COSC 6374
Parallel Computation

Dense Matrix Operations

Edgar Gabriel
Fall 2011

Terminology

- Dense Matrix: all elements of the matrix contain relevant values
  - Typically stored as 2-D array, (e.g. double a[16][32]);
- Sparse matrix: most elements of the matrix are zero
  - Optimized storage techniques
    - Band matrices: store only the relevant diagonals of the matrix
    - Highly irregular sparse matrices: store the coordinates of every non-zero element together with the content
    - Boeing-Harwell format: exploit certain regularities (e.g. nearly constant number of entries per row or column)
    - Jagged Diagonal storage format: see Boeing-Harwell format
Replication vs. Communication

- Large data items typically distributed across multiple processes
  - What is large?
- Small data items can replicated on all processes or communicated whenever required
  - Costs for communication: network latency
  - Costs for replication: memory consumption + repeated computation operations

Matrix operations: \( B = c \times A \)

- Multiplying a Matrix \( A \) with a constant \( c \)
- Constant \( c \) is definitely small and is thus replicated on all processes
  - E.g. compiled in the code
  - Read from a configuration file
- Operation does not require any communication to be performed
  - Trivially parallel
- Operation can be performed independent of the way the matrix has been distributed across the processes
Matrix Operations: $B = A^T$

- Transpose a Matrix
  - Often not necessary, since the operations (e.g. Matrix-vector multiply) can be (easily) reformulated for Matrix-Transpose-vector multiply operations and avoid the data transpose
  - Operations requiring the transpose: multi-dimensional FFT
- Assumption:
  - Matrices $A$, $B$ are square
  - Element $A[x][y]$ should be on the same process as element $B[y][x]$
  - $\Rightarrow$ requires communication across the processes

$B = A^T$: One element per process

- Initial data distribution: one element of the Matrix $A$ per process

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

- Process with coordinates $(x,y)$ needs to send its data item to the process with the coordinates $(y,x)$ and receive its data item from $(y,x)$
\[ B = A^T : \text{One element per process} \]

```c
// Assumptions:
//    newcomm has been created using MPI_Cart_create
//    double A, B are the element of the matrices
//    owned by each process. A is already set.

int coords[2]; // my coordinates in the 2-D topology
int rem_coords[2]; // coordinates of my counterpart
MPI_Request req[2];
MPI_Status stats[2];

// Determine my own rank in newcomm
MPI_Comm_rank (newcomm, &rank);
// Determine my own coordinates in newcomm
MPI_Cart_coords (newcomm, rank, ndims, coords);

// Determine the coordinates of my counterpart
rem_coords[0] = coords[1];
rem_coords[1] = coords[0];
```

\[ B = A^T : \text{One element per process} \]

```c
// Determine the rank of my counterpart using his coordinates
MPI_Cart_rank (newcomm, rem_coords, &rem_rank);

// Initiate non-blocking communication to send A
MPI_Isend (&A, 1, MPI_DOUBLE, rem_rank, 0, newcomm,&req[0]);
// Initiate non-blocking communication to receive B
MPI_Irecv (&B, 1, MPI_DOUBLE, rem_rank, 0, newcomm,&req[1]);

// Wait on both non-blocking operations to finish
MPI_Waitall (2, req, stats);
```

- **Notes:**
  - using non-blocking communication avoids to have to ‘schedule’ messages to avoid deadlock
  - processes on the main diagonal send a message to themselves
$B = A^T$: Column-wise data distribution

- One column per process

\[
\begin{array}{c}
\text{rank} = 0 1 2 3 4 5 6 7 8 \\
\text{rank} = 0 1 2 3 4 5 6 7 8
\end{array}
\]

- Element $A[i]$ needs to be sent to process $i$
- Element $B[i]$ will be received from process $i$

\[\text{T}A
\begin{bmatrix}
A & B \\
\end{bmatrix}
\]

```c
MPI_Request *reqs;
MPI_Status *stats;
int rank, size;
double A[N], B[N];

// Determine the number of processes working on the problem
// and my rank in the communicator
MPI_Comm_size ( comm, &size);
MPI_Comm_rank ( comm, &rank);

// Allocate the required number of Requests and Statuses. Since
// the code is supposed to work for arbitrary numbers of
// processors, you can not use static arrays for reqs and stats
reqs = (MPI_Request *) malloc ( 2*size*sizeof(MPI_Request) );
stats = (MPI_Status *) malloc ( 2*size*sizeof(MPI_Status) );
```
$B = A^T$: Column-wise data distribution

```c
// Start now all non-blocking communication operations
for (i=0; i<size; i++) {
    MPI_Isend (&A[i], 1, MPI_DOUBLE, i, 0, comm, &reqs[2*i]);
    MPI_Irecv (&B[i], 1, MPI_DOUBLE, i, 0, comm, &(reqs[2*i+1]));
}

// Wait for all non-blocking operations to finish
MPI_Waitall (2*size, reqs, stats);
```

• Notes:
  - identical approach and code for row-wise data distribution as long as the local portions of both A and B are stored as one-dimensional arrays
  - number of messages: $N^2 = np^2$

---

$B = A^T$: Block column-wise data distribution

• Each process holds $N_{local}$ columns of each matrix with
  
  $$N = \sum_{r=0}^{np-1} N_{local}$$

  - assuming $N$ can be divided evenly onto $np$ processes

```
 rank = 0 1 2
```

```
 rank = 0 1 2
```

A

B
\[ B = A^T : \text{Block column-wise data distribution} \]

- Element \( A[i][j] \) has to become element \( B[j][i] \)
  - assuming \( i, j \) are global indexes
- Variable declarations on each process:
  
  ```
  double A[N][N_{local}];
  double B[N][N_{local}];
  ```
  
- \( A[i][j] \)
  - is located on the process with the rank \( r = j/N_{local} \)
  - has the local indexes \( A[i_1][j_1] \) with
    
    \[
    i_1 = i \quad \text{and} \quad j_1 = j \% N_{local}
    \]
- \( B[j][i] \)
  - is located on the process with the rank \( s = i/N_{local} \)
  - has the local indexes \( B[j][i_2] \) with \( j_2 = j \) and
    
    \[
    i_2 = i \% N_{local}
    \]

\[ B = A^T : \text{Block column-wise data distribution} \]

```cpp
// code fragment for the communication
for ( j1=0; j1<N_{local}; j1++) {
    for (i=0; i<N; i++ ) {
        dest = i / N_{local};
        MPI_Isend ( &(A[i][j1], 1, MPI_DOUBLE, dest, 0, comm, &reqs[...]));
    } }

for ( j=0; j<N; j++ ) {
    for ( i2=0; i2<N_{local}; i2++) {
        src = j / N_{local};
        MPI_Irecv ( &(B[j][i2]), 1, MPI_DOUBLE, src, 0, comm, &reqs[...]);
    } }
```
\[ B = A^T: \] Block column-wise data distribution

- The algorithm on the previous slide is good because
  - it doesn’t require any additional temporary storage
- The algorithm on the previous slide is bad because
  - it sends \( N^2 \) messages, with \( N \gg np \)
  - costs of each message is proportional to the network latency for short messages
  - Matrix \( A \) has to be traversed in a non-contiguous manner
    - \( C \) stores multi-dimensional arrays in row-major order
    - accessing \( A[0][0] \) than \( A[1][0] \) means that we jump in the main memory and have a large number of cache misses

Memory layout of multi-dimensional arrays

- E.g. 2-D matrix

- Memory layout in C

- Memory layout in Fortran
Parallel Computation
Edgar Gabriel

$B = A^T$: Block column-wise data distribution

- Alternative algorithm
  - each process sends in reality $N_{local} \times N_{local}$ elements to every other process
  - send an entire block of $N_{local} \times N_{local}$ elements
  - block has to be transposed either at the sender or at the receiver

\[
A \quad B
\]
\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{array}
\]

// Send the matrix block-by-block
for ( i=0; i<N; i+=N_{local} ) {
    MPI_Isend ( &(A[i][0], N_{local} \times N_{local}, MPI_DOUBLE, j, 0, comm, &reqs[2*i]);
    MPI_Irecv( &(B[i][0], N_{local} \times N_{local}, MPI_DOUBLE, j, 0, comm, &reqs[2*i+1]);
} } MPI_Waitall ( 2*size, reqs, stats);

// Now transpose each block
for ( i=0; i<N; i+=N_{local} ) {
    for ( k=0; k<N_{local}; k++ ) {
        for ( j=k; j<N_{local}; j++ ) {
            temp = B[i+k][j];
            B[i+k][j] = B[i+j][k];
            B[i+j][k] = temp;
        }
    }
}
$B = A^T$: other 1-D data distributions

- Block row-wise data distribution
  - algorithm very similar to block column-wise data distribution
- Cyclic column-wise data distribution
  - process with rank $r$ gets the columns $r, r+np, r+2np, \text{etc...}$
  - advantage:
    - none for the Matrix transpose operations
    - for some other operations, this data distribution leads often to better load balance than block column-wise distribution
- Cyclic row-wise data distribution
- Block-cyclic column-wise data distribution
- Block-cyclic row-wise data distribution

$B = A^T$: 2-D data distribution

- Each process holds a block of $N_{local} \times N_{local}$ elements
  - 2-D distribution avoids ‘skinny’ matrices
  - often easier to create load balance than with 1-D block column/row distribution
**B = A^T**: 2-D data distribution

- Assumption: using 2-D cartesian communicator
- Algorithm:
  - Determine your rank using `MPI_Comm_rank`
  - Determine your coordinates using `MPI_Cart_coords`
  - Determine the coordinates of your communication partner by reverting the x and y coordinates of your coordinates
  - Determine the rank of your communication partner using `MPI_Cart_rank`
  - Send a block of \( N_{\text{local}} \times N_{\text{local}} \) elements to comm. partner
  - Receive block of \( N_{\text{local}} \times N_{\text{local}} \) elements from comm. partner
  - Transpose the block that has been received
- Algorithm combines techniques from the ‘one element per process’ distribution and the ‘block column-wise’ distribution

**c = A \cdot b**: block row-wise distribution

- replicating the vector

```c
double A[nlocal][n], b[n];
double c[nlocal], cglobal[n];
int i, j;
...
for (i=0; i<nlocal; i++) {
    for ( j=0; j<n; j++ ) {
        c[i] = c[i] + A(i,j)*b(j);
    }
}
MPI_Allgather( c, nlocal, MPI_DOUBLE, cglobal, nlocal, MPI_DOUBLE, MPI_COMM_WORLD );
```
\[ c = A \cdot b : \text{block row-wise distribution} \]

- Why replicate the vector?
  - memory requirement is \( O(N) \) with \( N \) being the size of the vector
  - in contrary to Matrix \( O(N^2) \) or other higher dimensional arrays
  - increases the performance of the Matrix-vector multiply operation
- Why do we need the \texttt{Allgather} at the end?
  - most applications require a uniform treatment of similar objects
    - e.g. one vector is replicated, all should be replicated
    - if the result vector \( c \) is used in a subsequent operation, you would need different implementations in the code depending on whether the vector is distributed or replicated

```c
int main( int argc, char **argv) {
    double A[n][nlocal], b[nlocal];
    double c[n], ct[n];
    int i,j;
    ...
    for (i=0; i<n; i++) {
        for ( j=0; j<nlocal; j++ ) {
            ct[i] = ct[i] + A(i,j)*b(j);
        }
    }
    MPI_Allreduce ( ct, c, n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD );
}
```
\[ c = A \cdot b : \text{block column-wise distribution} \]

- Why not replicate the vector \( b \) in this distribution
  - there is no benefit in doing that for this operation
  - there might be other operations in the code that mandate that
- But the result vector is replicated...
  - sure, the algorithm mandates that
  - you can still drop the elements that you don’t need afterwards

\[ C = A \cdot B : \text{Matrix-Matrix Multiply} \]

- Block-column wise data distribution
- Example for 2 processes and a 4x4 matrix
- Example uses global indices for the matrix

\[
\begin{align*}
\text{Matrix A} & \quad \text{Matrix B} & \quad \text{Matrix C} \\
\begin{bmatrix}
    a_{00} & a_{01} \\
    a_{10} & a_{11} \\
    a_{20} & a_{21} \\
    a_{30} & a_{31}
\end{bmatrix}
& \begin{bmatrix}
    b_{00} & b_{01} \\
    b_{10} & b_{11} \\
    b_{20} & b_{21} \\
    b_{30} & b_{31}
\end{bmatrix}
& \begin{bmatrix}
    c_{00} & c_{01} \\
    c_{10} & c_{11} \\
    c_{20} & c_{21} \\
    c_{30} & c_{31}
\end{bmatrix}
\end{align*}
\]

\[ = \]

\[
\begin{bmatrix}
    c_{00} & c_{01} & c_{02} & c_{03} \\
    c_{10} & c_{11} & c_{12} & c_{13} \\
    c_{20} & c_{21} & c_{22} & c_{23} \\
    c_{30} & c_{31} & c_{32} & c_{33}
\end{bmatrix}
\]
$C = A \cdot B$ : Block-column wise distribution

• 1<sup>st</sup> step: each process calculates part of the result elements and stores it in matrix $C$

\[
\begin{bmatrix}
    a_{00} & a_{01} \\
    a_{10} & a_{11} \\
    a_{20} & a_{21} \\
    a_{30} & a_{31}
\end{bmatrix}
\begin{bmatrix}
    a_{02} & a_{03} \\
    a_{12} & a_{13} \\
    a_{22} & a_{23} \\
    a_{32} & a_{33}
\end{bmatrix}
\begin{bmatrix}
    b_{00} & b_{01} & b_{02} & b_{03} \\
    b_{10} & b_{11} & b_{12} & b_{13} \\
    b_{20} & b_{21} & b_{22} & b_{23} \\
    b_{30} & b_{31} & b_{32} & b_{33}
\end{bmatrix}
= \begin{bmatrix}
    a_{00} \cdot b_{00} + a_{01} \cdot b_{10} & a_{00} \cdot b_{01} + a_{01} \cdot b_{11} & a_{02} \cdot b_{22} + a_{03} \cdot b_{32} & a_{02} \cdot b_{23} + a_{03} \cdot b_{33} \\
    a_{10} \cdot b_{00} + a_{11} \cdot b_{10} & a_{10} \cdot b_{01} + a_{11} \cdot b_{11} & a_{12} \cdot b_{22} + a_{13} \cdot b_{32} & a_{12} \cdot b_{23} + a_{13} \cdot b_{33} \\
    a_{20} \cdot b_{00} + a_{21} \cdot b_{10} & a_{20} \cdot b_{01} + a_{21} \cdot b_{11} & a_{22} \cdot b_{22} + a_{23} \cdot b_{32} & a_{22} \cdot b_{23} + a_{23} \cdot b_{33} \\
    a_{30} \cdot b_{00} + a_{31} \cdot b_{10} & a_{30} \cdot b_{01} + a_{31} \cdot b_{11} & a_{32} \cdot b_{22} + a_{33} \cdot b_{32} & a_{32} \cdot b_{23} + a_{33} \cdot b_{33}
\end{bmatrix}
\]

• 2<sup>nd</sup> step: Process 0 and 1 swap there portions of Matrix $A$
  - Matrix $B$ and $C$ unchanged, e.g.

Matrix $A$

<table>
<thead>
<tr>
<th>rank=0</th>
<th>rank=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{02}$</td>
<td>$a_{03}$</td>
</tr>
<tr>
<td>$a_{12}$</td>
<td>$a_{13}$</td>
</tr>
<tr>
<td>$a_{22}$</td>
<td>$a_{23}$</td>
</tr>
<tr>
<td>$a_{32}$</td>
<td>$a_{33}$</td>
</tr>
</tbody>
</table>
\[ C = A \cdot B : \text{Block-column wise distribution} \]

- Finish matrix multiply operation

\[
\begin{bmatrix}
  a_{02} & a_{03} \\
  a_{12} & a_{13} \\
  a_{22} & a_{23} \\
  a_{32} & a_{33}
\end{bmatrix}
\begin{bmatrix}
  a_{00} & a_{01} \\
  a_{10} & a_{11} \\
  a_{20} & a_{21} \\
  a_{30} & a_{31}
\end{bmatrix}
\begin{bmatrix}
  b_{00} & b_{01} & b_{02} & b_{03} \\
  b_{10} & b_{11} & b_{12} & b_{13} \\
  b_{20} & b_{21} & b_{22} & b_{23} \\
  b_{30} & b_{31} & b_{32} & b_{33}
\end{bmatrix}
= \]

\[
\begin{bmatrix}
  a_{02}b_{00} + a_{03}b_{01} + a_{03}b_{02} + a_{03}b_{03} \\
  a_{12}b_{10} + a_{13}b_{11} + a_{13}b_{12} + a_{13}b_{13} \\
  a_{22}b_{20} + a_{23}b_{21} + a_{23}b_{22} + a_{23}b_{23} \\
  a_{32}b_{30} + a_{33}b_{31} + a_{33}b_{32} + a_{33}b_{33}
\end{bmatrix}
\]

- Generalization for \( np \) processes
$C = A \cdot B :$ Block-column wise distribution

- np-1 steps required to give every process access to the entire matrix $A$
  - algorithm does not require a process to hold the entire matrix at any point in time
  - final shift operation required in order for each process to have its original portion of the matrix $A$ back
- Communication between processes often using a ring, e.g.
  - process $x$ sends to $x-1$ and receives from $x+1$
  - special case for process 0 and np-1
  - need to use a temporary buffer if simultaneously sending and receiving the matrix $A$

```c
// nlocalcols: no. of columns held by a process
// nrows: no. of rows of the matrix
// np: number of processes
// rank: rank of this process

sendto = rank-1;
recvfrom = rank+1;
if ( rank == 0 ) sendto = np-1;
if ( rank == np-1 ) recvfrom = 0;

MPI_Isend( A, nrows*nlocalcols, MPI_DOUBLE, sendto, 0, comm, &req[0]);
MPI_Irecv( tempA, nrows, nlocalcols, MPI_DOUBLE, recvfrom, 0, comm, &req[1]);
MPI_Waitall ( req, statuses );

// Copy data from temporary buffer into A
memcpy ( A, tempA, nrows*nlocalcols*sizeof(double));
```
$$C = A \cdot B : \text{Block-column wise distribution}$$

- Mapping of global to local indices
  - required since a C data structure can not start at an arbitrary value, but has to start at index 0
  - need to know from which process we hold the actual data item in order to know which elements of the Matrix B to use
  - mapping will depend of the direction of the ring communication

```c
// nlocalcols: no. of columns held by a process
// nrows: no. of rows of the matrix
// np: number of processes
// rank: rank of this process
for ( it=0; it < np; it++ ) {
    offset = (rank+it)%np * nlocalcols;
    for (i=0; i<nrows; i++) {
        for ( j=0; j<nlocalcols; j++ ) {
            for (k=0; k<nlocalcols; k++) {
                C[i][j] += A[i][k] + B[offset+k][j];
            }
        }
    }
    // Communication as shown on previous slides
}
```
$C = A \cdot B$ : Block-column wise distribution

- Alternative communication pattern for block-column wise distribution:
  - in iteration $it$, process with $rank=it$ broadcasts its portion of the Matrix $A$ to all processes.
  - Mapping of global to local indices a bit simpler
  - Communication costs higher than for the ring communication

$C = A \cdot B$ : Block-row wise distribution

- Similar algorithm as for block-column wise, e.g. 1$^{st}$ step

rank=0

$$
\begin{bmatrix}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33}
\end{bmatrix}
\begin{bmatrix}
  b_{00} & b_{01} & b_{02} & b_{03} \\
  b_{10} & b_{11} & b_{12} & b_{13} \\
  b_{20} & b_{21} & b_{22} & b_{23} \\
  b_{30} & b_{31} & b_{32} & b_{33}
\end{bmatrix} =
$$

\[
\begin{bmatrix}
  a_{00} \cdot b_{00} + a_{01} \cdot b_{00} + a_{02} \cdot b_{01} + a_{03} \cdot b_{01} & a_{00} \cdot b_{02} + a_{01} \cdot b_{02} + a_{02} \cdot b_{03} + a_{03} \cdot b_{03} \\
  a_{10} \cdot b_{10} + a_{11} \cdot b_{10} + a_{12} \cdot b_{11} + a_{13} \cdot b_{11} & a_{10} \cdot b_{12} + a_{11} \cdot b_{12} + a_{12} \cdot b_{13} + a_{13} \cdot b_{13} \\
  a_{20} \cdot b_{20} + a_{21} \cdot b_{20} + a_{22} \cdot b_{21} + a_{23} \cdot b_{21} & a_{20} \cdot b_{22} + a_{21} \cdot b_{22} + a_{22} \cdot b_{23} + a_{23} \cdot b_{23} \\
  a_{30} \cdot b_{30} + a_{31} \cdot b_{30} + a_{32} \cdot b_{31} + a_{33} \cdot b_{31} & a_{30} \cdot b_{32} + a_{31} \cdot b_{32} + a_{32} \cdot b_{33} + a_{33} \cdot b_{33}
\end{bmatrix}
\]

- 2$^{nd}$ step omitted here, only difference to block-column wise distribution is that the matrix $B$ is rotated among the processes
- mapping of local to global indices relevant for Matrix $A$
\[ C = A \cdot B : \text{2-D data distribution} \]

- Both matrices need to be rotated among the processes
  - only processes holding a portion of the same rows of Matrix A need to rotate amongst each other
  - only processes holding a portion of the same columns of Matrix B need to rotate amongst each other

- e.g. for 2\(^{nd}\) step:
  - assuming a 2-D process topology
  - Matrix A is communicated in a ring to the left neighbor
  - Matrix B is communicated in a ring to the upper neighbor
\( C = A \cdot B \) : 2-D data distribution

Cannon’s algorithm for square matrices

- Set up 2-D process topology
- determine nlocalcols and nlocalrows for each process
- initial shift operation such that each process multiplies its local submatrices by \( i \) steps (see next slide)
- for \( i = 0; i < \) number of processes in a row (or column)
  - calculate local part of matrix-matrix multiply operation
  - send local portion of \( A \) to the left neighbor
  - receive next portion of \( A \) from the right neighbor
  - send local portion of \( B \) to the upper neighbor
  - receive next portion of \( B \) from the lower neighbor

Initial assignment of Matrices \( A \) and \( B \)

Initial shift of Matrices \( A \) and \( B \) such that:

- Matrix \( A \) is shifted by \( i \) processes left for processes in the \( i \)-th column of the process topology
- Matrix \( B \) is shifted by \( j \) processes up for processes in the \( j \)-th column of the process topology

\[\begin{array}{cccc}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\
A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\
A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \\
\end{array}\quad\begin{array}{cccc}
B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\
B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\
B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\
B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \\
\end{array}\]