Introduction to MPI Programming – Part 2
Outline

- Collective communication
- Derived data types
Collective Communication

- Collective communications involves all processes in a communicator
  - One to all, all to one and all to all
- Three types of collective communications
  - Data movement
  - Collective computation
  - Synchronization
Collective vs. Point-to-point

• More concise program
  – One collective operation can replace multiple point-to-point operations

• Optimized collective communications usually are faster than the corresponding point-to-point communications
Data Movement: Broadcast

- Broadcast copies data from the memory of one processor to that of other processors
  - One to all operation

- Syntax: `MPI_Bcast (send_buffer, send_count, send_type, rank, comm)`
Data Movement: Gather

- Gather copies data from each process to one process, where it is stored in rank order
  - One to all operation
- **Syntax:** `MPI_GATHER ( send_buffer, send_count, send_type, recv_buffer, recv_count, recv_type, recv_rank, comm )`
Example: MPI_Gather

```fortran
... integer, allocatable :: array(:,), array_r(:)
! Initialize MPI
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world, nprocs, ierr)
call mpi_comm_rank(mpi_comm_world, myid, ierr)
! Initialize the array
allocate(array(2), array_r(2*nprocs))
array(0) = 2*myid
array(1) = 2*myid+1
! Gather data at the root process
call mpi_gather(array, 2, mpi_integer, &
    array_r, 2, mpi_integer, &
    0, mpi_comm_world)
if (myid.eq.0) then
    write(*,*) "The content of the array_r:"
    write(*,*) array_r
```
Example: MPI_Gather

```fortran
... integer, allocatable :: array (:), array_r (:)
! Initialize MPI
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world, nprocs, ierr)
call mpi_comm_rank(mpi_comm_world, myid, ierr)
! Initialize the array
allocate(array(2), array_r(2*nprocs))
array(0) = 2*myid
array(1) = 2*myid + 1
! Gather data at the root process
call mpi_gather(array, 2, mpi_integer, &
array_r, 2, mpi_integer, &
0, mpi_comm_world)
if (myid.eq.0) then
  write(*,*) "The content of the array_r:"
  write(*,*) array_r
end if
```

[lyan1@qb563 ex]$ mpirun -np 4 ./a.out
The content of the array:

0 1 2 3 4 5 6 7

array_r:
0 1 2 3 4 5

if (myid.eq.0) then
  write(*,*) "The content of the array_r:"
  write(*,*) array_r
```
**Collective Computation: Reduction**

- MPI reduction collects data from each process, reduces them to a single value, and stores it in the memory of one process
  - One to all operation
- **Syntax:** `MPI_Reduce( send_buffer, recv_buffer, count, data_type, reduction_operation, rank_of_receiving_process, communicator )`
Reduction Operation

- Summation and production
- Maximum and minimum
- Max and min location
- Logical
- Bitwise
- User defined
Collective Computation: Allreduce

- MPI allreduce collects data from each process, reduces them to a single value, and store it in the memory of EVERY process
  - All to all operation
- Syntax: `MPI_Allreduce( send_buffer, recv_buffer, count, data_type, reduction_operation, communicator )`
Synchronization

• MPI_Barrier (Communicator)
  – Blocks processes in a group until all processes have reached the same synchronization point
  – Synchronization is collective since all processes are involved
  – Could cause significant overhead, so do NOT use it unless absolutely necessary
Other Collective Communications

Source: Practical MPI Programming, IBM Redbook
Exercise 3a: Find Global Maximum

• Goal: repeat Exercise 2a with appropriate collective communication function(s)
Exercise 3b: Matrix Multiplication version 3

• Goal: Replace the part in version 2 that sends the result to the root process with appropriate collective operation(s)
Exercise 3c: Laplace Solver version 2

- Goal: Replace the part in version 1 that finds the global maximum convergence and distributes it to all processes with appropriate collective operation(s)
## Basic Data Types

<table>
<thead>
<tr>
<th>MPI Data Type</th>
<th>C Data Type</th>
<th>MPI Data Type</th>
<th>Fortran Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>Signed char</td>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>Signed short int</td>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>Signed int</td>
<td>MPI_REAL8</td>
<td>REAL*8</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>Signed long int</td>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>Unsigned char</td>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>Unsigned short</td>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>Unsigned int</td>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>Unsigned long int</td>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>Float</td>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
Why Derived Data Types?

• The communication functions we have seen so far deal with contiguous data of the same type

• What if the data to be transferred is
  – Not contiguous?
  – Not of the same type?
Why Derived Data Types

Process 0

Process 1

Solution 1
Call `MPI_SEND/MPI_RECEIVE` multiple times (once for each contiguous segment)
Why Derived Data Types

Solution 2
Process 0: copy data into contiguous buffer, then call MPI_SEND
Process 1: call MPI_RECEIVE, then copy data to destination
Why Derived Data Types

Solution 3
Define a new data type to describe the data to be transferred, then call MPI_SEND/MPI_RECV
- Most efficient
Derived Data Type: Contiguous

• Allows replication of one data type into contiguous locations

• Syntax: \texttt{MPI\_Type\_Contiguous}(\texttt{count, old\_type, new\_type})

• The new data type must be commited before being used for communication:
  – \texttt{MPI\_Type\_Commit(new\_type)}
Derived Data Type: Vector

• Allows replication of a data type into locations that consist of equally spaced blocks

• Syntax: MPI_Type_Vector(count, block size, stride, old_type, new_type)

```plaintext
Call mpi_type_vector(3,2,4,mpi_real8,my_vector_type,ierr)
Call mpi_type_commit(my_vector_type,ierr)
```

- Block size = 2
- Stride = 4
Data Ordering: C vs. Fortran

C

Fortran
Example: Broadcasting Submatrix

```
!Number of blocks
nbblocks=6
!Block length
blocklen=6
!Stride
stride=8

!Define the new data type
CALL MPI_TYPE_VECTOR(NBLCK,BLCKLEN,STRD, &
    MPI_INTEGER,submat_type,IERR)
CALL MPI_TYPE_COMMIT(submat_type,IERR)

!Call broadcast
CALL MPI_BCAST(AMAT(2,2),1,submat_type,0, &
    MPI_COMM_WORLD,IERR)
```
Derived Data Type: HVector

• The same with MPI_Type_Vector, except that the unit of the stride is byte instead of old_type
  – More flexible than the vector type
  – We can use
    MPI_Type_Extent(datatype,extent) to decide the extent (in bytes) of an MPI data type
Nested Derived Data Type

• New data types can be created out of user-defined data types

```
call mpi_type_extent(mpi_integer, & size_of_int, ierr)
call mpi_type_vector(2,1,2,mpi_integer, & column_type,ierr)
call mpi_type_hvector(4,1,6*size_of_int, & column_type,new_type,ierr)
```
Derived Data Type: Indexed

- Allows replication of a data type into locations that consist of unequally spaced blocks with varying length
- Syntax: MPI_Type_indexed (count, blocklens[], offsets[], old_type, new_type)
  - blocklens and offsets are array of size count that specify the length and displacement of each block, respectively

Count=4
blocklens=[4,1,4,2]
Offsets=[0,6,12,18]
Derived Data Type: Struct

- Most general data type constructor
- Allows a new data type that represents arrays of types, each of which has a different block length, displacement (in bytes) and type

- Syntax: `MPI_Type_struct (count, blocklens[], offsets[], old_types[], new_type)`
Exercise 4a: Matrix Transposition

• Goal: write a MPI program that transposes a matrix in parallel
  – 2-d process grid
  – Each process initializes its own sub-matrix
  – Do we really need the local transposition?
Exercise 4b: Matrix Multiplication

Version 4

• Goal: change the matrix multiplication to two-dimensional decomposition
  – Arrange the processes into a 2-d grid
  – Each process should only owns a sub-matrix of A, B and C
  – Assemble the matrix C at the root process using the partial result from each process
Exercise 4c: Laplace Solver Version 3

• Goal: change our Laplace solver to two-dimensional decomposition

• Hints
  – Change how the size of sub-domain is calculated
  – Change how the boundary condition is set
  – Data transfer along two more directions
    • Those data are not contiguous
      – Fortran: use derived data types
      – C: need to pack it into a contiguous buffer