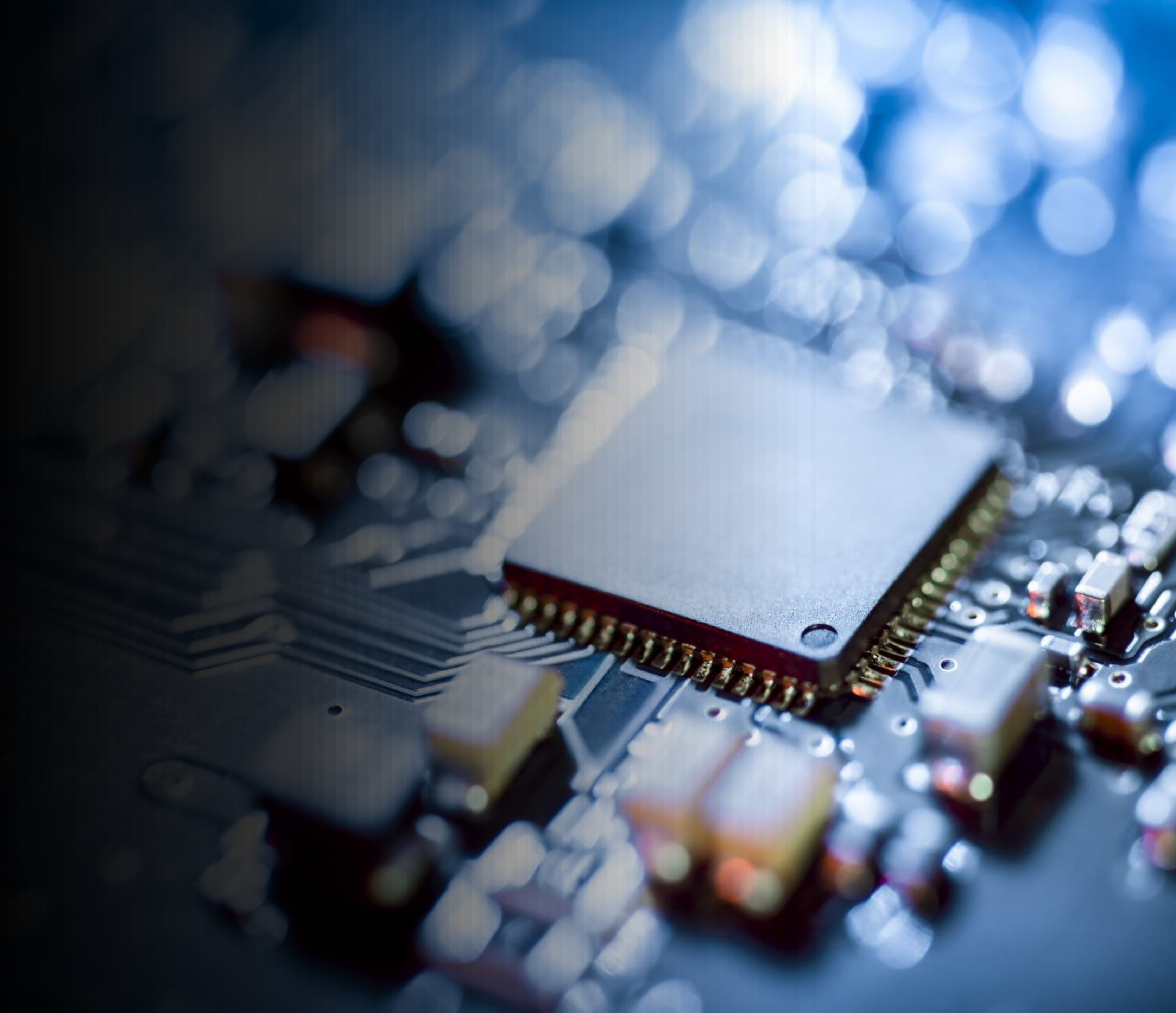
MPI
Message Passing Interface

From the Transistor to the Cluster
DERFAST - D.4.

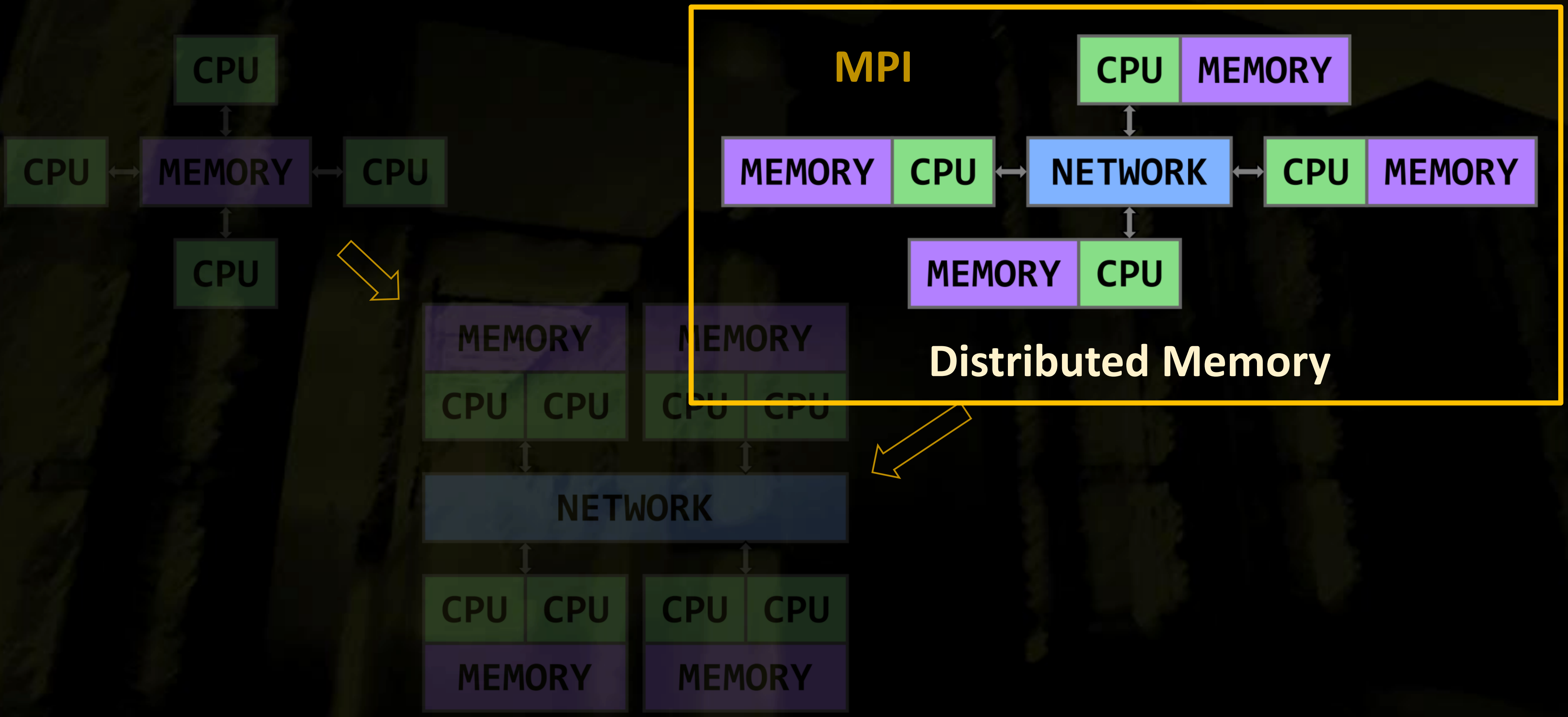
By Diego Roa

Overview

- MPI Basics
- MPI Collectives
- Blocking / Non-blocking
- Stencil Example



Programming Models



MPI (Message Passing Interface)

MPI Standard

Defines the syntax and semantics of library routines to write portable message-passing programs in C, C++, and Fortran.

Each parallel process has its own local memory, and data must be explicitly shared by passing messages between processes

MPI Implementations

MPICH

IntelMPI

OpenMPI:

An open-source MPI implementation that is developed and maintained by a consortium of academic, research, and industry partners

MPI Basics

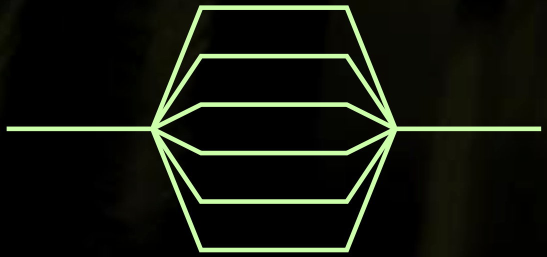
MPI_Init:

Initializes the MPI execution environment.

MPI_Finalize:

Terminates the MPI execution environment.

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    printf("MPI env initialized.\n");

    MPI_Finalize();
    printf("MPI env finalized.\n");
    return 0;
}
```



Initializes MPI processes in the hardware resources

Point-to-Point Communication

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    if (world_rank == 0) {
        int data = 100;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data %d to process 1\n", data);
    } else if (world_rank == 1) {
        int data;
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data %d from process 0\n", data);
    }

    MPI_Finalize();
    return 0;
}
```

MPI_Send

MPI_Recv

Point-to-Point Communication: MPI_Send

`MPI_Send(`	
`void* data,`	Memory location (pointer)
`int count,`	Number of elements
`MPI_Datatype datatype,`	Data type
`int destination,`	Destination Rank
`int tag,`	ID
`MPI_Comm communicator)`	Communicator

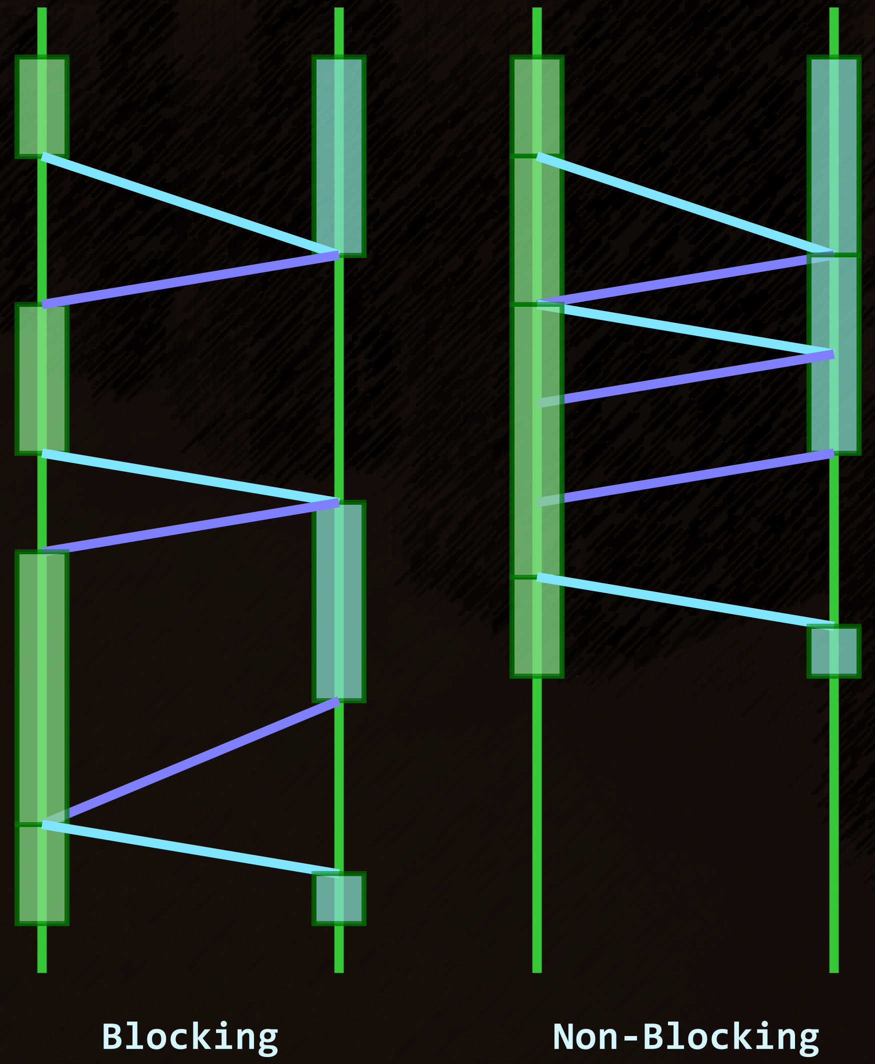
Sends a message to another process.

Point-to-Point Communication

`int MPI_Recv(`	
`void *buf,`	Memory location (pointer)
`int count,`	Number of elements
`MPI_Datatype datatype,`	Data type
`int source,`	Source Rank
`int tag,`	ID
`MPI_Comm comm,`	Communicator
`MPI_Status *status)`	Status

Receives a message from another process

Blocking vs Non-Blocking Communication



<u>Blocking:</u> involves processes halting their execution until the communication operation is complete.

Processes can spend significant time waiting for communication to complete.

<u>Non-blocking:</u> return immediately.

Buffering: data is kept until it is received.
Synchronization: when a send is completed.

Processes can continue while waiting for the operation to be completed

Non-Blocking Communication

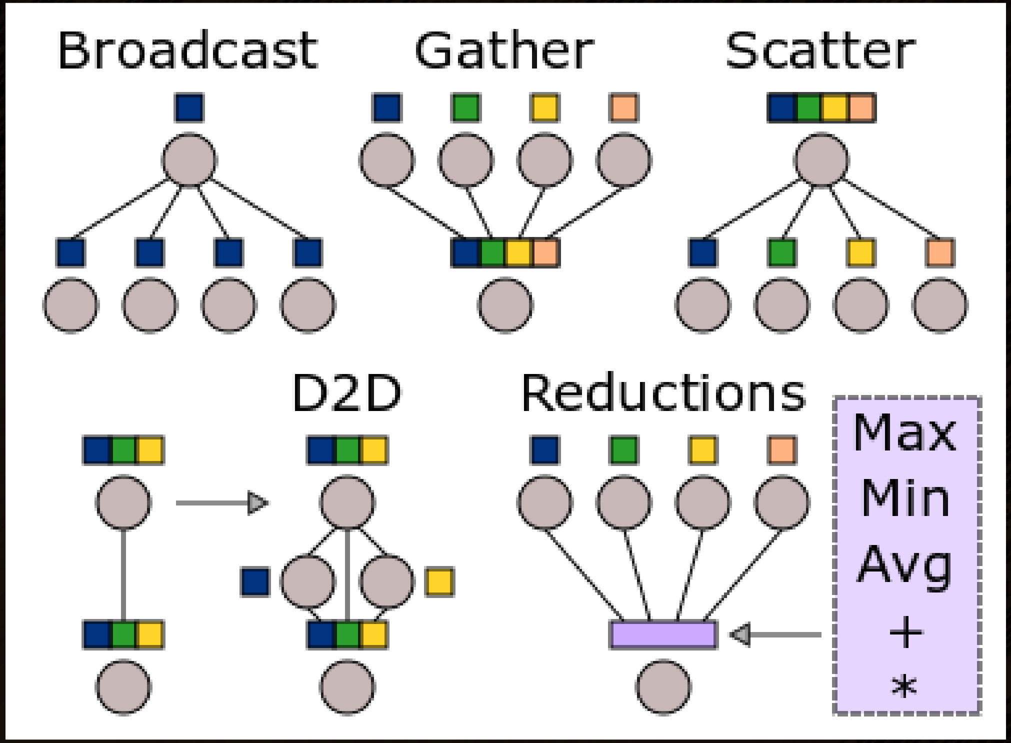
Requires Waiting or testing to ensure that the operation is completed

MPI_Send
MPI_Wait

MPI_Recv
MPI_Wait

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    int data;
    MPI_Request request;
    MPI_Status status;

    if (world_rank == 0) {
        data = 123;
        MPI_Isend(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
        printf("Process 0 initiated non-blocking send of data %d\n", data);
        // Perform some work while the send operation completes
        printf("Process 0 is doing other work while waiting for send to complete\n");
        MPI_Wait(&request, &status);  // Ensure the send operation is complete
    } else if (world_rank == 1) {
        MPI_Irecv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
        printf("Process 1 initiated non-blocking receive\n");
        // Perform some work while the receive operation completes
        printf("Process 1 is doing other work while waiting for receive to complete\n");
        MPI_Wait(&request, &status);  // Ensure the receive operation is complete
        printf("Process 1 received data %d\n", data);
    }

    MPI_Finalize();
    return 0;
}```
```

Collective Communication

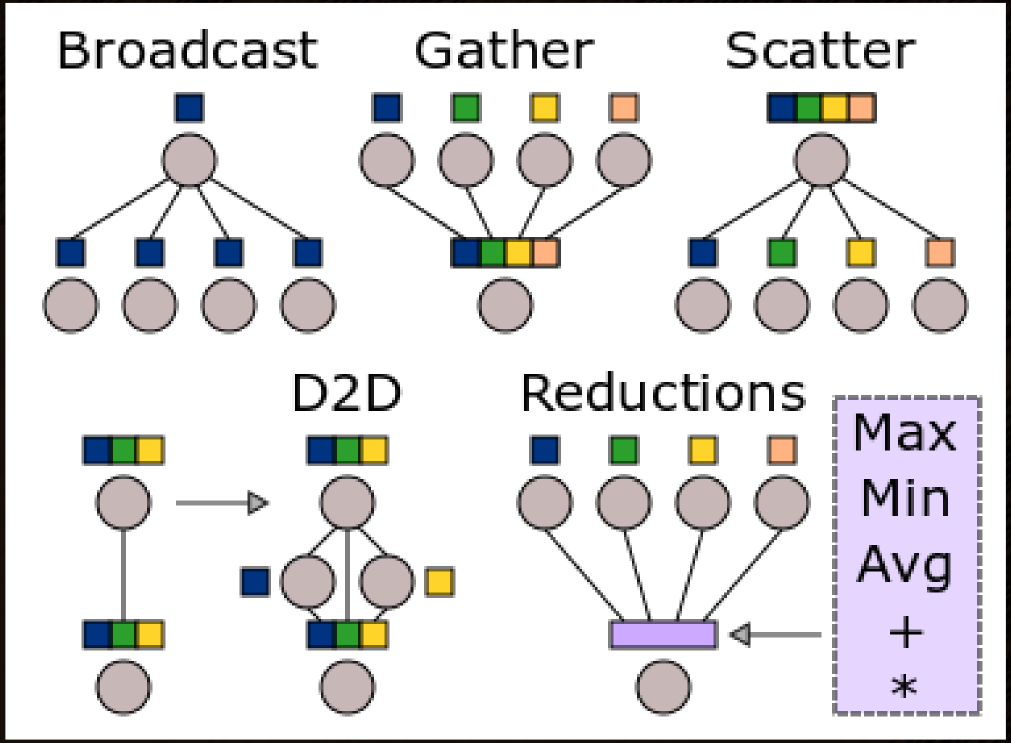


Collective operations:

Involve coordination and communication among multiple devices

- D2D (Device-to-Device)
- Broadcast
- Gather
- Scatter
- Reductions

Collective Operations



D2D (Device-to-Device): transfer data from one device's memory to another device's memory

Broadcast: Copies data from a source device to all other devices (or a subset of devices)

Gather: Consolidates data from multiple computing devices into a single location

Scatter: Distributes data from a single computing device to multiple destination devices

Reductions: Aggregate data from multiple computing devices into a device, performing an operation that consolidates the data. (sum, average, max, or min)

Collective Communication: Broadcast

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    int data = 0;
    if (world_rank == 0) {
        data = 100;
    }
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d received data %d\n", world_rank, data);

    MPI_Finalize();
    return 0;
}
```

MPI_Bcast(

void* data,

int count,

MPI_Datatype datatype,

int root,

MPI_Comm communicator)

Stencil Example

MPI_Scatter

MPI_Send
MPI_Recv

MPI_Gather

```c
int main(int argc, char *argv[]) {

    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (N % size != 0) {
        if (rank == 0) {
            fprintf(stderr, "The grid size N must be divisible by the number of processes.\n");
        }
        MPI_Finalize();
        return -1;
    }

    int rows_per_process = N / size;
    int start_row = rank * rows_per_process;
    int end_row = start_row + rows_per_process;

    double grid[N][N], new_grid[N][N];
    double local_grid[rows_per_process][N], local_new_grid[rows_per_process][N];

    if (rank == 0) {
        initialize_grid(grid);

        write_grid_to_file("initial_grid.txt", grid);
    }

    // Scatter the initial grid to all processes
    MPI_Scatter(grid, rows_per_process * N, MPI_DOUBLE, local_grid, rows_per_process * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    for (int iter = 0; iter < ITERATIONS; iter++) {
        stencil_step(local_grid, local_new_grid, 1, rows_per_process-1);

        // Copy new local grid to old local grid
        copy_grid(local_grid, local_new_grid, 1, rows_per_process-1);

        // Exchange boundary rows between neighboring processes
        if (rank > 0) {
            MPI_Send(local_grid[1], N, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD);
            MPI_Recv(local_grid[0], N, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        if (rank < size-1) {
            MPI_Send(local_grid[rows_per_process-2], N, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
            MPI_Recv(local_grid[rows_per_process-1], N, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }

    // Gather the final grid from all processes

    MPI_Gather(local_grid, rows_per_process * N, MPI_DOUBLE, grid, rows_per_process * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    if (rank == 0) {

        write_grid_to_file("final_grid.txt", grid);
    }


    MPI_Finalize();
    return 0;
}
```

Stencil Example

MPI_Scatter

MPI_Send
MPI_Recv

MPI_Gather

```
// Scatter the initial grid to all processes
MPI_Scatter(grid, rows_per_process * N, MPI_DOUBLE, local_grid, rows_per_process * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

for (int iter = 0; iter < ITERATIONS; iter++) {
    stencil_step(local_grid, local_new_grid, 1, rows_per_process-1);

    // Copy new local grid to old local grid
    copy_grid(local_grid, local_new_grid, 1, rows_per_process-1);

    // Exchange boundary rows between neighboring processes
    if (rank > 0) {
        MPI_Send(local_grid[1], N, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD);
        MPI_Recv(local_grid[0], N, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    if (rank < size-1) {
        MPI_Send(local_grid[rows_per_process-2], N, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
        MPI_Recv(local_grid[rows_per_process-1], N, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}

// Gather the final grid from all processes

MPI_Gather(local_grid, rows_per_process * N, MPI_DOUBLE, grid, rows_per_process * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

if (rank == 0) {

    write_grid_to_file("final_grid.txt", grid);
}
```