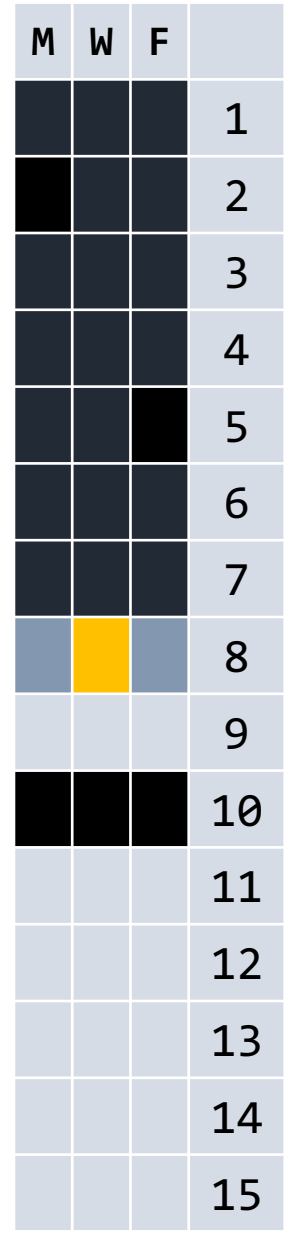
CMOR 421/521:

MPI: Collectives



Topics

- MPI Collective Operations
  - What are they?
  - When to use them?

Collective Operations

- Without shared memory, for many problems we now have to introduce communication in order to parallelize computation
- Just as computation has common processing patterns when in parallel, so does communication
- OpenMP provided constructs for easily parallelizing common processing patterns
- MPI provides support for common communicating patterns seen in parallel computing

Collective Operations

- Send/Recv and Isend/Irecv are point-to-point communication
  - They allow one process to speak to one other process
- It is common for communication to involve more than just 2 processes (often all)
- Collective communication is communication that involves a collection of processes

Library Code vs Hand-Written

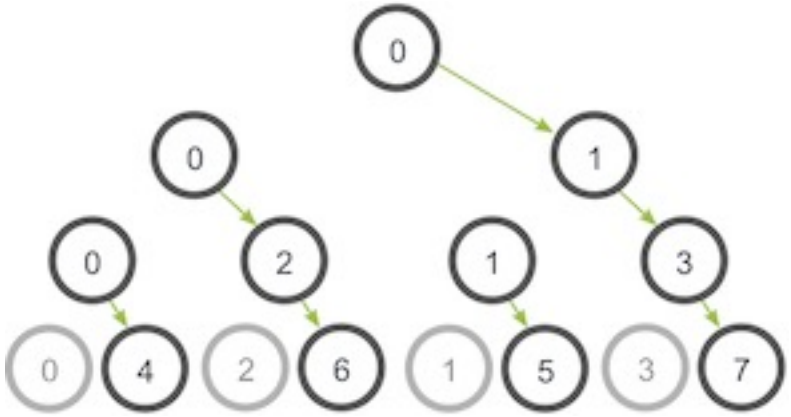
- Most of the collective operations in MPI are wrappers which use Send/Recv and/or Isend/Irecv
- Why use them if we can do it ourselves? *Efficiency*.
- Consider the problem of one rank sending to all others.

How to implement broadcasting?

- Suppose one rank (rank 0) has information we'd like all the other ranks to have.
- How to send messages to each rank?
- Naïve approach: O(n_ranks) communications

A more efficient implementation of broadcasting

- Better tree-like approach: O(lg(n_ranks))
  - Each rank sends to another one, so that 1, 2, 4, 8, … ranks have info
- Step 1: R0 -> R1; Now both R0 **AND** R1 have the info
- Step 2: R0 -> R2
  R1 -> R3, Now R0, R1, R2, R3 have the info…
- Step 3: R0 -> R4, R1 -> R5
  R2 -> R6, R3 -> R7…



Hand-Written vs Library Code

- The naïve broadcasting implementation (O(n_ranks)) is what most people would start with
- The branching/tree/logarithmic implementation is harder to do, but has much better performance
  - There are also potential deadlock issues!
- Library code will have efficiency and safety issues solved—you don't have to think about them yourself
  - Exception: thread safety is not always guaranteed, check documentation for your specific implementation.

New MPI routines

- MPI Collective Operations
  - Broadcast
  - Reduction
  - Gather
  - Scatter

- Additional variations to come

What to Keep in Mind: Mappings

- Some collectives seem very similar
- They differ in the ranks they involve and how much of the data is communicated to each rank
  - **Ranks:** Collectives communicate/map data from one or all ranks to one or all ranks in a communicator
  - **Data:** Collectives communicate/map the entirety (whole or single if just 1 variable) or part of a data set between ranks
- Collective operations are meant to provide functionality for **common** processing patterns
- More specialized communication is left to the user

Broadcasting

- Mapping Guide:
  - Ranks: MPI_Bcast: one (rank) -> all (ranks)
  - Data: MPI_Bcast: whole -> whole
- This is the problem we've previously discussed: sending the same set of data from one rank to all others
- Implementations of MPI generally use the efficient tree/branching implementation.

Visual MPI Collective Cheat Sheet

- MPI_Bcast:

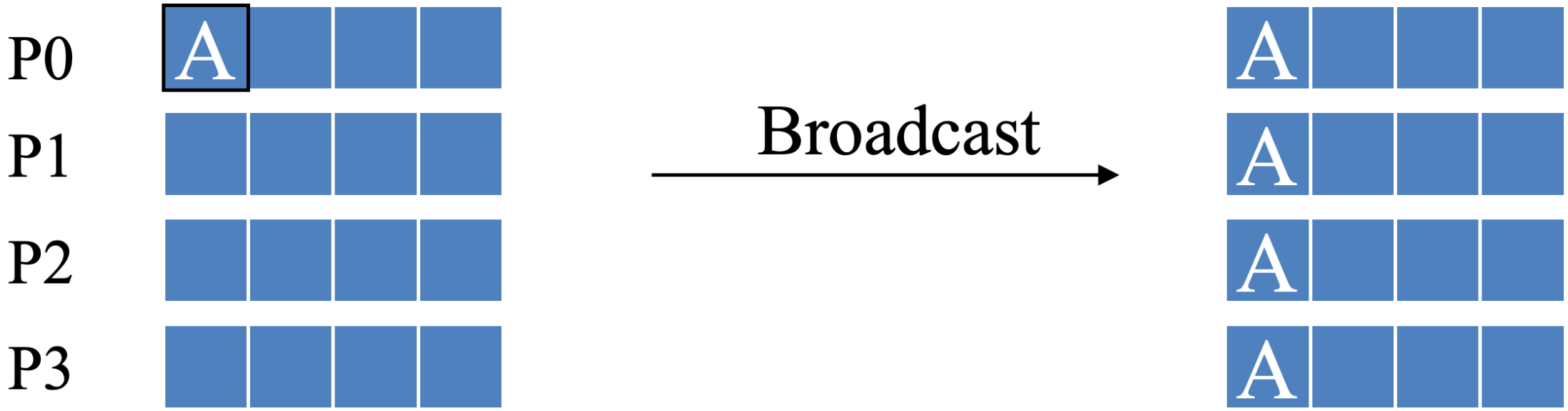


Image from Aydin Buluc

MPI_Bcast

```
int MPI_Bcast(
    void* buffer,
    int count,
    MPI_Datatype data_type,
    int root_rank,
    MPI_Comm comm
);

// Note: no tag anymore
```

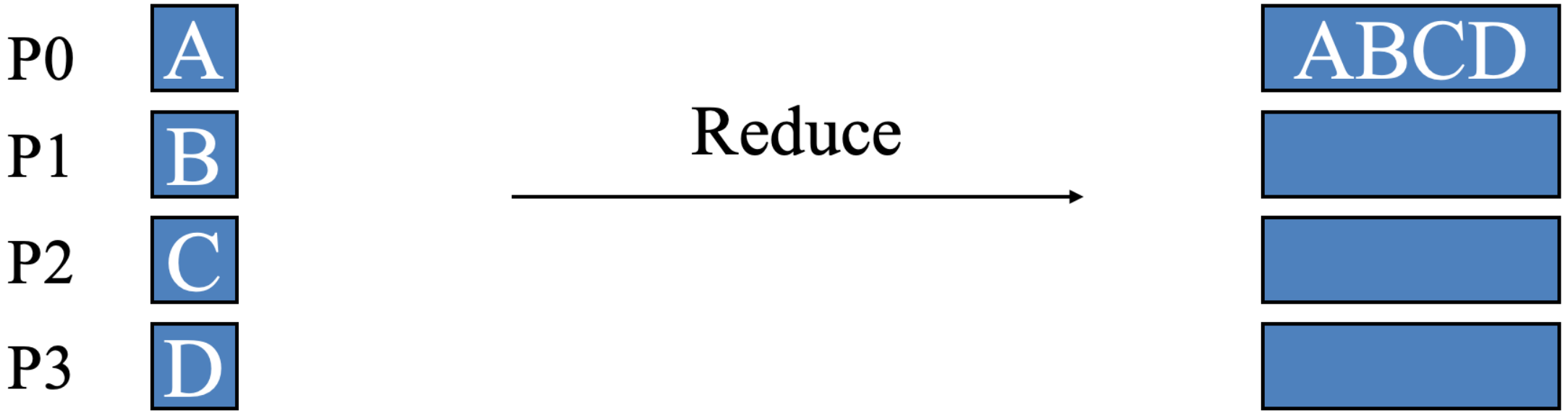
Argument	Description
buffer	Root rank: the data to be sent
	Other ranks: where to put the received data
count	How many entries to send
data_type	Data type of the data to be sent
root_rank	The rank with the info to send
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

Reductions

- Mapping Guide:
  - Ranks: MPI_Reduce: all (ranks) -> one (rank)
  - Data: MPI_Reduce: whole/single -> whole/single
- Like OpenMP's reduction clause, defined for common reduction operations (sum, min, max,…)
- Communicates and reduces the answer on each rank (the local answer) to a global answer
- **Only one rank gets the result though**

Visual MPI Collective Cheat Sheet

- `MPI_Reduce:`



MPI_Reduce

```
int MPI_Reduce(
    const void* sendbuf,
    void* recvbuf
    int count,
    MPI_Datatype data_type,
    MPI_Op operation,
    int root_rank
    MPI_Comm comm
);
```

Argument	Description
sendbuf	All: the data to reduce
recvbuf	Root: where to store the results of the reduction
	Others: not needed (pass NULL)
count	the number of reduction sets/the rank-wise length of the data
data_type	Data type of the data to be sent
operation	The operation to reduce wrt
root_rank	The rank to receive the results
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

MPI Reduction Operations

Operation	MPI_OP
minimum	MPI_MIN
maximum	MPI_MAX
sum	MPI_SUM
product	MPI_PROD

There are others too (logical operations, arg min/max + rank)

You can also provide your own reduction operator via MPI_OP_CREATE (stores the result)

Gather

- Mapping Guide:
  - Ranks: MPI_Gather: all (ranks) -> one (rank)
  - Data: MPI_Gather: part -> whole

- Gather collects data from all ranks and stores it on the root rank

- Can send an array from each rank

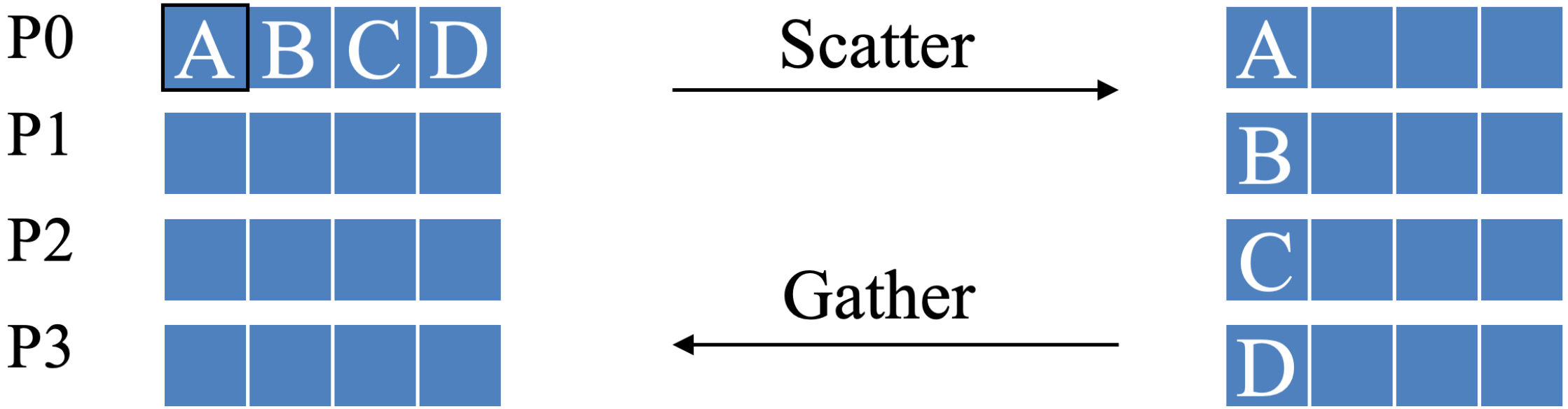
MPI_Gather

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

Argument	Description
sendbuf	ALL: Data to send to the root rank
sendcount	How many elements each rank should send
sendtype	Data type of the send data
recvbuf	Root: where to store the recv'd data
	Others: can be NULL
recvcount	How many elements to recv from each rank
recv_type	Data type of the recv'd data
root_rank	Rank to receive the result
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

Visual MPI Collective Cheat Sheet

- MPI_Gather:



Scatter

- Mapping:
  - Ranks: MPI_Scatter: one (rank) -> all (ranks)
  - Data: MPI_Scatter: whole -> part

- One rank has an array
- An equal portion of that array is sent to each rank
- Can be thought of as the “inverse” of MPI_Gather

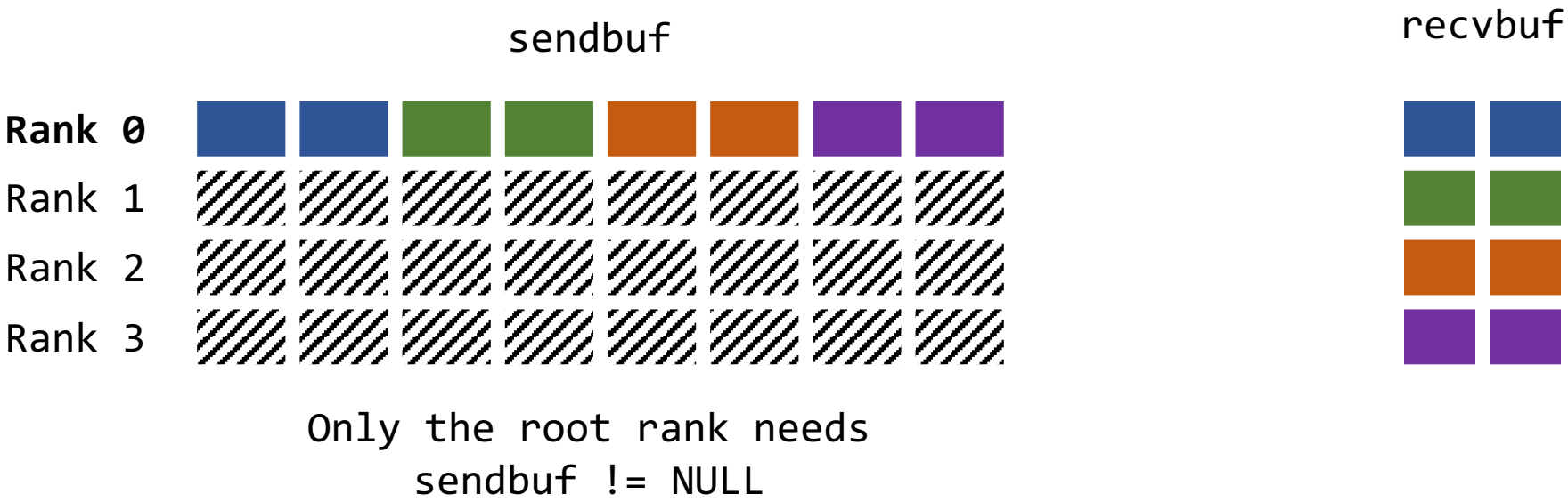
MPI_Scatter

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

Argument	Description
sendbuf	Root: Data to send from the root rank
	Others: can be NULL
sendcount	How many elements each rank should send
sendtype	Data type of the send data
recvbuf	All: where to store the recv'd data
recvcount	How many elements each rank will receive
recv_type	Data type of the recv'd data
root_rank	Sending rank
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

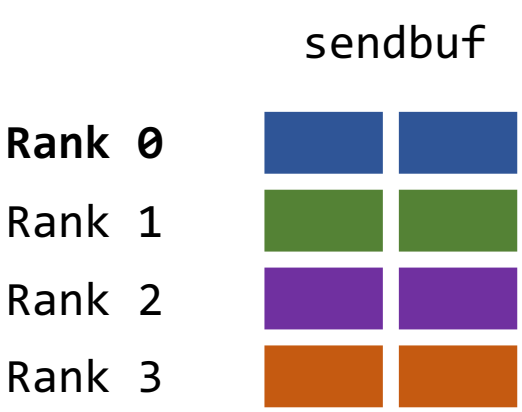
Visual MPI Collective Cheat Sheet

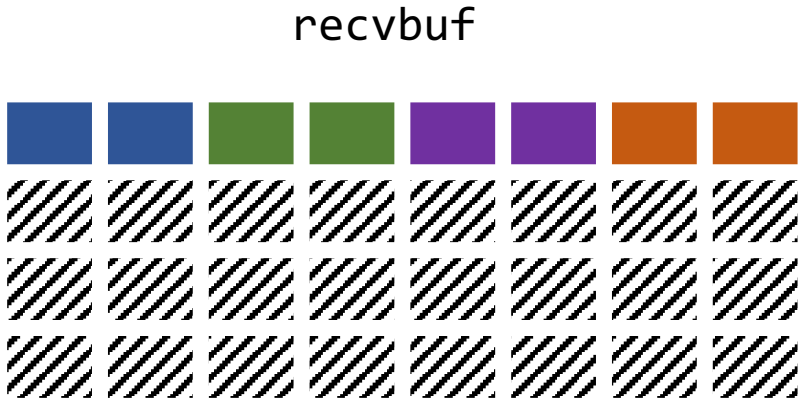
- MPI_Scatter: root=0, sendcount=2, recvcount=1,



Visual MPI Collective Cheat Sheet

- MPI_Gather: root=0, sendcount=2, recvcount=1,
- “Inverse” of MPI_Scatter





Only the root rank needs recvbuf != NULL

Collectives and Blocking

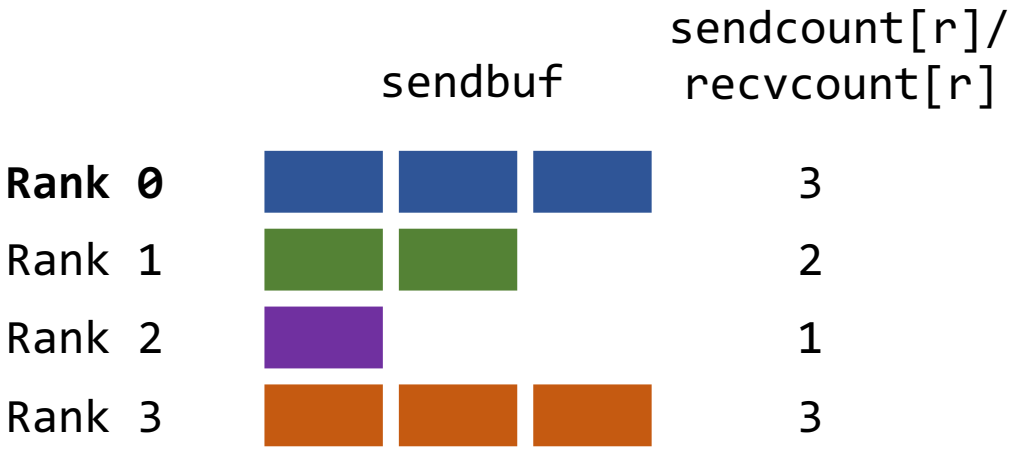
- Bcast, Reduce, Gather, and Scatter are *blocking*
- How they block is tied to the send/recv buffers like Send/Recv: when the buffers are safe to use the process can move on
- They have asynchronous counterparts:
  - MPI_Ibcast, MPI_Ireduce, MPI_Igather, MPI_Iscatter
  - As before, a MPI_Request argument is added to the function arguments

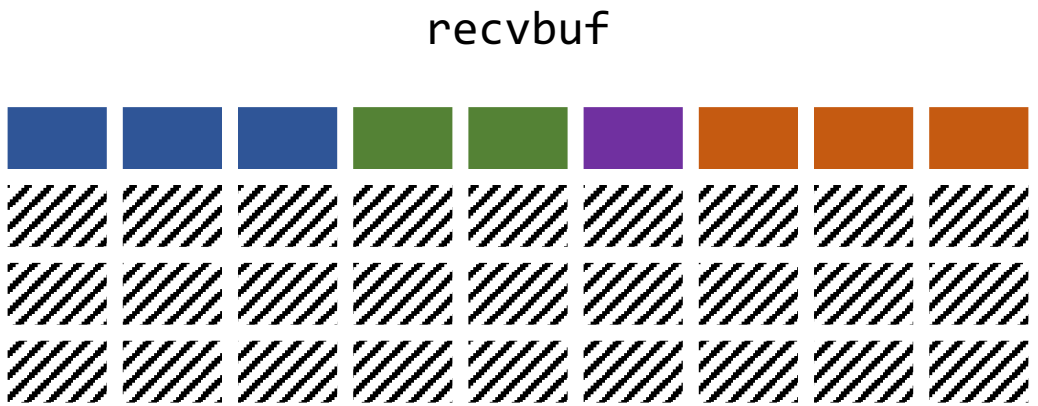
Variable counts

- What if some ranks have more (or less) data to send than others?
  - i.e. we want count/sendcount/recvcount to be different for each rank
- MPI does that too!
  - Not applicable to broadcasting and reductions
  - Gather: MPI_Gatherv (v is for “vector”…)
  - Scatter: MPI_Scatterv (…but should be for “variable”)
  - sendcount/recvcount: int -> int*
  - Example: sendcount[r] = send count for rank r

Custom Counts: Gather Example

- MPI_Gather: root=0, sendcount={…}, recvcount={…},





Only the root rank needs
recvbuf != NULL

Visibility and Mem Allocation

- All processes must participate in collective communication
  - Collective function calls need to be visible to all ranks so they know to participate (otherwise can deadlock)
- Allocation of send/recv buffers can be localized
  - Some ranks may only recv, so they don’t need to allocate a send buffer (and similarly for sending)
  - Rather than using if statements around function calls, you can use if statements around allocations of arrays

Visibility and Mem Allocation

- Some ranks don't need a send/recv buffer; can pass NULL in these cases
- Prefer using if statements when allocating memory rather than when calling (ensures collectives visible to all ranks)

```
// Okay but icky; easy to make mistakes
if (rank == root)
    MPI_Gather(sendbuf, sendcount, MPI_INT, // they all need to send
               recvbuf, recvcount, MPI_INT,
               root, MPI_COMM_WORLD);
else
    MPI_Gather(sendbuf, sendcount, MPI_INT,
               NULL, recvcount, MPI_INT,
               root, MPI_COMM_WORLD);
```

Visibility and Mem Allocation

- Some ranks don't need a send/recv buffer; can pass NULL in these cases
- For collectives, better to use if statements when allocating memory rather than when making calls

```
// Nicer version; still easy to make mistakes
int* sendbuf = rank == root ? NULL       : new int[N];
int* recvbuf = rank == root ? new int[N] : NULL;

MPI_Gather(sendbuf, sendcount, MPI_INT,
           recvbuf, recvcount, MPI_INT,
           root, MPI_COMM_WORLD);
```

Deadlocks and Collectives

- Deadlocks can still happen with collectives
- Visibility: all processes should be able see the call to a collective operations
  - Specifying the root is all you need to do
  - You do not need to protect it with an “if (rank == …)” statement. If you do, it will cause a deadlock!
- For collectives that map to/from a single rank, you will specify that rank in the function call
  - Again, this is called the root rank

Visibility and Deadlocks

- MPI collective do not deadlock from Send/Recv order mismatch
- But all ranks are expected to participate
- **Example 1:** Deadlock!
  - All non-root ranks cannot see the call to communicate, so they cannot participate

```
// Deadlock!
if (rank == root)
    MPI_Bcast(buffer, count, MPI_INT, root, MPI_COMM_WORLD);
```

In summary

- MPI collectives simplify common communication patterns
- MPI Collectives Operations
  - Broadcasting (1 -> all)
  - Reductions (all -> 1)
  - Gather (all -> 1)
  - Scatter (1 (n) -> all)

MPI “All” collectives

- What if we need the result of a reduction on all ranks?
  - MPI_Reduce + MPI_Bcast
- What if all ranks have info to broadcast?

Example Problem: Reduction

- Normalizing an array
  - Find the maximum element (assume positive)
- We can use MPI_Reduce to find the max
- But only the root rank has the final answer
  - Reduce and then broadcast?
  - It is common for all ranks to need the answer of a reduction…

More things to think about…

- Gather:
  - Gather sends data from all ranks to one rank
  - What if all ranks need the gathered information?
- Broadcast and Scatter:
  - Broadcast sends the entirety of tis data from one rank to all ranks
  - Scatter sends a part of one rank’s data to all ranks
  - What if we want all ranks to send information to all others?

Many-to-many MPI collectives

- MPI collective operations
  - Broadcasting (1 -> all) **=> All-to-all**
  - Reductions (all -> 1) **=> All-reduce**
  - Gather (all -> 1) **=> All-gather**
  - Scatter (1 (n) -> all) => ***All-to-all***
- **There is no root rank now for these**

What to Keep in Mind: Mappings

- Some collectives seem very similar
- They differ in the ranks they involve and how much of the data is communicated to each rank
  - **Ranks:** Collectives communicate/map data from one or all ranks to one or all ranks in a communicator
  - **Data:** Collectives communicate/map the entirety or part of a data set between ranks
- For many-to-many collectives, this is even harder

All-to-All

- Mapping Guide:
  - Ranks: MPI_Alltoall: all (ranks) -> all (ranks)
  - Data: MPI_Alltoall: part -> part
- Each rank sends and receives data from/to all other ranks; the resulting data each rank receives is not the same, however.
- More like an abstraction of Scatter than Broadcast

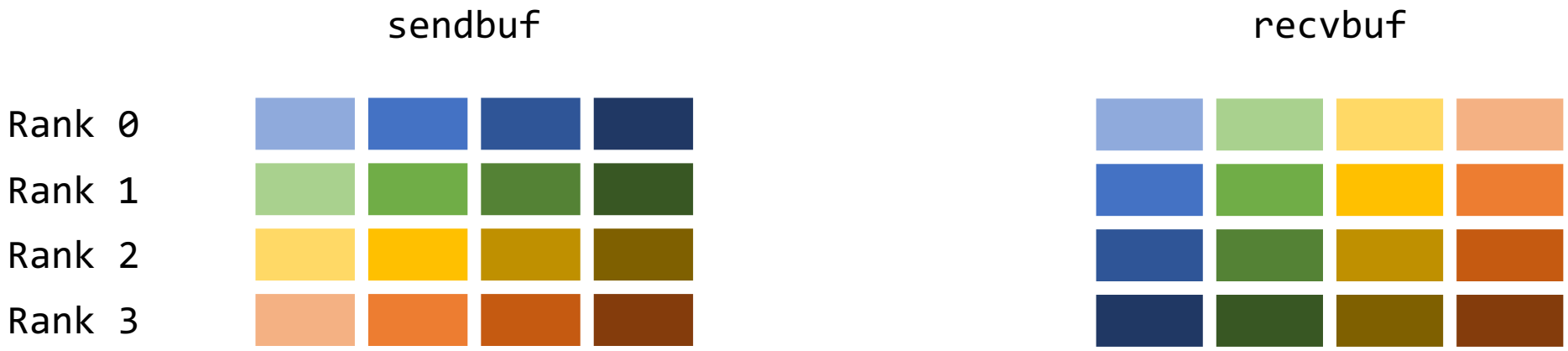
MPI_Alltoall

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

Argument	Description
sendbuf	Data to send
sendcount	How many elements each rank should send
sendtype	Data type of the send data
recvbuf	Where to store recv'ed data
recvcount	How many elements to recv from each rank
recv_type	Data type of the recv'd data
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

Visual MPI Collective Cheat Sheet

- MPI_Alltoall: sendcount = recvcount = 1
- Similar to matrix transposition



All-Reduce

- Mapping Guide:
  - Ranks: MPI_Allreduce: all (ranks) -> all (ranks)
  - Data: MPI_Allreduce: whole/single -> whole/single
- Each rank sends its portion of the data to be reduced
- Each rank gets the result of the reduction

MPI_Allreduce

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

Argument	Description
sendbuf	The data to reduce
recvbuf	Where to store the results of the reduction
count	the number of reduction sets/the rank-wise length of the data
data_type	Data type of the data
operation	The operation to reduce wrt
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

Visual MPI Collective Cheat Sheet

- `MPI_Allreduce: count=5, op=max`

- `recvbuf[i] = max(sendbuf[i], rank=0,…, 3)`
- **All ranks need to allocate recvbuf now**

All-Gather

- Mapping Guide:
  - Ranks: MPI_Allgather: all (ranks) -> all (ranks)
  - Data: MPI_Allgather: part -> whole
- Each rank sends and receives data from/to all other ranks
- All ranks have the same result: the collection of the data sent from all ranks

MPI_Allgather

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

Argument	Description
sendbuf	Data to send
sendcount	How many elements each rank should send
sendtype	Data type of the send data
recvbuf	Where to store recv'ed data
recvcount	How many elements to recv from each rank
recv_type	Data type of the recv'd data
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

Visual MPI Collective Cheat Sheet

- MPI_Allgather: sendcount = recvcount = 2,

Visual MPI Collective Cheat Sheet

- Another interpretation of MPI_Allgather

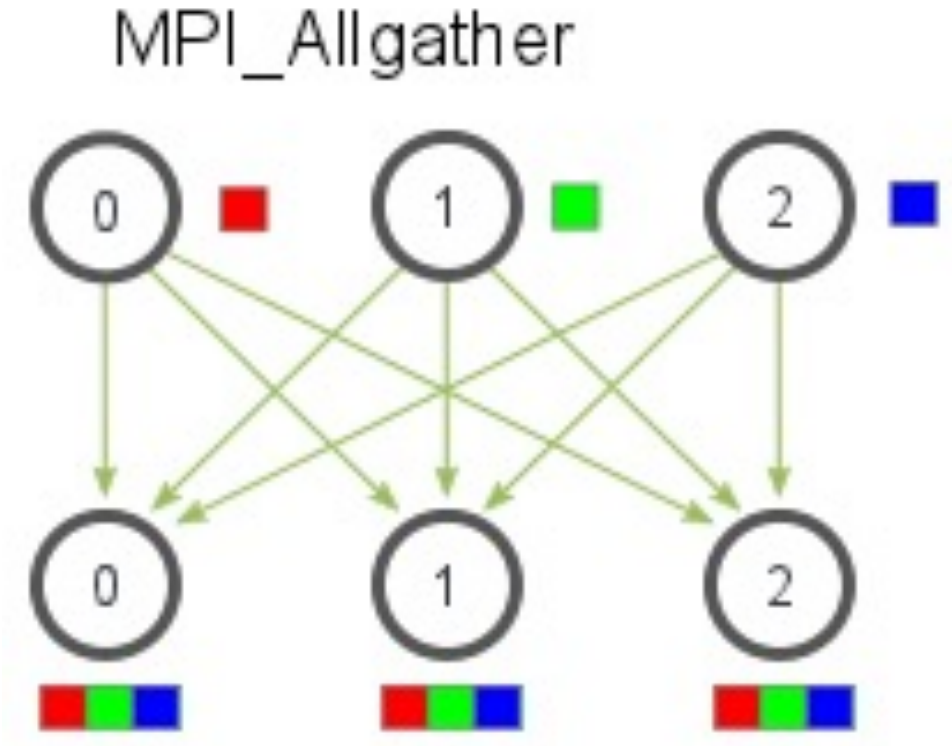


Image from Wes Kendall's MPI Tutorial

All-Type Collectives and Blocking

- MPI_Alltoall, MPI_Allreduce, and MPI_Allgather are blocking
- They have asynchronous counterparts:
  - MPI_Ialltoall
  - MPI_Iallreduce
  - MPI_Iallgather
  - As before, a MPI_Request argument is added to the function's parameters

Custom Counts and Types

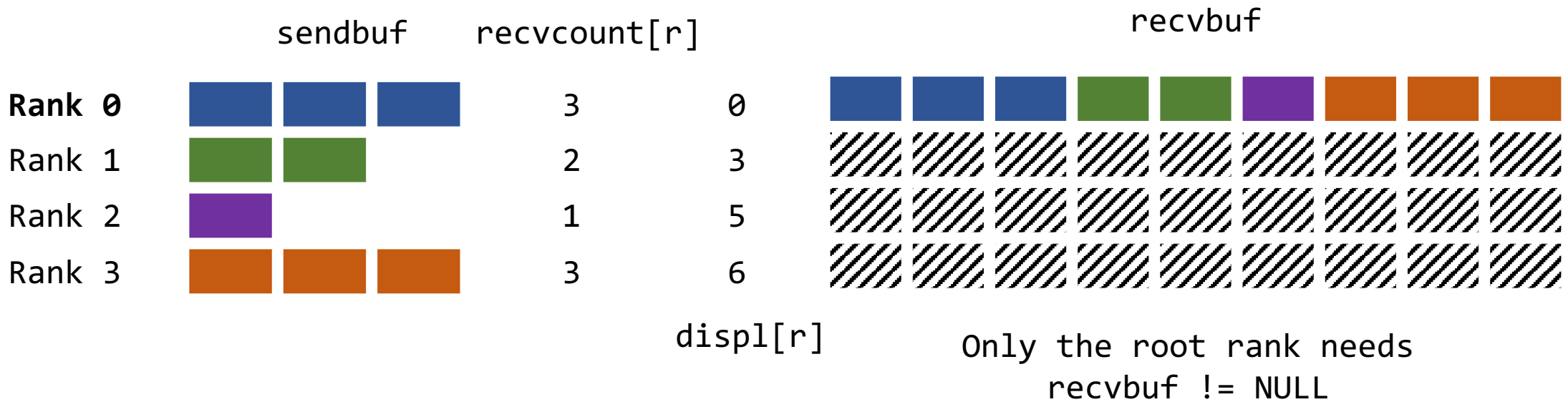
- What if some ranks have more (or less) data to send than others?
  - i.e. we want something like MPI_Gatherv, MPI_Scatterv
- What if some ranks have ints to send and another doubles?
  - i.e., different data types for each rank

Custom Counts and Types

- Yes, MPI can do that too!
- Custom counts by rank:
  - MPI_Alltoallv
  - MPI_Allgatherv
- Custom types by rank:
  - MPI_Alltoallw
  - Not going to look at; just know it exists

Custom Counts: Gather Example

- MPI_Gather: root=0, sendcount=?, recvcount={…},



MPI_Gatherv

```
int MPI_Gatherv(
    const void* sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void* recvbuf,
    const int recvcounts[],
    const in displ[],
    MPI_Datatype recvtype,
    int root,
    MPI_Comm comm
);
```

Argument	Description
sendbuf	Data to send
sendcount	How many elements each rank should send (unique to each rank)
sendtype	Data type of the send data
recvbuf	Where to store recv'ed data
recvcounts	Array telling how many elements to receive from each rank recvcounts[r] -> rank r
displ	The starting index into recvbuf for data from each rank displ[r] = start ind. for rank r
recv_type	Data type of the recv'd data
root	The root rank
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

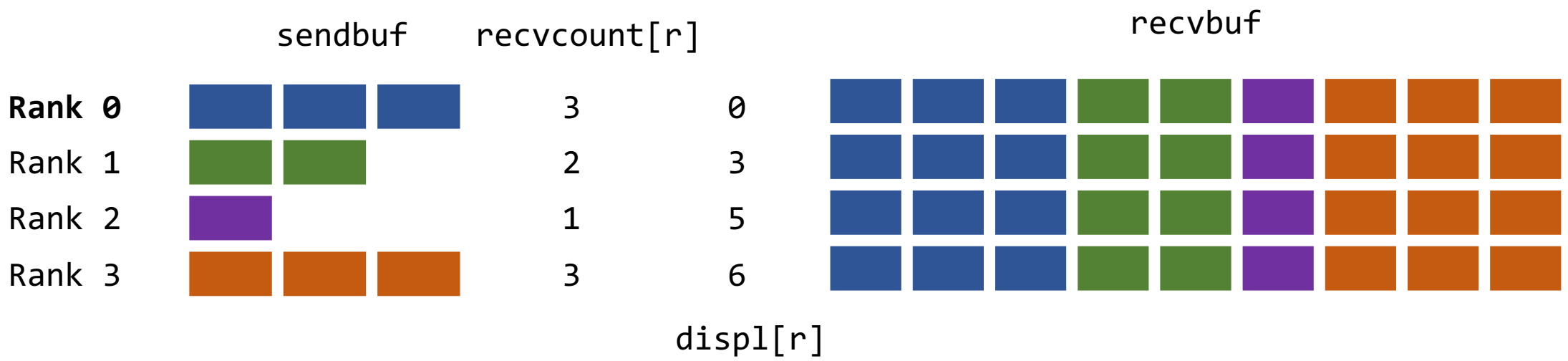
MPI_Allgatherv

```
int MPI_Allgatherv(
    const void* sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void* recvbuf,
    const int recvcounts[],
    const in displ[],
    MPI_Datatype recvtype,
    MPI_Comm comm
);
```

Argument	Description
sendbuf	Data to send
sendcount	How many elements each rank should send (unique to each rank)
sendtype	Data type of the send data
recvbuf	Where to store recv'ed data
recvcounts	Array telling how many elements to receive from each rank recvcounts[r] -> rank r
displ	The starting index into recvbuf for data from each rank displ[r] = start ind. for rank r
recv_type	Data type of the recv'd data
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

Custom Counts: Allgatherv Example

- MPI_Allgather: sendcount=?, recvcount={…},



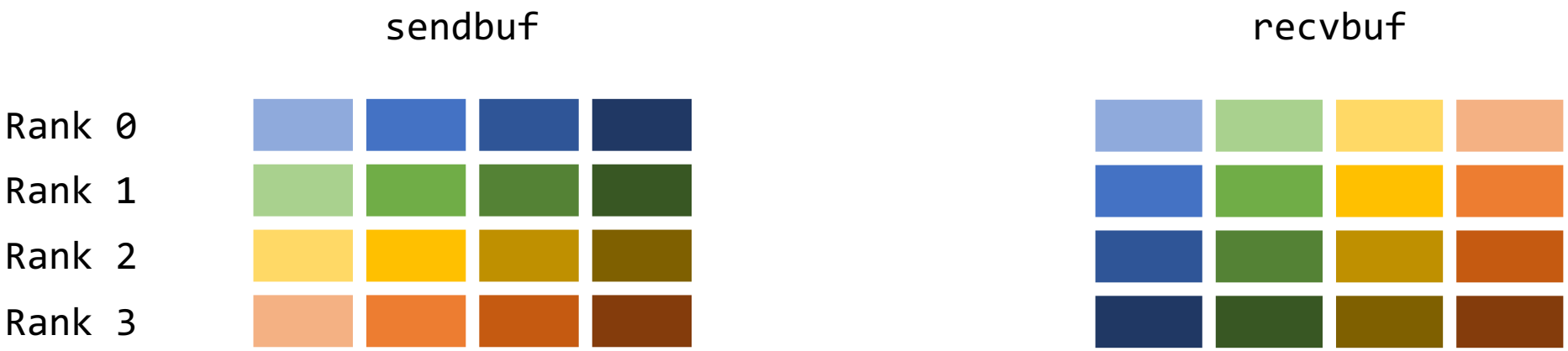
MPI_Alltoallv

```
int MPI_Alltoallv(
    const void* sendbuf,
    const int sendcounts[],
    const int senddispl[],
    MPI_Datatype sendtype,
    void* recvbuf,
    const int recvcounts[],
    const in recvdispl[],
    MPI_Datatype recvtype,
    MPI_Comm comm
);
```

Argument	Description
sendbuf	Data to send
sendcounts	How many elements to send to each rank
senddispl	Starting index into the send data by rank
sendtype	Data type of the send data
recvbuf	Where to store recv'ed data
recvcounts	Array telling how many elements to receive from each rank recvcounts[r] -> rank r
recvdispl	The starting index into recvbuf for data from each rank displ[r] = start ind. for rank r
recv_type	Data type of the recv'd data
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

Visual MPI Collective Cheat Sheet

- MPI_Alltoall: sendcount = recvcount = 1



Custom Counts: All-to-all(v)

- MPI_Alltoallv:

	sendcounts	senddispl	recvcounts	recvdispl
Rank 0	{1, 1, 2, 3}	{0, 1, 2, 4}	{1, 1, 3, 1}	{0, 1, 2, 5}
Rank 1	{1, 1, 1, 1}	{0, 1, 2, 3}	{1, 1, 1, 2}	{0, 1, 2, 3}
Rank 2	{3, 1, 2, 1}	{0, 3, 4, 6}	{2, 1, 2, 1}	{0, 2, 3, 5}
Rank 3	{1, 2, 1, 2}	{0, 1, 3, 4}	{3, 1, 1, 2}	{0, 3, 4, 5}

Notes on MPI_Alltoallv

- Because of how MPI_Alltoallv works, the data chunks to be sent do not need to be in order of rank!
  - You can use the senddispl array to point to the correct starting index
  - The same is true of recvbuf: data chunks do not need to be stored in order of rank
- You have to know how much to receive from each rank
  - You might have to do a regular all-to-all call first to build up the sendcounts and senddispl arrays

One Last Collective: Reduce-Scatter

- Mapping Guide:
  - Ranks: MPI_Reduce_scatter: all (ranks) -> all (ranks)
  - Data: MPI_Reduce_scatter: whole/single -> part
- A combination of a Reduce and a Scatter
- Each rank gets some of the reduction results

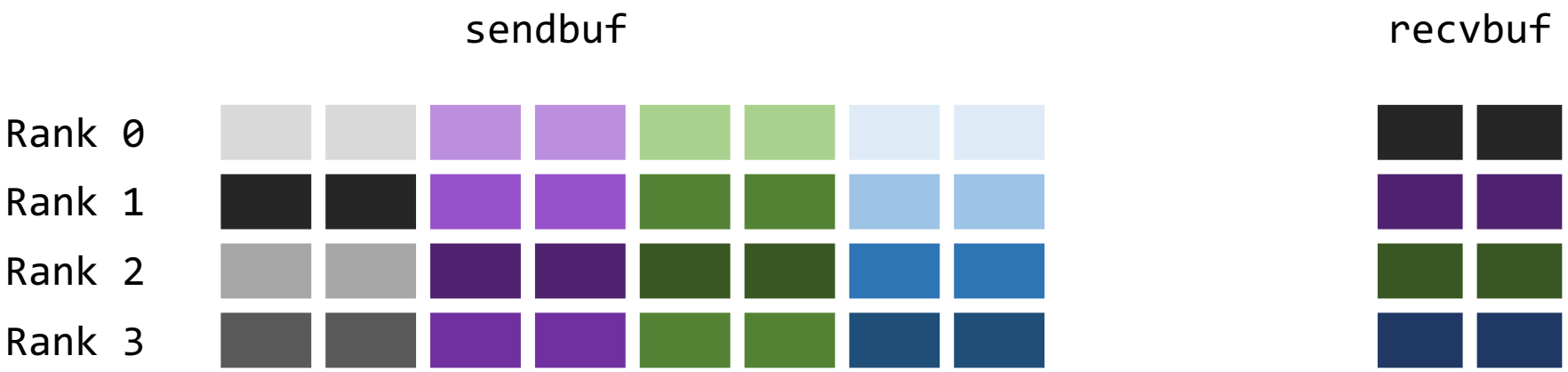
MPI_Reduce_scatter

```
int MPI_Allreduce(
    const void* sendbuf,
    void* recvbuf,
    const int recvcount[],
    MPI_Datatype datatype,
    MPI_Op operation,
    MPI_Comm comm
);
```

Argument	Description
sendbuf	The data to reduce
recvbuf	Where to store the results of the reduction
recvcount	The number of result entries that rank should receive
data_type	Data type of the data
operation	The operation to reduce wrt
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

Visual MPI Collective Cheat Sheet

- MPI_Reduce_scatter: recvcount=1, op=max



How do we estimate the cost of different communication patterns?

- Cost of communication: alpha-beta model

$$\text{Cost of communication} = \alpha + \beta n$$

- α = *latency* (time per msg). Message overhead costs.
- β = *inverse bandwidth* (time per word).
- Typically $\alpha \gg \beta$ and getting worse

Cost estimates

Name	# senders	# receivers	# messages	Computations?	Complexity
Broadcast	1	p	1	no	$\mathcal{O}(\alpha \log p + \beta n)$
Reduce	p	1	p	yes	$\mathcal{O}(\alpha \log p + \beta n)$
All-reduce	p	p	p	yes	$\mathcal{O}(\alpha \log p + \beta n)$
Prefix sum	p	p	p	yes	$\mathcal{O}(\alpha \log p + \beta n)$
Barrier	p	p	0	no	$\mathcal{O}(\alpha \log p)$
Gather	p	1	p	no	$\mathcal{O}(\alpha \log p + \beta pn)$
All-Gather	p	p	p	no	$\mathcal{O}(\alpha \log p + \beta pn)$
Scatter	1	p	p	no	$\mathcal{O}(\alpha \log p + \beta pn)$
All-To-All	p	p	p^2	no	$\mathcal{O}(\log p(\alpha + \beta pn))$ or $\mathcal{O}(p(\alpha + \beta n))$

Communication lower bounds

Communication	Latency	Bandwidth	Computation
Broadcast	$\lceil \log_2(p) \rceil \alpha$	$n\beta$	—
Reduce(-to-one)	$\lceil \log_2(p) \rceil \alpha$	$n\beta$	$\frac{p-1}{p} n\gamma$
Scatter	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	—
Gather	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	—
Allgather	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	—
Reduce-scatter	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	$\frac{p-1}{p} n\gamma$
Allreduce	$\lceil \log_2(p) \rceil \alpha$	$2\frac{p-1}{p} n\beta$	$\frac{p-1}{p} n\gamma$

Note: Pay particular attention to the conditions for the lower bounds given in the text.

Chan, Heimlich, Purkayastha, and Van De Geijn. "Collective communication: theory, practice, and experience." (2007)

Collective communication bounds

- Can estimate the costs associated with communication for different algorithms.
- Consider Allgather

Latency

$$\lceil \log_2(p) \rceil \alpha$$

Bandwidth

$$\frac{p-1}{p} n\beta$$

Collective communication bounds

- Can estimate the costs associated with communication for different algorithms.
- Consider Allgather

Latency

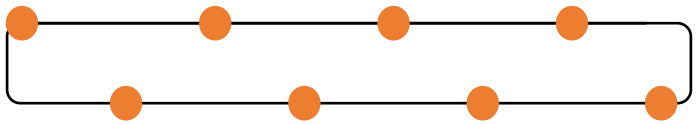
$$\lceil \log_2(p) \rceil \alpha$$

Bandwidth

$$\frac{p-1}{p} n\beta$$

Example: Allgather

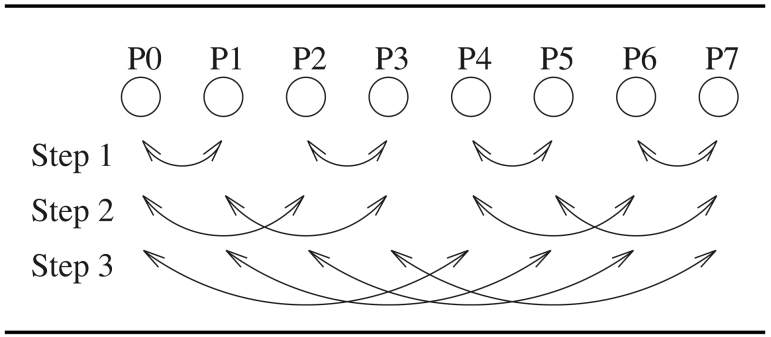
Ring Algorithm



$T_{ring} = \alpha (p-1) + \beta n (p-1)/p$

- **At time 0**, send your original data
- **At time t**: send the data you received at time t-1 to your right, and receive new data from your left.
- Optimal bandwidth, suboptimal latency
- Not that bad if pipelined properly (Nvidia's NCCL library uses only pipelined ring algorithms)

Recursive Doubling Algorithm



$T_{rec\text{-}dbl} = \alpha \log(p) + \beta n (p-1)/p$

- **At time t**: process i exchanges (send/recv) all its current data (its original data plus anything received until then) with process $i \pm 2^t$
- Optimal bandwidth and latency
- Data exchanged at each step: n/p, 2n/p, 4n/p,…, $2^{\lg(p)-1}$ n/p
- Tricky for non-power-of-two

Figures from Aydin Buluc