MPI Analyser: User & Development Guide

1. Introduction

Welcome to the MPI Analyser, a standalone command-line tool designed to perform static analysis on MPI-based C/C++ programs. By parsing LLVM Intermediate Representation (IR), this tool detects MPI communication patterns, identifies matching Send/Recv pairs, and warns about potential deadlocks caused by unmatched calls.

This guide provides instructions for both end-users who want to run the tool and developers who want to build it from source.

Key Features

- Rich Command-Line Interface: Professional CLI options for specifying inputs, outputs, formats, and behavior.
- Multiple Output Formats: Generate human-readable text reports for the terminal, or machine-readable JSON and CSV formats for integration with other tools.
- Efficient Analysis Engine: Uses modern C++ and hash maps to efficiently group and match MPI operations by communicator and tag.
- Robust Error Handling: Provides clear warnings for unmatched calls and supports a `--strict` mode to treat these warnings as fatal errors.
- Verbose Logging: A `--verbose` mode provides detailed insight into the tool's parsing and analysis steps.
- Cross-Platform Build System: Uses CMake and Ninja to enable building the tool on Windows, Linux, and macOS.

2. How to Use the Tool (User Guide)

The tool is designed to be run from the command line. The primary input is one or more LLVM IR files (`.ll`).

Command Syntax:
`mpi-analyser.exe [OPTIONS] --input <file1.ll> [<file2.ll> ...]`

Command-Line Options:
`-i, --input <file>`: (Required) One or more input LLVM IR files.
`-o, --output <file>`: Write the report to a specified output file instead of the console.
`--json`: Format the report as JSON.
`--csv`: Format the report as CSV.
`-v, --verbose`: Enable detailed processing and debugging logs.
`--strict`: Treat any unmatched MPI calls as a fatal error and exit with an error code.
`--version`: Print the tool's version information and exit.
`--help`: Display the full list of commands and options.

Example Usage:

1. Basic analysis of a single file, printing to the console:

(PowerShell)
```
D:\...\build> ./Debug/mpi-analyser.exe --input ../test-build/uniform_comm.ll
```

2. Analyzing multiple files and saving the report as a JSON file:

(PowerShell)
```
D:\...\build>          ./Debug/mpi-analyser.exe          --input          ../test-build/uniform.ll
../test-build/mismatched.ll --json -o report.json
```

3. Running in verbose and strict mode:

(PowerShell)
```
D:\...\build> ./Debug/mpi-analyser.exe --input ../test-build/mismatched_tag.ll --verbose --strict
```

3. How to Build From Source (Developer Guide)

To compile the MPI Analyser from source, you need a specific set of tools and a correctly structured project directory. This guide assumes the project is located at `D:\MPI-polished-tool` and its dependencies are at `D:\CompilerDesign-MPI-Analysis`.

Prerequisites

- **LLVM + Clang Development Kit:** The `.tar.xz` archive, not the `.exe` installer.
- **Microsoft C++ (MSVC) Build Tools:** With the "Desktop development with C++" workload installed.
- **Microsoft MPI SDK:** The `.msi` file, which contains the required `mpi.h`.
- **CMake (Version 3.20+):** The official version from kitware.com.
- **Ninja Build System:** A fast, modern build tool.

Build Steps

All commands should be run from a PowerShell terminal.

1. Generate Test IR Files (One-Time Setup):
First, compile the C test files into the LLVM IR that the tool analyzes.

(PowerShell)

```
# Run from the project root (e.g., D:\MPI-polished-tool)
mkdir test-build
D:/CompilerDesign-MPI-Analysis/LLVM/bin/clang     -S     -emit-llvm     tests/uniform_comm.c     -o
test-build/uniform_comm.ll -I"D:/CompilerDesign-MPI-Analysis/MPI/Include"
```

2. Configure and Build the Tool:
This two-step process uses CMake to prepare the build files and Ninja to compile the code.

(PowerShell)

```
# Run from the project root
mkdir build
cd build
cmake     -G     "Ninja"     -DCMAKE_C_COMPILER="D:/CompilerDesign-MPI-Analysis/LLVM/bin/clang.exe"
-DCMAKE_CXX_COMPILER="D:/CompilerDesign-MPI-Analysis/LLVM/bin/clang++.exe" ..
ninja
```

4. The Development Journey (Project History)

We faced a series of classic and complex setup issues. Here is a summary of each problem and its solution.

- Challenge 1: `command not found` for compilers.
  - Solution: Manually add the path to LLVM's `bin` directory to the system PATH.

- Challenge 2: Hardcoded Paths in `llvm-config`.
  - Solution: Extract the LLVM archive directly to its final destination instead of moving it after extraction.

- Challenge 3: `'mpi.h' file not found`.
  - Solution: Install the MS-MPI SDK and use the `-I` flag to provide the include path to the compiler.

- Challenge 4: Compiler Errors related to C++17 features.
  - Solution: Add the `-std=c++17` flag to the compiler command.

- Challenge 5: Linker Errors (`unresolved external symbol`).
  - Solution: A "dependency chase." We had to add linker flags (`-l...`) for every missing LLVM and system library.

- Challenge 6: CMake using MSVC instead of Clang on Windows.
  - Solution: Use the `-G "Ninja"` generator flag and explicitly set `CMAKE_CXX_COMPILER` to force CMake to use Clang.

- Challenge 7: Linker errors due to C Runtime Mismatches (`_ITERATOR_DEBUG_LEVEL`, `RuntimeLibrary`).
  - Solution: Explicitly set `CMAKE_BUILD_TYPE` to `Release` and `CMAKE_MSVC_RUNTIME_LIBRARY` to `MultiThreaded` to match the pre-built LLVM libraries.

Appendix A: Final CMakeLists.txt

(CMake)

```
# Minimum CMake version required
cmake_minimum_required(VERSION 3.20)

# Define the project
project(MPIPolishedTool VERSION 2.0 LANGUAGES CXX)

# Set the C++ standard to C++17
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# CRITICAL FIX 1: Force the build type to Release
set(CMAKE_BUILD_TYPE Release)
message(STATUS "Build type set to: ${CMAKE_BUILD_TYPE}")

# CRITICAL FIX 2: Match the C Runtime Library with LLVM's
set(CMAKE_MSVC_RUNTIME_LIBRARY "MultiThreaded")
message(STATUS "MSVC Runtime Library set to: ${CMAKE_MSVC_RUNTIME_LIBRARY}")

# Use llvm-config precisely
set(LLVM_CONFIG_EXECUTABLE "D:/CompilerDesign-MPI-Analysis/LLVM/bin/llvm-config.exe")

# Get ONLY the library names (as a string)
execute_process(
        COMMAND ${LLVM_CONFIG_EXECUTABLE} --libs core irreader support asmparser bitreader
bitstreamreader remarks binaryformat targetparser demangle --system-libs
    OUTPUT_VARIABLE LLVM_LIBS_STR
    OUTPUT_STRIP_TRAILING_WHITESPACE
)

# Remove the problematic libxml2s.lib
string(REPLACE "libxml2s.lib" "" LLVM_LIBS_STR_CLEAN "${LLVM_LIBS_STR}")
separate_arguments(LLVM_LIBS_LIST NATIVE_COMMAND "${LLVM_LIBS_STR_CLEAN}")

# Configure the Project
add_executable(mpi-analyser
    src/main.cpp
    src/Analysis.cpp
    src/Reporter.cpp
)

# Apply the flags and paths correctly
execute_process(COMMAND ${LLVM_CONFIG_EXECUTABLE} --includedir OUTPUT_VARIABLE LLVM_INCLUDE_DIR
OUTPUT_STRIP_TRAILING_WHITESPACE)
target_include_directories(mpi-analyser PRIVATE
    "${CMAKE_CURRENT_SOURCE_DIR}/lib"
    "${LLVM_INCLUDE_DIR}"
    "D:/CompilerDesign-MPI-Analysis/MPI/Include"
)

execute_process(COMMAND ${LLVM_CONFIG_EXECUTABLE} --libdir OUTPUT_VARIABLE LLVM_LIB_DIR
OUTPUT_STRIP_TRAILING_WHITESPACE)
target_link_directories(mpi-analyser PRIVATE
    "${LLVM_LIB_DIR}"
```

```
)

target_link_libraries(mpi-analyser PRIVATE
    ${LLVM_LIBS_LIST}
)

message(STATUS "Successfully configured MPI Polished Tool")
```