



# Advanced MPI-1 and MPI-2

Philip Blood

John Urbanic

Pittsburgh Supercomputing Center

# Overview

- In the MPI Basics talk we have only touched upon a few of the 120+ MPI-1 routines and didn't use any MPI-2.
- We won't cover everything here, but we'll discuss some general areas of functionality provided by advanced MPI-1 and MPI-2 routines.

# MPI-I Advanced Routines

- Communicators
- User Defined Data Types
- Non-Blocking Messaging

# Communicators

**Convenience and Safeguard:** Allow communications occurring in different contexts, or amongst distinct groups of processors to be insulated from each other.

How to create communicators:

1. Identify an existing group (default group of all processes associated with `MPI_COMM_WORLD`)
2. Create a new group by including or excluding members of the existing group.
3. Create a new communicator using the handle returned by the newly created group.

# Communicators

```
main(int argc, char **argv) {
    int me, count, count2;
    void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
    MPI_Group MPI_GROUP_WORLD, grpem;
    MPI_Comm commslave;
    static int ranks[] = {0};

    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD,
    &MPI_GROUP_WORLD);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grpem);
    MPI_Comm_create(MPI_COMM_WORLD, grpem,
    &commslave);
    if(me != 0){ /* compute on slave */
        MPI_Reduce(send_buf,recv_buf,count, MPI_INT,
    MPI_SUM, 1, commslave);
    }
    /* zero falls through immediately to this reduce, others do
    later... */

    MPI_Reduce(send_buf2, recv_buf2, count2, MPI_INT,
    MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Comm_free(&commslave);
    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Group_free(&grpem);
    MPI_Finalize();
}
```

1. Handle obtained from group associated with `MPI_COMM_WORLD` (all processors)
  2. Create new group of slave processors by excluding the master process (process 0)
  3. Create a new communicator amongst the slave processes using the handle of the newly created group.
- Now you can do collective communication that excludes the master process.
  - And then do a separate collective communication that includes all processes
  - Communicators and groups can be freed when no longer needed.

# User Defined Data Types

- Organizationally useful
- Necessity when dealing with certain data structures

For example:

- A column type (or row type for Fortran) with built-in stride
- Array sub-blocks
- More esoteric structures

# User Defined Data Types

## Example - Transpose a matrix

```
REAL a(100,100), b(100,100)
```

```
INTEGER row, xpose, sizeofreal, myrank, ierr
```

```
INTEGER status(MPI_STATUS_SIZE)
```

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
```

```
CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)
```

**C** create datatype for one row of 100x100 matrix

**C** (100 blocks, 1 real/block, 100 elements between blocks)

```
CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
```

**C** create datatype for matrix in row-major order

**C** (100 blocks, 1 row/block, sizeofreal bytes between blocks)

```
CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)
```

**C** must commit new datatype before using it in a communication

```
CALL MPI_TYPE_COMMIT( xpose, ierr)
```

**C** send matrix in row-major order and receive in column major order

```
CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,  
                  MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

# Non-Blocking Messaging

- Can make sending massive numbers of messages, or large messages, possible without deadlock or buffer overruns.
- **Essential** for scaling applications to large numbers of processors

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

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

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

```
MPI_Irecv (B ... request[0])  
MPI_Isend (A ... request[1])  
MPI_WAIT (request[0])  
COMPUTE LOOP1 (uses B)  
MPI_WAIT (request[1])  
COMPUTE LOOP2 (modifies A)
```

# Contents of MPI-2

- One-sided communication (put / get)
- Dynamic process management
- Parallel I/O (MPI-IO)
- Miscellaneous
  - Extended collective communication operations
  - C++ interface
  - limited F90 support
  - language interoperability
  - Thread support

# Not Really Implemented (Yet)

- Few full MPI-2 implementations
  - Fujitsu
  - Hitachi
  - NEC

*Some contention even here.*

- Most just have pieces (Put/Get, MPIO)

# One-sided Communication (Remote Memory Access)

Define a 'memory window', which can be accessed by remote tasks, then use:

```
MPI_Put( origin_addr, origin_count,  
         origin_datatype, target_addr,  
         target_count, target_datatype, window )
```

```
MPI_Get( ... )
```

```
MPI_Accumulate( ..., op, ... )
```

op is as in MPI\_Reduce, (MPI\_SUM, MPI\_PROD, MPI\_MIN, etc.) but no user-defined operations are allowed.

# One-sided Communication

These can be very useful and efficient routines (especially with hardware support for RMA, like on SMPs).

They are similar to the shmem message Passing interface offered on several high Performance platforms. You may even “optimize” an MPI-I code to use these in a fairly painless manner.

# Dynamic Process Management

`MPI_Comm_spawn` creates a new group of tasks and returns an intercommunicator:

`MPI_Comm_spawn(command, argv, numprocs, info, root, comm, intercomm, errcodes)`

- Tries to start *numprocs* processes running *command*, passing them command-line arguments *argv*.
- The operation is collective over *comm*.
- Spawnees are in remote group of *intercomm*.
- Errors are reported on a per-process basis in *errcodes*.
- *info* used to optionally specify hostname, archname, wdir, path, file.

# Dynamic Process Management

This can seem tempting, but as most MPP's require you to run on a specific number of PE's, adding tasks can often cause load balance problems – if it is even permitted. For optimized scientific codes, this is rarely a useful approach.

One possible use: multiscale, multiphysics applications

# MPI-IO

- Portable interface to parallel I/O, part of MPI 2 standard
- Many tasks can access the same file in parallel,
- Each task may define its own view of a file
- I/O in MPI can be considered as Unix I/O plus (lots of) other stuff.
- Basic operations: `MPI_File_{open, close, read, write, seek}`
- Parameters to these operations match Unix, aiding straightforward port from Unix I/O to MPI I/O.
- *However, to get performance and portability, more advanced features MUST be used.*