Enabling Clang to statically check MPI type safety

Dmitri Gribenko¹,², Alexander Zinenko¹,²

1 High-performance computing center NTUU “KPI”
2 Computer engineering department NTUU “KPI”

gribozavr@gmail.com, a.zinenko@koric.org.ua

Abstract. This paper introduces a static analysis method to check that the actual type of the data buffer and the type specified by MPI library constants match. The method is based on annotations in MPI library header files. Type checking is performed at compile time and has no run-time overhead. The paper also covers implementation of the method in Clang compiler and MPICH2 implementation of MPI.

Keywords

Compiler, Clang, MPI, type safety, language extension, MPICH2, static analysis.

1 Introduction

The C language, one of the most widely used languages for designing parallel computational software that uses MPI interface, has static type checking. This implies that all types to be known to the type checker at the compile time. The C++ language, being its descendant, preserves this trait, while introducing type inference capabilities, more strict implicit type conversions and run-time type information. However, both languages feature pointers enabling direct memory manipulation.¹ One of the key characteristics of pointer types in C++ is the opaque void* type which can represent a pointer to a value of any type. Despite being particularly helpful in shared library design, this feature is proven to be a source of numerous errors [1] that do not cause a compile-time error but appear mainly in the run time causing undefined behavior, crashes or data corruption.

A wide range of libraries and programming interfaces that are oriented towards C language use void* pointers to generalize provided functions. For example, all MPI communication functions use such unsafe data types to represent pointers to buffers with data to be sent/received. Most of these libraries also require passing an additional parameter which actually specifies the data type as an implementation-dependant value. (e.g., MPI_Datatype parameters in MPI). Although this provides sufficient information to MPI about in-memory data layout, ensuring that the real buffer type and the provided MPI_Datatype match is programmer’s responsibility and checked by the compiler in any way – by design of void* pointers. For example:

```c
int data[10];
MPI_Send(data, 10, MPI_DOUBLE, ..., ); // type mismatch int/double
```
Such mistakes could lead to erroneous results without any diagnostic or warning from a compiler, nor from the MPI run-time subsystem. But the type information is available at compile time, it is just the unfortunate limitation of C type system that does not allow to check this correspondence directly.

2 Concept

Most MPI data-related functions accept a void* pointer to data buffer and a type tag passed in as a MPI_Datatype value. MPI provides predefined type tag constants for standard primitive types while allowing users to describe their own data types. MPI imposes few constraints on type tag values, treating them as opaque handles, so implementations are free to choose the concrete type themselves. Typically type tags are integer handles or some small structures.

¹Hereafter we are describing C++ even though MPI can be treated as a pure C library. However, the key concept is applicable to both languages.
Given that much freedom in implementation-defined choices, there is no feasible way to use any built-in C++ type checking rules to address the issue. Furthermore, we can make no use of C++ templates since MPI is a pure C interface to ensure compatibility and interoperability with existing libraries and languages.

Modern compilers allow for extending the language by providing some extra information in specially formatted attribute annotations. These annotations being processed at semantic analysis step can be used to carry the necessary information on type correspondence. Attributes are attached to declarations, specifically they can be applied to functions and constants. Therefore, we need to provide modifications in MPI libraries in conjunction with additional semantic analysis rules to enable the required type checking.

MPI implementations are not the unique libraries that make use of void* plus type tag pattern. A broad range of data processing libraries use it to generalize themselves over standard C++ type system. Thus, making attribute-based type checking system applicable beyond MPI implementations is preferable.

Conceptually, we need to provide the following information in the source code for enable type checking:

- each function accepting a void* pointer along with some type tag should be annotated with indexes of these parameters; this also requires some special treatment for variadic functions;
- each constant used as type tag should be annotated with the corresponding type that the pointer value should have.

Given those semantic analysis extracts required arguments, determines the real type of the buffer and compares it with the type stated in tag annotation.

3 Implementation

The concept described above was implemented in Clang compiler [2]. Annotations are represented as GNU-style attributes. Although the recent C++ language standard comprises attributes in order to standardize compiler extensions [3], we decided to use GNU-style attributes to be compatible with C++98 and C89.

Original attribute implementation in Clang based on GNU grammar had no possibility of passing types as attribute arguments. We enhanced the Clang attribute parser to accept attribute lists containing types according to specification of attribute argument list as sequence of tokens with balanced delimiters in the latest C++ standard.

The key features are provided by two attributes, namely pointer_with_type_tag(kind, buffer_idx, type_idx) to annotate functions and type_tag_for_datatype(kind, type, optional) to annotate type tags. The former should be used to annotate all functions the user wants to perform type checking of void* arguments against the specified type tag. It accepts a kind, which we describe later, buffer and one-based type tag indices in the function argument list. The latter attribute accepts the kind as well as the type which has to be a valid type in the C++ type system.

In a call to an annotated function the static type of the object the buffer pointer points to should match the type stated in the datatype annotation. The attribute for describing type tags also accepts an optional argument that specifies exactly how the types should be compared. By default the underlying types have to match exactly, but when marked as layout_compatible, it applies layout compatibility rules of C++11 standard to perform checking which is useful for some MPI “compound” types like MPI_DOUBLE_INT. The optional argument can also enable checking that there is no buffer passed or, put differently, a null pointer value should be passed as the buffer if the must_be_null optional argument is present.

The kind attribute argument was introduced to generalize type safety checks over different libraries using the same pattern. It is used to distinguish between different sets of annotated functions and type tags. For example, MPI implementations use mpi kind.

Type tags can be references to some declared identifier, as it is done in OpenMPI [4] or integer literals in macros like the MPICH2 [5] implementation uses. The latter case, however, requires introducing a named constant since attributes can not be applied to macros.

We also implemented similar type checking functionality for variadic functions by introducing another attribute argument_with_type_tag. This attribute states that some function argument determines the type of some other argument (not the pointee type of that argument). For example, this can be used with fcntl-like functions.

This functionality described above is currently included in the main source trees of Clang compiler [6] and MPICH2 library implementation [7].
The features of the implemented annotations enable type checking for: MPI library implementations (both OpenMPI and MPICH2 approaches to type tags), HDF5 library for data storage (having the same usecase as MPI), and for system calls implemented as variadic functions (ioctl, fcntl etc.)

4 Discussion and conclusion

The C++ attribute system introduced in the recent standard generalizes the language extensions across different compilers and allows for performing better static analysis and finding non-trivial errors during compilation. Its specification enables compiler to parse types as attribute parameters which might help in ensuring type safety.

MPI communication functions have a specific pattern for describing data types of buffers being used. Although the type information is available during the compilation, it could not be checked by means of standard C++. Our extension enables type checking while using specific compilers in conjunction with MPI library implementations without having to modify any user code. The proposed attributes can be implemented in other compilers and library implementations.

Since many libraries used in high-performance computing share the same pattern of using type tag function arguments determining the actual type of other arguments, our implementation is generalized to allow handling all of them. This would require small changes to the source code of those libraries. Our approach allows to eliminate the specific kind of errors in programs that is extremely hard to discover before the program finished with incorrect results, which, in case of computationally-heavy MPI applications, would waste lots of computational time.

References


