CPSC 4770/6770

Distributed and Cluster Computing

Lecture 4: Introduction to MPI

Message Passing

- Processes communicate via messages
- Messages can be
  - Raw data to be used in actual calculations
  - Signals and acknowledgements for the receiving processes regarding the workflow

History of MPI

- **Early 80s:**
  - Various message passing environments were developed
  - Many similar fundamental concepts
  - N-cube (Caltech), P4 (Argonne), PICL and PVM (Oakridge), LAM (Ohio SC)
- **1992:**
  - More than 80 researchers from different institutions in US and Europe agreed to develop and implement a common standard for message passing
  - First meeting collocated with Supercomputing 1992
- **After finalization:**
  - MPI becomes the *de facto* standard for distributed memory parallel programming
  - Available on every popular operating system and architecture
  - Interconnect manufacturers commonly provide MPI implementations optimized for their hardware
  - MPI standard defines interfaces for C, C++, and Fortran
    - Language bindings available for many popular languages (quality varies)
    - MPI4PY: Bindings for python

History of MPI (Cont.)

- **1994: MPI-1**
  - Communicators
    - Information about the runtime environments
    - Creation of customized topologies
  - Point-to-point communication
    - Send and receive messages
    - Blocking and non-blocking variations
  - Collectives
    - Broadcast and reduce
    - Gather and scatter
- **1998: MPI-2**
  - One-sided communication (non-blocking)
    - Get & Put (remote memory access)
  - Dynamic process management
    - Spawn
  - Parallel I/O
    - Multiple readers and writers for a single file
    - Requires file-system level support (LustreFS, PVFS)
- **2012: MPI-3**
  - Revised remote-memory access semantic
  - Fault tolerance model
  - Non-blocking collective communication
  - Access to internal variables, states, and counters for performance evaluation purposes

Set up MPI on Palmetto for C/C++

- Interactive mode:
  - If use command line terminal:
    - qsub -I -l select=1:ncpus=8:mpiprocs=8:mem=10gb,walltime=01:00:00
    - module purge
    - module load openmpi/1.10.3-gcc/5.4.0-cuda9_2
  - Create a file named first.c

```c
#include <stdio.h>
#include <sys/utsname.h>
#include <mpi.h>
int main(int argc, char *argv[]){
  MPI_Init(&argc, &argv);
  struct utsname uts;
  uname (&uts);
  printf("My process is on node %s.\n", uts.nodename);
  MPI_Finalize();
  return 0;
}
```

  - Compile first.c
    mpicc first.c -o first
  - Run first.c
    mpirun -np 2 ./first or mpiexec –np 2 ./first
    mpirun -np 2 --mca mpi_cuda_support 0 first

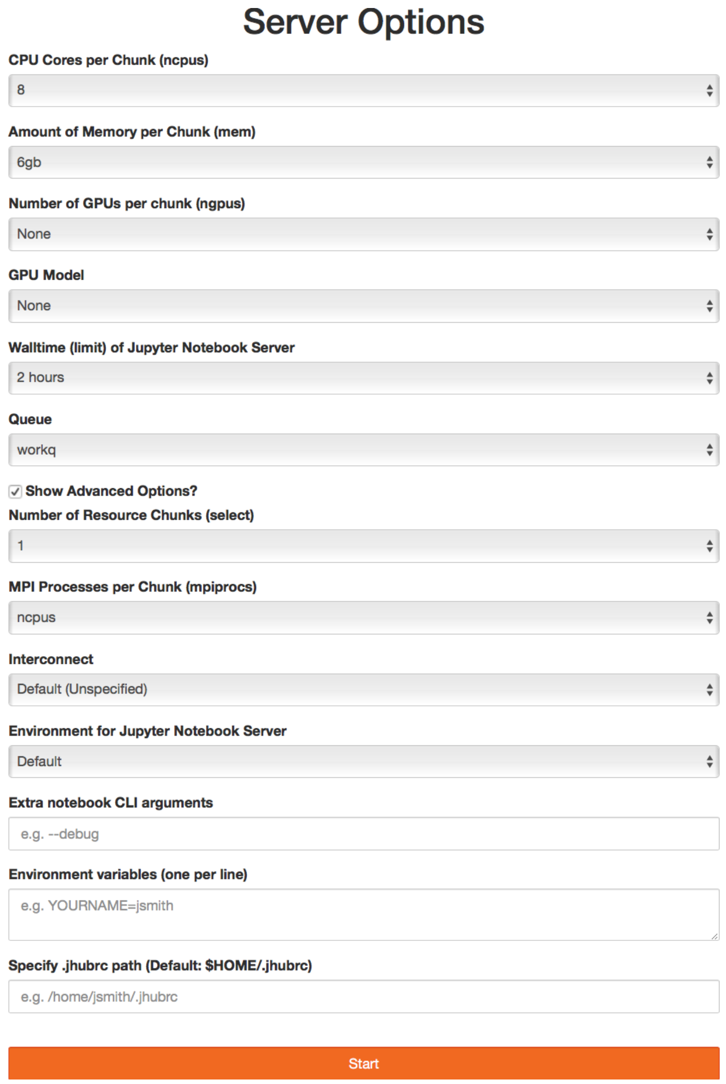
```
jin6@node0061:~/cpsc477(

[jin6@node0061 ~]$ module list

Currently Loaded Modules:
  1) anaconda3/5.1.0-gcc/8.3.1



[jin6@node0061 ~]$ module load openmpi/1.10.3-gcc/5.4.0-cuda9_2
[jin6@node0061 ~]$ module list

Currently Loaded Modules:
  1) anaconda3/5.1.0-gcc/8.3.1   2) cuda/9.2.88-gcc/5.4.0   3) openmpi/1.10.3-gcc/5.4.0-cuda9_2



[jin6@node0061 ~]$ cd cpsc4770-6770/04-intro-to-mpi/C
[jin6@node0061 C]$ mpicc first.c -o first
[jin6@node0061 C]$ mpirun -np 1 --mca mpi_cuda_support 0 first
--------------------------------------------------------------------------
[[12299,1],0]: A high-performance Open MPI point-to-point messaging module
was unable to find any relevant network interfaces:

Module: OpenFabrics (openib)
  Host: node0061

Another transport will be used instead, although this may result in
lower performance.
--------------------------------------------------------------------------
My process is on node node0061.palmetto.clemson.edu.
[jin6@node0061 C]$ mpirun -np 1 --mca mpi_cuda_support 0 hello
--------------------------------------------------------------------------
[[12308,1],0]: A high-performance Open MPI point-to-point messaging module
was unable to find any relevant network interfaces:

Module: OpenFabrics (openib)
  Host: node0061

Another transport will be used instead, although this may result in
lower performance.
--------------------------------------------------------------------------
Hello world from processor node0061.palmetto.clemson.edu, rank 0 out of 1 processes
```

```
[jin6@node1782 C]$ mpicc first.c -o first
[jin6@node1782 C]$ mpirun -np 2 first
My process is on node node1782.palmetto.clemson.edu.
My process is on node node1782.palmetto.clemson.edu.
```

```
[jin6@node0400 C]$ mpirun -np 2 --mca mpi_cuda_support 0 first
My process is on node node0400.palmetto.clemson.edu.
My process is on node node0400.palmetto.clemson.edu.
```

Set up MPI on Palmetto for Python (Interactive via Jupyter Notebook)

- **Before launching JupyterHub**
  - Make sure that you have the command module load openmpi/1.10.3-gcc/5.4.0-cuda9_2 in your .jhubrc file. If you are using JupyterHub to edit the file, the server will need to be stopped and started again.
- **Before launching a Jupyter notebook (only need to be done once)**
  - Install mpi4py by executing pip install --user mpi4py from a terminal. This needs to be done prior to launching a Jupyter notebook.



The Working of MPI in a Nutshell

- All processes are launched at the beginning of the program execution
  - The number of processes are user-specified
  - Typically, this number is matched to the total number of cores available across the entire cluster
- All processes have their own memory space and have access to the same source codes
- Basic parameters available to individual processors:
  - MPI_COMM_WORLD
  - MPI_Comm_rank()
  - MPI_Comm_size()
  - MPI_Get_processor_name()
- MPI defines **communicator** groups for point-to-point and collective communications
  - Unique IDs (**rank**) are defined for individual processes within a communicator group
  - Communications are performed based on these IDs
  - Default **global communication** (MPI_COMM_WORLD) contains all processes
  - For *N* processors, ranks go from *0* to *N-1*

hello.c

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
  int size;
  int my_rank;
  char proc_name[MPI_MAX_PROCESSOR_NAME];
  int proc_name_len;

  /* Initialize the MPI environment */
  MPI_Init(&argc, &argv);

  /* Get the number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  /* Get the rank of the process */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  /* Get the name of the processor */
  MPI_Get_processor_name(proc_name, &proc_name_len);

  /* Print off a hello world message */
  printf("Hello world from processor %s, rank %d out of %d processes\n",
         proc_name, my_rank, size);

  /* Finalize the MPI environment. */
  MPI_Finalize();
}
```

```
[jin6@node1782 C]$ mpicc hello.c -o hello
[jin6@node1782 C]$ mpirun -np 2 hello
Hello world from processor node1782.palmetto.clemson.edu, rank 0 out of 2 processes
Hello world from processor node1782.palmetto.clemson.edu, rank 1 out of 2 processes
```

evenodd.c

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
  int my_rank;

  /* Initialize the MPI environment */
  MPI_Init(&argc, &argv);

  /* Get the rank of the process */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if (my_rank % 2 == 1) {
    printf ("Process %d is odd\n", my_rank);
  } else {
    printf ("Process %d is even \n", my_rank);
  }
  MPI_Finalize();
}
```

Ranks are used to enforce execution/exclusion of code segments within the original source code

```
[jin6@node1782 C]$ mpicc evenodd.c -o evenodd
[jin6@node1782 C]$ mpirun -np 4 evenodd
Process 3 is odd
Process 0 is even
Process 1 is odd
Process 2 is even
[jin6@node1782 C]$ mpirun -np 4 evenodd
Process 0 is even
Process 1 is odd
Process 2 is even
Process 3 is odd
```

rank_size.c

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
  int size;
  int my_rank;
  int A[16] = {2,13,4,3,5,1,0,12,10,8,7,9,11,6,15,14};
  int i;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  for (i = 0; i < 16; i++){
    if (i % size == my_rank){
      printf ("Process %d has elements %d at index %d \n",
              my_rank, A[i], i);
    }
  }

  /* Finalize the MPI environment. */
  MPI_Finalize();
}
```

Ranks and size are used means to calculate and distribute workload (data) among the processes

```
[jin6@node1782 C]$ mpicc rank_size.c -o rank_size
[jin6@node1782 C]$ mpirun -np 4 rank_size
Process 2 has elements 4 at index 2
Process 2 has elements 0 at index 6
Process 2 has elements 7 at index 10
Process 2 has elements 15 at index 14
Process 3 has elements 3 at index 3
Process 3 has elements 12 at index 7
Process 3 has elements 9 at index 11
Process 3 has elements 14 at index 15
Process 0 has elements 2 at index 0
Process 0 has elements 5 at index 4
Process 0 has elements 10 at index 8
Process 0 has elements 11 at index 12
Process 1 has elements 13 at index 1
Process 1 has elements 1 at index 5
Process 1 has elements 8 at index 9
Process 1 has elements 6 at index 13
```

Communications

- Individual processes rely on communication (message passing) to enforce workflow
  - Point-to-point Communication
  - Collective Communication

Point-to-Point Communication: Send

- **Original MPI C Syntax: MPI_Send**

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

- MPI_Datatype may be MPI_BYTE, MPI_PACKED, MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_UNSIGNED_CHAR
- *dest* is the rank of the process the message is sent to
- *tag* is an integer identify the message. Programmer is responsible for managing tag

Point-to-Point Communication: Receive

- **Original MPI C Syntax: MPI_Recv**

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

- MPI_Datatype may be MPI_BYTE, MPI_PACKED, MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_UNSIGNED_CHAR
- *source* is the rank of the process from which the message was sent
- *tag* is an integer identify the message. MPI_Recv will only place data in the buffer if the tag from MPI_Send matches. The constant MPI_ANY_TAG may be used when the source tag is unknown or not important

Point-to-Point Communication

- Message passing between two, and only two, different MPI tasks: One task is performing a send operation and the other task is performing a matching receive operation

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

must correspond

must match

recv count must be equal to or higher than send count

must match

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

MPI data type	C data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing
MPI_BYTE	single byte value

Each Send must be matched with a corresponding Recv
Messages are delivered in the order in which they have been sent

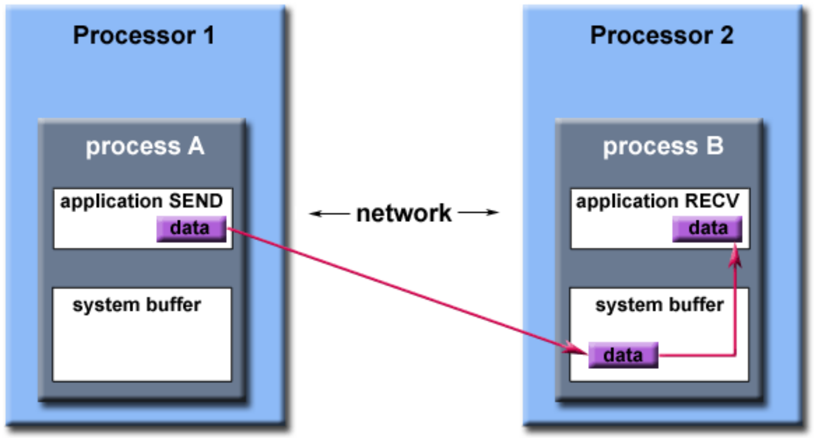
hello2.c

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]){
        int  my_rank; /* rank of process */
        int  p;       /* number of processes */
        int source;   /* rank of sender */
        int dest;     /* rank of receiver */
        int tag=0;    /* tag for messages */
        char message[100];        /* storage for message */
        MPI_Status status ;   /* return status for receive */

        /* start up MPI */
        MPI_Init(&argc, &argv);

        /* find out process rank */
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        /* find out number of processes */
        MPI_Comm_size(MPI_COMM_WORLD, &p);

        if (my_rank !=0){
                /* create message */
                sprintf(message, "Hello MPI World from process %d!", my_rank);
                dest = 0;
                /* use strlen+1 so that '\0' get transmitted */
                MPI_Send(message, strlen(message)+1, MPI_CHAR,
                   dest, tag, MPI_COMM_WORLD);
        }
        else{
                printf("Hello MPI World From process 0: Num processes: %d\n",p);
                for (source = 1; source < p; source++) {
                        MPI_Recv(message, 100, MPI_CHAR, source, tag,
                              MPI_COMM_WORLD, &status);
                        printf("%s\n",message);
                }
        }

        /* shut down MPI */
        MPI_Finalize();

        return 0;
}
```

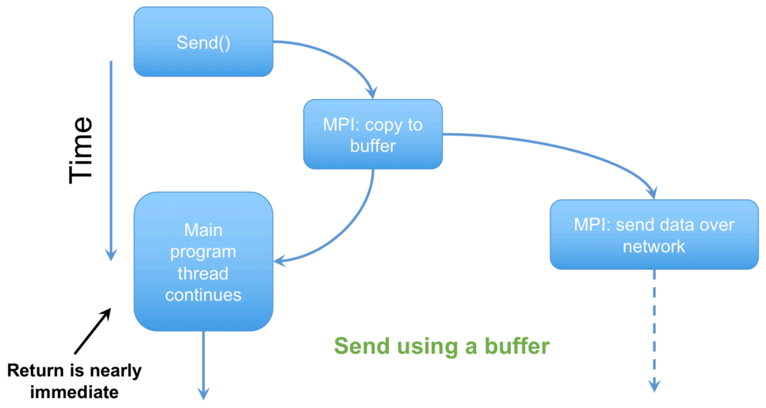
```
[jin6@node0125 hello2]$ mpirun -np 8 hello
Hello MPI World From process 0: Num processes: 8
Hello MPI World from process 1!
Hello MPI World from process 2!
Hello MPI World from process 3!
Hello MPI World from process 4!
Hello MPI World from process 5!
Hello MPI World from process 6!
Hello MPI World from process 7!
```

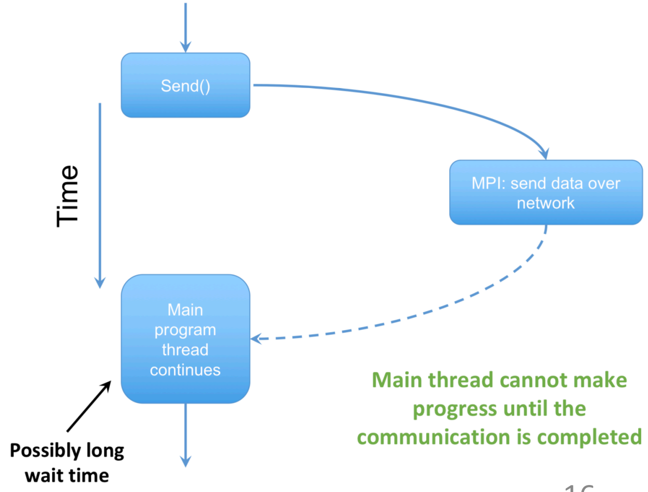
Buffering

- Consider the following two cases:
  - A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
  - Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?
- A **system buffer** area is reserved to hold data in transit



Path of a message buffered at the receiving process





Blocking Message Passing

- A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse
  - Safe means that modifications will not affect the data intended for the receive task
  - Safe does not imply that the data was actually received - it may very well be sitting in a system buffer
- A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send
- A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive
- A blocking receive only "returns" after the data has arrived and is ready for use by the program

Blocking Message Passing Routines

- Exact blocking behavior is implementation-dependent
  - MPI_Send **may** block until the message is sent
    - Sometimes depends on the size of the message
    - MPI_Ssend will **always** block until the message is received
  - MPI_Recv will **always** block until the message is received
  - MPI_SendRecv sends a message and post a receive before blocking

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

int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag,
                  MPI_Comm comm, MPI_Status *status )
```

block.c

```c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])  {
   int numtasks, rank, dest, source, rc, count, tag=1;
   char inmsg, outmsg='x';
   MPI_Status Stat;   // required variable for receive routines

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

   // task 0 sends to task 1 and waits to receive a return message
   if (rank == 0) {
     dest = 1;
     source = 1;
     MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
     MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
     }

   // task 1 waits for task 0 message then returns a message
   else if (rank == 1) {
     dest = 0;
     source = 0;
     MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
     MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
     }

   // query recieve Stat variable and print message details
   MPI_Get_count(&Stat, MPI_CHAR, &count);
   printf("Task %d: Received %d char(s) from task %d with tag %d \n",
          rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

   MPI_Finalize();
 }
```

Process 0	Process 1	Deadlock
Recv() Send()	Recv() Send()	Always
Send() Recv()	Send() Recv()	Depends on whether a buffer is used or not
Send() Recv()	Recv() Send()	Secure

Be aware of deadlock, do secure implementation!

```
[jin6@node1554 blocking]$ mpirun -np 2 block
Task 0: Received 1 char(s) from task 1 with tag 1
Task 1: Received 1 char(s) from task 0 with tag 1
```

Non-blocking Message Passing

- Non-blocking will return almost immediately
  - They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message
- Non-blocking operations simply "request" the MPI library to perform the operation when it is able
  - The user can not predict when that will happen
- It is unsafe to modify the application buffer (your variable space)
  - Until you know for a fact the requested non-blocking operation was actually performed by the library
  - There are "wait" routines used to do this
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains

Non-blocking Message Passing Routines

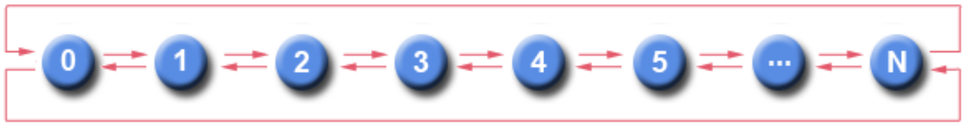
- Some operations are guaranteed not to block
  - MPI_Isend and MPI_Irecv
- These operations merely “request” some communication
  - MPI_Request variables can be used to track these requests
  - MPI_Wait blocks until an operation has finished
  - MPI_Test sets a flag if the operation has finished

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

int MPI_Wait ( MPI_Request *request, MPI_Status   *status)

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

nonblock.c

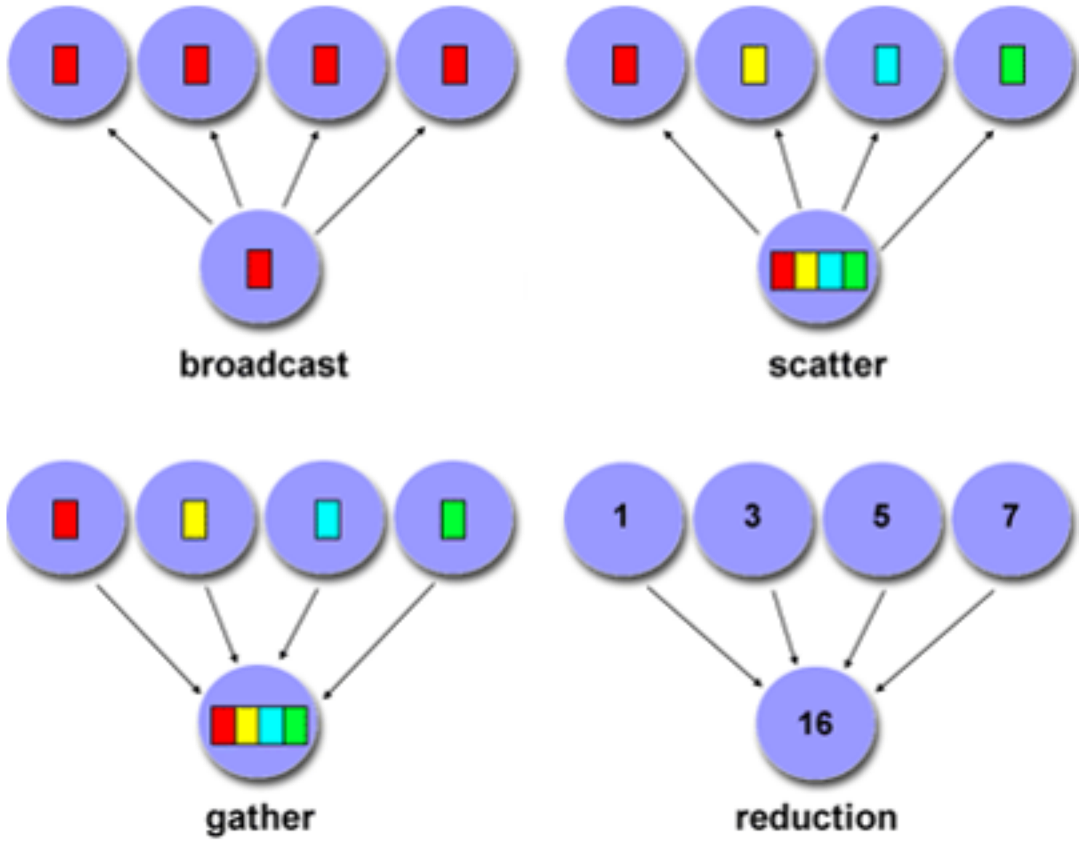
```c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])  {
   int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
   MPI_Request reqs[4];   // required variable for non-blocking calls
   MPI_Status stats[4];   // required variable for Waitall routine

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

   // determine left and right neighbors
   prev = rank-1;
   next = rank+1;
   if (rank == 0)  prev = numtasks - 1;
   if (rank == (numtasks - 1))  next = 0;

   // post non-blocking receives and sends for neighbors
   MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
   MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

   MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
   MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

      // do some work while sends/receives progress in background

   // wait for all non-blocking operations to complete
   MPI_Waitall(4, reqs, stats);

      // continue - do more work

   MPI_Finalize();
}
```

Nearest neighbor exchange in a ring topology



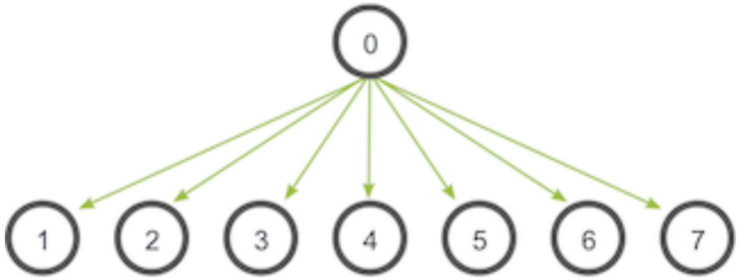
Collective Communication

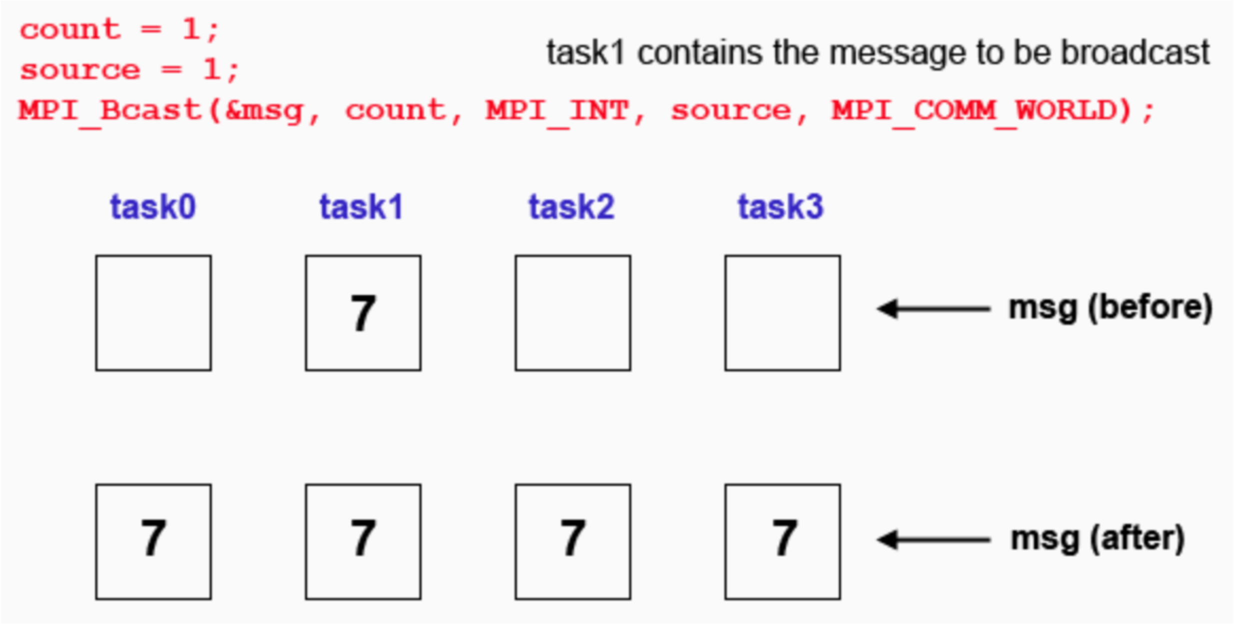
- Must involve ALL processes within the scope of a communicator
- Unexpected behavior, including programming failure, if even one process does not participate
- Types of collective communications:
  - Synchronization: barrier
  - Data movement: broadcast, scatter/gather
  - Collective computation (aggregate data to perform computation): Reduce



Broadcast

- int **MPI_Bcast** (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm): Broadcasts a message from the process with rank "root" to all other processes of the group





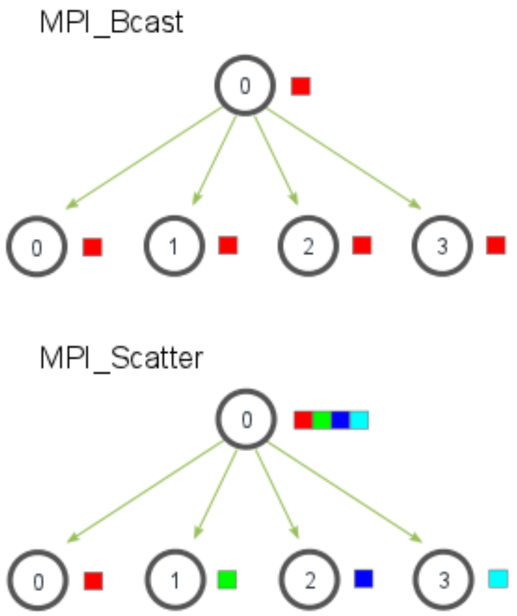
bcast.c

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
  int my_rank;
  int size;
  int value;

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

  value = my_rank;
  printf("process %d: Before MPI_Bcast, value is %d\n", my_rank, value);

  MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);
  printf("process %d: After MPI_Bcast, value is %d\n", my_rank, value);

  MPI_Finalize();
  return 0;
}
```

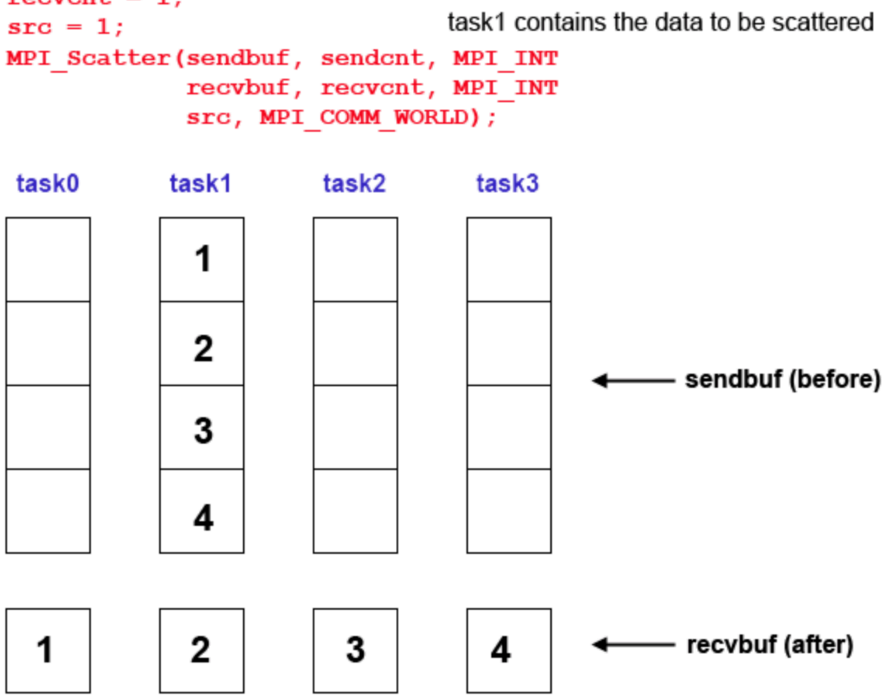
```
[jin6@node1169 C]$ mpirun -np 4 bcast
process 0: Before MPI_Bcast, value is 0
process 0: After MPI_Bcast, value is 0
process 1: Before MPI_Bcast, value is 1
process 1: After MPI_Bcast, value is 0
process 2: Before MPI_Bcast, value is 2
process 2: After MPI_Bcast, value is 0
process 3: Before MPI_Bcast, value is 3
process 3: After MPI_Bcast, value is 0
[jin6@node1169 C]$ mpirun -np 4 bcast
process 2: Before MPI_Bcast, value is 2
process 2: After MPI_Bcast, value is 0
process 1: Before MPI_Bcast, value is 1
process 3: Before MPI_Bcast, value is 3
process 3: After MPI_Bcast, value is 0
process 1: After MPI_Bcast, value is 0
process 0: Before MPI_Bcast, value is 0
process 0: After MPI_Bcast, value is 0
```

MPI_Scatter

- int **MPI_Scatter** (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm): Distributes distinct messages from a single source task to each task in the group




```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Scatter(sendbuf, sendcnt, MPI_INT
            recvbuf, recvcnt, MPI_INT
            src, MPI_COMM_WORLD);
```



scatter.c

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
  int size;
  int my_rank;
  int sendbuf[16] = {2,13,4,3,5,1,0,12,10,8,7,9,11,6,15,14};
  int recvbuf[5];
  int i;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  MPI_Scatter(&sendbuf, 5, MPI_INT, &recvbuf, 5, MPI_INT, 0, MPI_COMM_WORLD);
  for (i = 0; i < 5; i++){
    printf ("Process %d has element %d at index %d in its recvbuf \n",
                 my_rank, recvbuf[i], i);
  }

  /* Finalize the MPI environment. */
  MPI_Finalize();
}
```

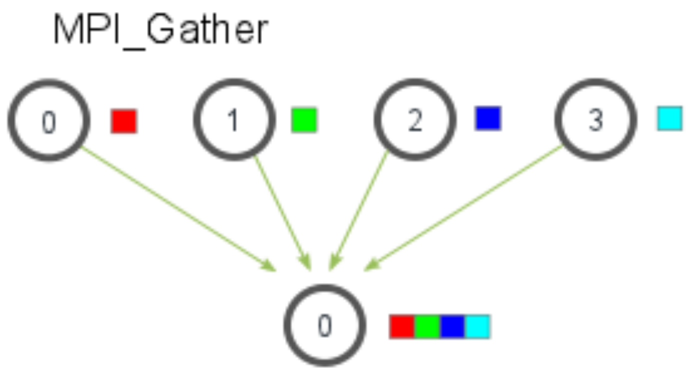
```
[jin6@node1169 C]$ mpirun -np 4 scatter
Process 0 has element 2 at index 0 in its recvbuf
Process 0 has element 13 at index 1 in its recvbuf
Process 0 has element 4 at index 2 in its recvbuf
Process 0 has element 3 at index 3 in its recvbuf
Process 0 has element 5 at index 4 in its recvbuf
Process 1 has element 1 at index 0 in its recvbuf
Process 1 has element 0 at index 1 in its recvbuf
Process 1 has element 12 at index 2 in its recvbuf
Process 1 has element 10 at index 3 in its recvbuf
Process 1 has element 8 at index 4 in its recvbuf
Process 2 has element 7 at index 0 in its recvbuf
Process 2 has element 9 at index 1 in its recvbuf
Process 2 has element 11 at index 2 in its recvbuf
Process 2 has element 6 at index 3 in its recvbuf
Process 2 has element 15 at index 4 in its recvbuf
Process 3 has element 14 at index 0 in its recvbuf
Process 3 has element 243294384 at index 1 in its recvbuf
Process 3 has element 0 at index 2 in its recvbuf
Process 3 has element 4 at index 3 in its recvbuf
Process 3 has element 0 at index 4 in its recvbuf
```

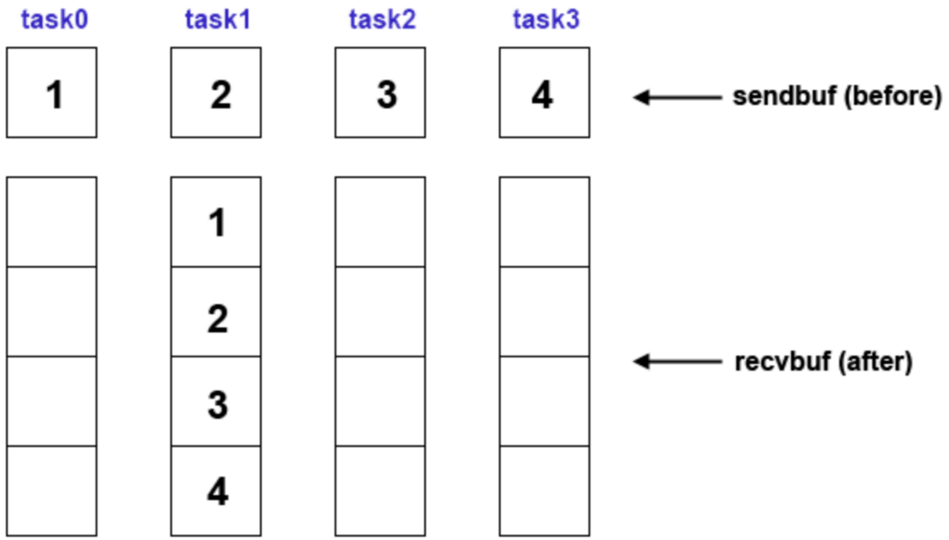
MPI_Gather

- int **MPI_Gather** (void *sendbuf, int sendcnt, MPI_Datatype sendtype,void *recvbuf, int recvcnt, MPI_Datatype recvtype,int root, MPI_Comm comm): Gathers distinct messages from each task in the group to a single destination task (the reverse operation of MPI_Scatter)

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Gather(sendbuf, sendcnt, MPI_INT
           recvbuf, recvcnt, MPI_INT
           src, MPI_COMM_WORLD);
```

message will be gathered into task1





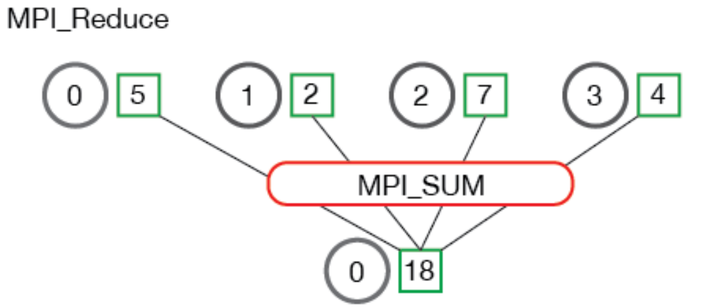
gather.c

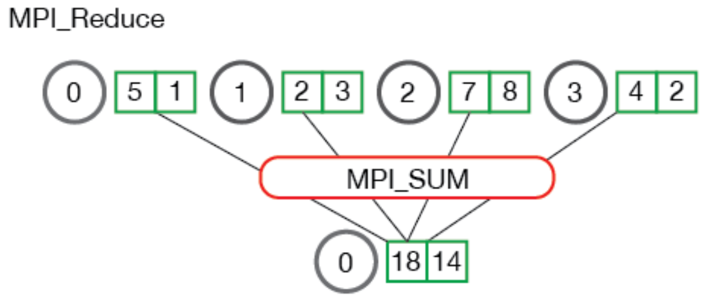
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
  int size;
  int my_rank;
  int sendbuf[2];
  int recvbuf[8] = {-1,-1,-1,-1,-1,-1,-1,-1};
  int i;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  for (i = 0; i < 2; i++){
    sendbuf[i] = my_rank;
  }
  MPI_Gather(&sendbuf, 2, MPI_INT, &recvbuf, 2, MPI_INT, 0, MPI_COMM_WORLD);
  for (i = 0; i < 8; i++){
    printf ("Process %d has element %d at index %d in its recvbuf \n",
                 my_rank, recvbuf[i], i);
  }

  /* Finalize the MPI environment. */
  MPI_Finalize();
}
```

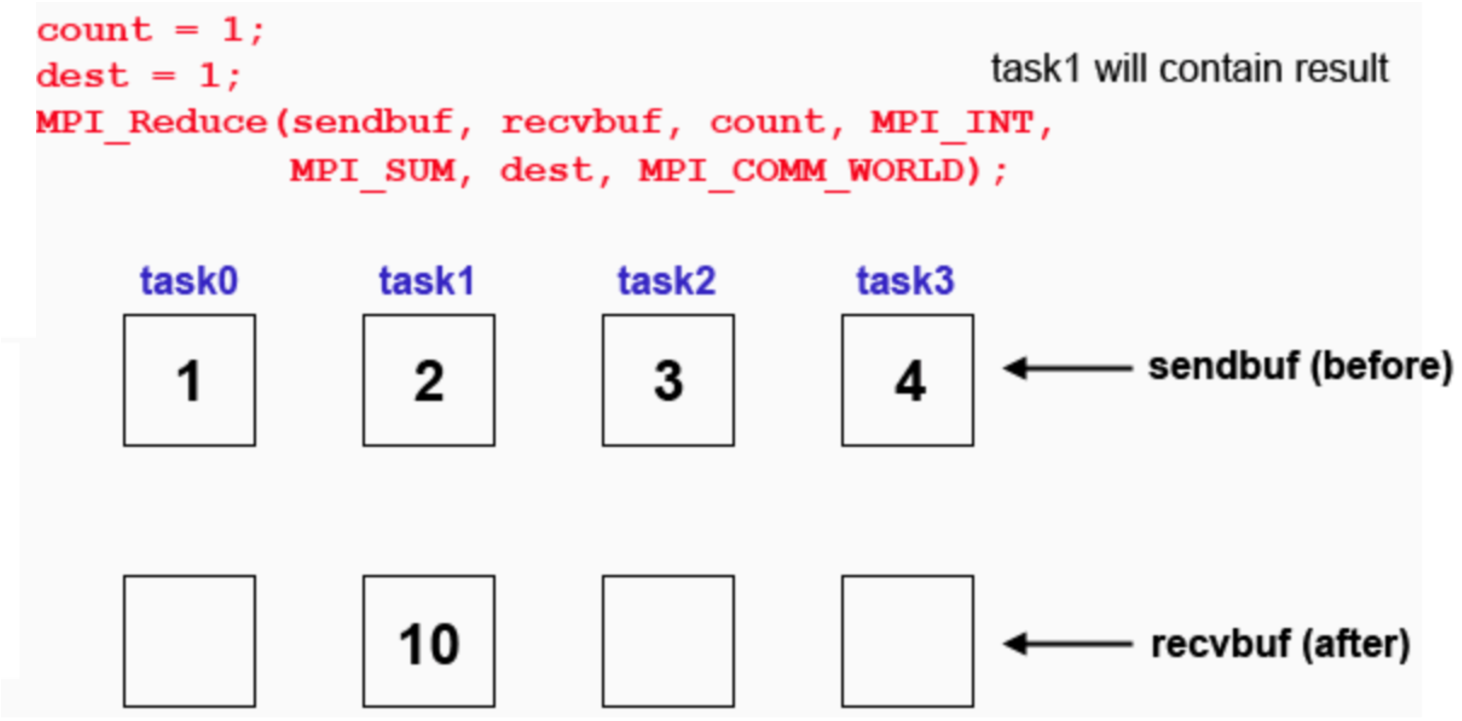
```
[jin6@node1169 C]$ mpirun -np 4 gather
Process 2 has element -1 at index 0 in its recvbuf
Process 2 has element -1 at index 1 in its recvbuf
Process 2 has element -1 at index 2 in its recvbuf
Process 2 has element -1 at index 3 in its recvbuf
Process 2 has element -1 at index 4 in its recvbuf
Process 2 has element -1 at index 5 in its recvbuf
Process 2 has element -1 at index 6 in its recvbuf
Process 2 has element -1 at index 7 in its recvbuf
Process 1 has element -1 at index 0 in its recvbuf
Process 1 has element -1 at index 1 in its recvbuf
Process 1 has element -1 at index 2 in its recvbuf
Process 1 has element -1 at index 3 in its recvbuf
Process 1 has element -1 at index 4 in its recvbuf
Process 1 has element -1 at index 5 in its recvbuf
Process 1 has element -1 at index 6 in its recvbuf
Process 1 has element -1 at index 7 in its recvbuf
Process 3 has element -1 at index 0 in its recvbuf
Process 3 has element -1 at index 1 in its recvbuf
Process 3 has element -1 at index 2 in its recvbuf
Process 3 has element -1 at index 3 in its recvbuf
Process 3 has element -1 at index 4 in its recvbuf
Process 3 has element -1 at index 5 in its recvbuf
Process 3 has element -1 at index 6 in its recvbuf
Process 3 has element -1 at index 7 in its recvbuf
Process 0 has element 0 at index 0 in its recvbuf
Process 0 has element 0 at index 1 in its recvbuf
Process 0 has element 1 at index 2 in its recvbuf
Process 0 has element 1 at index 3 in its recvbuf
Process 0 has element 2 at index 4 in its recvbuf
Process 0 has element 2 at index 5 in its recvbuf
Process 0 has element 3 at index 6 in its recvbuf
Process 0 has element 3 at index 7 in its recvbuf
```

MPI_Reduce

- int **MPI_Reduce** (void *sendbuf, void *recvbuf, int count,MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm): Reduces values on all processes to a single value







reduce.c

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
  int size;
  int my_rank;
  int rank_sum;
  int i;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  rank_sum = my_rank;

  MPI_Reduce(&my_rank, &rank_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  printf ("The total sum of all ranks at process %d is %d \n", my_rank, rank_sum);

  /* Finalize the MPI environment. */
  MPI_Finalize();
}
```

```
[jin6@node0329 C]$ mpirun -np 8 reduce
The total sum of all ranks at process 7 is 7
The total sum of all ranks at process 1 is 1
The total sum of all ranks at process 2 is 2
The total sum of all ranks at process 3 is 3
The total sum of all ranks at process 5 is 5
The total sum of all ranks at process 0 is 28
The total sum of all ranks at process 4 is 4
The total sum of all ranks at process 6 is 6
```