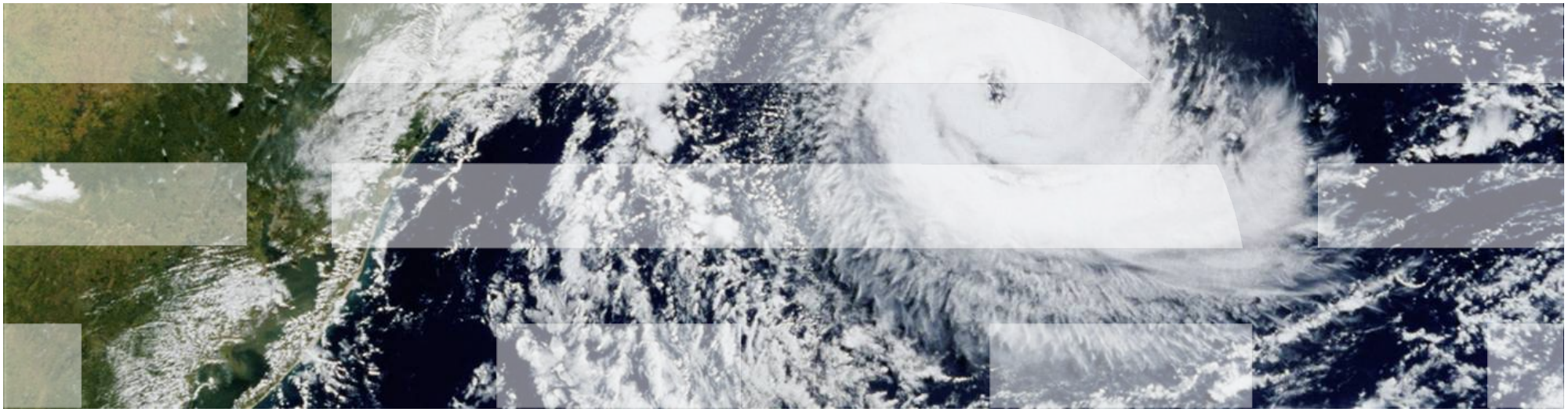
Programming Techniques for Supercomputers

(Heterogeneous and Distributed systems)

– Aditya Nitsure (IBM India)



MPI (Message Passing Interface)

(https://www.mpi-forum.org)

(MPI forum mail : mpi-comments@mpi-forum.org)

Take away

- What is MPI?
- How MPI programming model works ?
- How to write, compile and run MPI program?
- What are various MPI library API, type of communications, constants etc.?
- Can MPI take advantages of accelerators (GPUs)?

MPI Specification History

- MPI 1.0 (1994)
  - First specification published
  - Point to point communications
- MPI 1.1 (1995) / 1.2 (1997)
  - Minor corrections and clarifications to previous specification
- MPI 2.0 (1997)
  - Process creation and management
  - One sided communications
  - Extended collective communication
  - Parallel IO
- MPI 1.3 (2008)
  - Combines specification 1.1 & 1.2
  - New errata from MPI forum with respect to 1.1 & 1.2 specification
- MPI 2.1 (2008)
  - Merged MPI 1.3 and 2.0 specification
- MPI 2.2 (2009)
  - Mostly corrections and clarifications to MPI 2.1 specification
- MPI 3.0 (2012) – Major update
  - Extension to one sided communication operations
  - FORTRAN 2008 bindings
  - Non-blocking collective operations
- **MPI 3.1 (2015)**
  - New functions to manipulate non-blocking collective I/O routines
  - Mostly corrections and clarifications to MPI 3.0 specification

Introduction (1)

- Specification for distributed memory parallel programming model
- Specifies library interface (API) in C and Fortran programming languages (300+ APIs)
  - MPI wrappers available for Python, JAVA, MATLAB etc.
- MPI prescribes an API and the behaviour of those APIs. (300+)
  - Defines what happens, but not how it happens.
- MPI applications are source code compatible.
- Number of MPI libraries implementations
  - OpenMPI, MPICH, MVAPICH, SpectrumMPI (IBM), Intel MPI, Cray MPI etc.

Introduction (2)

- Open source, portable and efficient in communication
- Widely accepted by community – academician & industry
- Supports SPMD (Single Program Multiple Data) model
- Portable across heterogeneous systems
- Used in application development from variety of fields and systems
  - Aeronautics, Automotive, Pharmaceutics, Finance
  - Supercomputers, Accelerators, Embedded multi-core systems

MPI Terminology

- **Rank :** Unique identification number. Always starts with 0 (in C) and assigned sequentially
- **Communicator :** A communicator specifies the communication context for a communication operation. Each communication context provides a separate “communication universe”: messages are always received within the context they were sent, and messages sent in different contexts do not interfere.
- **Blocking :** A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.
- **Non-blocking :** A procedure is non-blocking if it may return before the associated operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call.
- **Local :** A procedure is local if completion of the procedure depends only on the local executing process.
- **Non-local :** A procedure is non-local if completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require communication occurring with another user process.
- **Collective :** A procedure is collective if all processes in a process group need to invoke the procedure.
- **Predefined :** Datatype with a predefined (constant) name e.g. MPI_INT
- **Derived :** Any datatype that is not predefined.
- **Portable :** predefined datatype or derived using MPI_TYPE * constructors
- **Equivalent :** Two data types are equivalent if they appear to have been created with same sequence of calls (arguments) and have same type map.
- **Opaque :** Objects which are not directly accessible to users and can only accessed via ‘handles’ e.g. groups, communicator etc.

MPI Hello World

helloWorld.c

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

int main(int argc, char* argv[])
{
   int myrank;
   int comm_size;
   // All MPI calls should be between MPI_Init() & MPI_Finalize()
   MPI_Init(&argc, &argv);
   // Get total number of tasks
   MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
   // Each task
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

   printf("my rank is : %d of %d\n", myrank, comm_size);

   // Last MPI API to be called in MPI program
   MPI_Finalize();
   return 0;
}
```

```
//compile helloWorld program
mpicc helloWorld.c -o helloWorld

// Run helloWorld program
mpirun -np 4 --hostfile hostlist ./helloWorld
```

Run this many copies of the program on the given nodes

Hostlist

```
hpcwsw7.aus.stglabs.ibm.com
hpcwsw7.aus.stglabs.ibm.com
hpcwsw8.aus.stglabs.ibm.com
hpcwsw8.aus.stglabs.ibm.com
```

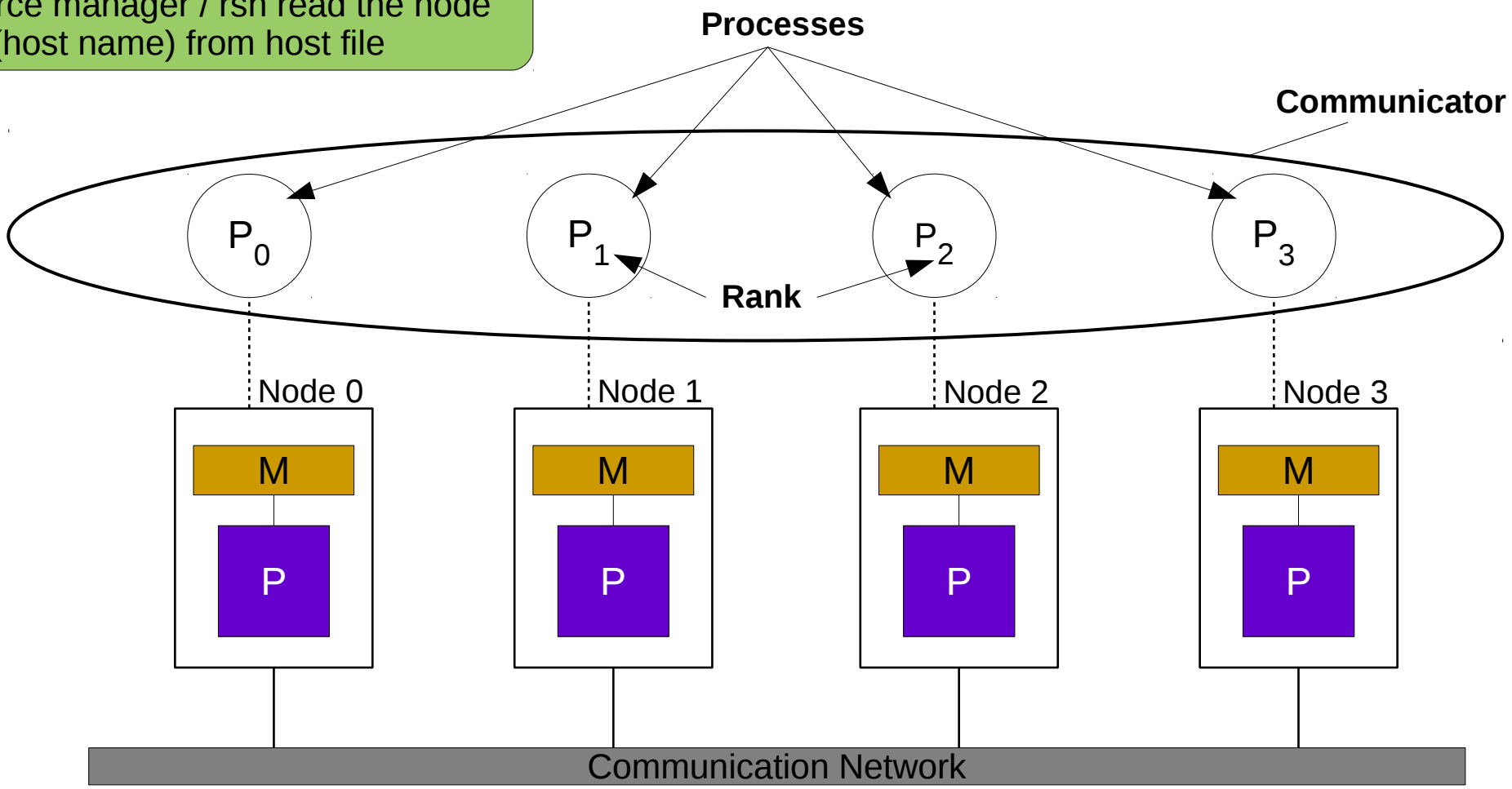
Output

```
[aditya@hpcwsw7 openmp_tutorial]$ ./run_mpi
my rank is : 3 of 4
my rank is : 1 of 4
my rank is : 2 of 4
my rank is : 0 of 4

[aditya@hpcwsw7 openmp_tutorial]$ ./run_mpi
my rank is : 2 of 4
my rank is : 3 of 4
my rank is : 0 of 4
my rank is : 1 of 4
```

MPI Execution

- ***mpirun*** uses ORTE (Open Run Time Environment) to launch jobs
- ORTE uses rsh/ssh or resource manager to launch process on remote nodes
- Resource manager / rsh read the node name (host name) from host file



MPI_Init / MPI_Finalize

int MPI_Init
(int *argc, char ***argv)
Example : *MPI_Init(&argc, &argv);*

int MPI_Finalize(void)

Example : *MPI_Finalize();*

- MPI program must contain exactly one call to MPI_Init()
- Application can access information about execution environment only after MPI Initialization e.g. MPI_COMM_WORLD

- Each process must call before it exits
- Cleans up all MPI states
- All MPI_* calls should be completed before calling MPI_Finalize()

No MPI communication routine can be called before MPI_Init and after MPI_Finalize

MPI Communication Types

- Point-to-point
- Collective
- One-sided
- MPI I/O

Point-to-point communication

- Basic communication mechanism
- One task (process) sends data and another task (process) receives it using ***send*** and ***receive*** operation

  i.e. data is moved from address space of one process to that of another process
- **Blocking** and **non-blocking** type
- In case of multiple send & receive, the order of messages is preserved in blocking and non-blocking communication.

  No overtaking allowed !

MPI Send and Receive (blocking)

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

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

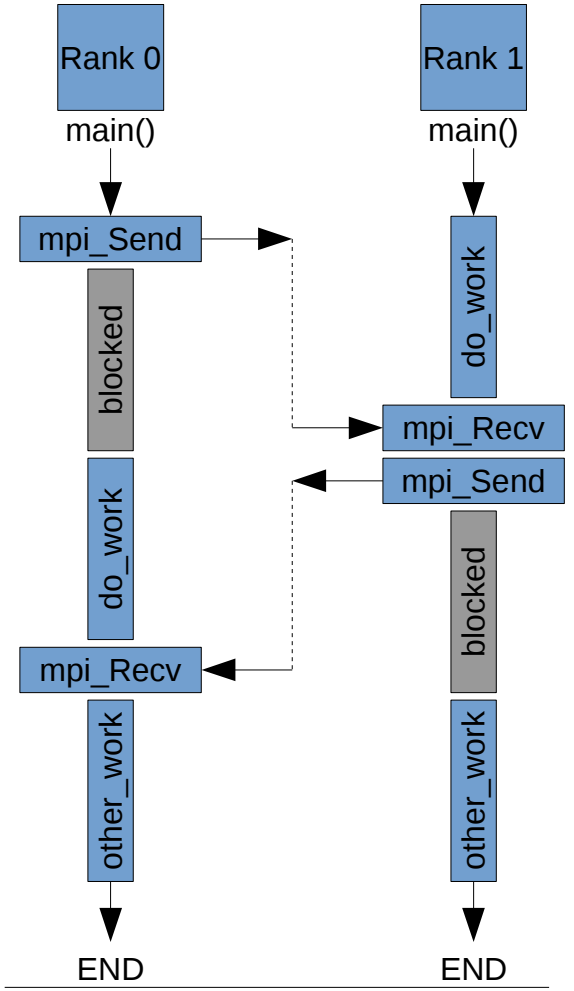
- *buf* : Initial address of send and receive buffer respectively
- *count* : number of elements in send/receive buffer
- *datatype* : data type of send/receive buffer elements
- *tag* : integer value to distinguish different types of messages
- *comm* : communicator used for this communication
- *source* : message source – rank of sender (process)
- *dest* : message destination – rank of receiver (process)
- Status : indicates delivery status. Contain information about error code, tag, source etc.

Point-to-point communication

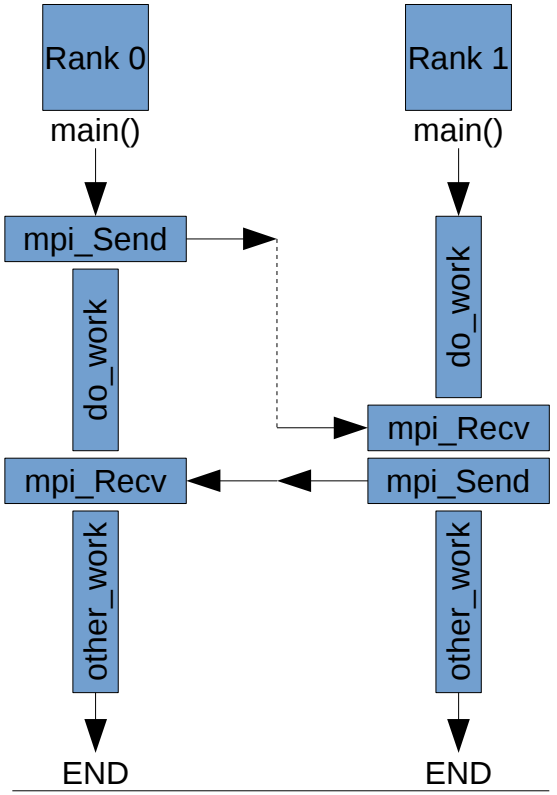
Blocking

```
int main(int argc, char**
argv)
{
    MPI_Init(&argc, &argv);
    . . .

    if (rank == 0)
    {
        MPI_Send(...);
        do_work();
        MPI_Recv(...);
        other_work();
    }
    else // (if rank == 1)
    {
        do_work();
        MPI_Recv(...);
        MPI_Send(...);
        other_work();
    }
    . . .
    MPI_Finalize();
}
```



Non-blocking




```
int main(int argc, char**
argv)
{
    MPI_Init(&argc, &argv);
    . . .

    if (rank == 0)
    {
        MPI_Isend(...);
        do_work();
        MPI_Irecv(...);
        other_work();
        MPI_Wait(&req0, ...);
    }
    else // (if rank == 1)
    {
        do_work();
        MPI_Irecv(...);
        MPI_Isend(...);
        other_work();
        MPI_Wait(&req1, ...);
    }
    . . .
    MPI_Finalize();
}
```

MPI Send & Receive (non-blocking)

```
int MPI_Isend(…, MPI_Request *request)
```

```
int MPI_Irecv(…, MPI_Request *request)
```

Blocking	Non-Blocking
Send and receive call return once data has been stored away safely (local or matching receive buffer)	Send and receive function calls returns immediately
Return of function call implies send buffer can be reused and completion of communication only in case of synchronous send	Return of function call does not imply completion of communication
Not required	Exists additional *request* parameter for checking completion status
Not required	Must have matching **wait** or **test** to check completion of communication. Data buffer can be reused only after completion of communication
Not possible to overlap communication and computation	Useful in **overlapping communication and computation**

MPI Send Modes

- **Synchronous**
  - Completes only if matching receive is posted
  - Completion ensures receiver has started receiving data
- **Buffer**
  - Completes irrespective of matching receive
  - Data is buffered on the sender side until matching receive is posted
  - Sufficient buffer space need to be allocated
- **Ready**
  - Completes only if matching receive is already posted
  - Otherwise fails and outcome is undefined !

Mode	Blocking	Non-blocking
Standard send	`int MPI_Send(...)`	`int MPI_ISend(...)`
Synchronous send	`int MPI_SSend(...)`	`int MPI_ISSend(...)`
Buffer send	`int MPI_BSend(...)`	`int MPI_IBSend(...)`
Ready send	`int MPI_RSend(...)`	`int MPI_IRSend(...)`

MPI_Sendrecv

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

- Send and receive combined in one call
- Avoids deadlocks (when compared to blocking send-receive)
- Useful in shift operation across a chain or ring of processes
- Can be used with standard send or receive

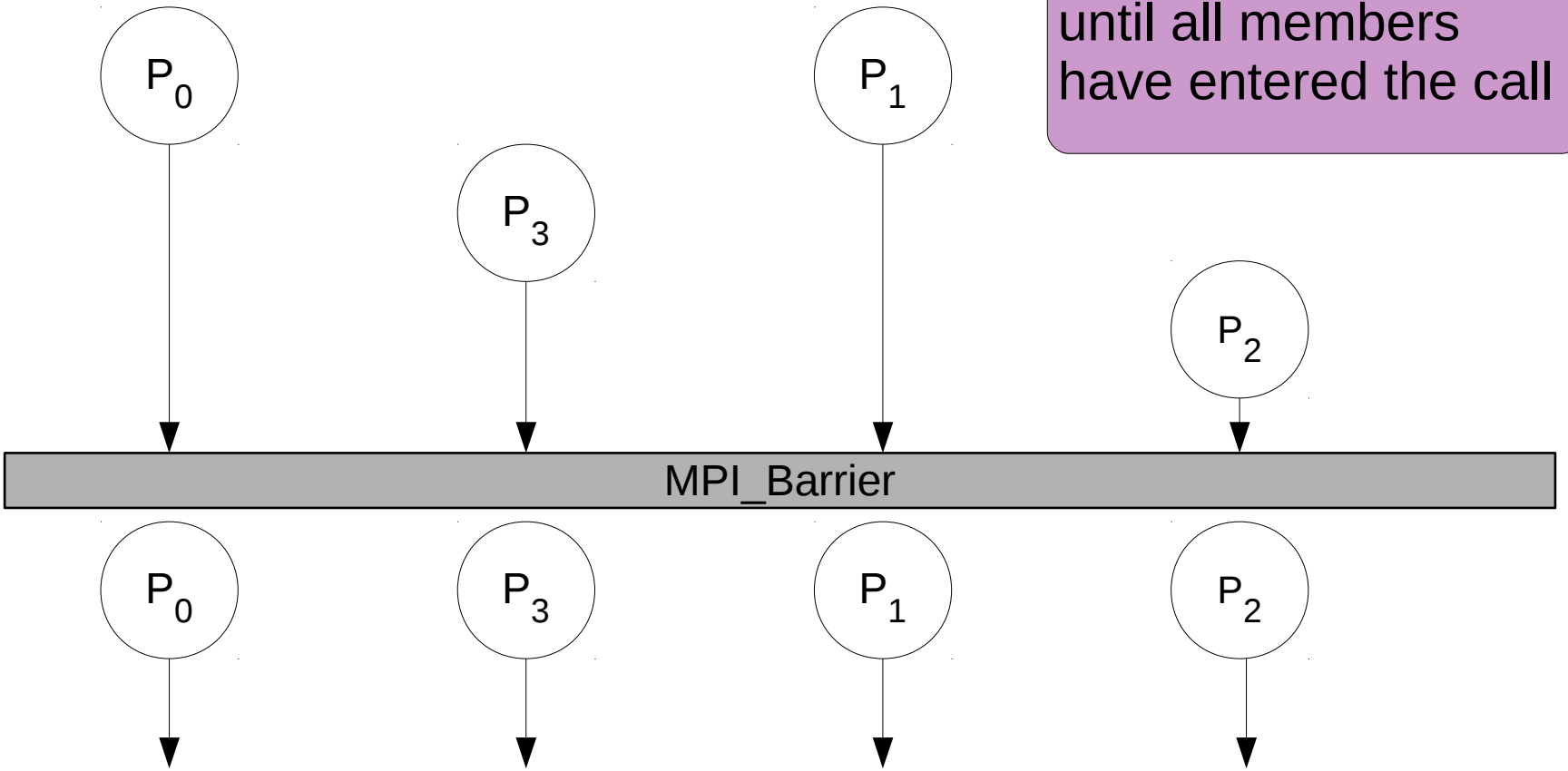
Collective Communication

MPI Barrier

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

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *req)
```

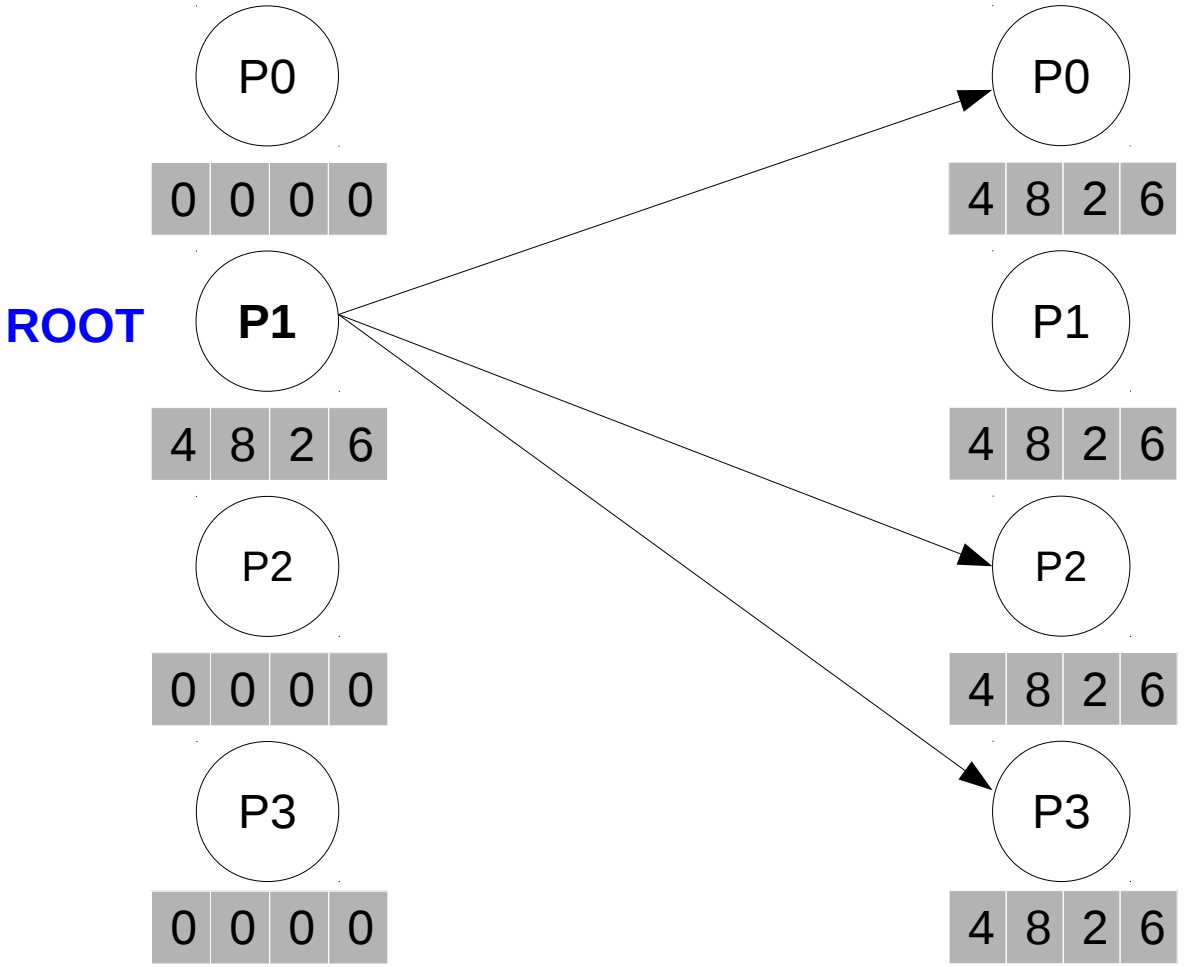
Blocks all processes until all members have entered the call



MPI Broadcast

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

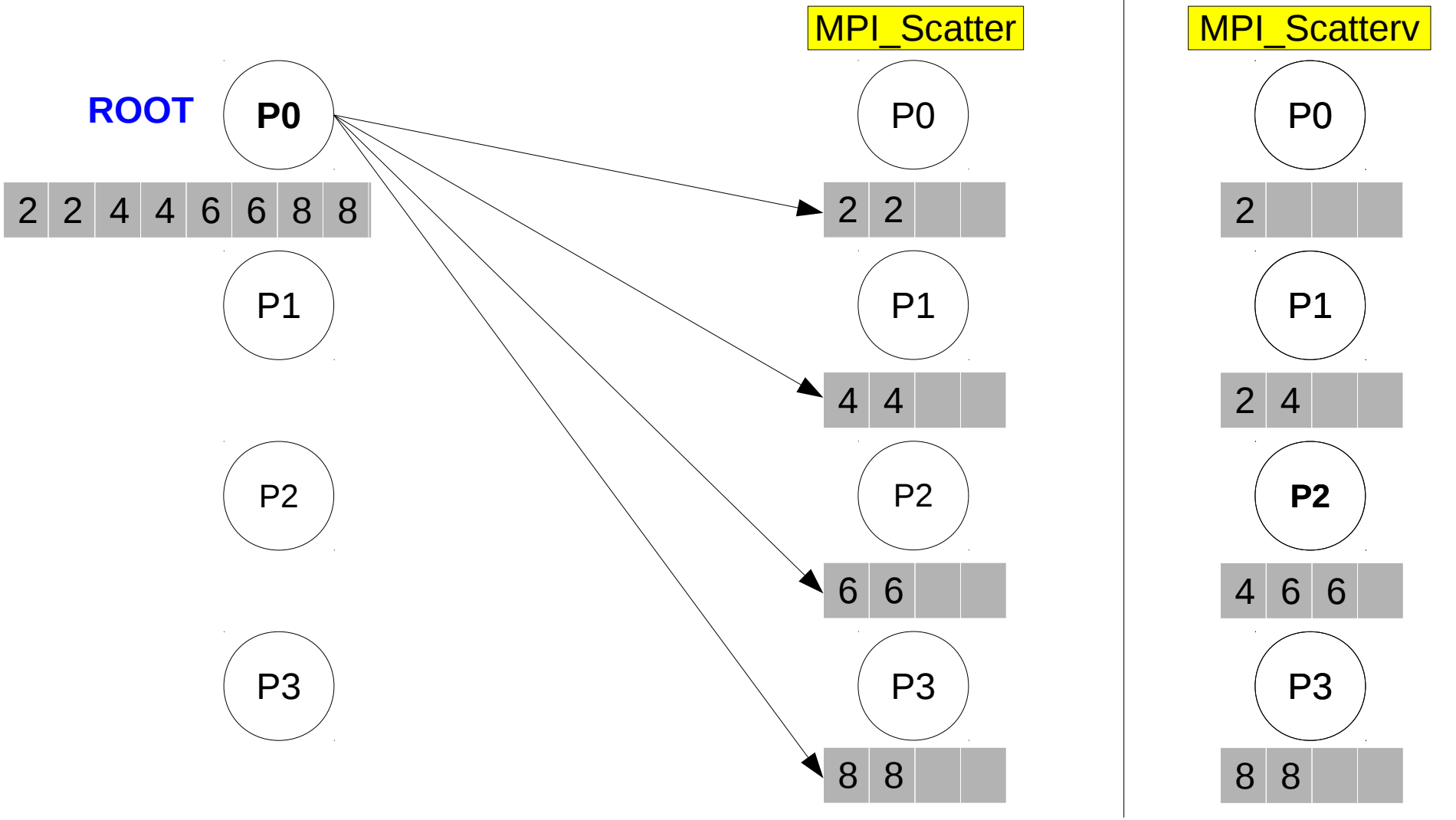
```
int MPI_IBcast(void* buffer, int count, MPI_Datatype
datatype, int root, MPI_Comm comm, MPI_Request *req)
```



Broadcasts message from root process to all processes

MPI Scatter

- Root process split message into *n* equal segments and sends to each process in the group
- Inverse of MPI_Gather



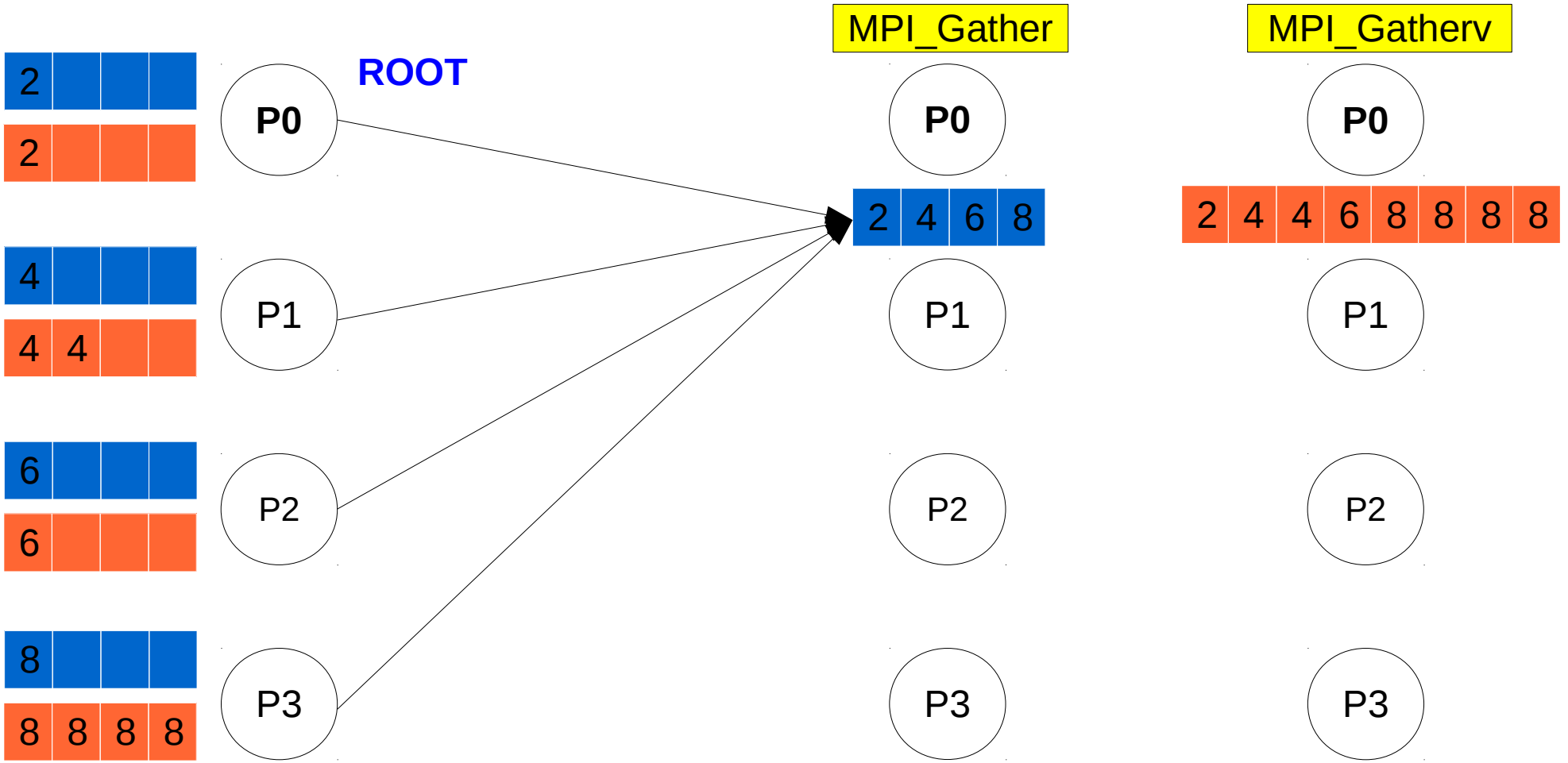
MPI Scatter

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

- MPI_Scatterv
  - `int MPI_Scatterv(..., const int sendcount[], const int displs[], ...)`
  - Allows root process to send varying count of data to each process
  - Inverse of MPI_Gatherv
- MPI_Iscatter
  - Non-blocking variant of MPI_Scatter
  - `int MPI_Iscatter(..., MPI_Request *req)`
- MPI_Iscatterv
  - Non-blocking version of MPI_Scatterv
  - `int MPI_Iscatterv(..., const int sendcount[], const int displs[], ..., MPI_Request *req )`

MPI_Gather

- Each process sends data/messages to root process.
- Root process receives and store messages in rank order
- Inverse of MPI_Scatter



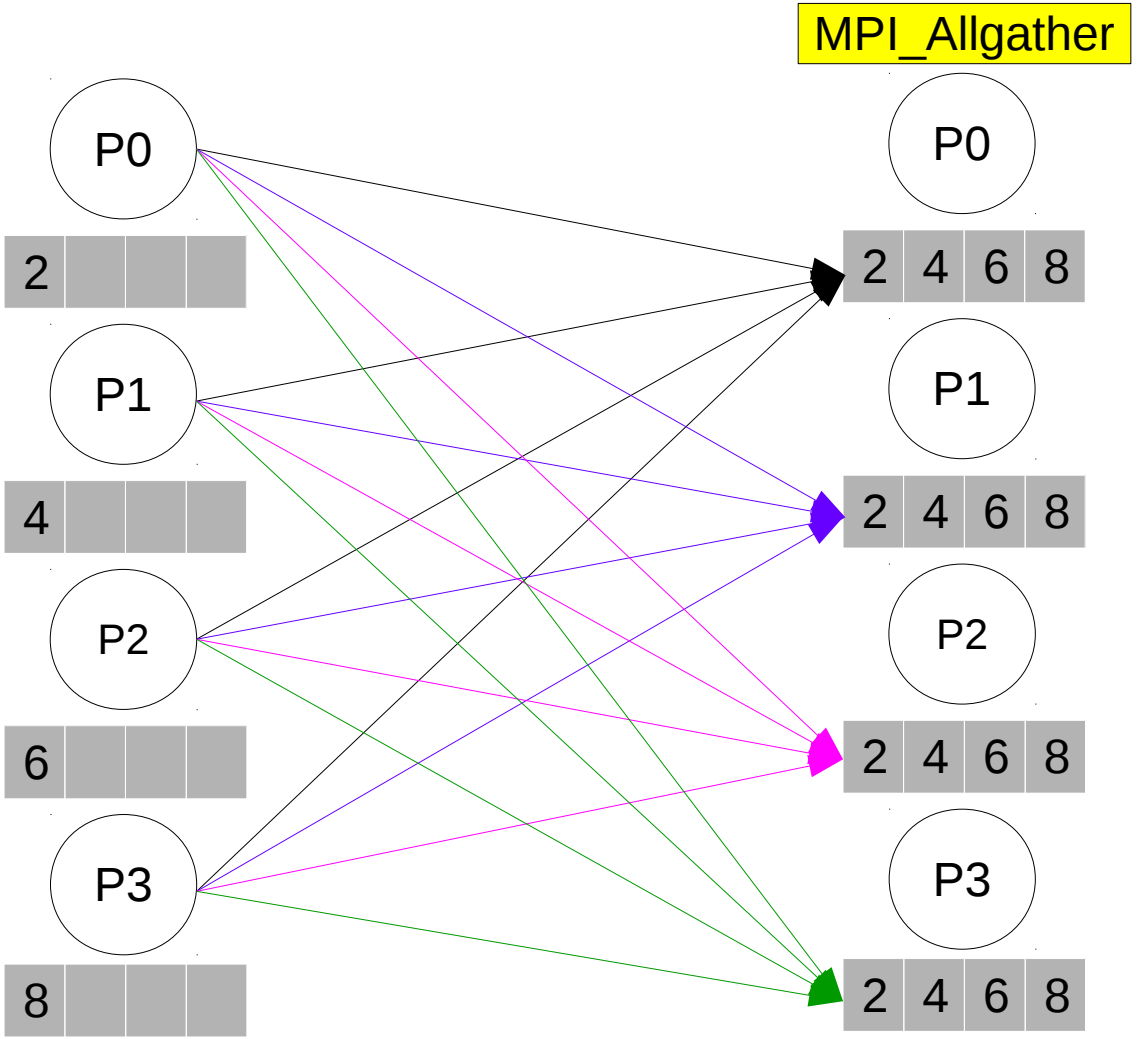
MPI Gather

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

- MPI_Gatherv
  - `int MPI_Gatherv(..., const int recvcount[], const int displs[], ...)`
  - Allows root process to receive varying count of data from each process
  - Inverse of MPI_Scatterv
- MPI_Igather
  - Non-blocking variant of MPI_Gather
  - `int MPI_Igather(..., MPI_Request *req)`
- MPI_Igatherv
  - Non-blocking variant of MPI_Gatherv
  - `int MPI_Igatherv(..., const int recvcount[], const int displs[], ..., MPI_Request *req)`

MPI_Allgather

- MPI_Gather with all processes receive result, instead of just root
- The data sent by *j-th* process received by all processes and placed at *j-th* block of the buffer



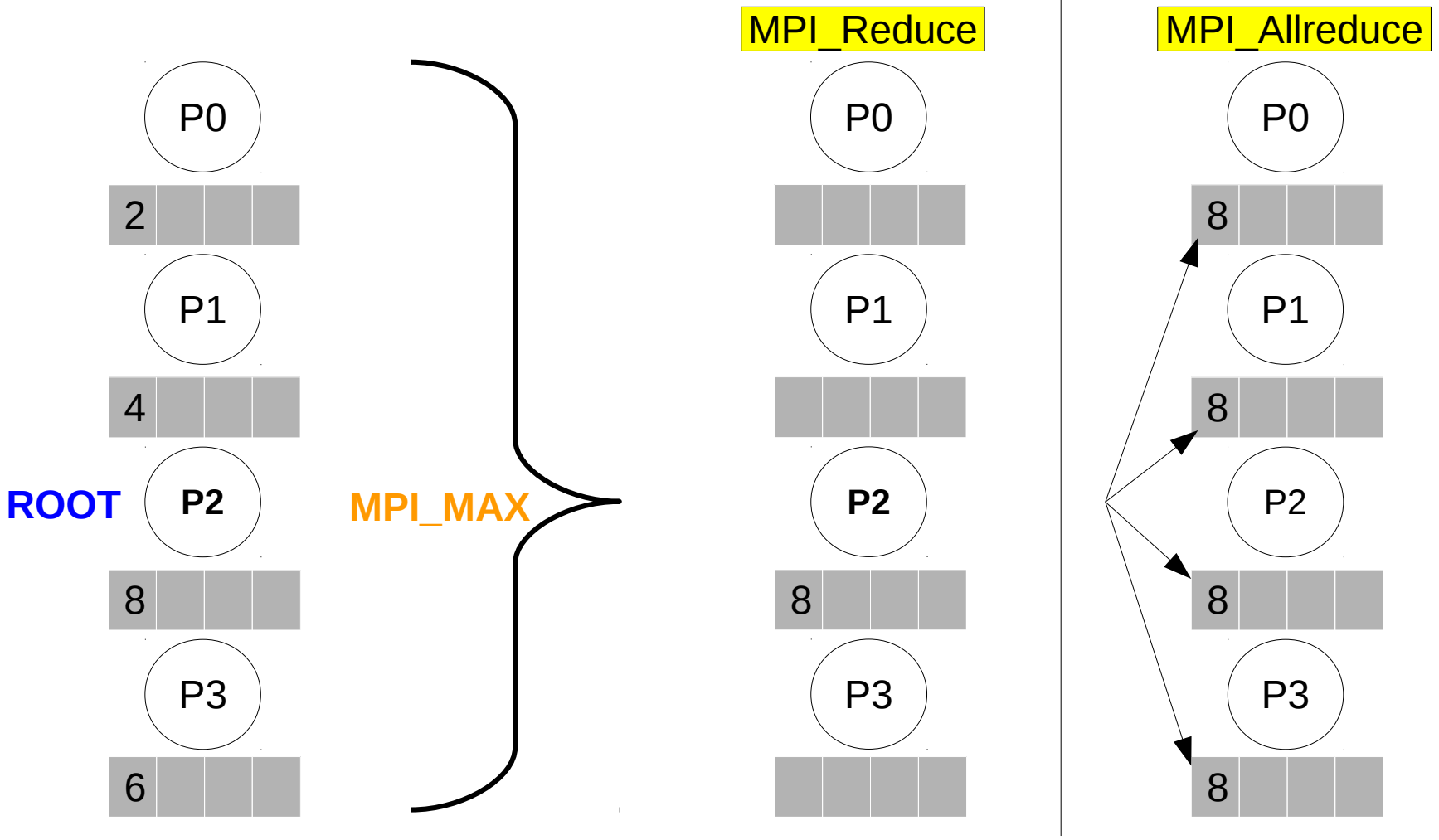
MPI Allgather

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

- MPI_Allgatherv
  - `int MPI_Allgatherv(..., const int recvcount[], const int displs[], ...)`
  - Similar to MPI_Gatherv, but where all processes receive result instead of just the root!
- MPI_Iallgather
  - Non-blocking variant of MPI_Allgather
  - `int MPI_Iallgather(..., MPI_Request *req)`
- MPI_Iallgatherv
  - Non-blocking variant of MPI_Allgatherv
  - `int MPI_Iallgatherv(..., const int recvcount[], const int displs[], ..., MPI_Request *req)`

MPI Reduce

- MPI_Reduce combines elements provided in input (send) buffer using specified operation and returns combined value in the output (receive) buffer of root process
- In MPI_Allreduce result is returned to all processes



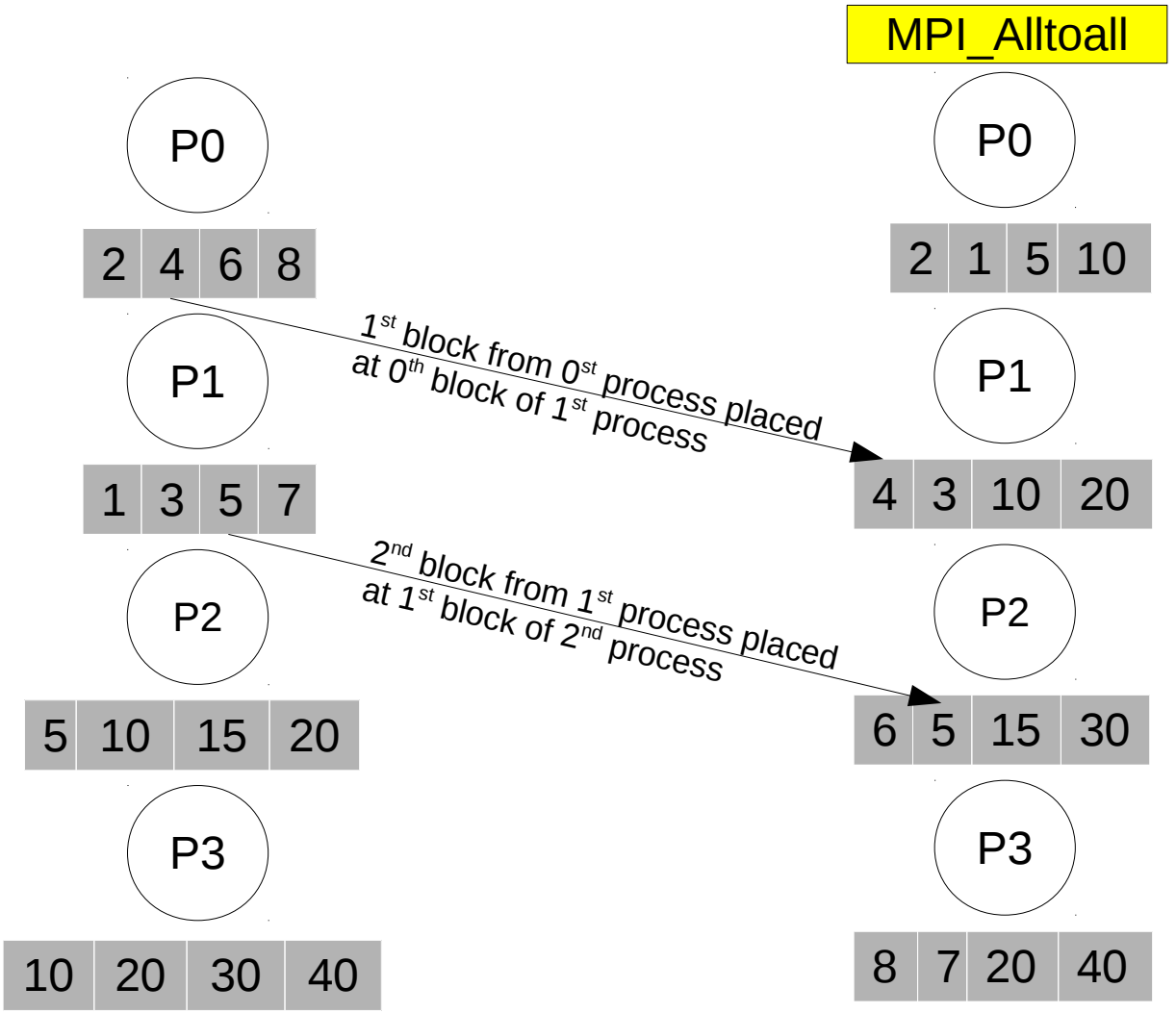
MPI Reduce

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

- MPI_Allreduce
  - `int MPI_Allreduce(...)` No root argument !
- MPI_Ireduce
  - Non-blocking variant of MPI_Reduce
  - `int MPI_Ireduce(..., MPI_Request *req)`
- MPI_Iallreduce
  - Non-blocking variant of MPI_Allreduce
  - `int MPI_Iallreduce(...,MPI_Request *req)`

MPI_Alltoall

- Each process sends distinct data to each receiver
- The *j-th* block sent from process *i* is received by process *j* and placed at *i-th* block of the buffer



MPI Alltoall

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

- MPI_Alltoallv
  - `int MPI_Alltoallv(..., const int sendcount[], const int sdispls[], ..., const int recvcount[], const int rdispls[], ...)`
  - Add flexibility with displacement arrays
- MPI_Ialltoall
  - Non-blocking variant of MPI_Alltoall
  - `int MPI_Ialltoall(..., MPI_Request *req)`
- MPI_Ialltoallv
  - Non-blocking variant of MPI_Allgatherv
  - `int MPI_Ialltoallv(..., const int sendcount[], const int sdispls[], ..., const int recvcount[], const int rdispls[], ..., MPI_Request *req)`

MPI Collectives Summary

- One-To-All
  - One process contributes to the result.
  - All processes receive the result.
  - The blocking, non-blocking and variable length collectives from
    - MPI_BCAST, MPI_SCATTER
- All-To-One
  - All processes contribute to the result.
  - One process receives the result.
  - The blocking, non-blocking and variable length collectives from
    - MPI_REDUCE, MPI_GATHER
- All-To-All
  - All processes contribute to the result.
  - All processes receive the result.
  - The blocking, non-blocking and variable length collectives from
    - MPI_ALLTOALL, MPI_ALLGATHER, MPI_ALLREDUCE, MPI_BARRIER

One-Sided Communication

- Facilitates Remote Memory Access (RMA)
- Remote write : MPI_PUT, MPI_RPUT
  - Transfer data from origin process (caller memory) to the target process memory
- Remote read : MPI_GET, MPI_RGET
  - Transfer data from target process memory to the origin process (caller memory)
- Remote update : MPI_ACCUMULATE, MPI_RACCUMULATE
  - Update location in the target process memory by values from origin (caller memory).

MPI I/O Support

- Partition of data among processes
- Transfer of global data structures between process memory and files
- collective and stand alone operations
  - MPI_File_read, MPI_File_read_all
- blocking and nonblocking I/O routines.
  - A blocking I/O call will not return until the I/O request is completed.
  - A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete.
  - MPI_File_iread, MPI_File_iread_all

MPI I/O functions

Functions	Description
MPI_File_open()	File opening. All processes must specify same mode.
MPI_File_close()	File closing. Ensure all outstanding operations associated with files are completed before closing
MPI_File_delete()	File deleting.
MPI_File_set_size()	File resizing.
MPI_File_preallocate()	Ensures storage space is allocated before use
MPI_File_set_info()	Specifies the hint
MPI_File_get_info()	Get the file hint
MPI_File_read()	Reads the file
MPI_File_write()	Writes the file
MPI_File_seek()	Updates the individual file pointer
MPI_File_get_position()	Get current position of individual file pointer

MPI Derived Data Types

- New data type defined based on built-in data types
- Derived data types provide a flexible tool to communicate complex data structures in an MPI environment
- Creating user defined data type
  - Construct with *MPI_Type**
  - Commit new data type with *MPI_Type_commit(MPI_Datatype * nt)*
  - Deallocate data type after use *MPI_Type_free(MPI_Datatype * nt)*
- Matching of data type in send and receive happens if specified basic data types match one by one, regardless of displacements

Reference : Advance MPI course - RRZE 2017

CUDA aware MPI

- MPI library can automatically detect the memory location i.e. pointer being passed is in device memory or host memory
- May take advantage of GPU direct RDMA technology
- MPI standard do not specify any guideline
- OpenMPI routines supporting CUDA awareness
  - All point to point (send-receive)and blocking collective APIs
  - Requires CUDA 4.0 or above
- CUDA aware MPI implementations
  - OpenMPI, Cray MPI, MVAPICH2, IBM SpectrumMPI etc.

More info at

i. https://www.open-mpi.org/faq/?category=runcuda#mpi-apis-cuda
ii. https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

Thank You !!!

References

- MPI standard – 3.1 (https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf)
- EPCC lecture series on MPI (search for EPCC + MPI on youtube) https://www.youtube.com/watch?v=0a0_JkQ7PLY&list=PLD0xgZGaUd1IV8VgXb1ggOLkEv19JmZiP&index=3
- MPI outside of C and Fortran https://blogs.cisco.com/performance/mpi-outside-of-c-and-fortran
- OpenMPI FAQ - https://www.open-mpi.org/faq/

Backup

MPI_Wait / MPI_Test

- Check completion of non-blocking communication
- MPI_Wait
  - *int MPI_Wait(MPI_Request *request, MPI_Status *status)*
  - Blocks the calling process/thread
  - Returns when operation identified by the request is complete.
  - Other variants *MPI_Waitany, MPI_Waitall, MPI_Waitsome*
- MPI_Test
  - *int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)*
  - Non-blocking call
  - Returns *true* flag if request is complete and *false* if request is incomplete
  - Other variants *MPI_Testany, MPI_Testall, MPI_Testsome*

Miscellaneous (1)

MPI_Reduce Operations	Description
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 exclusive *or (xor)*
MPI_BXOR	Bit-wise exclusive *or (xor)*
MPI_MAXLOC	Max value and location
MPI_MINLOC	Min value and location

Derived data type constructors	Description
Contiguous	Contiguous chunk of memory
Vector	Strided vector
Hvector	Strided vector in bytes
Indexed	Variable displacement
Hindexed	Variable displacement in bytes
Struct	General data type

MPI 'C' data types

- *MPI_CHAR*
- *MPI_SHORT*
- *MPI_INT*
- *MPI_FLOAT*
- *MPI_DOUBLE*
- *MPI_LONG*
- *MPI_C_BOOL*
- *MPI_BYTE*
- *MPI_PACKED*

Miscellaneous (2)

- MPI is thread safe.
- In multi-threaded environment one thread can send data to another thread of the same process using MPI function calls.
- MPI_Finalize should occur on the same thread (main thread) that initialized MPI or called MPI_Initialised
- MPI_Init_thread() : initializes MPI thread environment
- MPI generalized request is part of MPI external interfaces !
- MPI generalized requests allows user to create customized non-blocking operations using MPI interface
- Usually such code executes parallel to user code (MPI program) e.g. separate thread or signal handler
- MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)
  - Call back functions should be provided to query request or when request is freed or cancelled.

MPI Constants / Handles

MPI Constants / Handles	Comment
Constants	
MPI_SUCCESS	Constant representing *success*
MPI_ERR_*	There are number of error constants with respect to different error types
MPI_PROC_NULL	Dummy process used to specify dummy source or destination. Send/Receive with dummy process always succeed and has no effect.
MPI_ANY_SOURCE	Wild card value to represent any source. Generally matches to all senders.
MPI_ANY_TAG	Wild card value. Generally matches to all tags.
MPI_ROOT	Marks the root process (special value)
MPI_UNDEFINED	Represents/set undefined value
Null Handles	
MPI_COMM_NULL	Handle for invalid communicator
MPI_REQUEST_NULL	Handle for inactive request (when request is not associated with any ongoing communication)
MPI_FILE_NULL	Handle to closed/invalid file
Reserved Communicators Handles	
MPI_COMM_WORLD	Predefined communicator by MPI which allows communication with all processes accessible after MPI initialization
MPI_COMM_SELF	Communicator with one process (itself)

MPI Multicore awareness

- IBM Spectrum MPI multicore awareness
  - Direct shared memory implementation of MPI collective operations (SHMEM)
  - Policy-based, process-to-core affinity binding, including support for IBM Spectrum LSF® syntax
  - Multicore-aware copying and encoders/decoders optimized for intercore memory hierarchy