Lecture 6: Parallel Matrix Algorithms (part 2)
Column-wise Block-Striped Decomposition

Summary of algorithm for computing $c = Ab$

- Column-wise 1D block partition is used to distribute matrix.
- Let $A = [a_1, a_2, ..., a_n]$, $b = [b_1, b_2, ..., b_n]^T$, and $c = [c_1, c_2, ..., c_n]^T$
- Assume each task $i$ has column $a_i, b_i$ and $c_i$ (Assume a fine-grained decomposition for convenience)
1. Read in matrix stored in row-major manner and distribute by column-wise mapping

2. Each task $i$ compute $b_i a_i$ to result in a vector of partial result.

3. An all-to-all communication is used to transfer partial result: every partial result element $j$ on task $i$ must be transferred to task $j$.

4. At the end of computation, task $i$ only has a single element of the result $c_i$ by adding gathered partial results.
<table>
<thead>
<tr>
<th>Processor 0’s initial computation</th>
<th>Proc 2’s init. comput</th>
<th>Proc 3’s init. comput</th>
<th>Proc 4’s init.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_0 = a_{0,0} b_0$</td>
<td>$a_{0,1} b_1$</td>
<td>$a_{0,2} b_2$</td>
<td>$a_{0,3} b_3$</td>
</tr>
<tr>
<td>$c_1 = a_{1,0} b_0$</td>
<td>$a_{1,1} b_1$</td>
<td>$a_{1,2} b_2$</td>
<td>$a_{1,3} b_3$</td>
</tr>
<tr>
<td>$c_2 = a_{2,0} b_0$</td>
<td>$a_{2,1} b_1$</td>
<td>$a_{2,2} b_2$</td>
<td>$a_{2,3} b_3$</td>
</tr>
<tr>
<td>$c_3 = a_{3,0} b_0$</td>
<td>$a_{3,1} b_1$</td>
<td>$a_{3,2} b_2$</td>
<td>$a_{3,3} b_3$</td>
</tr>
<tr>
<td>$c_4 = a_{4,0} b_0$</td>
<td>$a_{4,1} b_1$</td>
<td>$a_{4,2} b_2$</td>
<td>$a_{4,3} b_3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$a_{4,4} b_4$</td>
</tr>
</tbody>
</table>

Processor 0’s initial computation

Processor 1’s initial computation

Processor 2’s initial computation

Processor 3’s initial computation

Processor 4’s initial computation

Processor 0’s initial computation

Processor 1’s initial computation

Processor 2’s initial computation

Processor 3’s initial computation

Processor 4’s initial computation

Processor 4’s initial computation
After All-to-All Communication

Proc 0

Proc 1

Proc 2

Proc 3

Proc 4

\[ a_{0,0} b_0 \]

\[ a_{1,0} b_0 \]

\[ a_{2,0} b_0 \]

\[ a_{3,0} b_0 \]

\[ a_{4,0} b_0 \]

\[ a_{0,1} b_1 \]

\[ a_{1,1} b_1 \]

\[ a_{2,1} b_1 \]

\[ a_{3,1} b_1 \]

\[ a_{4,1} b_1 \]

\[ a_{0,2} b_2 \]

\[ a_{1,2} b_2 \]

\[ a_{2,2} b_2 \]

\[ a_{3,2} b_2 \]

\[ a_{4,2} b_2 \]

\[ a_{0,3} b_3 \]

\[ a_{1,3} b_3 \]

\[ a_{2,3} b_3 \]

\[ a_{3,3} b_3 \]

\[ a_{4,3} b_3 \]

\[ a_{4,4} b_4 \]

\[ a_{1,4} b_4 \]

\[ a_{2,4} b_4 \]

\[ b_{3,4} b_4 \]

\[ a_{4,4} b_4 \]
Reading a Column-wise Block-Striped Matrix

read_col_striped_matrix()

- Read from a file a matrix stored in row-major order and distribute it among processes in column-wise fashion.
- Each row of matrix must be scattered among all of processes.

read_col_striped_matrix()
{
    ...  
    // figure out how a row of the matrix should be distributed
    create_mixed_xfer_arrays(id,p, *n, &send_count, &send_disp);
    // go through each row of the matrix
    for(i = 0; i < *m; i++)
    {
        if(id == (p-1)) fread(buffer,datum_size, *n, infleptr);
        MPI_Scatterv(...);
    }
}
• **int MPI_Scatterv( void *sendbuf, int *sendcnts, int *displs, MPI_Datatype sendtype, void *recvbuf, int recvnt, MPI_Datatype recvtype, int root, MPI_Comm comm)**
  
  – MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying count of data to be sent to each process.
  
  – *sendbuf*: address of send buffer
  
  – *sendcnts*: an integer array specifying the number of elements to send to each processor
  
  – *displs*: an integer array. Entry i specifies the displacement (relative to *sendbuf* from which to take the outgoing data to process i

http://www.mpi-forum.org/docs/mpi-11-html/node72.html
Printing a Column-wise Block-Striped Matrix

print_col_striped_matrix()

- A single process print all values
- To print a single row, the process responsible for printing must gather together the elements of that row from entire set of processes

```c
print_col_striped_matrix()
{
  ...
  create_mixed_xfer_arrays(id, p, n, &rec_count, &rec_disp);
  // go through rows
  for(i =0; i < m; i++)
  {
    MPI_Gatherv(a[i], BLOCK_SIZE(id,p,n), dtype, buffer,
                rec_count, rec_disp, dtype, 0, comm);
    ....
  }
}
```
• int MPI_Gatherv( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm )
  – Gathers into specified locations from all processes in a group.
  – sendbuf: address of send buffer
  – sendcnt: the number of elements in send buffer
  – recvbuf: address of receive buffer (choice, significant only at root)
  – recvcounts: integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
  – displs: integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)
Distributing Partial Results

- \( c_i = b_0 a_{i,0} + b_1 a_{i,1} + b_2 a_{i,2} + \cdots + b_n a_{i,n} \)

- Each process need to distribute \( n - 1 \) terms to other processes and gather \( n - 1 \) terms from them (assume fine-grained decomposition).
  - MPI_Alltoallv() is used to do this all-to-all exchange

![Diagram of MPI_Alltoallv()](image)

**Figure 8.13** Function MPI_Alltoallv allows every MPI process to gather data items from all the processes in the communicator. The simpler function MPI_Alltoall should be used in the case where all of the groups of data items being transferred from one process to another have the same number of elements.
int MPI_Alltoallv( void *sendbuf, int *sendcnts, int *sdispls, MPI_Datatype sendtype, void *recvbuf, int *rcvcnts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm );

• *sendbuf*: starting address of send buffer (choice)
• *sendcounts*: integer array equal to the group size specifying the number of elements to send to each processor
• *sdispls*: integer array (of length group size). Entry $j$ specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process $j$
• *recvbuf*: address of receive buffer (choice)
• *rcvcnts*: integer array equal to the group size specifying the maximum number of elements that can be received from each processor
• *rdispls*: integer array (of length group size). Entry $i$ specifies the displacement (relative to recvbuf at which to place the incoming data from process $i$
Send of MPI\_Alltoallv()

Each node in parallel community has

<table>
<thead>
<tr>
<th>proc 0</th>
<th>proc 1</th>
<th>proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 A</td>
<td>0 H</td>
<td>0 O</td>
</tr>
<tr>
<td>1 B</td>
<td>1 I</td>
<td>1 P</td>
</tr>
<tr>
<td>2 C</td>
<td>2 J</td>
<td>2 Q</td>
</tr>
<tr>
<td>3 D</td>
<td>3 K</td>
<td>3 R</td>
</tr>
<tr>
<td>4 E</td>
<td>4 L</td>
<td>4 S</td>
</tr>
<tr>
<td>5 F</td>
<td>5 M</td>
<td>5 T</td>
</tr>
<tr>
<td>6 G</td>
<td>6 N</td>
<td>6 U</td>
</tr>
</tbody>
</table>

send buffer

send count array

send displacement array
Process 0 Sends to Process 0

Send to receive buffer of proc with same rank as index

This chunk of send buffer goes to receive buffer of proc 0

Index
Proc 0 send buffer

Sendcount Array

Sdispl Array
Process 0 Sends to Process 1

0
1
2
3
4
5
6

A
B
C
D
E
F
G

Index
Proc 0 send buffer

send to receive buffer of proc 1

index
sendcount Array

0
1
2

0
2
2

0
2
5

sdispl Array
## Process 0 Sends to Process 2

The diagram illustrates the process of sending data from Process 0 to Process 2. The left side shows the index and the send buffer of Process 0, while the right side displays the sendcount array and the sdispl array.

### Process 0 Send Buffer

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
</tr>
</tbody>
</table>

### Sendcount Array

- 0: 2
- 1: 3
- 2: 2

### Sdispl Array

- 0: 2
- 1: 5

The data `A` is sent to the receive buffer of Process 2 at index 2.
Receive of MPI\_Alltoallv()

<table>
<thead>
<tr>
<th>Buffer</th>
<th>proc 0</th>
<th>proc 1</th>
<th>proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>
Parallel Run Time Analysis (Column-wise)

• Assume that the # of processes $p$ is less than $n$
• Assume that we run the program on a parallel machine adopting hypercube interconnection network (Table 4.1 lists communication times of various communication schemes)

1. Each process is responsible for $n/p$ columns of matrix. The complexity of the dot production portion of the parallel algorithm is $\Theta(n^2/p)$

2. After all-to-all personalized communication, each processor sums the partial vectors. There are $p$ partial vectors, each of size $n/p$. The complexity of the summation is $\Theta(n)$.

3. Parallel communication time for all-to-all personalized broadcast communication:
   – Each process needs to send $p$ messages of size $n/p$ each to all processes.
     \[ t_{comm} = (t_s + t_w \left(\frac{n}{p}\right))(p - 1) \]  
     Assume $p$ is large, then
     \[ t_{comm} = t_s(p - 1) + t_w n. \]

• The parallel run time: \[ T_p = \frac{n^2}{p} + n + t_s(p - 1) + t_w n \]
2D Block Decomposition

Summary of algorithm for computing $y = Ab$

• 2D block partition is used to distribute matrix.
• Let $A = [a_{ij}]$, $b = [b_1, b_2, ..., b_n]^T$, and $y = [y_1, y_2, ..., y_n]^T$
• Assume each task is responsible for computing $d_{ij} = a_{ij}b_j$ (assume a fine-grained decomposition for convenience of analysis).
• Then $y_i = \sum_{j=0}^{n-1} d_{ij}$: for each row $i$, we add all the $d_{ij}$ to produce the $ith$ element of $y$. 

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_4$</td>
<td>$P_5$</td>
<td>$P_6$</td>
<td>$P_7$</td>
</tr>
<tr>
<td>$P_8$</td>
<td>$P_9$</td>
<td>$P_{10}$</td>
<td>$P_{11}$</td>
</tr>
<tr>
<td>$P_{12}$</td>
<td>$P_{13}$</td>
<td>$P_{14}$</td>
<td>$P_{15}$</td>
</tr>
</tbody>
</table>
1. Read in matrix stored in row-major manner and distribute by 2D block mapping. Also distribute \( b \) so that each task has the correct portion of \( b \).

2. Each task computes a matrix-vector multiplication using its portion of \( A \) and \( b \).

3. Tasks in each row of the task grid perform a sum-reduction on their portion of \( y \).

4. After the sum-reduction, \( y \) is distributed by blocks among the tasks in the first column of the task grid.
Distributing $b$

- Initially, $b$ is divided among tasks in the first column of the task grid.
- Step 1:
  - If $p$ square
    - First column/first row processes send/receive portions of $b$
  - If $p$ not square
    - Gather $b$ on process 0, 0
    - Process 0, 0 broadcasts to first row processes
- Step 2: First row processes scatter $b$ within columns
When $p$ is a square number

(a)

When $p$ is not a square number

(b)