



703308 VO High-Performance Computing WS2021/2022

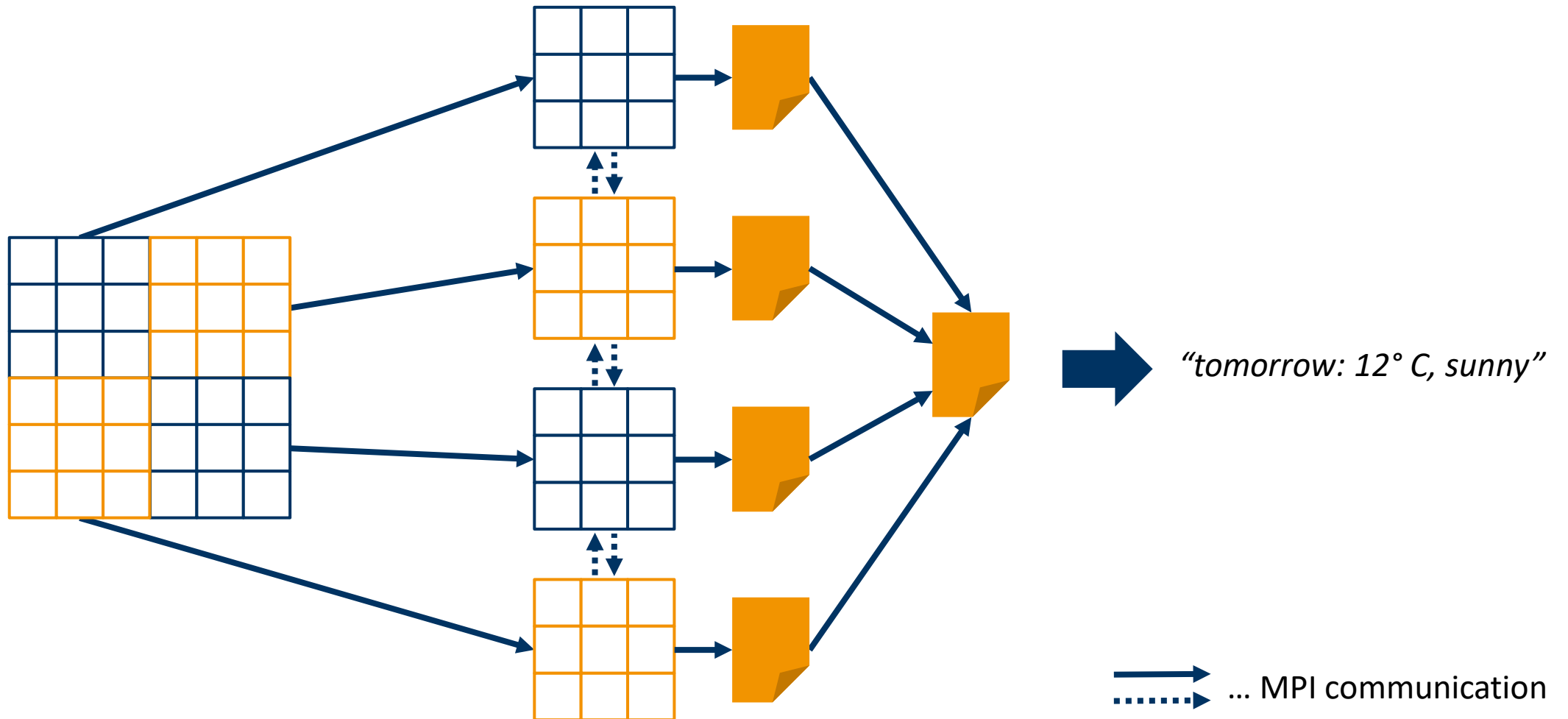
MPI - Message Passing Interface

Philipp Gschwandtner

Overview

- ▶ general concepts about MPI
 - ▶ characteristics
 - ▶ programming model
 - ▶ startup
- ▶ point-to-point communication
- ▶ collective communication
- ▶ practical example

Motivation for Using MPI: Data Distribution



Message Passing Interface (MPI)

- ▶ message passing library for distributed memory parallelism
- ▶ de-facto standard for C/C++ and Fortran
- ▶ maintained by the MPI Forum
 - ▶ initial release in 1994 (version 1.0)
 - ▶ updates in 1997 (2.0), 2012 (3.0), 2015 (3.1) and 202x (4.0)
 - ▶ specification updates slow, aim at stability and high TRL
 - ▶ “On Dec 6th 2017, the MPI forum voted for new voting rules (effective Dec 6th, 2017).”

MPI Implementations

- ▶ **OpenMPI**

- ▶ open source
- ▶ merge of multiple previous MPI implementations
- ▶ default on many systems

- ▶ **MPICH**

- ▶ also open source
- ▶ basis for many vendor implementations such as Intel, IBM, Cray, Microsoft, ...
 - ▶ default on many systems

- ▶ Do not confuse implementation versions with specification versions!
- ▶ Do not confuse implementation adherence with specification adherence!

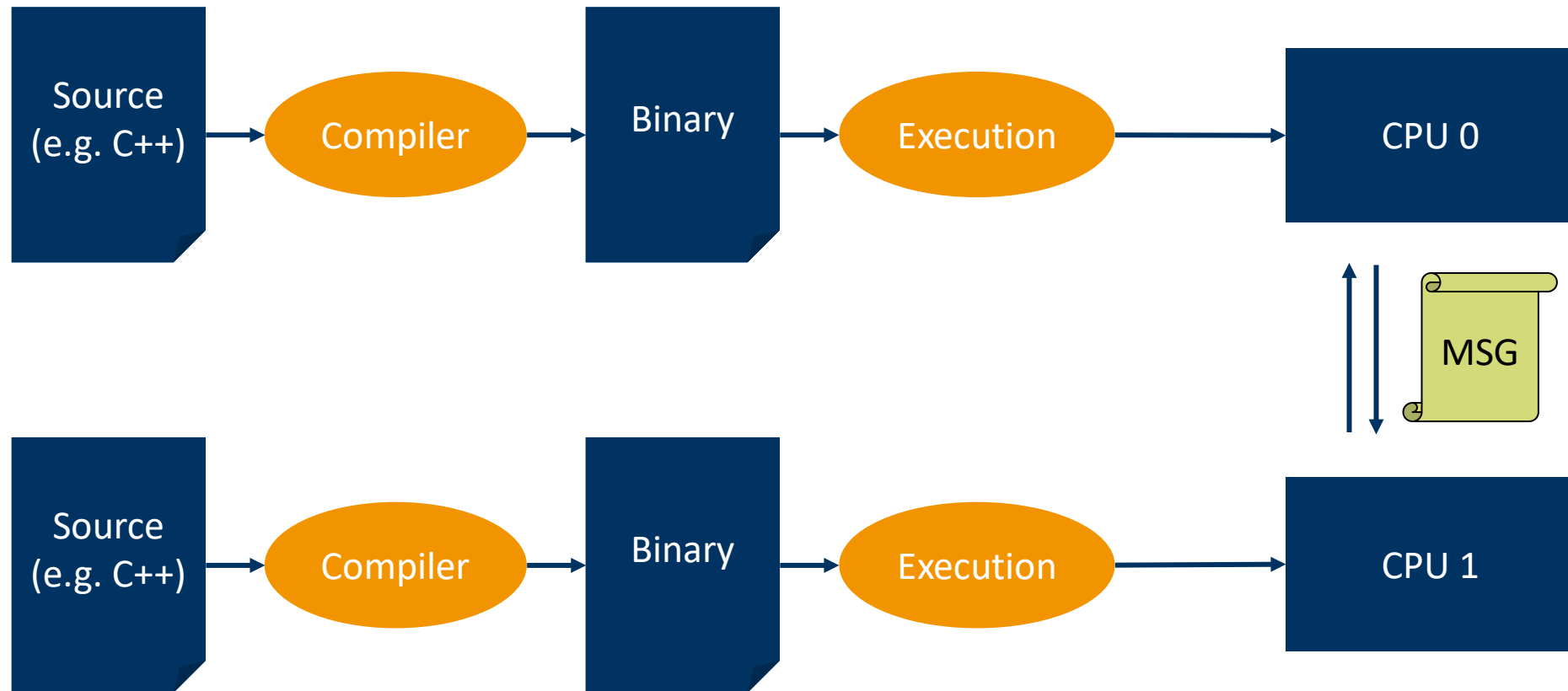
Main Characteristics

- ▶ offers specific tools for
 - ▶ sending and receiving messages
 - ▶ waiting and synchronization
 - ▶ identification of individual processes
 - ▶ ...
- ▶ additional convenience tools for
 - ▶ partitioning and distributing data
 - ▶ organizing processes in structures
 - ▶ large-scale I/O operations
 - ▶ ...

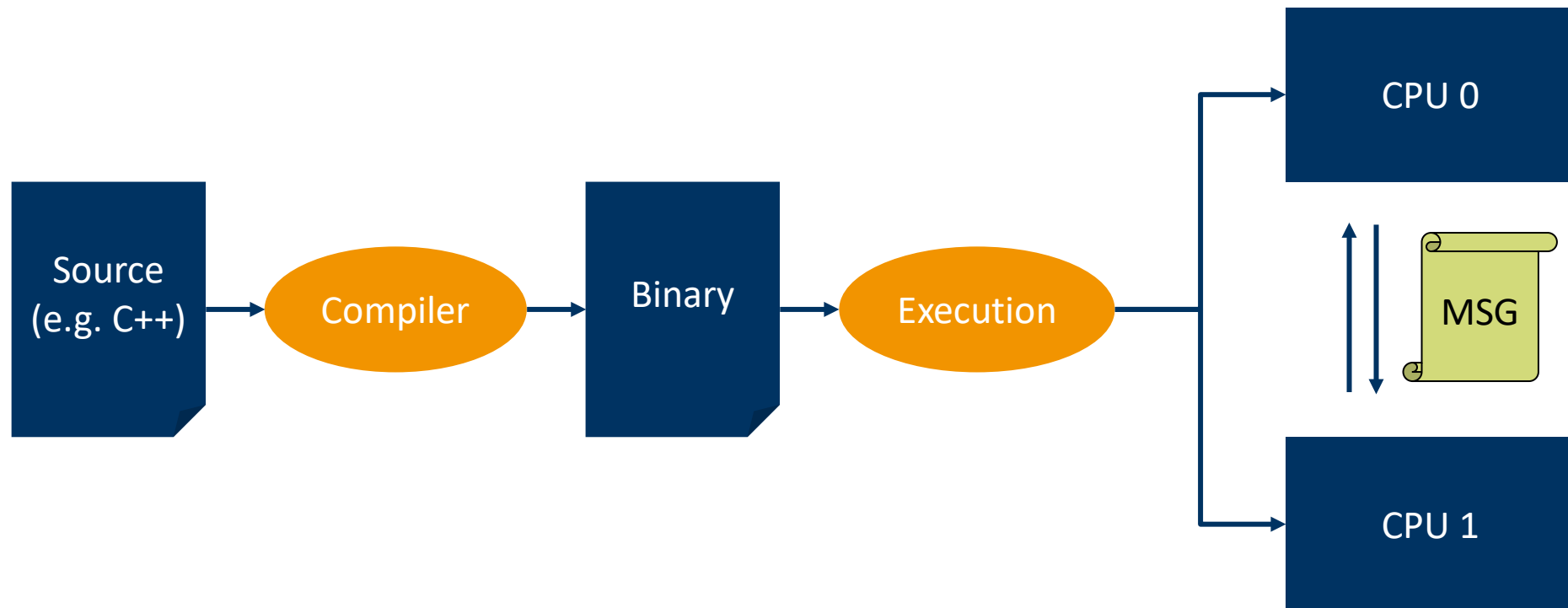
Main Characteristics cont'd

- ▶ a lot of user responsibility
 - ▶ explicit parallelism and communication
 - ▶ program correctness
 - ▶ performance optimization
 - ▶ (non-)blocking
 - ▶ (a)synchronous
- ▶ a lot of advantages
 - ▶ available everywhere
 - ▶ several implementations
 - ▶ portable to many architectures
 - ▶ very high performance

Recap: MIMD: MPMD



Recap: MIMD: SPMD



MPMD through SPMD

- ▶ many MPI implementations support only SPMD
- ▶ SPMD can emulate MPMD

```
int main() {  
    // get id information  
    int cpuID = ...;  
    if(cpuID==0) {  
        ... // program A  
    } else {  
        ... // program B  
    }  
}
```

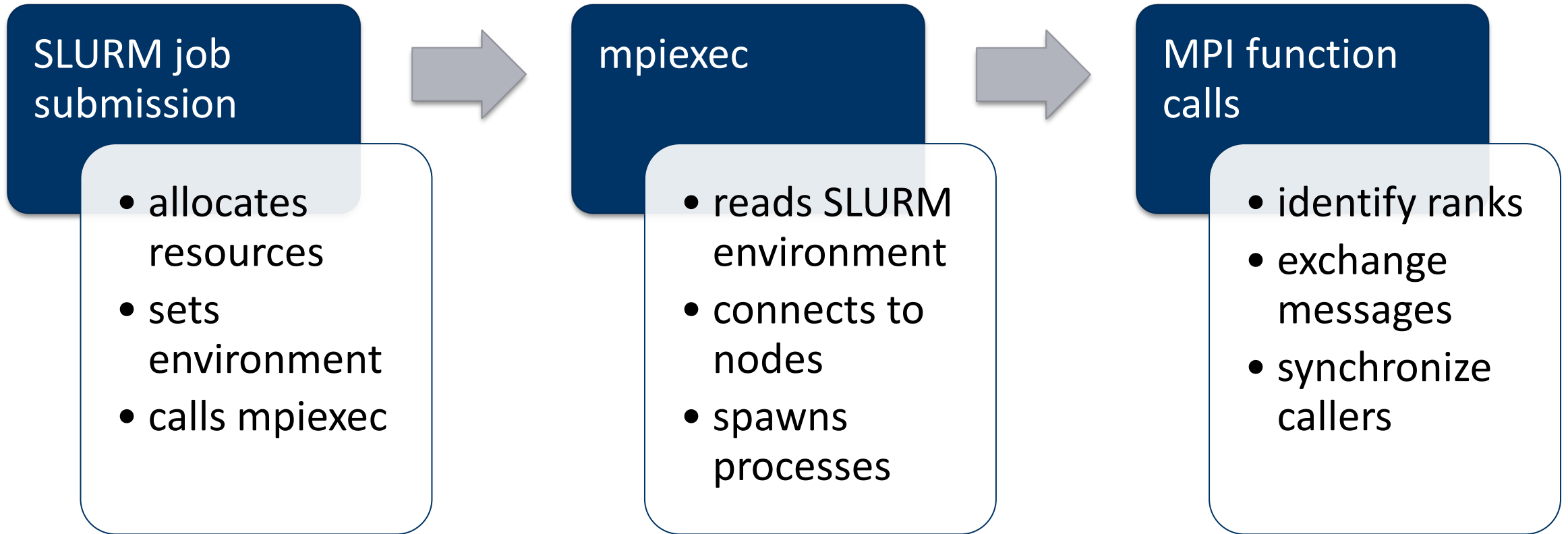
Parallelism Requires Two Mechanisms

- ▶ a mechanism for spawning processes
 - ▶ multiple ways of achieving this
 - ▶ we won't look at this in too much detail
 - ▶ simply rely on `mpiexec` to do the work for you
- ▶ a mechanism for sending and receiving messages
 - ▶ many, many ways of exchanging messages
 - ▶ we will definitely look at this in a lot of detail
 - ▶ tons of functionality to choose from, as we'll see in a bit

Compiler and Execution Wrapper

- ▶ `mpicc / mpic++` for compiling
 - ▶ OpenMPI: `--showme` prints compiler flags
 - ▶ passes all additional compiler flags to backend compiler (e.g. `mpicc -g`)
- ▶ `mpiexec` for running
 - ▶ formerly `mpirun`, but `mpiexec` is standardized

Startup Procedure of an MPI Application



Hello World in MPI

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv); // initialize the MPI environment
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size); // get the number of ranks
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get the rank of the caller
    printf("Hello world from rank %d of %d\n", rank, size);
    MPI_Finalize(); // cleanup
}
```

Setup and Teardown

- ▶ `int MPI_Init(int* argc, char*** argv)`
 - ▶ must be called by every process before calling any other MPI function
 - ▶ initializes the MPI library
- ▶ `int MPI_Finalize(void)`
 - ▶ must be the last MPI function called by every process
 - ▶ user must ensure completion of all (locally) pending communication
 - ▶ performs library cleanup

Who am I Talking to?

- ▶ in MPI-speak, processes are known as *“ranks”*
 - ▶ numbered from 0 to N-1
 - ▶ own rank can be queried with
`int MPI_Comm_rank(MPI_Comm comm, int* rank)`
 - ▶ number of ranks can be queried with
`int MPI_Comm_size(MPI_Comm comm, int* size)`
- ▶ almost all MPI semantics are relative to a *“communicator”* or *“group”*
 - ▶ identifies a set of ranks
 - ▶ `MPI_COMM_WORLD` means everyone, always available
 - ▶ new communicators and groups that hold subsets of ranks can be created
 - ▶ when developing a library, always create your own communicator!

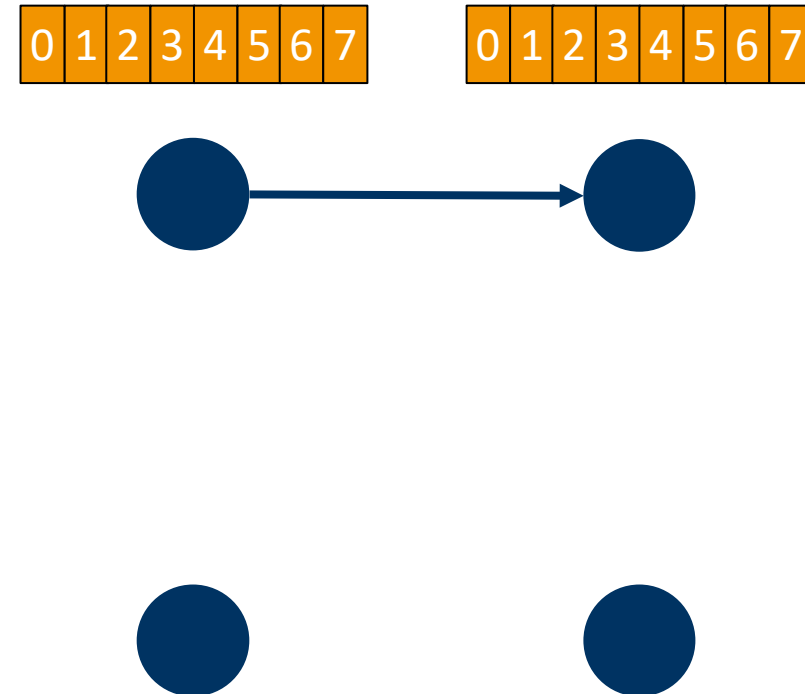


Point-to-Point Communication



Point-to-Point Communication

- ▶ `MPI_Send(...)/MPI_Recv(...)`
 - ▶ single sender, single receiver
(*“point-to-point”*)
- ▶ simplest form of communication available
 - ▶ not necessarily the best
- ▶ multiple different types
 - ▶ (a)synchronous
 - ▶ (non-)blocking



Basic Send/Receive Example

```
int number;
if (rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 42, MPI_COMM_WORLD);
} else if (rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 42, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Rank 1: Received %d from rank 0\n", number);
}
```

MPI_Send

- ▶ `int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
 - ▶ `buf`: source buffer to send data from
 - ▶ `count`: number of data elements to send
 - ▶ `datatype`: type of data to send
 - ▶ `dest`: destination rank
 - ▶ `tag`: user-defined message type or category
 - ▶ `comm`: communicator

- ▶ `MPI_Send(&number, 1, MPI_INT, 1, 42, MPI_COMM_WORLD);`

MPI_Recv

- ▶ `int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)`
 - ▶ `buf`: destination buffer to save data to
 - ▶ `count`: number of data elements to receive
 - ▶ `datatype`: type of data to receive
 - ▶ `source`: source rank
 - ▶ `tag`: user-defined message type or category
 - ▶ `comm`: communicator
 - ▶ `status`: holds additional information (e.g. rank of sender or tag of message)
- ▶ `MPI_Recv(&number, 1, MPI_INT, 0, 42, MPI_COMM_WORLD, MPI_STATUS_IGNORE);`

Predefined MPI Constants

- ▶ datatypes

- ▶ MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_BYTE, ...

- ▶ wildcards & misc

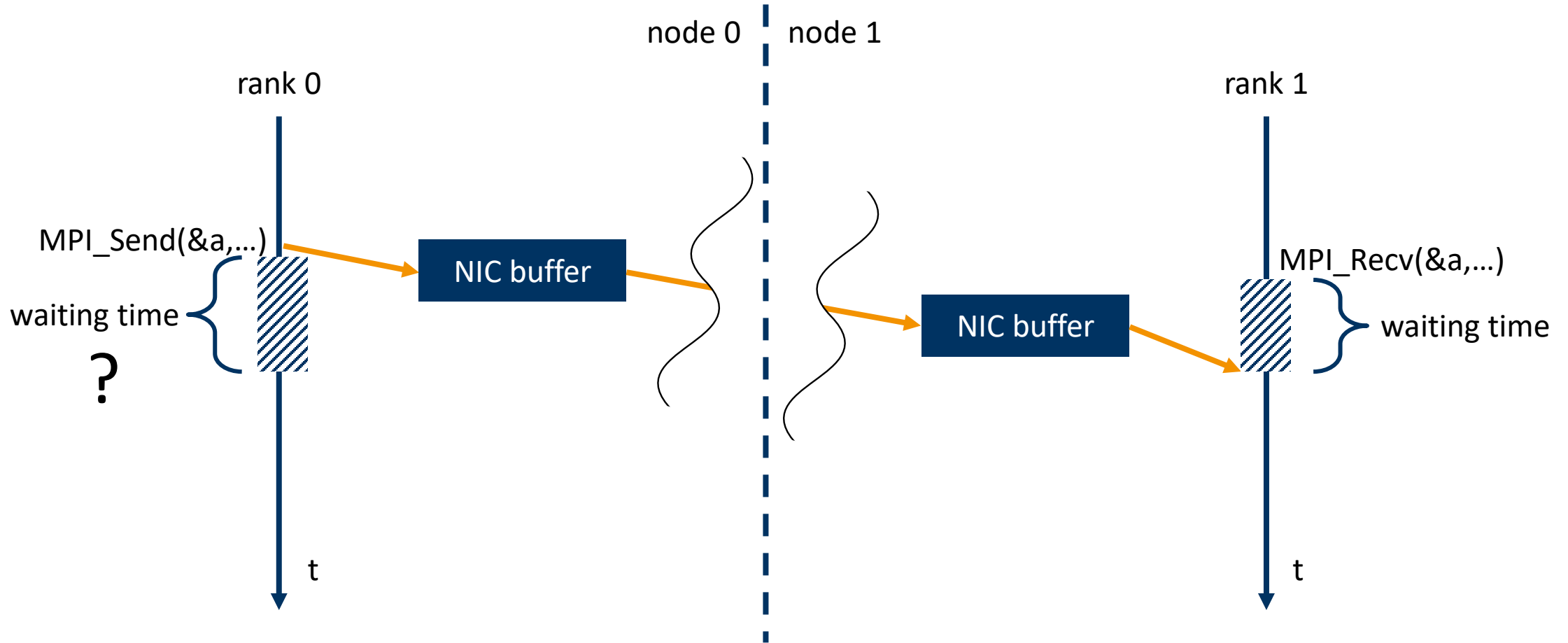
- ▶ MPI_ANY_SOURCE
- ▶ MPI_ANY_TAG
- ▶ MPI_COMM_WORLD
- ▶ MPI_STATUS_IGNORE
- ▶ ...

(Non-)Blocking and (A)Synchronous Communication

- ▶ distinguish two important properties
 - ▶ When does the MPI function call return?
 - ▶ Can I overwrite the send buffer?
 - ▶ When is all the data in the receive buffer?
 - ▶ When does the corresponding message transfer happen?
 - ▶ Do I need to wait for the receiver to get the entire message?
 - ▶ Do I need to wait for the receiver to begin receiving?

```
if (rank == 0) {  
    MPI_Send(&number, ...);  
} else if (rank == 1) {  
    MPI_Recv(&number, ...);  
}
```

(Non-)Blocking and (A)Synchronous Communication cont'd



Blocking vs. Non-Blocking Communication

- ▶ blocking point-to-point:
MPI_Send() and MPI_Recv()
 - ▶ allows to re-use send buffer after send call returns
 - ▶ allows to read receive buffer after receive call returns

```
if (rank == 0) {  
    MPI_Send(&number, ...);  
    // re-use number here  
} else if (rank == 1) {  
    MPI_Recv(&number, ...);  
    // use number here  
}
```

Blocking vs. Non-Blocking Communication cont'd

- ▶ non-blocking point-to-point:
MPI_Isend() and MPI_Irecv()
 - ▶ send and receive return almost immediately
 - ▶ MPI_Wait() calls block until buffers can be read/re-used

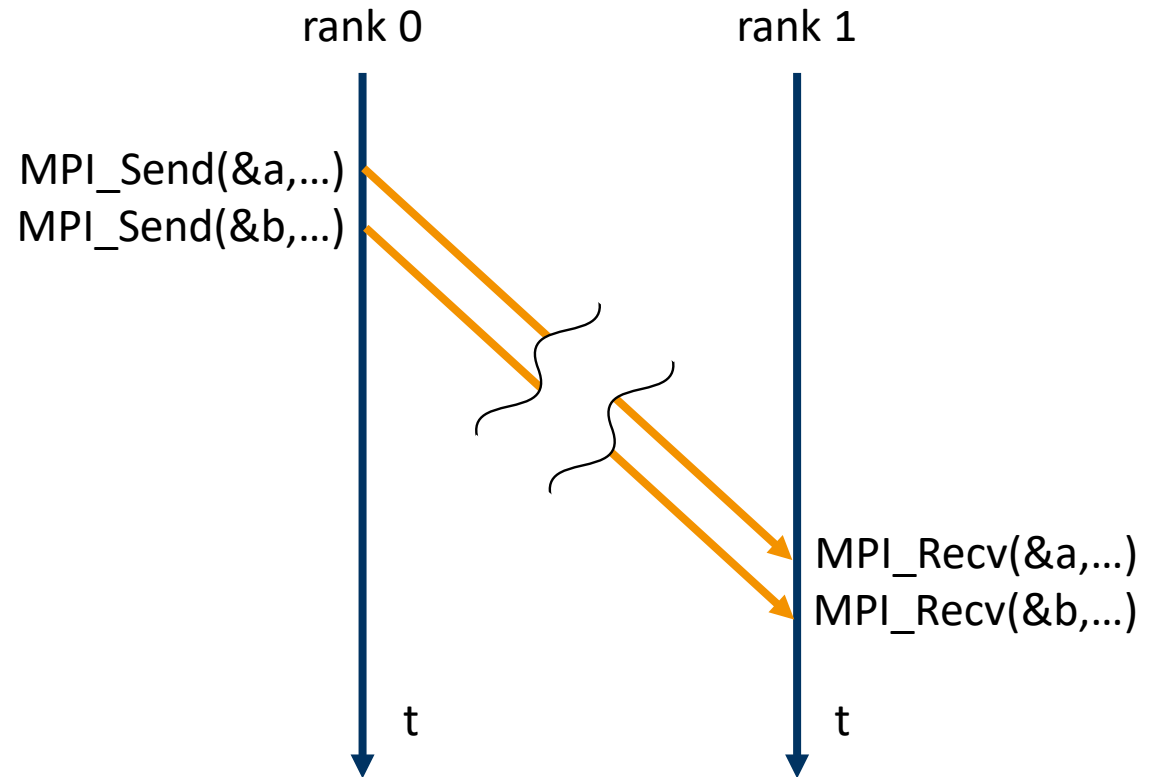
```
MPI_Request request;
if (rank == 0) {
    MPI_Isend(&number, ..., &request);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    // re-use number here
} else if (rank == 1) {
    MPI_Irecv(&number, ..., &request);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    // re-use number here
}
```

(A)Synchronous Send Modes

- ▶ `MPI_Ssend()` – synchronous mode
 - ▶ will wait for matching receive
 - ▶ `MPI_Bsend()` – buffered mode
 - ▶ will buffer, won't wait for a matching receive
 - ▶ `MPI_Rsend()` – ready mode
 - ▶ requires an already posted, matching receive (developer responsibility!)
 - ▶ `MPI_Send()` – standard mode
 - ▶ may buffer
 - ▶ may or may not wait for matching receive
- ▶ and there are also non-blocking variants for ALL of them...

Message Order Preservation

- ▶ messages do NOT overtake each other if
 - ▶ same communicator
 - ▶ same source rank
 - ▶ same destination rank
- ▶ regardless of blocking or synchronization mode
- ▶ mandated by MPI specification





Collective Communication

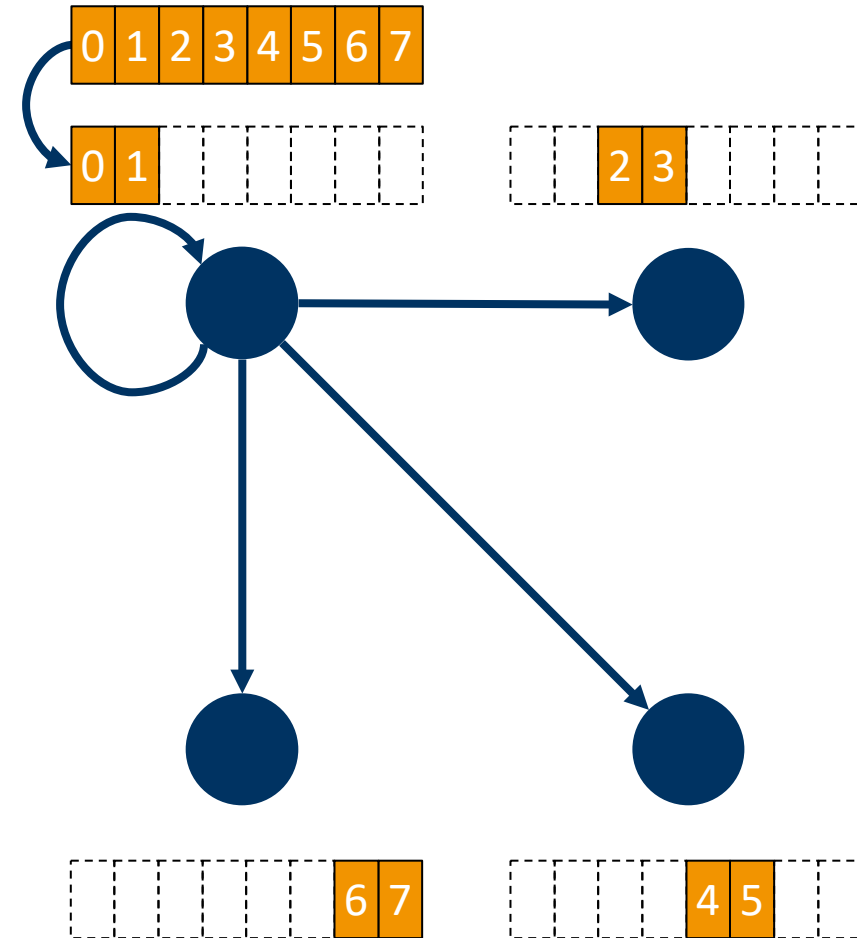


Collective Communication

- ▶ convenience function for frequently-used programming patterns (e.g. distributing data)
 - ▶ can involve several ranks at the same time, not just 2
 - ▶ must be called by ALL ranks in the communicator
 - ▶ must be called in-order by all ranks (no interleaving of multiple collective communication calls!)
 - ▶ is locally finished when local operation has finished
 - ▶ is globally finished when all participating ranks are finished
 - ▶ available as blocking and non-blocking variants (but cannot be mixed!)

MPI_Scatter/MPI_Scatterv

- ▶ sends chunks of data to multiple ranks, including root itself
 - ▶ not necessarily the best
- ▶ MPI_Scatterv() allows varying counts of elements to be distributed to each rank



MPI_Scatter

- ▶ `int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
 - ▶ `sendbuf`: source buffer to send data from
 - ▶ `sendcount`: number of data elements to send to each rank
 - ▶ `sendtype`: type of data to send
 - ▶ `recvbuf`: destination buffer to save data to
 - ▶ `recvcount`: number of data elements to receive at each rank
 - ▶ `recvtype`: type of data to receive
 - ▶ `root`: rank of the sender
 - ▶ `comm`: communicator

Scatter Example

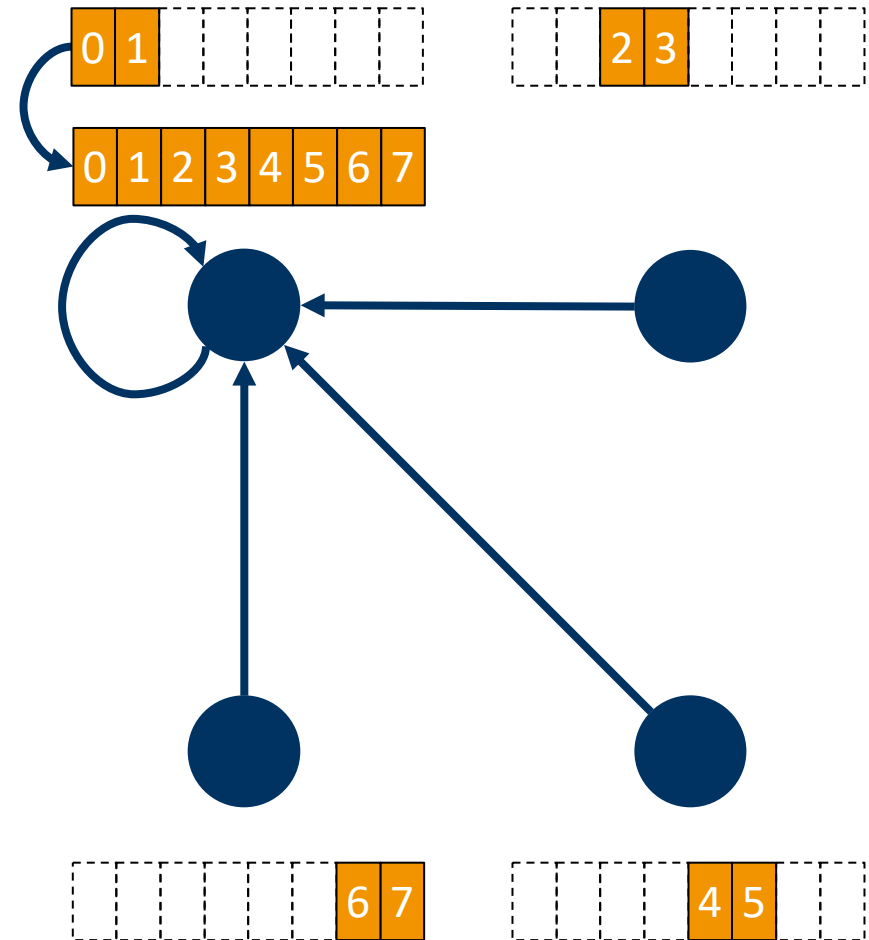
```
int globaldata[4];
int localdata;

if(rank==0) {
    for(int i = 0; i < 4; i++) {
        globaldata[i] = ...
    }
}

MPI_Scatter(globaldata, 1, MPI_INT, &localdata, 1, MPI_INT, 0,
            MPI_COMM_WORLD);
```

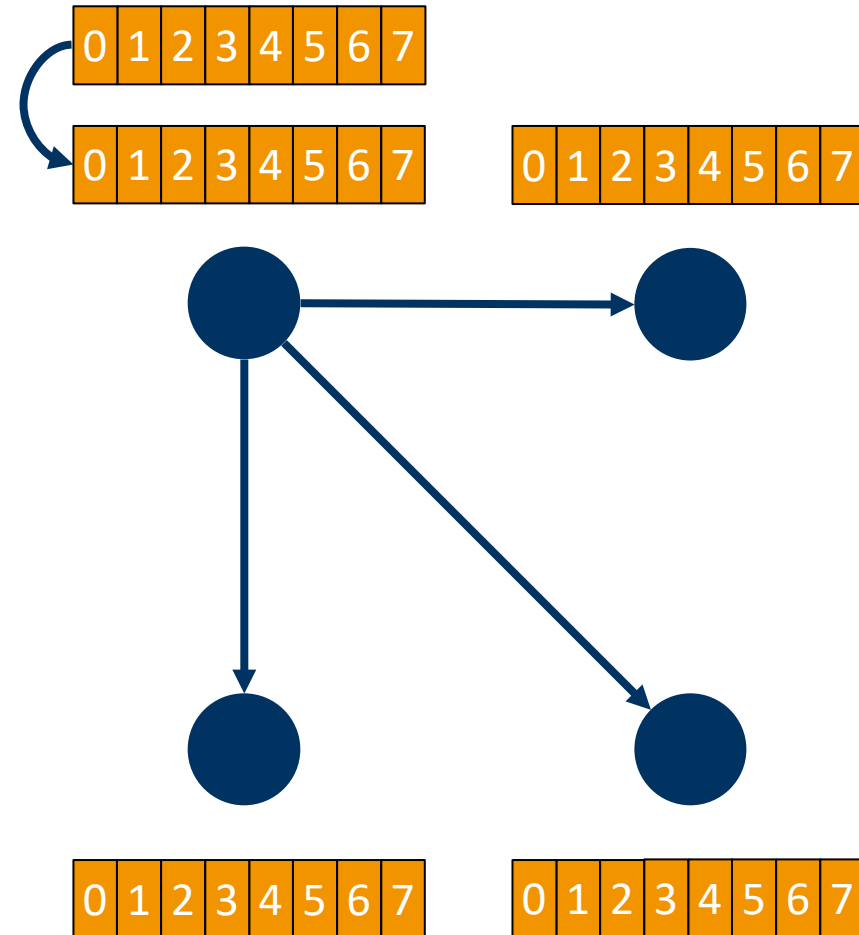
MPI_Gather/MPI_Gatherv

- ▶ sends chunks of data from multiple ranks, including root itself, to root
- ▶ simple way of collecting data
 - ▶ not necessarily the best
- ▶ MPI_Gatherv() allows varying counts of elements to be collected from each rank



MPI_Bcast

- ▶ broadcast operation
- ▶ sends copies of data to multiple ranks

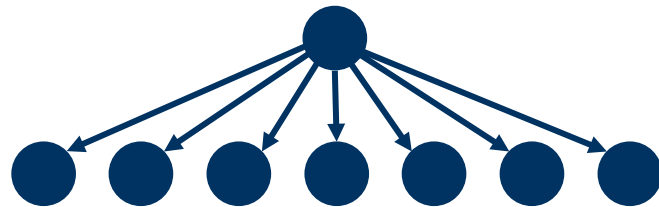


Emulating Broadcast with Point-to-Point?

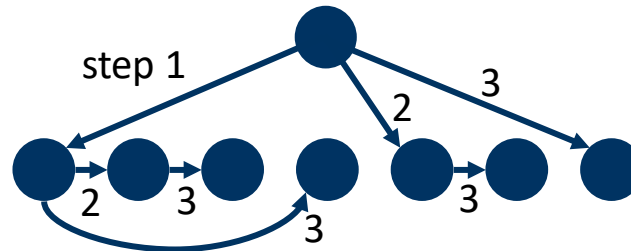
```
MPI_Bcast(&buf, 1, MPI_INT, 0, MPI_COMM_WORLD);  
// ##### OR, instead: #####  
if (rank == 0) {  
    for (int i = 0; i < size; i++) {  
        if (i != rank) {  
            MPI_Send(&buf, 1, MPI_INT, i, 0, MPI_COMM_WORLD);  
        }  
    }  
} else {  
    MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
}
```

Collective Communication Patterns

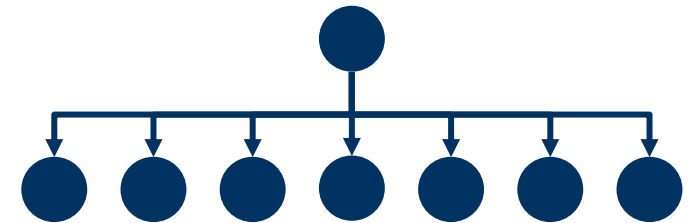
- ▶ chosen automatically at runtime by MPI implementation
- ▶ can depend on multiple parameters, including
 - ▶ type of operation (e.g. broadcast)
 - ▶ number and location of ranks
 - ▶ size and structure of data
 - ▶ hardware capabilities



sequential algorithm
 $O(\text{num_ranks})$



tree-based algorithm
 $O(\log_2(\text{num_ranks}))$



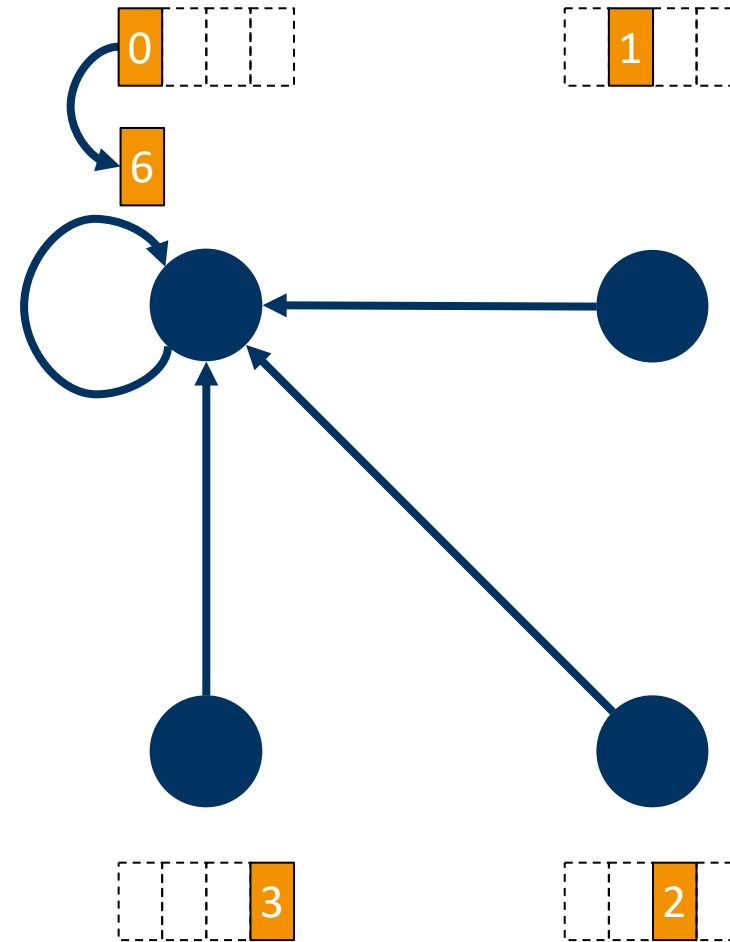
hardware operation
 $O(1)$

Barrier

- ▶ `int MPI_Barrier(MPI_Comm comm)`
 - ▶ `comm`: communicator
- ▶ causes all ranks to wait until everyone reached the barrier
 - ▶ normally not needed: explicit data communication inherently also synchronizes
 - ▶ often used for debugging and profiling
 - ▶ Don't forget to remove in production/release!

MPI_Reduce

- ▶ aggregate data from multiple ranks, including root itself, to root
 - ▶ e.g. MPI_SUM
- ▶ requires associative reduction operation \circ such that e.g.
$$(x_0 \circ x_1) \circ (x_2 \circ x_3) = ((x_0 \circ x_1) \circ x_2) \circ x_3$$
- ▶ be careful with floating point types!



MPI_Reduce cont'd

- ▶ `int MPI_Reduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
 - ▶ `sendbuf`: source buffer to reduce data from
 - ▶ `recvbuf`: destination buffer to reduce data into
 - ▶ `count`: number of data elements in source and destination buffers
 - ▶ `datatype`: type of data to reduce
 - ▶ `op`: reduction operation
 - ▶ `root`: rank of the sender
 - ▶ `comm`: communicator

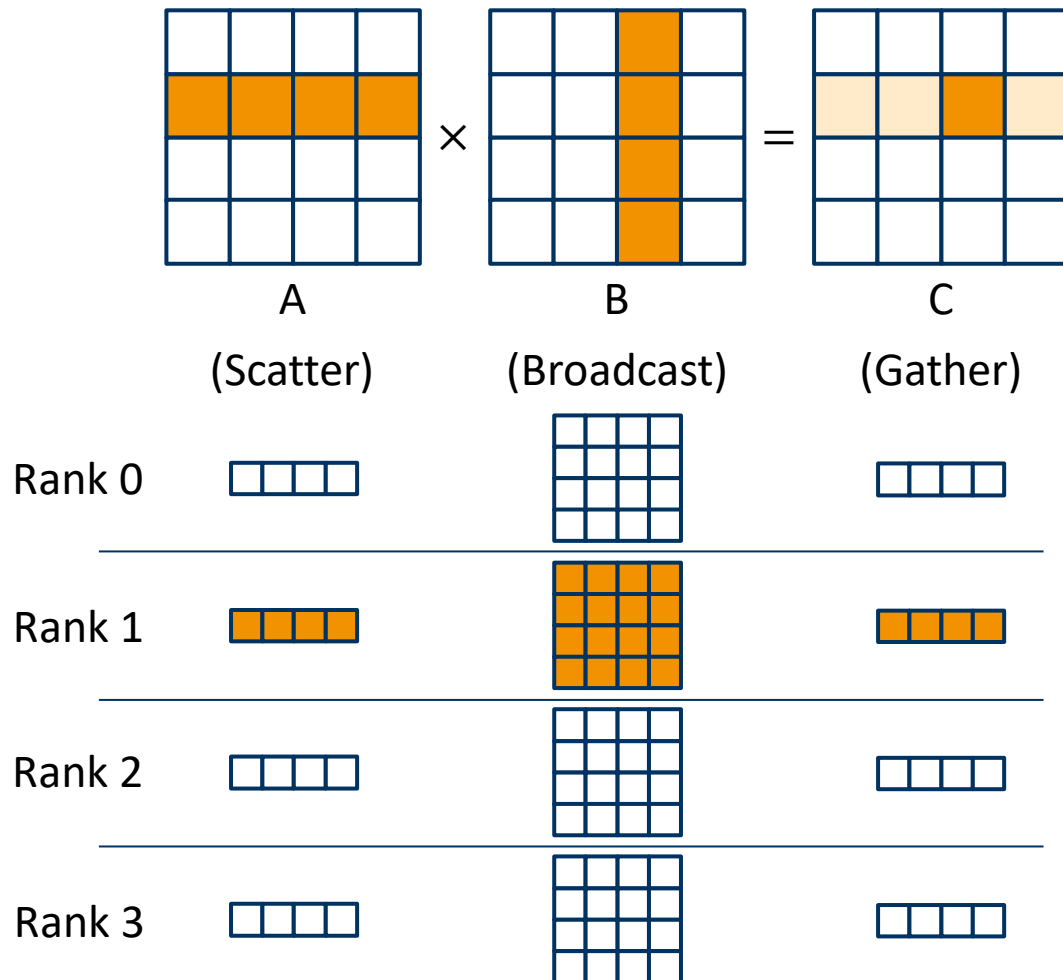
Available Reduction Operations

- ▶ several pre-defined
 - ▶ MPI_MAX, MPI_MIN
 - ▶ MPI_SUM, MPI_PROD
 - ▶ MPI_LAND, MPI_LOR, MPI_LXOR
 - ▶ MPI_BAND, MPI_BOR, MPI_BXOR
 - ▶ MPI_MAXLOC, MPI_MINLOC
- ▶ also user-defined ops are possible
 - ▶ must be associative
 - ▶ requires a specific function signature
 - ▶ requires to register an MPI handle

Additonal MPI Functions

- ▶ **MPI_Wtime**
 - ▶ returns time in seconds since an arbitrary time in the past (Wall clock time)
- ▶ **MPI_Sendrecv**
 - ▶ convenience wrapper for blocking send and receive
- ▶ **MPI_Allreduce/MPI_Allgather/...**
 - ▶ same as non-all versions, but result is available everywhere (performance impact!)
- ▶ **MPI_Scan/MPI_Exscan**
 - ▶ inclusive and exclusive prefix reductions
- ▶ **MPI_Wait/MPI_Test**
 - ▶ blocking/non-blocking check whether pending operation completed
- ▶ **MPI_Probe/MPI_Iprobe**
 - ▶ blocking/non-blocking check for new message without actually receiving it

Example Code: Naïve Matrix Multiplication



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

```
#define SIZE 4
```

```
int A[SIZE][SIZE];
int B[SIZE][SIZE];
int C[SIZE][SIZE];
```

```
void fill_matrix(int m[SIZE][SIZE]);
void print_matrix(int m[SIZE][SIZE]);
```

Example Code: Naïve Matrix Multiplication cont'd

```
int myrank, numProcs;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
// if matrix size not divisible
if(SIZE % numProcs != 0) {
    MPI_Finalize();
    return EXIT_FAILURE;
}
// root generates input data
if(myrank == 0) {
    fill_matrix(A);
    fill_matrix(B);
}
```

```
// compute boundaries of local computation
int start = myrank*SIZE/numProcs;
int end = (myrank+1)*SIZE/numProcs;
// distribute rows of A to everyone
MPI_Scatter(A, SIZE*SIZE/numProcs, MPI_INT,
    A[start], SIZE*SIZE/numProcs, MPI_INT, 0,
    MPI_COMM_WORLD);
// send entire matrix B to everyone
MPI_Bcast(B, SIZE*SIZE, MPI_INT, 0,
    MPI_COMM_WORLD);
```

Example Code: Naïve Matrix Multiplication cont'd

```
// local computation of every rank
for(int i = start; i < end; i++) {
    for(int j = 0; j < SIZE; j++) {
        C[i][j] = 0;
        for(int k = 0; k < SIZE; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
// gather result rows back to root
MPI_Gather(C[start], SIZE*SIZE/numProcs,
          MPI_INT, C, SIZE*SIZE/numProcs, MPI_INT, 0,
          MPI_COMM_WORLD);

if(myrank == 0) { print_matrix(C); }

MPI_Finalize();

return EXIT_SUCCESS;
```

Submitting to a Cluster (SLURM & SGE)

```
#!/bin/bash

# submission partition
#SBATCH -p std.q
# name of the job
#SBATCH --job-name my_test_job
# redirect output stream to this file
#SBATCH -o output.dat
# specify parallel environment
#SBATCH -N 8
#SBATCH --ntasks-per-node 2

srun /path/to/application
# or
mpiexec -n $SLURM_NTASKS /path/to/application
```

```
#!/bin/bash

# submission queue
#$ -q std.q
# change to current directory
#$ -cwd
# name of the job
#$ -N my_test_job
# redirect output stream to this file
#$ -o output.dat
# join the output and error stream
#$ -j yes
# specify parallel environment
#$ -pe openmpi-2perhost 8

mpiexec -n 8 /path/to/application
```

Summary

- ▶ general concepts about MPI
 - ▶ characteristics
 - ▶ program model
 - ▶ startup
- ▶ point-to-point communication
- ▶ collective communication
- ▶ practical example (matrix multiplication)