COSC 4397
Parallel Computation

Dense Matrix Operations

Edgar Gabriel
Spring 2010

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 diagonales 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 \( B[x][y] \)
    - 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 \((y,x)\)
\( B = A^T \): One element per process

```c
// Assumes:
// 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];
```

\( B = A^T \): 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

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

- Element \( A[i] \) needs to be sent to process \( i \)
- Element \( B[i] \) will be received from process \( i \)

\[
A \\
B
\]

\[
A \\
T
\]

\[
B \]
$B = A^T$: Column-wise data distribution

```
// 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

```
A
rank = 0, 1, 2
```

```
B
rank = 0, 1, 2
```
$B = A^T$: 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];
  double B[N][N];
  ```
- $A[i][j]$
  - is located on the process with the rank $r = j/N_{local}$
  - has the local indexes $A[i_1][j_1]$ with
    
    $\begin{align*}
    i_1 &= i \\
    j_1 &= j \% N_{local}
    \end{align*}$
  - $B[j][i]$
  - is located on the process with the rank $s = i/N_{local}$
  - has the local indexes $B[j_2][i_2]$ with $j_2 = j$ and
    
    $\begin{align*}
    i_2 &= i \% N_{local}
    \end{align*}$
  ```

---

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

```c

// 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 $N >> np$
  - 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
$B = A^T$: Block column-wise data distribution

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

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

// Now transpose each block
for ( i=0; i<N; i+=N_{\text{local}} ) {
    for ( k=0; k<N_{\text{local}}; k++ ) {
        for ( j=k; j<N_{\text{local}}; j++ ) {
            temp = B[i+k][j];
            B[i+k][j] = B[i+j][k];
            B[i+j][k] = temp;
        }
    }
}
```

$B = A^T$: Block column-wise data distribution
\[ B = A^T : \text{other 1-D data distributions} \]

- 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 \( r, r+np, r+2np, \) etc...
  - advantage:
    - non 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

\[ B = A^T : \text{2-D data distribution} \]

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

- Assumption: using 2-D cartesian communicator
- Algorithm:
  - Determine your rank using \texttt{MPI\_Comm\_rank}
  - Determine your coordinates using \texttt{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 \texttt{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

\[ c = A \cdot b : \text{ block row-wise distribution} \]

- 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 );
```
Parallel Computation

$c = A \cdot b$: 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 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

```
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