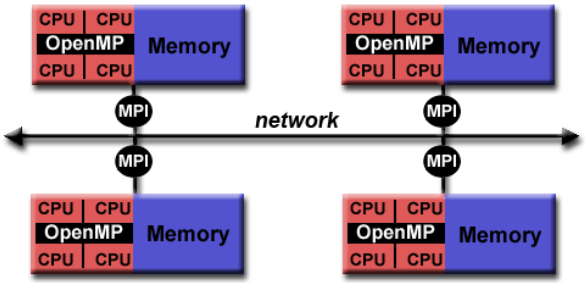
CMSC 691
High Performance Distributed Systems

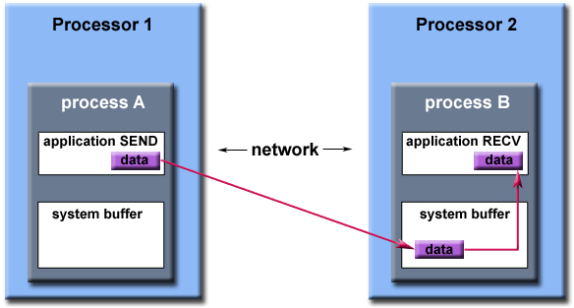
Message Passing Interface

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

MPI (Message Passing Interface)

- Standardized message-passing library for C/C++/Fortran
- De facto standard for communication among processes running on a distributed memory system
- Message passing between processes
- Notoriously difficult to debug!!!
- Low-level of abstraction
- Portable
- Linux users! install *mpi-default-dev* package



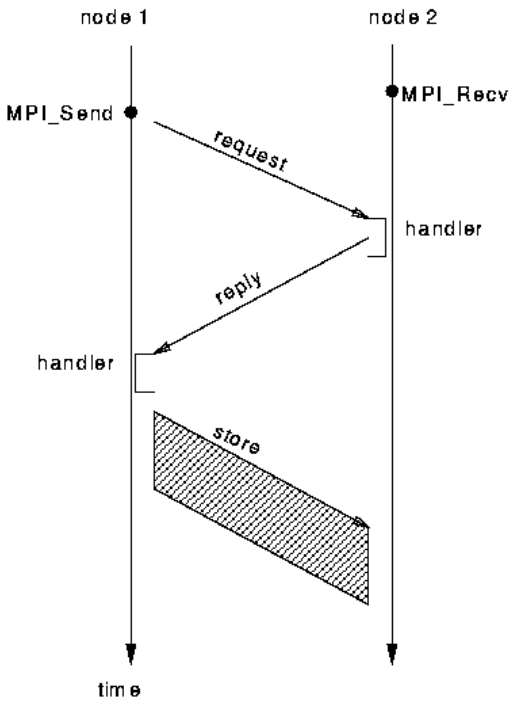


Some definitions

- Distribution of the work and data into multiple distributed programs are based on the **rank** (identification number)
- **Group**: set of processes that communicate with one another
- **Context**: prevents an operation on one communicator from matching with a similar operation on another communicator
- **Communicator**: object that specifies the scope of a communication operation, i.e., the group of processes involved and the context. `MPI_COMM_WORLD` is the default communicator.

Message Passing

- Messages among processes are handled via send/receive calls
- Communication modes:



- **MPI_Send**. Standard send. Completes when message is sent (receive state unknown)
- **MPI_Recv**. Receives a message and blocks until a message has arrived.
- **MPI_Ssend**. Synchronous blocking send. Completes when the message has been received by the other process.
- Many other non-blocking , buffered, and asynchronous modes

MPI Tags

- Messages are sent with an accompanying user-defined integer TAG, to assist the receiving progress in identifying the message
- Messages can be screened at the receiving end by specifying a tag, or not screened by specifying *MPI_ANY_TAG* as in tag in a receive

MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype) where the datatype matches the corresponding data type from the language, e.g.:

MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_FLOAT, MPI_DOUBLE, etc

MPI SEND

MPI_Send(start, count, datatype, destination, tag, comm)

- start: memory position where the message begins
- count: number of elements in the message
- datatype: type of data, MPI::CHAR, MPI::FLOAT, etc
- destination: rank of the target process in the communicator
- tag: sequence number
- comm: object communicator, default MPI_COMM_WORLD

- Important! MPI_Send does not wait until message is received

MPI RECEIVE

MPI_Recv(start, count, datatype, source, tag, comm, status)

Waits until a matching (source **and** tag) message is received

- start: memory position where the message is stored
- source: rank of the sending process or *MPI_ANY_SOURCE*
- tag: sequence number or *MPI_ANY_TAG*
- status: contains additional information
- Receiving fewer than count occurrences of datatype is OK, but receiving more produces an error

Hello World! MPI Context initialization

- Compile with `mpicc hello.c -o hello` (mpicxx for C++)
- Execute with `mpiexec -np 8 hello` (-np number processes)

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

void main (int argc, char * argv[])
{
  int rank, size;

  /* Start MPI */
  MPI_Init (&argc, &argv);
  /* Get current process id */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  // MPI_COMM_WORLD is default communicator
  /* Get number of processes */
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  printf("Hello world from process %d of %d\n", rank, size);
  MPI_Finalize();
}
```

Hello World! (for real) the ping-pong example

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

void 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);
  }
   else if (rank == 1) {    // task 1 waits for task 0 message then returns a message
     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();
}
```

MPI_Status

- Status is a data structure allocated in the user's program.
- Allows to retrieve the tag, source and count of the message.

```
int recvd_tag, recvd_from, recvd_count;
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &recvd_count);
```

Synchronous blocking

- MPI_Send and MPI_Recv are blocking, which means they will wait until the message has been sent and received, respectively
- MPI_Ssend also awaits for the confirmation message is received

Nonblocking message passing

- Nonblocking sends return no matter the message is received
- Messages are stored in a buffer
- MPI_Isend() and MPI_Irecv() are asynchronous nonblocking calls
- Program continues as soon as they are called, a communication request handle is returned for handling pending messages
- Data in the send buffer subject to change before actually sent!!
- MPI_Wait() or MPI_Test() to determine when message is received

Nonblocking message passing API

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

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

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

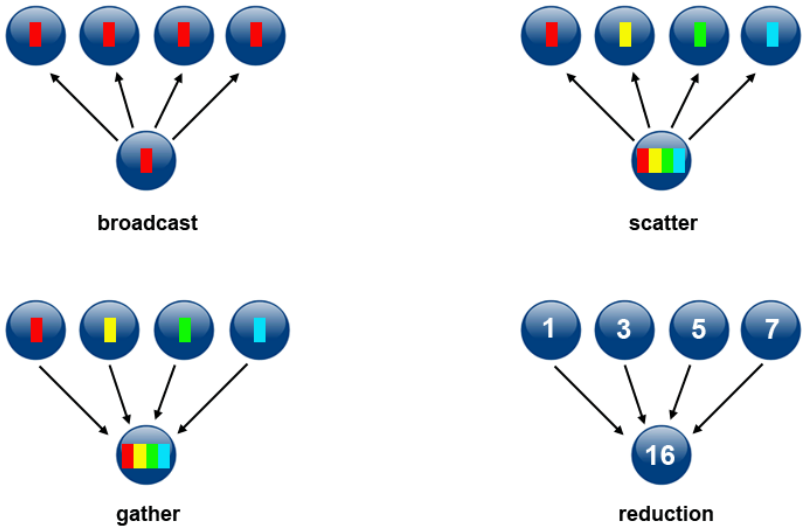
```
MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Extended functions

```
MPI_Testany, MPI_Testall, MPI_Testsome
MPI_Waitany, MPI_Waitall, MPI_Waitsome
```

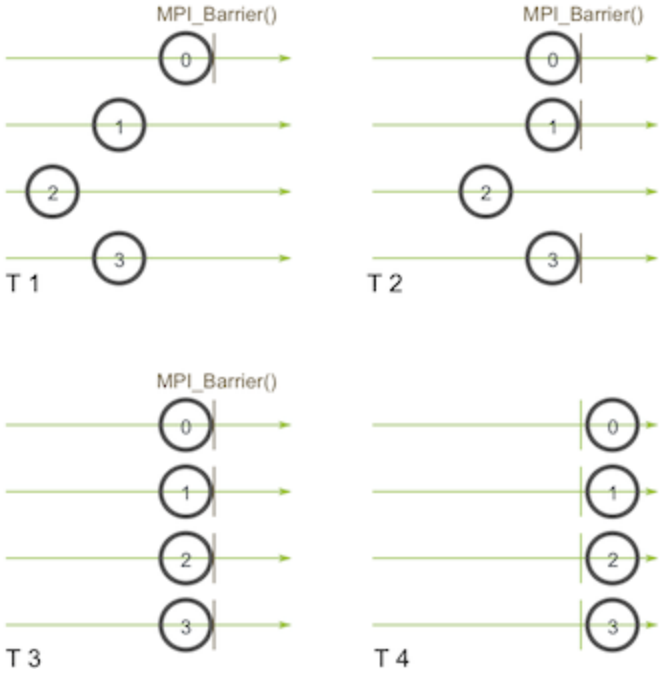
Collective Communication Routines

- Synchronization: processes wait until all members of the group have reached the synchronization point.
- Data Movement: broadcast, scatter/gather, all to all.
- Collective Computation (reductions): one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.



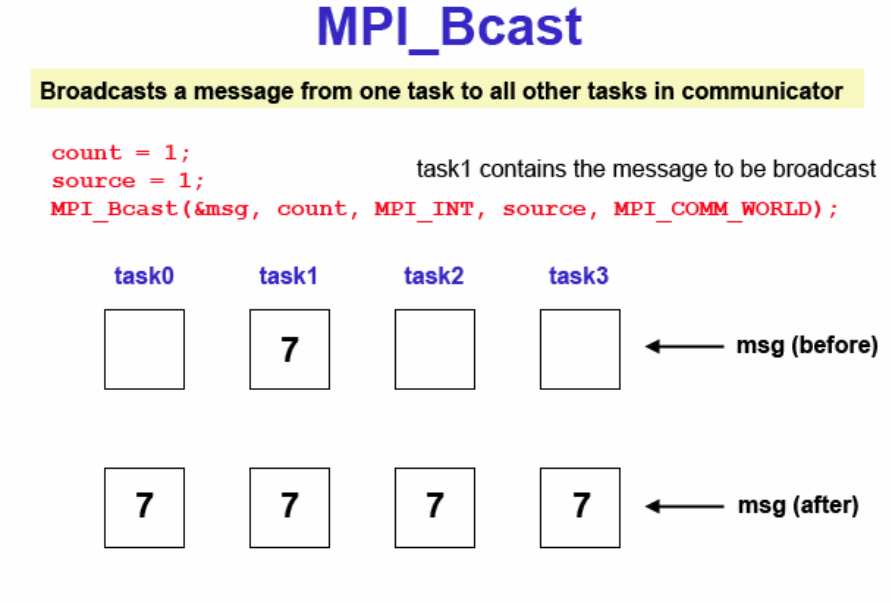
Barrier

- MPI_Barrier (comm) creates a barrier synchronization in a group



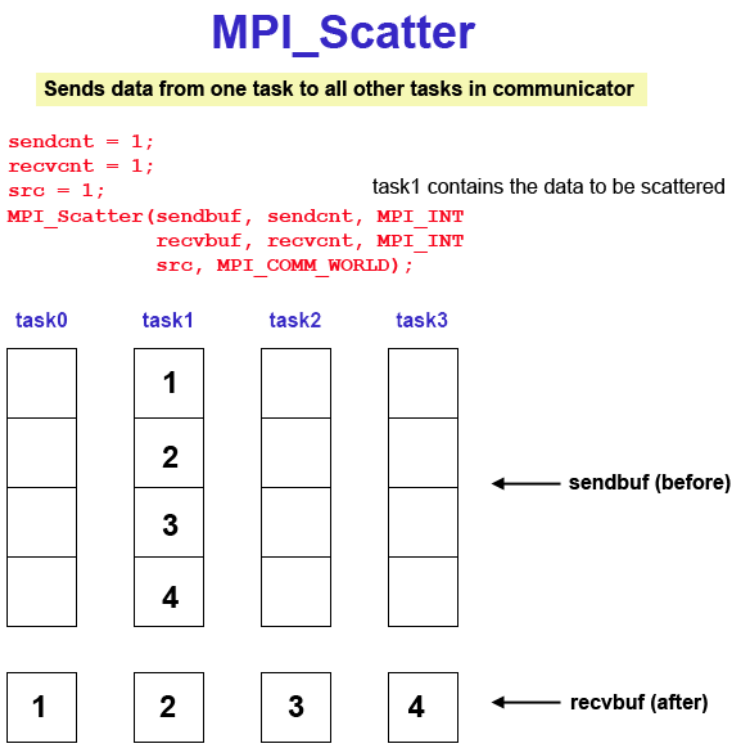
Broadcast

- MPI_Bcast (&buffer,count,datatype,rank,comm)
- Sends a message to all the other processes in the group
- The buffer will contain the message in every process



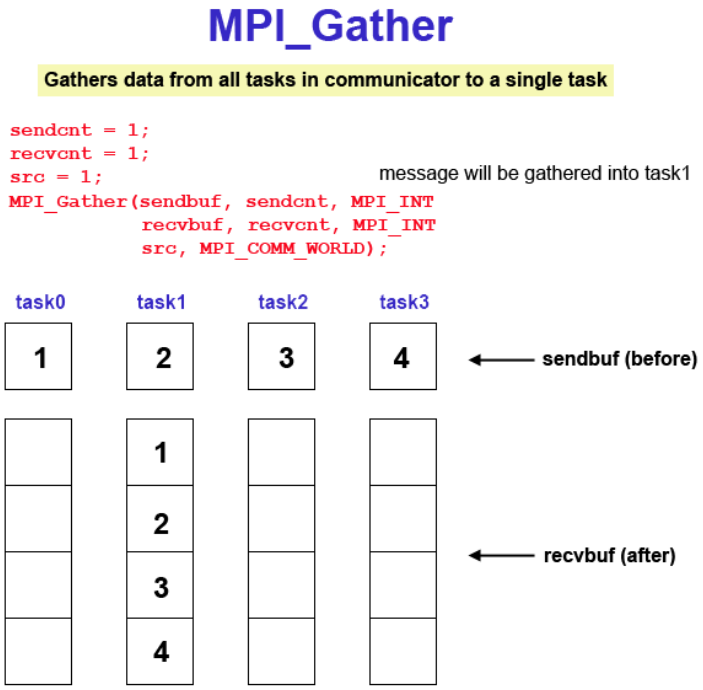
Scatter

- MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf, recvcnt,recvtype,root,comm)
- Distributes distinct messages to each task in the group



Gather

- MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf, recvcount,recvtype,root,comm)
- Collects distinct messages from each task in the group to a single destination task
- MPI_Gatherv to control explicitly the locations



Reduce

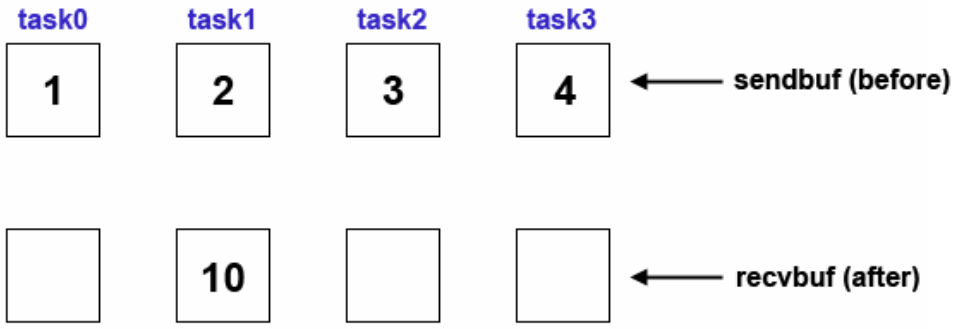
- MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
- Collects and applies a reduction operation on all tasks in the group and places the result in one task

MPI_Reduce

Perform reduction across all tasks in communicator and store result in 1 task

```
count = 1;
dest = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT,
           MPI_SUM, dest, MPI_COMM_WORLD);
```

task1 will contain result



CMSC 691
High Performance Distributed Systems

Message Passing Interface

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu