# Distributed Memory Programming with MPI

Wei Wang

# MPI: the Message Passing Interface

- Standard library for message-passing
  - Portable
    - There are several MPI implementations, we will use MPICH for this class
  - almost ubiquitously available
  - high performance
  - C and Fortran APIs, also available in Java, Python, and R
- MPI standard defines
  - syntax of library routines (APIs)
  - semantics of library routines (APIs)
- Details
  - MPI routines, data-types, and constants are prefixed by “MPI_”
- Simple to get started
  - fully-functional programs using only six library routines

# Scope of the MPI Standards

- Communication contexts
- Datatypes
- Point-to-point communication
- Collective communication (synchronous, non-blocking)
- Process groups
- Process topologies
- Environmental management and inquiry
- The Info object
- Process creation and management
- One-sided communication (refined for MPI-3)
- External interfaces
- Parallel I/O
- Language bindings for Fortran, C and C++
- Profiling interface (PMPI)

# What We Will Learn

- MPI management routines
  - Initialize and finalize MPI run-time
  - Get execution environment information
- Point-to-Point Communication Routines
  - Send and receive messages
- Collective Communication Routines
  - Broadcasting, gather/scatter, reduction
- Derived and customs data types
- Grouped communications
- Virtual Topologies

# Compiling MPI Programs

- Compile MPI program with the "*mpicc*" command
  - mpicc  mpi_program.c -o  mpi_executable
- *mpicc* is just a wrapper of another compiler with additional options to support MPI routines
  - "mpicc -show" gives you the actual compiler command
  - e.g., on fox servers, mpicc = "gcc -I/usr/lib/openmpi/include -I/usr/lib/openmpi/include/openmpi -pthread -L/usr//lib -L/usr/lib/openmpi/lib -lmpi -ldl -lhwloc"
  - The actual options may be different on each installation

# Executing MPI Programs

- MPI programs should be started with "*mpirun*" or "*mpiexec*" command
  - mpirun [ -np PE ] [ --hostfile <filename> ] <pgm>
    - -np PE: defines how many (PE) processes to use
    - -hostfile: defines the servers to use to execute. If no hostfile, then the job will only run on local server
      - For this class, use no hostfile
    - pgm: is the MPI program to execution, along with "pgm's" command line options
    - E.g., "mpirun -np 2 MPI_executable -cmdopt" executes "MPI_executable" with 2 processes and passes "cmdopt" to it as a command line option.
  - For most implementation, mpiexec is the same as mpirun

# MPI Management Routines

# MPI_Init

- Initializes the MPI execution environment.
- This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.
- If you have strange errors in your program, it may be that you do not call MPI_init at the beginning of your code.
  - The MPI Standard does not say what a program can do before an MPI_Init or after an MPI_Finalize. In the Open MPI implementation, it should do as little as possible. In particular, avoid anything that changes the external state of the program, such as opening files, reading standard input, or writing to standard output.

# MPI_Init cont'd

- Syntex:

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

- For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.
  - Most of the time,  you do not need to pass in any parameters
  - Also, note there is some difference between "MPI_Init" and standard C entry function "main"

# MPI_Comm_size

- Returns the total number of MPI processes in the specified communicator,
  - A MPI communicator defines a group of processes participated in certain communications.
  - The most common communicator MPI_COMM_WORLD. MPI_COMM_WORLD represents the communication in which all processes are participating.
  - If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

# MPI_Comm_size cont'd

- Syntax:

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

- Parameters:
  - Input parameter: *comm*, the MPI communicator
  - Output parameter: *size*, the number of processes participating in this communicator

# MPI_Comm_rank

- Returns the rank (id) of the calling MPI process within the specified communicator.
  - For *n* processes, the rank can be *0* to (*n-1)*
  - Similar to MPI_Comm_size, a communicator is required.
  - For a given process, if it participate in multiple communicators, this process may receive different ranks/ids in each communicator group.

# MPI_Comm_rank cont'd

- Syntax:

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

- Parameters:
  - Input parameter: *comm*, the MPI communicator
  - Output parameter: *rank*, the rank/id of the calling process

# MPI_Get_processor_name

- Returns the name of the processor on which the calling process is running
  - Note that, this function usually (and more accurately speaking) returns the host name  of the server on which the calling process is running
  - However, because implementation difference, some implementation of MPI may not return the host name.

# MPI_Get_processor_name cont'd

- Syntax:

```
int MPI_Get_processor_name( char *name,
                            int *resultlen )
```

- Parameters:
  - Output parameter: *name*, the string buffer that holds the processor name; make sure your buffer is long enough or it may cause buffer overflow
  - Output parameter: *resultlen*, the length of the returned processor name

# MPI_Finalize

- Terminates the MPI execution environment.
- This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.
- Syntax:

```
int MPI_Finalize( void )
```

# Code Example

- Example (MPI_management.c):

```
int main (int argc, char *argv[])
{
    int  proc_cnt, proc_id, len;
    char host_name[MPI_MAX_PROCESSOR_NAME];

    /* initialize MPI */
    MPI_Init(&argc, &argv);

    /* query execution environment */
    MPI_Comm_size(MPI_COMM_WORLD, &proc_cnt);
    MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);

    MPI_Get_processor_name(host_name, &len);

    printf ("Hello from process %d on %s!\n", proc_id, host_name);

    if(proc_id == 0)
        printf("MASTER: Number of MPI tasks is: %d\n", proc_cnt);

    /* cleanup */
    MPI_Finalize();
}
```

# Some Less-used Management Functions

- MPI_abort
  - Terminates all MPI processes associated with a communicator.
- MPI_Get_version
  - Returns the version and subversion of the MPI standard that's implemented by the library.
- MPI_Initialized
  - Check if MPI_init has been called
- MPI_Wtime
  - Returns an elapsed wall clock time in seconds (double precision) on the calling processor.
- MPI_Wtick
  - Returns the resolution in seconds (double precision) of MPI_Wtime.

# Point-to-Point Communications

# Point-to-Point Communication

- Communication between one process with another process
- Message-based communication (the M in MPI means “message”)
- Involves several types of send and receive functions
  - Asynchronous send (buffered) and synchronous sends (may also be buffered)
  - Blocking and non-blocking send/receive functions
  - Combined send/receives

# Basic Blocking Send

- Blocking: send function will not return until message is copied into buffer
  - Will buffered send cause dead-lock?
- Asynchronous: send function may return before the message is received by the receiving process
  - Note that the asynchronism is implementation dependent, i.e., in some implementation, MPI_send may be blocked until the message is received
- Function name: MPI_send

# Blocking Asynchronous Send cont'd

- Syntax:

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

- Parameters (all are input parameters):
  - buf: the buffer that holds the message
  - count: number of data elements in the message
  - datatype: the type of the data elements, MPI has several built in data types
  - dest: the rank/id of the receiving process
  - tag: tag of this message; usually used as sequence id of the message
  - comm: communicator, e.g., MPI_COMM_WORLD

# Blocking Asynchronous Send cont’d

- Some MPI built-in data types:
  - MPI_CHAR
  - MPI_SHORT
  - MPI_INT
  - MPI_LONG
  - MPI_LONG_LONG_INT
  - MPI_UNSIGNED
  - MPI_FLOAT
  - MPI_DOUBLE
  - MPI_LONG_DOUBLE
- A full list can be found at: https://computing.llnl.gov/tutorials/mpi/#Derived_Data_Types
- Custom types are also supported

# Basic Blocking Receive

- Blocking: receive function will not return until message is indeed receive
  - Will blocking receive cause dead-lock?
- Function: MPI_recv

# Basic Blocking Receive cont'd

- Syntax:

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

- Input parameters:
  - count: maximum number of data elements can be hold by buf
  - datatype: the type of the data elements
  - source: the rank/id of the sending process
  - tag: tag of the receiving message; must match the tag used in MPI_send
  - comm: communicator, e.g., MPI_COMM_WORLD
- Output parameters:
  - buf: the buffer that holds the received message
  - status: the status of the receive invocation

# Basic Blocking Receive cont’d

- Note on the tag:
  - If the sender process sends multiple messages at the same time (recall that send is asynchronous), each message should has an unique tag.
  - When invoking the MPI_recv, receiver process must specify which message it wants to receive by specifying the tag of the message
- Note on the order of send/receive code of two processes that send messages to each other
  - Because blocking receive can cause deadlock, it is important that these two processes do not get in to blocking receive at the same time

# Blocking Send and Receive Example

- Code example: MPI_block_comm.c

# Non-blocking Send

- Non-blocking: send returns almost immediately, it does not wait for any communication events to complete, including buffer copying.
- 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.
  - "wait" routines are provided to check if send has complete
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

# Non-blocking Send cont'd

- Syntax:

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

- Input parameters:
  - buf, count, datatype, dest, tag, and comm are the same as MPI_Send
- Output parameters
  - request: a handle of this request, used later to determine if send has finished

# Non-blocking Receive

- Non blocking: receive function returns immediately, even if the message has not been received
- Process must check later if the message has been received or not

# Non-blocking Receive cont'd

- Syntax:

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

- Except "request" all other parameters are the same as MPI_recv
- Output parameter:
  - request: a handle of this request, used later to determine if receive has finished

# Non-blocking Send/Receive: Checking Request Status

- MPI provides several APIs to wait on requests:
  - MPI_Wait: wait on a single requests, blocking wait
  - MPI_Waitall: wait on multiple requests, blocking wait
  - MPI_Test: tests if a request has finished, non-blocking

# MPI_Wait:

- Syntax:

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

- Input parameter:
  - request: the request handle of a non-blocking send or receive; the output parameter “request” of MPI_Isend or MPI_Irecv
- Output parameter:
  - status: the status of the request
- Note: when MPI_Wait returns, the request has finished, although it may succeed or fail.

# MPI_Waitall:

- Syntax:

```
int MPI_Waitall(int count,
                MPI_Request array_of_requests[],
                MPI_Status array_of_statuses[])
```

- Input parameter:
  - Count: the number of requests in “array_of_requests”
  - array_of_requests: array of handles of non-blocking sends or receives; may contain both send and receive requests at the same time
- Output parameter:
  - array_of_statuses: the array of the statuses of the requests
- Note: when MPI_Waitall returns, all requests have finished, although they may succeed or fail.

# MPI_Test:

- Syntax:

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

- Input parameter:
  - request: the request handle of a non-blocking send or receive; the output parameter “request” of MPI_Isend or MPI_Irecv
- Output parameter:
  - flag: true if operation completed
  - status: the status of the request
- Note: when MPI_Test returns, the request may not have finished

# Non-blocking Example

- Code example 1: MPI_nonblock_comm.c
- Code example 2: MPI_nonblock_comm_2.c

# Blocking vs Non-blocking Communication

- Blocking communication:
  - It is easier to code, there is no logic break in the send/receive process
  - Blocking may cause idling
- Non-block communication
  - More difficult to code; send/receive logic interleaves with other logics and computations
  - Non-blocking overlaps communication with computation, reduces idling

# MPI_Status

- MPI_Status is data structure (a C struct) holding information of request status.
- It has at least the following members:
  - MPI_SOURCE: rank/id of the sender process
  - MPI_TAG: tag of the message
  - MPI_ERROR: error status, such as,
    - MPI_SUCCESS: request is successful
    - MPI_ERR_RANK: invalid rank
    - MPI_ERR_TYPE: invalid data type

# Combined Send and Receive

- It is not uncommon that a process engaged in continuous bi-directional communications that it sends and receives messages back to back
- Coding bi-directional communication can be tricky due to the possibility of dead lock
- MPI provides API to send and receive messages simultaneously to simplify programming
  - The API uses blocking send and receive

# Combined Send and Receive cont'd

- Syntax:

```
int MPI_Sendrecv(const 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)
```

- Parameters are essentially the combination of MPI_Send and MPI_Recv

# Combined Send and Receive cont’d

- Code Example: MPI_comb_sendrecv.c

# Collective Communication

# Collective Communication Routines

- Collective communication routines are use to dissipate or collect data, or to synchronize operations among processes
- Common routines for:
  - Barrier synchronization
  - Broadcast data
  - Scatter data
  - Gather data
  - Reduce

# MPI_Barrier

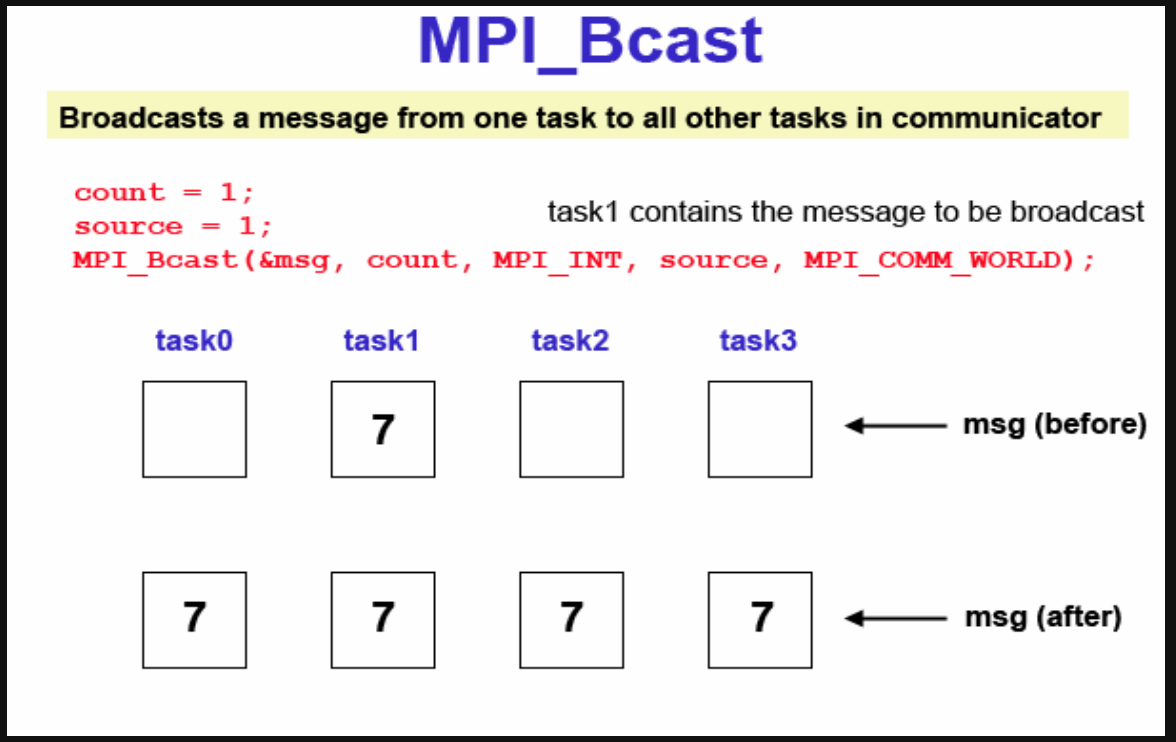
- Similar to Pthread barrier, MPI_Barrier creates a barrier synchronization for all processes in a communication group
- Syntax:

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

- Input parameters:
  - comm: communicator, e.g., MPI_COMM_WORLD

# MPI_Bcast

- Broadcasts (sends) a message from one process all other processes in a communication group.
- Both the broadcaster and the receivers use the same function for this communi-cation



# MPI_Bcast

- Syntax:

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

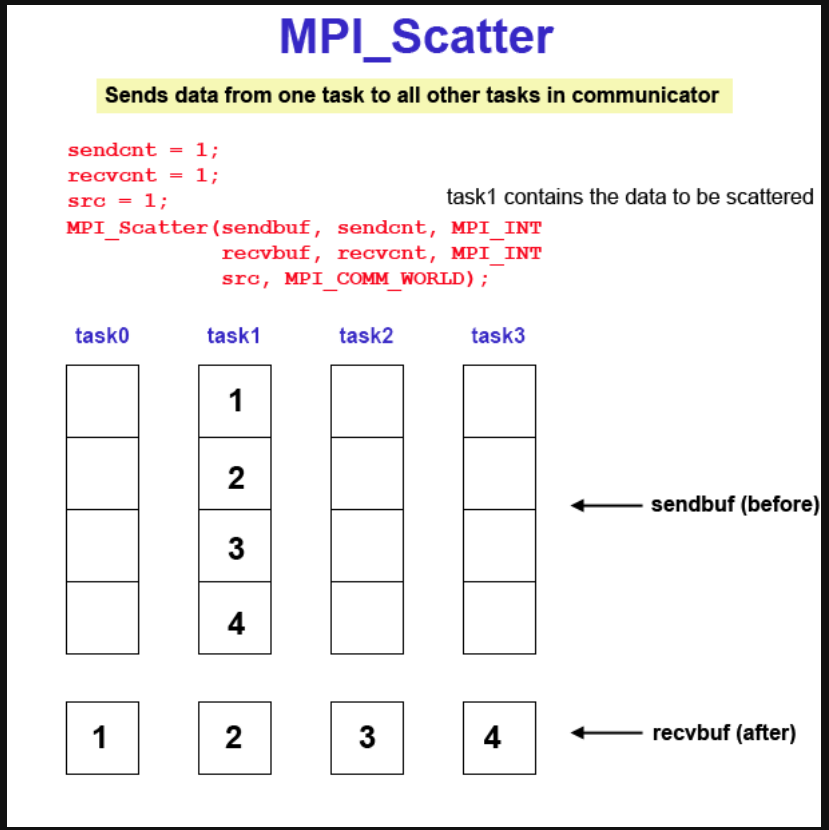
- Input/output parameter:
  - buffer: for broadcaster, this is the buffer holding the data to be broadcasted; for receiver, this is the buffer receiving the broadcasted data
- Input parameters:
  - count: number of elements in the message
  - datatype: data type of the elements
  - root: rank/id of the broadcaster
  - comm: comm: communicator, e.g., MPI_COMM_WORLD

# MPI_Bcast cont’d

- Code example: MPI_bcast.c

# MPI_Scatter

- Distributes distinct messages from a single source task to each task in the group.
- Both the distributer and the receivers use the same function for this communi-cation



# MPI_Scatter cont'd

- Syntax:

```
int MPI_Scatter( void *sendbuf,
                 int sendcnt,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcnt,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm  )
```

- Input parameter:
  - sendbuf: the buffer holding the data to be scattered
  - sendcnt: number of elements to send to each process
  - sendtype: data type of the elements to be scattered
  - root: rank/id of the owner of all data
  - comm: comm: communicator, e.g., MPI_COMM_WORLD

# MPI_Scatter cont'd

- Syntax:

```
int MPI_Scatter( void *sendbuf,
                 int sendcnt,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcnt,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm  )
```

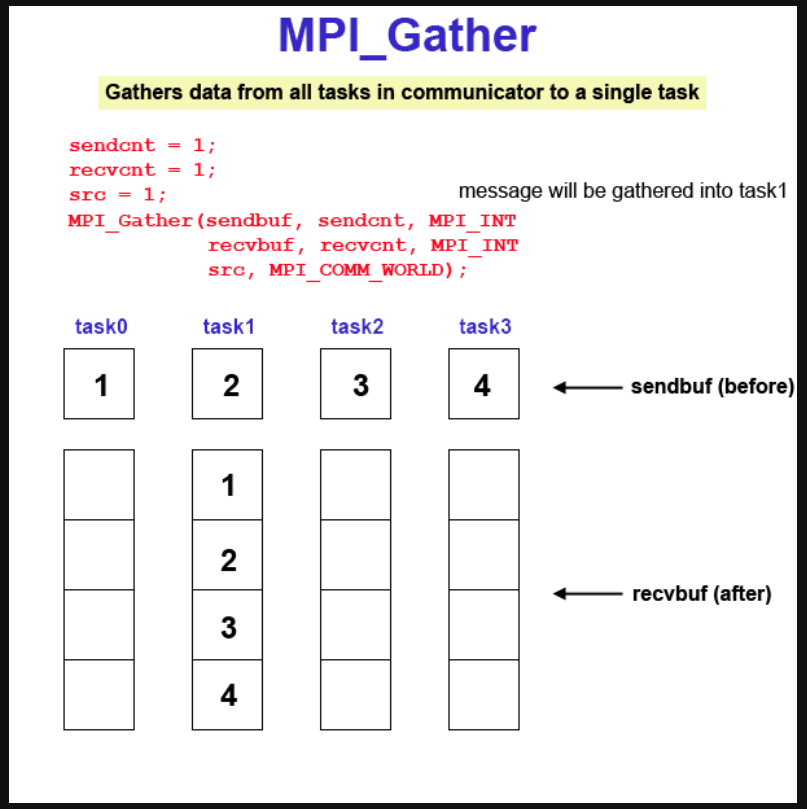
- Input parameter cont'd:
  - recvcnt: number of elements to receive for each process
  - recvtype: data type of the elements to be received
- Output parameter:
  - recvbuf: the buffer holding the data to be received

# MPI_Scatter cont’d

- Code example: MPI_scatter.c

# MPI_Gather

- Gathers distinct messages from each task in the group to a single destination task.
- This routine is the reverse operation of MPI_Scatter.
- Both the collector and the sender use the same function for this communi-cation



# MPI_Gather cont'd

- Syntax:

```
int MPI_Gather( void *sendbuf,
                int sendcnt,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcnt,
                MPI_Datatype recvtype,
                int root,
                MPI_Comm comm  )
```

- Input parameter:
  - sendbuf: the buffer holding the data to be sent to the gathering process
  - sendcnt: number of elements in the send buffer
  - sendtype: data type of the elements to send
  - root: rank/id of the gathering process
  - comm: comm: communicator, e.g., MPI_COMM_WORLD

# MPI_Gather cont'd

- Syntax:

```
int MPI_Gather( void *sendbuf,
                int sendcnt,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcnt,
                MPI_Datatype recvtype,
                int root,
                MPI_Comm comm  )
```

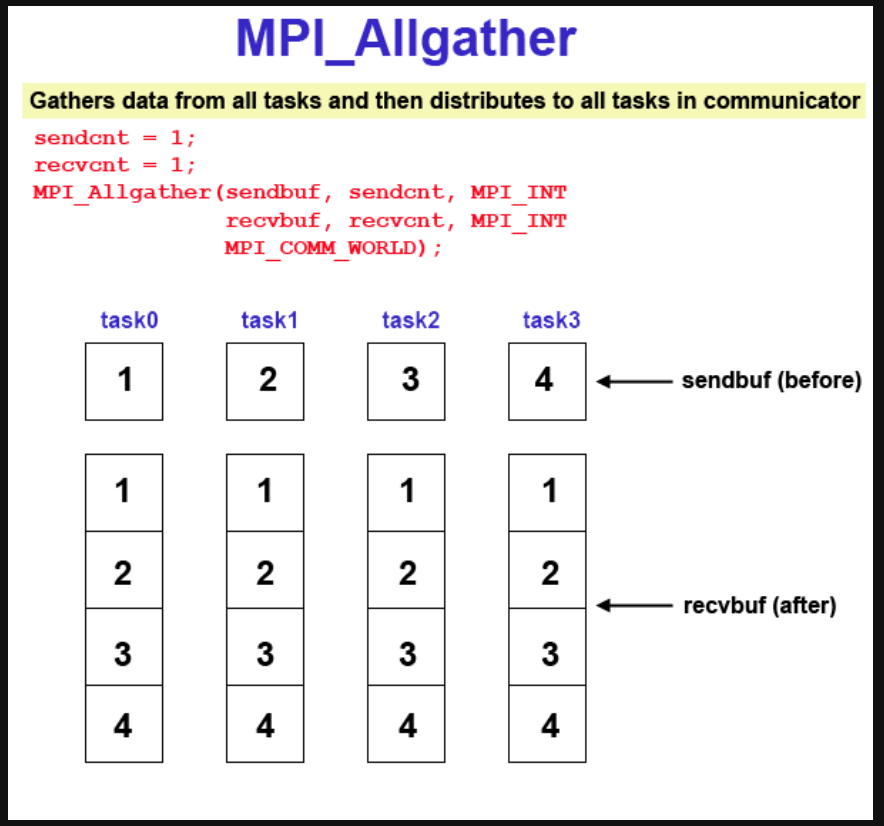
- Input parameter cont'd:
  - recvcnt: number of elements to gather from each process
  - recvtype: data type of the elements to be gathered
- Output parameter:
  - recvbuf: the buffer holding the data to be gathered

# MPI_Gather cont’d

- Code example: MPI_gather.c

# MPI_Allgather

- Similary to MPI_Gather, except that every process retain a copy of gathered data
- All processes use the same function for this communi-cation



# MPI_Allgather cont'd

- Syntax:

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

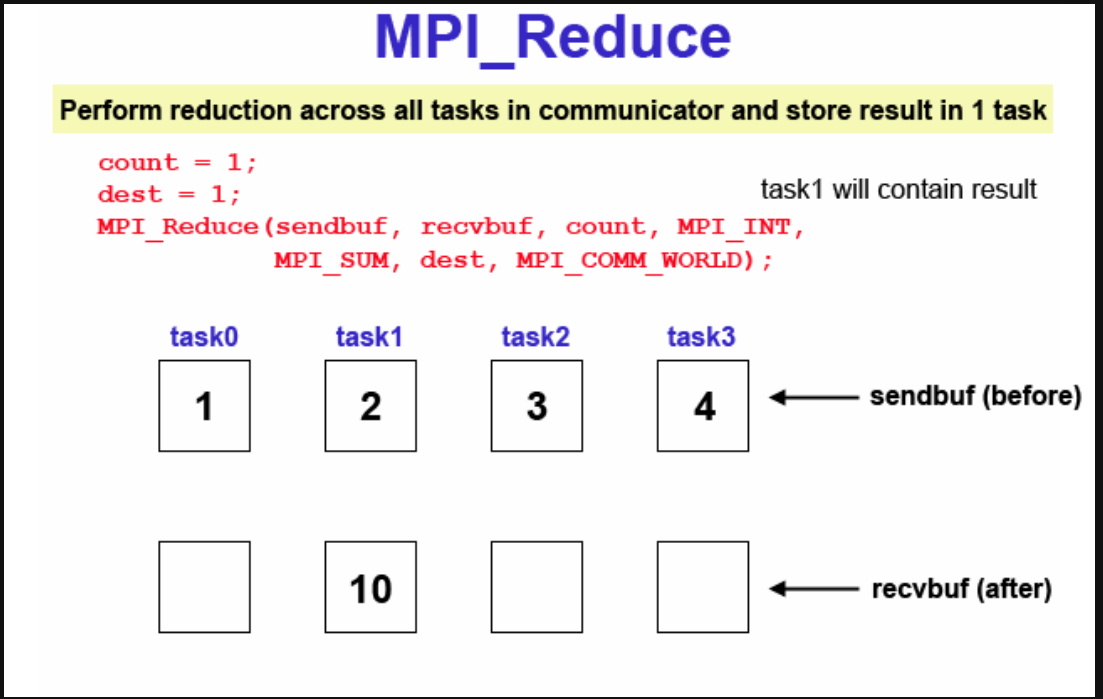
- Input parameters:
  - sendbuf: the buffer holding the data to be sent
  - sendcnt: number of elements in the send buffer
  - sendtype: data type of the elements to send
  - recvcnt: number of elements to receive from each process
  - comm: comm: communicator, e.g., MPI_COMM_WORLD
- Output parameter:
  - recvbuf: the buffer holding gathered data

# MPI_Allgather cont’d

- Code example: MPI_allgather.c

# MPI_Reduce

- Applies a reduction operation on all tasks in the group and places the result in one task.
- Similar to the reduction operation in OpenMP
- All participating processes use the same function for this communi-cation



# MPI_Reduce cont'd

- Syntax:

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

- Input parameter:
  - sendbuf: the buffer holding the data for reduction
  - count: number of elements in the send buffer
  - datatype: data type of the elements to send
  - op: the reduction operation
  - root: rank/id of the process holding the result
  - comm: comm: communicator, e.g., MPI_COMM_WORLD
- Output parameter:
  - recvbuf: the buffer holding the reduction result

# MPI_Reduction cont’d

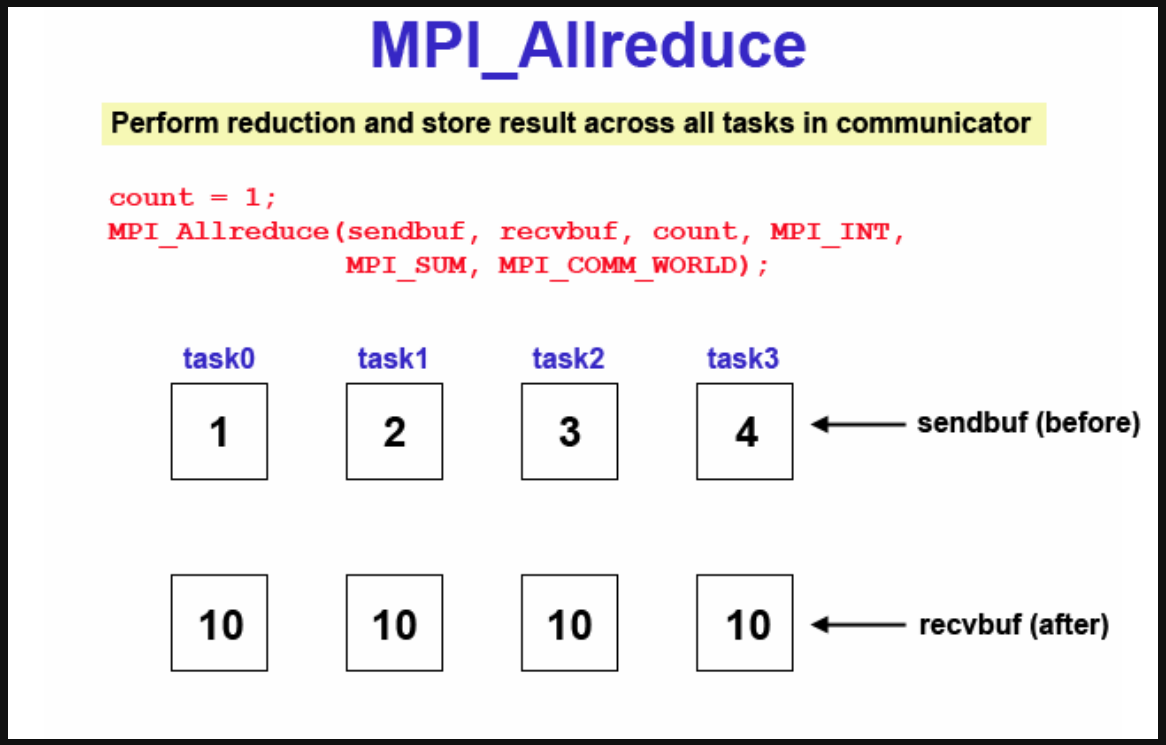
- Builtin reduction operations (for explanation -- https://linux.die.net/man/3/mpi_land) :
  - MPI_MAX
  - MPI_MIN
  - MPI_SUM
  - MPI_PROD
  - MPI_LAND
  - MPI_BAND
  - MPI_LOR
  - MPI_BOR
  - MPI_LXOR
  - MPI_BXOR
  - MPI_MINLOC
  - MPI_MAXLOC
- Custom operations are also allowed

# MPI_Reduce cont’d

- Code example: MPI_reduce.c

# MPI_Allreduce

- Applies a reduction operation on all tasks in the group and places the result in alls tasks.
- All participating processes use the same function for this communi-cation



# MPI_Allreduce cont'd

- Syntax:

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

- Input parameter:
  - sendbuf: the buffer holding the data for reduction
  - count: number of elements in the send buffer
  - datatype: data type of the elements to send
  - op: the reduction operation
  - comm: comm: communicator, e.g., MPI_COMM_WORLD
- Output parameter:
  - recvbuf: the buffer holding the reduction result

# MPI_Allreduce cont’d

- Code example: MPI_allreduce.c

# Derived Types and Custom Operations

# Derived Types and Custom Operations

- Although MPI provides many built-in types and operations, it is common that a problem requires to communicate data in custom types and perform custom operations on custom data.
  - Just like defining custom types and overriding operations with arrays, structures and classes in C and C++
- MPI provides functions to defined both custom types and operations.

# Derived Data

- Similar to structs, classes and arrays, custom data types in MPI are usually defined based on the built-in types
  - So the custom data are called “derived data” in MPI, i.e., derived from built-in types
- Flavors of Derived data
  - Contiguous – similar to array
  - Vector – similar to contigous, with gaps in the array
  - Index – similar to contigous, except the array is partitioned into blocks
  - Struct – similar to C struct

# Derived Struct

- MPI derived structs are partitioned into blocks
  - A block is a group of variables of the same type

```
struct item{
  int id;
  int count;
  double weight;
  double quality;
  float price;

};
```

Block 1

Block 2

Block 3

# Derived Struct cont'd

- Each block is identified by its data type, length (number of internal variables) and offset

```
struct item{
  int id;
  int count;
  double weight;
  double quality;
  float price;

};
```

Block 1: type: MPI_INT

Block 2: type: MPI_DOUBLE

Block 3: type: MPI_FLOAT

# Derived Struct cont'd

- Each block is identified by its data type, length (number of internal variables) and offset

```
struct item{
  int id;
  int count;
  double weight;
  double quality;
  float price;

};
```

Block 1: len: 2 (ints)

Block 2: len: 2 (doubles)

Block 3: len: 1 (float)

# Derived Struct cont'd

- Each block is identified by its data type, length (number of internal variables) and offset
  - Offset: beginning location of the block

```
struct item{
   int id;
   int count;
   double weight;
   double quality;
   float price;

};
```

Block 1: offset: 0

Block 2: offset: (after) the size of 2 MPI_INT

Block 3: offset:  (after) the size of 2 MPI_INT+2 MPI_DOUBLE

# Derived Struct cont’d

- To get the actual size of a MPI built-in type (such as MPI_INT and MPI_double), call MPI_Type_extent
- To create a custom MPI derived struct, call “MPI_Type_struct” and pass in the informations of the blocks

# Derived Struct cont'd

- Syntax:

```
int MPI_Type_struct( int count,
                     int blocklens[],
                     MPI_Aint offsets[],
                     MPI_Datatype old_types[],
                     MPI_Datatype *newtype )
```

- Input parameters:
  - Count: the number of blocks
  - Blocklens: array with the length of each block
  - Offsets: array with the offset of each block
  - old_types: array with the MPI built-in type of each block
- Output parameters:
  - new_type: the new MPI type for this struct

# Derived Struct cont’d

- Syntax:

```
int MPI_Type_Extent( MPI_Datatype datatype,
                     MPI_Aint *extent )
```

- Input parameter:
  - Datatype: the built-in MPI data type
- Output parameter:
  - Extent: the size of the built-in MPI data type

# Derived Struct cont'd

- Computing the offsets for block 2 and 3 in the previous example:

```
MPI_Aint block2_offset, block3_offset;
MPI_Aint mpi_int_size, mpi_dbl_size;

/* get the sizes of MPI_INT and MPI_DOUBLE */
MPI_Type_extent(MPI_INT, &mpi_int_size);
MPI_Type_extent(MPI_DOUBLE, &mpi_dbl_size);

/* compute the offsets */
block2_offset = 2 * mpi_int_size;
block3_offset = 2 * mpi_int_size + 2 * mpi_dbl_size;
```

# Derived Struct cont'd

- Code to create the derived struct (using the offsets from previous slide)

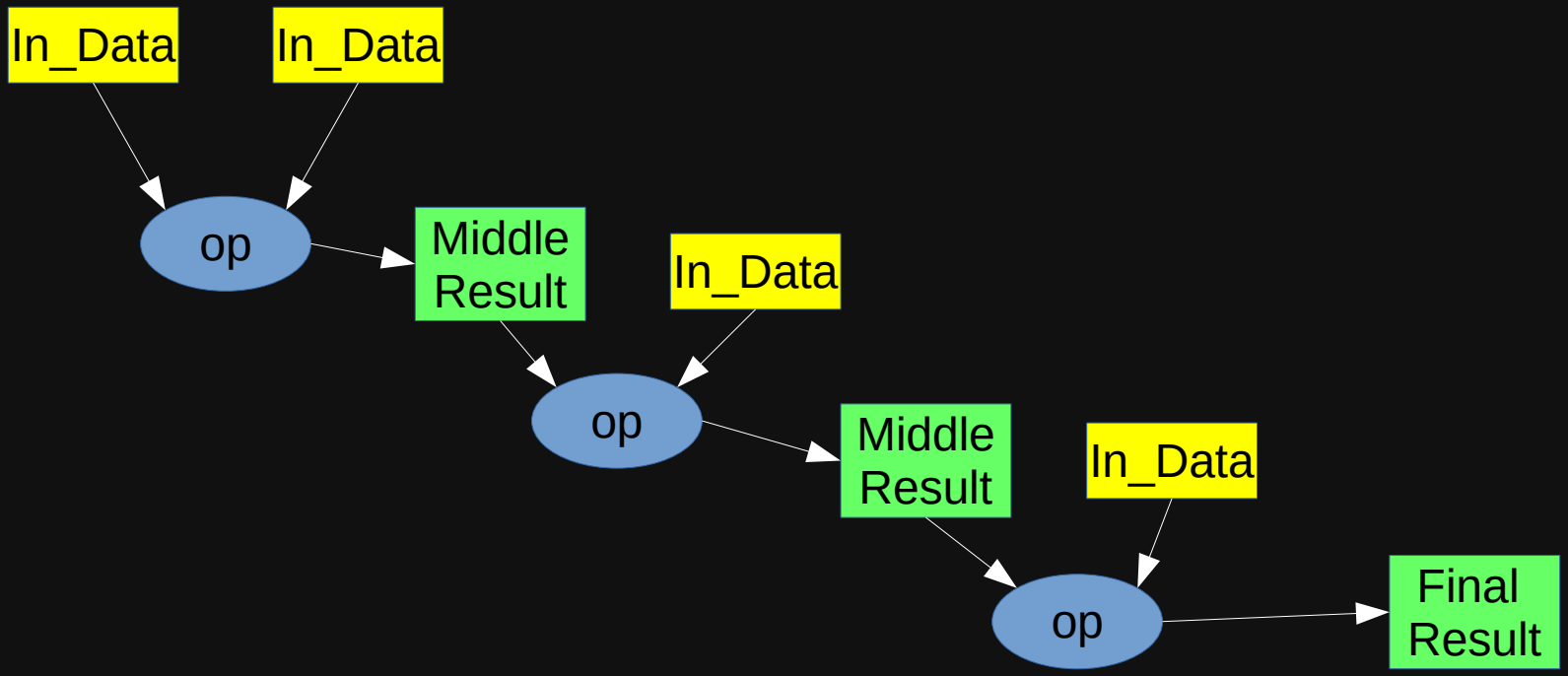
```
MPI_Datatype new_type; // the new type

int count = 3; // three blocks

int blocklens[3] = {2,2,1}; //lengths of the blocks

MPI_Aint offsets[3] = {0, block2_offset, block3_offset};

MPI_Datatypes oldtypes[3] = {MPI_INT, MPI_DOUBLE,
MPI_FLOAT}; // three blocks are of int, double and float types

/* create the new type */
MPI_Type_struct(count, blocklens, offsets, oldtypes,
&newt_type);
```

# Custom Operation

- MPI allows creating custom operations to be used in reductions on derived or built-in types
- MPI allows commutative and non-commutative operations
  - But all operations are assumed to be associative

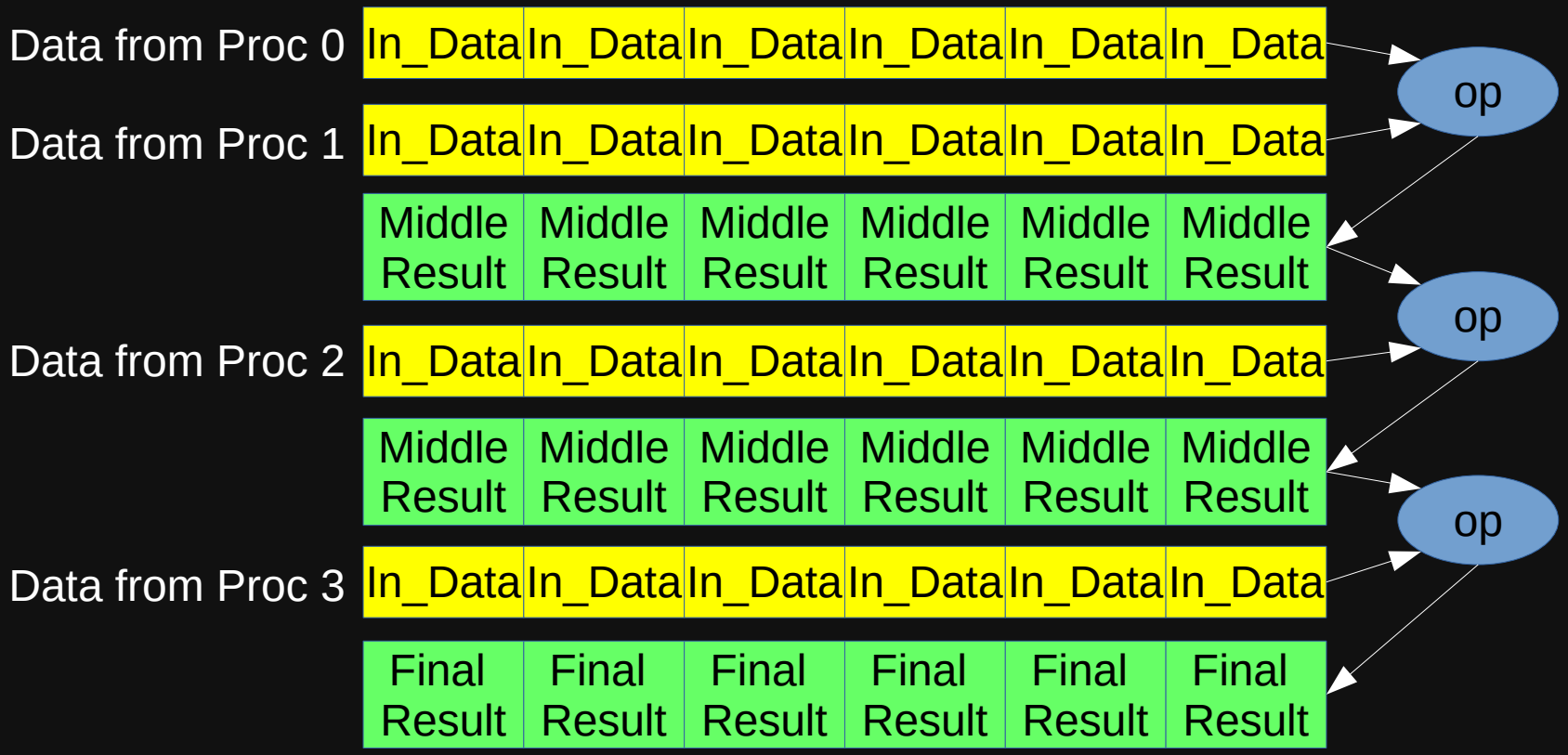
# Custom Operation cont'd

- An illustration of the commutative reduction procedure:
  - Four processes, operation is op, each process has in_data to be reduced on



# Custom Operation cont'd

- An illustration of the commutative reduction procedure with an array of data from each process



# Custom Operation cont’d

- For any custom operation,
  - MPI passes in two arrays of operands
    - One operand array is from a process, the other one may or may not be an array of middle results
  - The custom operation return the result array

# Custom Operation cont'd

- Every custom MPI operation should be a function in this format:

```
void your_op_name(void * in_data_array,
                  void * inout_data_array,
                  int * len,
                  MPI_Datatype * datatype);
```

- Input parameters:
  - in_data_array: the array of one operands; typed (void *), so it can hold data of any type
  - len: the length of the array
  - datatype: the type of the operands
- Input/out parameter:
  - inout_data_array: as an input parameter, it holds the array of the other operands; as an output parameter, the results should be in this paramter

# Custom Operation cont'd

- An example: sum the price and weight of data of *struct item* (from slide 69)

```
void sum_all(void * in_data_array,
             void * inout_data_array,
             int * len,
             MPI_Datatype * datatype)
{
   // convert void * to struct item start
   struct item *in = in_data_array;
   struct item *inout = inout_data_array;
   struct item tmp;
   int i;

   for(i = 0; i < len; i++){
       // sum weight and price
       tmp.weight = in[i].weight + inout[i].weight;
       tmp.price = in[i].price + inout[i].price
       // set result to the output array
       inout[i] = tmp;
   }
}
```

# Custom Operation cont'd

- To register the custom operation with MPI, use API "MPI_Op_create"
- Syntax:

```
int MPI_Op_create(MPI_User_function *user_fn,
                  int commute,
                  MPI_Op *op)
```

- Input parameters:
  - user_fn: the name of the custom operation function
  - commute: 1 if commutative; 0 if not commutative.
- Output parameter:
  - Op: the handle of the custom operation

# Custom Operation cont’d

- An example of registering the custom operation of summing prices and weights:

```
MPI_Op custom_sum_op;

MPI_Op_create(sum_all, 1, &custom_sum_op);
```

# Combining Derived Data and Custom Operation

- Code Example: MPI_custom_type_op.c

# User Defined Communication Groups

# User Defined Communication Groups

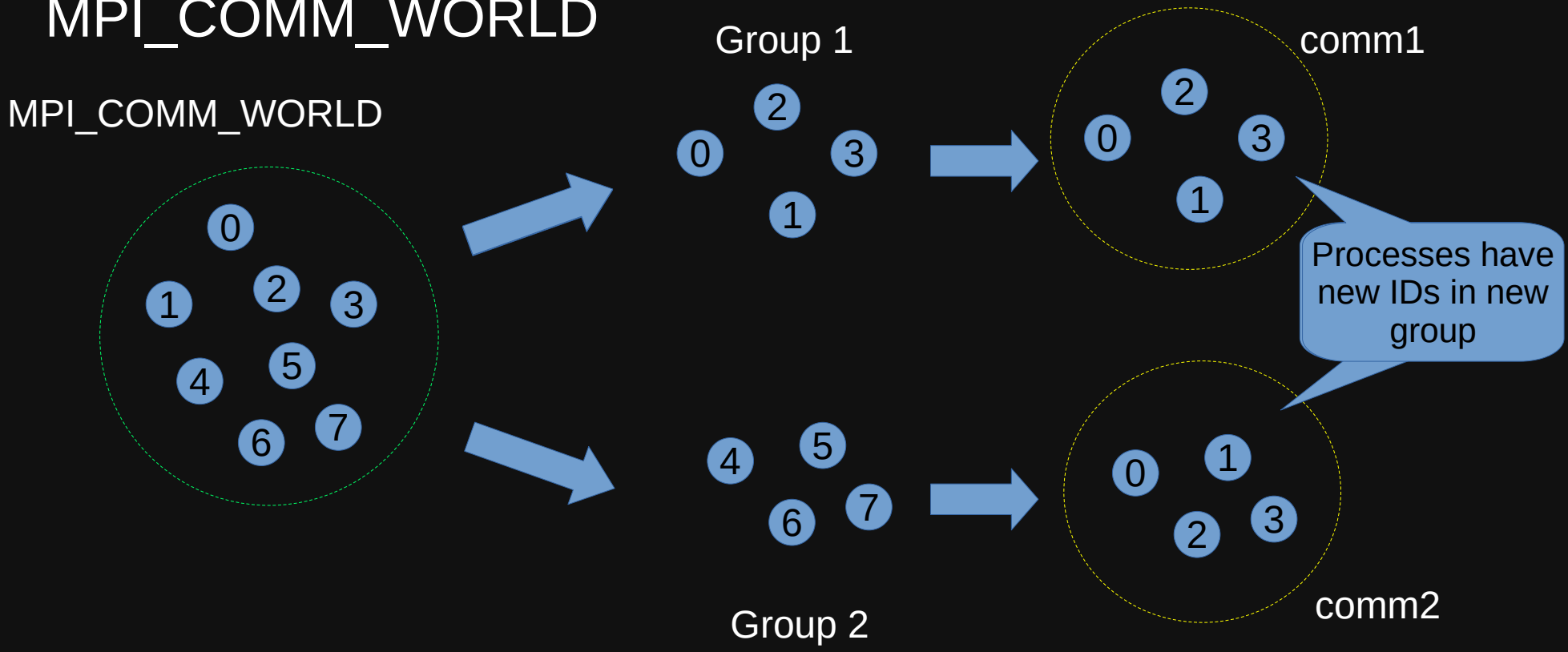
- User defined communication groups provide better control over communication to optimize performance or simplify coding
  - Allow you to organize tasks, based upon function, into task groups.
  - Enable collective communications operations across a subset of related tasks.
  - Provide basis for implementing user defined virtual topologies

# Groups vs Communicator

- A group is an ordered set of processes. A group is always associated with a communicator object.
- A communicator encompasses a group of processes that may communicate with each other.
- From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

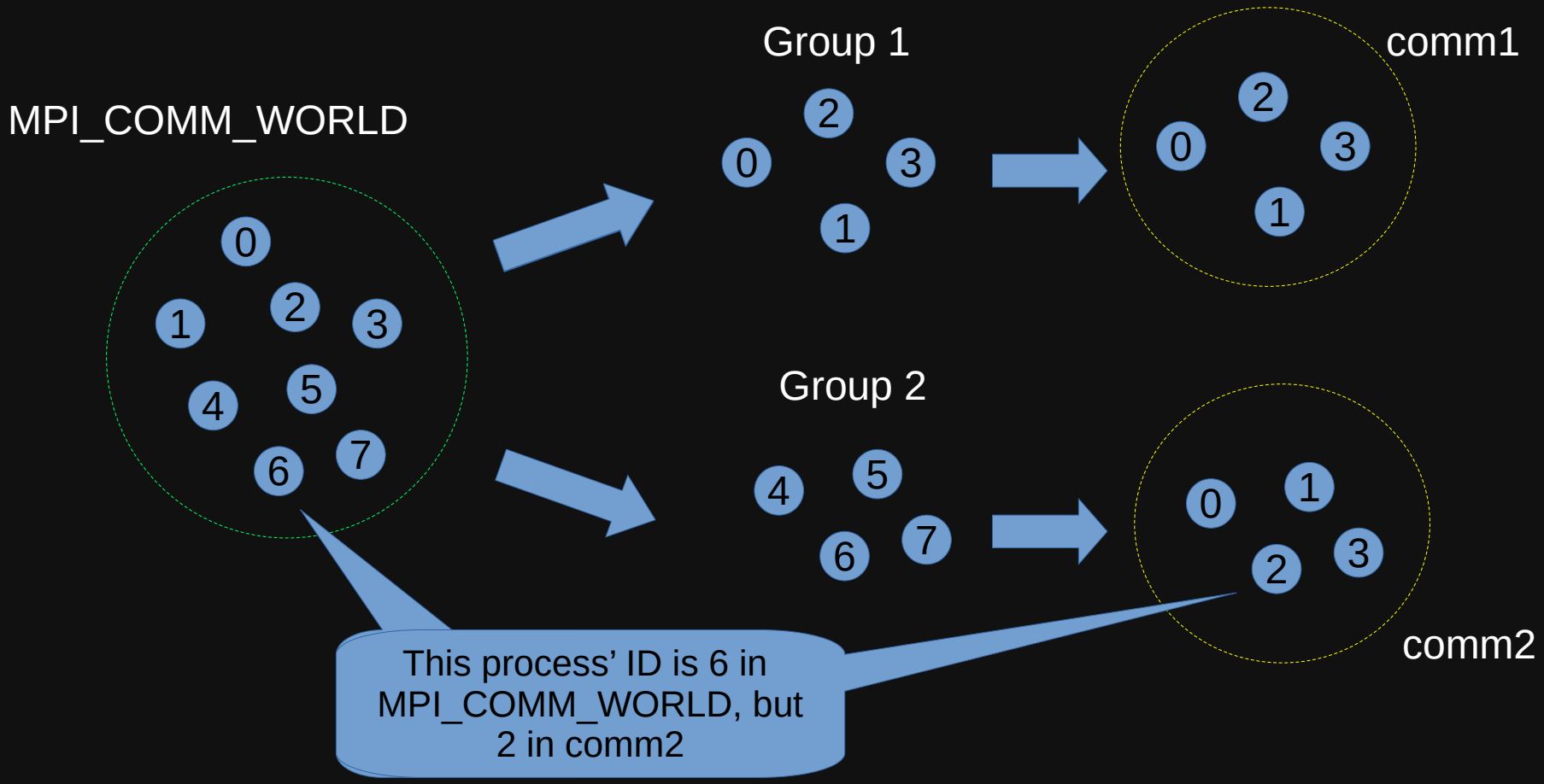
# Defining Custom Groups

- Typically, a new group is created from an existing group. In particular, created from the all-in group represented by MPI_COMM_WORLD



# Defining Custom Groups cont'd

- Note that a process can participate in multiple groups, and it may have different IDs in each group.



# The Procedure of Defining a New Group

- Extract the group handler from an old group.
- Create the new group using the old group's handle and,
  - The IDs of processes that go to the new group
  - The number of processes in the new group
- Create a new communicator from the new group.

# Extracting the Handle of an Old Group

- Syntax:

```
int MPI_Comm_group(MPI_Comm comm,
                   MPI_Group *group)
```

- Input parameter:
  - comm: the communicator of the old group
- Output parameter:
  - group: the handle of the old group

# Creating a New Group from an Old Group

- Syntax:

```
int MPI_Group_incl(MPI_Group group,
                   int n,
                   const int ranks[],
                   MPI_Group *newgroup)
```

- Input parameter:
  - group: the handle of the old group
  - n: number of processes in the new group
  - Ranks:  IDs (in the old group) of the processes that are to be included the new group
- Output parameter:
  - new_group: the handle of the new group

# Create a Communicator for A New Group

- Syntax:

```
int MPI_Comm_create(MPI_Comm comm,
                    MPI_Group group,
                    MPI_Comm *newcomm)
```

- Input parameter:
  - comm: the communicator of the old group
  - Group: the handle of the new group
- Output parameter:
  - newcomm: the communicator of the new group

# Example of Creating A New Group

- Example: creating a new group using processes 4, 5, 6 and 7 of the old all-in group

```
// IDs of the procs in the old group that are to be
// included in the new group
int ranks[4] = {4,5,6,7};
// handles of the old and new groups
MPI_Group  orig_group, new_group;
MPI_Comm   new_comm; //communicator of the new group

// get the group handle of the all-in group
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

// create the new group
MPI_Group_incl(orig_group, 4, ranks, &new_group);

// craete the new communicator
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
```

# Complete Code Example

- Code example: MPI_group.c