Improving MPI Memory Safety for Modern Languages

Jake Tronge, Howard Pritchard
{jtronge,howardp}@lanl.gov

What is memory safety?

Memory safety protects a program from errors that could lead to memory corruption and undefined behavior (UB) [9].

MPI and memory safety

For MPI programs to be valid and well-defined, there are many requirements that need to be checked: collective call arguments must match between processes; datatypes must also match with point-to-point messages [1, 2].

Programs that do not follow these rules are considered invalid and memory unsafe. MPI profiling and debugging tools can find a lot of these errors, but not all, especially memory safety errors that only occur under certain conditions.

More recent languages, such as Rust [4], are designed to be memory safe and this makes using MPI as a safe parallel library challenging.

Rust

Provides memory safety using a mix of compiler and runtime checks.

Code is divided into **safe** and **unsafe** blocks.

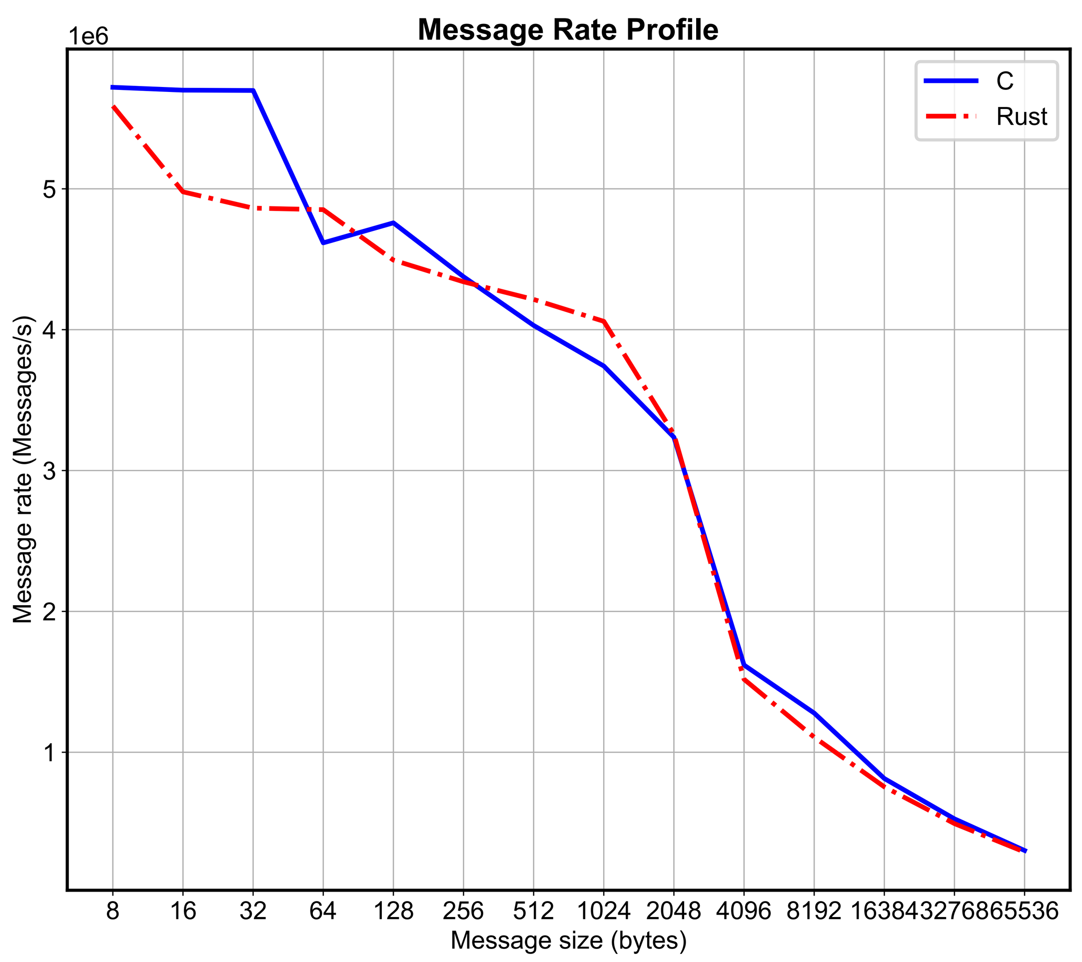
Rust also has a number of special rules and properties that are checked by the compiler:

ownership - all variables are owned by one and only one function

exclusion rule - code can have multiple immutable references to a variable or one mutable reference, but never both at the same time

lifetime analysis - all references have a *lifetime* in which they can be used.

Below is the message rate profile from osu_mbw_mr [5] written in C compared with a version implemented in Rust with RSMPI [7].



Despite being memory safe, Rust has comparable performance to C [8].

Contributions

Implementations of memory safe point-to-point (P2P) messaging in Rust.

Analysis of performance and overhead of the different solutions.

Methods for doing efficient and memory safe messaging with multiple languages

Safe P2P communication

Possible methods: **serialization** or **type IDs** (internal type ID or hash of type signature [3])

We implemented three different designs, all of which are based on UCX [10]:
bincode [6] - a serialization method
flat - uses type IDs with no data preprocessing or packing
iovec - variation of flat allowing for more complicated types

FlatBuffer trait/interface used for messages:

```
pub unsafe trait FlatBuffer: Any {
    /// Size of this buffer in bytes
    fn size(&self) -> usize;
    /// Pointer to the buffer
    fn ptr(&self) -> *const u8;
    /// Mutable pointer to the buffer
    fn ptr_mut(&mut self) -> *mut u8;
    /// Hashed type ID of the inner type
    fn type_id() -> u64;
    /// Number of elements in the buffer (default: 1)
    fn count(&self) -> usize {
        1
    }
}
```

Testing

We wrote latency and bandwidth benchmarks based on the OSU Micro-Benchmarks [5].

RSMPI [7], the existing MPI binding, is used as a baseline.

Different datatypes were used to best show performance of the different methods:

simple:

```
&[i32]
```

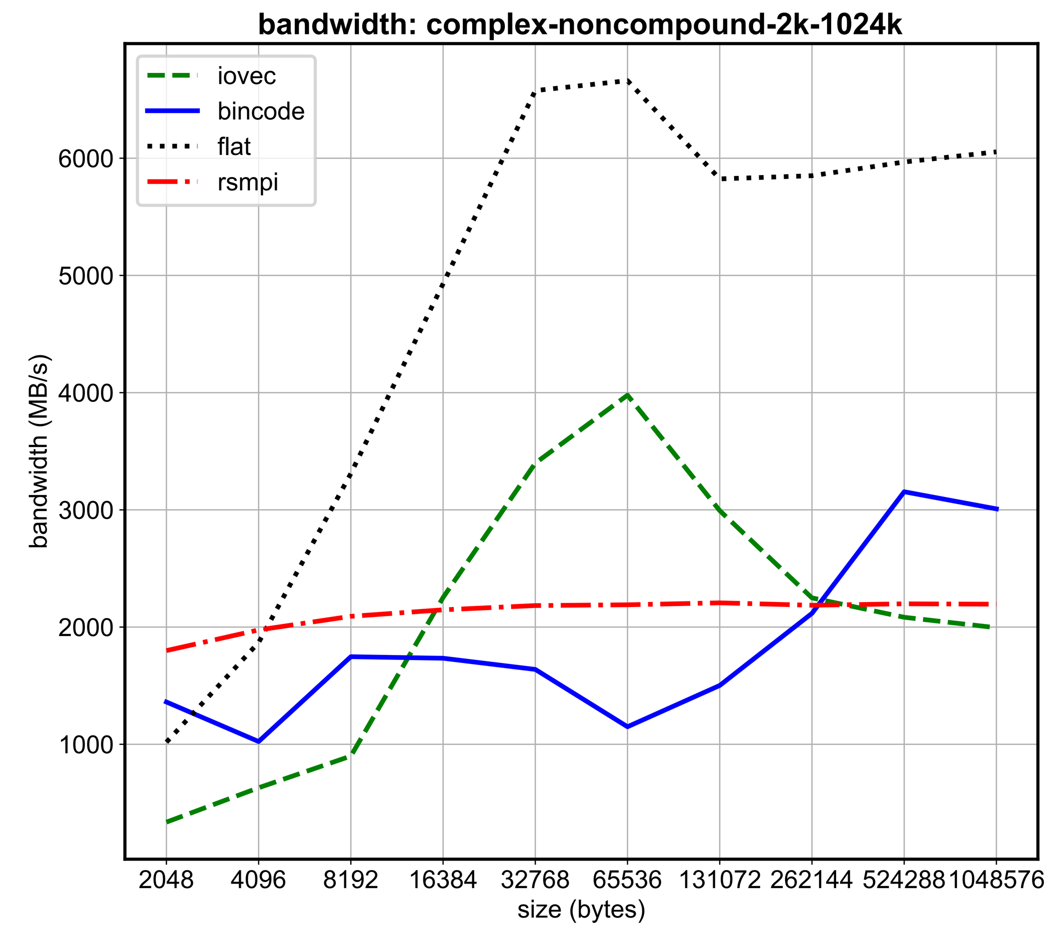
complex-compound:

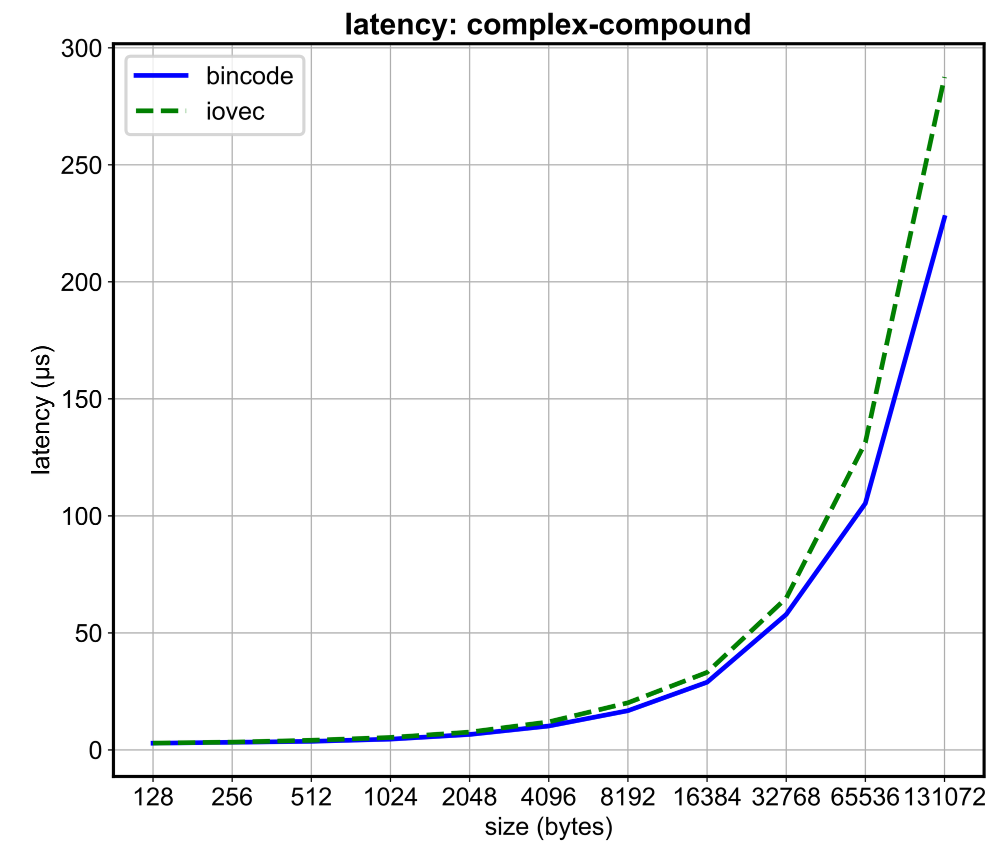
```
pub struct ComplexCompound {
    i: i32,
    d: f64,
    x: Vec<f32>,
}
```

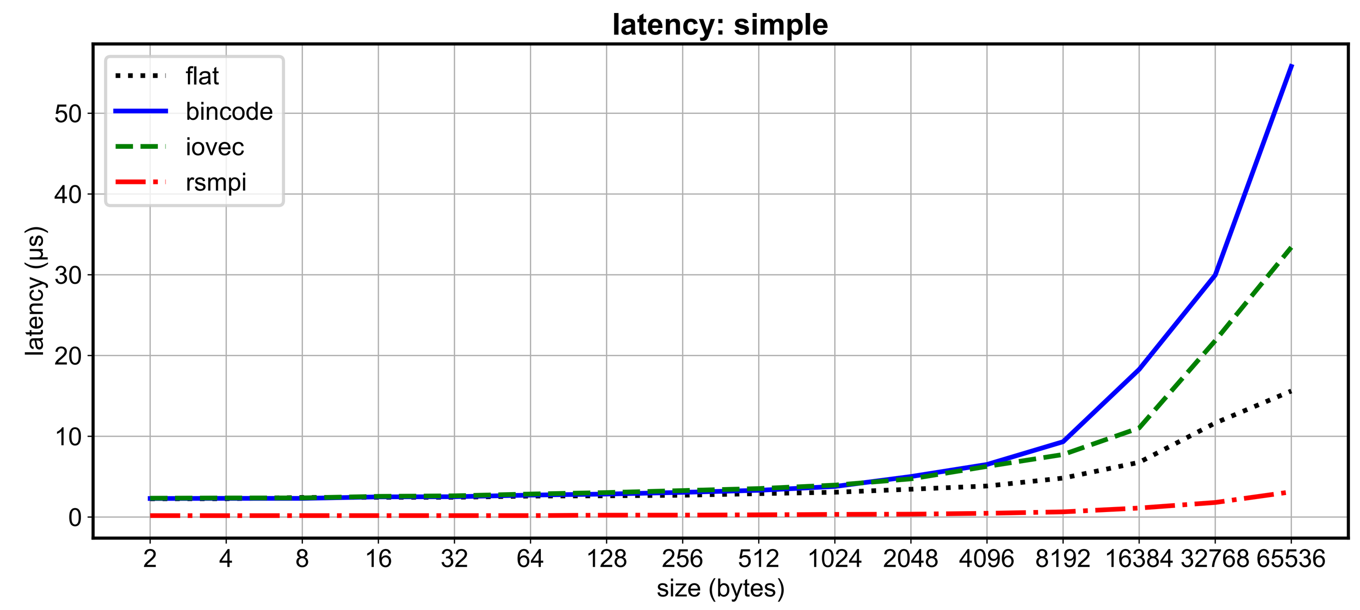
complex-noncompound:

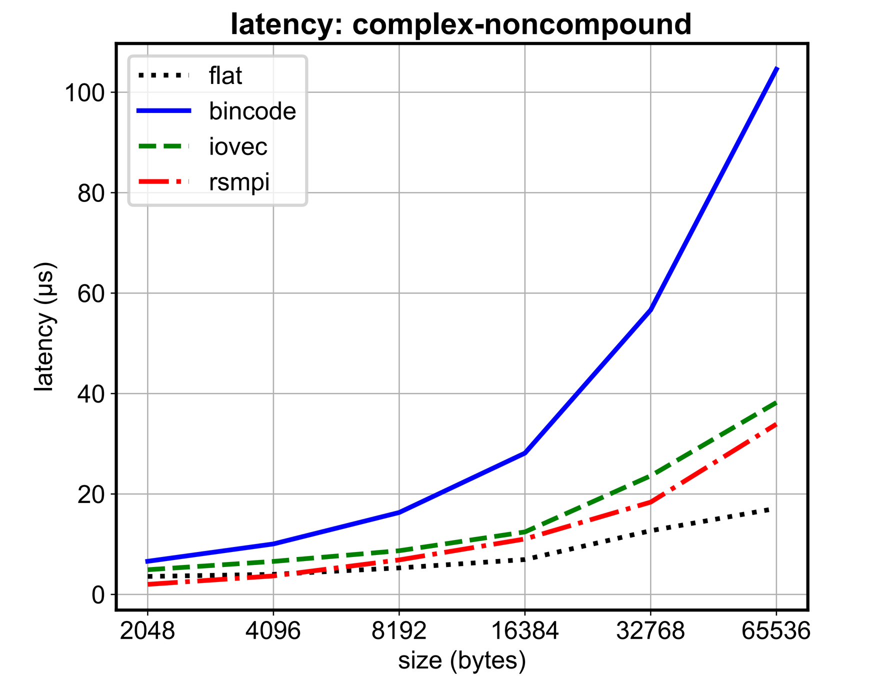
```
pub struct ComplexNoncompound {
    i: i32,
    d: f64,
    x: [f32; 16],
}
```

Results









Conclusion

Improving memory safety can increase usability and decrease development errors in MPI applications.

P2P messaging can be made safe without prohibitive performance loss.

Methods such as serialization work better for more complicated types, but are not as performant.

Type IDs are best used for validating messages with simpler types.

Further research is needed for other areas, such as with collective argument mismatches.

Acknowledgements

Jake Tronge and Howard Pritchard acknowledge support by the National Nuclear Security Administration. Los Alamos National Laboratory is operated by Triad National Security, LLC for the U.S. Department of Energy under contract 89233218CNA000001. LA-UR-23-23979.

References

[1] Message Passing Interface Forum. 2021. MPI: A Message-Passing Interface Standard Version 4.0. Retrieved from https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.
[2] Tim Jammer, Alexander Hück, Jan-Patrick Lehr, Joachim Protze, Simon Schwitanski, and Christian Bischof. 2022. Towards a Hybrid MPI Correctness Benchmark Suite. In Proceedings of the 29th European MPI Users' Group Meeting (EuroMPI/USA'22). Association for Computing Machinery, New York, NY, USA, 46–56. https://doi.org/10.1145/3555819.3555853
[3] William Gropp. 2000. Runtime Checking of Datatype Signatures in MPI. In Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer-Verlag, Berlin, Heidelberg, 160–167.
[4] Rust Community. 2023. The Rust Reference. Retrieved from https://doc.rust-lang.org/reference/index.html.
[5] Dhabaleswar K. Panda. 2023. OSU Micro-Benchmarks 7.1. Retrieved from https://mvapich.cse.ohio-state.edu/benchmarks/.
[6] Bincode. 2023. Bincode. Retrieved from https://github.com/bincode-org/bincode.
[7] RSMPI. 2023. MPI bindings for Rust. Retrieved from https://github.com/rsmpi/rsmpi.
[8] Manuel Costanzo, Enzo Rucci, Marcelo R. Naiouf, and Armando De Giusti. 2021. Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body. Retrieved from https://arxiv.org/abs/2107.11912.
[9] NSA. 2022. Software Memory Safety. Retrieved from https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF.
[10] Unified Communication X. 2023. Unified Communication X. Retrieved from https://openucx.org/.

Los Alamos NATIONAL LABORATORY