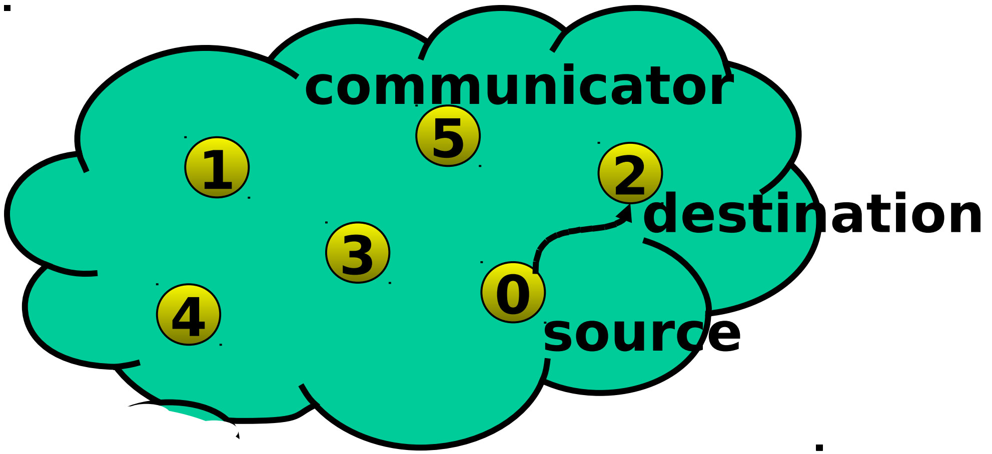
Point to Point Communication



- Communication between two processes
- Source process *sends* message to destination process
- Destination process *receives* the message
- Communication takes place within a communicator
- Destination process is identified by its rank in the communicator

Sending a Message

- MPI_Send

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

buf starting *address* of the data to be sent

count number of elements to be sent

datatype MPI datatype of each element

dest rank of destination process

tag message marker (set by user)

comm MPI communicator of processors involved

MPI_SEND(data,500,MPI_REAL,6,33,MPI_COMM_WORLD)

Receiving a Message

- MPI_Recv

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

The status variable can be used to get information about the MPI_Recv operation (source, tag and error)

```
double num;
MPI_Status status;
MPI_Recv(&num,1,MPI_DOUBLE,3,100,MPI_COMM_WORLD,&status);
```

For a communication to succeed

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver’s buffer must be large enough

Wildcarding

- Receiver can wildcard
- To receive from any source

  `MPI_ANY_SOURCE`

  To receive with any tag

  `MPI_ANY_TAG`

- Actual source and tag are returned in the receiver's `status` parameter

Blocking

- Function call does not return until the communication is complete.
- MPI_Send & MPI_Recv are blocking calls.
- Calling order matters: ELSE DEADLOCK possible.
- Improper order can also result in loss of performance and serialization.

- MPI_SendRecv : Send and receive in 1 call.
- Cleans code and can avoid possible deadlocks.

Non-Blocking

- Function call returns immediately – without completion of data transfer.
- Need additional checks in code to ensure completion (e.g: MPI_Wait).
- No Deadlock.
- Possible performance gain.
- E.g: MPI_Isend & MPI_Irecv.
- ***Sending process should not access send buffer (possible overwrites) until transfer is complete.

Collective Communication

- Communication involving a group of processes
- All collective communication operations in MPI take as argument a communicator that defines the group of processes that participate in the operation
- All the processes participating must call the collective operation
- Operations are synchronous, so do not require tags

Characteristics of Collective Communication

- Collective communication will not interfere with point-to-point communication and vice-versa
- All processes must call the collective routine
- No non-blocking collective communication
- No tags
- Receive buffers must be exactly the right size

Barrier

- Barrier synchronisation operation

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

- Only argument: communicator that defines the group of processes that are synchronized
- The call to MPI_Barrier returns only after all the processes in the group have called this function
- *Red light for each processor: turns green when all processors have arrived*

Broadcast

- One-to-all broadcast

```
int MPI_Bcast(void *buf, int count,
  MPI_Datatype datatype, int source,
  MPI_comm comm)
```

- Sends data stored in buf of process source to all other processes in the group
- Data received by each process is stored in buf
- Data consists of count entries of type datatype
- Count and datatype must match on all processes

Sample Broadcast Program

```
#include<mpi.h>
void main (int argc, char *argv[]) {
  int rank;
  double param;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  if(rank==5) param=23.0;
  MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
  printf("P:%d after broadcast parameter is
%f\n",rank,param);
  MPI_Finalize();
}
```

```
P:0 after broadcast parameter is 23.000000
P:6 after broadcast parameter is 23.000000
P:5 after broadcast parameter is 23.000000
P:2 after broadcast parameter is 23.000000
P:3 after broadcast parameter is 23.000000
P:7 after broadcast parameter is 23.000000
P:1 after broadcast parameter is 23.000000
P:4 after broadcast parameter is 23.000000
```

Reduction

- All-to-one reduction

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

- Combines the elements in sendbuf of each process in the group, using the operation op, and returns the combined values in recvbuf of the process with rank target
- Sendbuf and recvbuf must have same number of count items of type datatype
- Note: all processes must provide recvbuf, even if they are not the target
- If count is more than one, op is applied element-wise

Predefined Reduction Operations

- MPI_MAX (Maximum)
- MPI_MIN (Minimum)
- MPI_SUM (Sum)
- MPI_PROD (Product)
- MPI_LAND (Logical AND)
- MPI_BAND (Bit-wise AND)
- MPI_LOR (Logical OR)
- MPI_BOR (Bit-wise OR)
- MPI_LXOR (Logical XOR)
- MPI_BXOR (Bit-wise XOR)
- MPI_MAXLOC (max-min value-location)
- MPI_MINLOC (min-min value-location)

Sample Reduction Program

```
#include <mpi.h>
/* Run with 16 processes */

void main (int argc, char *argv[]) {
   int rank;
   double value;
   int rank;
   int root;
   double sum;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   value=rank+1;

   root=7;

MPI_Reduce(&value,&sum,1,MPI_DOUBLE,MPI_SUM,root,MPI_COMM_WORLD
);
   if(rank==root) printf("PE:%d sum=%lf”, rank, sum);
   MPI_Finalize();
}
```

PE:7 sum=136.0

All-Reduce

- When the result of the reduction operation is needed by all the processes

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

- No target argument

References

- Cheerkoot-Jalim, Sudha (2018). Message Passing Interface. Retrieved from University of Mauritius, CSE3057Y Parallel and Distributed Systems,www.uom.ac.mu .

- Huang, Shao-Ching (2013). MPI Tutorial. IDRE High Performance Computing Workshop, https://idre.ucla.edu .