Hessisches Kompetenzzentrum
für Hochleistungsrechnen

Correctness Checking of MPI Derived Datatypes Using TypeART



Project Manager
Dr. Alexander Hück

Principal Investigator
Prof. Dr. Christian Bischof

Project Term
2022 - 2023

Clusters
Lichtenberg II Cluster Darmstadt

Additional Software
OpenMPI, LULESH, 104.milc, TypeART, Clang/LLVM

Institute
Institute of Scientific Computing

University
Technische Universität Darmstadt

JUSTUS-LIEBIG-UNIVERSITÄT GIESSEN
TECHNISCHE UNIVERSITÄT DARMSTADT
UNIKASSEL VERSITÄT
Philipps Universität Marburg

Introduction

The message passing interface (MPI) is the de-facto standard for distributed high performance computing. However, it defines a low level interface. Data is transferred as a type less void buffer and the user has to specify the data length and data type manually. This is error prone.
Consider the MPI function MPI_Send(const void* buffer, int count, MPI_Datatype datatype, ...).
In this function, the following key arguments exist:

- **buffer**: A pointer to the data for the send operation. It is a typeless void* buffer, hence, MPI is unaware of the actual datatype.
- **count**: The number of elements in the buffer.
- **datatype**: The user-specified MPI datatype (e.g., MPI_INT, MPI_DOUBLE) that describes each element in the buffer.

Without extra tools, there is no built-in mechanism in MPI libraries to verify:

1. Type compatibility: Whether the actual datatype in the buffer matches the MPI datatype as declared. Sending an array of integers with MPI_DOUBLE would lead to erroneous results.
2. Buffer overflow: Whether the count specified is larger than the allocated size of the buffer. This could cause a program to read or write from memory out of bounds, leading to crashes or undefined behavior.

To help developers write correct code, we developed a correctness checker that compares the type of the buffer allocation against what the user specified in the MPI call. For any mismatch, an error message is generated that describes the exact type of mismatch that happened.

Methods

We developed a library that intercepts all MPI communication calls in the target program using PMPI, called CAMPICC.
It compared the user-allocated buffer against the specified MPI_Datatype argument.
We deconstruct the latter (with appropriate MPI API calls) for a detailed comparison of the memory layout of the buffer.
Built-in MPI datatypes (such as MPI_INT) have a pre-defined memory layout, however user-defined MPI datatypes such as structs (MPI_Type_create_struct) must be compared at the granularity of each struct member.
To query type information of the typeless buffer, we use the existing tool called TypeART, which is a type and memory allocation tracking sanitizer based on the LLVM compiler toolchain for C/C++ (OpenMP) codes. It consists of an LLVM compiler pass plugin for instrumentation, and a corresponding runtime to track memory allocations during the execution of a target program. For each MPI call, we use the TypeART runtime to query information of the respective allocation and compare to the data specified in the MPI call.
By utilizing this information, we can reconstruct the MPI datatype information and verify its correctness.

Results

With the successful end of our project, we now support type checking of built-in MPI datatypes as well as derived datatypes, e.g., MPI_type_struct.
We evaluated our tool LULESH and 104.milc. The performance overhead is at an acceptable level of less than 10 percent compared to vanilla (uninstrumented version without extra tooling).

Discussion

CAMPICC supports nested MPI data types by intercepting MPI calls and using TypeART to verify that the user specified type information is accurate. Our evaluation has shown that we can ensure type correct MPI codes add an acceptable performance overhead.
In the future we want to extend our tool to support hybrid MPI codes such as MPI plus OpenMP, or plus CUDA.

Publications

Hück, A; Kreutzer, S; Protze, J; Lehr, J-P; Bischof, C; Terboven, C; Müller, M. S. Compiler-Aided Type Correctness of Hybrid MPI-OpenMP Applications. In IT Professional, vol. 24, no. 2, pages 45–51. IEEE (2022) https://doi.org/10.1109/MITP.2021.3093949

Last Update: 2024-08-15 11:53