carried out are independent of the data. Exception is sparse matrix multiplication: take advantage of the fact that most of the matrices elements to be multiplied are equal to zero.
- **Regularity of data organization and operations** carried out on data: data are organized in two-dimensional structures (the same matrices), and the operations basically consist of multiplication and addition.
- Parallel matrix multiplication follows SPMD (Single Program - Multiple Data) parallel computing model

ref: http://ftinetti.zxq.net/phdthesis/EngVersion-chap2.pdf

Foster 1-D matrix data decomposition.

- 1-D column wise decomposition
- Each task:
  - Utilizes subset of cols of A, B, C.
  - Responsible for calculating its C_{ij}
  - Requires full copy of A
  - Requires $\frac{N^2}{P}$ data from each of the other $(P-1)$ tasks.
- # Computations: $\mathcal{O}\left(N^3/P\right)$
- $T_{mat-mat-1D} = (P-1)\left(t_{st} + t_{wall}\frac{N^2}{P}\right)$

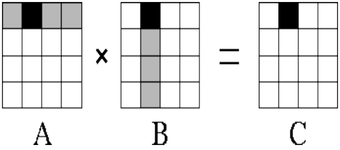


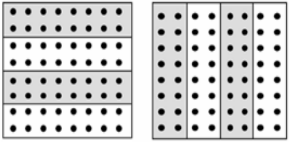
Not very efficient

REF: Foster1995, Ch 4.6

Block-striped 2D matrix data decomposition

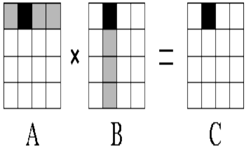
- Each processor is assigned a subset of:
  - matrix rows (row-wise or horizontal partitioning) OR
  - matrix columns (column-wise or vertical partitioning)
- To compute a row of matrix C each subtask must have
  - a row of the matrix A &
  - access to all columns of matrix B.
- # Computations $\mathcal{O}\left(\frac{N^2}{\sqrt{P}}\right)$





REFS: Foster1995 Ch4, Pacheco1997, Ch 7,
P. Anghelescu, http://www.ier-institute.org/2070-1918/lnit23/v23/065.pdf

Block-striped matrix data decomposition pseudocode




```
For each row of C
  For each column of C {
    C[row][column] = 0.0
    For each element of this row of A
      Add A[row][element]*B[element][column]
            to  C[row][column]
```

Parallel implementation costly: # Computations: $\mathcal{O}\left(N^3/P\right)$

```
For each column of B  {
  Allgather(column)
  Compute dot product of my row of A with column
}
```

REFS: Pacheco PPMPI [?], Ch 7

Block-striped matrix data decomposition - Alg 1

- #Iterations = #Subtasks.
- Pseudocode:

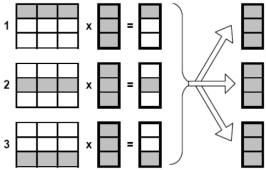
  For each *Iteration*

      Subtask has row $\hat{A}_i$, column $\hat{B}_j$

          Elements C_{ij} are computed.

      Subtask $\Leftarrow \hat{B}_{j+1}$

          C elements are calculated.
- Transmission of columns ensures that each task gets copy of all B columns.
- Performance:
  $T_p = \left(\frac{n^2}{p}\right) * (2n-1) * \tau_{op}$
- # Computations $\mathcal{O}\left(n^3/P\right)$



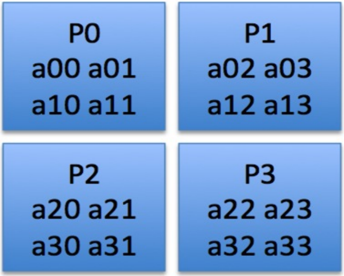
REFS: Foster Ch4, and Anghelescu

Block-striped matrix data decomposition - Alg 2)

- Distribute A and C, move cols of B across tasks
- Define $\#Iterations = \#Subtasks$
- Pseudocode:
  For each *Iteration*
    Subtask has row $\hat{A}_i$, and all rows of B
      Subset C_i row elems computed.
    Subtask $\Leftarrow \hat{B}_{j+1}$
      C elms are calculated.
- Transmission of columns ensures that each task gets copy of all B columns.

2D "Checkerboard" (or Block-Block) Decomposition

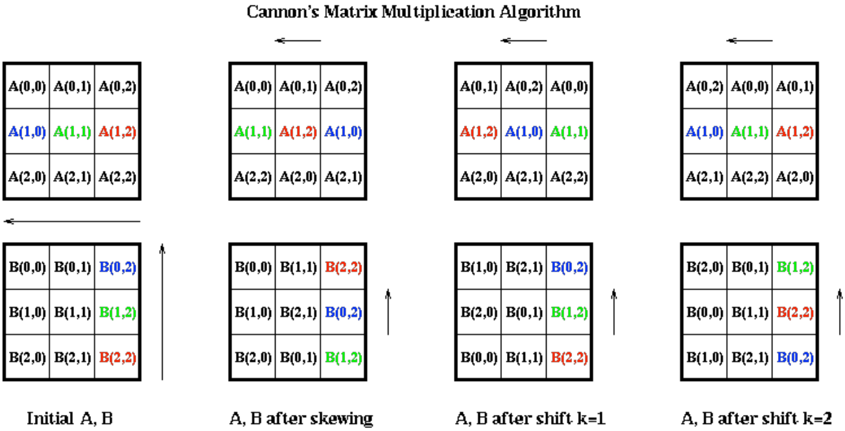
- Use 2D cartesian mapping for Processors
- Use 2D cartesian mapping of the data
- Allocate space on each processor P_{ij} for subarrays of A, B, and C.
- Distribute A,B,C subarrays
- Calculate local data points for C
- Exchange A, B data as needed with neighbors: Cannon, Fox algorithms.



Cannons' Algorithm

- The matrices A and B are NxN matrices
- Compute $C = AxB$
- Circulate blocks of B vertically and blocks of A horizontally in ring fashion
- Blocks of both matrices must be initially aligned using circular shifts so that correct blocks meet as needed
- Requires less memory than Fox algorithm, but trickier to program because of shifts required
- Performance and scalability of Cannon algorithm are not significantly different from other 2-D algorithm, but memory requirements are much less

Cannon Algorithm



Foxs' Algorithm

- See Pacheco: Parallel Programming with MPI (1997): http://www.cs.usfca.edu/ peter/ppmpi/, Ch07.
- Uses matrices $A = [MxN]$ and $B = [NxQ]$
- Computes $C = A \cdot B$, in N Stages, where C is an $[MxQ]$ matrix
- The matrices A and B are partitioned among p processors using "checkerboard" decomposition where:

  $\hat{A}_{00}$ denotes the sub matrix A_{ij} with $0 \leq i \leq M/4$, and $0 \leq j \leq N/4$
- Each processor stores $\left(n/\sqrt{p}\right) x \left(nN/\sqrt{p}\right)$ elements
- At each stage, sub-blocks of A and B are "rotated" into a processor.
- Communication:
  - Broadcast sub-blocks of matrix A along the processor rows.
  - Single-stage circular upwards shifts of the blocks of B sub-diagnals along processor columns
  - Intially, B is distributed across the processors.
  - Initially, each diagonal block $\hat{A}_{ii}$ is selected for broadcast

References:
Goeffrey. Fox, et. al., "Matrix algorithms on a hypercube I: Matrix Multiplication" P. Pacheco, PPMPI 1987 J. Otto, Fox Algorithm descriptions

Foxs' Algorithm

Matrix elements after multiplication for the case of $P = [P_i, P_j] = [3x4] = 12$ processors:

$$P = \begin{bmatrix} P_{00} & P_{01} & P_{02} \\ P_{10} & P_{11} & P_{12} \\ P_{20} & P_{21} & P_{22} \\ P_{30} & P_{31} & P_{32} \end{bmatrix}, A = \begin{bmatrix} \hat{A}_{00} & \hat{A}_{01} & \hat{A}_{02} \\ \hat{A}_{10} & \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{20} & \hat{A}_{21} & \hat{A}_{22} \\ \hat{A}_{30} & \hat{A}_{31} & \hat{A}_{32} \end{bmatrix}, B = \begin{bmatrix} \hat{B}_{00} & \hat{B}_{01} & \hat{B}_{02} \\ \hat{B}_{10} & \hat{B}_{11} & \hat{B}_{12} \\ \hat{B}_{20} & \hat{B}_{21} & \hat{B}_{22} \end{bmatrix},$$

$C = A \cdot B =$

$$\begin{bmatrix} \hat{A}_{00} \cdot \hat{B}_{00} + \hat{A}_{01} \cdot \hat{B}_{10} + \hat{A}_{02} \cdot \hat{B}_{20} & \hat{A}_{00} \cdot \hat{B}_{01} + \hat{A}_{01} \cdot \hat{B}_{11} + \hat{A}_{02} \cdot \hat{B}_{21} & \hat{A}_{00} \cdot \hat{B}_{02} + \hat{A}_{01} \cdot \hat{B}_{12} + \hat{A}_{02} \cdot \hat{B}_{22} \\ \hat{A}_{10} \cdot \hat{B}_{00} + \hat{A}_{11} \cdot \hat{B}_{10} + \hat{A}_{12} \cdot \hat{B}_{20} & \hat{A}_{10} \cdot \hat{B}_{01} + \hat{A}_{11} \cdot \hat{B}_{11} + \hat{A}_{12} \cdot \hat{B}_{21} & \hat{A}_{10} \cdot \hat{B}_{02} + \hat{A}_{11} \cdot \hat{B}_{12} + \hat{A}_{12} \cdot \hat{B}_{22} \\ \hat{A}_{20} \cdot \hat{B}_{00} + \hat{A}_{21} \cdot \hat{B}_{10} + \hat{A}_{22} \cdot \hat{B}_{20} & \hat{A}_{20} \cdot \hat{B}_{01} + \hat{A}_{21} \cdot \hat{B}_{11} + \hat{A}_{22} \cdot \hat{B}_{21} & \hat{A}_{20} \cdot \hat{B}_{02} + \hat{A}_{21} \cdot \hat{B}_{12} + \hat{A}_{22} \cdot \hat{B}_{22} \\ \hat{A}_{30} \cdot \hat{B}_{00} + \hat{A}_{31} \cdot \hat{B}_{10} + \hat{A}_{32} \cdot \hat{B}_{20} & \hat{A}_{30} \cdot \hat{B}_{01} + \hat{A}_{31} \cdot \hat{B}_{11} + \hat{A}_{32} \cdot \hat{B}_{21} & \hat{A}_{30} \cdot \hat{B}_{02} + \hat{A}_{31} \cdot \hat{B}_{12} + \hat{A}_{32} \cdot \hat{B}_{22} \end{bmatrix}$$

Sequential Fox Alg. proceeds in n Stages, where n is the order of the matrices:

$\underline{Stage \quad 0}: c_{ij} = \hat{A}_{i0} \times \hat{B}_{0j}$

$\underline{Stage \quad 1}: c_{ij} = \hat{A}_{i1} \times \hat{B}_{1j}$

$\underline{Stage \quad 2}: c_{ij} = \hat{A}_{i2} \times \hat{B}_{2j}$

$\underline{Stage \quad k}: (1 \leq k < n): c_{ij} = c_{ij} + \hat{A}_{ik} \times \hat{B}_{kj}$

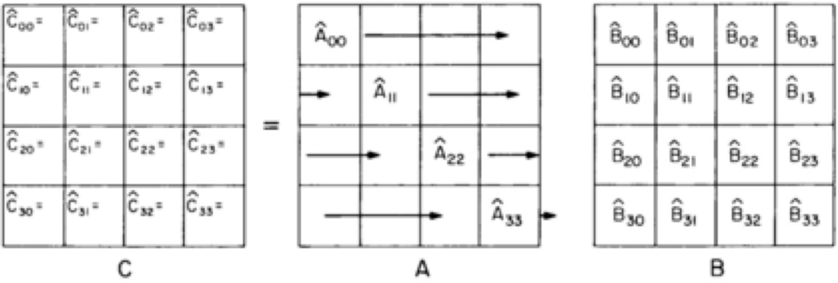
where: $\bar{k} = (i + k) \; mod \quad n.$

Fox: Coarse-Grain 2-D Parallel Algorithm:

- all-to-all bcast $\hat{A}_{ij}$ in *ith* process row horizontal broadcast
- all-to-all bcast $\hat{B}_{ik_J}$ in *jth* process column vertical broadcast

  $c_{ij} = 0$
  for $k = 1; :::; n$
  $\quad c_{ij} = c_{ij} + \hat{A}_{ik} \times \hat{B}_{kj}$
- Algorithm requires excessive memory – each process accumulates blocks of A, B
- Foxs' Solution: Reduce memory:
  - broadcast blocks of A successively across process rows,
  - circulate blocks of B in ring fashion vertically along process columns stage by stage
  - each block of B arrives at appropriate block of A.

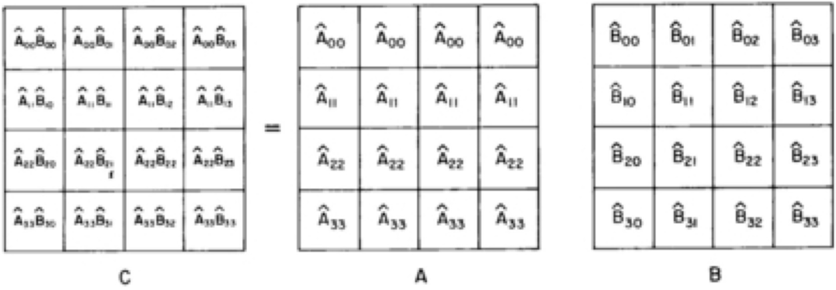
Foxs' Algorithm: Stage 1



(a)

Reference:
G. Fox, et. al., "Matrix algorithms on a hypercube I: Matrix Multiplication" [?]

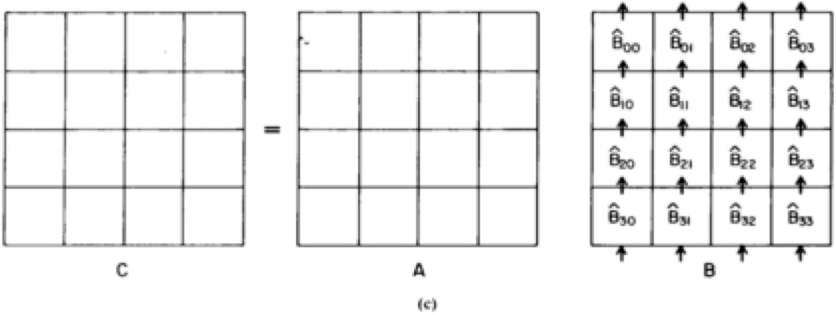
Foxs' Algorithm: Stage 2



(b)

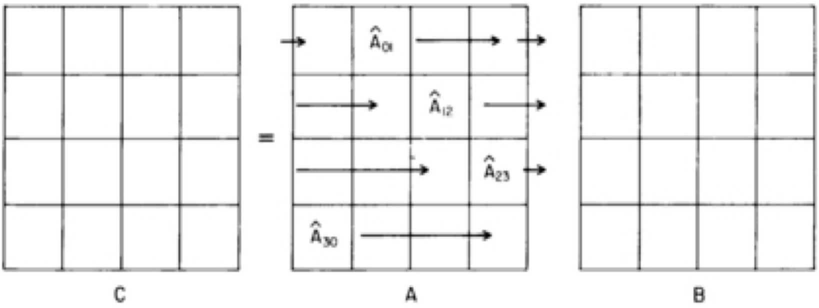
Reference:
G. Fox, et. al., "Matrix algorithms on a hypercube I: Matrix Multiplication" [?]

Foxs' Algorithm: Stage 3



Reference:
G. Fox, et. al., "Matrix algorithms on a hypercube I: Matrix Multiplication" [?]

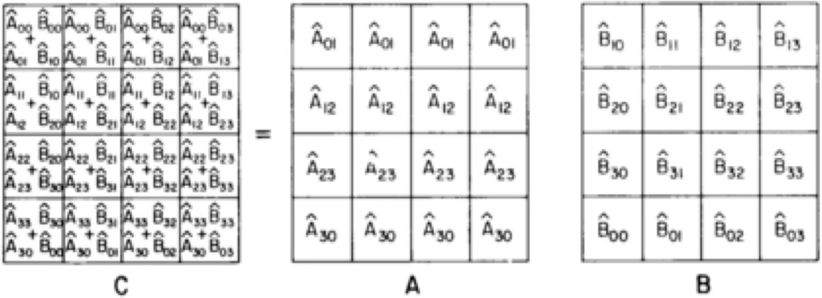
Foxs' Algorithm: Stage 4



(d)

Reference:
G. Fox, et. al., "Matrix algorithms on a hypercube I: Matrix Multiplication" [?]

Foxs' Algorithm: Stage 5



(c)

Reference:
G. Fox, et. al., "Matrix algorithms on a hypercube I: Matrix Multiplication" [?]

Foxs' Algorithm (Otto descr): Stage 0

Stage 0 – uses $diag_0(B)$, original columns of A:

$$A = \begin{bmatrix} \hat{A}_{00} & \hat{A}_{01} & \hat{A}_{02} \\ \hat{A}_{10} & \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{20} & \hat{A}_{21} & \hat{A}_{22} \\ \hat{A}_{30} & \hat{A}_{31} & \hat{A}_{32} \end{bmatrix}, \quad B = \begin{bmatrix} \hat{\mathbf{B}}_{00} & \hat{B}_{01} & \hat{B}_{02} \\ \hat{B}_{10} & \hat{\mathbf{B}}_{11} & \hat{B}_{12} \\ \hat{B}_{20} & \hat{B}_{21} & \hat{\mathbf{B}}_{22} \end{bmatrix} \rightarrow B_0 = \begin{bmatrix} \hat{\mathbf{B}}_{00} \\ \hat{\mathbf{B}}_{10} \\ \hat{\mathbf{B}}_{20} \end{bmatrix},$$

$C = A \cdot B$

$$= \begin{bmatrix} \hat{\mathbf{A}}_{00} \cdot \hat{\mathbf{B}}_{00} + \hat{A}_{01} \cdot \hat{B}_{10} + \hat{A}_{02} \cdot \hat{B}_{20} & \hat{A}_{00} \cdot \hat{B}_{01} + \hat{\mathbf{A}}_{01} \cdot \hat{\mathbf{B}}_{11} + \hat{A}_{02} \cdot \hat{B}_{21} & \hat{A}_{00} \cdot \hat{B}_{02} + \hat{A}_{01} \cdot \hat{B}_{12} + \hat{\mathbf{A}}_{02} \cdot \hat{\mathbf{B}}_{22} \\ \hat{\mathbf{A}}_{10} \cdot \hat{\mathbf{B}}_{00} + \hat{A}_{11} \cdot \hat{B}_{10} + \hat{A}_{12} \cdot \hat{B}_{20} & \hat{A}_{10} \cdot \hat{B}_{01} + \hat{\mathbf{A}}_{11} \cdot \hat{\mathbf{B}}_{11} + \hat{A}_{12} \cdot \hat{B}_{21} & \hat{A}_{10} \cdot \hat{B}_{02} + \hat{A}_{11} \cdot \hat{B}_{12} + \hat{\mathbf{A}}_{12} \cdot \hat{\mathbf{B}}_{22} \\ \hat{\mathbf{A}}_{20} \cdot \hat{\mathbf{B}}_{00} + \hat{A}_{21} \cdot \hat{B}_{10} + \hat{A}_{22} \cdot \hat{B}_{20} & \hat{A}_{20} \cdot \hat{B}_{01} + \hat{\mathbf{A}}_{21} \cdot \hat{\mathbf{B}}_{11} + \hat{A}_{22} \cdot \hat{B}_{21} & \hat{A}_{20} \cdot \hat{B}_{02} + \hat{A}_{21} \cdot \hat{B}_{12} + \hat{\mathbf{A}}_{22} \cdot \hat{\mathbf{B}}_{22} \\ \hat{\mathbf{A}}_{30} \cdot \hat{\mathbf{B}}_{00} + \hat{A}_{31} \cdot \hat{B}_{10} + \hat{A}_{32} \cdot \hat{B}_{20} & \hat{A}_{30} \cdot \hat{B}_{01} + \hat{\mathbf{A}}_{31} \cdot \hat{\mathbf{B}}_{11} + \hat{A}_{32} \cdot \hat{B}_{21} & \hat{A}_{30} \cdot \hat{B}_{02} + \hat{A}_{31} \cdot \hat{B}_{12} + \hat{\mathbf{A}}_{32} \cdot \hat{\mathbf{B}}_{22} \end{bmatrix}$$

Foxs' Algorithm: Stage 1

Stage 1 – uses $diag_{-1}(B)$, [shift B ↓] original columns of A:

$$A = \begin{bmatrix} \hat{A}_{00} & \hat{A}_{01} & \hat{A}_{02} \\ \hat{A}_{10} & \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{20} & \hat{A}_{21} & \hat{A}_{22} \\ \hat{A}_{30} & \hat{A}_{31} & \hat{A}_{32} \end{bmatrix}, \quad B = \begin{bmatrix} \hat{B}_{00} & \hat{B}_{01} & \mathbf{\hat{B}_{02}} \\ \mathbf{\hat{B}_{10}} & \hat{B}_{11} & \hat{B}_{12} \\ \hat{B}_{20} & \mathbf{\hat{B}_{21}} & \hat{B}_{22} \end{bmatrix} \rightarrow \quad B_0 = \begin{bmatrix} \mathbf{\hat{B}_{10}} \\ \mathbf{\hat{B}_{21}} \\ \mathbf{\hat{B}_{02}} \end{bmatrix},$$

$$C = A \bullet B$$

$$= \begin{bmatrix} \mathbf{\hat{A}_{00}} \cdot \mathbf{\hat{B}_{00}} + \mathbf{\hat{A}_{01}} \cdot \mathbf{\hat{B}_{10}} + \hat{A}_{02} \cdot \hat{B}_{20} & \hat{A}_{00} \cdot \hat{B}_{01} + \mathbf{\hat{A}_{01}} \cdot \mathbf{\hat{B}_{11}} + \mathbf{\hat{A}_{02}} \cdot \mathbf{\hat{B}_{21}} & \mathbf{\hat{A}_{00}} \cdot \mathbf{\hat{B}_{02}} + \hat{A}_{01} \cdot \hat{B}_{12} + \mathbf{\hat{A}_{02}} \cdot \mathbf{\hat{B}_{22}} \\ \mathbf{\hat{A}_{10}} \cdot \mathbf{\hat{B}_{00}} + \mathbf{\hat{A}_{11}} \cdot \mathbf{\hat{B}_{10}} + \hat{A}_{12} \cdot \hat{B}_{20} & \hat{A}_{10} \cdot \hat{B}_{01} + \mathbf{\hat{A}_{11}} \cdot \mathbf{\hat{B}_{11}} + \mathbf{\hat{A}_{12}} \cdot \mathbf{\hat{B}_{21}} & \mathbf{\hat{A}_{10}} \cdot \mathbf{\hat{B}_{02}} + \hat{A}_{11} \cdot \hat{B}_{12} + \mathbf{\hat{A}_{12}} \cdot \mathbf{\hat{B}_{22}} \\ \mathbf{\hat{A}_{20}} \cdot \mathbf{\hat{B}_{00}} + \mathbf{\hat{A}_{21}} \cdot \mathbf{\hat{B}_{10}} + \hat{A}_{22} \cdot \hat{B}_{20} & \hat{A}_{20} \cdot \hat{B}_{01} + \mathbf{\hat{A}_{21}} \cdot \mathbf{\hat{B}_{11}} + \mathbf{\hat{A}_{22}} \cdot \mathbf{\hat{B}_{21}} & \mathbf{\hat{A}_{20}} \cdot \mathbf{\hat{B}_{02}} + \hat{A}_{21} \cdot \hat{B}_{12} + \mathbf{\hat{A}_{22}} \cdot \mathbf{\hat{B}_{22}} \\ \mathbf{\hat{A}_{30}} \cdot \mathbf{\hat{B}_{00}} + \mathbf{\hat{A}_{31}} \cdot \mathbf{\hat{B}_{10}} + \hat{A}_{32} \cdot \hat{B}_{20} & \hat{A}_{30} \cdot \hat{B}_{01} + \mathbf{\hat{A}_{31}} \cdot \mathbf{\hat{B}_{11}} + \mathbf{\hat{A}_{32}} \cdot \mathbf{\hat{B}_{21}} & \mathbf{\hat{A}_{30}} \cdot \mathbf{\hat{B}_{02}} + \hat{A}_{31} \cdot \hat{B}_{12} + \mathbf{\hat{A}_{32}} \cdot \mathbf{\hat{B}_{22}} \end{bmatrix}$$

Foxs' Algorithm: Stage 2

Stage 2 – uses $diag_{-2}(B)$, [shift B ↓] original columns of A:

$$A = \begin{bmatrix} \hat{A}_{00} & \hat{A}_{01} & \hat{A}_{02} \\ \hat{A}_{10} & \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{20} & \hat{A}_{21} & \hat{A}_{22} \\ \hat{A}_{30} & \hat{A}_{31} & \hat{A}_{32} \end{bmatrix}, \quad B = \begin{bmatrix} \hat{B}_{00} & \hat{B}_{01} & \hat{B}_{02} \\ \hat{B}_{10} & \hat{B}_{11} & \hat{B}_{12} \\ \hat{B}_{20} & \hat{B}_{21} & \hat{B}_{22} \end{bmatrix} \rightarrow B_0 = \begin{bmatrix} \hat{B}_{20} \\ \hat{B}_{01} \\ \hat{B}_{12} \end{bmatrix},$$

$$C = A \bullet B$$

$$= \begin{bmatrix} \hat{A}_{00}\cdot\hat{B}_{00} + \hat{A}_{01}\cdot\hat{B}_{10} + \hat{A}_{02}\cdot\hat{B}_{20} & \hat{A}_{00}\cdot\hat{B}_{01} + \hat{A}_{01}\cdot\hat{B}_{11} + \hat{A}_{02}\cdot\hat{B}_{21} & \hat{A}_{00}\cdot\hat{B}_{02} + \hat{A}_{01}\cdot\hat{B}_{12} + \hat{A}_{02}\cdot\hat{B}_{22} \\ \hat{A}_{10}\cdot\hat{B}_{00} + \hat{A}_{11}\cdot\hat{B}_{10} + \hat{A}_{12}\cdot\hat{B}_{20} & \hat{A}_{10}\cdot\hat{B}_{01} + \hat{A}_{11}\cdot\hat{B}_{11} + \hat{A}_{12}\cdot\hat{B}_{21} & \hat{A}_{10}\cdot\hat{B}_{02} + \hat{A}_{11}\cdot\hat{B}_{12} + \hat{A}_{12}\cdot\hat{B}_{22} \\ \hat{A}_{20}\cdot\hat{B}_{00} + \hat{A}_{21}\cdot\hat{B}_{10} + \hat{A}_{22}\cdot\hat{B}_{20} & \hat{A}_{20}\cdot\hat{B}_{01} + \hat{A}_{21}\cdot\hat{B}_{11} + \hat{A}_{22}\cdot\hat{B}_{21} & \hat{A}_{20}\cdot\hat{B}_{02} + \hat{A}_{21}\cdot\hat{B}_{12} + \hat{A}_{22}\cdot\hat{B}_{22} \\ \hat{A}_{30}\cdot\hat{B}_{00} + \hat{A}_{31}\cdot\hat{B}_{10} + \hat{A}_{32}\cdot\hat{B}_{20} & \hat{A}_{30}\cdot\hat{B}_{01} + \hat{A}_{31}\cdot\hat{B}_{11} + \hat{A}_{32}\cdot\hat{B}_{21} & \hat{A}_{30}\cdot\hat{B}_{02} + \hat{A}_{31}\cdot\hat{B}_{12} + \hat{A}_{32}\cdot\hat{B}_{22} \end{bmatrix}$$

Modifications

- Source: fox.c – uses Foxs' algorithm to multiply two square matrices
- From: Pacheco: Parallel Programming with MPI (1997): http://www.cs.usfca.edu/ peter/ppmpi/, Ch07.
- If you work with Pacheco Code, some changes are required
  - modify basic data type.
  - hard coded dimensions.
  - change from terminal input to command line args

Example: *fox.c* (1/9)

```
/*****************************************************/
* uses Foxs' algorithm to multiply two square matrices
 * Input:
 *     n: global order of matrices
 *     A,B: the factor matrices
 * Output:
 *     C: the product matrix
 *
 * Notes:
 *     1.  Assumes the number of processes is a perfect square
 *     2.  The array member of the matrices is statically allocated
 *     3.  Assumes the global order of the matrices is evenly divisible by sqrt(p).
 *
 * See Chap 7, pp. 113 & ff and pp. 125 & ff in PPMPI
 */
#include <stdio.h>
#include "mpi.h"
#include <math.h>
#include <stdlib.h>
```

```
typedef struct {
    int       p;         /* Total number of processes    */
    MPI_Comm  comm;      /* Communicator for entire grid */
    MPI_Comm  row_comm;  /* Communicator for my row      */
    MPI_Comm  col_comm;  /* Communicator for my col      */
    int       q;         /* Order of grid                */
    int       my_row;    /* My row number                */
    int       my_col;    /* My column number             */
    int       my_rank;   /* My rank in the grid comm     */
} GRID_INFO_T;
```

```
#define MAX 65536
typedef struct {
    int     n_bar;
#define Order(A) ((A)->n_bar)
    float  entries[MAX];
#define Entry(A,i,j) (*(((A)->entries) + ((A)->n_bar)*(i) + (j)))
} LOCAL_MATRIX_T;
```

```
/*****************  fox.c c  (2/9)  ****************/
 /* Function Declarations */
LOCAL_MATRIX_T*  Local_matrix_allocate(int n_bar);
void             Free_local_matrix(LOCAL_MATRIX_T** local_A);
void             Read_matrix(char* prompt, LOCAL_MATRIX_T* local_A,
                     GRID_INFO_T* grid, int n);
void             Print_matrix(char* title, LOCAL_MATRIX_T* local_A,
                     GRID_INFO_T* grid, int n);
void             Set_to_zero(LOCAL_MATRIX_T* local_A);
void             Local_matrix_multiply(LOCAL_MATRIX_T* local_A,
                     LOCAL_MATRIX_T* local_B, LOCAL_MATRIX_T* local_C);
void             Build_matrix_type(LOCAL_MATRIX_T* local_A);
MPI_Datatype     local_matrix_mpi_t;

LOCAL_MATRIX_T*  temp_mat;
void             Print_local_matrices(char* title, LOCAL_MATRIX_T* local_A,
                     GRID_INFO_T* grid);

/*********************************************************/
main(int argc, char* argv[]) {
    int              p;
    int              my_rank;
    GRID_INFO_T      grid;
    LOCAL_MATRIX_T*  local_A;
    LOCAL_MATRIX_T*  local_B;
    LOCAL_MATRIX_T*  local_C;
    int              n;
    int              n_bar;

    void Setup_grid(GRID_INFO_T*  grid);
    void Fox(int n, GRID_INFO_T* grid, LOCAL_MATRIX_T* local_A,
             LOCAL_MATRIX_T* local_B, LOCAL_MATRIX_T* local_C);

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
/******************  fox.c c  (3/9)  *****************/
    Setup_grid(&grid);
   if (my_rank == 0) {
       printf("What's the order of the matrices?\n");
       scanf("%d", &n);
   }

   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   n_bar = n/grid.q;

   local_A = Local_matrix_allocate(n_bar);
   Order(local_A) = n_bar;
   Read_matrix("Enter A", local_A, &grid, n);
   Print_matrix("We read A =", local_A, &grid, n);

   local_B = Local_matrix_allocate(n_bar);
   Order(local_B) = n_bar;
   Read_matrix("Enter B", local_B, &grid, n);
   Print_matrix("We read B =", local_B, &grid, n);

   Build_matrix_type(local_A);
   temp_mat = Local_matrix_allocate(n_bar);

   local_C = Local_matrix_allocate(n_bar);
   Order(local_C) = n_bar;
   Fox(n, &grid, local_A, local_B, local_C);

   Print_matrix("The product is", local_C, &grid, n);

   Free_local_matrix(&local_A);
   Free_local_matrix(&local_B);
   Free_local_matrix(&local_C);

   MPI_Finalize();
}  /* main */
```

```
/*****************  fox.c  (4/9)  *****************/
void Setup_grid(
    GRID_INFO_T*  grid  /* out */) {
    int old_rank;
    int dimensions[2],   wrap_around[2];
    int coordinates[2],    free_coords[2];

    /* Set up Global Grid Information */
    MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
    MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);

    /* We assume p is a perfect square */
    grid->q = (int) sqrt((double) grid->p);
    dimensions[0] = dimensions[1] = grid->q;

    /* We want a circular shift in second dimension. */
    /* Don't care about first                        */
    wrap_around[0] = wrap_around[1] = 1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
        wrap_around, 1, &(grid->comm));
    MPI_Comm_rank(grid->comm, &(grid->my_rank));
    MPI_Cart_coords(grid->comm, grid->my_rank, 2,
        coordinates);
    grid->my_row = coordinates[0];
    grid->my_col = coordinates[1];

    /* Set up row communicators */
    free_coords[0] = 0;
    free_coords[1] = 1;
    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->row_comm));

    /* Set up column communicators */
    free_coords[0] = 1;
    free_coords[1] = 0;
    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->col_comm));
} /* Setup_grid */
```

```
/*****************  fox.c  (5/9)  ****************/
void Fox(  int              n         /* in  */,
       GRID_INFO_T*     grid      /* in  */,
       LOCAL_MATRIX_T*  local_A   /* in  */,
       LOCAL_MATRIX_T*  local_B   /* in  */,
       LOCAL_MATRIX_T*  local_C   /* out */) {
/* Storage for submatrix of A used during current stage */
    LOCAL_MATRIX_T*  temp_A;
    int  stage, bcast_root, n_bar, source, dest;
    MPI_Status    status;
    n_bar = n/grid->q;
    Set_to_zero(local_C);

    /* Calculate addresses for circular shift of B */
    source = (grid->my_row + 1) % grid->q;
    dest = (grid->my_row + grid->q - 1) % grid->q;

    /* Set aside storage for the broadcast block of A */
    temp_A = Local_matrix_allocate(n_bar);

    for (stage = 0; stage < grid->q; stage++) {
        bcast_root = (grid->my_row + stage) % grid->q;
        if (bcast_root == grid->my_col) {
            MPI_Bcast(local_A, 1, local_matrix_mpi_t,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(local_A, local_B,
                local_C);
        } else {
            MPI_Bcast(temp_A, 1, local_matrix_mpi_t,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(temp_A, local_B,
                local_C);
        }
        MPI_Sendrecv_replace(local_B, 1, local_matrix_mpi_t,
            dest, 0, source, 0, grid->col_comm, &status);
    } /* for */
} /* Fox */
```

```
/*********************************************/
void Local_matrix_multiply(
         LOCAL_MATRIX_T*  local_A  /* in  */,
         LOCAL_MATRIX_T*  local_B  /* in  */,
         LOCAL_MATRIX_T*  local_C  /* out */) {
  int i, j, k;

  for (i = 0; i < Order(local_A); i++)
    for (j = 0; j < Order(local_A); j++)
      for (k = 0; k < Order(local_B); k++)
        Entry(local_C,i,j) = Entry(local_C,i,j)
            + Entry(local_A,i,k)*Entry(local_B,k,j);

}  /* Local_matrix_multiply */
```

```
/***************   fox.c  (6/9)   ********************/
 * Read and distribute matrix:
 *  foreach global row of the matrix,
 *  foreach grid column
 *    - read block of n_bar floats  on proc 0.
 *    - send  to the appropriate  processor
 */
void Read_matrix(
   char*             prompt   /* in  */,
   LOCAL_MATRIX_T*  local_A  /* out */,
   GRID_INFO_T*     grid     /* in  */,
   int              n        /* in  */)  {
   int        mat_row, mat_col,    grid_row, grid_col,    dest;
   int        coords[2];
   float*     temp;
   MPI_Status status;
 if (grid->my_rank == 0) {
  temp = (float*) malloc(Order(local_A)*sizeof(float));
  printf("%s\n", prompt);
  fflush(stdout);
  for (mat_row = 0;  mat_row < n; mat_row++) {
    grid_row = mat_row/Order(local_A);
    coords[0] = grid_row;
    for (grid_col = 0; grid_col < grid->q; grid_col++) {
        coords[1] = grid_col;
        MPI_Cart_rank(grid->comm, coords, &dest);
        if (dest == 0) {
            for (mat_col = 0; mat_col < Order(local_A); mat_col++)
                    scanf("%f", (local_A->entries)+mat_row*Order(local_A)+mat_col);
        } else {
            for(mat_col = 0; mat_col < Order(local_A); mat_col++)
                scanf("%f", temp + mat_col);
                MPI_Send(temp, Order(local_A), MPI_FLOAT, dest, 0, grid->comm);     }       }      }
        free(temp);
  } else {
    for (mat_row = 0; mat_row < Order(local_A); mat_row++)
        MPI_Recv(&Entry(local_A, mat_row, 0), Order(local_A),
        MPI_FLOAT, 0, 0, grid->comm, &status);
  }
}  /* Read_matrix */
```

```
 /***************   fox.c (7/9)   ********************/
void Print_matrix(
         char*            title    /* in  */,
         LOCAL_MATRIX_T*  local_A  /* out */,
         GRID_INFO_T*     grid     /* in  */,
         int              n        /* in  */) {
    int        mat_row, mat_col, grid_row, grid_col, source;
    int        coords[2];
    float*     temp;
    MPI_Status status;

    if (grid->my_rank == 0) {
        temp = (float*) malloc(Order(local_A)*sizeof(float));
        printf("%s\n", title);
        for (mat_row = 0;  mat_row < n; mat_row++) {
            grid_row = mat_row/Order(local_A);
            coords[0] = grid_row;
            for (grid_col = 0; grid_col < grid->q; grid_col++) {
                coords[1] = grid_col;
                MPI_Cart_rank(grid->comm, coords, &source);
                if (source == 0) {
                    for(mat_col = 0; mat_col < Order(local_A); mat_col++)
                        printf("%4.1f ", Entry(local_A, mat_row, mat_col));
                } else {
                    MPI_Recv(temp, Order(local_A), MPI_FLOAT, source, 0,
                        grid->comm, &status);
                    for(mat_col = 0; mat_col < Order(local_A); mat_col++)
                        printf("%4.1f ", temp[mat_col]);
                }
            }
            printf("\n");
        }
        free(temp);
    } else {
        for (mat_row = 0; mat_row < Order(local_A); mat_row++)
            MPI_Send(&Entry(local_A, mat_row, 0), Order(local_A),
                MPI_FLOAT, 0, 0, grid->comm);
    }
}  /* Print_matrix */
```

Example: *fox.c c* (4/5)

```
 /****************   fox.c (8/9)   *********************/
void Print_local_matrices(
         char*            title    /* in */,
         LOCAL_MATRIX_T*  local_A  /* in */,
         GRID_INFO_T*     grid     /* in */) {

    int         coords[2];
    int         i, j;
    int         source;
    MPI_Status  status;

    if (grid->my_rank == 0) {
        printf("%s\n", title);
        printf("Process %d > grid_row = %d, grid_col = %d\n",
            grid->my_rank, grid->my_row, grid->my_col);
        for (i = 0; i < Order(local_A); i++) {
            for (j = 0; j < Order(local_A); j++)
                printf("%4.1f ", Entry(local_A,i,j));
            printf("\n");
        }
        for (source = 1; source < grid->p; source++) {
            MPI_Recv(temp_mat, 1, local_matrix_mpi_t, source, 0,
                grid->comm, &status);
            MPI_Cart_coords(grid->comm, source, 2, coords);
            printf("Process %d > grid_row = %d, grid_col = %d\n",
                source, coords[0], coords[1]);
            for (i = 0; i < Order(temp_mat); i++) {
                for (j = 0; j < Order(temp_mat); j++)
                    printf("%4.1f ", Entry(temp_mat,i,j));
                printf("\n");
            }
        }
        fflush(stdout);
    } else {
        MPI_Send(local_A, 1, local_matrix_mpi_t, 0, 0, grid->comm);
    }
}  /* Print_local_matrices */
```

```
/****************   fox.c (9/9)   *****************/
void Build_matrix_type(
         LOCAL_MATRIX_T*  local_A  /* in */) {
    MPI_Datatype  temp_mpi_t;
    int           block_lengths[2];
    MPI_Aint      displacements[2];
    MPI_Datatype  typelist[2];
    MPI_Aint      start_address;
    MPI_Aint      address;

    MPI_Type_contiguous(
        Order(local_A)*Order(local_A),
        MPI_FLOAT, &temp_mpi_t  );

    block_lengths[0] = block_lengths[1] = 1;

    typelist[0] = MPI_INT;
    typelist[1] = temp_mpi_t;

    MPI_Address(local_A, &start_address);
    MPI_Address(&(local_A->n_bar), &address);
    displacements[0] = address - start_address;

    MPI_Address(local_A->entries, &address);
    displacements[1] = address - start_address;

    MPI_Type_struct(2, block_lengths, displacements,
        typelist, &local_matrix_mpi_t);
    MPI_Type_commit(&local_matrix_mpi_t);
}  /* Build_matrix_type */
```

```
LOCAL_MATRIX_T* Local_matrix_allocate(int local_order)
{
  LOCAL_MATRIX_T* temp;
  temp = (LOCAL_MATRIX_T*) malloc(sizeof(LOCAL_MATRIX_T));
  return temp;
}  /* Local_matrix_allocate */


/*********************************************/
void Free_local_matrix(
         LOCAL_MATRIX_T** local_A_ptr  /* in/out */) {
    free(*local_A_ptr);
}  /* Free_local_matrix */



/*********************************************/
void Set_to_zero(
         LOCAL_MATRIX_T*  local_A  /* out */) {

    int i, j;

    for (i = 0; i < Order(local_A); i++)
        for (j = 0; j < Order(local_A); j++)
            Entry(local_A,i,j) = 0.0;

}  /* Set_to_zero */
```