

# ENHANCING TYPE SAFETY IN MPI WITH RUST

Nafees Iqbal & Jed Brown  
University of Colorado Boulder  
WAMTA 2025



# Why Type Safety in MPI Matters?

```
1  int x[4] = {1, 2, 3, 4};
2  MPI_Request send_req, recv_req;
3
4  MPI_Isend(x, 4, MPI_INT, right, tag, comm, &send_req);
5
6  float buf[4]; // Different type
7
8  MPI_Irecv(buf, 4, MPI_FLOAT, left, tag, comm, &recv_req);
9  MPI_Wait(&send_req, MPI_STATUS_IGNORE);
10 MPI_Wait(&recv_req, MPI_STATUS_IGNORE);
```

Fig: Type Mismatch in C++

# Why C/C++ Falls Short in Type Safety and How Rust Prevent it?

```
std::vector<int> v {10, 11};  
int *vptr = &v[1]; // Points to v[1]  
v.push_back(12);    // Reallocation occurs  
std::cout << *vptr; // Bug (use-after-free)
```

Fig: Use-after-free bug in C++

```
let mut v = vec![10, 11];  
let vptr = &v[1]; // Immutable borrow  
v.push(12);  
println!("{}", *vptr); // Compiler error!
```

Fig: Bug is prevented in Rust

# Undefined Behavior (UB) is painful and costly in parallel

- UB is masked by abstraction
- Avoiding UB in Code Review and Continuous Integration (CI) is Nearly Impossible
- Debugging at Scale is Hard
- Nondeterministic Failures Make Debugging Worse

# MPI Is Error-Prone Without Type Safety

## Common MPI Errors in C++

- Users must manually ensure that data types match across ranks.
- MPI does not perform **automatic** type verification, leading to potential type mismatches.
- Without strict type enforcement, type mismatches and memory errors can easily go unnoticed, leading to hard-to-debug issues.

```
int x[] = {1, 2, 3, 4}, buf[4];
MPI_Request reqs, reqr;

MPI_Isend(x, 4, MPI_INT, right, tag, comm, &reqs);
MPI_Irecv(buf, 4, MPI_INT, left, tag, comm, &reqr);

buf[0]; // Data race

MPI_Wait(&reqs, MPI_STATUS_IGNORE);
MPI_Wait(&reqr, MPI_STATUS_IGNORE);
```

Fig: Race Condition in MPI (C++)

# RSMPI: Safe Rust Bindings for MPI

- RSMPI provides **safe and ergonomic Rust bindings** for MPI.
- Ensures compile-time safety, memory safety.
- Prevents **common MPI errors** such as:
  - Type mismatches
  - Data races
  - Buffer overflow

```
let mut buf = [0; 4];
mpi::request::scope(|sc| {
    let reqs = world.process_at_rank(right)
        .immediate_send(sc, &x[0]);

    let reqr = world.process_at_rank(left)
        .immediate_receive_into(sc, &mut buf);

    buf[0]; // Data race (Compiler Error)
    reqs.wait();
    reqr.wait();
});
println!("{:?}", buf);
```

Fig: Ensuring Safe MPI Communication with Rust

# Improving MPI safety for modern languages

## - Jake Tronge, Howard Pritchard, Jed Brown

- Mismatches detected at run-time (hard to trace back to root cause, especially for nonblocking)
- Relies on serialization methods (e.g., **bincode**, **iovec**, and **flat**) to encode and transmit data safely.
- Requires additional **metadata and memory allocations** for each message.
- Errors arise only when the mismatched message is sent/received.

<https://doi.org/10.1145/3615318.3615328>

# Introducing TypedCommunicator in RSMPI

- **Safe** Rust bindings for **MPI**.
- Prevents **type mismatches** across ranks.
- Ensures **compile-time** type checking.

```
let rank = world.rank();

// Create a TypedCommunicator for f32
let typed_comm: TypedCommunicator<f32> = TypedCommunicator::new(world_ref);

if rank == 0 {
    let data_to_send: f32 = 42.5;
    typed_comm.send(&data_to_send, 1, 0);
    // This works as the type matches the communicator's expected type.
} else if rank == 1 {
    let mut data_to_receive: i32 = 0;
    // This will fail to compile because the communicator expects f32, not i32.
    typed_comm.receive(&mut data_to_receive, 0, 0);
}
```

Fig: Type Mismatch is Detected at Compile Time Using TypedCommunicator



# When is Dynamic Checking Required?

- Rust enforces type safety at **compile time**, but **type mismatches can still occur dynamically** if different ranks initialize **TypedCommunicator**.

```
if world.rank() == 0 {
    let typed_comm: TypedCommunicator<f32> = TypedCommunicator::new(&world);
    let data: f32 = 42.0;
    typed_comm.send(&data, 1, 0);
} else if world.rank() == 1 {
    let typed_comm: TypedCommunicator<i32> = TypedCommunicator::new(&world);
    let mut buffer: i32 = 0;
    typed_comm.receive(&mut buffer, 0, 0);
}
```

Fig: Inconsistent Type Across Ranks

```
let typed_comm: TypedCommunicator<f32> = TypedCommunicator::new(&world); // Move outside

if world.rank() == 0 {
    let data: f32 = 42.0;
    typed_comm.send(&data, 1, 0);
} else if world.rank() == 1 {
    let mut buffer: f32 = 0.0; // Must match the communicator type
    typed_comm.receive(&mut buffer, 0, 0);
}
```

Fig: Enforcing Type Consistency Across Ranks

# Ensuring Type Congruence in MPI

- Why Simple Type Equality Is Not Enough?
  - **Type Congruence** ensures compatibility between different data layouts.
  - MPI can safely interpret and manage data with varying structures.

```
let x = [[1.0_f32; 2]; 3];  
typed_comm.send(&x, 1, 0);  
  
let mut buffer = vec![0.0_f32; 6];  
typed_comm.receive(&mut buffer, 0, 0);  
  
let y: [[f32; 3]; 2] = [  
    [buffer[0], buffer[1], buffer[2]],  
    [buffer[3], buffer[4], buffer[5]]  
];
```

Fig: Sending (3x2 Matrix) and Receiving (2x3 Matrix)

# Summary & Takeaways

- C++ lacks type safety, leading to potential bugs.
- Rust + MPI (rsmpi) enforces type safety at compile-time.
- TypedCommunicator ensures safe and efficient MPI communication.
- Type congruence allows flexible data handling while maintaining correctness.



**Thank you!**