Message Passing Interface (MPI)

Lars Pastewka

Table of contents

Overview

The **Message Passing Interface (MPI)** is the standard approach for distributed-memory parallel programming on HPC systems. While Kokkos handles parallelization within a single computing node (through threads, OpenMP, or GPU execution), MPI enables communication between *processes* – whether they run on the same node or are distributed across many nodes. Each process can run Kokkos-accelerated code independently.

This lecture covers: - MPI fundamentals and basic communication patterns - How to integrate MPI with Kokkos for distributed GPU computing - **GPU-aware MPI** (CUDA-aware and HIP-aware), which enables direct GPU-to-GPU communication without transferring data through host (CPU) memory

References and Documentation

- Open MPI Documentation
- MPI Standard
- Kokkos + MPI Integration
- CUDA-Aware MPI

MPI Fundamentals

What is MPI?

MPI is a message-passing library specification that allows: - Multiple processes to run independently on different nodes or cores - Explicit point-to-point communication between processes - Collective operations (broadcast, reduce, all-gather, etc.) across all processes - Synchronization barriers

Unlike shared-memory threading models (OpenMP, Kokkos::Threads), MPI processes have **completely separate address spaces**. Data must be explicitly sent and received using MPI functions.

MPI follows the **SPMD** (single program, multiple data) model: every process executes the *same* executable, and behavior only branches based on the process's rank (Figure 1). All parallelism is expressed by giving each rank different data to work on.

Basic MPI Program Structure

A minimal MPI program has the following structure:

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

int main(int argc, char* argv[]) {
  // Initialize MPI
  MPI_Init(&argc, &argv);

  // Get process rank and total number of processes
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

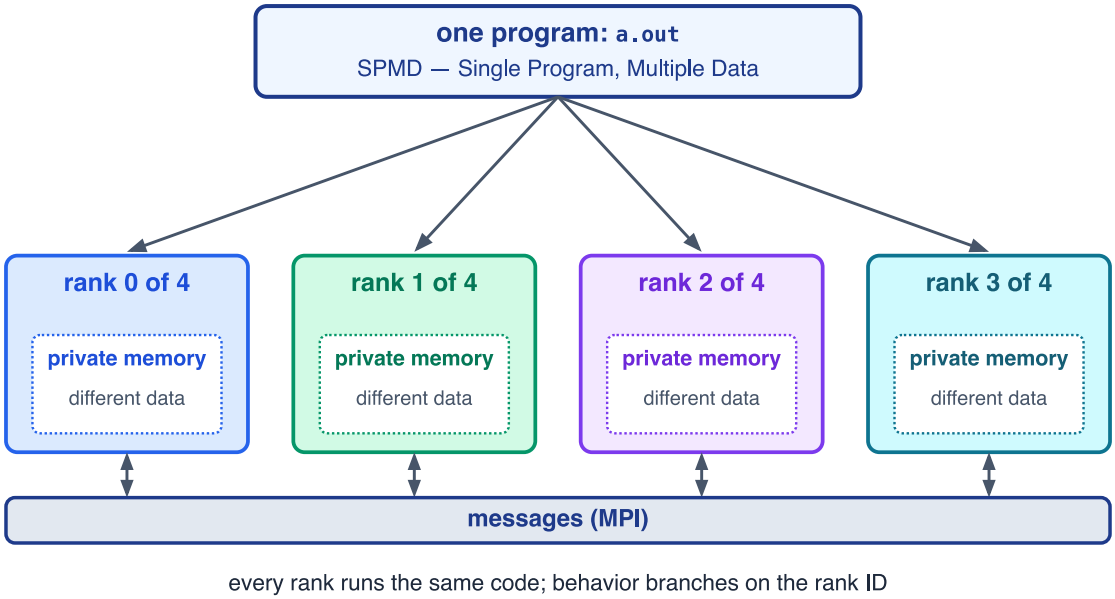

Figure 1: The SPMD execution model: one program is launched as many processes (ranks), each with its own private memory and its own slice of the data; the ranks coordinate exclusively by exchanging messages.

```
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::cout << "Rank " << rank << " out of " << size << std::endl;

  // Finalize MPI
  MPI_Finalize();
  return 0;
}
```

Key MPI Concepts

Communicator: Defines a group of processes that can communicate with each other. `MPI_COMM_WORLD` includes all processes started by the MPI program.

Rank: A unique identifier assigned to each process, numbered from 0 to (size - 1). Think of it as the process ID within your communicator.

Size: The total number of processes running in the communicator.

Point-to-Point Communication

Blocking Send/Receive

The most basic MPI communication uses blocking send (`MPI_Send`) and receive (`MPI_Recv`):

```cpp
// Process 0 sends to process 1
if (rank == 0) {
  double value = 42.0;
  // Send: destination is rank 1, message tag is 0
  MPI_Send(&value, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
  double value;
  // Receive: source is rank 0, message tag is 0
  MPI_Recv(&value, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  std::cout << "Received: " << value << std::endl;
}
```

How does MPI know which receive belongs to which send? Every message carries an **envelope** – source, destination, tag and communicator – alongside its payload of `count` elements of a given datatype, and a receive only matches a send whose envelope agrees with the parameters of the `MPI_Recv` call (Figure 2).

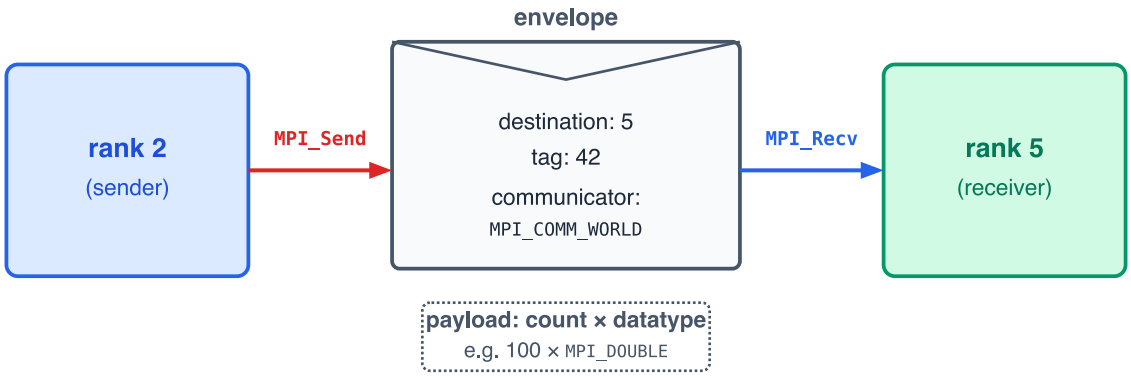

Figure 2: Anatomy of a point-to-point message: the envelope (destination, tag, communicator) determines which `MPI_Recv` the message matches, while the payload is described by a count and a datatype.

Important: Blocking operations do not return until the operation is complete. This can cause **deadlocks** if processes wait for each other. For example, if all processes try to send before receiving, they will all wait forever. Later in this lecture, we'll see how non-blocking operations can help avoid this.

Non-blocking Communication

Non-blocking operations (with names starting with `I` for "immediate") return immediately without waiting for the operation to complete:

```cpp
MPI_Request request;

if (rank == 0) {
  double value = 42.0;
  // Start the send - returns immediately
  MPI_Isend(&value, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);
```

```
  // Can do other work here while message is being sent
  std::cout << "Send started" << std::endl;
  // Wait for send to complete before accessing value again
  MPI_Wait(&request, MPI_STATUS_IGNORE);
} else if (rank == 1) {
  double value;
  // Start the receive - returns immediately
  MPI_Irecv(&value, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &request);
  // Can do other work here while receiving
  std::cout << "Receive started" << std::endl;
  // Wait for receive to complete
  MPI_Wait(&request, MPI_STATUS_IGNORE);
  std::cout << "Received: " << value << std::endl;
}
```

Non-blocking operations are useful when you want to interleave computation and communication. However, you must call `MPI_Wait()` before accessing the data to ensure the operation is complete.

Combined Send and Receive: MPI_Sendrecv

A very common pattern is that each process must *both* send and receive – for example, when exchanging boundary data with a neighbor. `MPI_Sendrecv` performs a send and a receive as a single operation:

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest,   sendtag,
             recvbuf, recvcount, recvtype, source, recvtag,
             comm, status);
```

The MPI library schedules the two halves internally so that matching send/receive pairs on different processes can never block each other (Figure 3). This makes `MPI_Sendrecv` the standard deadlock-free tool for halo exchanges in domain-decomposed solvers, as discussed in the Domain decomposition chapter. Passing `MPI_PROC_NULL` as `dest` or `source` turns the corresponding half into a no-op, which conveniently handles domain boundaries.

Collective Operations

Collective operations involve all processes in a communicator. Figure 4 gives an overview of the most common data-movement patterns before we look at each in code.

Broadcast

Broadcast sends data from one process to all others. All processes must call this function together:

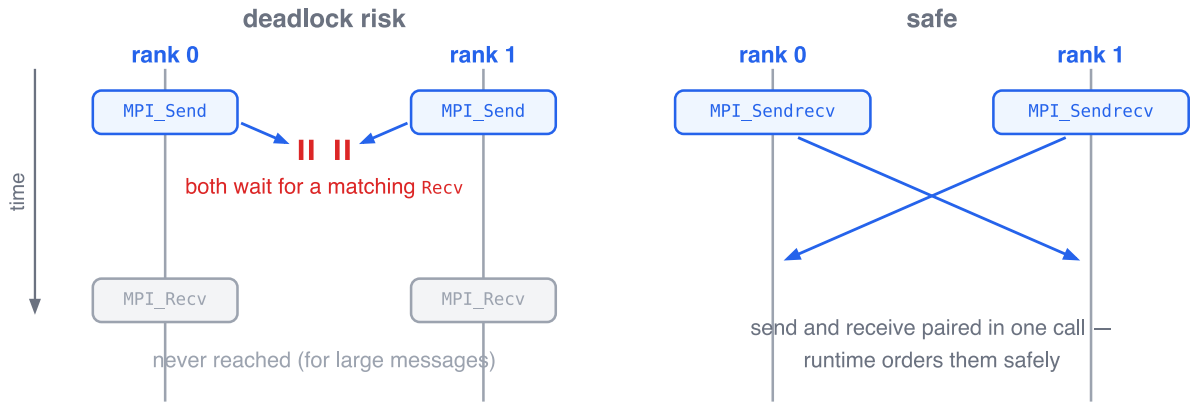

Figure 3: Why `MPI_Sendrecv` is safe: if two processes both call a blocking `MPI_Send` first, each waits for the other's receive and the program deadlocks, whereas `MPI_Sendrecv` lets the library pair the send and receive halves internally so the exchange always completes.

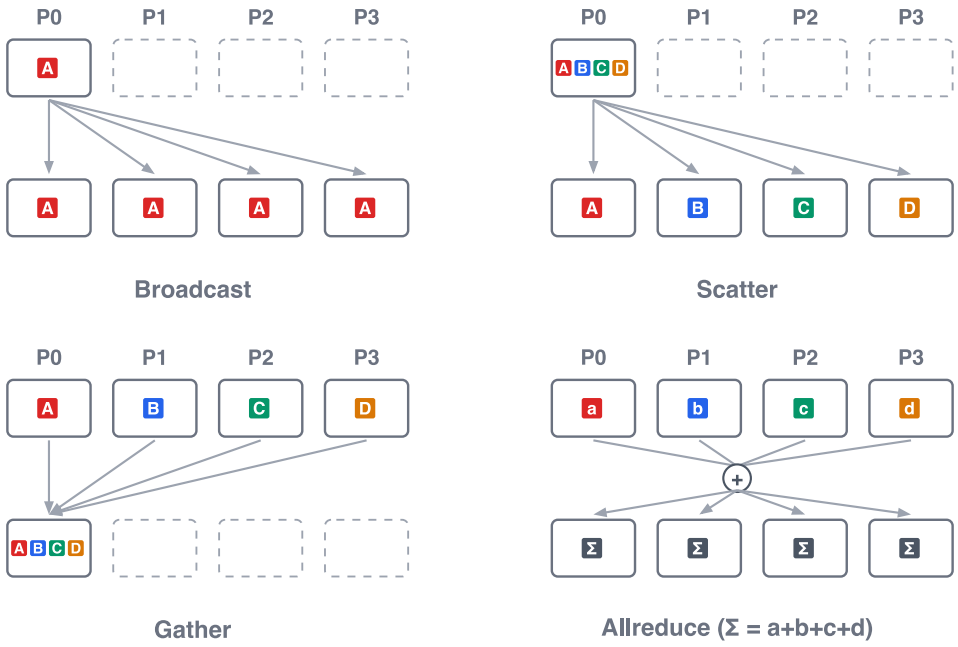

Figure 4: Four common MPI collectives at a glance: broadcast copies one value to all ranks, scatter distributes distinct pieces from the root, gather collects pieces onto the root, and allreduce combines every rank's contribution and gives the result to all ranks.

```cpp
double value;
if (rank == 0) value = 3.14159;
else value = 0.0;  // Other processes need to have variable allocated

// All processes participate: broadcast from rank 0
MPI_Bcast(&value, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Now all processes have value = 3.14159
std::cout << "Rank " << rank << " has value: " << value << std::endl;
```

Notice that **all processes call the same broadcast function**. This is different from point-to-point communication where only sender and receiver participate.

Reduction

Reduction combines data from all processes using an operation like sum, maximum, minimum, etc. The result is gathered on a single process (the root):

```cpp
double local_value = rank * 2.0;  // Each process has a local value
double global_sum = 0.0;

// Combine all local values using sum operation
// Result appears only on rank 0
MPI_Reduce(&local_value, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

// Only rank 0 has the result
if (rank == 0) {
  // With 4 processes: 0*2.0 + 1*2.0 + 2*2.0 + 3*2.0 = 12.0
  std::cout << "Global sum: " << global_sum << std::endl;
}
```

Common reduction operations include `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD` (product).

All-Gather

All-gather collects data from every process and distributes the result to all processes:

```cpp
int local_value = rank * 10;
std::vector<int> all_values(size);  // Need space for data from all processes

// Each process sends its local_value, receives from all others
MPI_Allgather(&local_value, 1, MPI_INT, all_values.data(), 1, MPI_INT, MPI_COMM_WORLD);

// Now all processes have the same array: [0, 10, 20, 30, ...]
// (assuming 4 processes with ranks 0, 1, 2, 3)
for (int i = 0; i < size; i++) {
  std::cout << "Rank " << rank << " sees: all_values[" << i << "] = " << all_values[i] << st
}
```

The key difference from `MPI_Reduce`: all-gather distributes the result to all processes, while reduce only gives the result to one (the root) process.

Worked Example: Monte-Carlo Estimate of Pi

A classic example that exercises exactly these collectives is the Monte-Carlo estimate of π. Draw N points (x_i, y_i) uniformly from the unit square $[0, 1]^2$ and count how many fall inside the unit circle. Since the quarter circle covers a fraction $\pi/4$ of the square, the count yields an estimator for π:

$$\pi \approx \frac{4}{N} \sum_{i=1}^{N} \Theta(1 - (x_i^2 + y_i^2)) \,, \tag{1}$$

where Θ is the Heaviside step function – it contributes 1 if the point lies inside the circle and 0 otherwise (Figure 5).

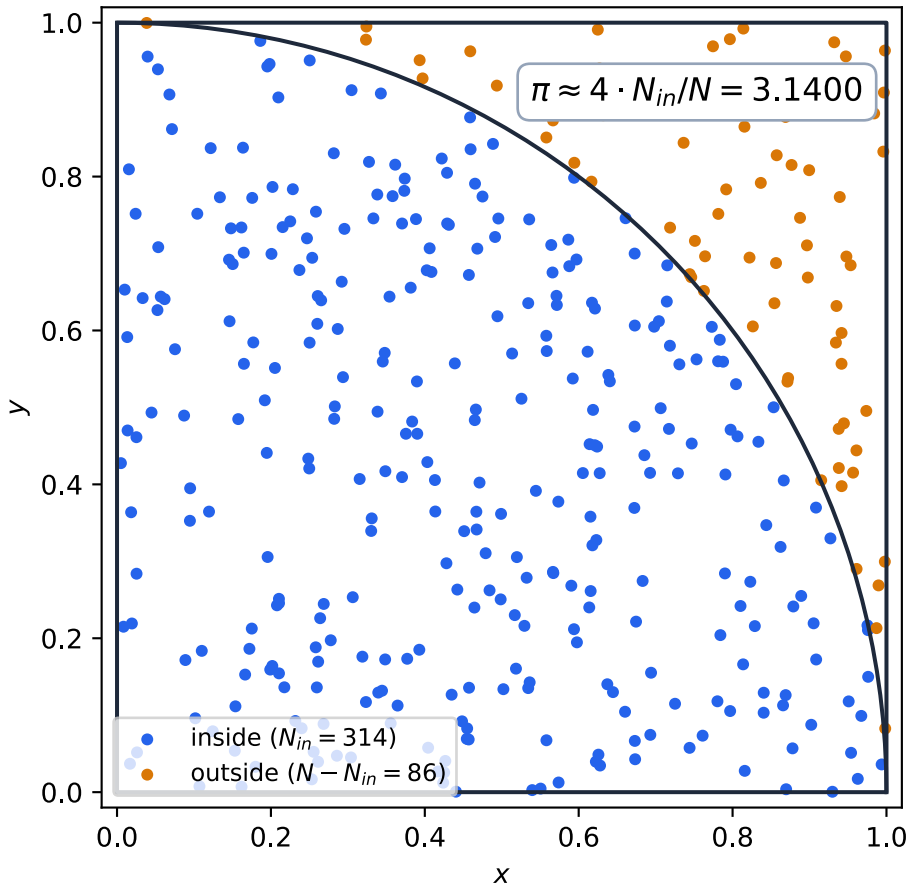

Figure 5: Monte-Carlo estimate of π: sample points uniformly in the unit square and count the fraction that lands inside the quarter circle, which covers an area of $\pi/4$.

This problem parallelizes perfectly: every sample is independent of every other sample, so each rank can simply generate and test its own share of the N points without any communication – a pattern called **embarrassingly parallel**. (Make sure each rank seeds its random number generator differently, e.g. with its rank, so the ranks do not all draw the same points.) The only communication is the sum in Equation 1, which is precisely a **reduction**: `MPI_Reduce` if only one rank needs the result, `MPI_Allreduce` if all do.

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

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

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

  // Each rank generates its own samples with its own seed
  const int nb_local_trials = 100000000 / size;
  std::mt19937_64 rng(rank);
  std::uniform_real_distribution<double> uniform(0, 1);

  // Rank-local count of points inside the unit circle
  int local_hits = 0;
  for (int i = 0; i < nb_local_trials; ++i) {
    const double x = uniform(rng), y = uniform(rng);
    if (x * x + y * y < 1) local_hits++;
  }

  // Sum the counts over all ranks; result lands on rank 0
  int hits = 0;
  MPI_Reduce(&local_hits, &hits, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  if (rank == 0) {
    const long nb_trials = static_cast<long>(nb_local_trials) * size;
    std::cout << "pi = " << 4.0 * hits / nb_trials << std::endl;
  }

  MPI_Finalize();
  return 0;
}
```

The full example, which also handles a sample count that is not divisible by the number of ranks, is in monte_carlo_pi.cpp.

How expensive is the final reduction? If rank 0 collected the partial sums one by one, the s contributions would arrive in $s-1$ sequential steps. A **tree reduction** instead combines pairs of ranks in parallel – ranks reduce in pairs, then pairs of pairs, and so on – finishing in $\lceil \log_2 s \rceil$ steps. You do not have to implement this yourself: MPI implementations use tree-based (or better) algorithms internally, which is one good reason to call `MPI_Reduce` rather than hand-rolling a loop over `MPI_Recv`.

> **Note**
>
> Floating-point addition is not associative: $(a+b)+c$ and $a+(b+c)$ can differ in the last bits. Since the order in which a reduction combines contributions depends on the number of ranks and the internal algorithm, floating-point reduction results may differ in the last bits

between runs with different rank counts. The integer count in this example is exact, but keep this in mind when you reduce floating-point quantities (such as total mass or kinetic energy) and compare runs bit by bit.

MPI with Kokkos

Basic Integration

When using MPI and Kokkos together, keep these principles in mind: 1. Initialize and finalize **both Kokkos and MPI** properly 2. Each MPI process runs its own independent Kokkos kernels 3. Use MPI functions to communicate between processes 4. Call `Kokkos::fence()` before MPI operations to ensure GPU work is complete

```
#include <mpi.h>
#include <Kokkos_Core.hpp>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);
  Kokkos::initialize(argc, argv);

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

    // Each process runs Kokkos kernels independently
    int N = 1000;
    Kokkos::View<double*> data("data", N);

    // Fill with rank-dependent data
    Kokkos::parallel_for(N, KOKKOS_LAMBDA(int i) {
      data(i) = rank + i * 0.1;
    });
    Kokkos::fence();  // Important: ensure kernel completion before MPI

    // Communicate data between processes
    double local_sum = 0.0;
    Kokkos::parallel_reduce(N, KOKKOS_LAMBDA(int i, double& sum) {
      sum += data(i);
    }, local_sum);

    double global_sum;
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
      std::cout << "Global sum: " << global_sum << std::endl;
    }
  }

  Kokkos::finalize();
```

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

Important Synchronization Pattern

This section is critical for GPU computing. **Always call `Kokkos::fence()` before MPI operations** to ensure all GPU kernels have completed:

```
// Run Kokkos kernels on GPU
Kokkos::parallel_for(N, KOKKOS_LAMBDA(int i) {
  // GPU work happens asynchronously
  data(i) = i * 2.0;
});

// CRITICAL: wait for GPU to finish all work
Kokkos::fence();

// Now safe to use MPI with GPU data
// The data is guaranteed to be ready
MPI_Send(data.data(), N, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
```

Why is this necessary? GPU kernels execute asynchronously—they launch quickly and return control to your CPU code immediately, while the GPU continues working in the background. Without `Kokkos::fence()`, your `MPI_Send` might try to send data that hasn't finished being computed yet, resulting in incorrect communication or corrupted data.

GPU-Aware MPI (CUDA-Aware and HIP-Aware)

The Problem: GPU Data Transfer

Without GPU-aware MPI, GPU-to-GPU communication requires staging data through CPU memory:

```
GPU 0 Memory → (cudaMemcpy to CPU) → CPU RAM → MPI Network → CPU RAM → (cudaMemcpy to GPU) -
```

This approach has several problems: 1. **Extra copy overhead**: Data must be copied from GPU to CPU, then sent, then copied back to GPU 2. **Limited CPU memory bandwidth**: The CPU memory system may become a bottleneck 3. **Network interface limitation**: Traditional network interfaces cannot directly access GPU memory

The Solution: GPU-Aware MPI

GPU-Aware MPI allows direct GPU-to-GPU communication without CPU staging:

```
GPU 0 Memory → MPI Network → GPU 1 Memory (direct)
```

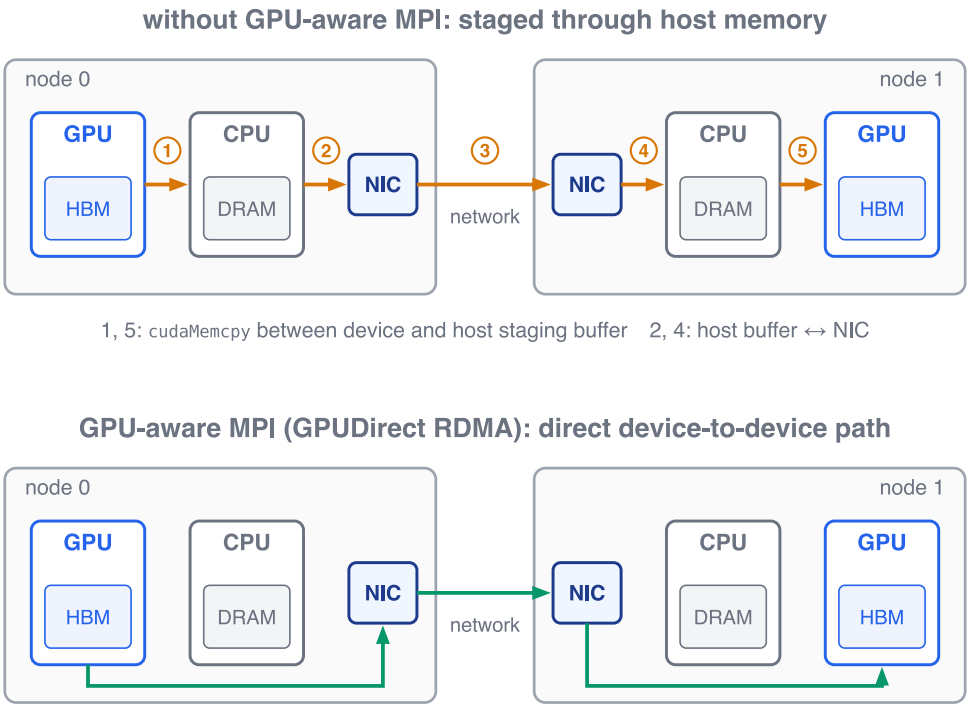

Figure 6: Data path of a GPU-to-GPU MPI message: without GPU-aware MPI the payload is staged through host memory with a `cudaMemcpy` on each side, while GPU-aware MPI with GPUDirect RDMA lets the network adapter read and write GPU memory directly, eliminating both copies.

Figure 6 contrasts the two data paths.

For GPU-aware MPI to work, you need: 1. **MPI library compiled with GPU support**: Either CUDA-aware (for NVIDIA) or HIP-aware (for AMD) 2. **Network interface that supports GPU memory access**: Modern high-performance networks like InfiniBand support this 3. **GPU pointers passed directly to MPI functions**: Your Kokkos views contain GPU pointers that MPI can use directly

Checking GPU-Aware MPI Support

Open MPI provides query functions for this in its extension header `<mpi-ext.h>`.

For CUDA-Aware MPI:

```
#include <mpi-ext.h>  // Open MPI extensions

// Check if Open MPI is CUDA-aware
#ifdef MPIX_CUDA_AWARE_SUPPORT
  if (MPIX_Query_cuda_support() == 1) {
    std::cout << "CUDA-aware MPI is available" << std::endl;
  }
#endif
```

For ROCm-Aware MPI (AMD GPUs):

```
#include <mpi-ext.h>  // Open MPI extensions

// Open MPI calls HIP-aware support "ROCm-aware"
#ifdef MPIX_ROCM_AWARE_SUPPORT
  if (MPIX_Query_rocm_support() == 1) {
    std::cout << "ROCm-aware MPI is available" << std::endl;
  }
#endif
```

Example: CUDA-Aware MPI with Kokkos

```
#include <mpi.h>
#include <Kokkos_Core.hpp>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);
  Kokkos::initialize(argc, argv);

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

    if (size < 2) {
      std::cerr << "Need at least 2 processes" << std::endl;
```

```
      // MPI_Abort terminates all processes cleanly; a plain `return`
      // would skip Kokkos::finalize() and MPI_Finalize()
      MPI_Abort(MPI_COMM_WORLD, 1);
    }

    // Create GPU-resident data using Kokkos
    int N = 10000;
    Kokkos::View<double*, Kokkos::CudaSpace> gpu_data("gpu_data", N);

    // Initialize with rank-dependent data
    Kokkos::parallel_for(N, KOKKOS_LAMBDA(int i) {
      gpu_data(i) = rank + i * 0.01;
    });

    // Wait for GPU kernel to complete
    Kokkos::fence();

    // Now we can directly communicate GPU-resident data.
    // With CUDA-aware MPI, this pointer is GPU memory
    MPI_Request request;

    if (rank == 0) {
      // Send GPU data directly - no CPU staging needed
      MPI_Isend(gpu_data.data(), N, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);
    } else if (rank == 1) {
      // Receive directly into GPU memory
      MPI_Irecv(gpu_data.data(), N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &request);
    }

    // MPI_Wait completes the MPI transfer. The Kokkos::fence() above was needed
    // so that the kernel's writes to the buffer were visible before MPI used it.
    MPI_Wait(&request, MPI_STATUS_IGNORE);

    std::cout << "Rank " << rank << " received: " << gpu_data(0) << std::endl;
  }

  Kokkos::finalize();
  MPI_Finalize();
  return 0;
}
```

Performance Considerations

To judge the benefit of GPU-aware MPI, it helps to know the typical orders of magnitude of communication cost on current HPC systems (exact numbers vary with hardware, MPI implementation and configuration):

Quantity	Typical order of magnitude
Point-to-point latency, small message (intra-node)	~0.3-1 s
Point-to-point latency, small message (inter-node, InfiniBand)	~1-2 s
Bandwidth per InfiniBand NDR link	~25-50 GB/s
Time to transfer 1 GB between nodes	~20-40 ms

Staging GPU data through the host adds a device-to-host and a host-to-device copy, each limited by the CPU-GPU interconnect (PCIe or NVLink), plus extra latency for small messages. GPU-aware MPI removes these copies – with GPUDirect RDMA the network adapter reads GPU memory directly – which can roughly halve large-message transfer times and frees the CPU to do other work. These are typical orders of magnitude; measure on your own system (see the benchmarking notes).

HIP-Aware MPI (AMD GPUs)

Installation

For AMD GPUs with HIP, you need HIP-aware (ROCm-aware) MPI. Install with:

```
# On systems with ROCm (module names are site-specific and vary by cluster)
module load rocm mpi/openmpi/5.0-rocm

# Or manually build Open MPI with HIP support
cd openmpi
./configure --with-pmix --with-hip=/opt/rocm
make -j $(nproc)
sudo make install
```

HIP-Aware MPI Example

The code is nearly identical to CUDA-aware, just with HIP memory spaces:

```
#include <mpi.h>
#include <Kokkos_Core.hpp>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);
  Kokkos::initialize(argc, argv);

  {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Create HIP-resident data
    int N = 10000;
    Kokkos::View<double*, Kokkos::HIPSpace> gpu_data("gpu_data", N);

    // Initialize on GPU
    Kokkos::parallel_for(N, KOKKOS_LAMBDA(int i) {
      gpu_data(i) = rank + i * 0.01;
    });
    Kokkos::fence();

    // Use HIP-aware MPI to communicate
    if (rank == 0) {
      MPI_Send(gpu_data.data(), N, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
      MPI_Recv(gpu_data.data(), N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    Kokkos::fence();
  }

  Kokkos::finalize();
  MPI_Finalize();
  return 0;
}
```

Building with MPI and Kokkos

CMakeLists.txt Example

```
cmake_minimum_required(VERSION 3.21)
project(MPIKokkos LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Find MPI
find_package(MPI REQUIRED)

# Find Kokkos (compiled with CUDA support)
find_package(Kokkos REQUIRED)

add_executable(mpi_kokkos mpi_kokkos.cpp)
target_link_libraries(mpi_kokkos PRIVATE MPI::MPI_CXX Kokkos::kokkos)
```

Compiling on HPC Cluster

```
# Load modules (names vary by cluster; check `module avail`)
module load compiler/gnu mpi/openmpi

# Configure Kokkos with CUDA support
cd kokkos
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release \
      -DKokkos_ENABLE_CUDA=ON \
      -DKokkos_ARCH_AMPERE80=ON \
      ..
cmake --build . -j $(nproc)
cmake --install . --prefix ~/kokkos-install

# Build your project
cd ../..
mkdir build && cd build
cmake -DKokkos_DIR=~/kokkos-install/lib/cmake/Kokkos ..
cmake --build . -j $(nproc)
```

Applying MPI to Grid Solvers

How all of this machinery is applied to a distributed grid solver – ghost cells, halo exchange, and the decomposition of a 2D lattice across ranks, as needed for milestone 6 – has its own chapter: Domain decomposition.

Common Pitfalls

1. Forgetting Kokkos::fence() Before MPI

As explained in the synchronization section above, GPU kernels run asynchronously: launching a kernel and immediately handing the same buffer to `MPI_Send` transmits data that may not have been computed yet, and the receiver gets garbage. Always call `Kokkos::fence()` between the kernel and the MPI call. This bug is particularly difficult to debug because the program may run fine with small data or on slower GPUs.

2. Deadlock from Blocking Communication

Blocking MPI operations (`MPI_Send`, `MPI_Recv`) can cause deadlocks if processes wait for each other.

Bad (can deadlock with 2+ processes):

```
if (rank == 0) {
  MPI_Send(data, N, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
  MPI_Recv(other_data, N, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
```

```
  MPI_Send(data, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  MPI_Recv(other_data, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
// With 2 processes: rank 0 sends to 1, rank 1 sends to 0
// If the network buffer is small, both processes block on their MPI_Send
```

Good (use non-blocking to avoid deadlock):

```
MPI_Request send_req, recv_req;
int dest = (rank + 1) % size;  // Ring topology: send to next rank
int src = (rank - 1 + size) % size;  // receive from previous rank

// Both start immediately without blocking
MPI_Isend(data, N, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD, &send_req);
MPI_Irecv(other_data, N, MPI_DOUBLE, src, 0, MPI_COMM_WORLD, &recv_req);

// Wait for operations to complete
MPI_Wait(&send_req, MPI_STATUS_IGNORE);
MPI_Wait(&recv_req, MPI_STATUS_IGNORE);
```

This "ring" communication pattern allows each process to send and receive simultaneously without blocking. Alternatively, `MPI_Sendrecv` (see above) performs the matched send/receive pair in a single, deadlock-free call – as used for the halo exchanges in the Domain decomposition chapter.

3. Not Checking GPU-Aware Support

Not all HPC systems have GPU-aware MPI. Always check if it's available before assuming your GPU pointers work with MPI:

```
bool has_gpu_aware_mpi = false;
#ifdef MPIX_CUDA_AWARE_SUPPORT
  has_gpu_aware_mpi = (MPIX_Query_cuda_support() == 1);
#endif

if (has_gpu_aware_mpi) {
  // Use GPU pointers directly with MPI
  MPI_Send(gpu_data.data(), N, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
} else {
  // Fall back: copy to CPU, send, copy back to GPU
  std::vector<double> host_buffer(N);
  Kokkos::deep_copy(host_buffer, gpu_data);
  MPI_Send(host_buffer.data(), N, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
}
```

This fallback approach ensures your code works on systems without GPU-aware MPI, though it will be slower.

Further Resources

- Open MPI CUDA Support
- NVIDIA GPU-Aware MPI
- Kokkos MPI Integration
- MPI for Beginners (tutorial)
- bwUniCluster Documentation