MPI Introduction

Ramakrishnan(Ramki) Kannan

Shruti Shivakumar

Motivated out to LLNL MPI Tutorial, OLCF MPI Tutorial, SC22 Parallel Computing 101 Tutorial

A General HPC Architecture

Multiple CPU cores on each **compute node** **share memory**

CPU CPU CPU CPU Memory

CPU CPU CPU CPU Memory

network

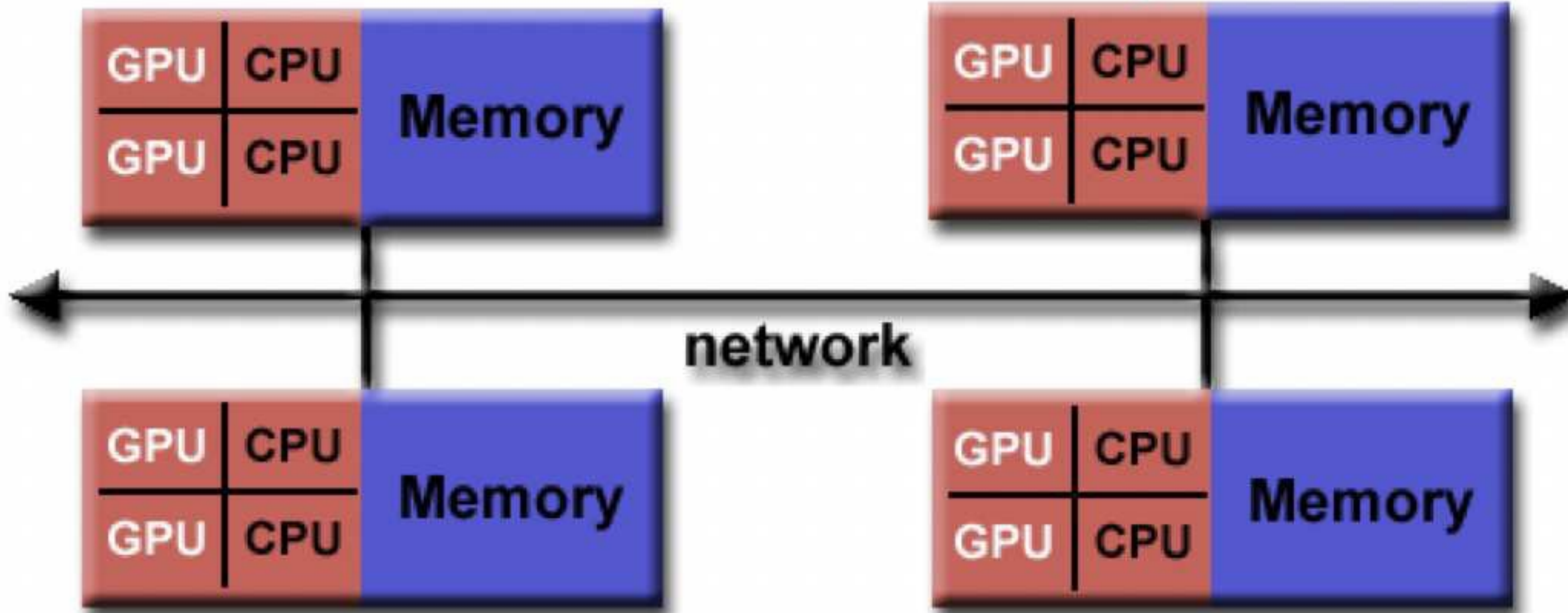
CPU CPU CPU CPU Memory

CPU CPU CPU CPU Memory

Shared Memory concept within a node

plus *Distributed Memory* concept: Non-local data can be sent across the network to other CPUs

HPC Architectures with Accelerators



Shared Memory within a node with CPUs and GPUs

plus *Distributed Memory* concept: Non-local data can be sent across the network to other CPUs

Frontier System

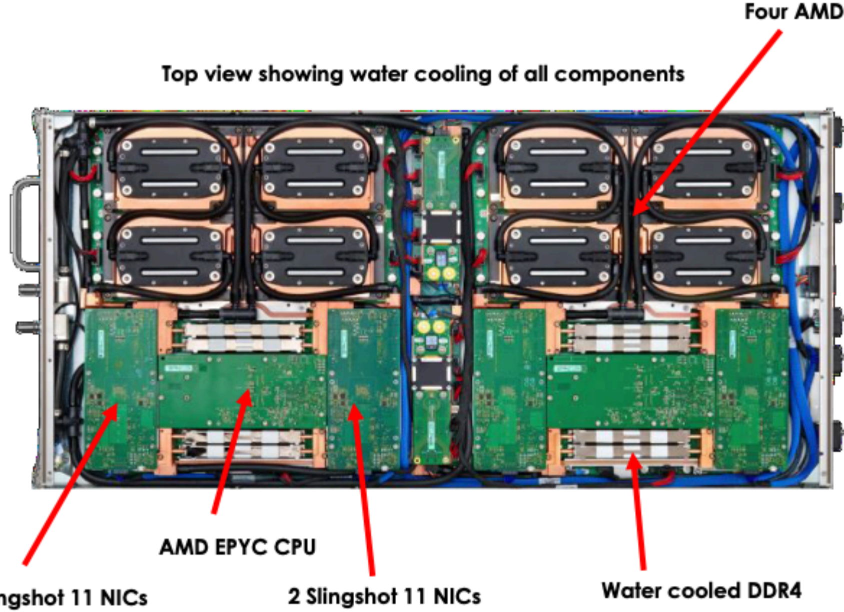
System	Titan (2012)	Summit (2017)	Frontier (2021)
Peak	27 PF	200 PF	2 EF
# nodes	18,688	4,608	9,408
Node	1 AMD Opteron CPU 1 NVIDIA Kepler GPU	2 IBM POWER9™ CPUs 6 NVIDIA Volta GPUs	1 AMD EPYC CPU 4 AMD Radeon Instinct GPUs
Memory	0.6 PB DDR3 + 0.1 PB GDDR	2.4 PB DDR4 + 0.4 HBM + 7.4 PB On-node storage	4.6 PB DDR4 + 4.6 PB HBM2e + 36 PB On-node storage, 66 TB/s Read 62 TB/s Write
On-node interconnect	PCI Gen2 No coherence across the node	NVIDIA NVLINK Coherent memory across the node	AMD Infinity Fabric Coherent memory across the node
System Interconnect	Cray Gemini network 6.4 GB/s	Mellanox Dual-port EDR IB 25 GB/s	Four-port Slingshot network 100 GB/s
Topology	3D Torus	Non-blocking Fat Tree	Dragonfly
Storage	32 PB, 1 TB/s, Lustre Filesystem	250 PB, 2.5 TB/s, IBM Spectrum Scale™ with GPFS™	695 PB HDD+11 PB Flash Performance Tier, 9.4 TB/s and 10 PB Metadata Flash. Lustre
Power	9 MW	13 MW	29 MW

Node 1 - AMD EPYC 7A53 64 Cores CPU
& 4 - AMD Instinct MI250X GPUs
Nodes per Blade - 2
Nodes per Cabinet - 128
Total cabinets - 74
Nodes per System – 9408
Total system memory: 9.2 PB (4.6 PB HBMe2 + 4.6 PB DDR4)
Total on-node NVM - 37 PB (66 TB/s read, 62 TB/s write)
Memory Bandwidth between HBM2e and each GPU - 3,200 GB/s (3.2 TB/s)
Memory Bandwidth between DDR4 and the CPU - 205 GB/s
Bandwidth between CPU and GPU - 288 GB/s

Report on ORNL Frontier System – Jack Donggarra and Al Geist https://icl.utk.edu/files/publications/2022/icl-utk-1570-2022.pdf

Frontier System

Four AMD MI200 GPUs

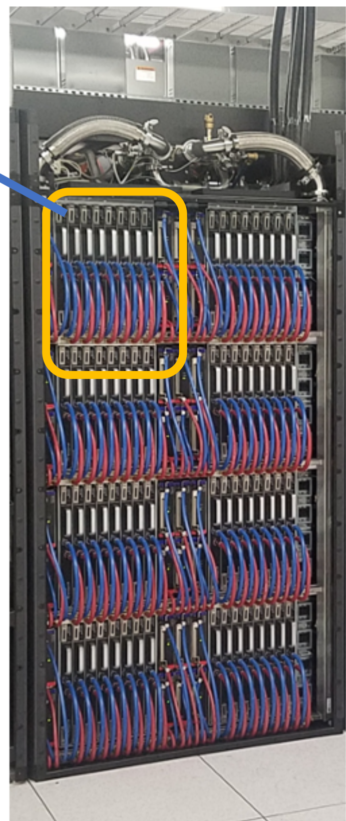


The rear has Slingshot 11 Connectors

and Power input

One Compute Blade

8 compute blades



Programming HPC Systems with Accelerators

- CPU Core parallelism – OpenMP
- Distributed parallel programming across Nodes -- MPI
- GPU Accelerator Programming – CUDA

Message Passing Interface (MPI)

- Communication standard
- MPI 1.0 was first introduced in 1994
- Works both on distributed and shared memory
- MPI 4.1 – May 2022
- This tutorial is about gentle introduction to complex MPI
- https://hpc-tutorials.llnl.gov/mpi/
- Code examples - https://github.com/mpitutorial/mpitutorial

Why MPI?

- Distributed memory model
  - Provides mechanisms to move data among disjoint processes
  - Can still be used within a node, but other strategies might be better (e.g. OpenMP
- Requires explicit code for parallelism
  - No magic from the compiler
  - No transparent large arrays spanning processes for example
- Why should I use MPI?
  - Many different alternate DM programming models exist – Map-reduce
  - Standardized - All HPC vendors support MPI; most scientific/HPC libraries support MPI; most parallel codes use MPI
  - Portable - MPI defines an API, so as long your code is MPI compliant and your implementation is too, your MPI parts should be portable
  - Functionality - Well over 400 routines
  - Performance - Implementations are encouraged to optimize for performance

Many different implementations

- OpenMPI
- MPICH
- Many vendors provided
  - IBM Spectrum MPI
  - Cray-MPICH
- C/C++, python and Fortran

Different MPI Functionalities

- Point-2-Point communications
- Collective Communications
- MPI-IO
- Tools Interface
- One sided message passing
- Derived Datatypes

Programming introduction

```
int main(int argc, char** argv) {
// Initialize the MPI environment. The two arguments to MPI Init are not
// currently used by MPI implementations, but are there in case future
// implementations might need the arguments.
MPI_Init(NULL, NULL);

// Get the number of processes
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the rank of the process
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Do something here.

// Finalize the MPI environment. No more MPI calls can be made after this
MPI_Finalize();
}
```

- Mpi.h
- Mpicc

Point to Point communication

- MPI provides four variants on send with blocking and nonblocking versions of each.
- Blocking means the call will not complete until the local data is safe to modify
- Nonblocking means the call returns “immediately”
    - Nonblocking data movement calls in MPI are MPI_I{command}, e.g. MPI_Irecv() or MPI_Ialltoallv() (capital “Eye”)
    - Nonblocking calls require a mechanism to tell when they are done – MPI_Wait*, MPI_Test*
    - Data may or may not actually move before a call to MPI_Wait*/MPI_Test*
    - It is not safe to reuse buffers until the Wait/Test says the operation is locally done.
    - Nonblocking calls (can) allow for compute and communication to overlap

Three Illustrations

- Point to point send and receive
- Ping pong – Two people sending back and forth messages
- Ring – A broadcast variant

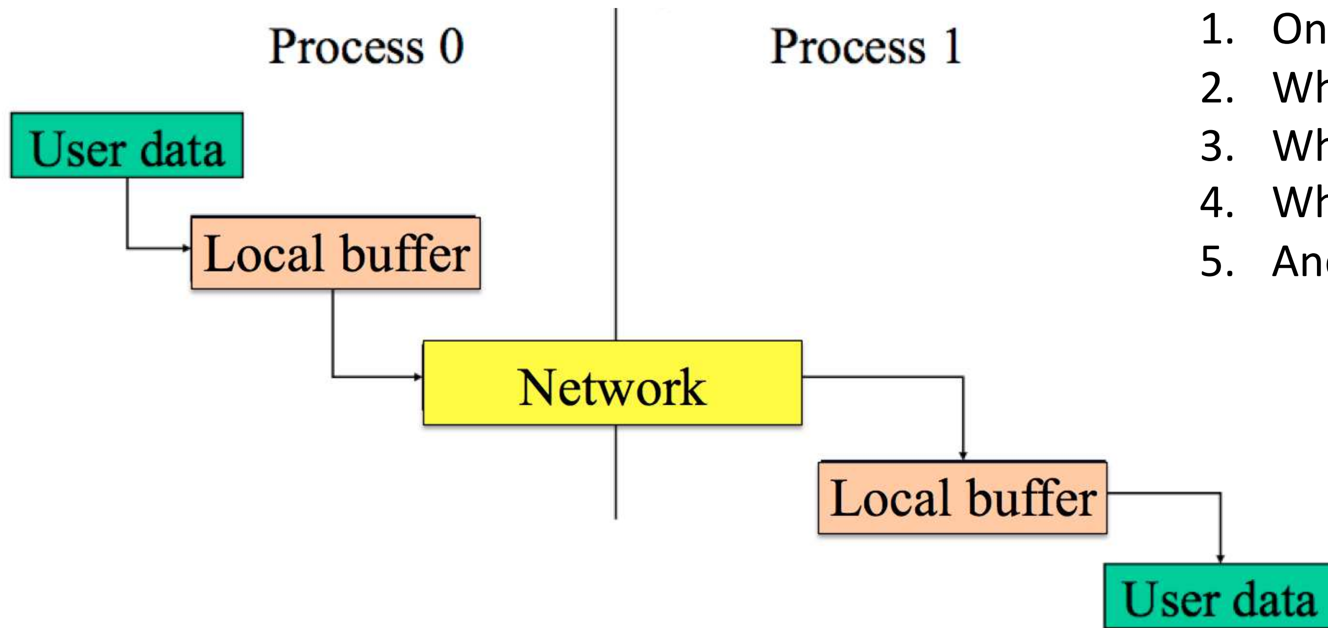
Deadlocks

- Blocking point-to-point calls make it possible to deadlock a program

Process 0	Process 1
MPI_Recv(from process 1) MPI_Send(to process 1)	MPI_Recv(from process 0) MPI_Send(to process 0)

- Use nonblocking calls
- Have odd numbered processes post Sends first
- Use MPI_Sendrecv() call

Different Send and Receive Types



When do you call a send is complete?

1. Once I call MPI Library (MPI_Isend). I don't care what MPI does
2. When MPI library gave it to Network layer? (MPI_Bsend)
3. When a receiver identified to receive the data? (MPI_send)
4. When the receiver got the data? (MPI_Ssend)
5. And other complex combinations.

https://iamsorush.com/posts/mpi-send-types/ - For different

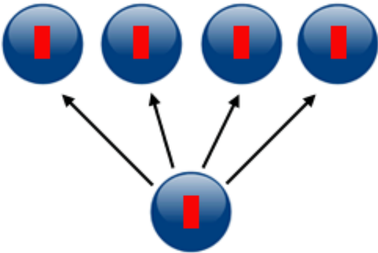
Non-blocking send and receive

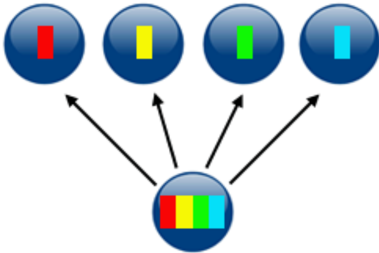
- MPI_ISEND(), MPI_IRECV(), MPI_WAIT()
- MPI_WAIT is a blocking call – (MPI_ISEND + MPI_WAIT = MPI_SEND)
- We can check if we the call is completed and the data is obtained with a test call – MPI_Test(). If the flag is true, the call is complete. While(MPI_Test()) is same as MPI_Wait()
- Non-blocking operation is slightly complicated. But boosts performance
- Mainly used for look-ahead techniques – overlapping compute and communication.

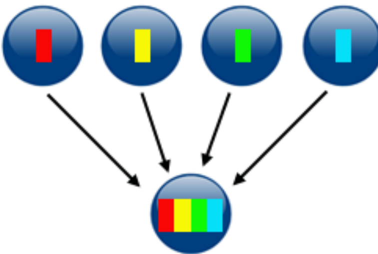
```
MPI_Request reqs[4]; // required variable for non-
blocking calls
MPI_Status stats[4]; // required variable for Waitall
routine
// determine left and right neighbors
prev = rank - 1;
next = rank + 1;
if (rank == 0)
prev = numtasks - 1;
if (rank == (numtasks - 1))
next = 0;
// post non-blocking receives and sends for neighbors
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1,
MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2,
MPI_COMM_WORLD, &reqs[1]);

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

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

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

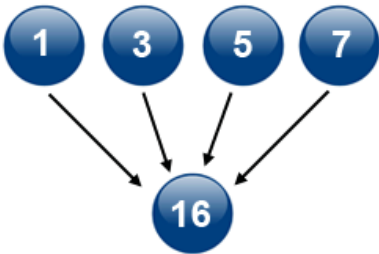
Collective Communication

- One process wants to communicate with multiple process
  - One Process to all other process

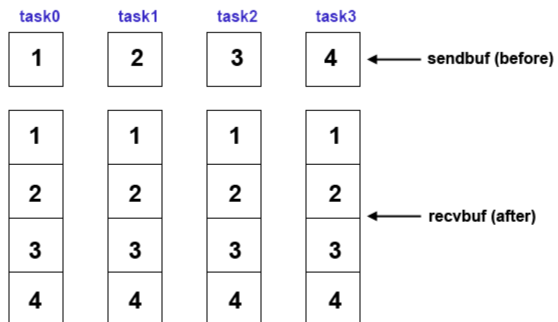


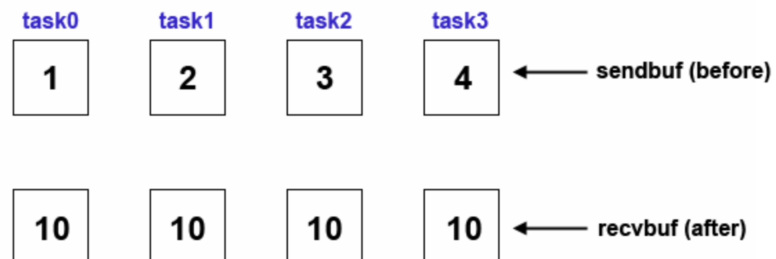


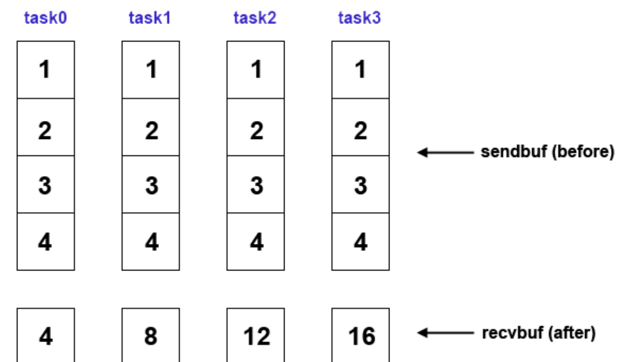


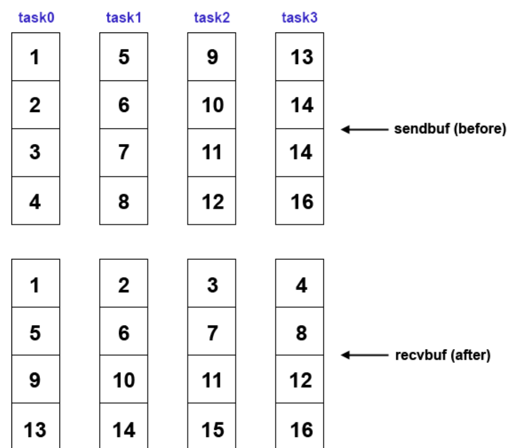


MPI_All*





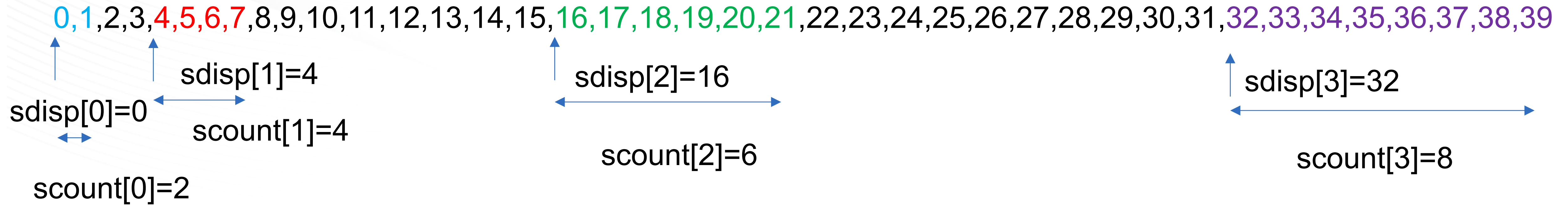




MPI_*v class

- Scatterv, gatherv, allgatherv, alltoallv, alltoallw
- Each process can contribute a different amount of data
- Take an array of counts and an array of displacements (distance between each element)
- These function calls can be very expensive for memory and data movement – Alltoallv requires 4 arrays of size(commsize) ints, plus the actual data

Scatterv example



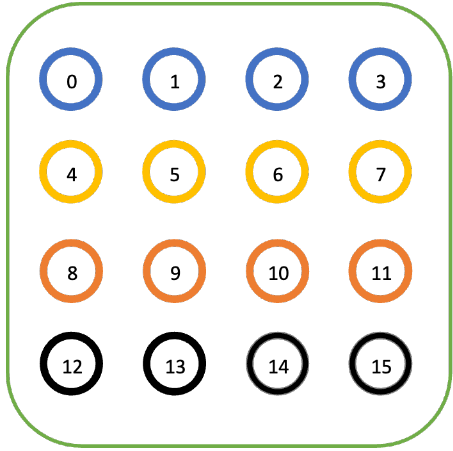
Rank 0 (also the root) Recvbuf 0,1

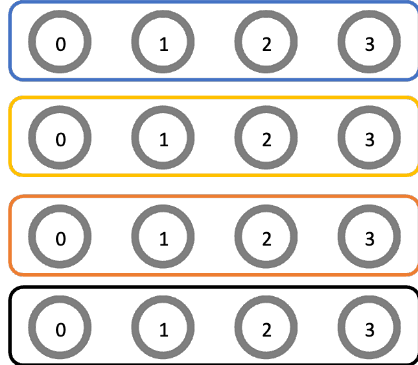
Rank 1 Recvbuf 4,5,6,7

Rank 2 Recvbuf 16,17,18,19,20,21

Rank 3 Recvbuf 32,33,34,35,36,37,38,39

MPI Communicator Split






```
// Get the rank and size in the original
communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color
based on row

// Split the communicator based on the color and
use the original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank,
&row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d --- ROW RANK/SIZE:
%d/%d\n",
world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);

MPI_Finalize();
```