DISTRIBUTED MEMORY PROGRAMMING WITH MPI

Orian Louant

CÉCI/CISM training sessions

November 26 2025 - Louvain-La-Neuve

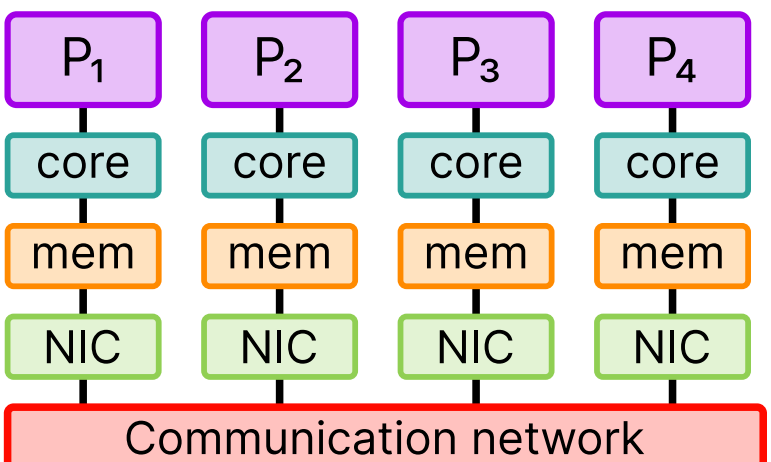
Distributed Memory

In a distributed memory architecture:

- Each processing element (process) has access only to its own local memory or address space
- There is no globally shared address space

Consequence:

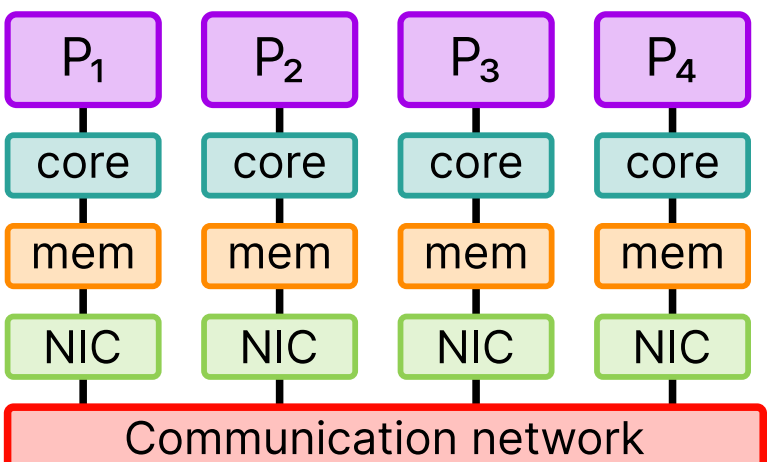
- Data exchange and communication occur explicitly through message passing over a communication network



Message Passing Library

The fact that data exchange and communication occur explicitly through message passing over a communication network creates the need for a **message-passing library** that is:

- Flexible, efficient and portable
- Capable of hiding low-level hardware and software communication details from the user



The Message Passing Interface

The Message Passing Interface (MPI) is a standard to enable portable, efficient, and scalable parallel programming, especially on distributed-memory systems:

- **Portability:** the same MPI-code should run on many platforms without changes
- **Efficiency and scalability:** minimize overhead, allow overlap of computation and communication, avoid unnecessary data copying
- **Flexibility:** support various programming models (point-to-point, collective, one-sided/remote memory access), ...
- **Standardization:** define syntax, semantics clearly; define bindings for multiple languages: C, Fortran, C++ (deprecated)

The History of MPI

MPI was not the first attempt at implementing a message-passing library. Earlier efforts included *Express* (ParaSoft), *P4* (Argonne), *PARMACS* (GMD), *PVM* (Oak Ridge), *NX/2* (Intel), and *Vertex* (Cornell)

- **Early 1990s:** Recognition that many incompatible message-passing systems existed and a standard was needed
- **April 1992:** Workshop on Standards for Message Passing in a Distributed Memory Environment launched the effort
- **November 1992:** First draft proposals (MPI-1) were put forward, followed by revisions in early 1993
- **November 1993:** Draft standard presented at Supercomputing 93[1]
- **June 1994:** MPI-1.0 officially released

[1] https://dl.acm.org/doi/pdf/10.1145/169627.169855

MPI today

As of today, MPI[2] is the most widely used programming paradigm for distributed-memory high-performance computing and continue to evolve[3]:

- The most recent officially approved version is MPI-4.1, approved in November 2023
- MPI-3 and 4 introduced improvements in collective operations and remote memory access
- MPI-5.0 has been approved as of June 5, 2025, bringing new features including a standard Application Binary Interface (ABI)

MPI is designed and updated to fully exploit the computational power of large-scale supercomputers, making it the **de facto standard** for scientific and engineering applications that require scalability and high performance.

---

[2] https://www.mpi-forum.org/docs/
[3] https://www.mpi-forum.org/implementation-status/

Standard *vs* Implementation

- MPI is a standard, not a specific software or implementation.
- It defines a specification: a set of rules, functions, and behaviors that ensure portability and interoperability across systems
- Multiple independent implementations of the MPI standard exist, developed by different organizations and optimized for various architectures
- Popular implementations include OpenMPI[4], (Cray) MPICH[5] and Intel MPI
- Programs written using the MPI standard can run on any compliant implementation without code changes, ensuring portability.

[4] https://www.open-mpi.org/
[5] https://www.mpich.org/

A FIRST MPI APPLICATION: COMPILING AND RUNNING

MPI Hello World (C/C++)


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

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

    int num_ranks;
    MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    printf("Hello world from processor %s,"
           "rank %d out of %d ranks\n",
           processor_name, rank, num_ranks);

    MPI_Finalize();
    return 0;
}
```

This simple Hello World program illustrates the fundamental structure of any MPI application:

- It starts by initializing the MPI environment, enabling all processes to run in parallel
- Each process queries the total number of processes and determines its own **rank** within the global **communicator**
- Every process then prints a message identifying itself
- Finally, the program finalizes the MPI environment for a clean shutdown

MPI Hello World (Fortran)


```
program hello_mpi
  use mpi_f08
  implicit none

  integer :: ierr, num_ranks, rank, name_len
  character(len=MPI_MAX_PROCESSOR_NAME) :: processor_name

  call MPI_Init(ierr)

  call MPI_Comm_size(MPI_COMM_WORLD, num_ranks, ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

  call MPI_Get_processor_name(processor_name, &
                              name_len, ierr)

  print *, 'Hello world from processor ',         &
           trim(processor_name), ', rank ', rank, &
           ' out of ', num_ranks, ' ranks.'

  call MPI_Finalize(ierr)
end program
```

This simple Hello World program illustrates the fundamental structure of any MPI application:

- It starts by initializing the MPI environment, enabling all processes to run in parallel
- Each process queries the total number of processes and determines its own **rank** within the global **communicator**
- Every process then prints a message identifying itself
- Finally, the program finalizes the MPI environment for a clean shutdown

MPI initialization

`MPI_Init` initializes the MPI execution environment:

- **Must be called before any other MPI routine**.
- In C/C++, the `argc` and `argv` arguments from `main` can be passed, though most implementations ignore them
- Returns an error code in C/C++. In Fortran, the error code is returned via the last argument

C/C++ Syntax

```c
#include <mpi.h>
int MPI_Init(int *argc, char ***argv)
```

Fortran Syntax

```fortran
USE MPI
MPI_INIT(IERROR)
INTEGER   IERROR
```

Fortran 2008 Syntax

```fortran
USE mpi_f08
MPI_Init(ierror)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI finalization

`MPI_Finalize` terminates MPI execution environment:

- Cleans up all MPI states
- **Once this routine is called, no MPI routine (not even `MPI_Init`) may be called**
- All pending communications involving a process need to be completed before the process calls `MPI_Finalize`

C/C++ Syntax

```
int MPI_Finalize()
```

Fortran Syntax

```
MPI_FINALIZE(IERROR)
INTEGER   IERRO
```

Fortran 2008 Syntax

```
MPI_Finalize(ierror)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Querying the number of processes/ranks

`MPI_Comm_size` returns the size of the group associated with a communicator:

- The number of processes involved in the communicator `comm` is returned via the `size` argument
- A communicator is an abstract type describing a group of processes
- If `MPI_COMM_WORLD`, a communicator available by default, is used for the `comm` argument it returns the total number of processes available

C/C++ Syntax

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Fortran Syntax

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
INTEGER COMM, SIZE, IERROR
```

Fortran 2008 Syntax

```
MPI_Comm_size(comm, size, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Querying the processes/ranks identifier

`MPI_Comm_rank` returns rank of the calling process in the communicator:

- The rank of the calling process within the communicator `comm` is returned in the `rank` argument
- An MPI process rank is an integer ranging from `0` to `NRANKS - 1`, where `NRANKS` is the total number of processes in the communicator.
- When `MPI_COMM_WORLD` is used as the communicator, the function returns the rank of the process among all processes

C/C++ Syntax

```
int MPI_Comm_rank(MPI_Comm comm, int
*rank)
```

Fortran Syntax

```
MPI_COMM_RANK(COMM, RANK, IERROR)
INTEGER COMM, RANK, IERROR
```

Fortran 2008 Syntax

```
MPI_Comm_rank(comm, rank, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: rank
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Querying the hardware name

`MPI_Get_processor_name` gets the name of the “processor”. Most of the time this is the name of the compute node

C/C++ Syntax

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

Fortran Syntax

```
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
CHARACTER*(*)  NAME
INTEGER        RESULTLEN, IERROR
```

Fortran 2008 Syntax

```
MPI_Get_processor_name(name, resultlen, ierror)
CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
INTEGER, INTENT(OUT) :: resultlen
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Compile an MPI application (OpenMPI)

All CÉCI clusters have an MPI implementation available:

- The most commonly used implementation is OpenMPI which is accessible by loading the corresponding environment module (`module load OpenMPI`)
- Compilation is done using the provided compiler wrappers

C	`mpicc [COMPILER_OPTIONS] [-o EXECUTABLE_NAME] SOURCES`
C++	`mpicxx [COMPILER_OPTIONS] [-o EXECUTABLE_NAME] SOURCES`
Fortran	`mpifort [COMPILER_OPTIONS] [-o EXECUTABLE_NAME] SOURCES`

Compiler wrappers are not compilers themselves. They add the necessary compiler and linker flags for Open MPI, then invoke the underlying compiler (GNU compilers) to perform the build (see output of `mpicc -show`)

Compile an MPI application (Intel MPI)

CÉCI clusters also offer the Intel MPI implementation which is accessible by loading the corresponding environment module (`module load impi`) and then use the wrappers:

Intel compilers:

C	`mpiicx [COMPILER_OPTIONS] [-o EXECUTABLE_NAME] SOURCES`
C++	`mpiicpx [COMPILER_OPTIONS] [-o EXECUTABLE_NAME] SOURCES`
Fortran	`mpiifx [COMPILER_OPTIONS] [-o EXECUTABLE_NAME] SOURCES`

GNU compilers:

C	`mpigcc [COMPILER_OPTIONS] [-o EXECUTABLE_NAME] SOURCES`
C++	`mpigxx [COMPILER_OPTIONS] [-o EXECUTABLE_NAME] SOURCES`
Fortran	`mpif90 [COMPILER_OPTIONS] [-o EXECUTABLE_NAME] SOURCES`

Run an MPI application (mpirun)

To start `NPROCESSES` copies of the `EXECUTABLE` application, you can use `mpirun`

```
mpirun -np NPROCESSES EXECUTABLE
```

- `mpirun` will usually automatically use the resource manager process starter (`srun` for SLURM) or use SSH/RSH if no resource manager is available
- In a SLURM job, mpirun automatically detects the number of processes to launch, so the `-np` option can be omitted. However, on a login node, omitting `-np` will cause MPI to use all available cores. **Therefore, always specify a value for `-np` when running on a login node**.
- You may sometimes see mpiexec used instead of mpirun. In most cases, the two commands are interchangeable. While mpiexec is the launcher recommended by the MPI standard, mpirun remains the most commonly used in practice.

Run an MPI application (srun)

To start `NPROCESSES` copies of the `EXECUTABLE` application with the builtin SLURM launcher, you can use `srun`

```
srun --ntasks=NPROCESSES EXECUTABLE
```

A minimal SLURM batch job to run an MPI application would look like this

```
#!/bin/bash
#SBATCH --ntasks=NPROCESSES
#SBATCH --time=TIME

module load OpenMPI # or impi
srun EXECUTABLE
```

or use `mpirun` as an alternative to `srun`

Common pitfalls with srun and OpenMPI

With *OpenMPI*, if you see an error message starting with this

```
The application appears to have been direct launched using "srun", but OMPI was
not built with SLURM's PMI support and therefore cannot execute.
```

then, you need to pass the `--mpi=pmix` option to `srun`

```
srun --mpi=pmix OTHER_OPTIONS EXECUTABLE
```

You can also make this option permanent by setting the value of `SLURM_MPI_TYPE` to `pmix` in your `bashrc`

```
echo 'export SLURM_MPI_TYPE=pmix' >> ~/.bashrc
```

Common pitfalls with srun and Intel MPI

With *Intel MPI*, if you see an error message looking like this

```
MPI startup(): PMI server not found. Please set I_MPI_PMI_LIBRARY variable if it
is not a singleton case.
```

then, you need to pass the `--mpi=pmi2` option to `srun` and set the value of the `I_MPI_PMI_LIBRARY` environment variable to point to the `pmi2` library

```
export I_MPI_PMI_LIBRARY="/usr/lib64/libpmi2.so"
srun --mpi=pmi2 OTHER_OPTIONS EXECUTABLE
```

Running the MPI hello world

First, on the login node, we compile the C version of the hello world with `mpicc`

```
module load OpenMPI
mpicc -o mpi_hello mpi_hello.c
```

Then, we prepare batch job by creating a file named `mpi_hello.job` with content

```
#!/bin/bash
#SBATCH --ntasks=4
#SBATCH --time=01:00
#SBATCH --output=mpi_hello.out

module load OpenMPI
srun ./mpi_hello
```

Finally, we submit the job using the SLURM `sbatch` command

```
sbatch mpi_hello.job
```

Running the MPI hello world

Once the job is finished, we can inspect the output:

```
$ cat mpi_hello.out
Hello world from processor nic5-w041, rank 2 out of 4 ranks
Hello world from processor nic5-w041, rank 3 out of 4 ranks
Hello world from processor nic5-w041, rank 1 out of 4 ranks
Hello world from processor nic5-w041, rank 0 out of 4 ranks
```

From the output we can see

- The four processes of our application ran on a single compute node `nic5-w041`. However, this is not always the case. SLURM may distribute processes across different nodes.
- The output shown is not sorted: rank 2 prints first. This illustrates a key aspect of **parallel execution**. Without explicit synchronization, processes run independently, and the order of operations is **nondeterministic**.

Playing with the SLURM options

SLURM provide options to control how your MPI job is distributed across the available compute nodes in the cluster by defining both the number of processes to launch and how these processes are mapped to the allocated nodes:

`--ntasks/-n NTASKS`	specifies the total number of MPI processes to start for the job. This is the main option for determining the overall level of parallelism
`--nodes/-N MIN_NODES`	requests that a minimum of `MIN_NODES` compute nodes be allocated. The actual number of nodes used depends on availability
`--ntasks-per-node NTASKS`	defines how many MPI processes to launch on each node. This setting helps control process placement by controlling the distribution of the ranks to the compute nodes

Early exit of MPI application

Calling `MPI_Abort` will terminates MPI execution environment:

- In theory, `MPI_Abort` is designed to abort all processes that belong to the specified communicator `comm`. This allows a controlled shutdown of a subset of processes if needed
- In practice, most MPI implementations interpret this call as a fatal error and terminate the entire MPI job, not just the communicator group
- After `MPI_Abort` is called, the MPI environment becomes invalid. No further MPI calls should be made

C/C++ Syntax

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

Fortran Syntax

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
INTEGER COMM, ERRORCODE, IERROR
```

Fortran 2008 Syntax

```
MPI_Comm_rank(comm, rank, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: errorcode
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Measuring wallclock time

`MPI_Wtime` returns an double precision value representing an elapsed time in seconds on the calling processor:

```
double start_time = MPI_Wtime();

//...  stuff to be timed  ...

double end_time = MPI_Wtime();

printf("That took %g seconds\n",
       end_time - start_time);
```

C/C++ Syntax

```
double MPI_Wtime()
```

Fortran Syntax

```
DOUBLE PRECISION MPI_WTIME()
```

Fortran 2008 Syntax

```
DOUBLE PRECISION MPI_Wtime()
```

WORK DISTRIBUTION AND POINT TO POINT COMMUNICATION

Computing π with trapezoidal rule (C)


```c
const uint64_t NUM_STEPS = 1000000000;

const double step = 1.0 / ((double)NUM_STEPS);

double sum = 0.0;
for (uint64_t i = 0; i < NUM_STEPS; i++)
{
    const double x = ((double)i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);
}

const double pi = step * sum;
```

As a first example on how to parallelize and distribute work with MPI, we will consider a simple application: computing the value of π using trapezoidal rule.

$$\int_0^1 \frac{1}{x^2+1} dx = \frac{\pi}{4}$$

To parallelize this code we need to

- distribute the work of the loop among the MPI processes
- communicate in order to compute the final sum

Computing π with trapezoidal rule (Fortran)


```
integer, parameter :: dp = kind(1.0d0)
integer(kind=8), parameter :: num_steps &
  = 1000000000_8

real(dp) :: step, x, sum, pi, elapsed
integer :: count_rate, count_start, count_end
integer(kind=8) :: istep

step = 1.0_dp / real(num_steps, dp)

sum = 0.0_dp
do istep = 0_8, num_steps - 1_8
    x = (real(istep, dp) + 0.5_dp) * step
    sum = sum + 4.0_dp / (1.0_dp + x * x)
end do

pi = step * sum
```

As a first example on how to parallelize and distribute work with MPI, we will consider a simple application: computing the value of π using trapezoidal rule.

$$\int_0^1 \frac{1}{x^2+1} dx = \frac{\pi}{4}$$

To parallelize this code we need to

- distribute the work of the loop among the MPI processes
- communicate in order to compute the final sum

Computing π: distributing the work

To distribute a loop across processes, each process determines which iterations belong to its rank based on the total number of processes. One simple approach is to rewrite the loop as:

```
for (uint64_t i = rank; i < NUM_STEPS; i += num_ranks)
```

Alternatively, the loop can be divided into contiguous chunks, where each process computes its own start and end indices based on its rank and the total number of ranks:

$$\text{start} = \frac{\text{NumSteps} \cdot \text{rank}}{\text{NumRanks}}$$

$$\text{end} = \frac{\text{NumSteps} \cdot (\text{rank} + 1)}{\text{NumRanks}} - 1$$

While the first approach is easier to implement, we will use the second strategy because it is more general and applies to a wider range of problems

Computing π: distributing the work (C implementation)


```c
int rank, num_ranks;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);

const uint64_t start = (NUM_STEPS * rank) / num_ranks;
const uint64_t end = (NUM_STEPS * (rank + 1UL)) / num_ranks;

double sum = 0.0;
for (uint64_t istep = start; istep < end; istep++)
{
    const double x = ((double)istep + 0.5) * step
    sum += 4.0 / (1.0 + x * x);
}
```

Computing π: distributing the work (Fortran implementation)


```
real(dp) :: step, x, sum, pi, elapsed
integer :: rank, num_ranks, ierr
integer(kind=8) :: istep, istart, iend

call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, num_ranks, ierr)

istart = (num_steps * rank) / num_ranks + 1_8
iend = (num_steps * (rank + 1_8)) / num_ranks

step = 1.0_dp / real(num_steps, dp)

sum = 0.0_dp
do istep = istart, iend
    x = (real(istep, dp) + 0.5_dp) * step
    sum = sum + 4.0_dp / (1.0_dp + x * x)
end do
```

Sending and receiving messages: requirements

Now that the work has been distributed among the processes, we still need a way to compute the final sum. To achieve this, the processes must communicate their partial results to one designated process, which will then combine these values to produce the final result. We need to send and receive messages, which means

- **A description of the data to send/receive**: the MPI library needs to know where the data to send/receive begins in memory and the size of this data (number of elements of a given datatype)
- **The rank of the sender/receiver:** MPI need to be able determine which process should receive the message and from which process
- **A way to identify the type of message:** to allow multiple independent communications between the same pair of processes without confusion, we need a way to uniquely identify each message.

Sending a message: syntax

C/C++ Syntax

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
     int tag, MPI_Comm comm)
```

Fortran Syntax

```
 MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
 <type>    BUF(*)
 INTEGER   COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Fortran 2008 Syntax

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Receiving a message: syntax

C/C++ Syntax

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
     int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Fortran Syntax

```
 MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
 <type>    BUF(*)
 INTEGER   COUNT, DATATYPE, SOURCE, TAG, COMM
 INTEGER   STATUS(MPI_STATUS_SIZE), IERROR
```

Fortran 2008 Syntax

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
TYPE(*), DIMENSION(..) :: buf
INTEGER, INTENT(IN) :: count, source, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Receiving a message: MPI_Status

- A call to `MPI_Recv` includes a `status` argument, which allows the system to return information about the received message
- In C, `MPI_Status` is a structure containing three fields: `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`
- In many cases, the status information is not needed; in such situations, a special constant can be passed instead: `MPI_STATUS_IGNORE`
- Inspecting the values in the status field is useful in situations where flexible receive operations are used. For example, when `MPI_ANY_SOURCE` is specified to accept a message from any process, or when `MPI_ANY_TAG` is used to accept a message with any tag value. In these cases, the status structure allows the receiving process to determine which process sent the message and which tag was used

Receiving a message: MPI_Status (Fortran)

Fortran

Status is an array of integers of size `MPI_STATUS_SIZE`. The constants `MPI_SOURCE`, `MPI_TAG` and `MPI_ERROR` are the indices of the entries that store the source, tag and error fields. Thus, `status(MPI_SOURCE)`, `status(MPI_TAG)` and `status(MPI_ERROR)` contain, respectively, the source, tag and error code of the received message

Fortran 2008

The derived type `TYPE(MPI_Status)` contains three public `INTEGER` fields named `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`. Thus, `status%MPI_SOURCE`, `status%MPI_TAG` and `status%MPI_ERROR` contain the source, tag, and error code of a received message respectively

MPI datatype to native type mapping

MPI datatype	C/C++ type	MPI datatype	C/C++ type
MPI_CHAR	char	MPI_UNSIGNED_CHAR	unsigned char
MPI_INT	int	MPI_UNSIGNED	unsigned int
MPI_LONG	long int	MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float	MPI_BYTE	unsigned char
MPI_DOUBLE	double		

MPI datatype	Fortran type	MPI datatype	Fortran type
MPI_INTEGER	integer	MPI_REAL4	real*4
MPI_REAL	real	MPI_REAL8	real*8
MPI_DOUBLE_PRECISION	double precision	MPI_INTEGER4	integer*4
MPI_COMPLEX	complex	MPI_INTEGER8	integer*8
MPI_LOGICAL	logical	MPI_DOUBLE_COMPLEX	double complex
MPI_CHARACTER	character(1)		

About the count argument of MPI_Recv


```c
if (rank == 0) {
    // Sender: send 4 integers
    int data[4] = {10, 20, 30, 40};
    MPI_Send(data, 4, MPI_INT, 1, tag, MPI_COMM_WORLD);
}
else if (rank == 1) {
  MPI_Status status;
  // Receiver: prepare a buffer for 10 integers
  int buffer[10];
  MPI_Recv(buffer, 10, MPI_INT, 0, tag,
           MPI_COMM_WORLD, &status);

  int count;
  MPI_Get_count(&status, MPI_INT, &count);

  printf("Process %d received %d elements:\n",
         rank, count);

  for (int i = 0; i < count; i++)
    printf(" buffer[%d] = %d\n", i, buffer[i]);
}
```

The `count` argument of `MPI_Recv` is the **maximum number** of elements to receive. The actual value of count on the sending size can be lower

The exact number of elements actually received can be obtained using the `MPI_Get_count` function

Get the number of elements received

`MPI_Get_count` stores in `count` the number of elements actually received by the `MPI_Recv` call that returned the given `status`. The `datatype` argument must match the datatype specified in the corresponding `MPI_Recv` call.

C/C++ Syntax

```
int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count)
```

Fortran Syntax

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
INTEGER   STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

Fortran 2008 Syntax

```
MPI_Get_count(status, datatype, count, ierror)
TYPE(MPI_Status), INTENT(IN) :: status
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(OUT) :: count
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Message matching rules

- **Communicator:** both the send and receive operations must be on the same communicator
- **Tag:** both the sender and receiver must use the same user-defined tag value. A receiver can use a wildcard tag (`MPI_ANY_TAG`) to match any tag
- **Source Rank:** the receiver receive operation must specify the rank of the sender, or use a wildcard to match any source rank (`MPI_ANY_SOURCE`)
- **Destination Rank:** the sender send operation must specify the rank of the destination process
- **Ordering:** MPI guarantees that if two messages are sent from the same sender to the same receiver and they both match the same receive, the messages will be received in the order they were sent. Similarly, if a receiver posts two receives in succession for the same message, the first posted receive will be matched first

Computing π: computing the final result

Now that we have discussed how to send and receive messages with MPI, we can compute the final result for our π calculation code:

- Each process with a rank greater than 0 sends its result to rank 0
- The process with rank 0 receives the results from all other ranks and combines them to compute the final value of π

⚠ We will see later that this is not the most efficient way of doing this

```
if (rank == 0) {
  double pi = step * sum;
  for (int srank = 1; srank < num_ranks; srank++) {
    double remote_sum;
    MPI_Recv(&remote_sum, 1, MPI_DOUBLE, srank, TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    pi += step * remote_sum;
  }

  printf(" Computed value of pi with %" PRIu64
         " steps is %.12lf\n", NUM_STEPS, pi);
  printf(" Computation took %lf seconds\n", elapsed);
}
else {
  MPI_Send(&sum, 1, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD);
}
```

IN-DEPTH LOOK AT POINT-TO-POINT COMMUNICATION

Communication performance: effective bandwidth

The total communication time T_{comm} for transferring a message of size N_{bytes} over a network can be modeled as:

$$T_{\text{comm}} = T_{\text{latency}} + \frac{N_{\text{bytes}}}{B_{\text{peak}}}$$

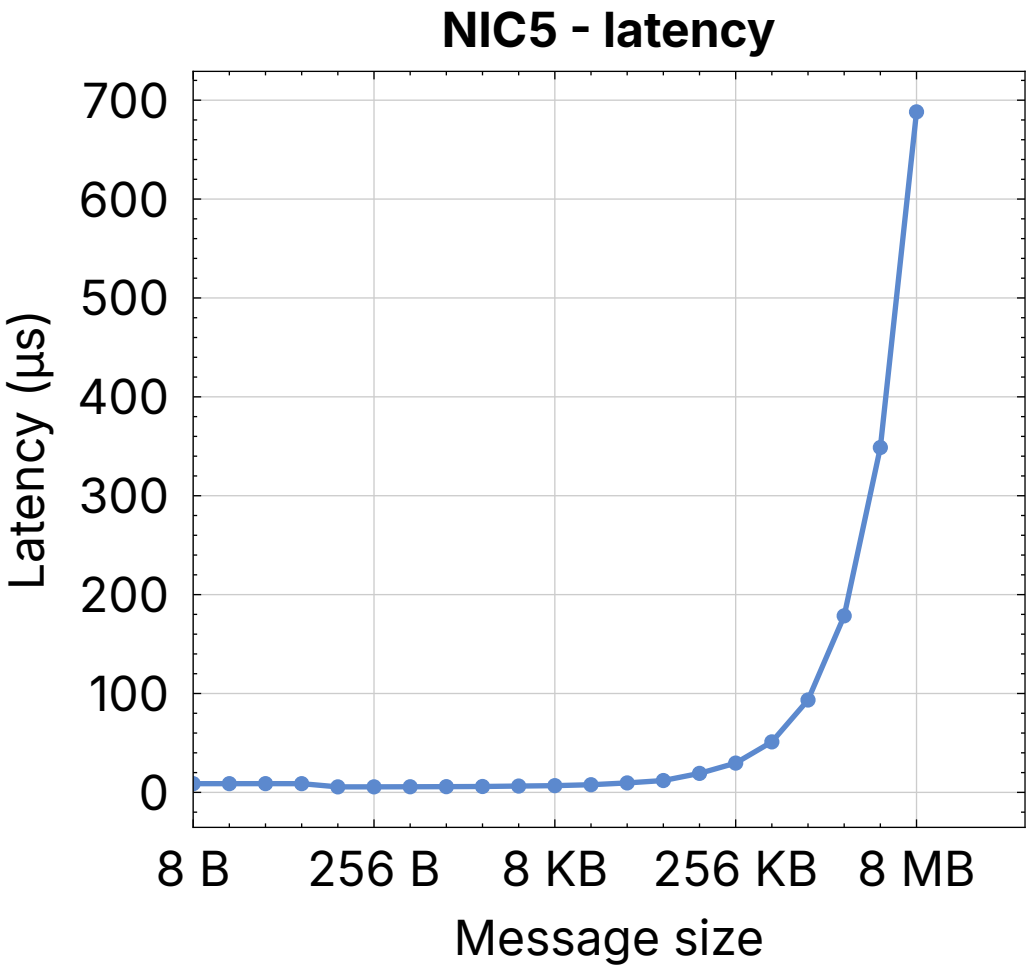
where:

- T_{latency} is the fixed communication latency, representing the time to initiate a transfer
- B_{peak} is the peak bandwidth of the network

The effective bandwidth $B_{\text{effective}}$ is then defined as the ratio of the message size to the total communication time:

$$B_{\text{effective}} = \frac{N_{\text{bytes}}}{T_{\text{latency}} + \frac{N_{\text{bytes}}}{B_{\text{peak}}}}$$

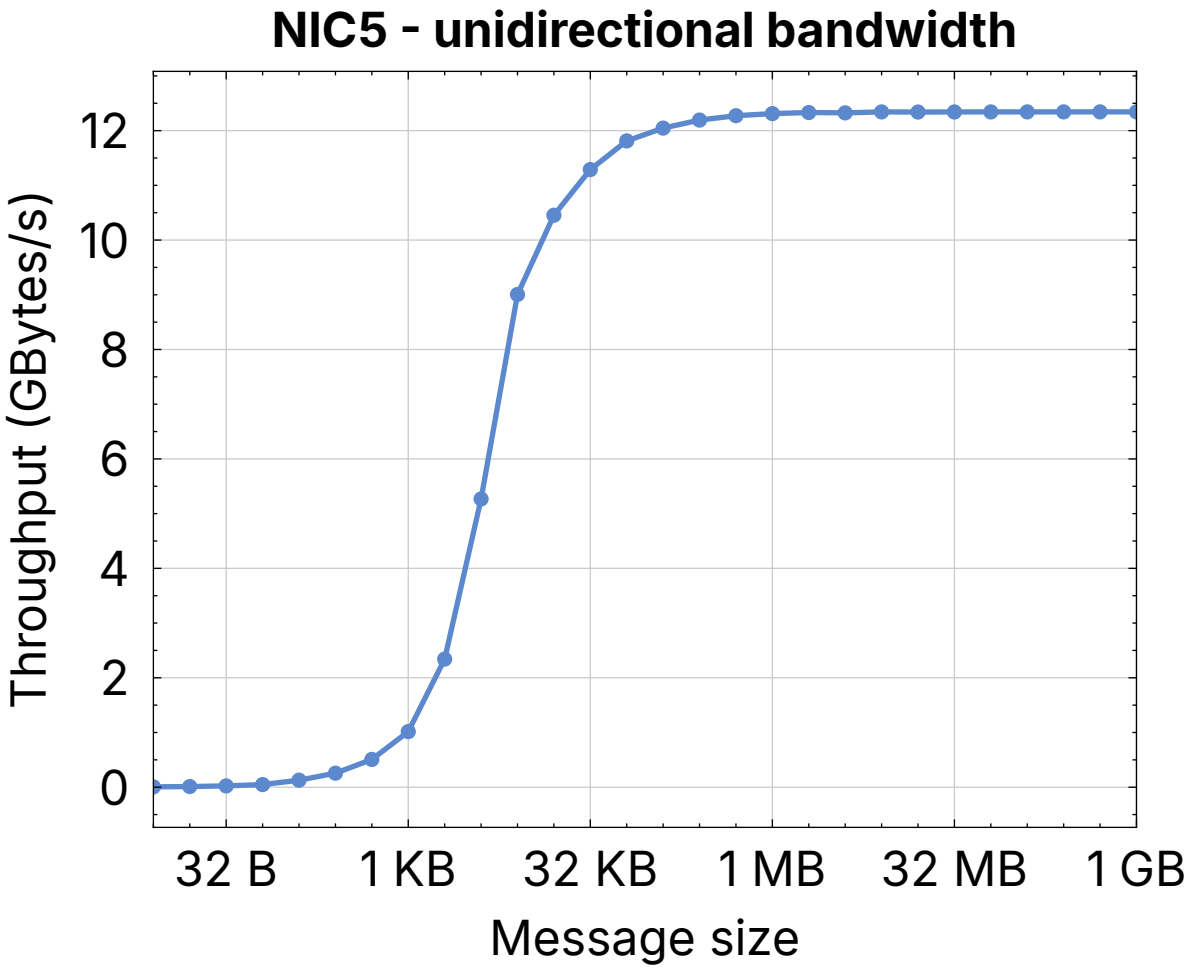
Communication performance: latency



When we measure the bidirectional bandwidth, Ddta flows in both directions simultaneously. The processes sends and receives data at the same time

- For small messages, the latency is nearly constant and very low. This region is latency-limited
- For larger messages, the latency grows sharply. This region is bandwidth-limited

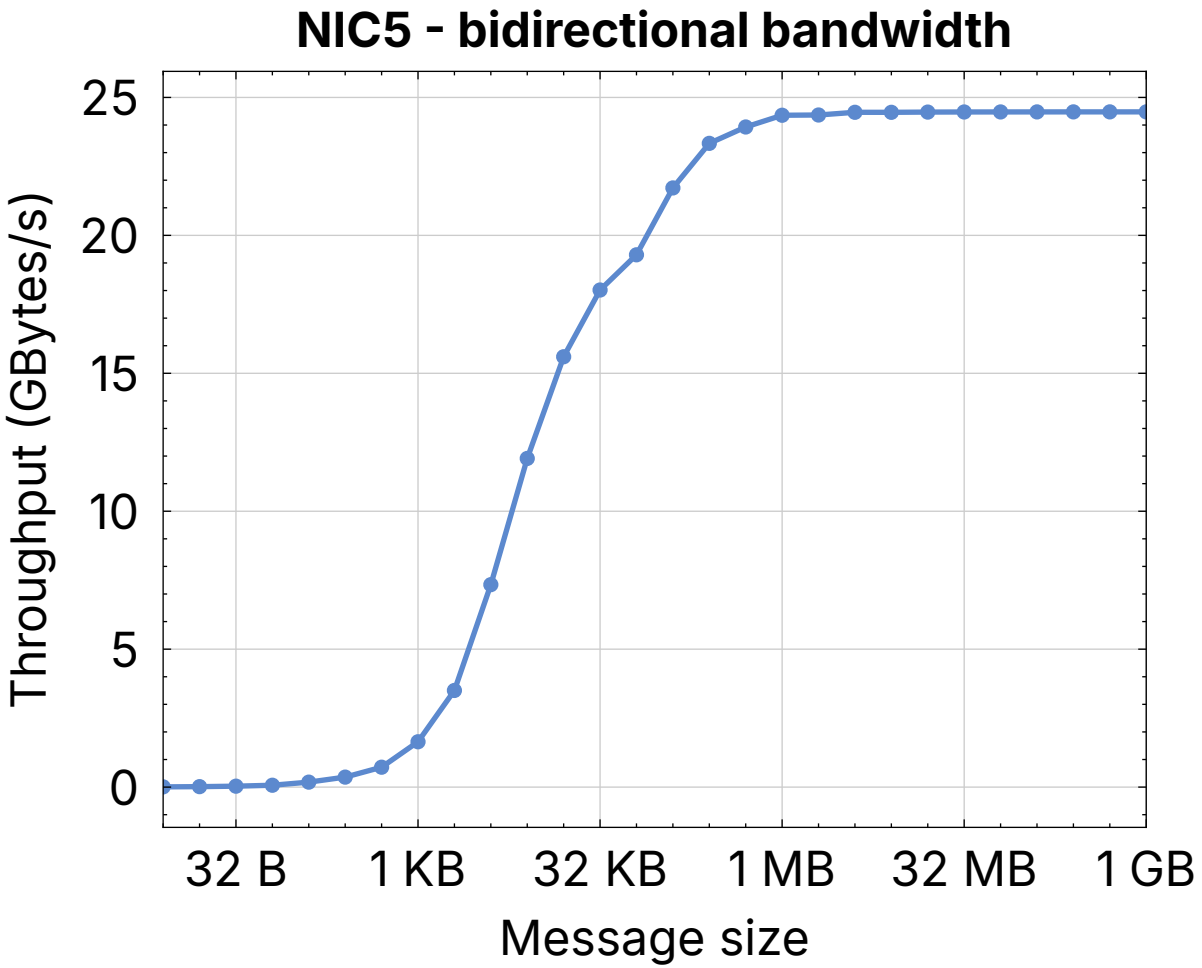
Communication performance: unidirectional bandwidth



When we measure the unidirectional bandwidth, data flows in one direction only: from sender to receiver

- With message size <1KB, the communication is dominated by the latency
- A transfer rate plateau close to the theoretical performance of the network is observed for message size >1MB

Communication performance: bidirectional bandwidth



When we measure the bidirectional bandwidth, data flows in both directions simultaneously. The processes sends and receives data at the same time.

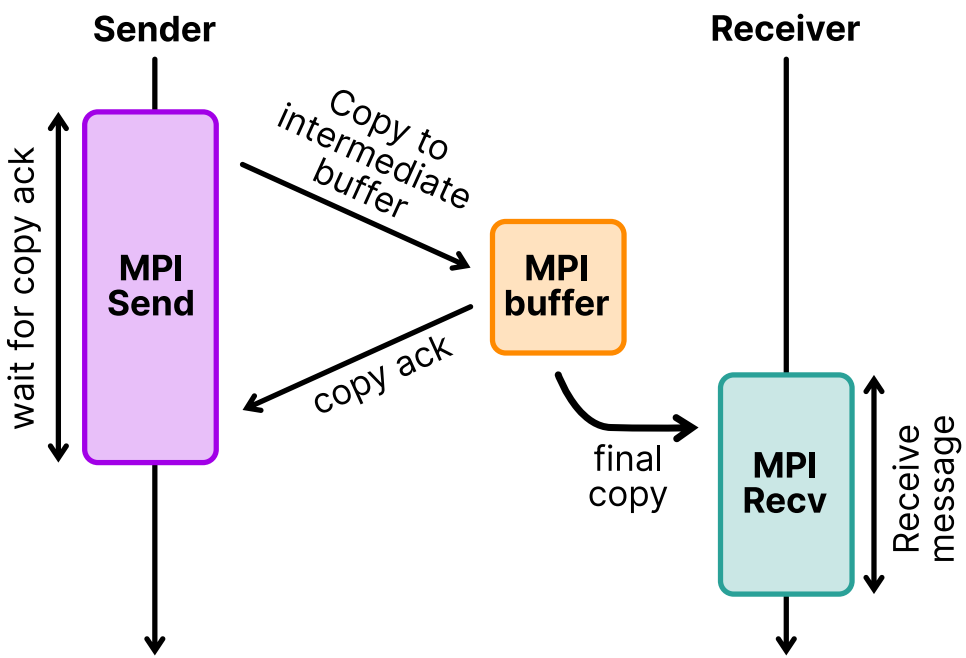
- With message size <1KB, the communication is dominated by the latency
- Transfer rate close to the theoretical performance of the network is observed for message size >1MB

Communication protocol: Eager

MPI defines multiple protocols for sending messages between processes: the Eager and Rendezvous protocols

The **Eager Protocol** is used for small messages to minimize latency:

- The sender immediately sends the message to the receiver without waiting for the receiver to post a matching receive.
- The message is placed directly into a pre-allocated buffer on the receiver's side.
- The receiver can later retrieve the data when the corresponding `MPI_Recv` is posted.

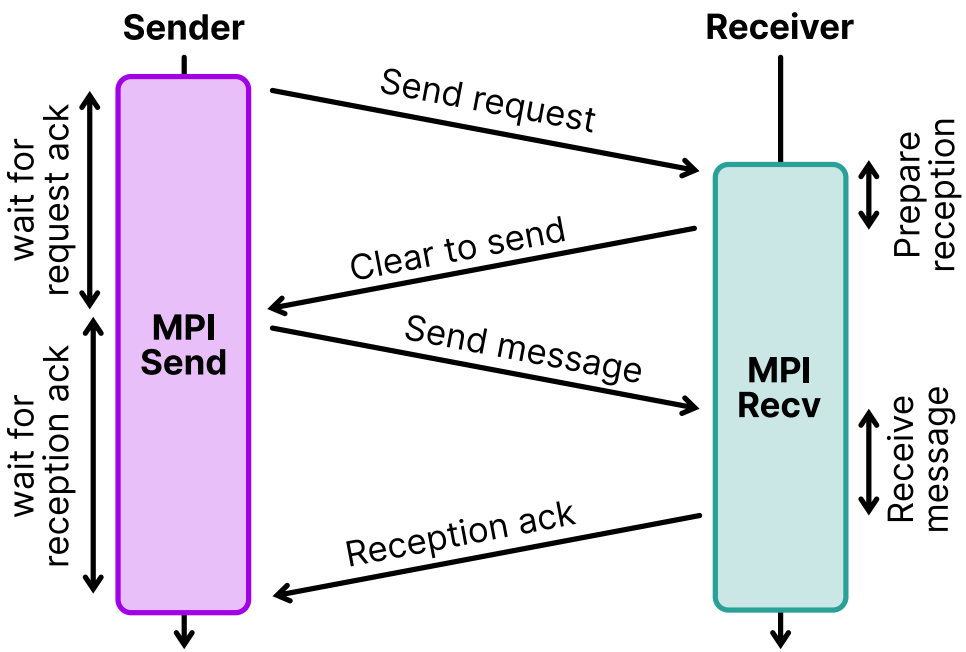


Communication protocol: Rendezvous

MPI defines multiple protocols for sending messages between processes: the Eager and Rendezvous protocols

The **Rendezvous Protocol** is used for large messages that cannot be buffered:

- The sender does not send the message immediately. Instead, the sender first sends a *Request to Send* (RTS) control message to the receiver.
- When the receiver has posted a matching `MPI_Recv`, it replies with a *Clear to Send* (CTS) acknowledgment.
- After receiving CTS, the sender transfers the actual message data directly to the receiver.



Communication protocol: MPI send return

When `MPI_Send` is called, it initiates sending a message from one process to another. However, the point at which `MPI_Send` returns (i.e., when control is given back to the program) does not necessarily mean the message has been received by the destination process

What it does mean depends on the send mode and underlying protocol used:

- **Eager protocol:** `MPI_Send` can return as soon as the message has been copied into the system buffer
- **Rendezvous protocol:** `MPI_Send` does not return until the receiver has posted a matching receive and the data transfer has completed

`MPI_Send` is a **blocking** send, meaning the function won't return until the send buffer can be **safely reuse**

Communication protocol: MPI receive return

A `MPI_Recv` is a blocking operation:

- The call does not return until the receive buffer has been completely filled with the incoming message, after which control is returned to the program
- This guarantees that it is safe to access or process the data in the receive buffer
- Unlike `MPI_Send`, whose behavior may vary depending on the communication protocol used, the receive operation always follows the same blocking semantics

Communication protocol: common pitfall

The use of different communication protocols by MPI implementations can lead to cases where an application runs successfully on small test problems but encounters deadlocks when scaled to larger systems.

```
for (size_t buffer_size = 8; buffer_size < 32768; buffer_size *= 2) {
  // ... allocate buffers ...
  if (rank == 0) {
    MPI_Send(send_buffer, buffer_size, MPI_DOUBLE, 1, FROM_ZERO_TAG, MPI_COMM_WORLD);
    MPI_Recv(recv_buffer, buffer_size, MPI_DOUBLE, 1, FROM_ONE_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    printf("Rank 0 - buffer size = %zu: OK\n", buffer_size);
  } else if (rank == 1) {
    MPI_Send(send_buffer, buffer_size, MPI_DOUBLE, 0, FROM_ONE_TAG, MPI_COMM_WORLD);
    MPI_Recv(recv_buffer, buffer_size, MPI_DOUBLE, 0, FROM_ZERO_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    printf("Rank 1 - buffer size = %zu: OK\n", buffer_size);
  }
}
```

Communication protocol: common pitfall

Let's consider a first scenario where the **Eager protocol** is used. In that case the application will be able to proceed and terminate successfully.

```
if (rank == 0) {
  // Copy to temporary buffer and returns
  MPI_Send(send_buffer, buffer_size, MPI_DOUBLE, 1, FROM_ZERO_TAG, MPI_COMM_WORLD);
  // Able to receive message from rank 1
  MPI_Recv(recv_buffer, buffer_size, MPI_DOUBLE, 1, FROM_ONE_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else if (rank == 1) {
  // Copy to temporary buffer and returns
  MPI_Send(send_buffer, buffer_size, MPI_DOUBLE, 0, FROM_ONE_TAG, MPI_COMM_WORLD);
  // Able to receive message from rank 0
  MPI_Recv(recv_buffer, buffer_size, MPI_DOUBLE, 0, FROM_ZERO_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Communication protocol: common pitfall

Let's consider a second scenario where the **Rendezvous protocol** is used. In that case the application will deadlock and will not terminate successfully.

```
if (rank == 0) {
  // Block until a maching Recv is posted
  MPI_Send(send_buffer, buffer_size, MPI_DOUBLE, 1, FROM_ZERO_TAG, MPI_COMM_WORLD);
  // Will never be posted because previous Send call is waiting for Recv from rank 1
  MPI_Recv(recv_buffer, buffer_size, MPI_DOUBLE, 1, FROM_ONE_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else if (rank == 1) {
  // Block until a maching Recv is posted
  MPI_Send(send_buffer, buffer_size, MPI_DOUBLE, 0, FROM_ONE_TAG, MPI_COMM_WORLD);
  // Will never be posted because previous Send call is waiting for Recv from rank 0
  MPI_Recv(recv_buffer, buffer_size, MPI_DOUBLE, 0, FROM_ZERO_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Use of tags: common pitfall

It is important to ensure that the message tags and their ordering match between the send and receive sides. A mismatch can lead to a deadlock when the rendezvous protocol is used.

```
for (size_t buffer_size = 8; buffer_size < 32768; buffer_size *= 2) {
  if (rank == 0) {
    // Block until a maching Recv with tag = FROM_ZERO_TAG1 is posted
    MPI_Send(send_buffer1, buffer_size, MPI_DOUBLE, 1, FROM_ZERO_TAG1, MPI_COMM_WORLD);
    MPI_Send(send_buffer2, buffer_size, MPI_DOUBLE, 1, FROM_ZERO_TAG2, MPI_COMM_WORLD);

    MPI_Recv(recv_buffer1, buffer_size, MPI_DOUBLE, 1, FROM_ONE_TAG1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(recv_buffer2, buffer_size, MPI_DOUBLE, 1, FROM_ONE_TAG2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else if (rank == 1) {
    // Block until a maching Send with tag = FROM_ZERO_TAG2 is posted
    MPI_Recv(recv_buffer1, buffer_size, MPI_DOUBLE, 0, FROM_ZERO_TAG2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(recv_buffer2, buffer_size, MPI_DOUBLE, 0, FROM_ZERO_TAG1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    MPI_Send(send_buffer1, buffer_size, MPI_DOUBLE, 0, FROM_ONE_TAG2, MPI_COMM_WORLD);
    MPI_Send(send_buffer2, buffer_size, MPI_DOUBLE, 0, FROM_ONE_TAG1, MPI_COMM_WORLD);
  }
}
```

Force synchronous (Rendezvous) send

It is possible to force synchronous (rendezvous) communication by using the blocking send routine `MPI_Ssend` which ensures that the send operation does not complete until the matching receive has been initiated. The syntax is identical to that of `MPI_Send`.

C/C++ Syntax

```
int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

Fortran Syntax

```
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type>    BUF(*)
INTEGER   COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Fortran 2008 Syntax

```
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

POINT-TO-POINT COMMUNICATION: MORE PATTERNS AND PITFALLS

Embarrassingly parallel *vs* tightly coupled

The previous example was **embarrassingly parallel**, meaning that in the “update” loop

```
for (uint64_t istep = start; istep < end; istep++)
{
  const double x = ((double)istep + 0.5) * step
  sum += 4.0 / (1.0 + x * x);
}
```

each process can compute its portion of the sum independently, without requiring any data or synchronization from the other processes

However, many scientific applications are **tightly coupled**, meaning that computations depend on frequent communication or synchronization between processes to exchange intermediate results or boundary data.

The 1D diffusion equation

The diffusion equation, also known as the heat equation, reads

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, L), t \in (0, T]$$

where $u(x, t)$ is the unknown function to be solved for, x is a coordinate in space, t is time and α is the diffusion coefficient

After discretization and using a forward difference in time and a central difference in space, we get

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

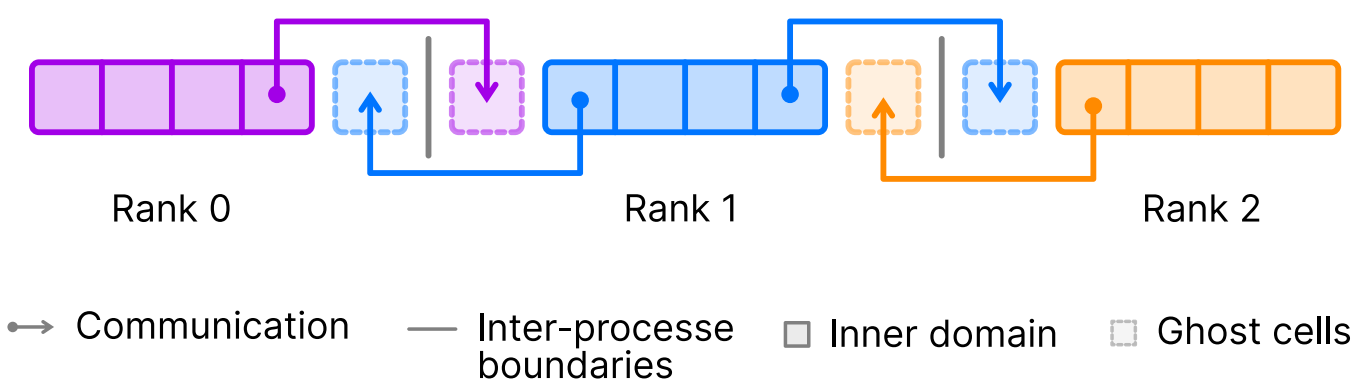
so that, at time step $n + 1$, we can update the value of u using

$$u_i^{n+1} = u_i^n + \frac{\alpha \Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

The 1D diffusion equation: domain decomposition

In order to parallelize the 1D diffusion equation:

- The computational domain is divided into subdomains, each assigned to a process
- Updating u_i^{n+1} requires the neighboring values u_{i-1}^n and u_{i+1}^n, which belong to the left and right neighboring processes
- To access these boundary values, ghost cells are introduced at the edges of each subdomain
- The ghost cell values are exchanged between neighboring processes at each time step via message passing.



The 1D diffusion equation: first implementation


```
const size_t subdom_start = GRID_SIZE * rank / num_ranks;
const size_t subdom_end   = GRID_SIZE * (rank + 1) / num_ranks;
const size_t subdom_size  = subdom_end - subdom_start;

const int left_rank  = rank > 0         ? rank - 1 : MPI_PROC_NULL;
const int right_rank = rank < num_ranks ? rank + 1 : MPI_PROC_NULL;

for (int tstep = 0; tstep < num_tsteps; tstep++) {
  MPI_Send(&uold[subdom_size], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD);
  MPI_Recv(&uold[0],           1, MPI_DOUBLE, left_rank,  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  MPI_Send(&uold[1],             1, MPI_DOUBLE, left_rank,  1, MPI_COMM_WORLD);
  MPI_Recv(&uold[subdom_size+1], 1, MPI_DOUBLE, right_rank, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  for (int i = 1; i <= subdom_size; i++) {
    unew[i] = uold[i] + alpha_dt_dx2 * (uold[i+1] - 2.0 * uold[i] + uold[i-1]);
  }
}
```

The 1D diffusion equation: first implementation (Fortran)


```
subdom_start = GRID_SIZE * rank / num_ranks
subdom_end   = GRID_SIZE * (rank + 1) / num_ranks
subdom_size  = subdom_end - subdom_start

left_rank  = merge(rank - 1, MPI_PROC_NULL, rank > 0)
right_rank = merge(rank + 1, MPI_PROC_NULL, rank < num_ranks-1)

do tstep = 1, num_tsteps
  call MPI_Send(uold(subdom_size),  1, MPI_DOUBLE_PRECISION, right_rank, 0, MPI_COMM_WORLD)
  call MPI_Recv(uold(0),            1, MPI_DOUBLE_PRECISION, left_rank,  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

  call MPI_Send(uold(1),             1, MPI_DOUBLE_PRECISION, left_rank,  1, MPI_COMM_WORLD)
  call MPI_Recv(uold(subdom_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

  do i = 1, subdom_size
    unew(i) = uold(i) + alpha_dt_dx2 * (uold(i+1) - 2.0 * uold(i) + uold(i-1))
  end do
end do
```

Null processes

In some situations, it is convenient to specify a “dummy” source or destination for communication. This simplifies boundary handling in non-periodic domains. For example, when performing a rank shift

```
const int left_rank  = rank > 0          ? rank - 1 : MPI_PROC_NULL;
const int right_rank = rank < num_ranks ? rank + 1 : MPI_PROC_NULL;
```

the special constant `MPI_PROC_NULL` can be used in place of a process rank wherever a source or destination argument is required. A communication involving `MPI_PROC_NULL` has no effect:

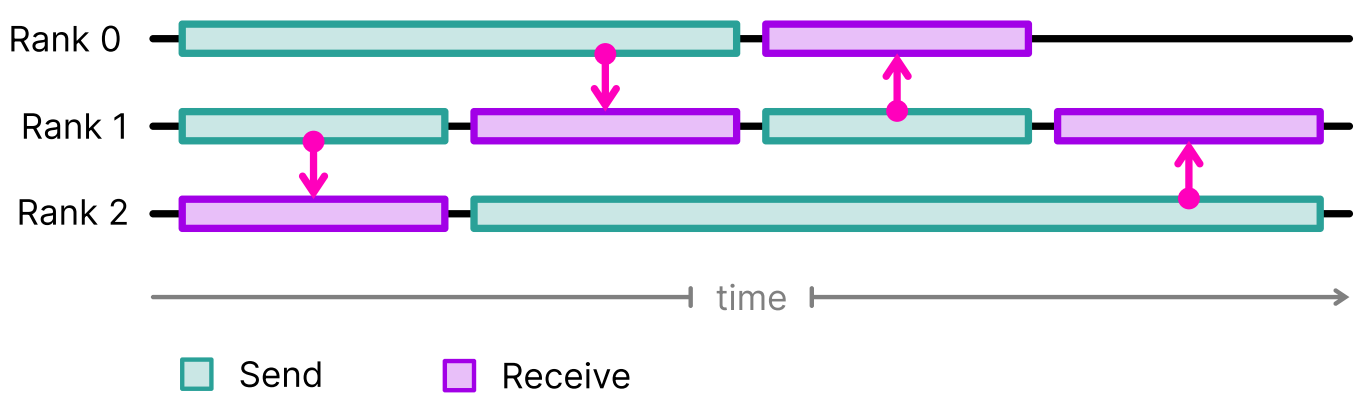
- A send to `MPI_PROC_NULL` returns immediately and performs no operation
- A receive from `MPI_PROC_NULL` also returns immediately, leaving the receive buffer unchanged

When a receive is issued with `source = MPI_PROC_NULL`, the status object is filled with `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG`, and `count = 0`

The 1D diffusion equation: performance

When implementing the exchange using blocking `MPI_Send` and `MPI_Recv`, communication performance can degrade significantly if the the synchronous (rendezvous) protocol is used:

- The send call waits for the matching receive to be posted before proceeding
- As a result, the exchange between neighboring processes becomes serialized. Each process must wait for its neighbor to reach the matching communication call
- This can lead to noticeable slowdowns, especially on large process counts. To mitigate this issue alternative communication patterns are typically preferred



The 1D diffusion equation: periodic boundary conditions

Using blocking send and receive operations in a communication pattern with cyclic dependencies (periodic boundary conditions) can easily lead to deadlocks:

- each process call `MPI_Send` first and then wait for its matching `MPI_Recv`
- if all processes are simultaneously waiting for each other to post their receives, none can proceed, and the program stalls indefinitely

To avoid this situation, the communication pattern must be designed so that not all processes block at the same time:

- **Solution 1** odd–even pattern
- **Solution 2** use the combined send–receive routine `MPI_Sendrecv`

The 1D diffusion equation: odd-even implementation

One approach is to use an odd–even communication pattern: even-ranked processes call `MPI_Send` first and `MPI_Recv` second, while odd-ranked processes perform the operations in the opposite order

```
if(rank % 2 == 0) {
  MPI_Send(&uold[subdom_size], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD);
  MPI_Recv(&uold[0],           1, MPI_DOUBLE, left_rank,  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  MPI_Send(&uold[1],             1, MPI_DOUBLE, left_rank,  1, MPI_COMM_WORLD);
  MPI_Recv(&uold[subdom_size+1], 1, MPI_DOUBLE, right_rank, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
  MPI_Recv(&uold[0],           1, MPI_DOUBLE, left_rank,  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(&uold[subdom_size], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD);

  MPI_Recv(&uold[subdom_size+1], 1, MPI_DOUBLE, right_rank, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(&uold[1],             1, MPI_DOUBLE, left_rank,  1, MPI_COMM_WORLD);
}
```

The 1D diffusion equation: odd-even implementation (Fortan)

One approach is to use an odd–even communication pattern: even-ranked processes call `MPI_Send` first and `MPI_Recv` second, while odd-ranked processes perform the operations in the opposite order

```
if (modulo(rank, 2) .eq. 0) then
  call MPI_Send(uold(subdom_size), 1, MPI_DOUBLE_PRECISION, right_rank, 0, MPI_COMM_WORLD)
  call MPI_Recv(uold(0),           1, MPI_DOUBLE_PRECISION, left_rank,  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

  call MPI_Send(uold(1),             1, MPI_DOUBLE_PRECISION, left_rank,  1, MPI_COMM_WORLD)
  call MPI_Recv(uold(subdom_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
else
  call MPI_Recv(uold(0),           1, MPI_DOUBLE_PRECISION, left_rank,  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
  call MPI_Send(uold(subdom_size), 1, MPI_DOUBLE_PRECISION, right_rank, 0, MPI_COMM_WORLD)

  call MPI_Recv(uold(subdom_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
  call MPI_Send(uold(1),             1, MPI_DOUBLE_PRECISION, left_rank,  1, MPI_COMM_WORLD)
end if
```

Combined sendrecv: syntax

The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process.

- The two (source and destination) are possibly the same
- When a send-receive operation is used, the communication subsystem takes care of the issue of preventing cyclic dependencies that may lead to deadlock

C/C++ Syntax

```
int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag,
     void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag,
     MPI_Comm comm, MPI_Status *status)
```

Combined sendrecv: syntax (Fortran)

Fortran Syntax

```
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST,   SENDTAG,
             RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
<type>    SENDBUF(*), RECVBUF(*)
INTEGER   SENDCOUNT, SENDTYPE, DEST, SENDTAG
INTEGER   RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM
INTEGER   STATUS(MPI_STATUS_SIZE), IERROR
```

Fortran 2008 Syntax

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest,   sendtag,
             recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source, recvtag
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

The 1D diffusion equation: combined sendrecv

With `MPI_Sendrecv` the exchange of the ghost cells is done in two MPI calls:

- In the first call, each process sends its leftmost interior value to its left neighbor and receives the right neighbor boundary value into its right ghost cell
- In the second call, the direction is reversed: each process sends its rightmost interior value to its right neighbor and receives the left neighbor boundary value into its left ghost cell.

```
// Send to the left and receive from the right
MPI_Sendrecv(&uold[            1], 1, MPI_DOUBLE, left_rank,  0,
             &uold[subdom_size+1], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// Send to the right and receive from the left
MPI_Sendrecv(&uold[subdom_size], 1, MPI_DOUBLE, right_rank, 1,
             &uold[          0], 1, MPI_DOUBLE, left_rank,  1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

The 1D diffusion equation: combined sendrecv (Fortran)

With `MPI_Sendrecv` the exchange of the ghost cells is done in two MPI calls:

- In the first call, each process sends its leftmost interior value to its left neighbor and receives the right neighbor boundary value into its right ghost cell
- In the second call, the direction is reversed: each process sends its rightmost interior value to its right neighbor and receives the left neighbor boundary value into its left ghost cell.

```
! Send to the left and receive from the right
call MPI_Sendrecv(uold(           1), 1, MPI_DOUBLE_PRECISION, left_rank,  0, &
                  uold(subdom_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

! Send to the right and receive from the left
call MPI_Sendrecv(uold(subdom_size), 1, MPI_DOUBLE_PRECISION, right_rank, 1, &
                  uold(          0), 1, MPI_DOUBLE_PRECISION, left_rank,  1, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
```

NON-BLOCKING COMMUNICATION

The advantage of non-blocking communication

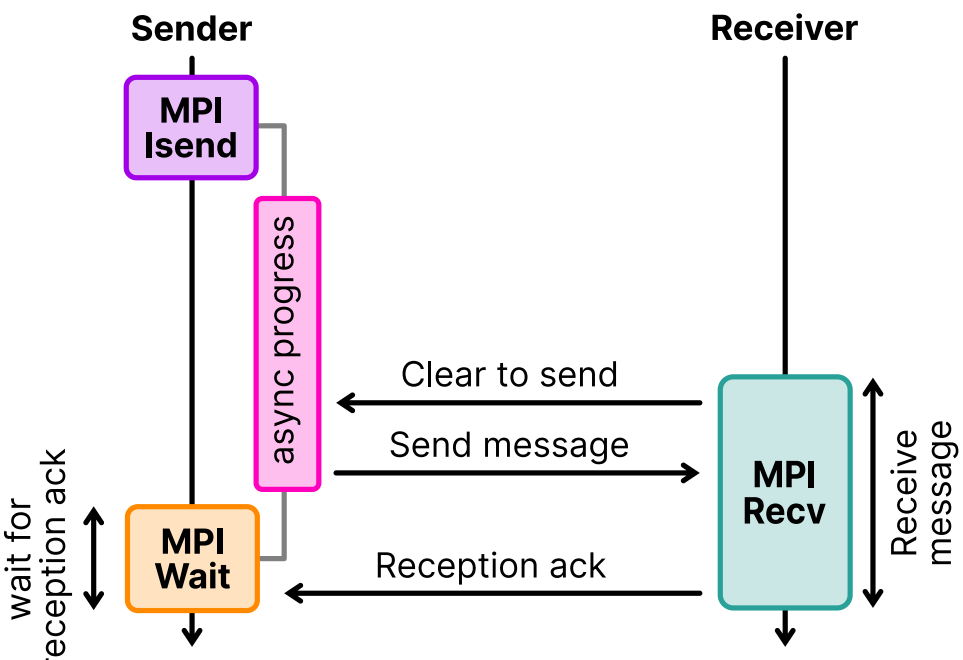
In many parallel applications, communication and computation do not have to happen one after the other. Non-blocking communication allows processes to initiate data transfers without waiting for them to complete, enabling useful work to be performed while messages are still in transit.

This approach can help reduce idle time and improve overall performance, especially in large-scale systems where communication can become a bottleneck.

MPI non-blocking communication

MPI provides the non-blocking routine `MPI_Isend`, which initiates a send operation but returns immediately, allowing the process to continue execution without waiting for the data transfer to complete

- The call to `MPI_Isend` starts the communication but doesn't guarantee that the data has been sent yet
- Since `MPI_Isend` is non-blocking, the process can perform other computations or initiate additional communications while the message is being transferred in the background
- `MPI_Wait` is used to check for completion.



Sending a non-blocking message: syntax

C/C++ Syntax

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,
     int tag, MPI_Comm comm, MPI_Request *request)
```

Fortran Syntax

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type>    BUF(*)
INTEGER   COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Fortran 2008 Syntax

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI request

The arguments of `MPI_Isend` are identical to those of `MPI_Send`, except for one additional parameter: an `MPI_Request` handle

- An `MPI_Request` handle represents a non-blocking communication operation initiated by routines such as `MPI_Isend` or other non-blocking calls
- The `MPI_Request` object keeps track of the state of the operation: whether it is still in progress or has completed
- This handle is then used by completion routines to manage and synchronize non-blocking operations:
  - The wait functions (`MPI_Wait*`) block the program until one or more of the associated operations have finished.
  - The test functions (`MPI_Test*`) allow the program to poll the request's status without blocking

MPI wait

`MPI_Wait` is used to wait for an MPI send or receive to complete

- A call to `MPI_Wait` returns when the operation identified by the request is completed
- If the request was created by a non-blocking send or receive call, then it is deallocated by the call to `MPI_Wait` and the request handle is set to `MPI_REQUEST_NULL`
- The call returns, in status, information on the completed operation

C/C++ Syntax

```
int MPI_Wait(MPI_Request *request,
      MPI_Status *status)
```

Fortran Syntax

```
MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

Fortran 2008 Syntax

```
MPI_Wait(request, status, ierror)
TYPE(MPI_Request), INTENT(INOUT) :: request
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI wait all

`MPI_Waitall` blocks until multiple (`count`) requests to completed and return the statuses information for the completed requests

C/C++ Syntax

```
int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status *array_of_statuses)
```

Fortran Syntax

```
MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
INTEGER   COUNT, ARRAY_OF_REQUESTS(*)
INTEGER   ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

Fortran 2008 Syntax

```
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
TYPE(MPI_Status) :: array_of_statuses(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI wait all

`MPI_Waitall` blocks until multiple (`count`) requests to completed and return the statuses information for the completed requests

C/C++ Syntax

```
int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status *array_of_statuses)
```

Fortran Syntax

```
MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
INTEGER   COUNT, ARRAY_OF_REQUESTS(*)
INTEGER   ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

Fortran 2008 Syntax

```
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
TYPE(MPI_Status) :: array_of_statuses(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

The 1D diffusion equation: using non-blocking send

We can modify the 1D diffusion code to use non-blocking communication by replacing the blocking `MPI_Send` calls with non-blocking `MPI_Isend`. This change allows each process to initiate data transfers and immediately proceed to the receive

In this version, we still use blocking `MPI_Recv` calls for simplicity. Mixing blocking and non-blocking communication is perfectly valid in MPI

```
MPI_Request requests[2];

MPI_Isend(&uold[subdom_size],   1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD, &requests[0]);
MPI_Recv(&uold[0],              1, MPI_DOUBLE, left_rank,  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Isend(&uold[1],             1, MPI_DOUBLE, left_rank,  1, MPI_COMM_WORLD, &requests[1]);
MPI_Recv(&uold[subdom_size+1], 1, MPI_DOUBLE, right_rank, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Waitall(2, requests, MPI_STATUSES_IGNORE);
```

The 1D diffusion equation: using non-blocking send (Fortran)

We can modify the 1D diffusion code to use non-blocking communication by replacing the blocking `MPI_Send` calls with non-blocking `MPI_Isend`. This change allows each process to initiate data transfers and immediately proceed to the receive

In this version, we still use blocking `MPI_Recv` calls for simplicity. Mixing blocking and non-blocking communication is perfectly valid in MPI

```
type(MPI_Request) :: requests(2)

call MPI_Isend(uold(subdom_size), 1, MPI_DOUBLE_PRECISION, right_rank, 0, MPI_COMM_WORLD, requests(1))
call MPI_Irecv(uold(0),           1, MPI_DOUBLE_PRECISION, left_rank,  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

call MPI_Isend(uold(1),            1, MPI_DOUBLE_PRECISION, left_rank,  1, MPI_COMM_WORLD, requests(2))
call MPI_Recv(uold(subdom_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

call MPI_Waitall(2, requests, MPI_STATUSES_IGNORE)
```

Receiving a non-blocking message: syntax

C/C++ Syntax

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
     MPI_Comm comm, MPI_Request *request)
```

Fortran Syntax

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
<type>  BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

Fortran 2008 Syntax

```
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count, source, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

1D diffusion equation: hiding communication

To improve performance, we can hide communication behind computation by using non-blocking communication routines such as `MPI_Isend` and `MPI_Irecv`. The idea is to initiate data transfers as early as possible, then perform computations that do not depend on the incoming ghost cells while the communication progresses in the background

For example, in our 1D domain decomposition:

- each process can first post non-blocking sends and receives for its boundary values
- proceed to compute the interior points of its subdomain: **those that do not require the ghost cells**
- once the interior computations are complete, the process can call `MPI_Wait` to ensure that the communication has finished and then update the boundary points using the newly received data

The 1D diffusion equation: hiding communication

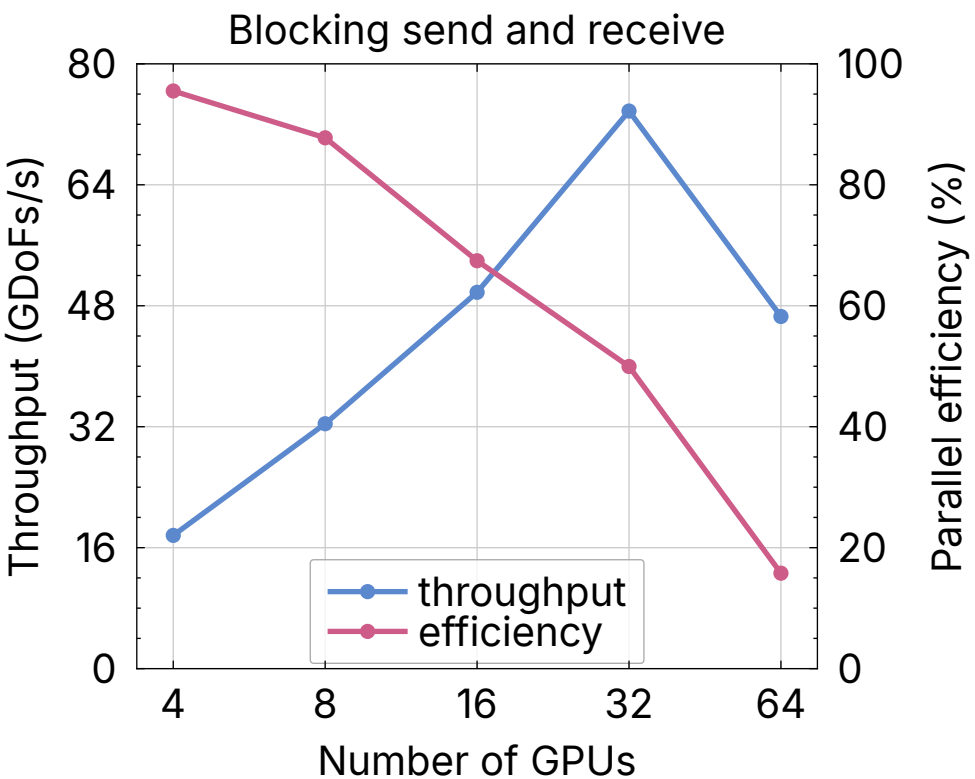

```
MPI_Request recv_requests[2];
MPI_Request send_requests[2];

MPI_Irecv(&uold[0],             1, MPI_DOUBLE, left_rank,  0, MPI_COMM_WORLD, &recv_requests[0]);
MPI_Irecv(&uold[subdom_size+1], 1, MPI_DOUBLE, right_rank, 1, MPI_COMM_WORLD, &recv_requests[1]);

MPI_Isend(&uold[subdom_size], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD, &send_requests[0]);
MPI_Isend(&uold[1],           1, MPI_DOUBLE, left_rank,  1, MPI_COMM_WORLD, &send_requests[1]);

for (int i = 2; i <= subdom_size-1; i++) {
  unew[i] =  uold[i] + alphadt_dx2 * (uold[i+1] - 2.0 * uold[i] + uold[i-1]);
}

MPI_Waitall(2, recv_requests, MPI_STATUSES_IGNORE);

unew[1] =  uold[1] + alphadt_dx2 * (uold[2] - 2.0 * uold[1] + uold[0]);
unew[subdom_size] =  uold[subdom_size] + alphadt_dx2
    * (uold[subdom_size+1] - 2.0 * uold[subdom_size] + uold[subdom_size-1]);

MPI_Waitall(2, send_requests, MPI_STATUSES_IGNORE);
```

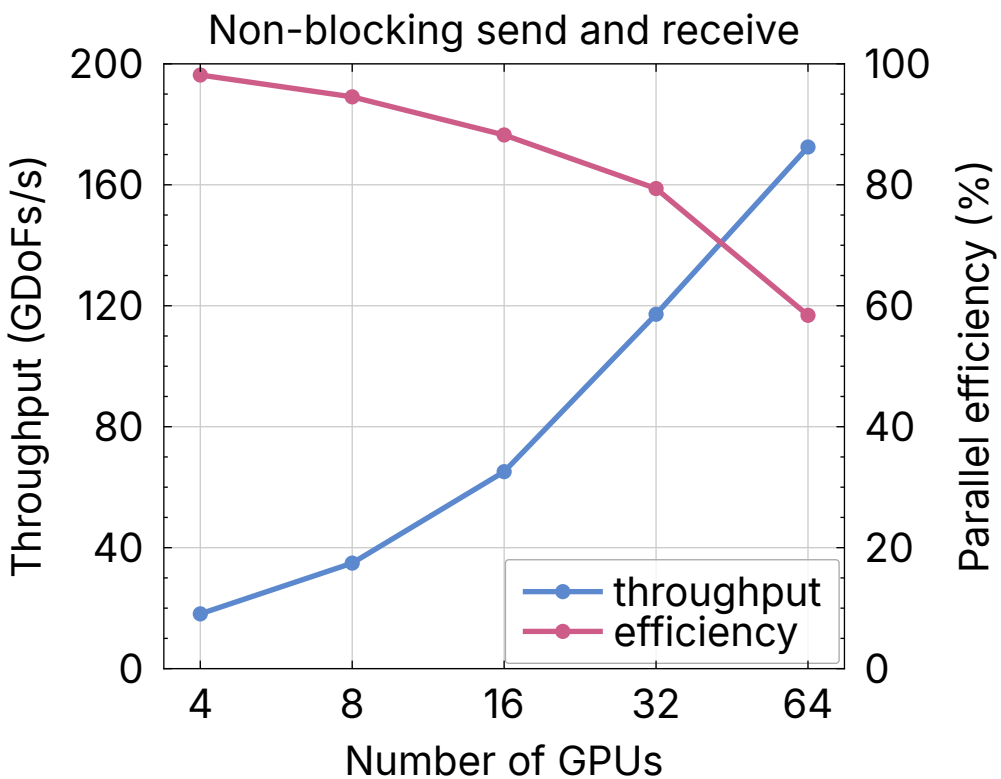
The 1D diffusion equation: hiding communication (Fortran)


```
type(MPI_Request) :: recv_requests(2), send_requests(2)

call MPI_Isend(uold(subdom_size), 1, MPI_DOUBLE_PRECISION, right_rank, 0, MPI_COMM_WORLD, send_requests(1))
call MPI_Isend(uold(1),           1, MPI_DOUBLE_PRECISION, left_rank,  1, MPI_COMM_WORLD, send_requests(2))

call MPI_Irecv(uold(0),             1, MPI_DOUBLE_PRECISION, left_rank,  0, MPI_COMM_WORLD, recv_requests(1))
call MPI_Irecv(uold(subdom_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, MPI_COMM_WORLD, recv_requests(2))

do i = 2, subdom_size-1
  unew(i) = uold(i) + alphadt_dx2 * (uold(i+1) - 2.0 * uold(i) + uold(i-1))
end do

call MPI_Waitall(2, recv_requests, MPI_STATUSES_IGNORE)

unew(1) =  uold(1) + alphadt_dx2 * (uold(2) - 2.0 * uold(1) + uold(0))
unew(subdom_size) =  uold(subdom_size) + alphadt_dx2 &
    * (uold(subdom_size+1) - 2.0 * uold(subdom_size) + uold(subdom_size-1))

call MPI_Waitall(2, send_requests, MPI_STATUSES_IGNORE)
```

Example of the effect of communication on performance (1/2)

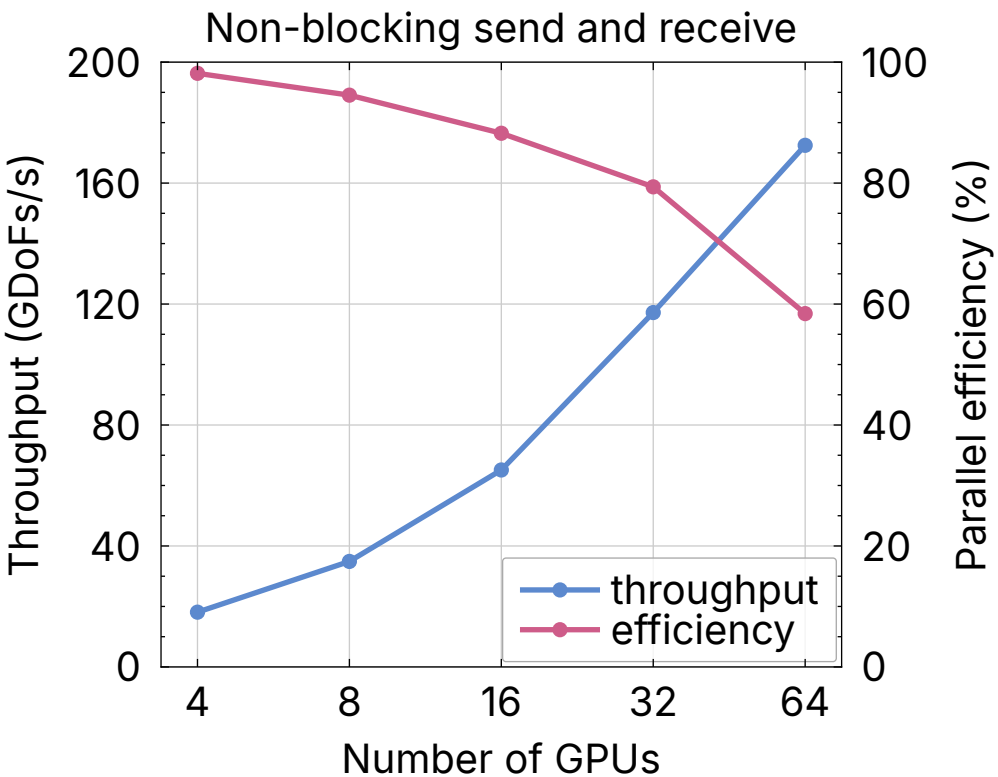
To illustrate the impact of communication on performance, we will examine a GPU-based Discontinuous Galerkin Maxwell solver, comparing results obtained with blocking and non-blocking send and receive operations

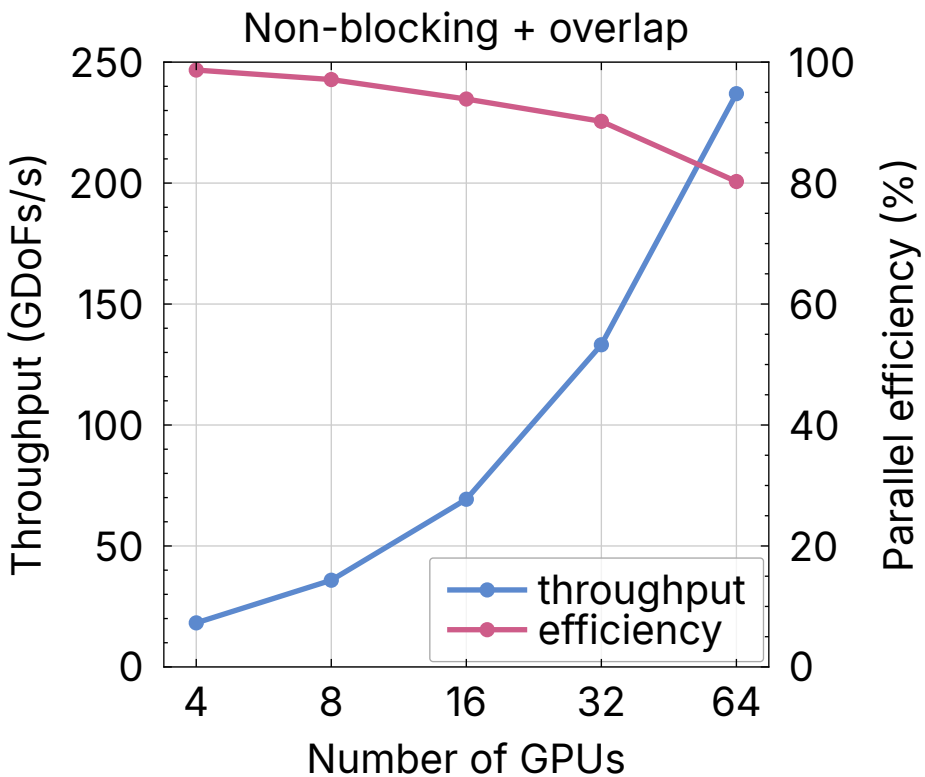




Example of the effect of communication on performance (2/2)

By using non-blocking send and receive operations and overlapping communication with computation, we achieve roughly a 30% gain in performance





MPI AND ACCELERATORS (GPU)

GPU aware MPI

Most MPI implementations nowadays are GPU-aware. GPU-aware MPI refers to MPI libraries that can directly send and receive data stored in GPU memory without requiring the application to copy data back to the CPU first. **With GPU-aware MPI, applications can call MPI functions using device pointers directly**

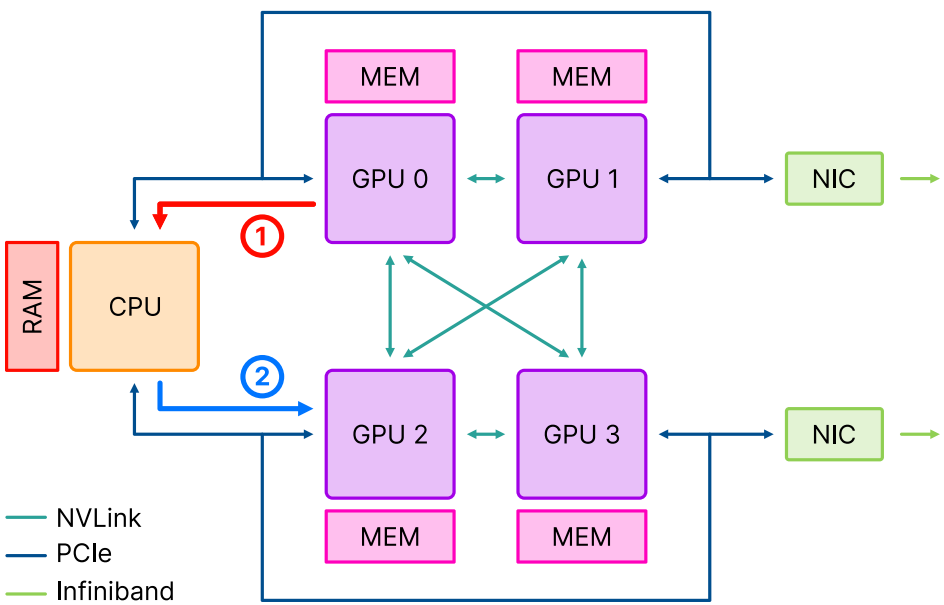
In traditional (non–GPU-aware) MPI:

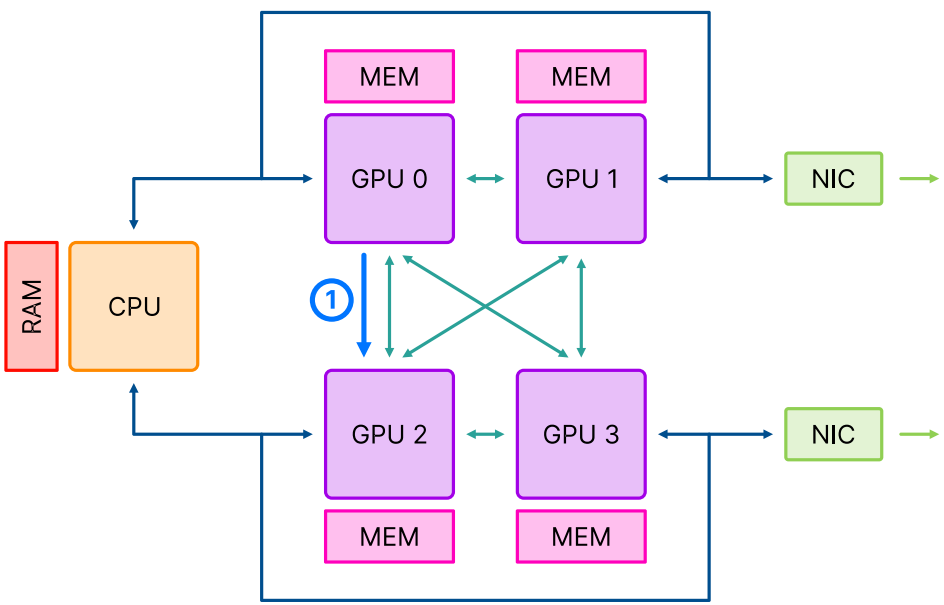
- Data must be copied from GPU memory to the host (CPU) memory
- MPI sends the CPU buffer over the network
- The receiver copies CPU memory to GPU memory

These extra copies cost time, bandwidth, and CPU involvement

GPU aware MPI: same node communication

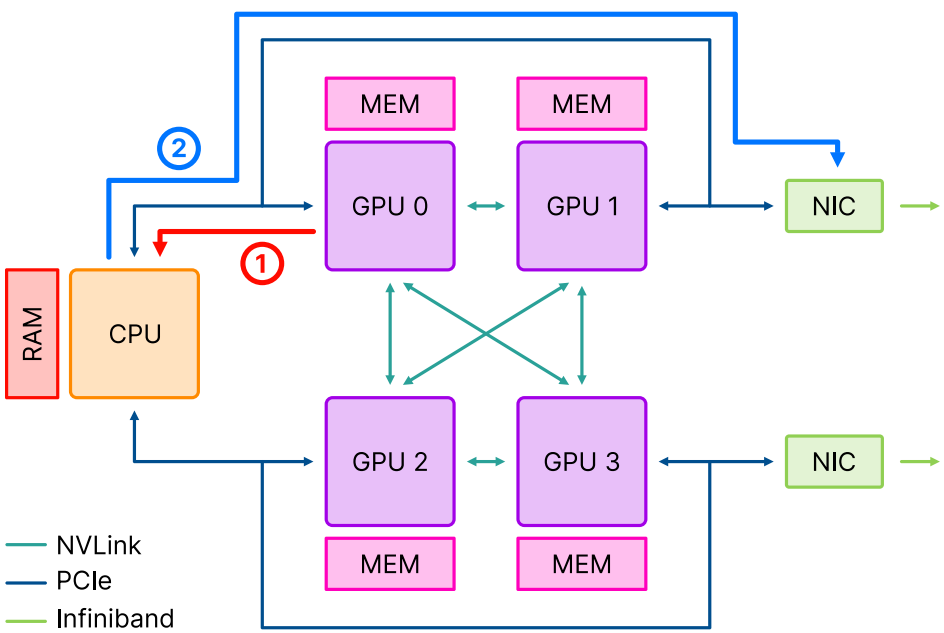
If two ranks are on the same node, MPI may use GPU peer-to-peer (P2P) transfers (via NVLink, PCIe, or xGMI), use shared memory segments exposed to GPUs or use CUDA IPC to map one process GPU memory into another's address space

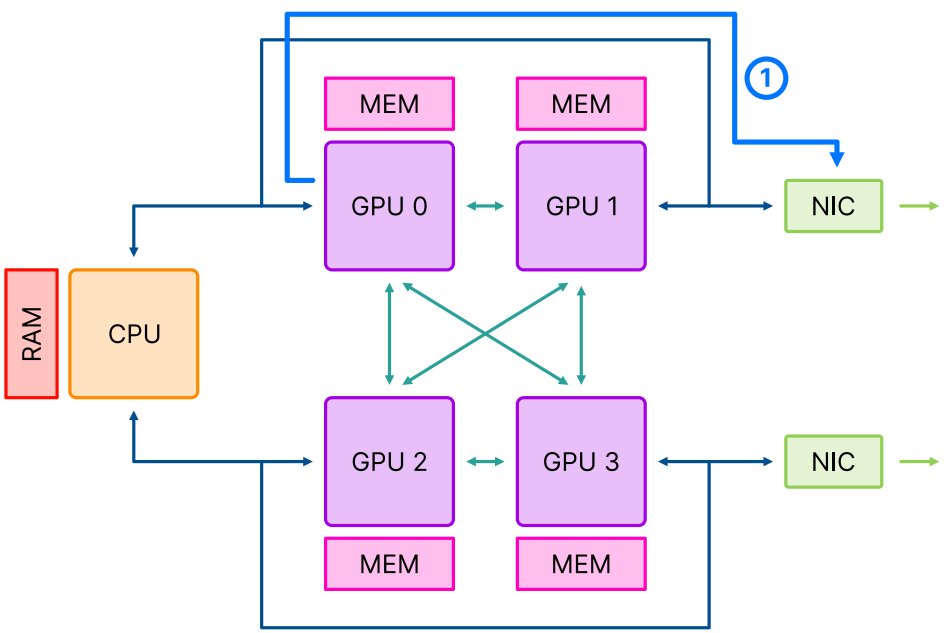




GPU aware MPI: node to node communication

For communication between nodes, GPU-aware MPI can use GPUDirect RDMA, which allows network interface (NIC) to read/write GPU memory directly, without CPU involvement and without staging through host memory





COLLECTIVE COMMUNICATION

Collective communication

So far, we have focused on **point-to-point** (communication, where data is explicitly **exchanged between two specific processes** using operations such as `MPI_Send` and `MPI_Recv`. While these routines give fine-grained control over data movement, they can become cumbersome and inefficient when multiple processes need to communicate in a coordinated way

In contrast, **collective** communication **involves all processes** in a communicator participating in a single, coordinated operation. Instead of managing many individual sends and receives between pairs of processes

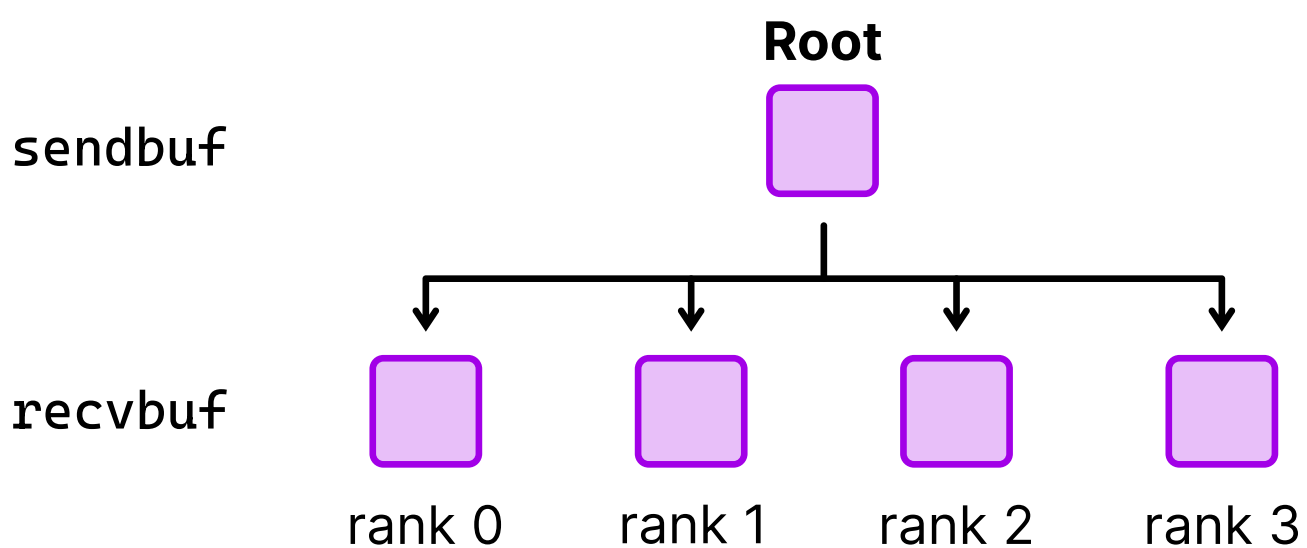
Examples include operations such as:

- **Broadcasts:** where one process sends data to all others
- **Gather and scatter:** operations which collect or distribute data among processes
- **Reductions:** combine values from all processes (e.g., sums, minima, maxima)

Broadcast: overview

An `MPI_Bcast` operation broadcasts a message stored in `buffer` of the process with rank `root` and store it in the `buffer` of all other processes of the communicator `comm`

To avoid deadlock, all processes within the communicator `comm` must invoke `MPI_Bcast`. The same requirement applies to all collective communication routines in MPI



Broadcast: syntax

C/C++ Syntax

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

Fortran Syntax

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
<type>    BUFFER(*)
INTEGER   COUNT, DATATYPE, ROOT, COMM, IERROR
```

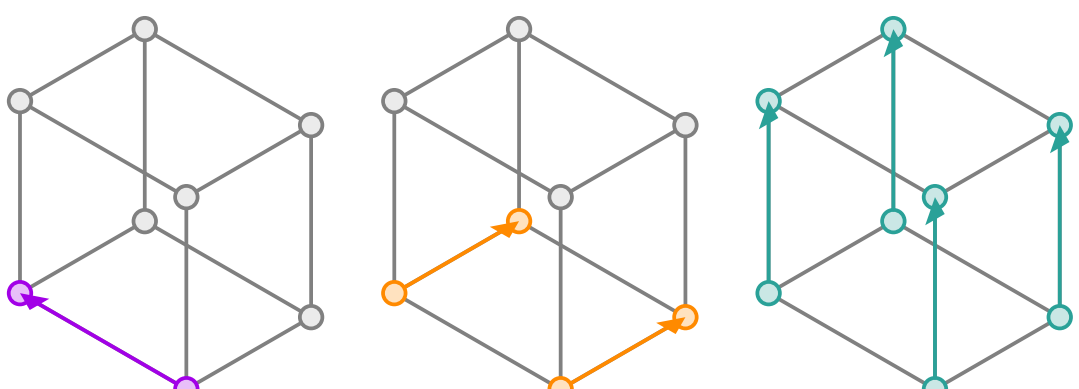
Fortran 2008 Syntax

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
TYPE(*), DIMENSION(..) :: buffer
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Why Use Collective Communication in MPI

Collective communication routines in MPI, such as `MPI_Bcast`, can, in principle, be implemented using only point-to-point operations. However, a naïve broadcast implemented with point-to-point calls often uses a simple linear pattern resulting in $O(p)$ steps, where p is the number of processes

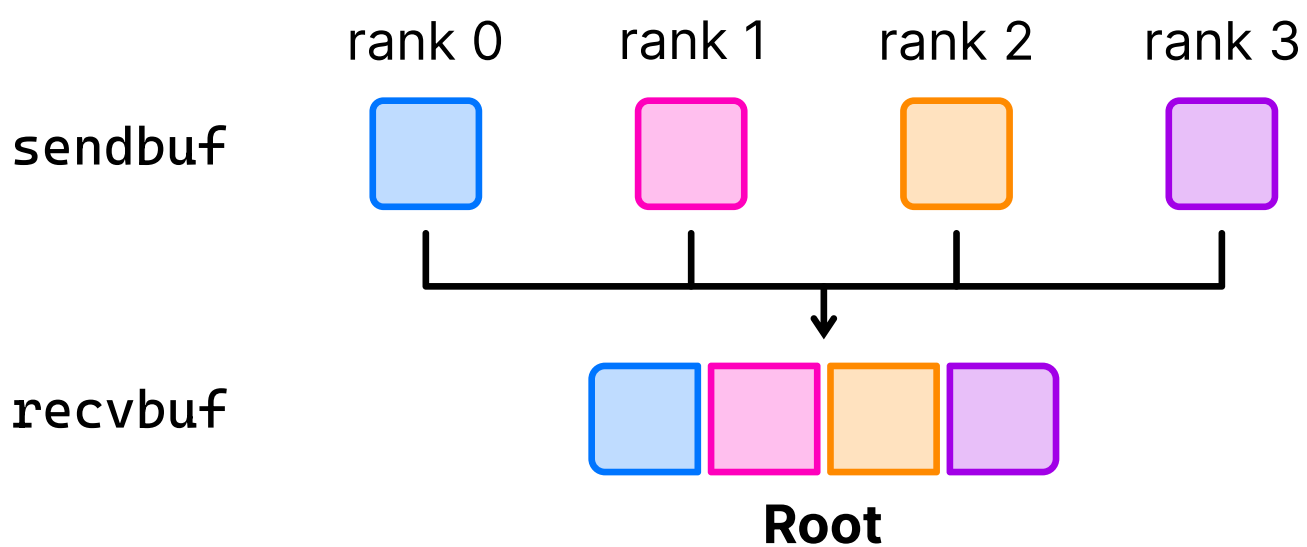
Many MPI collectives use communication patterns based on a hypercube or binomial tree, which have logarithmic depth. With such patterns, the number of processes that "have" the data doubles at each step, the broadcast completes in $\log_2(p)$ steps



Gather: overview

An `MPI_Gather` operation collects data from all processes in a communicator into a single receive buffer on the `root` process. Conceptually it is as if each process provides its local contribution in `sendbuf`, which contains `sendcount` elements of type `sendtype`

The `root` process gathers all these messages into its `recvbuf`, where the i^{th} block of `recvcount` elements of type `recvtype` corresponds to the data sent from process i



Gather: syntax

C/C++ Syntax

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

Fortran Syntax

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type>  SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT
INTEGER COMM, IERROR
```

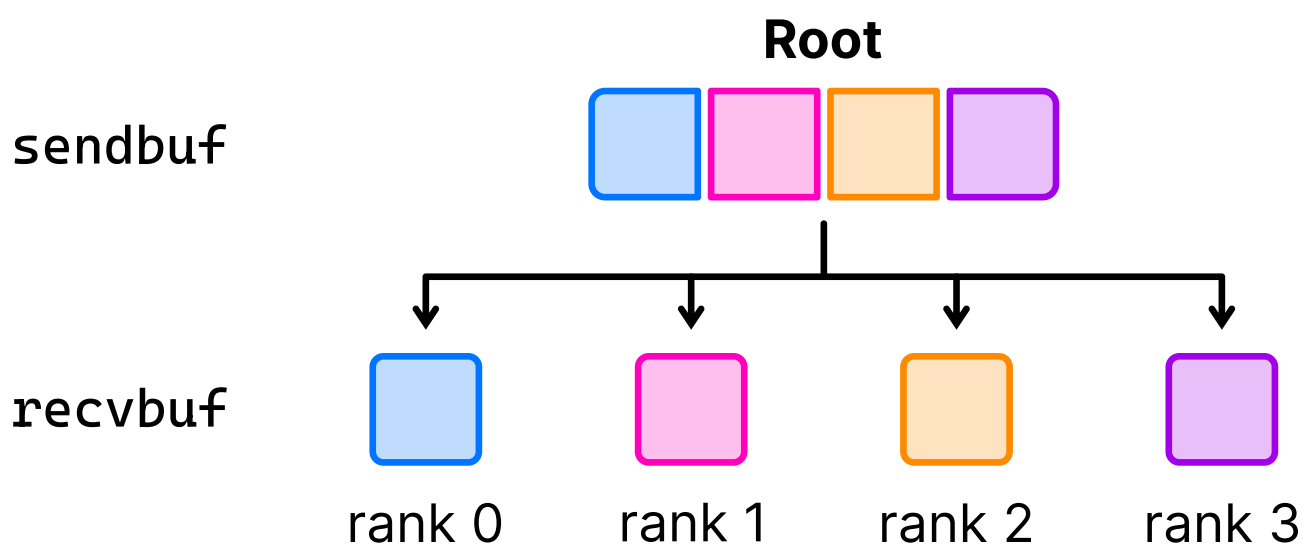
Fortran 2008 Syntax

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount, root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Scatter: overview

An `MPI_Scatter` performs the inverse operation of `MPI_Gather`. Conceptually, it is as if the `root` process takes a contiguous buffer `sendbuf`, splits it into `num_ranks` equal segments of size `sendcount`, and sends the i^{th} segment to the i^{th} process

Meanwhile, each process performs a corresponding receive to obtain its portion of the data in `recvbuf`



Scatter: syntax

C/C++ Syntax

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

Fortran Syntax

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type>  SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT
INTEGER COMM, IERROR
```

Fortran 2008 Syntax

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount, root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Gather and Scatter: regarding the arguments

- All arguments to `MPI_Gather` and `MPI_Scatter` are significant on process `root`
- On other processes:
  - **Gather:** only arguments `sendbuf`, `sendcount`, `sendtype`, `root` and `comm` are significant
  - **Scatter:** only arguments `recvbuf`, `recvcount`, `recvtype`, `root` and `comm` are significant
- The arguments `root` and `comm` must have identical values on all processes
- All processes in the communicator `comm` must call `MPI_Gather`/`MPI_Scatter` otherwise, the program will hang

Computing π: computing the final result with a gather

We can rewrite the way we computed the final sum for the computation of π using an `MPI_Gather`:

```
if (rank == 0) {
  double* remote_sums = malloc(num_ranks * sizeof(double));
  MPI_Gather(&sum, 1, MPI_DOUBLE, remote_sums, 1, MPI_DOUBLE, MPI_COMM_WORLD);

  double pi = 0.0;
  for (int srank = 1; srank < num_ranks; srank++) {
    pi += step * remote_sums[i];
  }

  free(remote_sums);

  printf(" Computed value of pi with %" PRIu64 " steps is %.12lf\n", NUM_STEPS, pi);
  printf(" Computation took %lf seconds\n", elapsed);
} else {
  MPI_Gather(&sum, 1, MPI_DOUBLE, NULL, 1, MPI_DOUBLE, MPI_COMM_WORLD);
}
```

Reduction: overview

The `MPI_Reduce` operation allows processes to combine data from all ranks in a communicator into a single result using a specified reduction operation (sum, maximum, minimum, ...)

- Each process provides a local value, and the result of applying the operation across all processes is stored on a designated root process.
- This is particularly useful when aggregating results from distributed computations: summing partial results computed by each process to obtain a global total, finding the maximum value across all ranks, or combining arrays element-wise

MPI Op	Operation	MPI Op	Operation	MPI Op	Operation
`MPI_MIN`	`min`	`MPI_SUM`	`+`	`MPI_BAND`	`&`
`MPI_MAX`	`max`	`MPI_PROD`	`*`	`MPI_BOR`	`|`
`MPI_LAND`	`&&`	`MPI_LOR`	`||`		

Reduction: syntax

C/C++ Syntax

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

Fortran Syntax

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
<type>  SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

Fortran 2008 Syntax

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Computing π: computing the final result with a reduction

We can rewrite the way we computed the final sum for the computation of π using an `MPI_Reduce`:

```
double final_sum = 0.0;
MPI_Reduce(&sum, &final_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0) {
  const double pi = final_sum * step;

  printf(" Computed value of pi with %" PRIu64 " steps is %.12lf\n", NUM_STEPS, pi);
  printf(" Computation took %lf seconds\n", elapsed);
}
```

DEBUGGING TIPS

Using GDB with MPI

Parallel debuggers such as Perforce TotalView and Linaro DDT provide powerful tools for inspecting MPI applications, but they are often expensive and not available on the CÉCI clusters

However, you can still perform basic debugging using a standard serial debugger like GDB in batch mode (`--batch`). In this mode, you can provide the commands to execute using the `-ex` option, for example:

```
-ex 'r' -ex 'GDB_COMMAND' -ex 'OTHER_GDB_COMMAND'
```

To run your MPI program and collect a backtrace when it crashes, you can use a command such as:

```
mpirun -np NPROCESS gdb --batch -ex 'r' -ex 'bt' --args EXECUTABLE OPTIONS
```

Memory debugging

The Address Sanitizer (ASan) is a powerful debugging tool that can be enabled at compile time to detect common memory-related errors such as use-after-free, memory leaks, and heap overflows. You can activate it by compiling your program with the flags:

```
-g -fsanitize=address
```

Compared to tools like Valgrind, the Address Sanitizer is significantly faster and therefore practical for use with HPC applications. However, it still introduces a noticeable performance overhead, so it should be used only during debugging and not for production runs

Memory debugging

The Address Sanitizer is particularly useful for diagnosing memory access issues in MPI programs, where traditional debuggers like GDB can sometimes produce misleading backtraces. For example, a segmentation fault may appear to originate from within `MPI_Finalize`:

```
Thread 1 "mpi_app" received signal SIGSEGV, Segmentation fault.
...
#7  0x00007ffff7de7fcd in ompi_mpi_finalize () from /lib/x86_64-linux-gnu/libmpi.so.40
#8  0x00005555555557fe in main (argc=<optimized out>, argv=<optimized out>) at ./mpi_app.c:159
```

In such cases, the error often stems not from `MPI_Finalize` itself, but from an invalid memory address passed to a previous MPI call. The Address Sanitizer can precisely identify where the invalid access occurred, helping you trace the true origin of the memory error rather than its downstream consequence.

QUESTIONS?