CMOR 421/521
Distributed parallelism

Jesse Chan

What is distributed parallelism?

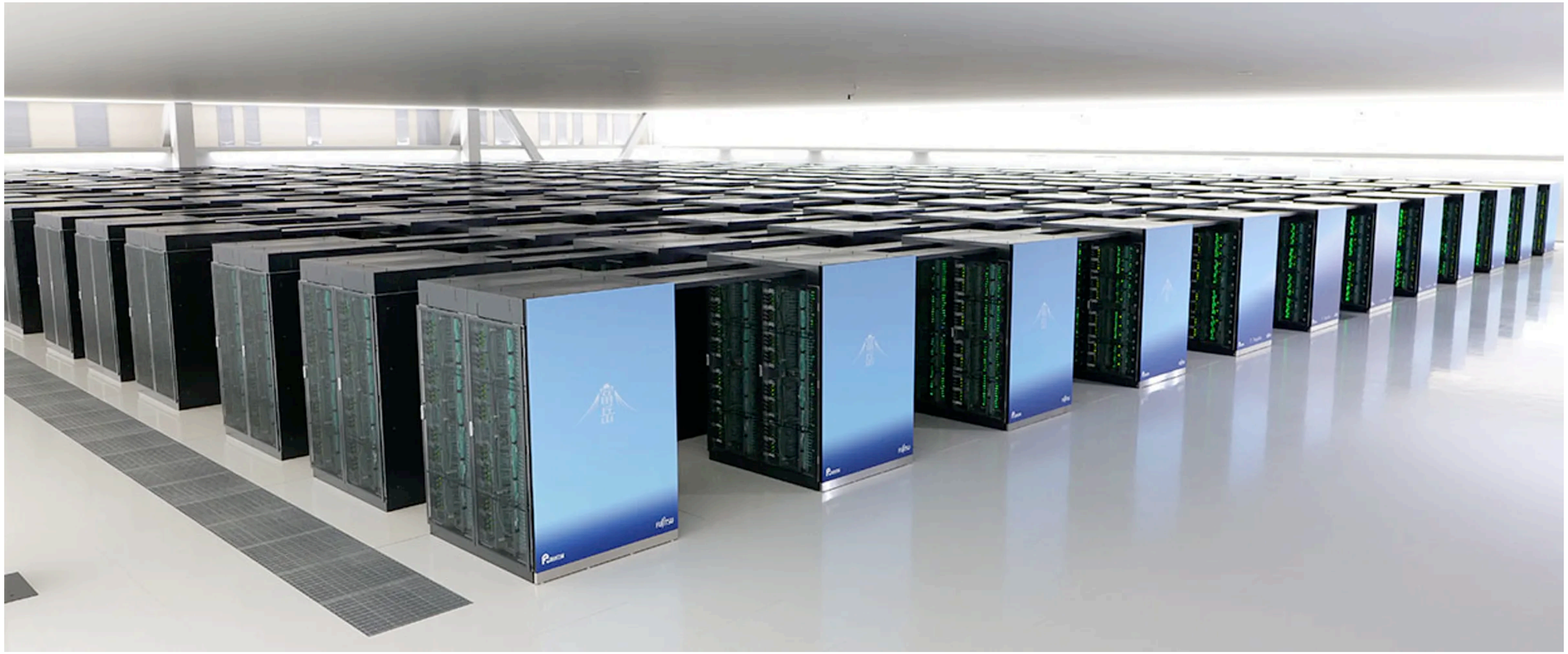
What is distributed parallelism?

And why do we need it?

- Physical limitations of hardware motivate different types of parallelism; shared memory parallelism overcomes limitations of single-core CPUs.
- Distributed memory parallelism addresses limitations of multi-core CPUs.
  - More cores increases cost of synchronization (cache coherency protocols).
  - Modern CPUs are still adding more cores, but are moving to alternative architectures (GPUs) to deal with scaling issues.
  - Memory: some problems are so large they need to be distributed across multiple processors just to be represented.

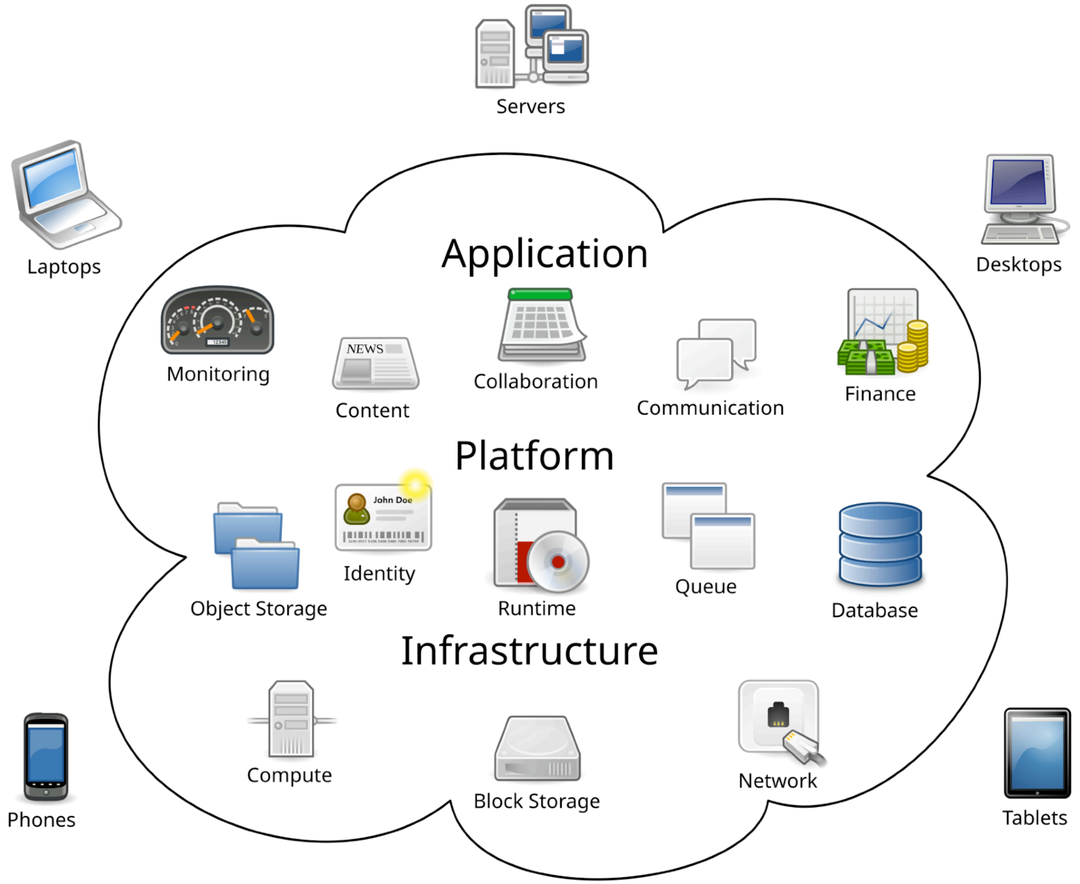
Some examples of distributed parallel machines

Japan’s Fugaku supercomputer



Some examples of distributed parallel machines

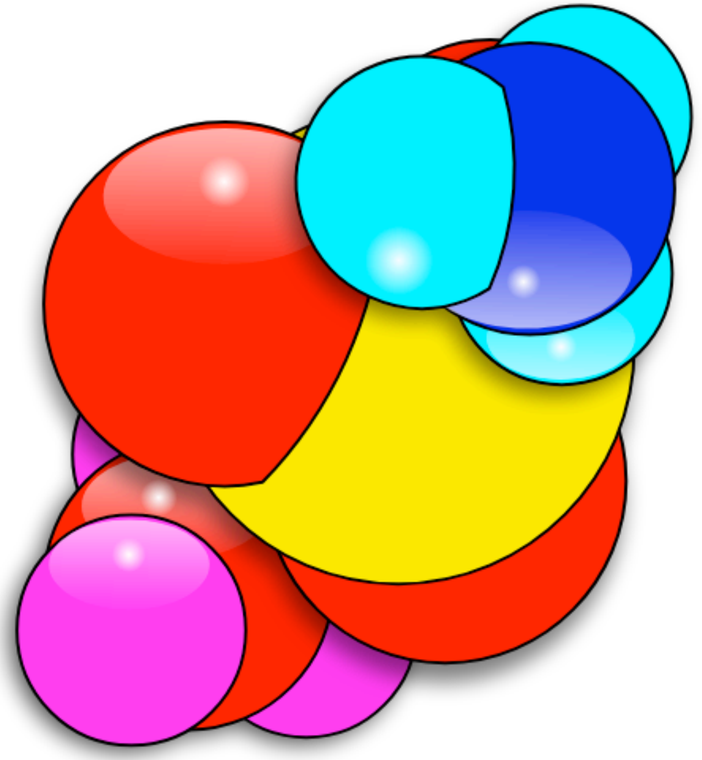
- Cloud computing is one type of distributed computing.
- Offload your main computations to the cloud (some distributed machine), communicate the result back through the internet.



Some examples of distributed parallel machines

Folding @ home

- Started in 2000, distributes work in molecular dynamics simulations to participating home computers as background processes.
- Technically became the first exascale “machine” during COVID.
- Highly heterogeneous computing: old/new CPUs, laptop/gaming GPUs.



Some examples of distributed parallel machines

B

Some examples of distributed parallel machines

Beowulf (LAN-networked) clusters

- Local area network (LAN) connections could be used to build a “Beowulf” cluster for cheap parallel computing
- Linux, MPI-based.
- Single server node + client nodes
- “In most cases, client nodes in a Beowulf system are dumb, the dumber the better.”

Communication-based parallelism

- Distributed parallelism is more flexible since it doesn't depend on hardware.
  - Allows for highly heterogeneous architectures (CPUs vs GPUs)
  - The internet is a communication-based distributed network for file-sharing.
- How is distributed parallelism different? Independent workers use explicit **communication** to share information, synchronize, etc.
  - Can also be used for shared memory parallelism.
- How do we send messages between independent workers? Typically using the **Message Passing Interface (MPI)**.

Intro to the Message Passing Interface (MPI)

Message-passing interface (MPI)

- MPI is a standard (similar to the BLAS *specification* or the OpenMP *standard*)
  - BLAS is implemented by multiple libraries, OpenMP is implemented by different compilers.
- MPI also has two popular implementations: OpenMPI and MPICH
  - MPICH is the original implementation by Argonne National Lab, and implements every routine in the MPI specification.
  - OpenMPI is newer and doesn't implement every MPI specification feature, but can be easier to install and more performant.

Installing MPI

- On Mac, “brew update” then “brew install openmpi” should work
- OpenMPI and MPICH are not supported on Windows. Alternatives:
  - Use VSCode and SSH to directly access NOTSx
  - Install OpenMPI or MPICH through Cygwin or WSL.
  - Install MS-MPI. Note: I don’t have experience with this.

Hello world with MPI

- “MPI_Init” creates the MPI environment. It optionally takes in “argc” and “argv” for command-line input arguments.
- “MPI_Comm_rank” and “MPI_Comm_size” query MPI for the “rank” (current process) and “size” (total number of processes)
  - The MPI communicator (the list of ranks to communicate among) is “MPI_COMM_WORLD” (all ranks)

```cpp
#include "mpi.h"
#include <iostream>

using namespace std;

int main(){

  MPI_Init(NULL, NULL);

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  cout << "Hello world on rank " << rank
    << " of " << size << endl;

  MPI_Finalize();
  return 0;
}
```

hello_world_mpi.cpp

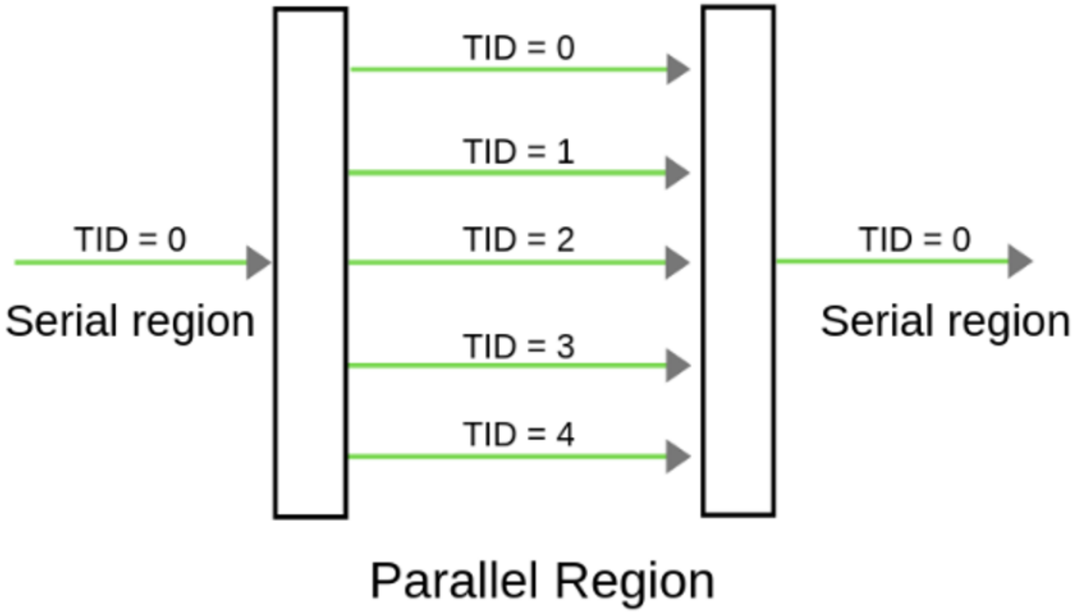
Running MPI “hello world”

- To compile, use “mpic++” to create an executable.
- To run your executable, you need to launch the program using “mpiexec” or “mpirun”
  - Just running the executable will execute with one single rank
  - E.g., “mpirun -n 4 ./a.out” will run “a.out” with 4 ranks.

```
(base) jchan985@XXH62CK9GR MPI % mpic++ hello_world_mpi.cpp
ld: warning: dylib (/opt/homebrew/Cellar/open-mpi/5.0.7/lib/
libmpi.dylib) was built for newer macOS version (15.0) than
being linked (14.0)
(base) jchan985@XXH62CK9GR MPI % ./a.out
Hello world on rank 0 of 1
(base) jchan985@XXH62CK9GR MPI % mpirun -n 2 ./a.out
Hello world on rank 1 of 2
Hello world on rank 0 of 2
(base) jchan985@XXH62CK9GR MPI %
```

Differences between OpenMP and MPI

- OpenMP's threading model is "top down" where you can see all serial/ parallel sections.
- MPI's model is "bottom up", where each rank corresponds to a single thread ID.
  - The code executes from the perspective of a single parallel process, e.g., all workers will execute the same MPI code.



Sending and receiving data via MPI

- No shared memory in MPI.
- MPI routines are typically fairly low-level and C-like (lots of pointer usage and thinking about memory layout).
- Here, we use “MPI_Send” to send information from rank 0 to rank 3.
- Rank 3 uses “MPI_Recv” to receive that information.

```
MPI_Status status;
int x;
if (rank == 0)
{
  x = 123;
  MPI_Send(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);
}
else if (rank == 3)
{
  MPI_Recv(&x, 1, MPI_INT, 0, MPI_ANY_TAG,
           MPI_COMM_WORLD, &status);
}
else
{
  x = -1;
}
```

What are the arguments of MPI_Send?

```
int MPI_Send(
    const void* buffer,
    int count,
    MPI_Datatype data_type,
    int dest_rank,
    int tag,
    MPI_Comm comm
);
```

Argument	Description
buffer	Pointer to the data to be sent
count	How many entries to send
data_type	Data type of the data to be sent
dest_rank	The rank to send to
tag	Identifier for the message
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

```
x = 123;
MPI_Send(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);
```

What are the arguments of MPI_Recv?

```
int MPI_Recv(
    void* buffer,
★   int count,
    MPI_Datatype data_type,
    int src_rank,
    int tag,
    MPI_Comm comm,
★   MPI_Status* status
);
```

Argument	Description
buffer	Pointer to where to copy the data
count	**AT MOST** how many entries to read
data_type	Data type of the data to be sent
src_rank	The rank to receive from
tag	Identifier for the message
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)
status	Information about the message

```
MPI_Recv(&x, 1, MPI_INT, 0, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status);
```

MPI has several pre-defined data types

Primitive Type	MPI Data Type
int	MPI_INT
long long int	MPI_LONG_LONG
float	MPI_FLOAT
double	MPI_DOUBLE
char	MPI_BYTE

What does an instance of MPI_Status store?

Fields	Description
int count	Number of *received* entries
int cancelled	Was the request cancelled?
int MPI_SOURCE	Source rank
int MPI_TAG	Tag value
int MPI_ERROR	Any errors associated with the message

If we don’t care about the status:

```
MPI_STATUS_IGNORE
```

Sending and receiving an array via MPI

- What will the following code return for n=10?

```
for (int i = 0; i < n; ++i)
{
  x[i] = (double)rank + i;
}
```

```
if (rank == 0)
{
  MPI_Send(x, n / 2, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
}
else if (rank == 1)
{
  MPI_Recv(x + n / 2, n / 2, MPI_DOUBLE, 0, MPI_ANY_TAG,
           MPI_COMM_WORLD, &status);
  cout << "on rank " << rank << endl;
  print_vector(n, x);
}
```

Sending and receiving an array via MPI

- What will the following code return for n=10?

```
for (int i = 0; i < n; ++i)
{
  x[i] = (double)rank + i;
}
```

  - Initialize a vector's values to "0, 1, 2, 3, …" plus the current rank.
  - Sends the second half of a vector from rank 0 to rank 1.
  - Should return:
    x = [1, 2, 3, 4, 5, 0, 1, 2, 3, 4]

```
if (rank == 0)
{
  MPI_Send(x, n / 2, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
}
else if (rank == 1)
{
  MPI_Recv(x + n / 2, n / 2, MPI_DOUBLE, 0, MPI_ANY_TAG,
           MPI_COMM_WORLD, &status);
  cout << "on rank " << rank << endl;
  print_vector(n, x);
}
```

Synchronization and non-blocking communication

Implicit synchronization and deadlock

- MPI provides *explicit* synchronization via “MPI_Barrier(MPI_Comm comm)”
- However, “MPI_Send” and “MPI_Recv” provide *implicit* synchronization.
  - The sending and receiving processes will not continue onward until the communication is completed.
  - Implicitly synchronized communication are referred to as *blocking.*
- *Blocking* communication can result in problematic situations related to synchronization, however.

What happens in this code?

```
int n = 10000;
int *x = new int[n];
if (rank == 0)
{
    MPI_Send(x, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(x, n, MPI_INT, 1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if (rank == 1)
{
    MPI_Send(x, n, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(x, n, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Implicit synchronization and deadlock

- The previous code was an example of “deadlock” where the program stalls indefinitely
  - Rank 0 sends, waits for Rank 1 to receive the message.
  - Rank 1 sends, waits for Rank 0 to receive the message.
- Can fix by reversing order of communication, or by not using blocking communication.

```
if (rank == 0)
{
    MPI_Send(x, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(x, n, MPI_INT, 1, MPI_ANY_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if (rank == 1)
{
    MPI_Send(x, n, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(x, n, MPI_INT, 0, MPI_ANY_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Implicit synchronization and deadlock

- The previous code was an example of “deadlock” where the program stalls indefinitely
  - Rank 0 sends, waits for Rank 1 to receive the message.
  - Rank 1 sends, waits for Rank 0 to receive the message.
- Can fix by reversing order of communication, or by not using blocking communication.

```
if (rank == 0)
{
    MPI_Send(x, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(x, n, MPI_INT, 1, MPI_ANY_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if (rank == 1)
{
    MPI_Recv(x, n, MPI_INT, 0, MPI_ANY_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(x, n, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

Another example of deadlock

- Another simple example of deadlock can be shown with “MPI_Barrier”, which is similar to an OpenMP barrier.
  - No ranks can pass until they’ve all arrived at the barrier.
- What happens if we run either of these code chunks on >1 ranks?

```
if (rank == 0)
{
    MPI_Barrier(MPI_COMM_WORLD);
}
```

```
if (rank != size-1)
{
    MPI_Barrier(MPI_COMM_WORLD);
}
```

Non-blocking communication and latency hiding

- *Non-blocking* communication does not wait for the communication to complete before moving past the send/ receive function call.
- “MPI_Isend” and “MPI_IRecv”; the “I” stands for “immediate return”
- Can also be used when overlapping communication and computation for *latency hiding.*

```
MPI_Request request;
if (rank==0){
  MPI_Isend(x, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
}else if (rank==1){
  MPI_Irecv(x, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
}
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

What are the arguments of MPI_ISend?

```
int MPI_Isend(
    const void* buffer,
    int count,
    MPI_Datatype data_type,
    int dest_rank,
    int tag,
    MPI_Comm comm,
    MPI_Request* request
);
```

Argument	Description
buffer	Pointer to the data to be sent
count	How many entries to send
data_type	Data type of the data to be sent
dest_rank	The rank to send to
tag	Identifier for the message
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)
request	**MPI data structure for monitoring the send's status**

```
MPI_Isend(x, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
MPI_Irecv(x, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
```

What are the arguments of MPI_IRecv?

```
int MPI_Irecv(
    void* buffer,
    int count,
    MPI_Datatype data_type,
    int src_rank,
    int tag,
    MPI_Comm comm,
    MPI_Status* status,
    MPI_Request* request
);
```

Argument	Description
buffer	Pointer to where to copy the data
count	**AT MOST** how many entries to read
data_type	Data type of the data to be sent
src_rank	The rank to receive from
tag	Identifier for the message
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)
status	Information about the message
request	**MPI data structure for monitoring the send's status**

```
MPI_Isend(x, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
MPI_Irecv(x, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
```

“MPI_Wait”, “MPI_Test”

- MPI_Wait is similar to “#pragma omp taskwait”
  - Stops a single rank until the MPI_Request is fulfilled
- Alternative: MPI_Test. Performs a non-blocking check of whether a request has finished.
  - Useful for event-driven programming

```
MPI_Isend(send_data, n, MPI_INT, 1, 0,
          MPI_COMM_WORLD, &request);

// do work ...

MPI_Wait(&request, MPI_STATUS_IGNORE);
```

```
if (rank == 0)
    MPI_Isend(x, n, MPI_INT, 1, 0,
              MPI_COMM_WORLD, &request);

if (rank == 1)
    MPI_Irecv(x, n, MPI_INT, 0, 0,
              MPI_COMM_WORLD, &request);

int is_not_complete = 0;
while (is_not_complete == 0)
{
    MPI_Test(&request, &is_not_complete,
             MPI_STATUS_IGNORE);
    // ...
}
```

Collective communication

“Point-to-point” vs “collective” communication

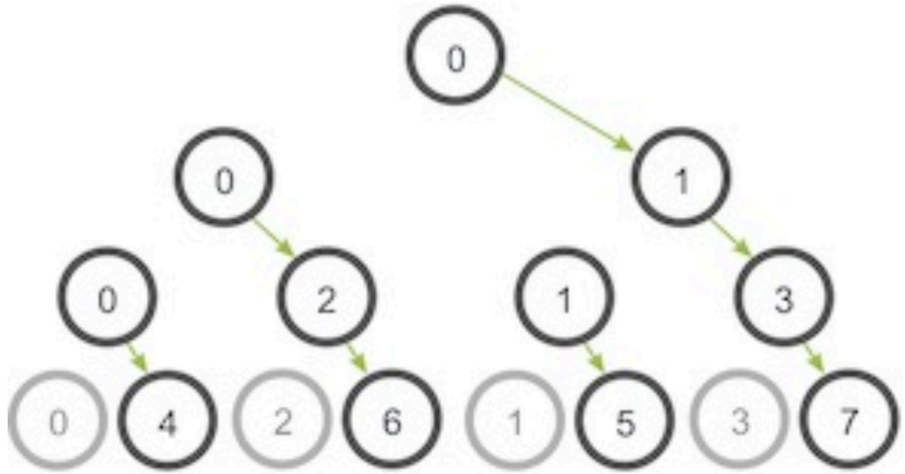
- MPI_Send, MPI_Isend, MPI_Recv, and MPI_Irecv are all “point-to-point” communication, where individual ranks communicate to each other.
- In practice, most parallel programs have communication which involves more than 2 processes (sometimes all).
- “Collective communication” involves a “collection” of processes.
  - MPI provides support for many collective communication patterns, just as OpenMP provided support for common parallelism patterns (e.g., parallel for loops, reductions).
  - Collective communications often easier to implement/debug (e.g., avoid deadlock by construction)

Motivating “collective” communication

- Consider broadcasting information from rank 0 to all other ranks.
- Naive approach: rank 0 sends info to all other ranks, resulting in O(n_ranks) communication cycles.
  - Loop over all ranks and use MPI_Send one-by-one.
- Doesn’t take full advantage of network bandwidth - all communications are being done through one node, while all other nodes are idle.

Motivating “collective” communication

- A better approach: tree-like broadcasting.
  - First, R0 -> R1. Now both R0 and R1 have the information.
  - R0 -> R2
    R1 -> R3. Now, R0, …, R3 have the info.
  - R0 -> R4, R1 -> R5
    R2 -> R6, R3 -> R7…
- O(log(n_ranks)) communications for each node.



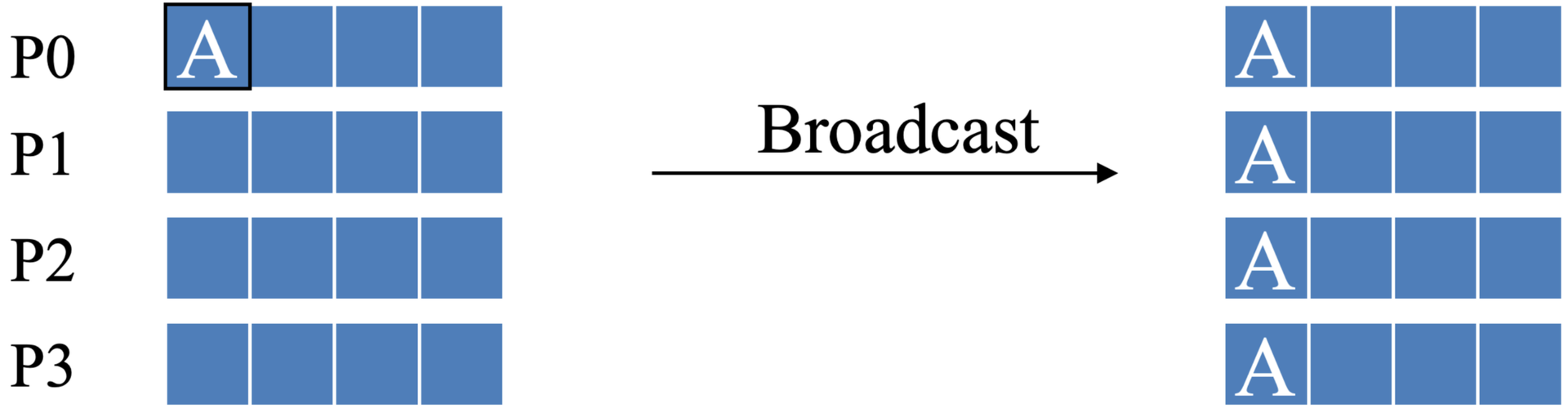
Collective communication in MPI

- Implementing tree-based broadcasting yourself has better performance, but is more complicated (and can introduce deadlock issues).
- This is a common communication pattern that MPI provides support for via collective communications.
  - First: introduce on broadcast/reduction type operations.
  - Next: introduce all-to-all communication patterns.
- Most collective operations in MPI use lower level communication (MPI_Send/MPI_Recv and/or MPI_Isend/MPI_Irecv).

Collective communication in MPI

- Some common one-to-all or all-to-one patterns:
  - Broadcast
  - Reduction
  - Gather
  - Scatter

MPI_Bcast visualization and example usage




```
int x;
if (rank==0){
  x = 1;
}
MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
cout << "On rank " << rank << "/" << size-1 <<
  ", x = " << x << endl;
```

Graphic from Aydin Buluc

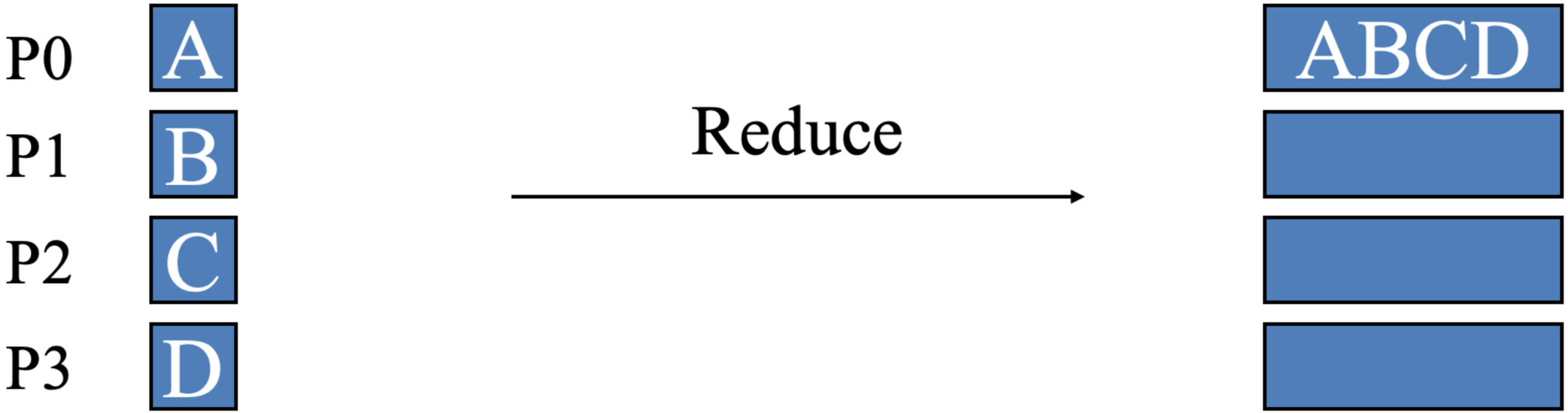
MPI_Bcast function signature

```
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)

MPI_Reduce visualization and example usage

Similar to OpenMP reduction, but only one rank gets the result.




```
int sum;
MPI_Reduce(&x, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0){
  cout << "On root rank, sum = " << sum << endl;
}
```

Graphic from Aydin Buluc

MPI_Reduce function signature

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

Here, MPI_Op is a supported reduction operation.

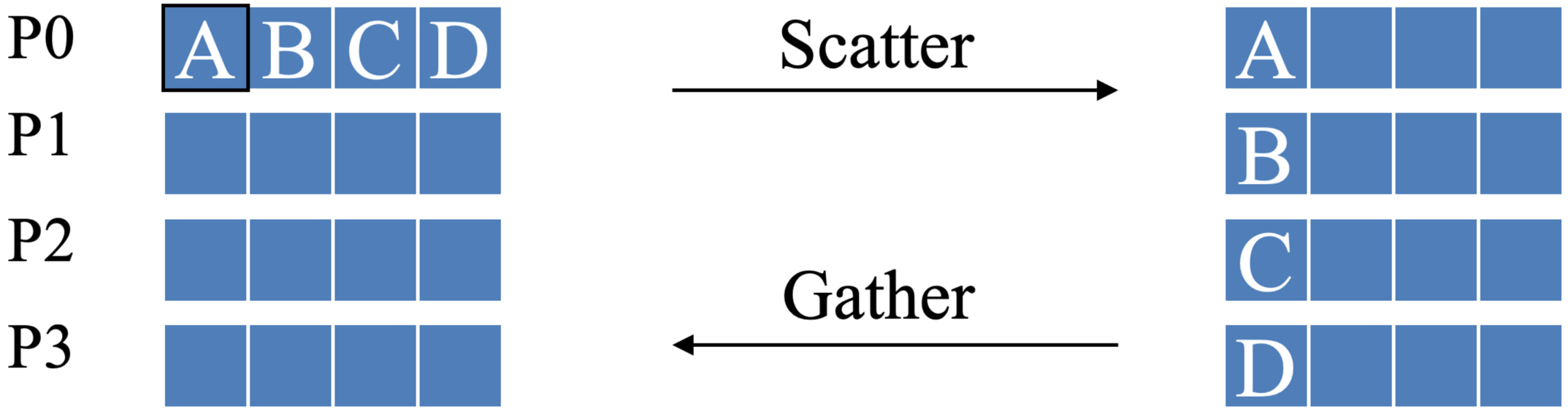
Argument	Description
sendbuf	All: the data to reduce
recvbuf	Root: where to store the results of the reduction
	Others: not needed (pass NULL)
count	the number of reduction sets/the rank-wise length of the data
data_type	Data type of the data to be sent
operation	The operation to reduce wrt
root_rank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

Reduction operations

Operation	MPI_OP
minimum	MPI_MIN
maximum	MPI_MAX
sum	MPI_SUM
product	MPI_PROD

- Others operations (logical operations, arg min/max + rank) are also supported.
- You can also create your own customized reduction operator via MPI_OP_CREATE (combines a function handle with a pointer to the appropriate MPI collective operation).

Scatter/Gather visualization and usage



- Gather/scatter are inverses of each other.

Graphic from Aydin Buluc

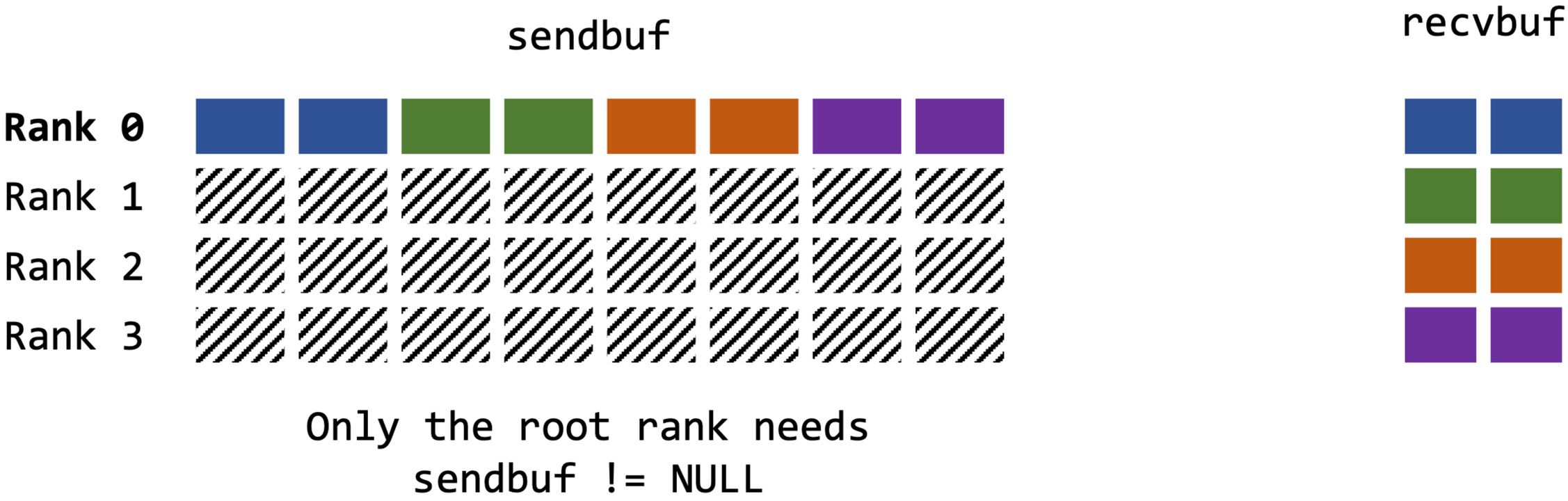
MPI_Scatter example usage

- All data is stored on rank 0 in the “xsend” buffer.
- This data is scattered to all other ranks in chunks of 2 entries each.
- The entries are received into the “xrecv” buffer.

```
int *xsend = (rank == 0) ? new int[2 * size] : NULL;
int *xrecv = new int[2];
if (rank == 0)
{
  for (int i = 0; i < 2 * size; ++i)
  {
    xsend[i] = i;
  }
}
MPI_Scatter(xsend, 2, MPI_INT,
            xrecv, 2, MPI_INT, 0,
            MPI_COMM_WORLD);
```

MPI_Scatter example usage

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



Non-blocking collective communication

- MPI Bcast, Reduce, Gather, and Scatter are *blocking* communications
  - How they block is tied to the send/recv buffers like MPI Send/Recv: when the buffers are safe to use the process can move on
- All collectives have asynchronous counterparts: MPI_Ibcast, MPI_Ireduce, MPI_Igather, MPI_Iscatter
  - As before, a “MPI_Request” argument is added to the function arguments, and “MPI_Wait” is used to define a barrier/synchronization point.

Variable-count collective communication

- What if each rank wants to send a different amount of data?
  - There is a “vector” or “variable” version of each MPI command
  - MPI_Gatherv, MPI_Scatterv, etc…
- These functions specify the count/sendcount/recvcount as pointers to arrays with different entries per rank.

Deadlocks with collective communication

- Deadlocks can still happen with collective communication! To avoid this, **all processes should execute a collective communication operation**.
  - For collectives that map to/from a single rank, you specify that rank in the function call (this is the “root rank”).
- The deadlock issue is similar to the MPI_Barrier deadlock example; if only one rank executes MPI_Barrier, the program deadlocks waiting for all the other ranks to reach MPI_Barrier. Collective communication calls behave similarly.

Exercise: integration and reduce

- Install MPI on your laptop, or load it on NOTSx using “module load …”
- Convert your OpenMP integration code to use MPI. Each rank should compute 1 / num_ranks of the summation.

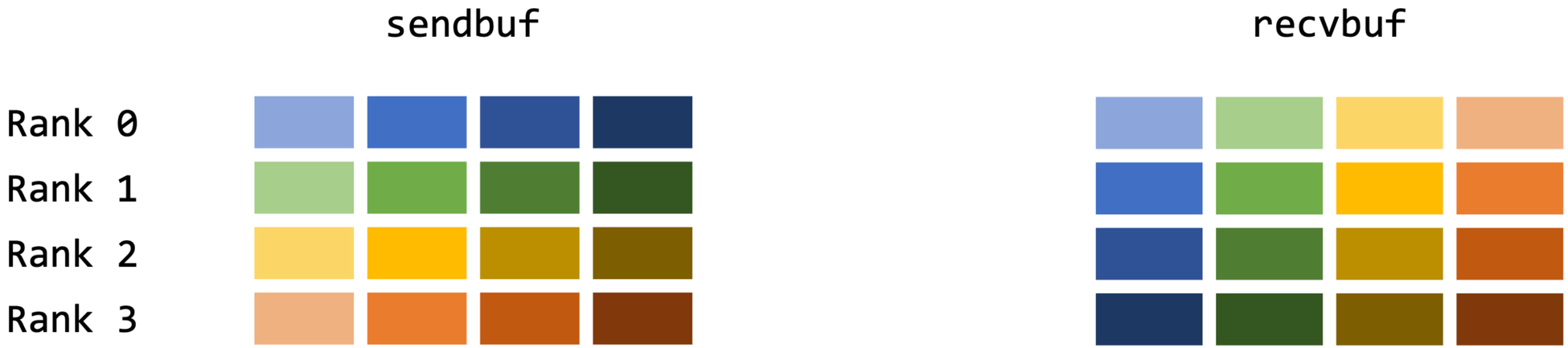
MPI “all” collective communications

- Motivation: consider the two examples:
  - How would you share the result of a reduction to all ranks?
  - How would you broadcast info from all ranks to all other ranks?
- Often you want to communicate information from all ranks to all other ranks; these can be done using MPI “all” collective operations.

MPI “all” collective communications

- For each all-to-one or one-to-all collective operation, there is an equivalent “all” collective operation.
  - Broadcasting (1 —> all) => **All-to-all**
  - Reductions (all —> 1) => **All-reduce**
  - Gather (all —> 1) => **All-gather**
  - Scatter (1 —> all) => **All-to-all**
- Note that for MPI “all” collectives, *no root rank* needs to be specified.

MPI “all-to-all” collective communication



- Similar conceptually to matrix transposition (if ranks correspond to rows and columns correspond to chunks of data on each rank)
- Can exchange single pieces of information (sendcount = recvcount = 1) or chunks of memory (e.g., 2 or more pieces of data)

MPI “all-to-all” function signature

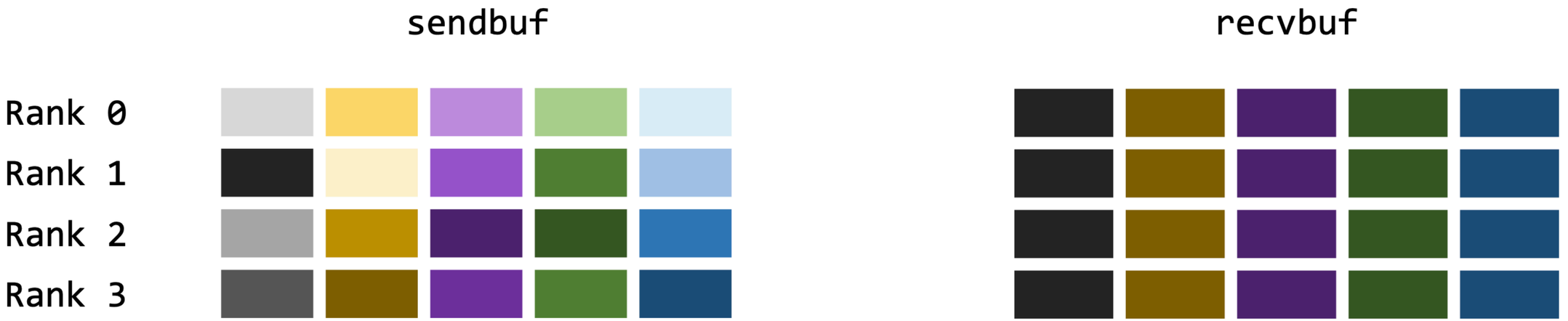
```
int MPI_Alltoall(
    const void* sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void* recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    MPI_Comm comm
);
```

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)

MPI “all-to-all” example

- Demo

MPI “all-reduce” collective



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

- Calculates a reduction among each “column” of ranks, and broadcasts the result to every other rank.
- Each rank receives the reduction result from every other rank.

MPI “all-reduce” function signature

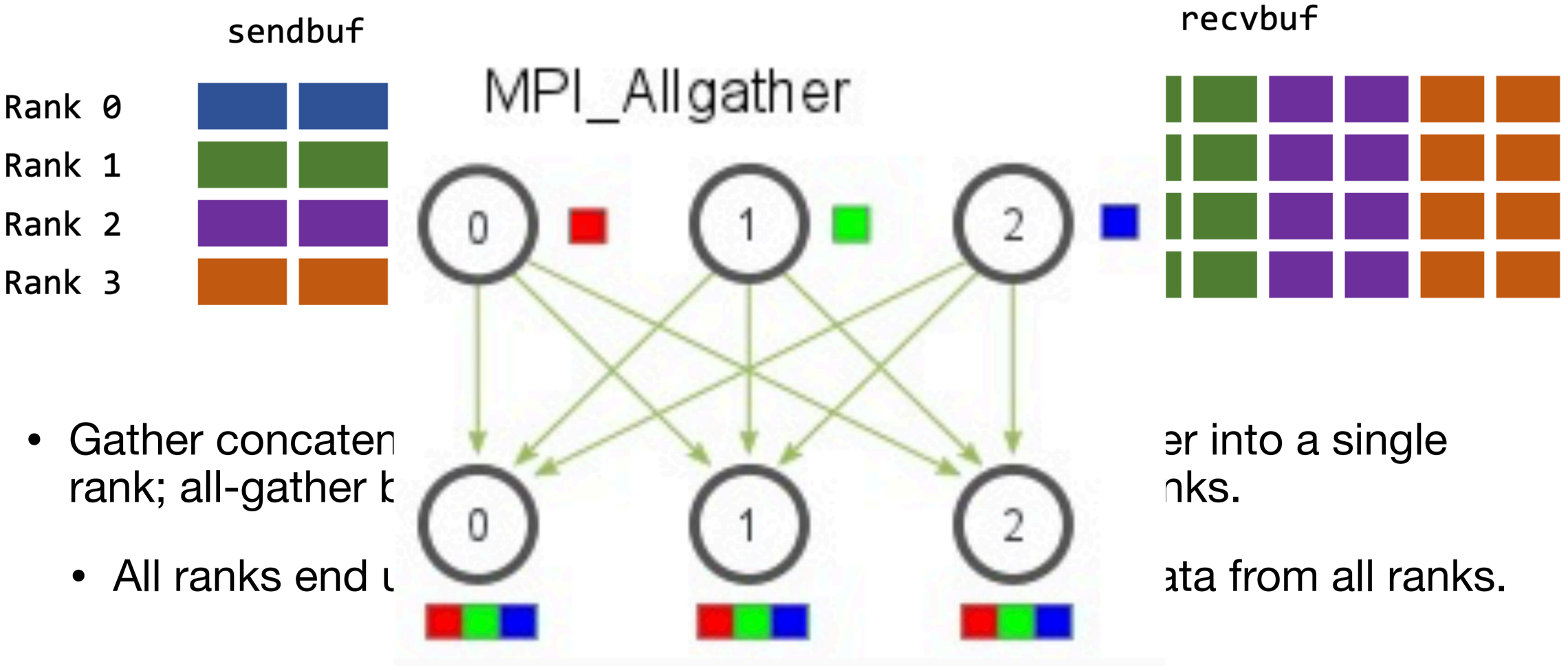
```
int MPI_Allreduce(
    const void* sendbuf,
    void* recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op operation,
    MPI_Comm comm
);
```

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)

MPI “all-gather” collective

- Gather concatenates/collects data on each rank together into a single rank; all-gather broadcasts that collected result to *all* ranks.
  - All ranks end up with the same result: the gathered data from all ranks.

MPI “all-gather” collective



- Gather concaten er into a single rank; all-gather b nks.
  - All ranks end u ata from all ranks.

MPI “all-gather” collective

```
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)

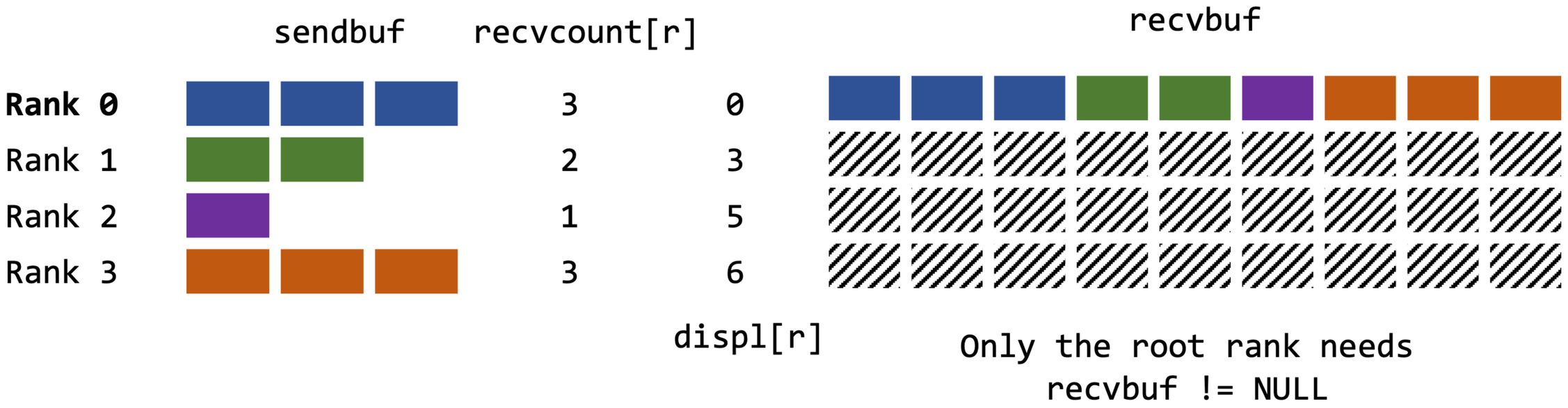
MPI “all” collectives and blocking

- MPI_Alltoall, MPI_Allreduce, and MPI_Allgather are all *blocking*
- They have asynchronous counterparts with an “I”:
  - MPI_Ialltoall
  - MPI_Iallreduce
  - MPI_Iallgather
- As before, a MPI_Request argument is added to the function arguments

Heterogeneous MPI “all” collectives

- What if some ranks have more (or less) data to send than others? MPI_Gatherv, MPI_Scatterv
  - Custom counts by rank: MPI_Alltoallv, MPI_Allgatherv
- What if some ranks have ints to send and another doubles (i.e., different data types for each rank)? Can send custom types by rank:
  - MPI_Alltoallw (ot going to look at; just know that it exists)
- Some other useful MPI collectives we won’t discuss (reduce-scatter)

Heterogeneous MPI_Gatherv illustration



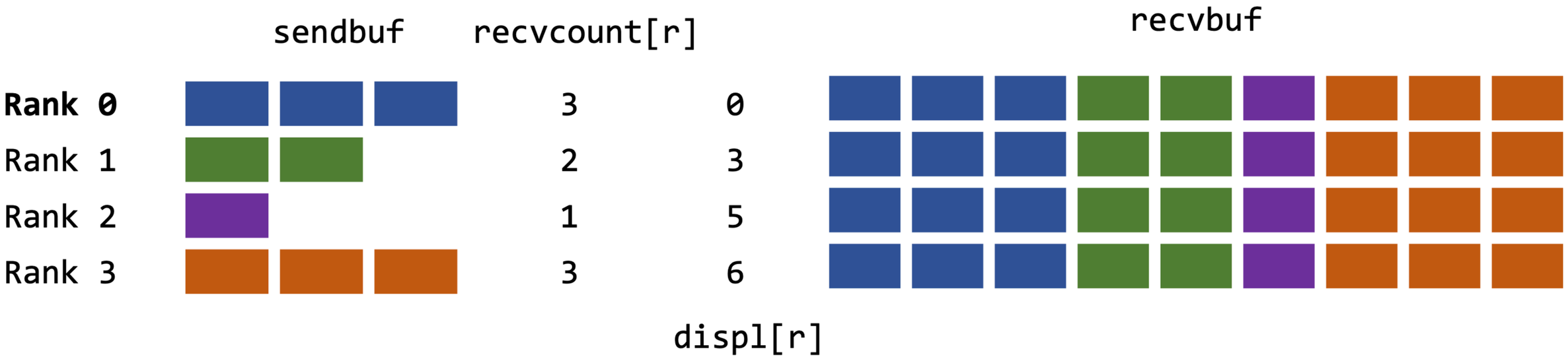
- Example of MPI_Gatherv: gather a different amount of data from each rank.
- Need to specify both recvcount (amount of data received from each rank) and “displ” (“displ[rank]” is the index that data from “rank” starts at)

Heterogeneous MPI_Gatherv signature

```
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)

Heterogeneous MPI_Allgatherv illustration



- Example of MPI_Gatherv: gather a different amount of data from each rank.
- Need to specify both recvcount (amount of data received from each rank) and “displ” (“displ[rank]” is the index that data from “rank” starts at)

Heterogeneous MPI_Allgatherv signature

```
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)

Heterogeneous MPI_Alltoallv illustration

	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}

- More general than Alltoall; rank ordering doesn't matter since senddispl/ recvdispl can be used to specify orderings.

Heterogeneous MPI_Alltoallv signature

```
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)

The cost of MPI communications

- We won't go into detail, but the cost of MPI communication is very similar to the cost of memory accesses for single-core CPU optimizations.
- Cost is $\alpha + \beta n$, where
  - α is the latency (time per message). A fixed overhead cost for each message being sent.
  - β is the bandwidth (time per "word" of memory). Typically much smaller than α (and getting worse)

Cost estimates for MPI communication

Can also derive lower bounds for each type of communication

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 $\mathcal{O}(p(\alpha + \beta n))$

MPI and linear algebra

- What MPI routines do parallel implementations of different linear algebra operations use? What is the data layout?
  - How do we compare the cost of different implementations?
- We'll compare some examples of linear algebra operations:
  - Matrix-vector product (row/column major storage)
  - Block matrix storage will motivate custom communicators (e.g., not MPI_COMM_WORLD)
  - Some parallel matrix-matrix multiply algorithms (SUMMA and Cannon's algorithm) using custom communicators

What MPI communication fits best?

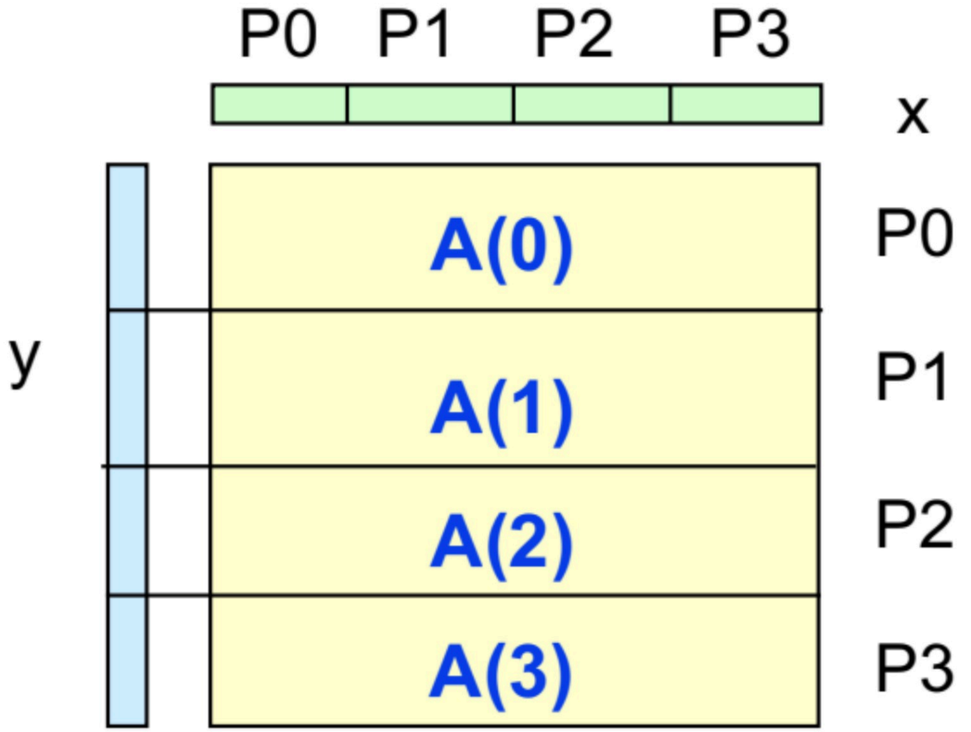
- I'll describe a specific data decomposition and parallelization, and outline the main computational steps necessary to compute the result in a distributed parallel fashion.
  - You'll guess what MPI collectives are best suited to each case.
- We'll cover some new MPI commands and concepts, motivated by different linear algebra applications.
  - MPI_Reduce_scatter and custom MPI communicators

Example: low rank mat-vec $y = ab^T x$

- Assume each processor owns a block of “b = n / size” entries of x, y, a, b.
- What MPI commands would work well for this example?
  - Compute in two steps: b_dot_x = $b^T x$, then $y = a$ * b_dot_x.
  - Use MPI_Allreduce (MPI_Reduce + MPI_Bcast) to compute b_dot_x and share the result with every rank
  - Calculating $y = a$ * b_dot_x can then be done locally.
  - Demo.

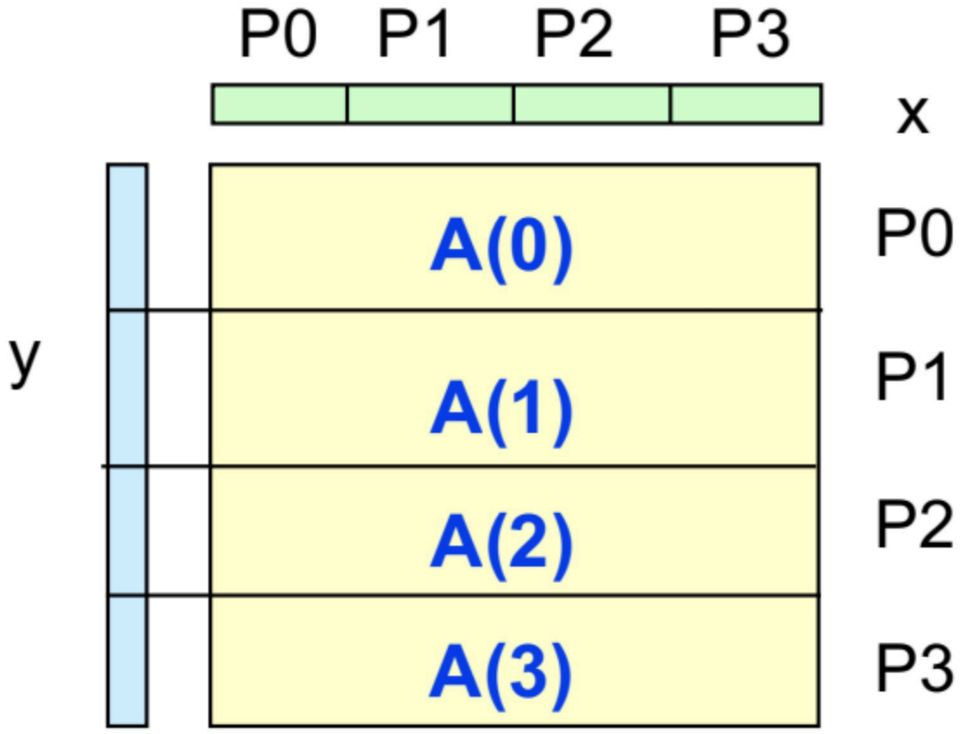
Row-major mat-vec $y = Ax$

- Assume every processor owns
  - A block of x and y
  - A block of rows of A
- What MPI commands are needed to be able to compute y?



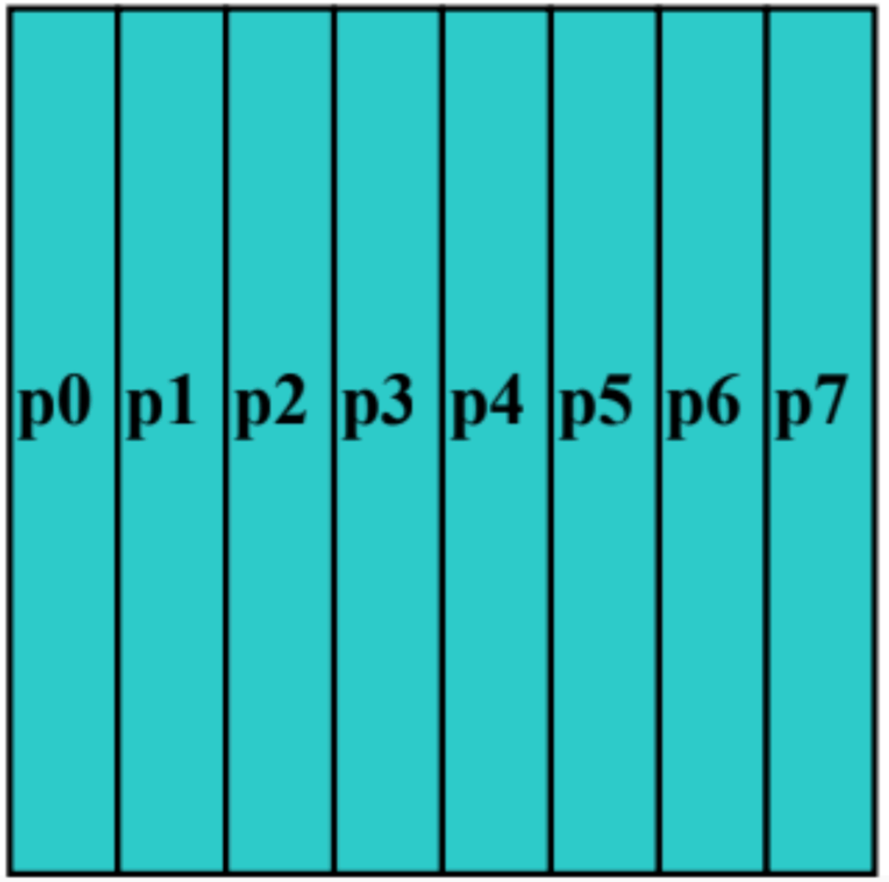
Row-major mat-vec $y = Ax$

- Assume every processor owns
  - A block of x and y
  - A block of rows of A
- What MPI commands are needed to be able to compute y?
  - Each processor needs the entirety of “x” to compute local “y” blocks
  - MPI_Allgather to collect the blocks of “x”
  - Leads to redundant storage of “x”



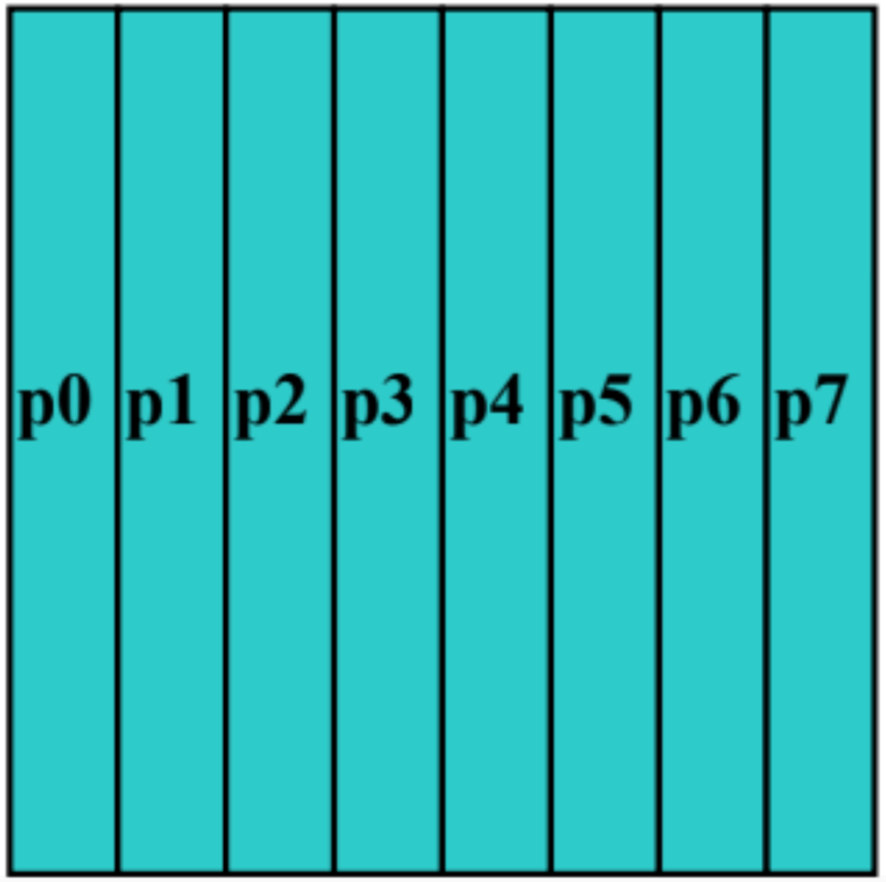
Column-major mat-vec $y = Ax$

- Assume every processor owns a block of x and y, and a block of columns of A
- What MPI commands could be used to to compute y?
  - Each processor computes a local matrix-vector product $y_i = A_i * x_i$
  - Need to distribute local block y_i to all processors and sum up.



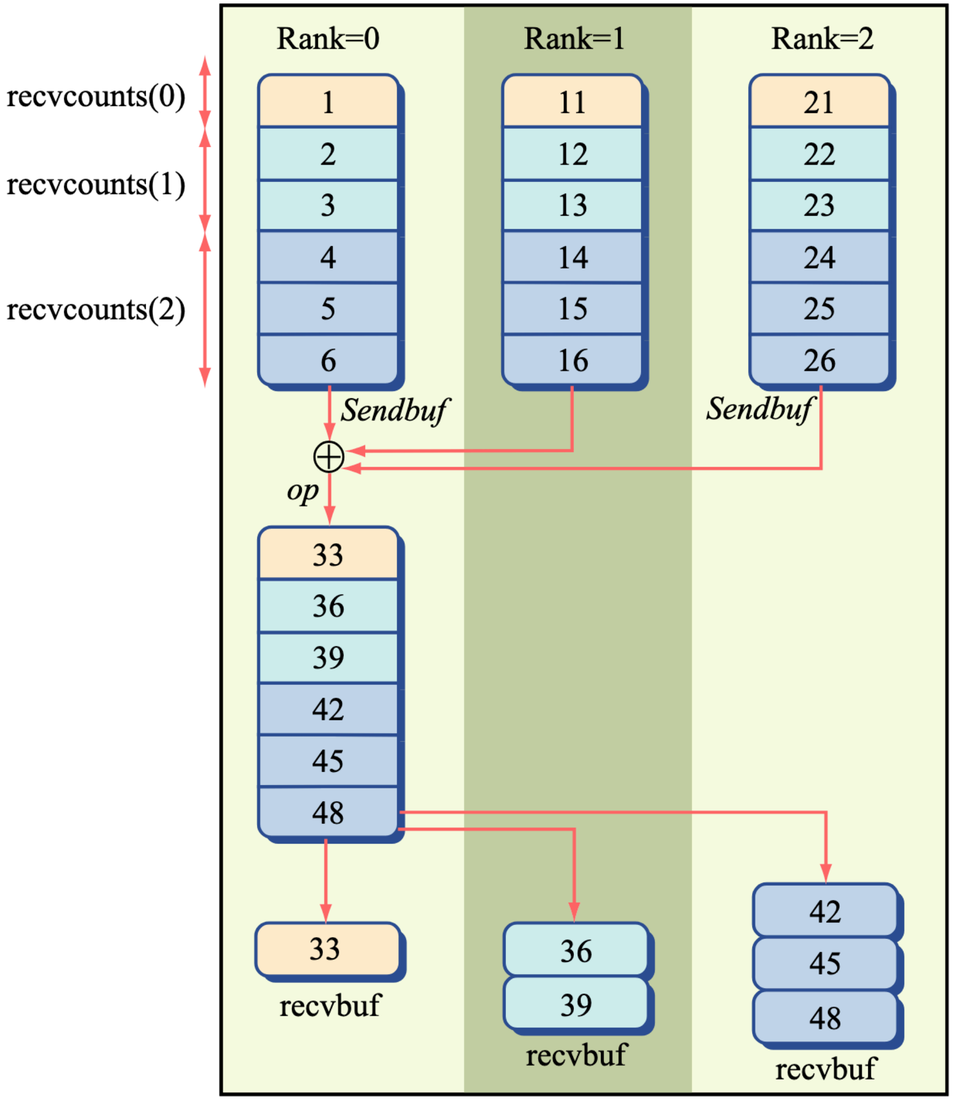
Column-major mat-vec $y = Ax$

- Assume every processor owns a block of x and y, and a block of columns of A
- What MPI commands could be used to to compute y?
  - Each processor computes a local matrix-vector product $y_i = A_i * x_i$
  - Need to distribute local block y_i to all processors and sum up.
  - Sub-optimal option: MPI_Reduce to compute $y_1 + y_2 \ldots$, then MPI_Scatter



A better way: MPI_Reduce_scatter

- Each processor computes a local matrix-vector product $A_i * x_i$
  - Need to sum up local contributions $A_i * x_i$ and distribute to all processors.
- MPI_Reduce_scatter: combine the reduction and scatter operation into one single collective.
  - Reduce-scatter with recvcount = length of local chunk.



MPI_Reduce_scatter illustration

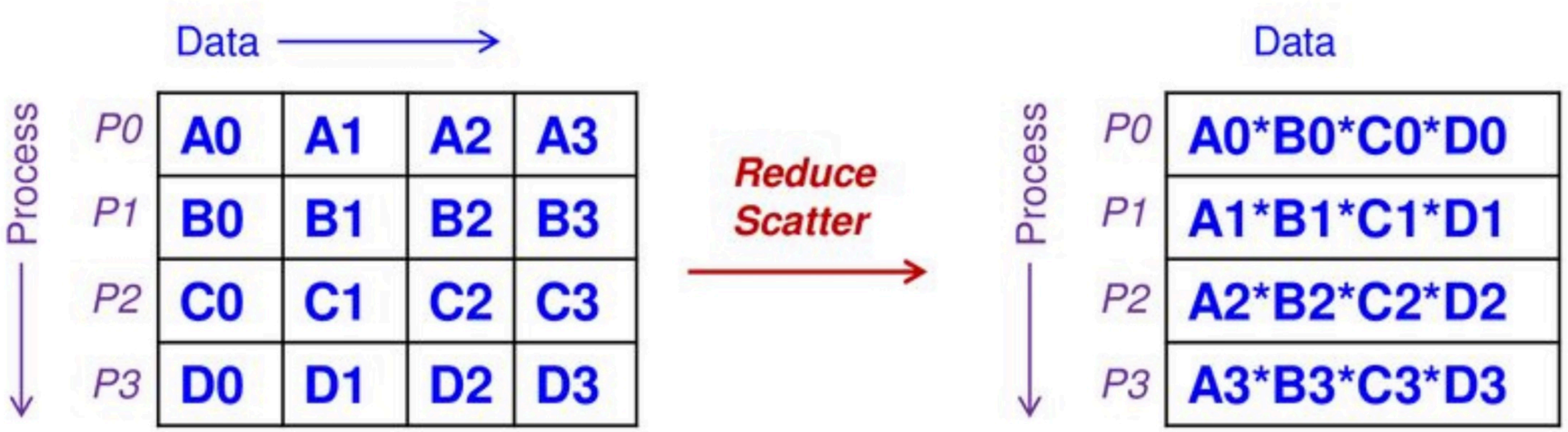


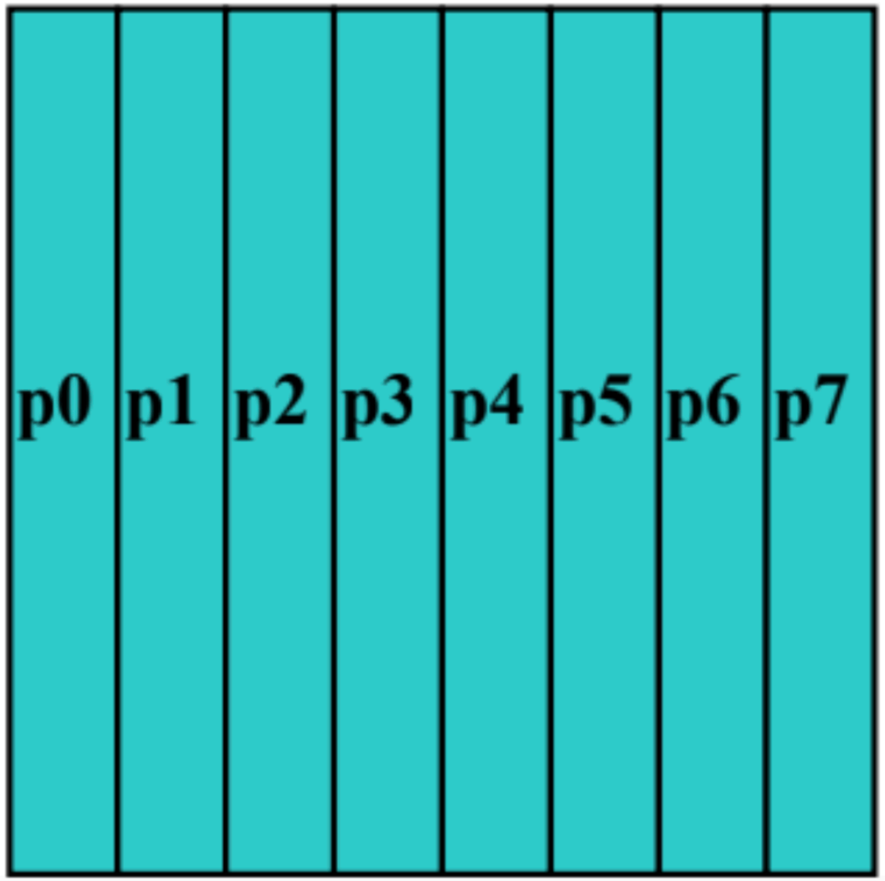
Figure from Mohammad Hammoud

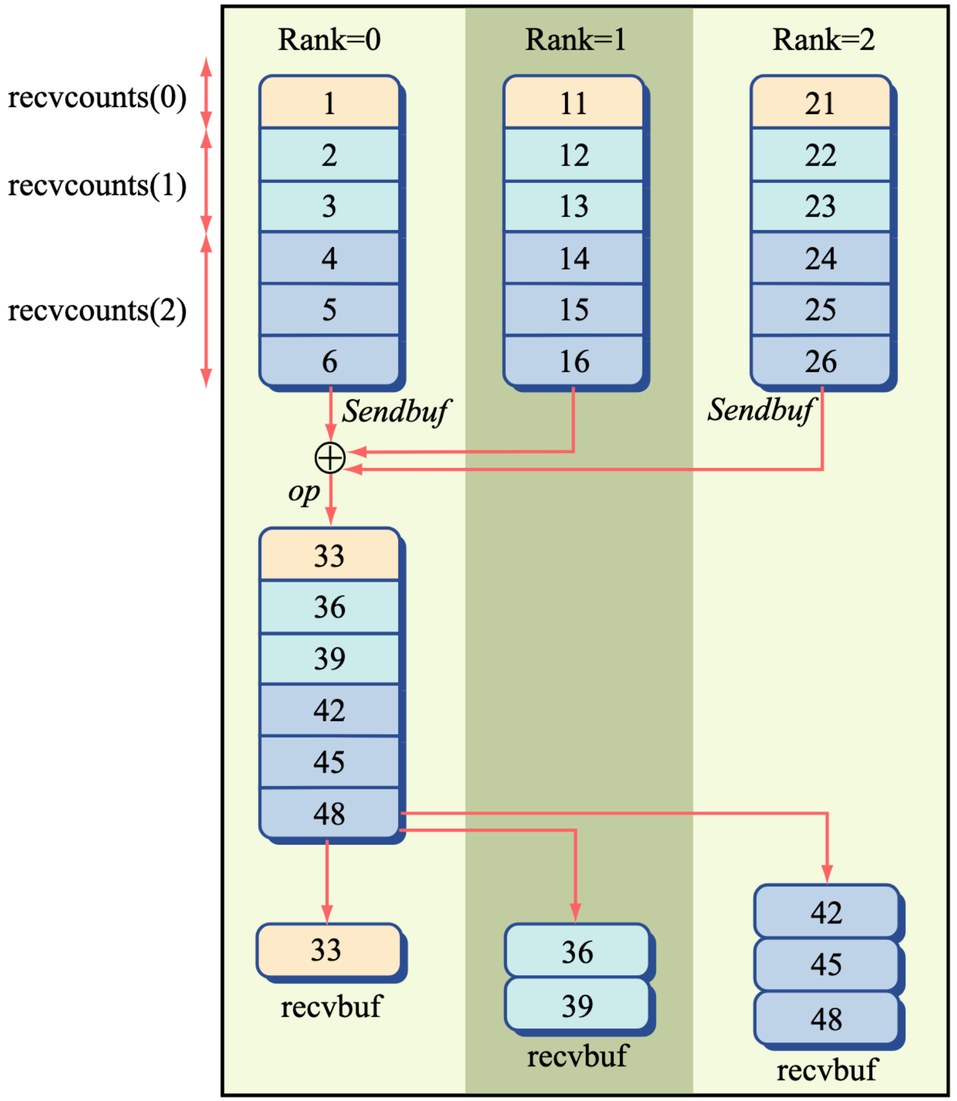
MPI_Reduce_scatter function signature

```
int MPI_Reduce_scatter(
    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)

Demo: mat-vec with MPI_Reduce_scatter





Which is better for MPI: row or column major?

- Row major uses Allgather, column major uses Reduce-scatter
  - Both require forming an entire vector (“x” for row major, “y” for column major)
  - MPI_Reduce_scatter has better bandwidth complexity
- Intuition: MPI_Allgather redundantly forms “x” on every single rank, resulting in a cost of βpn
  - Here, p, n are the number of ranks and length of x

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	$\mathcal{O}(\alpha \log p + \beta pn)$

arXiv:2410.14234v4 [cs.DC] 13 Feb 2025

Optimal, Non-pipelined Reduce-scatter and Allreduce Algorithms

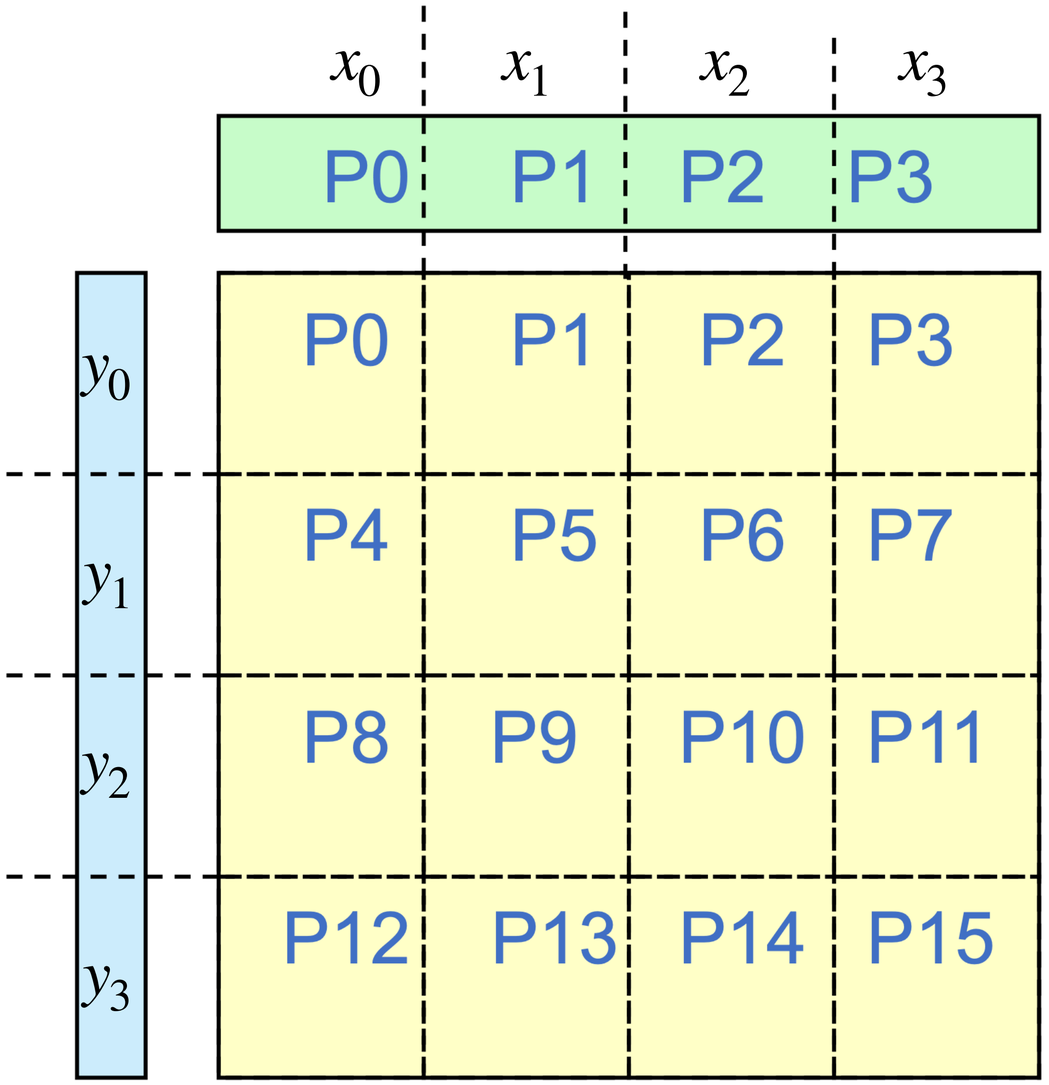
Jesper Larsson Träff
TU Wien Faculty of Informatics Institute of Computer Engineering, Research Group Parallel Computing 191-4

$$T(m,p) = \alpha\lceil \log_2 p \rceil + \beta \frac{p-1}{p} m + \gamma \frac{p-1}{p} m$$

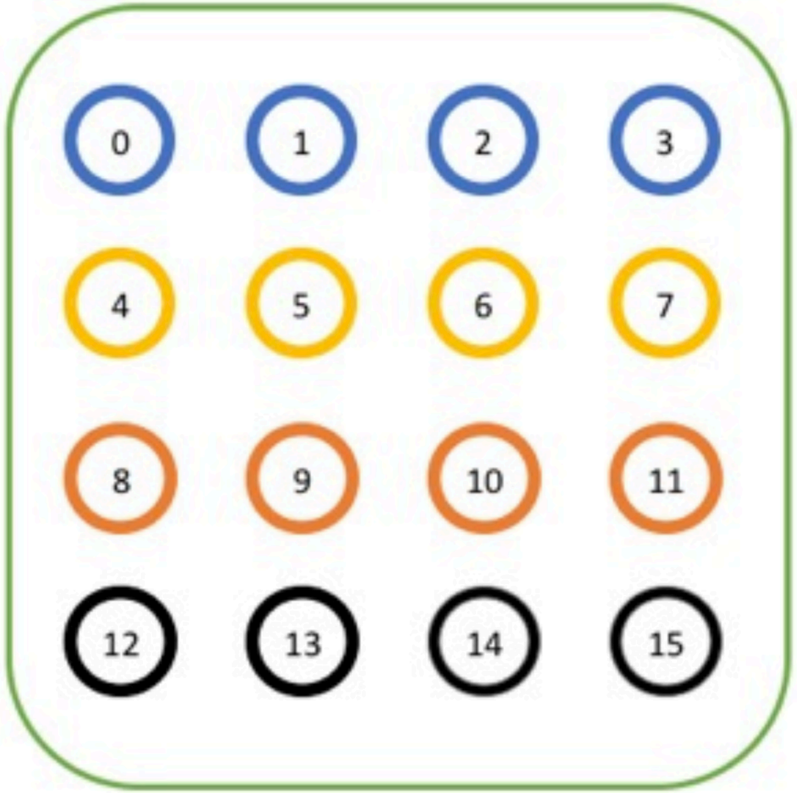
Arithmetic costs

Mat-vec $y = Ax$ with block matrix storage

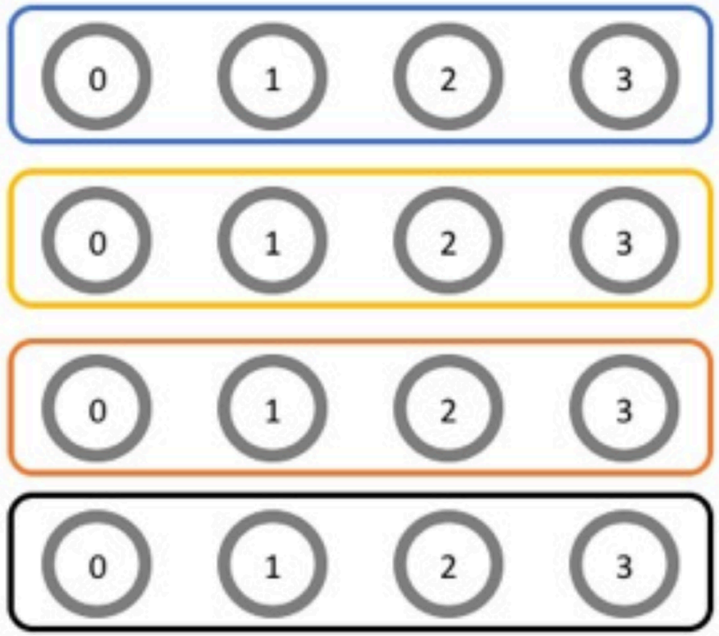
- Assume p processors arranged in a $\sqrt{p} \times \sqrt{p}$ grid. Who owns what now?
- Suppose y, x are distributed among the different processors. Then,
  - $y_0 = A_{p_0} x_0 + A_{p_1} x_1 + A_{p_2} x_2 + A_{p_3} x_3$
  - $y_1 = A_{p_4} x_0 + A_{p_5} x_1 + A_{p_6} x_2 + A_{p_7} x_3$
  - And so on for y_2, y_3…
- It would be nice if we could restrict collective operations to only a **subgroup** of processors!



Collectives and communicator groups



MPI_COMM_WORLD



Custom communicator groups for each row

- Collectives + MPI communicator groups: the bulk of the work gets moved to data decomposition and setup steps

Custom MPI communicators

- Custom MPI communicators are useful when you need to define **groups of processes** over which to perform collective MPI operations
- **MPI_Comm_split:** splits processes in an existing communicator (for example, MPI_COMM_WORLD) into a new communicator.
- **MPI_Comm_create, etc**: can be used to create more general groups
  - Can also communicate in-between communicator groups

MPI_Comm_split

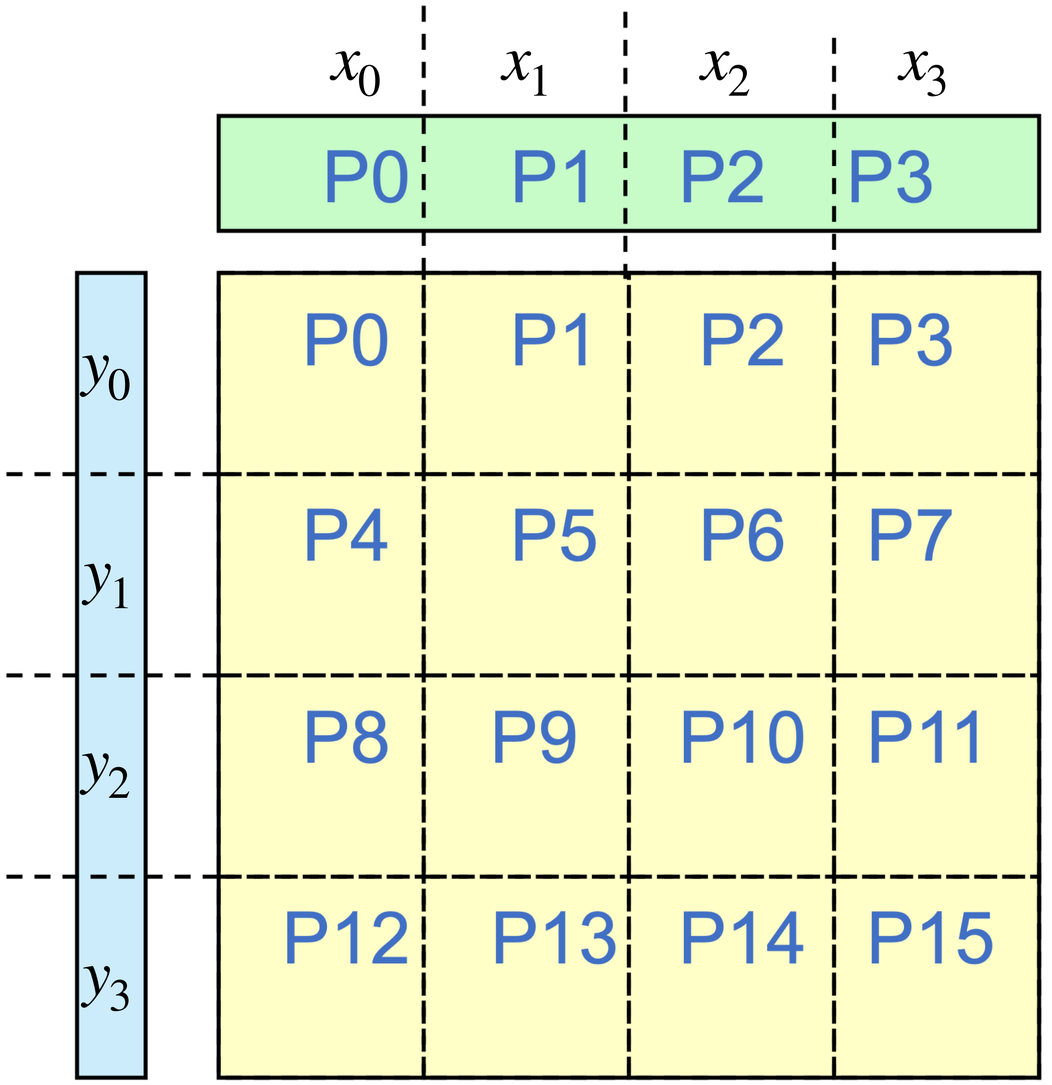
```
int MPI_Comm_split(
    MPI_Comm comm,
    int color,
    int key,
    MPI_Comm newcomm,
);

// Note: no tag anymore
```

Argument	Description
comm	The communicator to split
	For example, MPI_COMM_WORLD
color	Processes with the same “color” are in the same new communicator
key	Int
	The order (not value) determines the new local ordering
newcomm	The new communicator

- Probably easier to understand with an example: mpi_communicators.cpp

Revisiting $y = Ax$ with block matrix storage

- Assume p processors arranged in a $\sqrt{p} \times \sqrt{p}$ grid.
- How would you compute the matrix-vector product for this data decomposition?
  - What custom MPI communicator groups would you create?
  - What MPI collective commands would you use, and over which communicators?
- What advantages does block storage have over row major and column major storage?



Now…what about matrix-matrix multiplication?

Distributed matrix-matrix multiplication

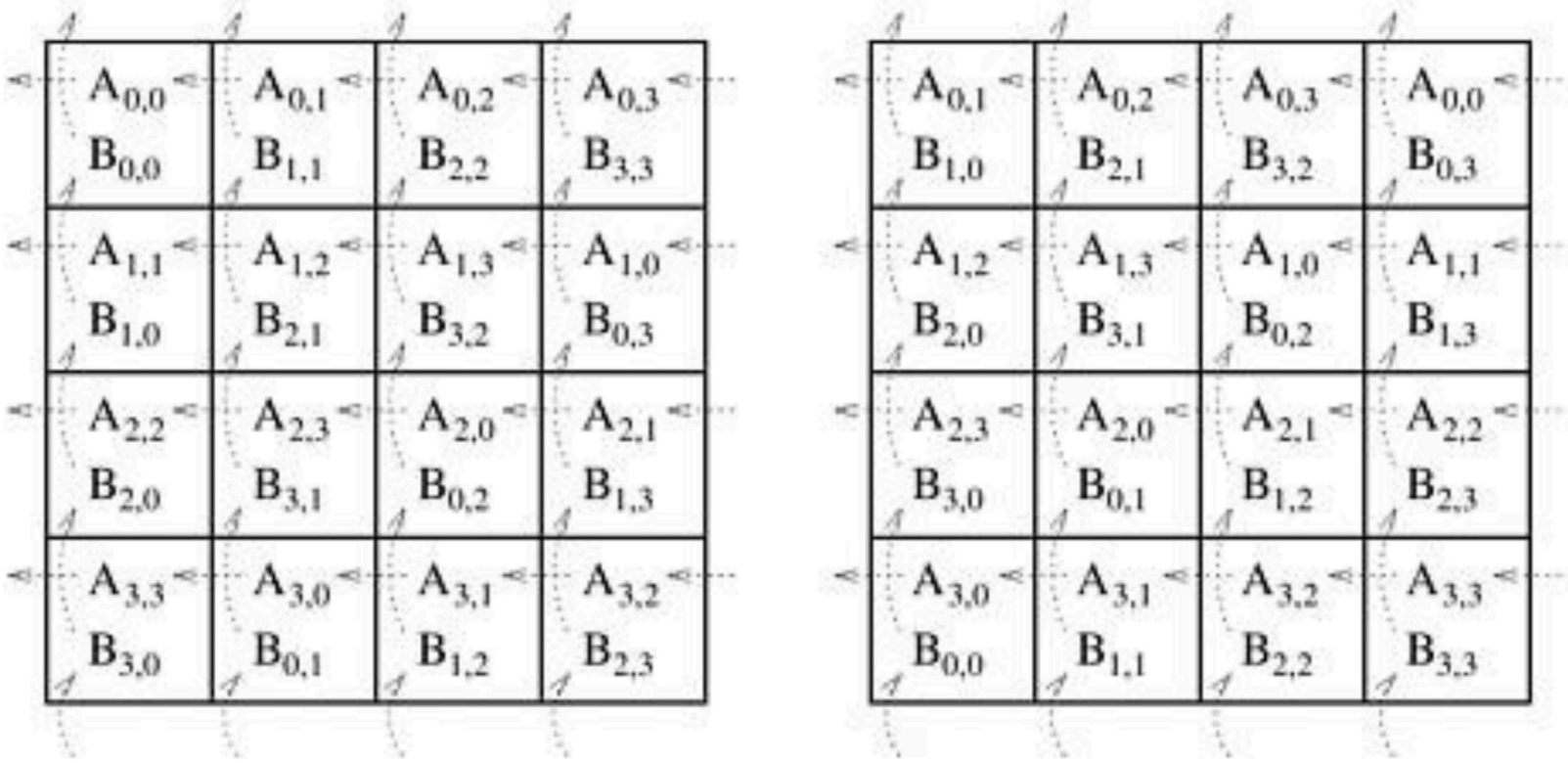
- As usual, we'll do a block decomposition of each matrix.
- Each processor holds a block of C, A, B, and the ordering of the blocks is the same across each matrix.
- What communication/MPI commands should we use?

Communication for matrix multiplication

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

- With matrix-vector multiplication, the access pattern is simple and there's not a lot of communication between different ranks.
- Memory access and communication patterns are more complex for matrix-matrix multiplication.
  - **Want to avoid redundant storage or communication of sub-blocks.**
  - As a result, distributed parallel matrix-matrix multiplication algorithms are a bit more nuanced in their design and implementation.

Cannon's algorithm



- Each processor holds A, B, C blocks, performs an initial "alignment" step.
- Cannon's algorithm performs $\sqrt{p}$ "shift" communication steps (which are actually pretty easy to do using point-to-point communication).

Cannon's algorithm for a 2x2 matrix

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

$$C_{00} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0} \qquad C_{01} = A_{0,1}B_{1,1} + A_{0,0}B_{0,1}$$

$$C_{10} = A_{1,1}B_{1,0} + A_{1,0}B_{0,0} \qquad C_{11} = A_{1,0}B_{0,1} + A_{1,1}B_{1,1}$$

Cannon's algorithm for a 2x2 matrix

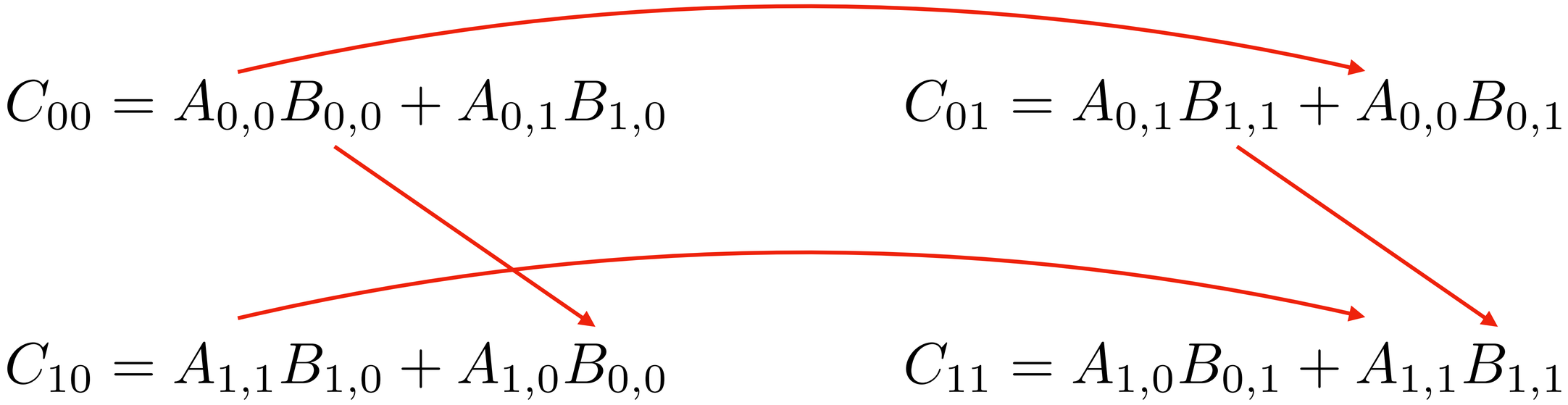
$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

$$C_{00} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0} \qquad C_{01} = A_{0,1}B_{1,1} + A_{0,0}B_{0,1}$$

$$C_{10} = A_{1,1}B_{1,0} + A_{1,0}B_{0,0} \qquad C_{11} = A_{1,0}B_{0,1} + A_{1,1}B_{1,1}$$

Cannon's algorithm for a 2x2 matrix

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$



Cannon's algorithm for a 3x3 matrix

3x3 Matrices Multiplication Formula

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

- Idea: identify products of sub-blocks that can be computed independently.
- Additional constraint: assume that each sub-block is only available to one processor at a time (to avoid redundant communication).

Cannon's algorithm for a 3x3 matrix

3x3 Matrices Multiplication Formula

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

- Idea: identify products of sub-blocks that can be computed independently.
- Additional constraint: assume that each sub-block is only available to one processor at a time (to avoid redundant communication).

Cannon's algorithm for a 3x3 matrix

3x3 Matrices Multiplication Formula

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

- Idea: identify products of sub-blocks that can be computed independently.
- Additional constraint: assume that each sub-block is only available to one processor at a time (to avoid redundant communication).

Cannon's algorithm for a 3x3 matrix

3x3 Matrices Multiplication Formula

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

- Idea: identify products of sub-blocks that can be computed independently.
- Additional constraint: assume that each sub-block is only available to one processor at a time (to avoid redundant communication).

Cannon's algorithm for a 3x3 matrix

Initial data layout for Cannon's algorithm

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{22} & a_{23} & a_{21} \\ a_{33} & a_{31} & a_{32} \end{bmatrix} \quad \begin{bmatrix} b_{11} & b_{22} & b_{33} \\ b_{21} & b_{32} & b_{13} \\ b_{31} & b_{12} & b_{23} \end{bmatrix}$$

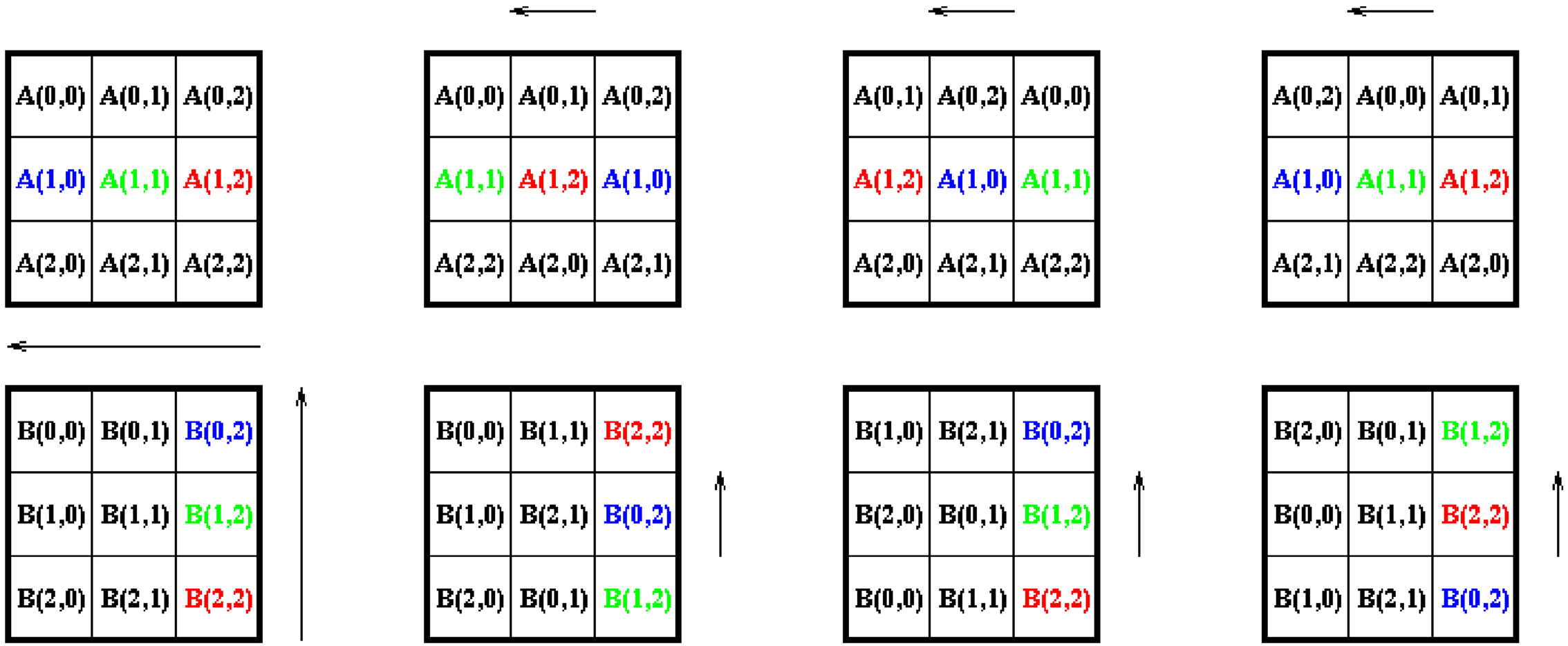
3x3 Matrices Multiplication Formula

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

- Idea: identify products of sub-blocks that can be computed independently.
- Additional constraint: assume that each sub-block is only available to one processor at a time (to avoid redundant communication).

Cannon's algorithm for a 3x3 matrix



Initial A, B

A, B initial alignment

A, B after shift step 1

A, B after shift step 2

Figure from Zhiliang Xu

Cannon's algorithm for a general matrix

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} B_{(i+j+k)\%\sqrt{p},j}$$

- Utilizes the above decomposition of a single block of a matrix-matrix product
- Initial alignment, starting with Aij and Bij on each processor:
  - For A blocks: shift each matrix in the ith block row "i" steps to the left
  - For B blocks: shift each matrix in the jth block column "j" steps up
  - Uses periodic shifts for both A and B.
- After initial alignment, accumulate local matrix products. Then, perform $\sqrt{p}$ additional shift and accumulate steps.

Figure from Zhiliang Xu

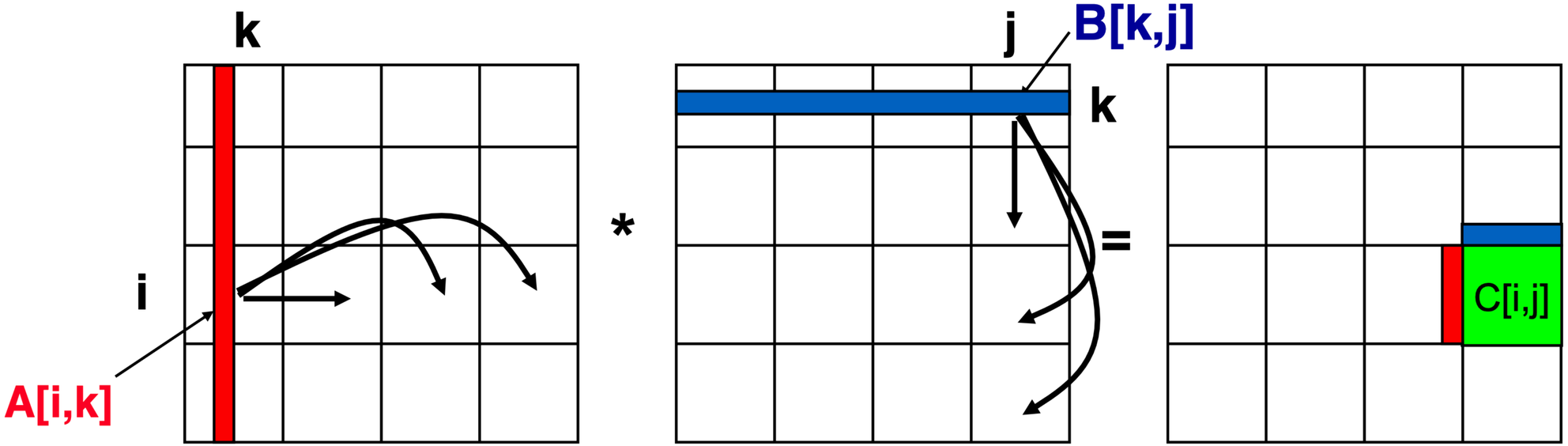
More on Cannon's algorithm

- Communication efficiency is pretty good!
- Main downside: not very flexible
  - Gets kind of ugly to generalize to rectangular matrices and matrices whose dimensions aren't easy to decompose into a processor grid.
- Also doesn't take advantage of MPI collective communication.
- Another variant: Fox's algorithm, which uses broadcast + shifts.
- Often implemented using an MPI Cartesian **topology.**

Scalable Universal Matrix Multiplication Algorithm

- An alternative distributed matrix multiplication algorithm: SUMMA (Scalable Universal Matrix Multiplication Algorithm) by van de Geijn and Watts (1997).
  - Slightly less efficient (communication-wise) but easy to implement + flexible.
- Idea: $C = \sum_{k=1}^{n} A_{(:,k)} B_{(k,:)}$, where $A_{(:,k)}$, $B_{(k,:)}$ are the kth column and row of A, B.
- Compute C as an accumulation of outer products: this yields a broadcast-friendly communication pattern.

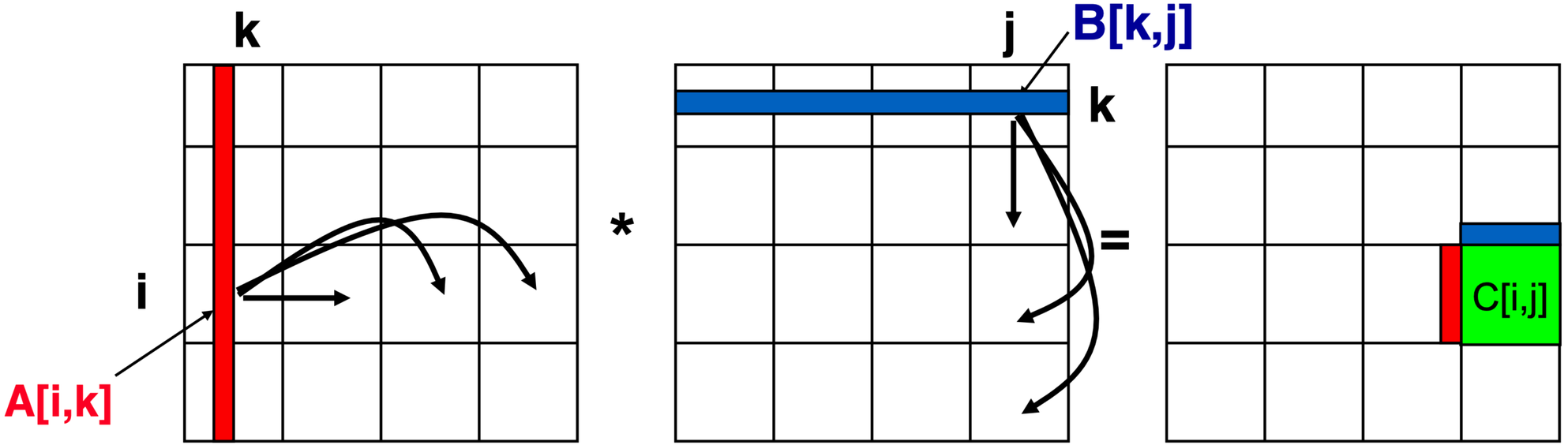
SUMMA matrix multiplication algorithm



- Idea: $C = \sum_{k=1}^{n} A_{(:,k)} B_{(k,:)}$, where $A_{(:,k)}$, $B_{(k,:)}$ are the kth column and row of A, B.
- Compute C as an accumulation of outer products: this yields a broadcast-friendly communication pattern.

Figure from Zhiliang Xu

Computing a rank-1 outer product in parallel



- Assume for now that each column of A is distributed among a “column” of processors, while a row of B is distributed across a “rows” of processors.
- Each processor needs the kth column of A and row of B; can use MPI_Bcast to share.
  - The processor that owns the current column of A broadcasts it across that row of processors
  - The current row of B is similarly broadcast across a column of processors.

More general implementations of SUMMA

- For $n \times n$ matrix multiplication, SUMMA computes n different outer products
  - If n is large, can incur overhead due to communication latency costs.
- Instead of computing n different rank-1 outer products, we can compute the matrix $C = AB$ as the sum of n/k rank-k outer products of $A_{(:,i)}, B_{(i,:)}$.
  - Increasing k communicates more information per broadcast, but results in fewer overall broadcasts.
- Fairly straightforward to extend to rectangular matrices.

Homework 3: to be posted tonight

- The homework will be to implement Cannon's algorithm and/or SUMMA.
  - Simple cases: square matrices, perfectly divisible dimensions
  - Using MPI collectives will make things much easier
  - Check the result by comparing it against serial matrix multiplication
    - Some memory wrangling and re-arranging will be necessary when gathering matrix blocks and re-assembling on a single rank.

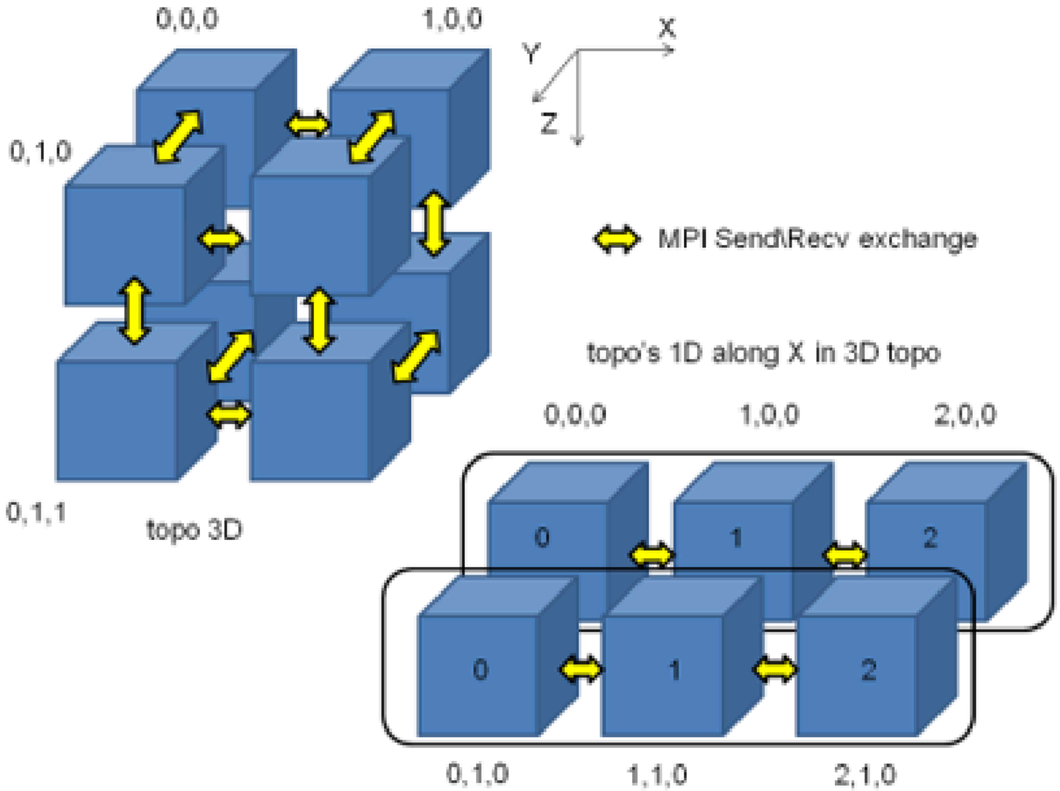
Some additional MPI topics

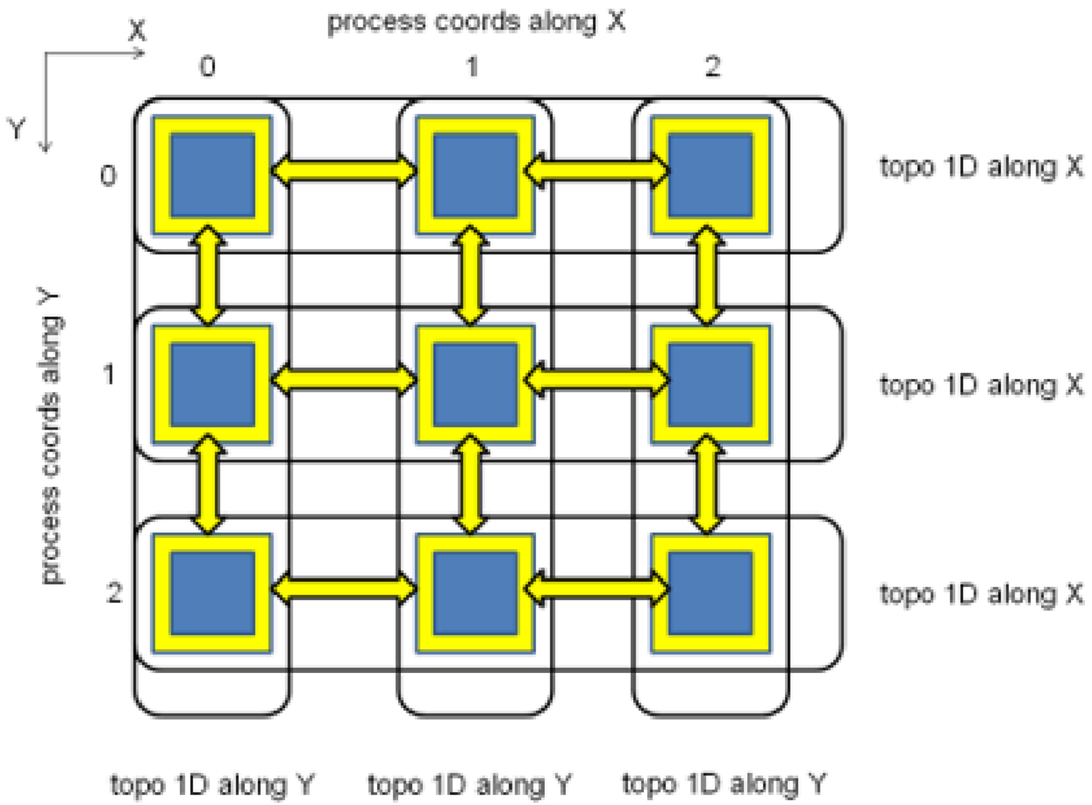
- We haven’t covered all of MPI, but it’s well documented.
- Some other things to be aware of:
  - MPI Cartesian topologies
  - MPI graph topologies
  - One-sided communication and MPI windows

MPI Cartesian topologies

- MPI_Comm_split allows you to create custom communicator groups
- However, for Cannon's algorithm, we need *periodicity* of the subgroup of ranks as well (e.g., the first and last ranks are neighbors).
- An MPI "Cartesian" topology represents an *arbitrarily high dimensional* grid of processors, and allows users to:
  - Easily reference "neighbor" ranks in each direction, as well as ranks which are some "distance" away in any direction
  - Specify periodicity along certain dimensions.

MPI Cartesian topology illustrations



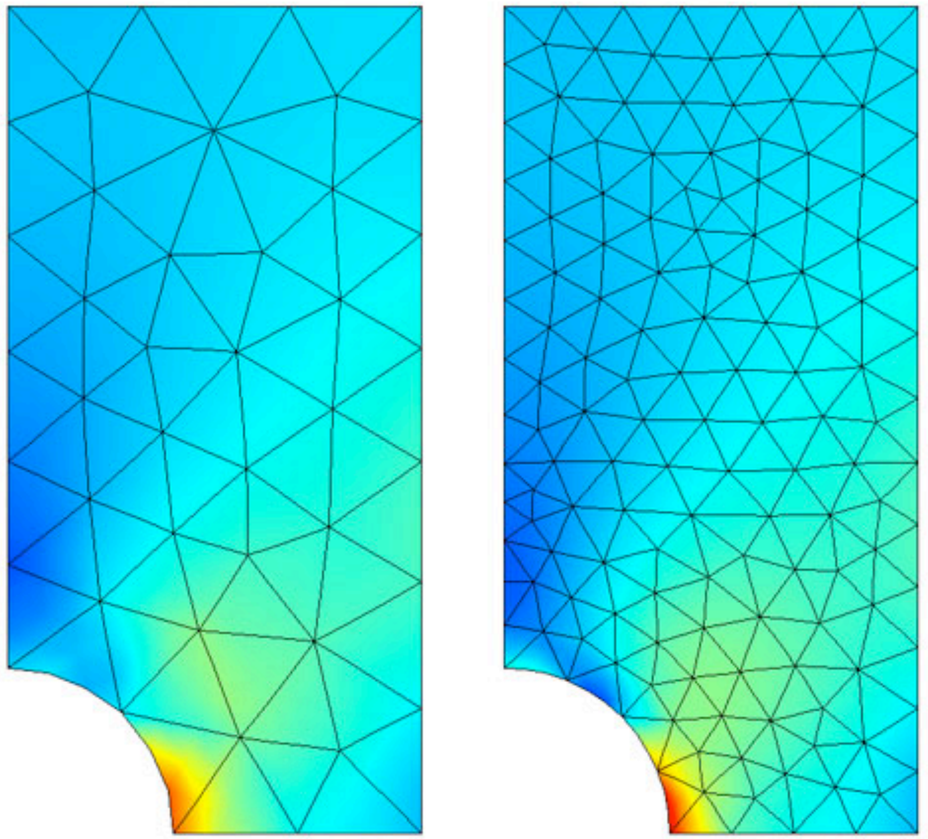


- Cartesian topologies can be arbitrarily high dimensional, anisotropic, etc.

Karavaev and Laevsky (2021)

MPI graph topologies

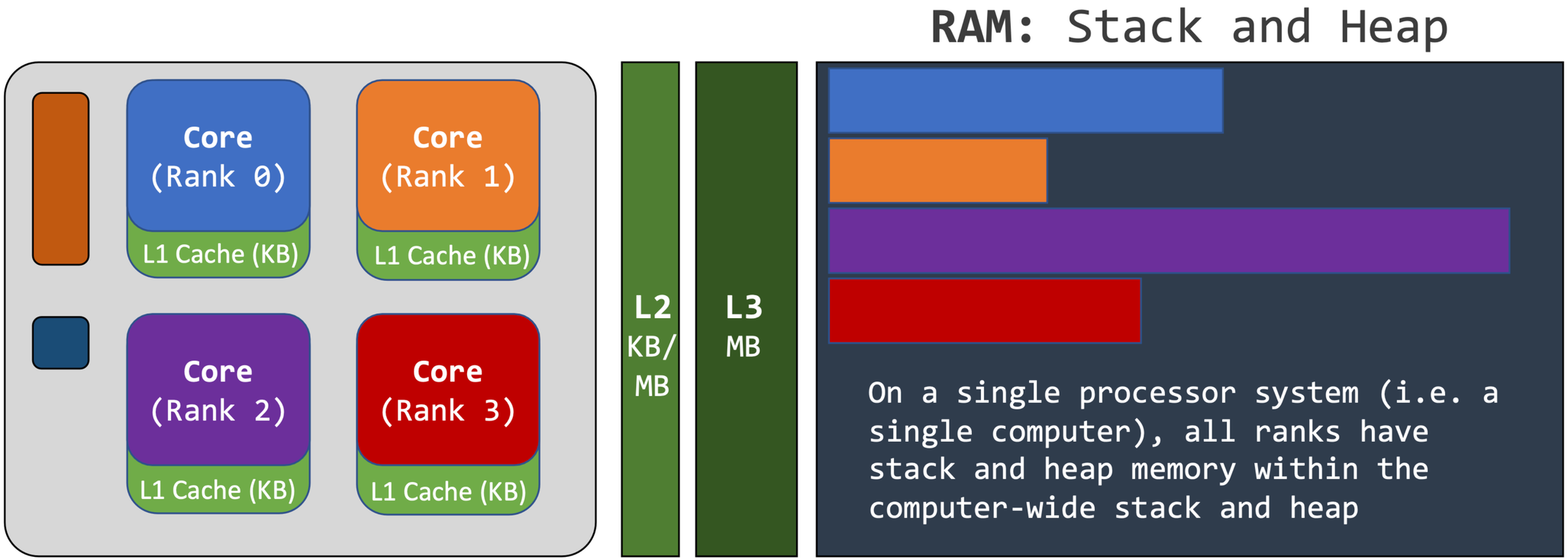
- MPI Graph topologies
  - Each “vertex” is an MPI process with “edges” between communicating processes.
  - MPI allows users to specify edge weights, which represent how much information is communicated between different processes.
- Useful for unstructured data (e.g., meshes)
- More general, but slower.



One-sided communication

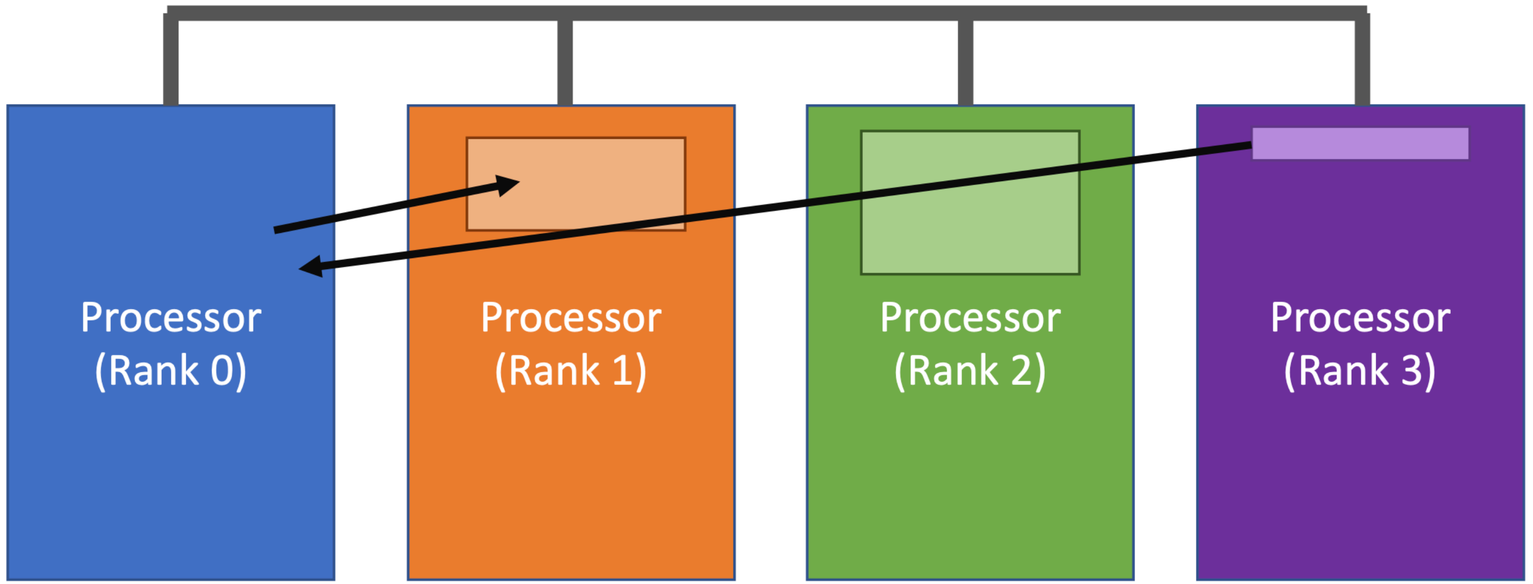
- Also known as “remote memory access” (RMA)
- One-sided communication allows for one rank to get data from other ranks without them needing to be involved in the communication
  - The receiving ranks still have to participate, so it is not *truly* one-sided.
- Since other ranks do not need to participate, the communication pattern:
  - Does not need to be known beforehand
  - Does not run the risk of communication-based deadlocks

What happens in one-sided communication?



- Instead of exchanging buffers of data, one rank directly accesses another rank's memory (remote memory access - RMA).
- Similar to shared memory (in fact, uses shared memory protocols when available)

What happens in one-sided communication?



- Processors can expose part of their memory for others to read and write to.
- Danger: race conditions and synchronization are issues again.

One-sided synchronization via “windows”

- MPI supports RMA via the MPI_Window object
  - The window reflects the section of exposed memory plus synchronization tools to check its status, i.e., whether it is valid to read/write from a window
- MPI provides methods for creating, reading from, writing to, synchronizing, and destroying windows
  - Creating an MPI window: MPI_Win_create (+ other more complex options)
  - Destroying, reading, and writing: MPI_Win_free, MPI_Get, MPI_Put
  - Synchronizing: MPI_Win_fence, MPI_Win_lock/unlock,