Parallel Programming
Lecture 11

Prof. Dr. Jonas Posner

jonas.posner@cs.hs-fulda.de

Hochschule Fulda
University of Applied Sciences

Winterterm 2025/2026
E-Learning: AI5085

23rd January 2026

Plan for Today

- Exam registration in Horstl by 25th January
  - Sign up for an appointment via Moodle starting 26th January
  - MPI assignment release: today
- Finish lecture 10 from slide 253
- Recap
  - Point-to-Point (P2P) Communication
- Today
  - Collective communication

Collective Communication: Introduction

- Collective communication routines are **high-level operations**
- They involve **multiple processes simultaneously**
- Enable **optimized internal implementations**
  - E.g., tree-based algorithms
- Can be built from point-to-point communication *in principle*
  - However, collectives are typically **significantly faster** than equivalent point-to-point implementations

Collective Communication: Categories

- **One-To-All**
  - One process computes/holds the data; all processes receive it
  - Examples: `MPI_Bcast`, `MPI_Scatter`
- **All-To-One**
  - All processes contribute data; one process receives the result
  - Examples: `MPI_Gather`, `MPI_Reduce`
- **All-To-All**
  - All processes send to and receive from all other processes
  - Examples: `MPI_Allgather`, `MPI_Alltoall`
- **Synchronization**
  - No data exchange; used for coordination (e.g., `MPI_Barrier`)

Collective Communication: General Properties

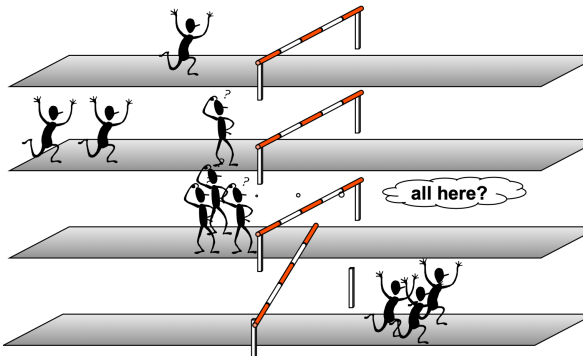
- Operate over a **communicator** (e.g., `MPI_COMM_WORLD`)
- **All** processes in the communicator must call the collective routine
- Synchronization may or may not occur, but all processes must be able to enter the collective routine
- Can be **blocking** or **non-blocking** (since MPI 3.0)
- **No** message tags are used
- Receive buffers must have **exactly** the same size as the corresponding send buffers

Why Prefer Collective Communication?

- **Conciseness:**
  - Programs are shorter and more readable
- **Efficiency:**
  - Implementations are highly optimized
- **Performance Portability:**
  - Implementations can be tailored to the target architecture

Barrier Synchronization

- Synchronizes all processes (in the communicator)



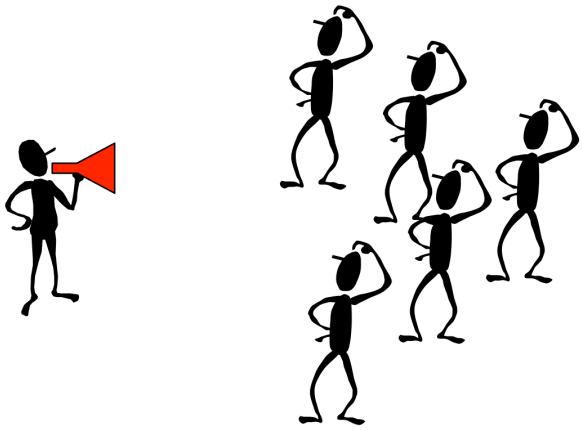
Barrier Synchronization

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

- Synchronizes **all** processes in the communicator
- Should be used **sparingly** — waiting time is unproductive
- Useful for:
  - Timing measurements
  - Debugging and profiling
- The **simplest** collective operation: no data is exchanged
- A process returns from `MPI_Barrier` only **after** all processes in the communicator have entered it

Broadcast: Overview

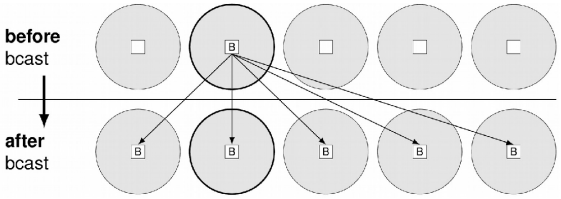
- A **one-to-all** communication pattern



Broadcast: Signature and Semantics

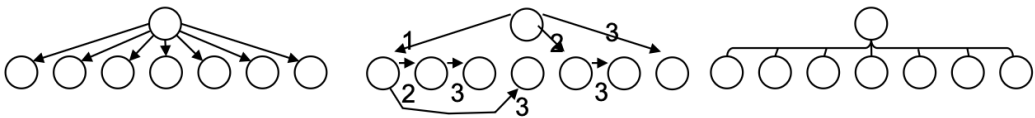
```
int MPI_Bcast(void* buffer, int count, MPI_Datatype
datatype, int root, MPI_Comm comm);
```

- **One-to-all** communication: the process with rank `root` sends identical data to all processes in the communicator
- **All** processes in the communicator must call `MPI_Bcast`
- The internal algorithm is **implementation-dependent**
  - Often uses tree-based algorithms for efficient distribution



Internal Implementation: Tree-Based Algorithms

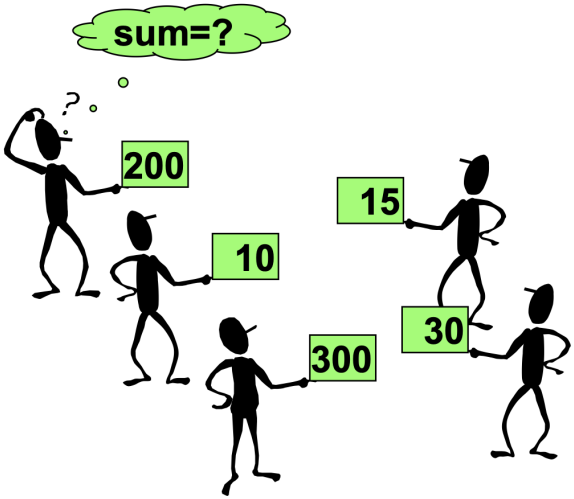
- Example: `MPI_Bcast`



- You do **not** need to implement this yourself
- This is the responsibility of the MPI library

Reduction: Overview

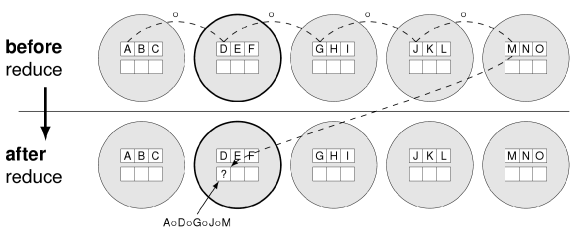
- Combine data from multiple processes to produce a single result



Reduction: Signature and Semantics

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

- Performs the global reduction operation `op`; the process with rank `root` receives the result
- The operation is applied **element-wise** to the send buffers
  - I.e., `recvbuf[k]` contains the reduction over the `k`-th element contributed by all processes



Predefined Reduction Operations

Operation Handle	Description
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical Exclusive OR
MPI_BXOR	Bitwise Exclusive OR
MPI_MAXLOC	Maximum + index of maximum value
MPI_MINLOC	Minimum + index of minimum value

User-Defined Reduction Operations: Requirements

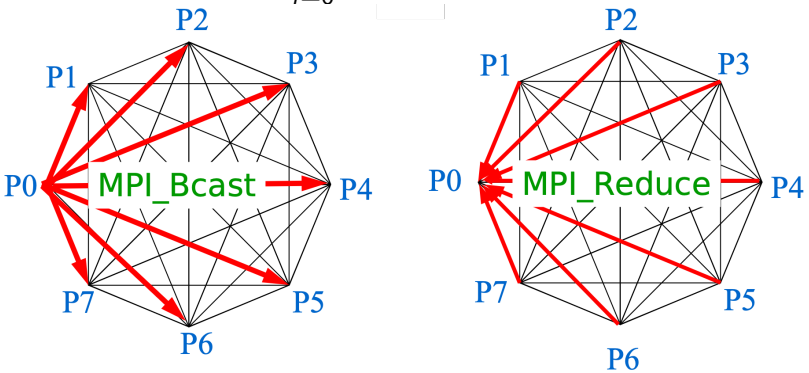
- MPI allows defining **custom reduction operators**
- Requirements for a user-defined operator $\square$:
  - Must be **associative**
  - Must compute the operation $\vec{A} \square \vec{B}$
- The defined operator can be used with:
  - `MPI_Reduce`, `MPI_Allreduce`, `MPI_Reduce_scatter`, `MPI_Scan`, `MPI_Exscan`
- Creating a user-defined operator:
  - `int MPI_Op_create(MPI_User_function* func, int commute, MPI_Op *op)`

User-Defined Reduction Operations: Semantics

- **Semantics of `MPI_Op_create`:**
  - Defines the function `func` as reduction operator `op`
  - `commute` specifies whether the operator is commutative
- **Freeing a user-defined operator:**
  - `int MPI_Op_free(MPI_Op *op)`
  - Deletes the operator `op` and sets it to `MPI_OP_NULL`

Example: Combining Broadcast and Reduction

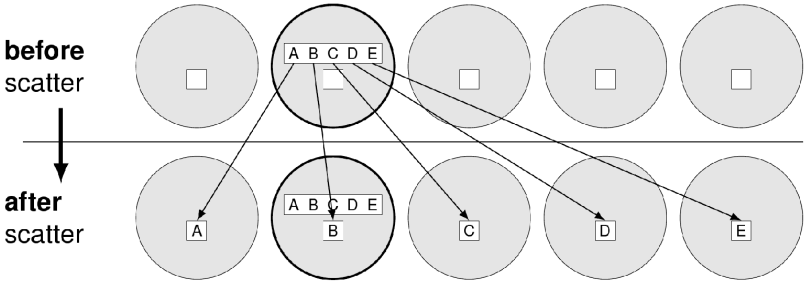
$$\sum_{i=0}^{\texttt{size}-1} i^k, \quad k \in \mathbb{N}$$



- **Step 1:** Distribute parameter k using `MPI_Bcast`
- **Step 2:** Compute global sum using `MPI_Reduce`

Scatter: Motivation

- **Problem:** A vector of length n shall be distributed evenly across p processes (where n is divisible by p)
- **Solution:** `MPI_Scatter`



- The vector is split into p equally sized chunks of length n/p
- The i-th chunk is sent to process i

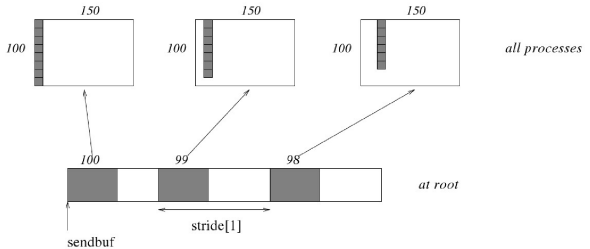
Scatter: Signature and Semantics

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

- Distributes data from the `root` process to all processes in the communicator
- Each process receives an **equal-sized** chunk (`recvcount` elements)
- The root provides `size` chunks of `sendcount` elements in `sendbuf`

Scatterv: Motivation

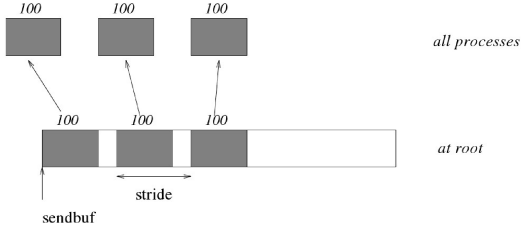
- `MPI_Scatter` is **only applicable** if every process receives the same number of elements
  - I.e., the vector length n must be divisible by the number of processes p
- `MPI_Scatter` is **not suitable** if:
  - Processes should receive **different** numbers of elements
  - The data to be distributed is **not contiguous** in memory



Scatterv: Signature and Example

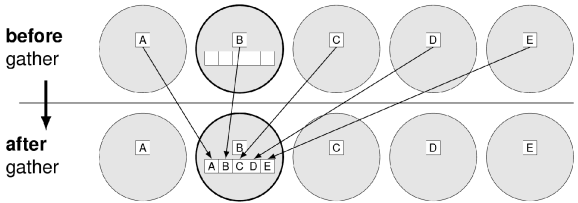
```
int MPI_Scatterv(const void* sendbuf, const int
sendcounts[], const int displs[], MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int
root, MPI_Comm comm);
```

- **Example:**
  - Root distributes blocks of 100 integers to all processes
  - The blocks are **not contiguous** in memory
  - Stride of 120 elements between block starts



Gather: Motivation

- **Problem:** The elements of a vector of length n are distributed evenly across p processes (where n is divisible by p) and shall be collected at the root process
- **Solution:** `MPI_Gather`



- In the receive buffer at the root, data is stored in **rank order**: first from process 0, then process 1, etc.
- A receive buffer of sufficient size must be provided (programmer's responsibility)

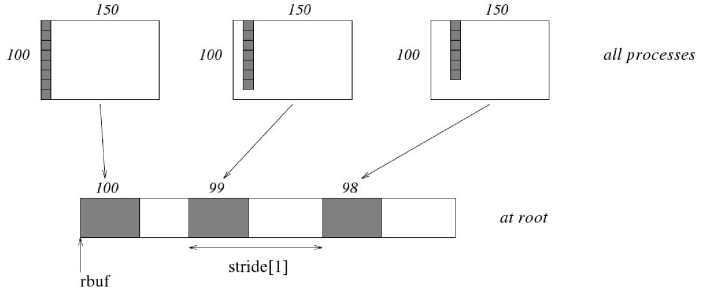
Gather: Signature and Semantics

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

- Collects data distributed across the process group at the `root` process
  - At `root`, data is stored in rank order: 0, 1, ..., `size-1`

Gatherv: Motivation

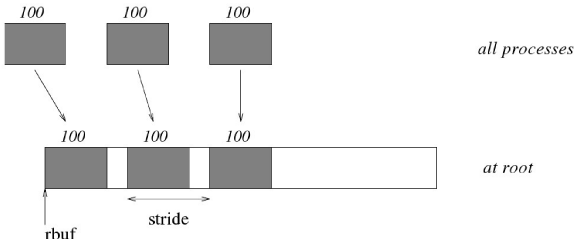
- `MPI_Gather` is **only applicable** if every process sends the same number of elements to the root process
- `MPI_Gather` is **not suitable** if:
  - Processes should send **different** numbers of elements to the root
  - The collected data should **not** be stored contiguously in memory at the root



Gatherv: Signature and Example

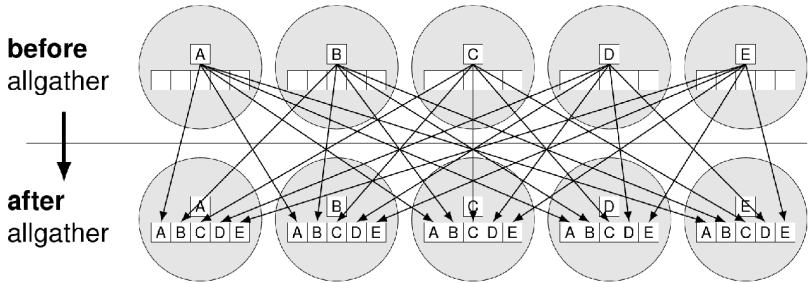
```
int MPI_Gatherv(const void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, const int
recvcounts[], const int displs[], MPI_Datatype recvtype,
int root, MPI_Comm comm);
```

- Example:
  - Collect data distributed across all processes at the `root` process
  - Each process contributes 100 elements of the global vector
  - At `root`, blocks are stored with a stride of 120 elements in `rbuf`



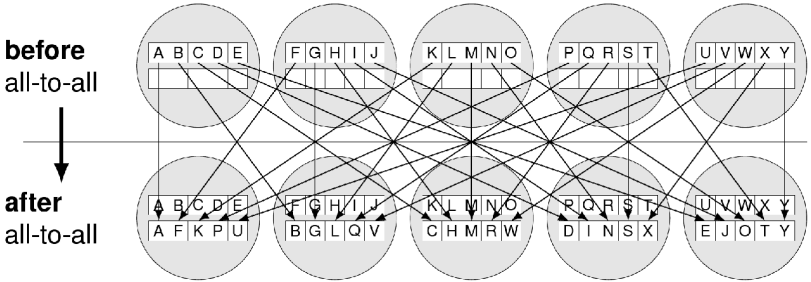
Allgather: Collective Gather with Distribution

- Collects data distributed across all processes and distributes the result to **every** process
- After completion, the gathered data is available on **all** processes
- Conceptually equivalent to `MPI_Gather` followed by `MPI_Bcast`



Alltoall: Complete Exchange

- `MPI_Alltoall` extends `MPI_Allgather`
  - Each process sends **different data** to all other processes
  - The j-th block sent by process i is received by process j and stored at the position of the i-th block
  - All blocks must have the **same size** and be stored **contiguously** in memory



Non-Blocking Collective Communication (1/2)

- Combines the benefits of **non-blocking** point-to-point communication with **optimized** collective implementations
- Non-blocking collective routines start with `I` (for *immediate*)
  - E.g., `MPI_Ibcast`, `MPI_Ireduce`
- Principle similar to non-blocking point-to-point communication
- A call to a non-blocking collective **initiates** the operation and returns **immediately**
- The operation must be completed with a separate call (e.g., `MPI_Wait`, `MPI_Test`)
- Once initiated, the operation can progress in the background **concurrently** with computation

Non-Blocking Collective Communication (2/2)

- Completion of a non-blocking collective is **local** to the calling process
- Completion does **not** imply:
  - That other processes have also completed (or even started) the operation
  - That previously posted non-blocking collective operations have completed
- Unlike point-to-point communication, non-blocking collectives must **not** be mixed with blocking collectives
- All processes must call all collective operations in the **same order**

Example: Incorrect — Deadlock

```
1  ...
2  if (my_rank == 0) {
3      MPI_Bcast(buf1, count, type, 0, comm);
4      MPI_Send(buf2, count, type, 1, tag, comm);
5  }
6  else if (my_rank == 1) {
7      MPI_Recv(buf2, count, type, 0, tag, comm, &status);
8      MPI_Bcast(buf1, count, type, 0, comm);
9  }
10 ...
```

- This example leads to a **deadlock**!
- Process 0 blocks in `MPI_Bcast` until process 1 enters the broadcast
- Process 1 never reaches the broadcast because it blocks in `MPI_Recv`, waiting for data from process 0

Example: Correct — Using Non-Blocking Collectives

```
1  ...
2  if (my_rank == 0) {
3      MPI_Ibcast(buf1, count, type, 0, comm, &request);
4      MPI_Send(buf2, count, type, 1, tag, comm);
5  }
6  else if (my_rank == 1) {
7      MPI_Recv(buf2, count, type, 0, tag, comm, &status);
8      MPI_Ibcast(buf1, count, type, 0, comm, &request);
9  }
10 MPI_Wait(&request, &status);
11 ...
```

Collective Communication: Summary (1/3)

- `MPI_Barrier`
  - Synchronizes all processes in the communicator
- `MPI_Bcast`
  - Broadcasts data from the root process to all processes
- `MPI_Reduce`
  - Reduces values from all processes using an operator; result stored at the root
- `MPI_Allreduce`
  - Like `MPI_Reduce`, but the result is returned on all processes

Collective Communication: Summary (2/3)

- `MPI_Scatter`
  - Splits a send buffer at the root into equal-sized chunks and distributes one to each process
- `MPI_Scatterv`
  - Like `MPI_Scatter`, but allows varying chunk sizes (and displacements) per process
- `MPI_Reduce_scatter`
  - Reduces values from all processes and scatters chunks of the reduced result to all processes
- `MPI_Scan`
  - Inclusive prefix reduction: process i receives the reduction over processes 0 to i

Collective Communication: Summary (3/3)

- `MPI_Gather`
  - Collects equal-sized chunks from all processes and assembles them at the root (in rank order)
- `MPI_Gatherv`
  - Like `MPI_Gather`, but allows varying chunk sizes (and displacements) at the root
- `MPI_Allgather`
  - Like `MPI_Gather`, but the assembled result is returned on all processes
- `MPI_Alltoall`
  - Each process sends a distinct block to every other process and receives one block from each