MPI Collective Operations

• An operation over an entire communicator
• Must be called by every member of the communicator
• Three classes of collective operations:
  – Synchronization (MPI_Barrier)
  – Data movement
  – Collective computation
Collective Patterns (Gropp)
Collective Computation Patterns (Gropp)
Collective Routines (Gropp)

• Many routines:
  Allgather          Allgatherv          Allreduce
  Alltoall           Alltoally           Bcast
  Gather             Gatherv             Reduce
  ReduceScatter      Scan                Scatter
  Scatterv

• ‘All’ versions deliver results to all participating processes.

• ‘V’ versions (e.g., Scatterv) allow chunks to have different sizes.

• Allreduce, Reduce, ReduceScatter, and Scan take both built-in and user-defined combination functions.
Example: Jacobi Iteration

For all $1 \leq i, j \leq n$, do until converged ...

$$u_{ij}^{(\text{new})} \leftarrow 0.25 \times (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})$$

(1D Decomp)
Jacobi: 1D Data Decomposition

• Assign responsibility for $n/p$ rows of the grid to each process.
• Each process holds copies ("ghost points") of one row of old data from each neighboring process.
• Algorithm for each process:

  Repeat until happy:
  exchange data
  do updates
Jacobi: Exchanging data with Neighbors

- Each processor exchanges a row of data with neighbor to the north & south, except possibly process 0 and process nprocs-1.
- Suppose each process posts recv’s before send’s
  - With periodic boundary conditions, program will deadlock
  - With Dirichlet boundary conditions, program is correct but likely to experience “serialized” communication, i.e., only one message sent at a time.
- Suppose each process posts send’s before recv’s
  - For large problem size, MPI implementation may require the corresponding recv to be posted before the send completes (due to insufficient buffer space).
  - So with periodic boundary conditions we could again have deadlock
  - And with Dirichlet boundary conditions, could have serializing.
Jacobi: Exchanging data with Neighbors

- One alternative is to organize communication so that even processes first recv and then send, while odd processes first send and then recv.
- An even better alternative is to use the MPI_Sendrecv function which exists to handle this “exchange of data” dance without all the potential buffering problems.
- Another alternative: non-blocking (“immediate”) sends and recv’s.
MPI Non-Blocking Message Passing

- MPI_Isend initiates send, returning immediately with a request handle.
- MPI_Irecv posts a receive and returns immediately with a request handle.
- MPI_Wait blocks until a given message passing event, specified by handle, is complete.
- MPI_Test can be used to check a handle for completion without blocking.
## MPI Non-Blocking Send

**MPI_Isend**

```c
int MPI_Isend (void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

### IN
- `buf`: initial address of send buffer (choice)
- `count`: number of entries to send (integer)
- `datatype`: datatype of each entry (handle)
- `dest`: rank of destination (integer)
- `tag`: message tag (integer)
- `comm`: communicator (handle)

### OUT
- `request`: request handle (handle)

### Example C Code

```c
int MPI_Isend (void *buf, int count, MPI_Datatype datatype, 
                int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

### Example Fortran Code

```fortran
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERR)
```

- `<type> BUF(*)`
- `INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERR`
MPI Non-Blocking Recv

MPI_Irecv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERR)

<int> BUF(*)

INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERR
Function MPI_Wait

MPI_WAIT(request, status)

 inout request request handle (handle)
 out status status object (Status)

int MPI_Wait (MPI_Request *request, MPI_Status *status)

MPI_WAIT(REQUEST, STATUS, IERR)

INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERR
Jacobi with Asynchronous Communication

- With non-blocking sends/recvs, can avoid any deadlocks or slowdowns due to buffer management.
- With some code modification, can improve performance by overlapping communication and computation:

**Old Algorithm:**  
exchange data  
do updates

**New Algorithm:**  
initiate exchange  
update strictly interior grid points  
complete exchange  
update boundary points