Collectives

• Using combined information from independent nodes
Examples

• How would you realize the following scenarios with MPI collectives?
  – Let each process compute a random number. You want to print the maximum of these numbers to your screen.
  – Each process computes a random number again. Now you want to scale these numbers by their maximum.
  – Let each process compute a random number. You want to print on what processor the maximum value is computed.
Commands Used

- MPI_Bcast, MPI_Reduce, MPI_Gather, MPI_Scatter
- MPI_All... variants, MPI_....v variants
- MPI_Barrier, MPI_Alltoall, MPI_Scan
Allreduce

- `int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

- **Semantics:**
  - IN `sendbuf`: starting address of send buffer (choice)
  - OUT `recvbuf`: starting address of receive buffer (choice)
  - IN `count`: number of elements in send buffer (non-negative integer)
  - IN `datatype`: data type of elements of send buffer (handle)
  - IN `op`: operation (handle)
  - IN `comm`: communicator (handle)
Example 1

- Each node should generate a single random number
- Use MPI_Allreduce to sum all the numbers then calculate the average
  - Divide by RAND_MAX to normalize between 0 and 1
  - Should be approx 0.5
- The “operation” is MPI_SUM
- The “datatype” is MPI_FLOAT
Example 2

- If one has two (large) vectors $x$ and $y$, such that each processor stores a “block” of each, compute the inner product of the two vectors

- Recall, $\langle x, y \rangle$ (inner product) is:

  $$ x \cdot y = \sum_{i=1}^{n} x_i \cdot y_i $$

- Method: Do the “local” inner product and Allreduce with the MPI_SUM op
Some necessities of C

- No more “new” command for arrays :(
- Instead, we call malloc directly!
  - `void* malloc(size_t)`
  - Returns an address
  - Recall void ptrs can be cast to whatever they need to be (but be careful!)
- Typical strategy: `malloc(num * sizeof(type))`
- Note: collectives are **blocking**!
  - Have to wait if some processes aren’t there yet
Example 3

- All processes generate 500k random doubles between 0 and 1
- Calculate the average, but use MPI_IN_PLACE, which overwrites the input data with the result
  - Saves half the memory!
  - Only need to calculate the average on one node
  - Can be cast and stored as a variable
More MPI Operators

<table>
<thead>
<tr>
<th>MPI type</th>
<th>meaning</th>
<th>applies to</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>maximum</td>
<td>integer, floating point</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>minimum</td>
<td></td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
<td>integer, floating point, complex, multilanguage types</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>product</td>
<td></td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>logical and</td>
<td>C integer, logical</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>logical xor</td>
<td></td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bitwise and</td>
<td>integer, byte, multilanguage types</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>bitwise xor</td>
<td></td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max value and location</td>
<td>MPI_DOUBLE_INT and such</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min value and location</td>
<td></td>
</tr>
</tbody>
</table>

- Can also create your own!
  - `MPI_Op_create( MPI_User_function * func, int commute, MPI_Op * op);`
Rooted Collectives

- We can designate one process with “root” status, giving higher priority and extra responsibility.
- Usage example: instead of using Allreduce, we can reduce to a single root node instead:
  - Fewer communications
  - Less memory overhead
    - Non-roots can use null receive buffer
  - Need to broadcast results
MPI_Reduce

- int MPI_Reduce(
  const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
  MPI_Op op, int root, MPI_Comm comm)
- Note that root is designated by its rank among the communicator
- Can also be done in place
Example 4

- Each process generates its own random number
- Reduce to a root, process 0, which reports the max of all the numbers
- Include output from all processes to check correctness
Broadcasting

- **MP_Bcast**(void* buffer, int count, MPI_Datatype t, int root, MPI_Comm c)
- Keep in mind buffer is an address
  - So will be &value for an int, etc.
  - But will be arrName for an array
- Result is that all non-roots get a copy of the root node’s “buffer” variable
  - Space must be pre-allocated (maybe need a broadcast beforehand)!
Example: Matrices

- Recall that for matrices $A$ and $B$ of sizes $n$-by-$k$ and $k$-by-$m$, respectively, their product, $A*B$ is defined as the $n$-by-$m$ matrix $C$ such that

$$C_{i,j} = \sum_{\ell=1}^{k} A_{i,\ell} \cdot B_{\ell,j}$$

That is, the $(i,j)$ entry of $C$ is the inner (or dot) product of the $i$th row of $A$ with the $j$th column of $B$. 
Matrix Multiplication

• Question: how to distribute the matrix product? Options? What collectives are needed?

• Basic task: given a matrix A, find another matrix, called $A^{-1}$, so that $A \times A^{-1}$ is a square matrix with 1’s on the main diagonal and 0’s everywhere else (i.e. the identity matrix)
Example from Numerical Linear Algebra

Exercise 3.6. The Gauss-Jordan algorithm for solving a linear system with a matrix $A$ (or computing its inverse) runs as follows:

for pivot $k = 1, \ldots, n$

let the vector of scalings $\ell_i^{(k)} = A_{ik}/A_{kk}$

for row $r \neq k$

for column $c = 1, \ldots, n$

$A_{rc} \leftarrow A_{rc} - \ell_r^{(k)} A_{rc}$

where we ignore the update of the righthand side, or the formation of the inverse. Let a matrix be distributed with each process storing one column. Implement the Gauss-Jordan algorithm as a series of broadcasts: in iteration $k$ process $k$ computes and broadcasts the scaling vector $\{\ell_i^{(k)}\}_i$. Replicate the right-hand side on all processors.