Los Alamos
NATIONAL LABORATORY

Improving MPI Safety for Modern Languages

Jake Tronge, Howard Pritchard, Jed Brown
{jtronge, howardp}@lanl.gov, jed.brown@colorado.edu

LA-UR-23-30214

Overview

- Problem: Modern languages (Rust) expect memory and type safety guarantees that are not provided by MPI
- We focus on type mismatching errors
- Present two prototype implementations providing better safety for point-to-point communication
  - Extension to Open MPI
  - UCX-based Rust prototype using three different methods

MPI Type Mismatch

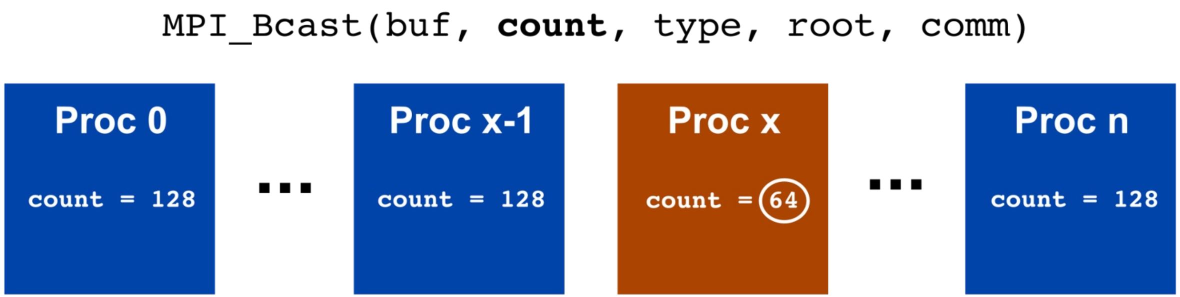
```
if (rank == 0)
    MPI_Send(buf, n, MPI_INT, 1, 0, comm);
else
    MPI_Recv(buf, n, MPI_FLOAT, 0, 0, comm, &status);
```

Why Is Type Matching Important?

- Program complexity makes it more of a problem
- Program correctness
- Modern languages expect type and memory safety

Common MPI Error Types (from Jammer et al. [4])

- Erroneous arguments
- Mismatching arguments (multiple processes)
- Erroneous program flow
- Concurrency (data races)
- Message races (wildcard matching)



Existing Solutions: Correctness Testing Tools

- Static analysis
- Correctness checking and profiling tools [4]

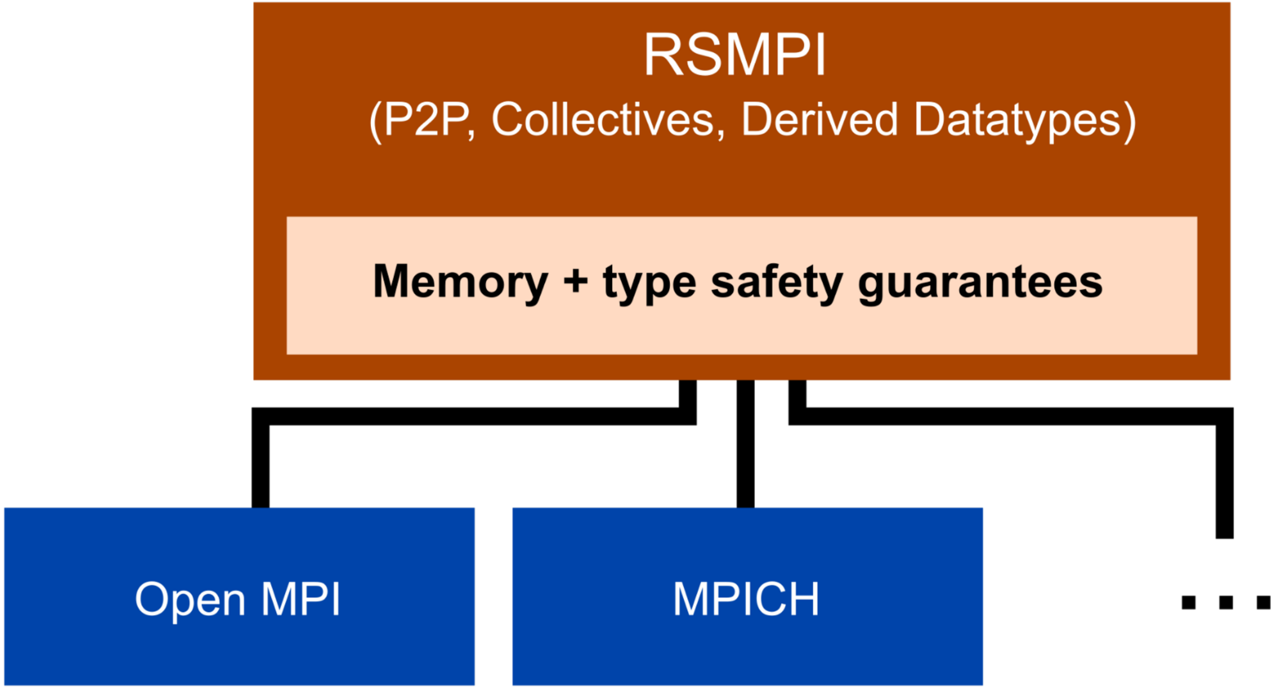
Problems with Correctness Checking Tools

- **Don't provide safety guarantees**
- Cannot catch ephemeral errors — testing tools can interfere with environment [4]
- Differences between programming environments
  - Errors on one system may never happen on another
- Testing for these error types is nearly impossible

The Rust Programming Language

- System-level programming language
- Close to C-level performance with less effort, better error checking [3]
- No garbage collection
- **Guarantees memory safety**
- Dependency management and powerful trait interfaces

Initial Work with Existing Rust MPI Bindings [2]



Underlying Safety Problems of RSMPI

- Some errors are difficult to check without hindering performance
  - At least at the binding level
- In Rust terms, some parts of RSMPI are currently **unsound**

MPI Type Matching Rules (§ 3.3.1 [1])

Typed Values

Typed values require types to match on both the sender and receiver side

Untyped Values
MPI_BYTE

MPI_BYTE can match any type

Packed Data
MPI_PACKED

MPI_PACKED is used for packed data and can match any type

Communication of Typed Values [1]

- Datatypes are required to match on both the sending and receiving side
- **Programs that don’t follow this are considered erroneous**
- Standard doesn’t require implementations to check this
- Note: partial receives are allowed for typed values

Possible Methods for Type Matching

- Use type signature hashing [5]
- Serialization
- Type IDs

```
struct example {
    int a;
    double d[4];
};
```

MPI_INT	MPI_DOUBLE	MPI_DOUBLE	MPI_DOUBLE	MPI_DOUBLE

Two Safe P2P Prototypes

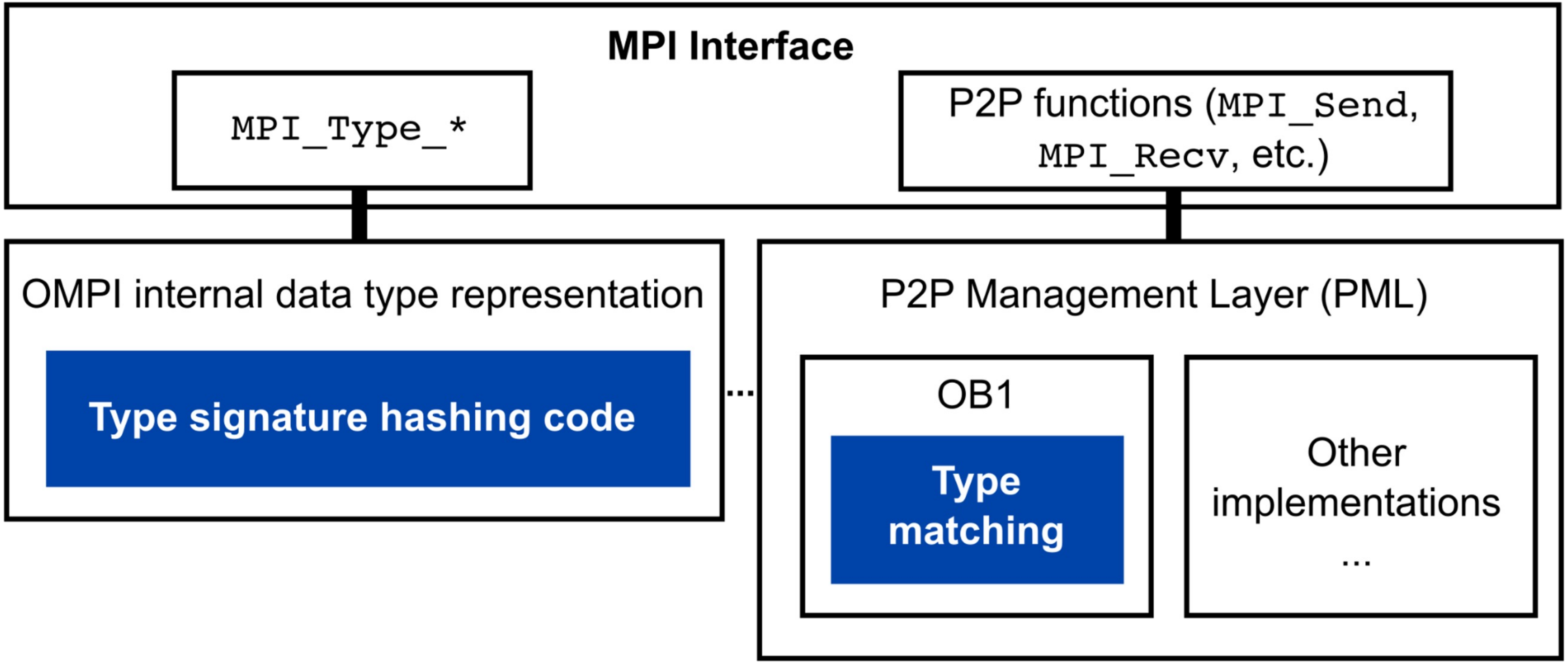
- Type hashing implemented in Open MPI (in C)
- Prototype interface based on UCX (in Rust)

Open MPI Extension

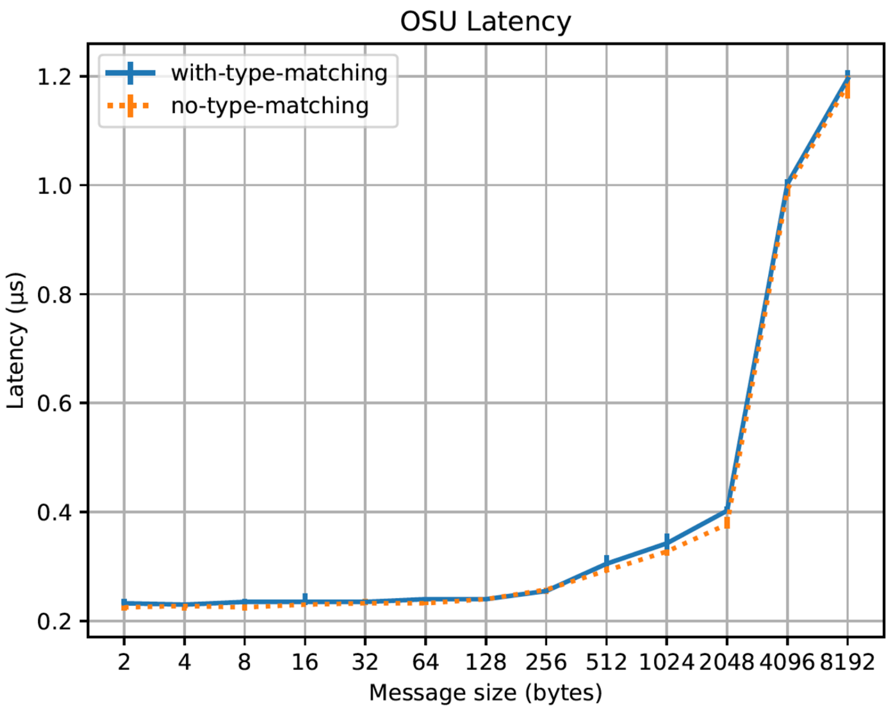
- Implements type hashing based on the type signature [5]
- Validation is done at the PML layer
- MPI_ERR_TYPE is returned on failure
- Note: all results run on a single node

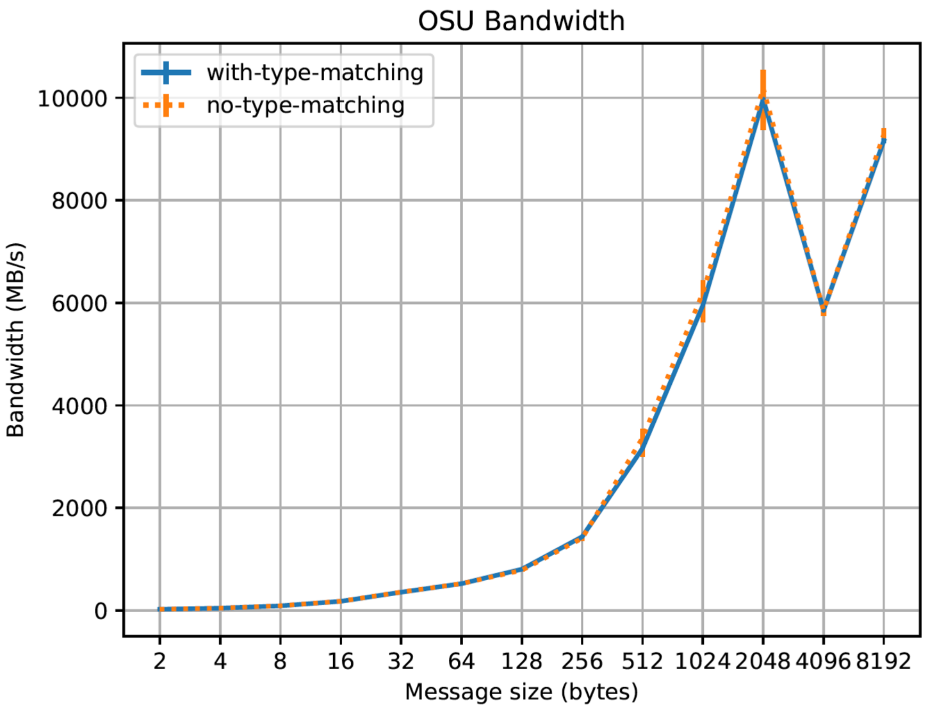
https://github.com/jtronge/ompi/tree/datatype_matching

Open MPI Type Matching Implementation



Open MPI Results





Rust Prototype: Safety for More Complicated Types

- More “Rusty” interface
- More complicated data structures are often used in Rust and other newer languages
- Note: all results run on a single node

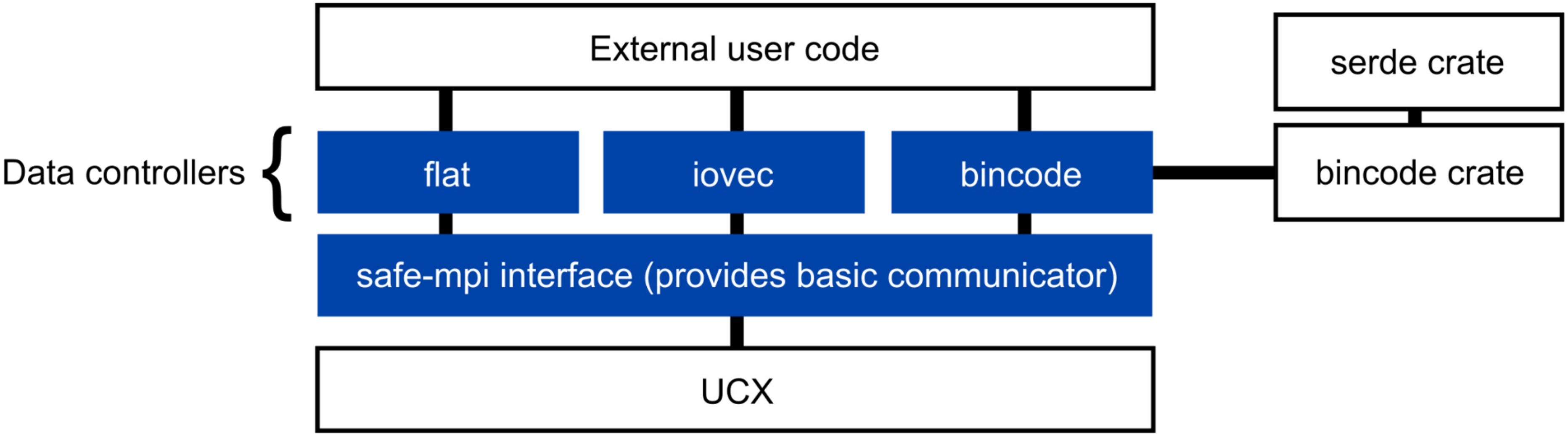
https://github.com/jtronge/safe-mpi/

Testing Data Types

Simple data type: `i32` (32-bit signed integer)

```rust
// complex-noncompound datatype
pub struct ComplexNoncompound {
    i: i32,
    d: f64,
    // Array of 16 IEEE 32 bit floating points
    x: [f32; 16],
}
// complex-compound datatype
pub struct ComplexCompound {
    i: i32,
    d: f64,
    // Heap-allocated array of 16 IEEE 32 bit
    // floating points
    x: Vec<f32>,
}
```

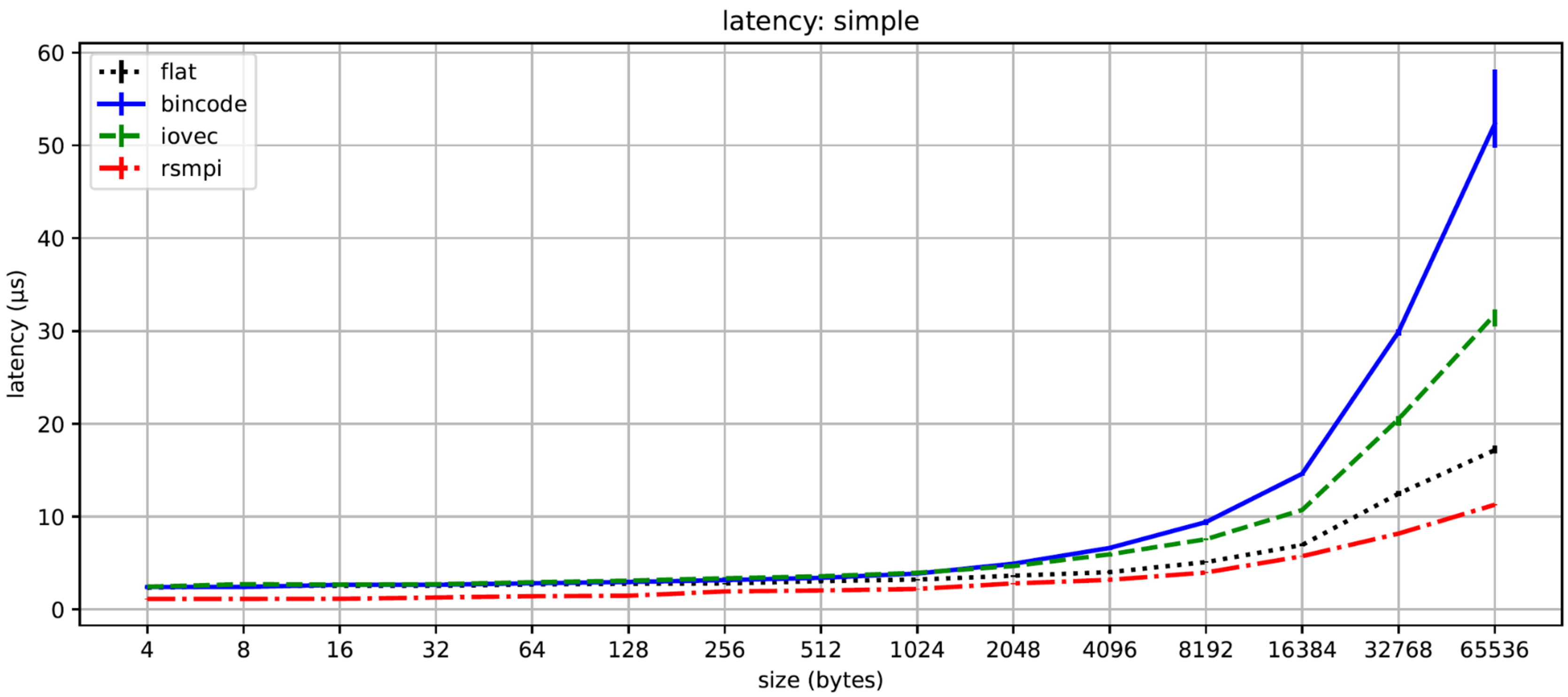
Rust Prototype Design



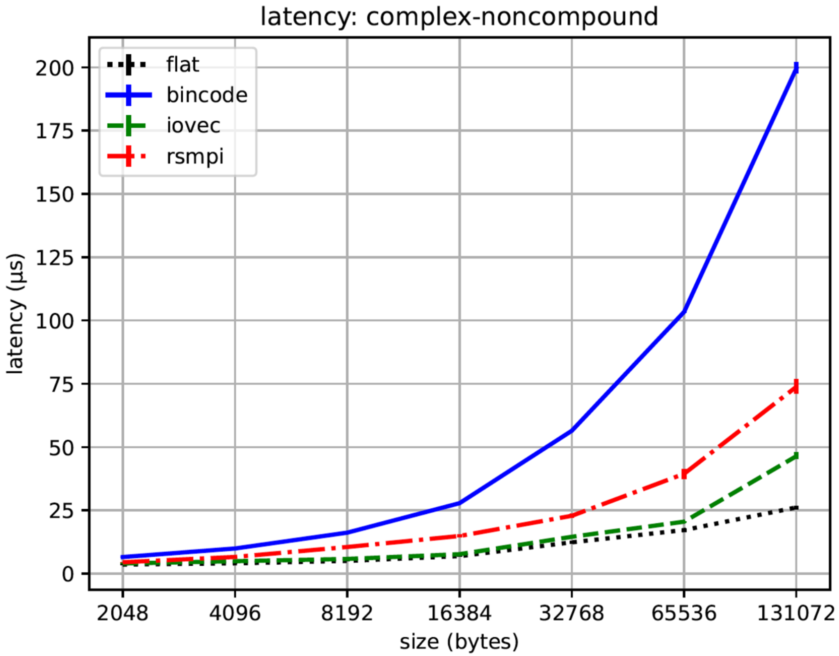
Example Rust API Usage

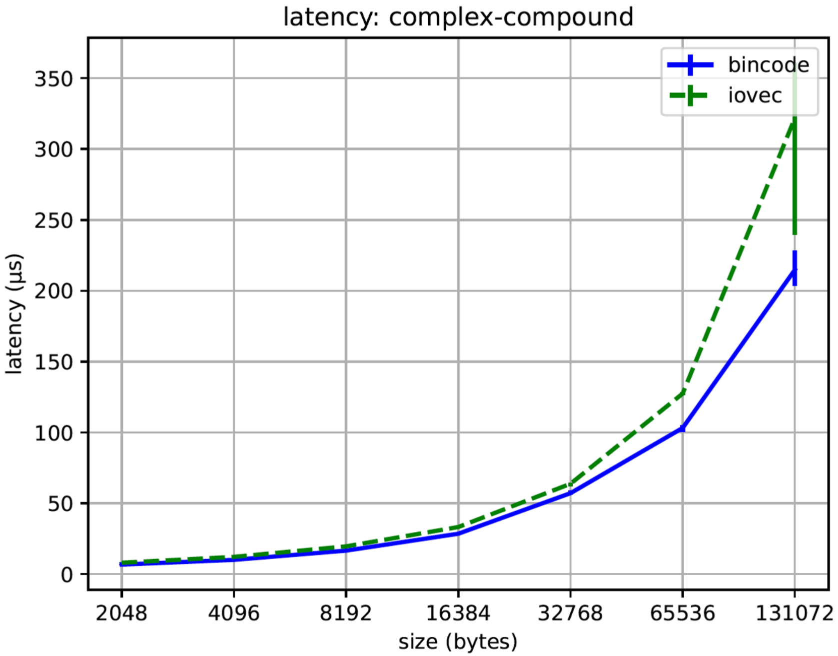
```rust
let sm = safe_mpi::init(sockaddr, args.server).expect("Failed to initialize safe_mpi");
let comm = FlatController::new(sm.world());
...
comm.recv(&mut data[..], 0).unwrap();
```

Latency for Simple Type

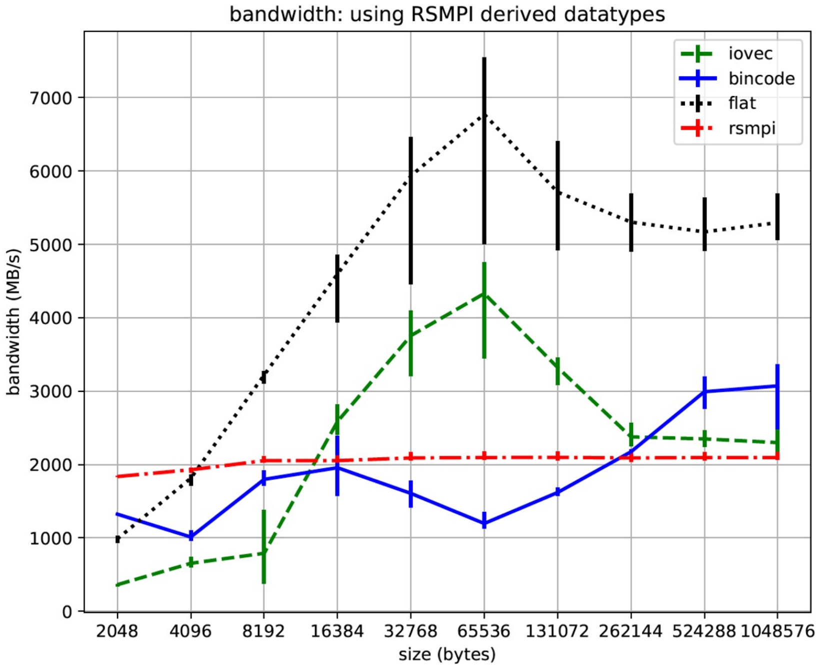


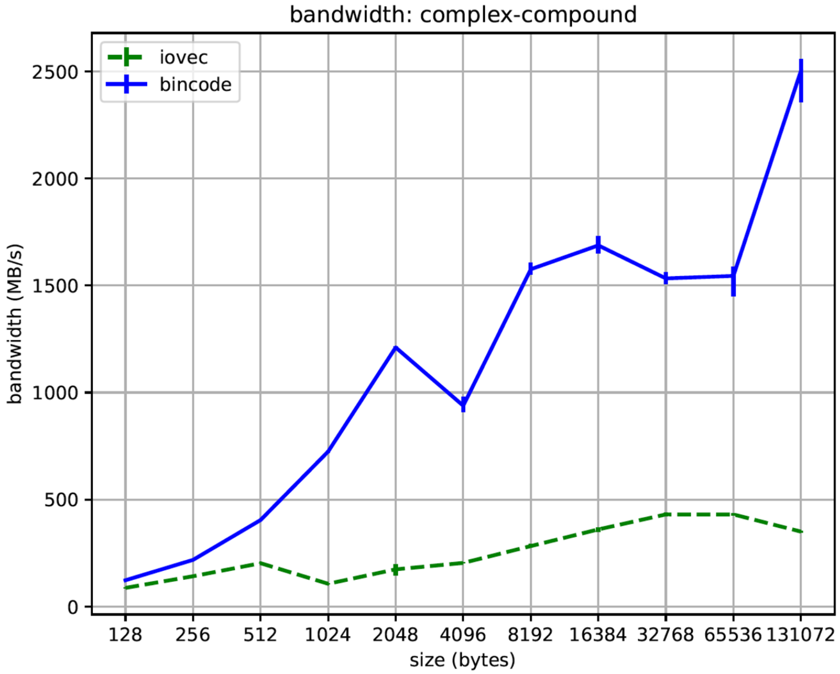
Latency for Complex Types





Bandwidth for Complex Types





Los Alamos
NATIONAL LABORATORY

Limitations of Rust Prototype

- Partial receives
- Collective call support

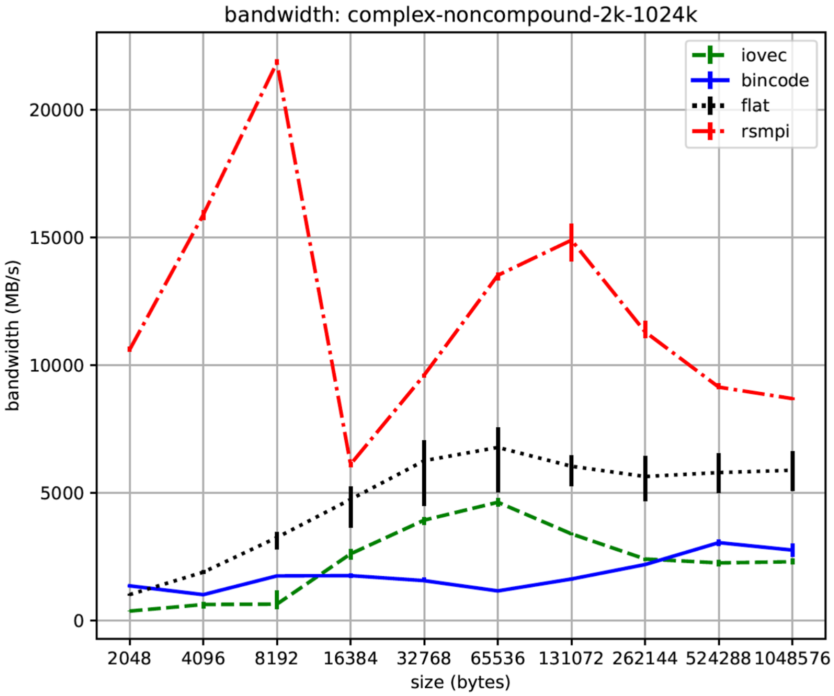
Conclusion: Type Matching Has Minimal Overhead

- P2P safety can be guaranteed with minimal overhead (at least for type signature hashing)
- Rust prototype requires more optimization
- New language-specific interfaces could be useful
- Further work is required for other types of errors (collective argument mismatch, etc.)

References

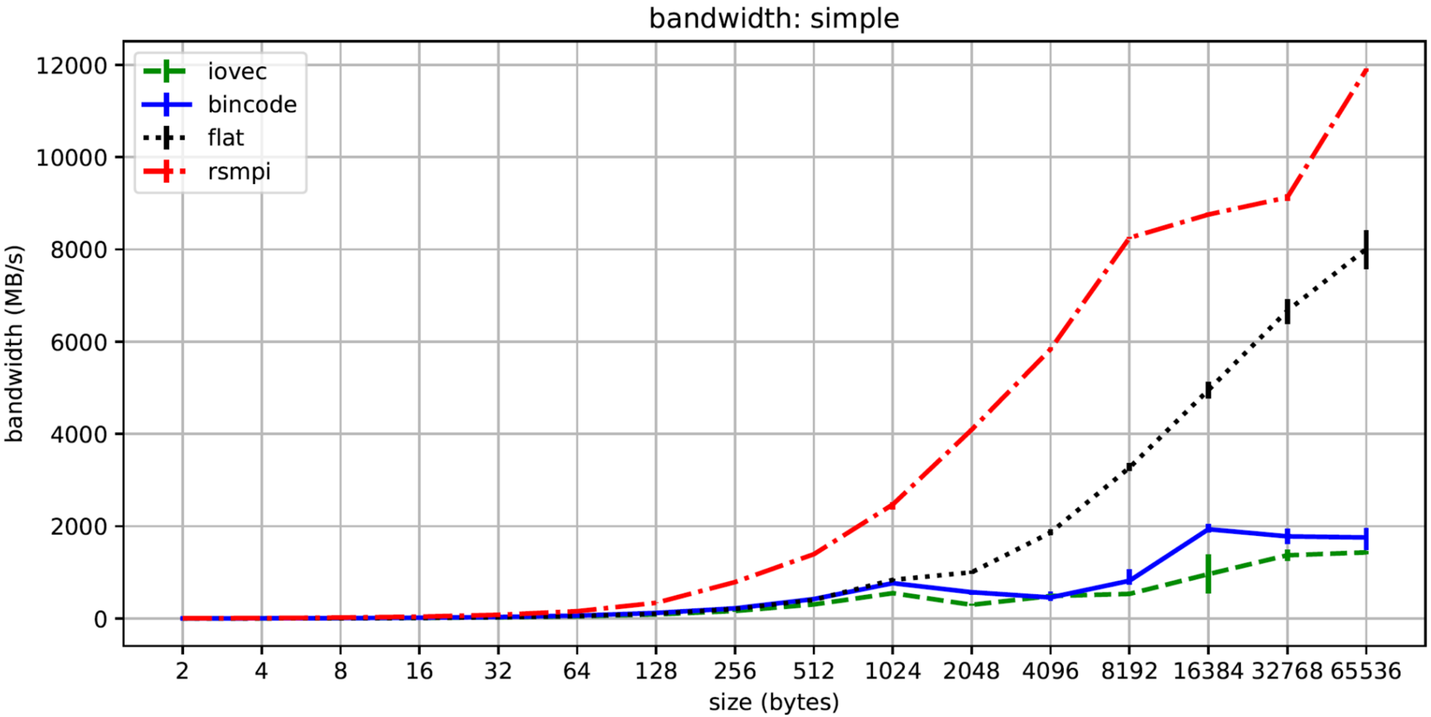
1. Message Passing Interface Forum. June 2021. MPI: A Message-Passing Interface Standard Version 4.0. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
2. RSMPI Developers. 2023. RSMPI: MPI bindings for Rust. https://github.com/rsmpi/rsmpi
3. Manuel Costanzo, Enzo Rucci, Marcelo Naiouf, and Armando De Giusti. 2021. Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body. https://doi.org/10.1109/CLEI53233.2021.9640225
4. Tim Jammer, Alexander Hück, Jan-Patrick Lehr, Joachim Protze, Simon Schwitanski, and Christian Bischof. 2022. Towards a Hybrid MPI Correctness Benchmark Suite. https://doi.org/10.1145/3555819.3555853
5. William Gropp. 2000. Runtime Checking of Datatype Signatures in MPI. https://dl.acm.org/doi/abs/10.5555/648137.746488

Extra Material: RSMPI Bandwidth with MPI_BYTE



*new result not in paper

Extra Material: Bandwidth for Simple Type



*new result not in paper