Cornell University
Center for Advanced Computing

Introduction to MPI

Steve Lantz
Senior Research Associate
Cornell CAC

Workshop: Introduction to Parallel Computing on Ranger, May 28, 2009

Based on materials developed by Luke Wilson and Byoung-Do Kim at TACC

Outline of presentation

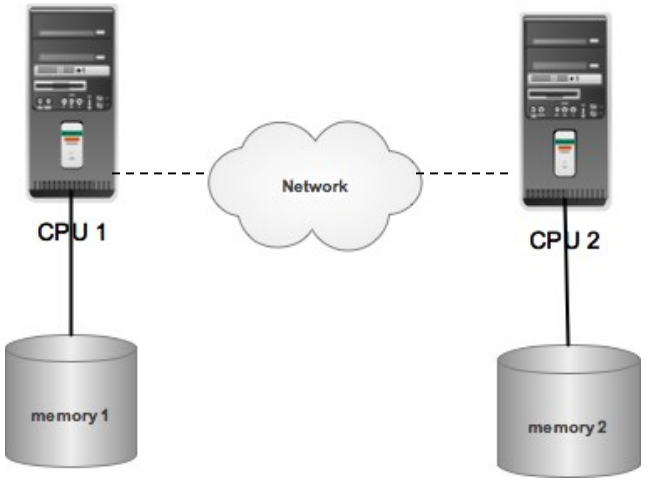
- Overview of message passing
- MPI: what is it and why should you learn it?
- Compiling and running MPI programs
- MPI API
  - Point-to-point communication
  - Collective communication and computation
- MPI references and documentation

Message passing overview

- What is message passing?
  - Sending and receiving messages between tasks or processes
  - Capabilities can include performing operations on data in transit and synchronizing tasks
- Memory model: *distributed*
  - Each process has its own address space and no way to get at another's, so it is necessary to send/receive data
- Programming model: *API*
  - Programmer makes use of an Application Programming *Interface* (API) that specifies the functionality of high-level communication routines
  - Functions give access to a low-level *implementation* that takes care of sockets, buffering, data copying, message routing, etc.

An API for distributed memory parallelism

- Assumption: processes do not see each other's memory
- Communication speed is determined by some kind of network
  - Typical network = switch + cables + adapters + software stack…
- Key: the implementation of a message passing API (like MPI) can be optimized for any given network
  - Program gets the benefit
  - No code changes required
  - Works in shared memory, too

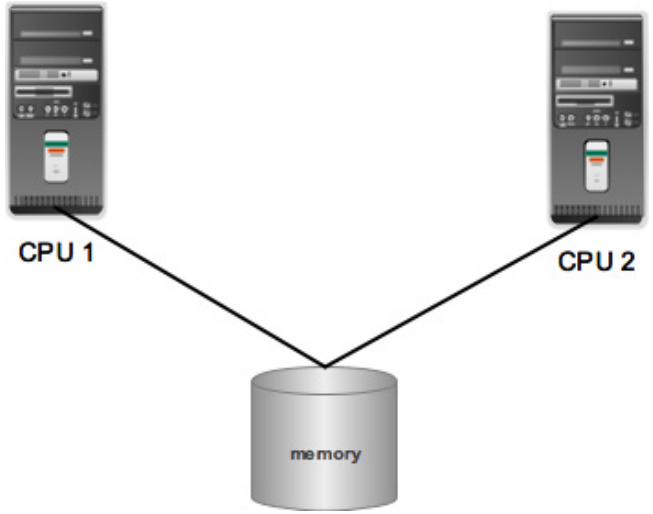


Pros and cons of the distributed memory model

- Advantages
  - Parallelism in an application is explicitly identified (*not* a disadvantage!)
  - Potential to scale very well to large numbers of processors
  - Avoids problems associated with shared memory: e.g., no interference or overhead due to maintaining cache coherency
  - Cost-effective: can use commodity, off-the-shelf processors and networking hardware
- Disadvantages
  - The programmer is responsible for controlling the data movement between processes, plus many associated details
  - NUMA (Non-Uniform Memory Access: true of shared memory, too)
  - It may be difficult to map an application's global data structures and/or data access patterns to this memory model

Contrast with shared memory parallelism

- Assumption: processes have access to the *same* memory
  - As usual, the compiler's job is to translate program variables into virtual memory addresses, which are global
  - Therefore, the compiler itself can potentially be used to parallelize code, perhaps with no need for a special API…



Pros and cons of the shared memory model

- Advantages
  - Programmer no longer needs to specify explicit communication of data between tasks
  - Tasks “communicate” via a common address space, into which they read and write asynchronously
- Disadvantages
  - Understanding performance and managing data locality become more difficult (the downside of giving up explicit control!)
  - Actual shared memory is usually limited to relatively few processors
  - Much harder to implement a shared memory model on a distributed memory machine, compared to the other way around!

Alternatives to MPI using a shared memory model

- Multithreading (useful for actual shared memory only)
  - OpenMP compiler directives
  - Pthreads = POSIX threads, and similar APIs
  - “The medium (i.e., memory) is the message”
- PGAS = Partitioned Global Address Space languages/extensions
  - Make physically distributed memory *appear* to be shared memory
  - UPC = Unified Parallel C
  - Co-Array Fortran (due to be included in next Fortran standard)
  - Fortress
- Higher-level Libraries/APIs
  - Global Arrays from PNNL
- Hybrids of the above with MPI message passing are possible

MPI-1

- MPI-1 - Message Passing Interface (v. 1.2)
  - Library standard defined by committee of vendors, implementers, and parallel programmers
  - Used to create parallel SPMD codes based on explicit message passing
- Available on almost all parallel machines with C/C++ and Fortran bindings (and occasionally with other bindings)
- About 125 routines, total
  - 6 basic routines
  - The rest include routines of increasing generality and specificity

MPI-2

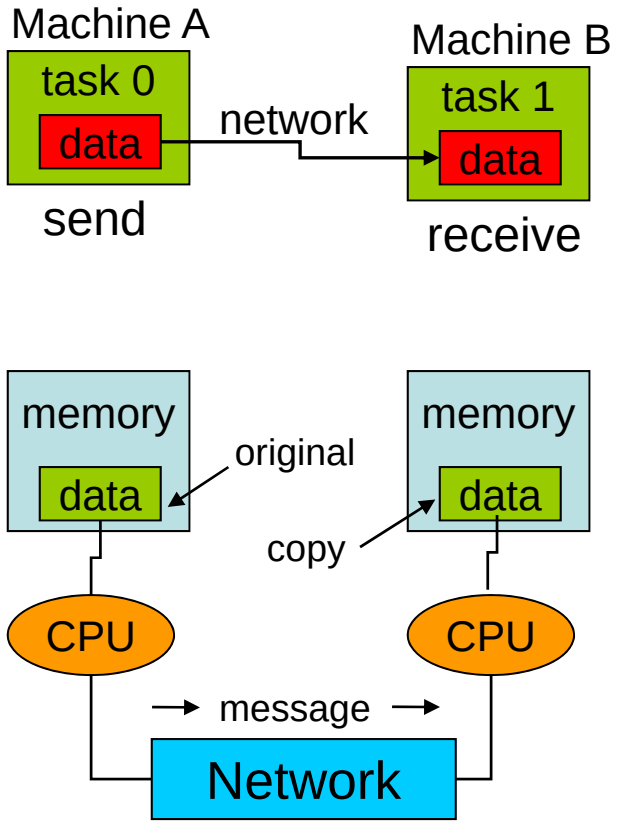
- Includes features left out of MPI-1
  - One-sided communications
  - Dynamic process control
  - More complicated collectives
  - Parallel I/O (MPI-IO)
- Implementations came along only gradually
  - Not quickly undertaken after the reference document was released (in 1997)
  - Now OpenMPI, MPICH2 (and its descendants), and the vendor implemenations are nearly complete or fully complete
- Most applications still rely on MPI-1, plus maybe MPI-IO

Why learn MPI?

- MPI is a de facto standard
  - Public domain versions are easy to install
  - Vendor-optimized version are available on most hardware
- MPI is “tried and true”
  - MPI-1 was released in 1994, MPI-2 in 1996
- MPI applications can be fairly portable
- MPI is a good way to learn parallel programming
- MPI is expressive: it can be used for many different models of computation, therefore can be used with many different applications
- MPI code is efficient (though some think of it as the “assembly language of parallel processing”)

Message passing with MPI

- Typically use SPMD-style coding: Single Program, Multiple Data
  - Each process will run a copy of the same code, but with different data
- Embed calls to MPI functions or subroutines in the source code
  - Data transfer is usually cooperative; both sender and receiver call an MPI function (see figure)
- Link the appropriate MPI library to the compiled application
- Run using “mpiexec” or equivalent



Compiling MPI programs

- Generally use a special compiler or compiler wrapper script
  - Not defined by the standard
  - Consult your implementation
  - Correctly handles include path, library path, and libraries
- MPICH-style (the most common)

  `mpicc -o foo foo.c`

  `mpif90 -o foo foo.f` (also mpif77)
- Some MPI specific compiler options

  `-mpilog` -- Generate log files of MPI calls

  `-mpitrace` -- Trace execution of MPI calls

  `-mpianim` -- Real-time animation of MPI (not available on all systems)
- Note: compiler/linker names are specific to MPICH. On IBM Power systems, they are *mpcc_r* and *mpxlf_r* respectively

Running MPI programs

- To run a simple MPI program using MPICH

```
mpirun -np 2 ./foo
mpiexec -np 2 ./foo
```

- Some MPI specific running options
  - `-t` -- shows the commands that *mpirun* would execute
  - `-help` -- shows all options for *mpirun*
- To run over Ranger's InfiniBand (as part of an SGE script)

```
ibrun ./foo
```

  - The scheduler handles the rest
- Note: *mpirun* and *mpiexec* are not part of MPI, but a similar command can be found in nearly all implementations
  - There are exceptions: on the IBM SP, for example, it is *poe*

Basic MPI

- It is possible to parallelize an entire application with just a few MPI functions
  - Initialization and termination
  - Point-to-point communication
  - Maybe a couple of types of collective communication/computation
- In principle this subset is enough for many applications
- However, “advanced” MPI functions can be more efficient and easier to use in the situations for which they were designed

Initialization and termination

- All processes must initialize and finalize MPI
  - These are collective calls
- All processes must include the MPI header file
  - Provides basic MPI definitions and types
  - Implementation-specific, so don't copy these from system to system

```
#include <mpi.h>
main(int argc char**&argv){
int ierr;
ierr = MPI_Init(&argc, &argv);
                    :
ierr = MPI_Finalize();
}
```

```
program init_finalize
include 'mpif.h'
integer ierr
call mpi_init(ierr)
          :
call mpi_finalize(ierr)
end program
```

Fortran and C differences…

- Header files

Fortran include file	C include file
include 'mpif.h'	#include "mpi.h"

  – Optionally, in Fortran 90/95, one can compile an mpif.f90 file to create the MPI module, then "use MPI" in the calling scope

- Format of MPI calls

Fortran Binding	C Binding
CALL MPI_XXX(parameters,…,ierr)	rc = MPI_Xxxx(parameters,…)

MPI communicators

- Communicators
  - Collections of processes that can communicate with each other
  - Most MPI routines require a communicator as an argument
  - Predefined communicator MPI_COMM_WORLD encompasses all tasks
  - New communicators can be defined; any number can co-exist
- Each communicator must be able to answer two questions
  - *How many processes exist in this communicator?*
  - MPI_Comm_size returns the answer, say, N_p
  - *Of these processes, which process (numerical rank) am I?*
  - MPI_Comm_rank returns the rank of the current process within the communicator, an integer between 0 and N_p-1 inclusive
  - Typically these functions are called just after MPI_Init

MPI_COMM_WORLD: C example

```
#include <mpi.h>
main(int argc, char **argv){
   int np, mype, ierr;

   ierr = MPI_Init(&argc, &argv);
   ierr = MPI_Comm_size(MPI_COMM_WORLD, &np);
   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &mype);
            :
   MPI_Finalize();
}
```

MPI_COMM_WORLD: C++ example

```
#include “mpif.h”
[other includes]
int main(int argc, char *argv[]){
   int np,  mype,  ierr;
 [other declarations]
              :
            MPI::Init(argc, argv);
   np   = MPI::COMM_WORLD.Get_size();
   mype = MPI::COMM_WORLD.Get_rank();
              :
          [actual work goes here]
              :
            MPI::Finalize();
}
```

MPI_COMM_WORLD: Fortran example

```
program param
  include 'mpif.h'
  integer ierr, np, mype

  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD, np  , ierr)
  call mpi_comm_rank(MPI_COMM_WORLD, mype, ierr)
            :
  call mpi_finalize(ierr)
end program
```

How size and rank are used during MPI execution

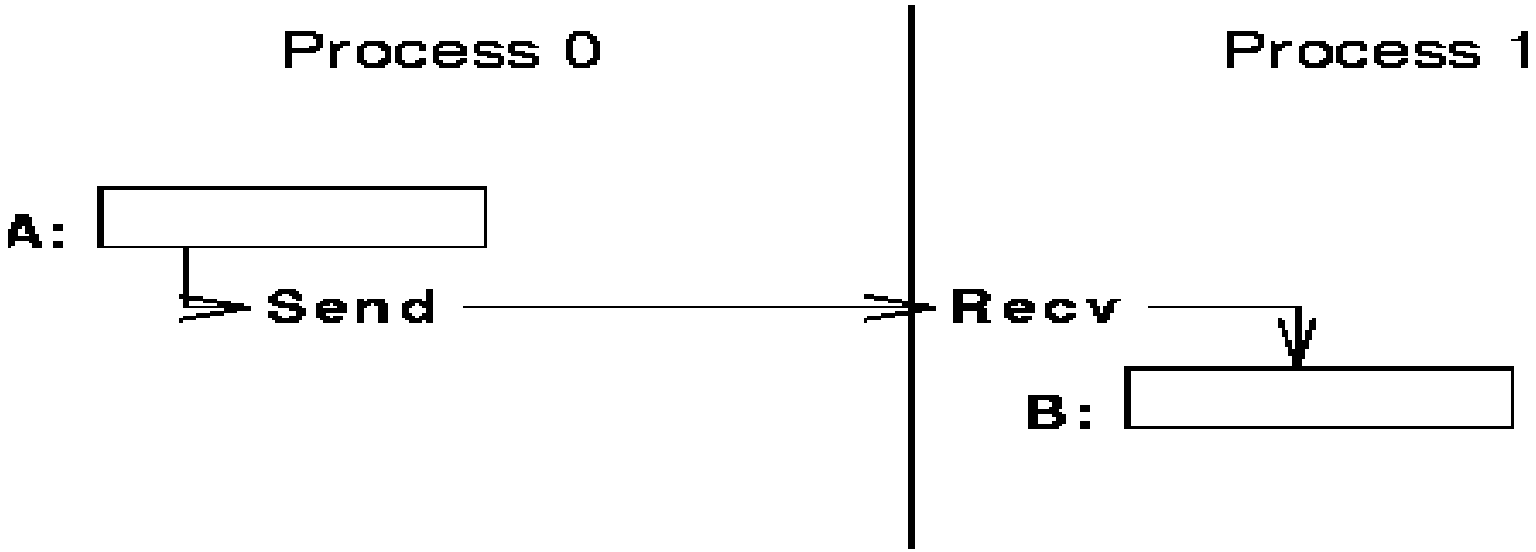
- Typically, every process will be an *exact duplicate* of the same MPI executable: ***Single Program, Multiple Data*** (SPMD).
- However, the *runtime environment* of each process is *not identical*; it includes an environment variable that holds the unique rank of that particular process within MPI_COMM_WORLD
- Each process can therefore check its own rank to determine which part of the problem to work on
- Once execution starts, processes work ***completely independently*** of each other, except when communicating

Topics in point-to-point communication

- MPI_SEND and MPI_RECV
- Synchronous vs. buffered (asynchronous) communication
- Blocking send and receive
- Non-blocking send and receive
- Combined send/receive
- Deadlock, and how to avoid it

Point-to-point communication

- Sending data from one point (process/task) to another point (process/task)
- One task sends while another recives



MPI_Send and MPI_Recv

- MPI_Send(): A blocking call which returns only when data has been sent from its buffer
- MPI_Recv(): A blocking receive which returns only when data has been received onto its buffer

```
MPI_Send (data, count, type, dest, tag, comm)
MPI_Recv (data, count, type, src,  tag, comm, status)
```

```
mpi_send (data, count, type, dest, tag, comm, ierr)
mpi_recv (data, count, type, src,  tag, comm, status, ierr)
```

An MPI message travels in an envelope

```
MPI_Send (data, count, type, dest, tag, comm)
MPI_Recv (data, count, type, src,  tag, comm, status)
```

- When MPI sends a message, it doesn't just send the contents; it also sends an "envelope" describing the contents
  - *void* data*: actual data being passed (via pointer to first element)
  - *int count*: number of *type* values in *data*
  - *MPI_Datatype type*: type of *data*
  - *int dest/src:* rank of the receiving/sending process
  - *int tag*: simple identifier that must match between sender/receiver
  - *MPI_Comm comm*: communicator (must match – no wildcards)
  - *MPI_Status* status*: returns information on the message received

Notes on the MPI envelope

```
mpi_send (data, count, type, dest, tag, comm, ierr)
mpi_recv (data, count, type, src,  tag, comm, status, ierr)
```

- A few Fortran particulars
  - All Fortran arguments are passed by reference
  - *INTEGER ierr:* variable to store the error code (in C/C++ this is the return value of the function call)
- Wildcards are allowed
  - *src* can be the wildcard MPI_ANY_SOURCE
  - *tag* can be the wildcard MPI_ANY_TAG
  - *status* returns information on the source and tag, useful in conjunction with the above wildcards (receiving only)

Assigning roles in point-to-point code

- Recall that all tasks execute the same code
- Thus, conditionals based on communicator rank are often needed
- Tags must match on sender and receiver for a message to succeed

```
MPI_Comm_rank(comm,&mytid);

if (mytid==0) {
  MPI_Send (buffer_A, /* target= */ 1, /* tag= */ 0, comm);
} else if (mytid==1) {
  MPI_Recv( buffer_B, /* source= */ 2, /* tag= */ 6, comm);
}
```

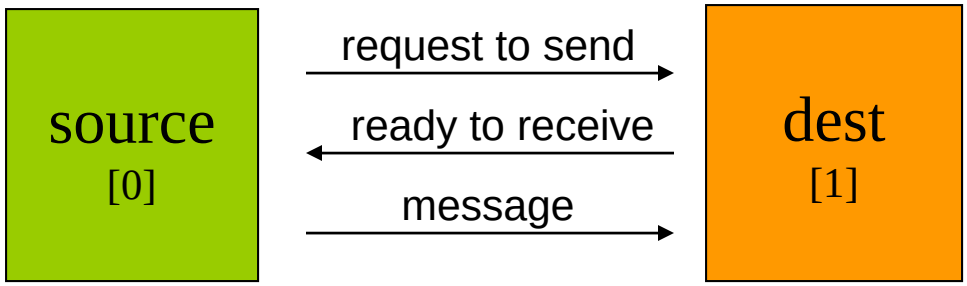
Complete point-to-point code: C example

```
#include  "mpi.h"
main(int argc, char **argv){
int ipe, ierr; double a[2];
MPI_Status status;
MPI_Comm icomm = MPI_COMM_WORLD;
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_rank(icomm, &ipe);
ierr = MPI_Comm_size(icomm, &myworld);
if(ipe == 0){
    a[0] = mype; a[1] = mype+1;
    ierr = MPI_Send(a,2,MPI_DOUBLE, 1,9, icomm);
  }
else if (ipe == 1){
    ierr = MPI_Recv(a,2,MPI_DOUBLE, 0,9,icomm,&status);
    printf("PE %d, A array= %f %f\n",mype,a[0],a[1]);
  }
  MPI_Finalize();
}
```

Complete point-to-point code: Fortran example

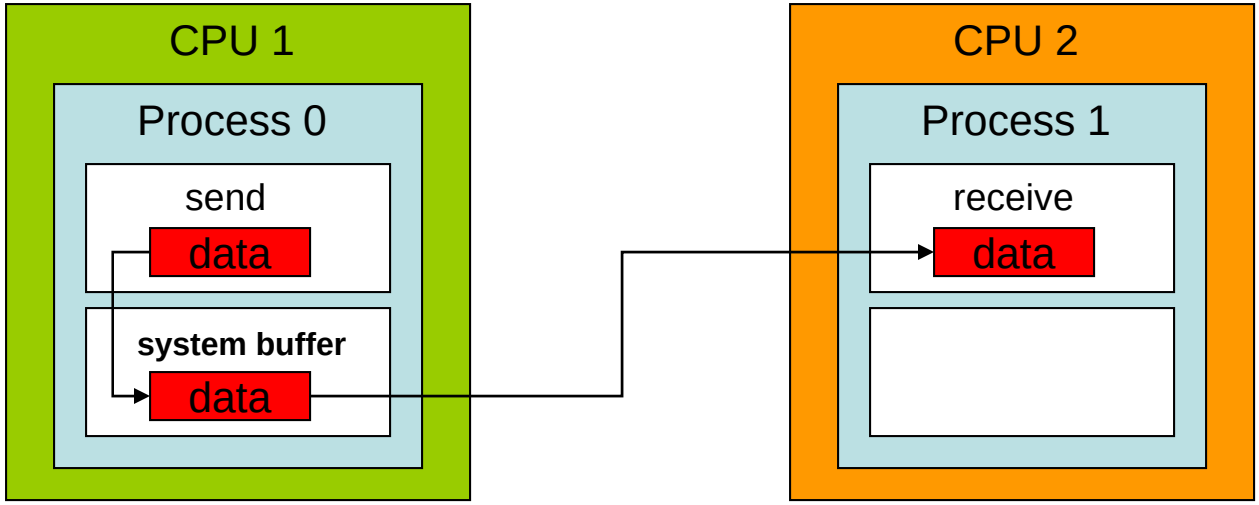
```
program sr
   include "mpif.h"
   real*8,  dimension(2)                :: A
   integer, dimension(MPI_STATUS_SIZE) :: istat
   icomm = MPI_COMM_WORLD
   call mpi_init(ierr)
   call mpi_comm_rank(icomm,mype,ierr)
   call mpi_comm_size(icomm,np  ,ierr);

   if(mype.eq.0) then
    a(1) = real(ipe); a(2) = real(ipe+1)
    call mpi_send(A,2,MPI_REAL8, 1,9,icomm, ierr)
   else if (mype.eq.1) then
    call mpi_recv(A,2,MPI_REAL8, 0,9,icomm, istat,ierr)
    print*,"PE ",mype,"received A array =",A
   endif

   call mpi_finalize(ierr)
end program
```

Synchronous send, MPI_Ssend



- Process 0 waits until process 1 is ready
- “Handshake” occurs to confirm a safe send
- Blocking send on 0 takes place along with a blocking receive on 1
- Rarely useful in the real world
  - Need to be able to proceed when multiple tasks are out of sync
  - Better to copy to a temporary buffer somewhere so tasks can move on

Buffered send, MPI_Bsend



- Message contents are sent to a system-controlled block of memory
- Process 0 continues executing other tasks; when process 1 is ready to receive, the system simply copies the message from the system buffer into the appropriate memory location controlled by process
- Must be preceded with a call to MPI_Buffer_attach

Blocking vs. non-blocking communication

- Blocking
  - A blocking routine will only return when it is safe to use the buffer again
  - On the sender, “safe” means only that modification will not affect the data to be sent, and it does not imply that the data was actually received
  - A blocking call can be either *synchronous* or *asynchronous* (buffered)
- Non-blocking
  - Non-blocking send and receive routines are simply requests; they return immediately without waiting for the communication events to complete
  - It is therefore unsafe to modify the buffer until you know the requested operation has completed: pair each non-blocking operation with an MPI_Wait to make sure (this will also clear the request handle)
  - The aim of non-blocking calls is to overlap computation with communication for possible performance gains

Blocking and non-blocking routines

Blocking send	**MPI_Send(buf, count, datatype, dest, tag, comm)**
Non-blocking send	**MPI_Isend(buf, count, datatype, dest, tag, comm, request)**
Blocking receive	**MPI_Recv(buf, count, datatype, source, tag, comm, status)**
Non-blocking receive	**MPI_Irecv(buf, count, datatype, source, tag, comm, request)**

Notes

1. request: unique handle passed to a non-blocking send or receive operation
2. MPI_Wait blocks until a specified non-blocking send or receive operation has completed: **MPI_Wait(request, status)**
3. Buffered and synchronous calls can be non-blocking: Ibsend, Issend

MPI_Sendrecv

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,
recvbuf, recvcount, recvtype, source, recvtag, comm,
status)
```

- Useful for communication patterns where each of a pair of nodes both sends and receives a message (two-way communication).
- Executes a blocking send and a blocking receive operation
- Both operations use the same communicator, but have distinct tag arguments

One-way blocking/non-blocking combinations

- Blocking send, blocking recv

```
IF (rank==0) THEN
   CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
   CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

- Non-blocking send, blocking recv

```
IF (rank==0) THEN
   CALL MPI_ISEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ie)
   CALL MPI_WAIT(req,status,ie)
ELSEIF (rank==1) THEN
   CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

More one-way blocking/non-blocking combos

- Blocking send, non-blocking recv

```
IF (rank==0) THEN
   CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
   CALL MPI_IRECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
   CALL MPI_WAIT(req,status,ie)
ENDIF
```

- Non-blocking send, non-blocking recv

```
IF (rank==0) THEN
   CALL MPI_ISEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ie)
ELSEIF (rank==1) THEN
   CALL MPI_IRECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
ENDIF
CALL MPI_WAIT(req,status,ie)
```

Two-way communication: deadlock!

- **Deadlock 1**

```
IF (rank==0) THEN
   CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
   CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
   CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
   CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
ENDIF
```

- **Deadlock 2**

```
IF (rank==0) THEN
   CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
   CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
   CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
   CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

Two-way communication: solutions

- Solution 1

```
IF (rank==0) THEN
   CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
   CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
   CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
   CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
ENDIF
```

- Solution 2

```
IF (rank==0) THEN
   CALL MPI_SENDRECV(sendbuf,count,MPI_REAL,1,tag, &
                     recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
   CALL MPI_SENDRECV(sendbuf,count,MPI_REAL,0,tag, &
                     recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

Two-way communication: more solutions

- Solution 3

```
IF (rank==0) THEN
   CALL MPI_IRECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ie)
   CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
   CALL MPI_IRECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
   CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
ENDIF
CALL MPI_WAIT(req,status)
```

- Solution 4

```
IF (rank==0) THEN
   CALL MPI_BSEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
   CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
   CALL MPI_BSEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
   CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

Two-way communications: summary

	CPU 0	CPU 1
Deadlock1	Recv/Send	Recv/Send
Deadlock2	Send/Recv	Send/Recv
Solution1	Send/Recv	Recv/Send
Solution2	SendRecv	SendRecv
Solution3	IRecv/Send, Wait	IRecv/Send, Wait
Solution4	BSend/Recv	BSend/Recv

Need for collective communication: broadcast

- What if one processor wants to send to everyone else?

```
if (mytid == 0 ) {
  for (tid=1; tid<ntids; tid++) {
    MPI_Send( (void*)a, /* target= */ tid, … );
  }
} else {
  MPI_Recv( (void*)a, 0, … );
}
```

- Implements a very naive, serial broadcast
- Too primitive
  - leaves no room for the OS / switch to optimize
  - leaves no room for more efficient algorithms
- Too slow: most receive calls will have a long wait for completion

MPI collective communications

- Involve ALL processes within a communicator
- There are three basic types of collective communications:
  - Synchronization (MPI_Barrier)
  - Data movement (MPI_Bcast/Scatter/Gather/Allgather/AlltoAll)
  - Collective computation (MPI_Reduce/Allreduce/Scan)
- Programming considerations & restrictions
  - Blocking operation
  - No use of message tag argument
  - Collective operation within subsets of processes require separate grouping and new communicator
  - Can only be used with MPI predefined datatypes

Barrier synchronization and broadcast

- *Barrier* blocks until all processes in comm have called it
- Useful when measuring communication/computation time
  - **`mpi_barrier(comm, ierr)`**
  - **`MPI_Barrier(comm)`**

- *Broadcast* sends data from root to all processes in comm
  - **`mpi_bcast(data, count, type, root, comm, ierr)`**
  - **`MPI_Bcast(data, count, type, root, comm)`**

MPI_Scatter

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount,
            recvtype, root, comm)
```

IN	sendbuf	starting address of send buffer
IN	sendcount	number of elements sent to each process
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements in receive buffer
IN	recvtype	data type of receive buffer elements
IN	root	rank of sending process
IN	comm	communicator

- Distributes distinct messages from a single source task to each task in the communicator

MPI_Gather

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
           recvtype, root, comm)
```

IN	sendbuf	address of send buffer
IN	sendcount	number of elements in send buffer
IN	sendtype	data type of send buffer elements
OUT	recvbuf	starting address of receive buffer
IN	recvcount	number of elements for any single receive
IN	recvtype	data type of receive buffer elements
IN	root	rank of receiving process
IN	comm	communicator

- Gathers distinct messages from each task in the group to a single destination task
- Inverse operation of MPI_Scatter

MPI_Allgather

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
              recvtype, comm)
```

IN	sendbuf	address of send buffer
IN	sendcount	number of elements in send buffer
IN	sendtype	data type of send buffer elements
OUT	recvbuf	starting address of receive buffer
IN	recvcount	number of elements received from any process
IN	recvtype	data type of receive buffer elements
IN	comm	communicator

- Concatenation of data to all tasks in a group
- In effect, each task performs a broadcast operation to the other tasks in the communicator

MPI_Alltoall

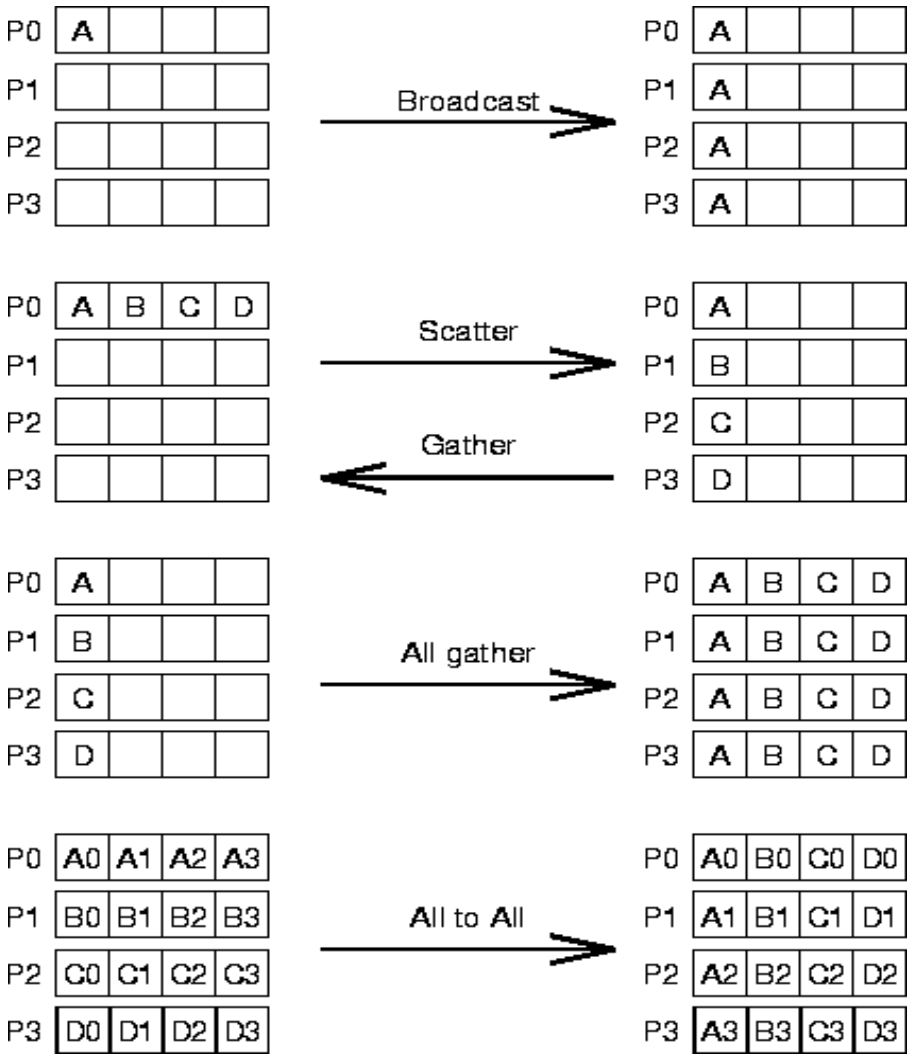
```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm)
```

IN	sendbuf	starting address of send buffer
IN	sendcount	number of elements sent to each process
IN	sendtype	data type of send buffer elements
OUT	recvbuf	starting address of receive buffer
IN	recvcount	number of elements received from any process
IN	recvtype	data type of receive buffer elements
IN	Comm	communicator

- Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index

Data movement…

- Broadcast
- Scatter/gather
- Allgather
- Alltoall



MPI_Reduce

MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)		
IN	sendbuf	address of send buffer
OUT	recvbuf	address of receive buffer
IN	count	number of elements in send buffer
IN	datatype	data type of elements of send buffer
IN	op	reduce operation
IN	root	rank of root process
IN	comm	communicator

- Applies a reduction operation on all tasks in the communicator and places the result in one task

MPI_Allreduce

MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)		
IN	sendbuf	address of send buffer
OUT	recvbuf	address of receive buffer
IN	count	number of elements in send buffer
IN	datatype	data type of elements of send buffer
IN	op	operation
IN	comm	communicator

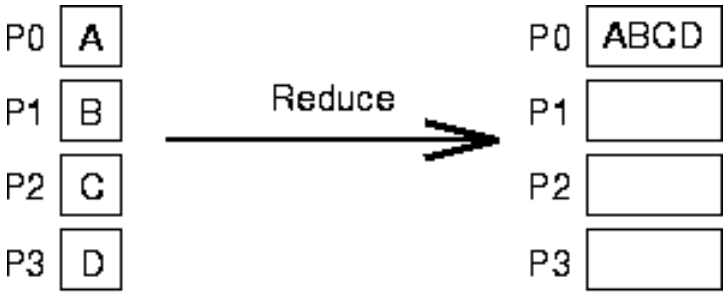
- Applies a reduction operation and places the result in all tasks in the communicator
- Equivalent to an MPI_Reduce followed by MPI_Bcast

Reduction operations

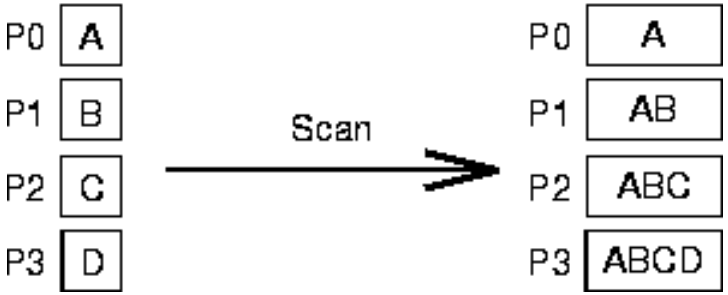
Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical xor
MPI_BXOR	Logical xor
MPI_MAXLOC	Max value and location
MPI_MINLOC	Min value and location

Collective computation patterns

- Reduce



- Scan



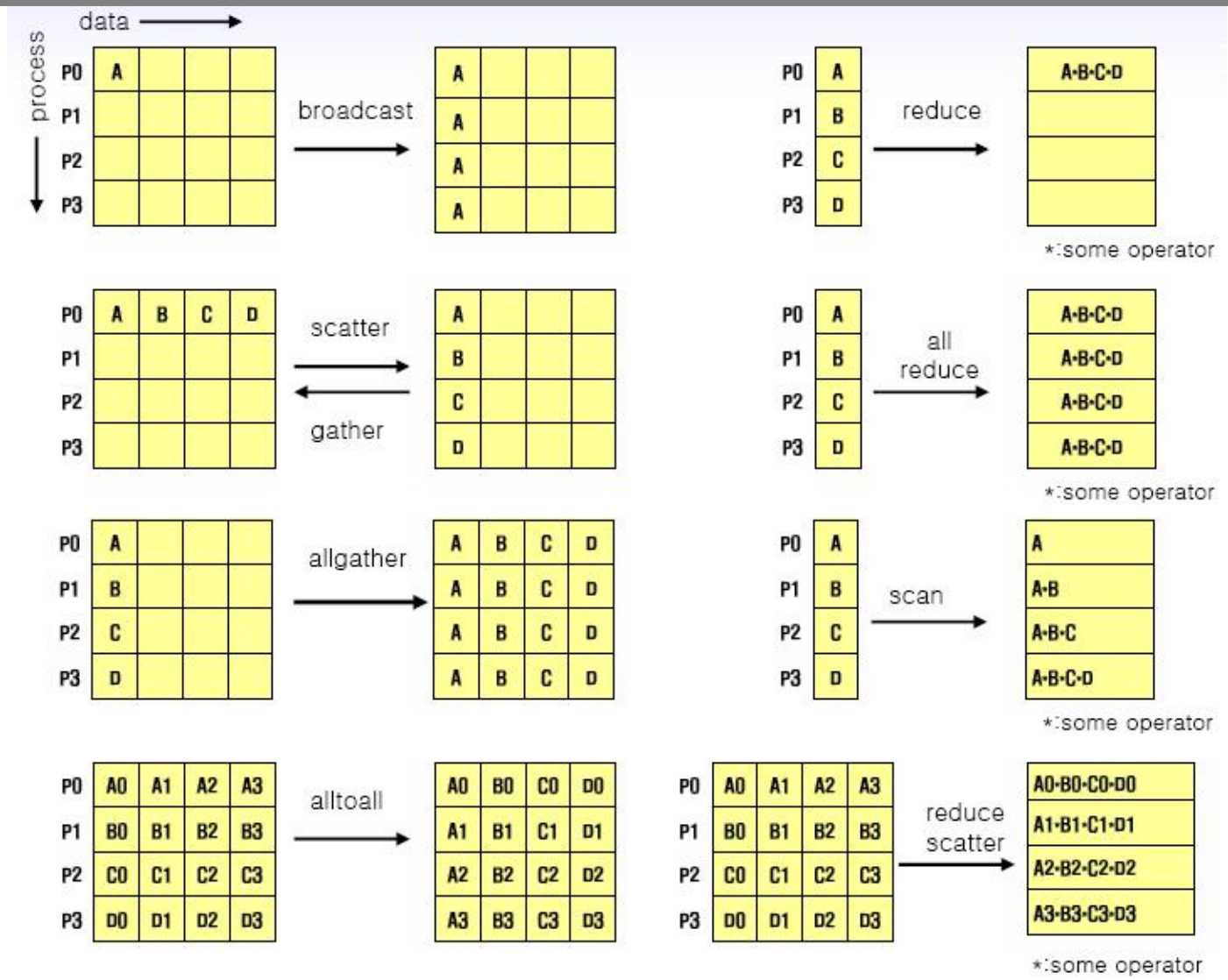
Collective Computation: C Example

```
#include <mpi.h>
#define WCOMM MPI_COMM_WORLD
main(int argc, char **argv){
  int npes, mype, ierr;
  double sum, val; int calc, knt=1;
  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_size(WCOMM, &npes);
  ierr = MPI_Comm_rank(WCOMM, &mype);

  val = (double) mype;

  ierr=MPI_Allreduce(&val,&sum,knt,MPI_DOUBLE,MPI_SUM,WCOMM);

  calc=(npes-1 +npes%2)*(npes/2);
  printf(" PE: %d sum=%5.0f calc=%d\n",mype,sum,calc);
  ierr = MPI_Finalize();
}
```

Collective Computation: Fortran Example

```
program sum2all
include 'mpif.h'

  icomm = MPI_COMM_WORLD
  knt = 1
  call mpi_init(ierr)
  call mpi_comm_rank(icomm,mype,ierr)
  call mpi_comm_size(icomm,npes,ierr)
  val = dble(mype)

  call mpi_allreduce(val,sum,knt,MPI_REAL8,MPI_SUM,icomm,ierr)

  ncalc=(npes-1 + mod(npes,2))*(npes/2)
  print*,' pe#, sum, calc. sum = ',mype,sum,ncalc
  call mpi_finalize(ierr)

end program
```

The MPI Collective Collection!



References

- MPI-1 and MPI-2 standards
  - http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
  - http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.htm
  - http://www.mcs.anl.gov/mpi/ (other mirror sites)
- Freely available implementations
  - MPICH, http://www.mcs.anl.gov/mpi/mpich
  - LAM-MPI, http://www.lam-mpi.org/
- Books
  - *Using MPI*, by Gropp, Lusk, and Skjellum
  - *MPI Annotated Reference Manual,* by Marc Snir, *et al*
  - *Parallel Programming with MPI*, by Peter Pacheco
  - *Using MPI-2*, by Gropp, Lusk and Thakur
- Newsgroup: comp.parallel.mpi