Introduction to Message Passing Interface (MPI) Programming

2024 NSF CyberTraining Workshop

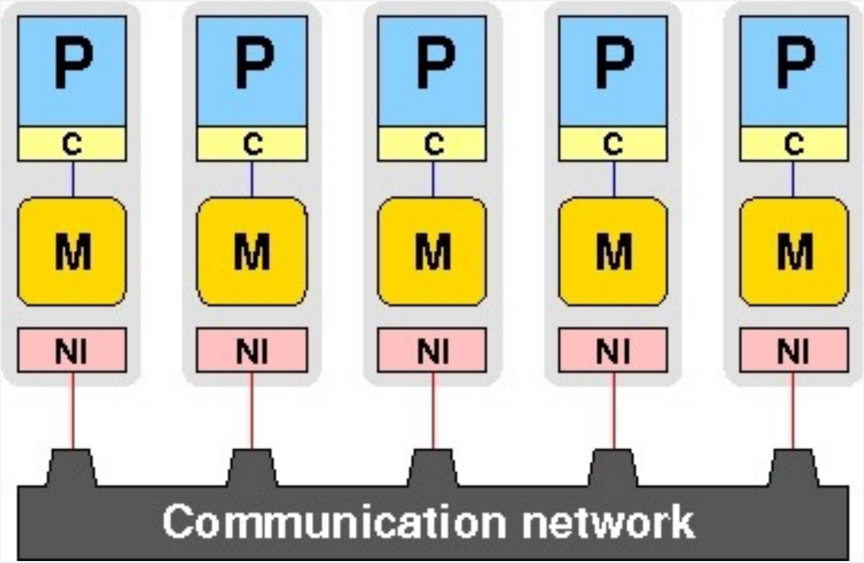
Jan. 8, 2024 – Jan. 19, 2024

Clarkson University

Message Passing Paradigm

Distributed memory architecture:

Each process(or) can only access its dedicated address space.
No global shared address space
Data exchange and communication between processes is done by explicitly passing messages through a communication network



Message passing library:
• Should be flexible, efficient and portable
• Hide communication hardware and software layers from application programmer

Message Passing Paradigm

- Widely accepted standard in HPC / numerical simulation: Message Passing Interface (MPI)
- Process based approach: All variables are local!
- Data exchange between processes(a.k.a. tasks): Send/receive messages via MPI library calls
- This is usually the most tedious but also the most flexible way of parallelization
- MPI is standard for explicit message passing today.

A Modern MPI Standard

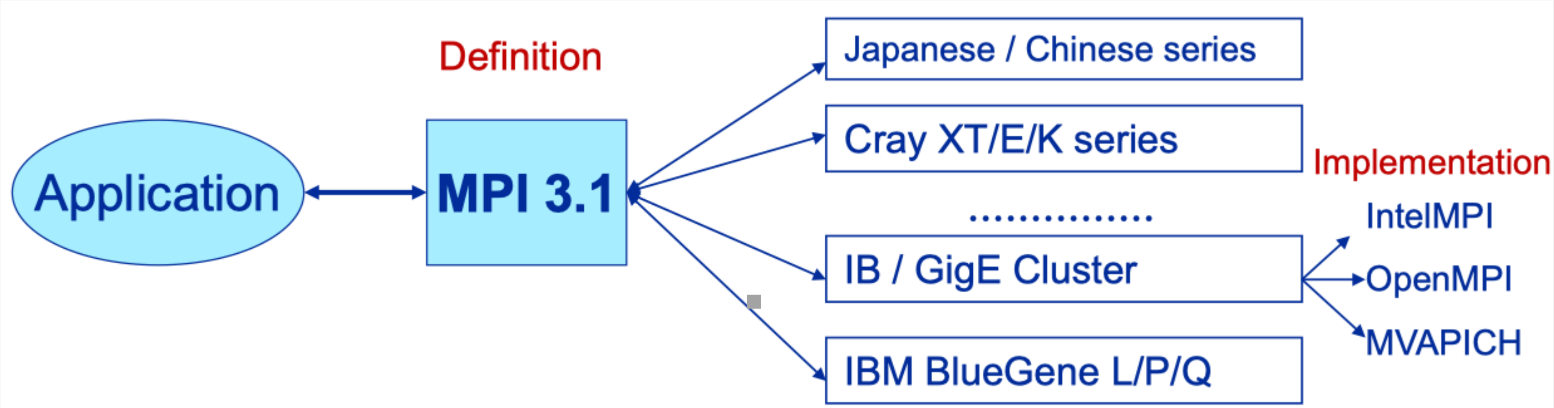
- MPI forum – defines MPI standard / library subroutine interfaces
- Beginning: April 1992 – Before: vendor specific libraries
- Latest standard: MPI 3.1 (2015) – MPI 4.0 under development

- Members (approx. 60) of MPI standard forum
  - Application programmers
  - Research institutes & Computing centres
  - Manufacturers of supercomputers & software designers

- Successful free implementation: MPICH, OpenMPI + many others + vendor libraries (Intel, IBM, CRAY)
- All documents and more pointers available at: www.forum.org/
- MPI defines more than 500 subroutines – typically only 10-30 are used

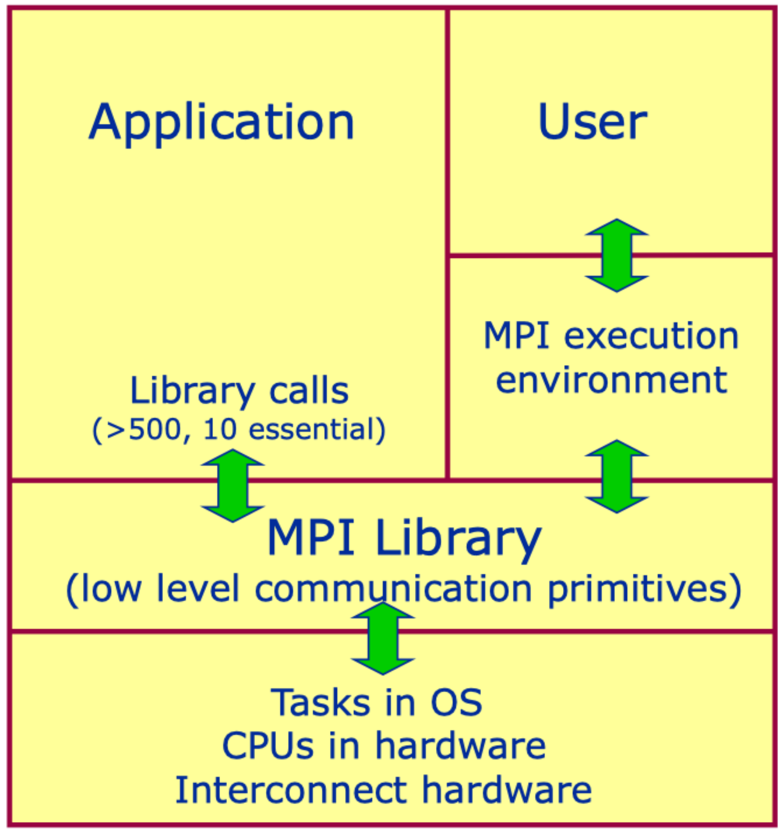
Goals and Scope

- PORTABILITY: Architecture and hardware independent code



- FORTRAN, C & C++ Interface
- Provides ´well-defined´and ´safe´data transfer
- Enables development of parallel libraries
- Support heterogeneous environment (e.g. clusters with heterogeneous compute nodes)

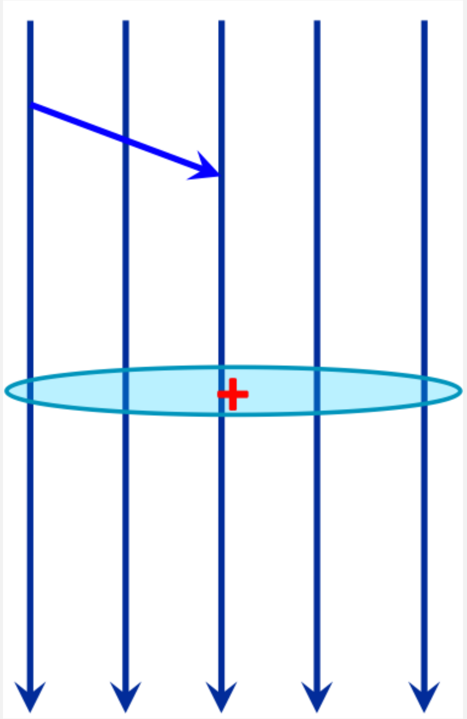
Software Architecture



- Operating system view:
  - parallel work done by tasks/processes
- Programmer's view: Library routines for
  - coordination
  - communication
  - synchronization
- User's view: MPI execution environment provides
  - resource allocation (with the support from LSF, SLURM, openPBS, etc.)

Parallel Execution

- Tasks run throughout program execution: All variables are local



- Startup phase: MPI
  - launches tasks
  - establishes communication context (communicator) among all tasks
- MPI Point-to-point data transfer:
  - usually between pairs of tasks
  - usually coordinated
  - may be blocking or non-blocking
- MPI Collective communication:
  - between all tasks or a subgroup of tasks
  - barrier, reductions, scatter/gather
- Shutdown by MPI

MPI Functions

- MPI consists of hundreds of functions
- Most users will only use a handful small groups of them
- All functions prefixed with MPI_
- C functions return integer error
  - MPI_SUCCESS if no error

Note: MPI functions will be illustrated in C adapted from ORNL training course

More Terminology

- Communicator
  - An object that represents a group of processes (a.k.a. tasks) than can communicate with each other
  - MPI_Comm type
  - Predefined communicator: MPI_COMM_WORLD
- Rank
  - Within a communicator each process is given a unique integer ID
  - Ranks start at 0 and are incremented contiguously
- Size
  - The total number of ranks in a communicator

Basic Functions

```
int MPI_Init( int *argc, char ***argv )
```

- `argc`
  - Pointer to the number of arguments
- `argv`
  - Pointer to argument vector
- Initializes MPI environment
- Must be called before any other MPI call

Basic Functions

```
int MPI_Finalize( void )
```

- Cleans up MPI environment
- Calling MPI functions after MPI_Finalize is undefined

Basic Functions

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- `comm` is an MPI communicator
  - Usually `MPI_COMM_WORLD`
- `rank`
  - will be set to the rank of the calling process in the communicator of `comm`

Basic Functions

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- `comm` is an MPI communicator
  - Usually `MPI_COMM_WORLD`
- `size`
  - will be set to the number of ranks in the communicator `comm`

Hello MPI_COMM_WORLD

```
#include “stdio.h”
#include “mpi.h”

int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf(“Hello from rank %d of %d total \n”, rank, size);

    MPI_Finalize();
    return 0;
}
```

Hello MPI_COMM_WORLD

- Compile
  - `mpicc` wrapper used to link in libraries and includes
  - Uses standard C compiler, such as `gcc`, under the hood

```
$ mpicc hello_mpi.c -o hello
```

Hello MPI_COMM_WORLD

- Run
  - `mpirun -n # ./a.out` launches # copies of `a.out`

```
$ mpirun -n 3 ./hello
Hello from rank 0 of 3 total
Hello from rank 2 of 3 total
Hello from rank 1 of 3 total
```

Hello MPI_COMM_WORLD

- Run
  - `mpirun -n # ./a.out` launches # copies of `a.out`

```
$ mpirun -n 3 ./hello
Hello from rank 0 of 3 total
Hello from rank 2 of 3 total
Hello from rank 1 of 3 total
```

Note the order

MPI_Datatype

- Many MPI functions require a datatype
- Built in types for all intrinsic C types
  - MPI_INT, MPI_FLOAT, MPI_DOUBLE, …

Point to Point

- Point to Point routines
  - Involves two and only two processes
  - One process explicitly initiates send operation
  - One process explicitly initiates receive operation
  - Several send/receive flavors available
    - Blocking/non-blocking
    - Combined send-receive
  - Basis for more complicated messaging

Point to Point

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

`buf`
Initial address of send buffer

`count`
Number of elements to send

`datatype`
Datatype of each element in send buffer

`dest`
Rank of destination

`tag`
Integer tag used by receiver to identify message

`comm`
Communicator

Point to Point

```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status status)
```

`buf`
Initial address of receive buffer

`count`
Maximum number of elements that can be received

`datatype`
Datatype of each element in receive buffer

`source`
Rank of source

`tag`
Integer tag used to identify message

`comm`
Communicator

`status`
Struct containing information on received message

Point to Point: try 1

```
#include “mpi.h”
int main(int argc, char **argv)
{
    int rank, other_rank, tag, send_buff, recv_buff;
    MPI_Status status;
    tag = 5;
    send_buff = 10;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank==0)
        other_rank = 1;
    else
        other_rank = 0;

    MPI_Recv(&recv_buff, 1, MPI_INT, other_rank, tag, MPI_COMM_WORLD, &status);
    MPI_Send(&send_buff, 1, MPI_INT, other_rank, tag, MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```

Point to Point: try 1

- Compile

```
$ mpicc dead.c –o lock
```

- Run

```
$ mpirun –n 2 ./lock
…
…
…
deadlock
```

Deadlock

- The problem
  - Both processes (a.k.a. tasks) are waiting to receive a message
  - Neither process ever sends a message
  - Deadlock as both processes wait forever

Point to Point: Fix 1

```
#include "mpi.h"
int main(int argc, char **argv)
{
    int rank, other_rank, recv_buff;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank==0){
        other_rank = 1;
        MPI_Recv(&recv_buff, 1, MPI_INT, other_rank, 5, MPI_COMM_WORLD, &status);
        MPI_Send(&rank, 1, MPI_INT, other_rank, 5, MPI_COMM_WORLD);
    } else {
        other_rank = 0;
        MPI_Send(&rank, 1, MPI_INT, other_rank, 5, MPI_COMM_WORLD);
        MPI_Recv(&recv_buff, 1, MPI_INT, other_rank, 5, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
    return 0;
}
```

Non blocking P2P

- Non blocking Point to Point functions
  - Allow Send and Receive to not block on CPU
    - Return before buffers are safe to reuse
  - Can be used to prevent deadlock situations
  - Can be used to overlap communication and computation
  - Calls prefixed with "`I`", because they return **i**mmediately

Non blocking P2P

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

buf
: Initial address of send buffer

count
: Number of elements to send

datatype
: Datatype of each element in send buffer

dest
: Rank of destination

tag
: Integer tag used by receiver to identify message

request
: Object used to keep track of status of receive operation

Non blocking P2P

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

buf
    Initial address of receive buffer

count
    Maximum number of elements that can be received

datatype
    Datatype of each element in receive buffer

source
    Rank of source

tag
    Integer tag used to identify message

comm
    Communicator

request
    Object used to keep track of status of receive operation

Non blocking P2P

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

- `request`
  - The request you want to wait to complete
- `status`
  - Status struct containing information on completed request

- Will block until specified `request` operation is complete
- MPI_Wait, or similar, must be called if request is used

Point to Point: Fix 2

```
#include “mpi.h”
int main(int argc, char **argv)
{
    int rank, other_rank, recv_buff;
    MPI_Request send_req, recv_req;
    MPI_Status send_stat, recv_stat;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank==0) other_rank = 1;
    else other_rank = 0;

    MPI_Irecv(&recv_buff, 1, MPI_INT, other_rank, 5, MPI_COMM_WORLD, &recv_req);
    MPI_Isend(&rank, 1, MPI_INT, other_rank, 5, MPI_COMM_WORLD, &send_req);

    MPI_Wait(&recv_req, &recv_stat);
    MPI_Wait(&send_req, &send_stat);

    MPI_Finalize();
    return 0;
}
```

Collectives

- Collective routines
  - Involves all processes in communicator
  - All processes in communicator **must** participate
  - Serve several purposes
    - Synchronization
    - Data movement
    - Reductions
  - Several routines originate or terminate at a single process known as the “root”

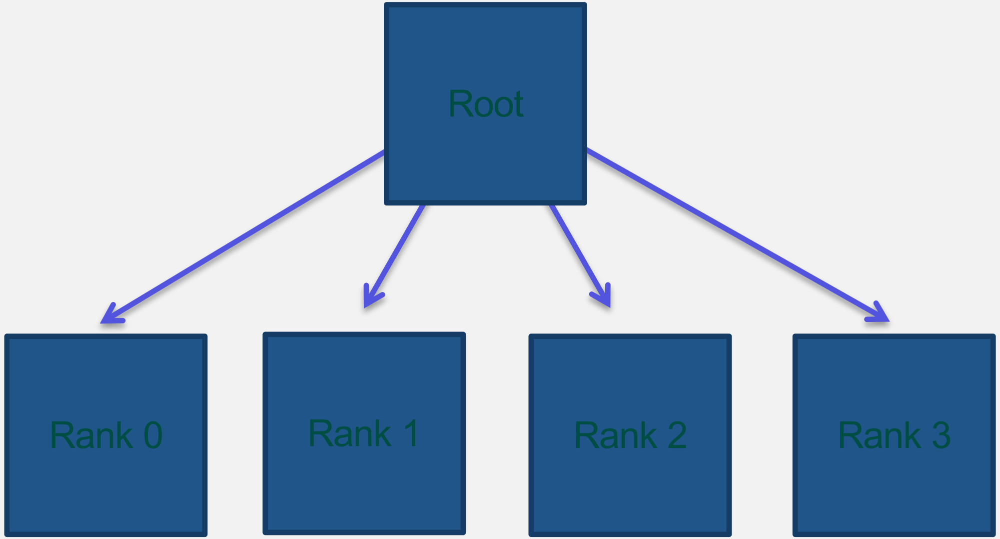
Collectives

```
int MPI_Barrier( MPI_Comm comm )
```

- Blocks process in `comm` until all process reach it
- Used to synchronize processes in `comm`

Collectives

Broadcast



Collectives

```
int MPI_Bcast(void *buf, int count,
             MPI_Datatype datatype, int root,
             MPI_Comm comm)
```

buf

Initial address of send buffer

count

Number of elements to send

datatype

Datatype of each element in send buffer

root

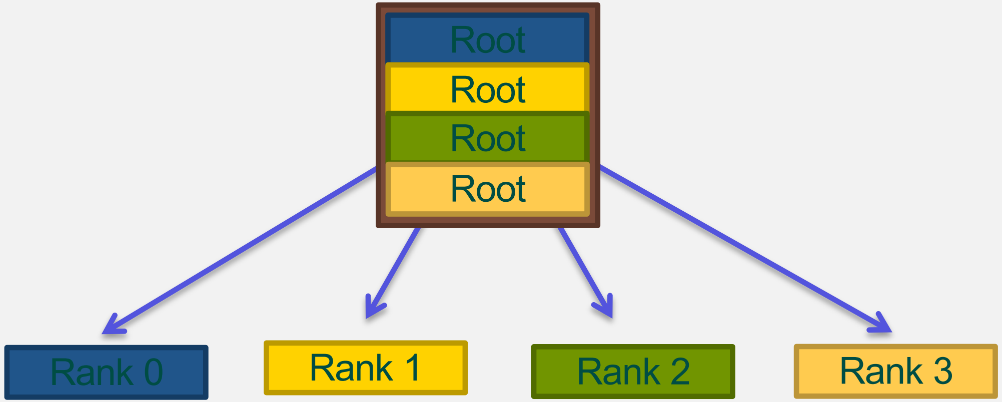
Rank of node that will broadcast buf

comm

Communicator

Collectives

Scatter



Collectives

```
int MPI_Scatter(const void *sendbuf, int sendcount,
                MPI_Datatype sendtype, void *recvbuf
                int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

sendbuf
    Initial address of send buffer on root node

sendcount
    Number of elements to send

sendtype
    Datatype of each element in send buffer

recvbuf
    Initial address of receive buffer on each node

recvcount
    Number of elements in receive buffer

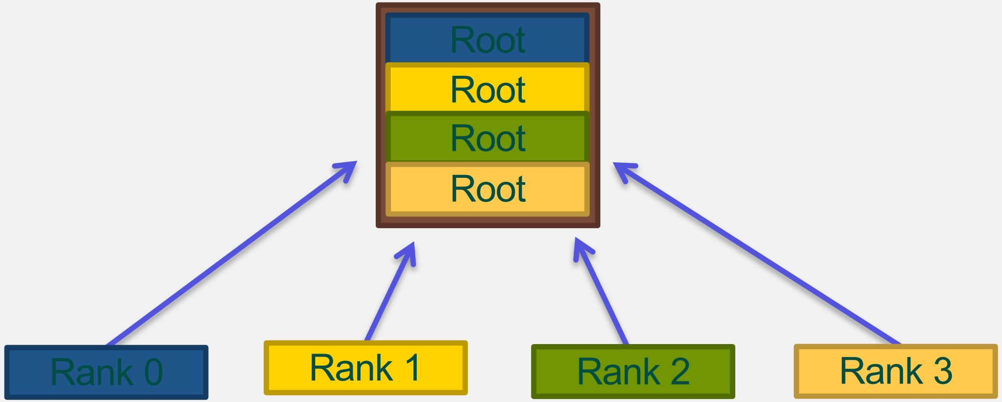
recvtype
    Datatype of each element in receive buffer

root
    Rank of node that will broadcast buf

comm
    Communicator

Collectives

Gather



Collectives

```
int MPI_Gather(const void *sendbuf, int sendcount,
               MPI_Datatype sendtype, void *recvbuf
               int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

sendbuf
Initial address of send buffer on root node

sendcount
Number of elements to send

sendtype
Datatype of each element in send buffer

recvbuf
Initial address of receive buffer on each node

recvcount
Number of elements in receive buffer

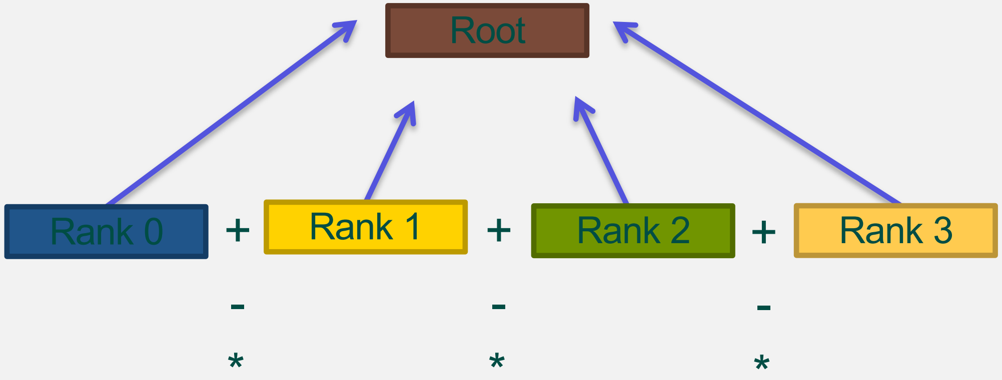
recvtype
Datatype of each element in receive buffer

root
Rank of node that will broadcast buf

comm
Communicator

Collectives

Reduce



Collectives

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

`sendbuf`
Initial address of send buffer

`recvbuf`
Buffer to receive reduced result on root rank

`count`
Number of elements in `sendbuf`

`datatype`
Datatype of each element in send buffer

`op`
Reduce operation (MPI_MAX, MPI_MIN, MPI_SUM, …)

`root`
Rank of node that will receive reduced value in recvbuf

`comm`
Communicator

Collectives: Example

```
#include “stdio.h”
#include “mpi.h”

int main(int argc, char **argv)
{
    int rank, root, bcast_data;
    root = 0;
    if(rank == root)
           bcast_data = 10;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Bcast(&bcast_data, 1, MPI_INT, root, MPI_COMM_WORLD);

    printf("Rank %d has bcast_data = %d\n", rank, bcast_data);

    MPI_Finalize();
    return 0;
}
```

Bug here???

Collectives: Example

```
$ mpicc collectives.c -o coll
$ mpirun -n 4 ./coll
Rank 0 has bcast_data = 10
Rank 1 has bcast_data = 10
Rank 3 has bcast_data = 10
Rank 2 has bcast_data = 10
```

Submit an MPI Job

- #!/bin/bash
- #
- #SBATCH --job-name=mpi_test
- #SBATCH --output=output.txt
- #
- # Number of MPI tasks
- #SBATCH -n 6
- #
- # Number of tasks per node
- #SBATCH --tasks-per-node=2
- #
- # Runtime of this jobs is less then 12 hours.
- #SBATCH --time=12:00:00
- # request to run on general nodes not dev nodes
- #SBATCH --partition=general
- #
- mpirun -n 6 sleep 120
- # End of submit file