

Introduction to Debugging on HPC

Holger Obermaier | 15. May 2023



Debugging

1. Debugging
2. GNU Compiler Collection
 - Static Analyzer
 - Sanitizer
3. Strace
4. Ltrace
5. Valgrind
6. GDB (The GNU Project Debugger)
7. Visual Studio Code
8. ARM Forge DDT (Distributed Debugging Tool)

Debugging ...

- Software contains errors and bugs!
- Debugging: Process of finding and fixing these bugs
- Debugging steps
 - Recognize that a bug exists
 - Determine the origin of the bug
 - Find a fix
 - Apply and test the fix
- Complex task $\Rightarrow$ Tools can assist with this
- Special attention should be paid to avoiding bugs (e.g. unit tests, documentation, ...)
- See: WikiBook: Computer Programming Principles [↗](#)

Additional challenges when debugging HPC codes

- Concurrency
- Multiple levels of parallelism (MPI, OpenMP, SIMD, ...)
- Heterogeneous execution platform (CPU, Accelerator)
- Code runtime
- Size of datasets
- Remote code execution on compute resources
- Asynchronous, non-interactive code execution on compute resources

Debugging, Common Bugs

- Use of uninitialized values

```
double x, y;  
x = y + 1.;
```

- Use after free

```
auto charArray = new char[10];  
delete[] charArray;  
std::cout << charArray[0] << std::endl;
```

- Out of bound access

```
auto charArray = new char[10];  
charArray[10] = 10;
```

Debugging, Common Bugs ...

- Numerical problems

```
double x = 1, y = 0;  
std::cout << x / y << std::endl;
```

- Datatype underflow / overflow

```
int i = INT_MAX;  
i++;
```

- Deadlocks

```
MPI_Ssend(&send_data, 1, MPI_INT, right_neighbour, 0, MPI_COMM_WORLD);  
MPI_Recv(&recv_data, 1, MPI_INT, left_neighbour, 0, MPI_COMM_WORLD, &status);
```

GNU Compiler Collection: Static Analyzer

- Static-analysis of source code during compile time
 - ⇒ A kind of extended warnings
 - ⇒ No need to run the binary
- Potentially false negatives
- Problems that can be detected
 - double memory free
 - double file close
 - use of memory after free
 - use of uninitialized values
 - ...
- GCC Static Analyzer Options [↗](#)
- Basic usage

```
gcc -fanalyzer ${SOURCE}
```

Example

- Use after free [↗](#)
- Uninitialized values [↗](#)

GNU Compiler Collection: Sanitizer

- Compiler adds run-time instrumentation
- Analysis during runtime
- Problems that can be detected
 - out of bounds memory access
 - use of uninitialized values
 - undefined behaviour
 - use of memory after free
 - thread data race conditions
 - ...
- GCC Program Instrumentation Options [↗](#)
- Basic usage

```
gcc -fsanitize=${SANITIZER_TYPE} ${SOURCE}
```

Example

- Use after free [↗](#)
- Datatype Overflow [↗](#)
- Division by zero [↗](#)
- Data race [↗](#)

Strace

- Linux syscall tracer
 - ⇒ Monitor interactions between processes and the Linux kernel
- Dynamic-analysis during runtime
- Potentially large overhead
- `strace.io` 
- Man page `strace` 
- Basic usage

```
module add devel/strace
strace    ${BINARY}    # trace binary
strace -p ${PID}       # trace already running process
```



Example

- Basic usage 
- MPI usage scenarios 

Ltrace

- Intercepts and records dynamic library calls
- Dynamic-analysis during runtime
- Potentially large overhead
- ltrace.org 
- Man page [ltrace](#) 
- Basic usage

```
ltrace    ${BINARY}    # trace binary
```

```
ltrace -p ${PID}      # trace already running process
```

Example

- Basic usage 

- MPI usage scenarios 

- Tool to detect
 - memory management bugs
 - threading bugs
 - profile your programs
- Prepare unoptimized binary with debug symbols during compile time
- Dynamic-analysis during runtime
- Binary is executed in virtual machine with JIT
 - ⇒ Massive increase in runtime and memory consumption
- Problems that can be detected by *memcheck*
 - use of memory after free
 - out of bounds memory access
 - use of uninitialized values
 - memory leaks
- Potentially many false positives ⇒ suppress mechanisms



Valgrind ...

- valgrind.org 
- [Valgrind User Manual](#) 
- Basic usage

```
module add \  
    compiler/gnu \  
    devel/valgrind  
gcc -O1 -g ${SRC} -o ${BINARY}  
valgrind ${BINARY}
```

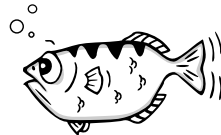
Example

- [Use after free](#) 
- [Out of bound access](#) 
- [MPI usage scenarios](#) 

GDB (The GNU Project Debugger)

- Inspect a program during execution
- Pause the program (at breakpoints or on conditions)
- Read values from memory or stack
- Change values in memory or stack
- GDB: The GNU Project Debugger [↗](#)
- GDB User Manual [↗](#), Man page GDB [↗](#)
- Basic usage

```
module add compiler/gnu  
gcc -O1 -ggdb ${SRC} -o ${BINARY}  
gdb ${BINARY}
```



GDB (The GNU Project Debugger) ...

Most frequently used GDB commands

`break <func>` Set a breakpoint at `<func>`
 `run` Run program
 `bt` Backtrace (show call stack)
`next` Execute next program line. Do not step into function calls
`step` Execute next program line. Step into function calls
 `c` Continue running program (until next break point)
`print <expr>` Print value of `<expr>`

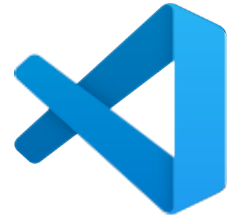
Example

■ Division by zero 

■ Remote debugging 

Visual Studio Code

- Visual Studio Code 
- Open source
- Platforms: Linux, macOS and Windows
- Feature rich integrated development environment
 - Syntax highlighting, bracket-matching, auto-indentation, ...
 - Intelligent code completion
 - Refactoring
 - Many programming languages
- Tools integrated for
 - Version control (GIT)
 - Build (code configuration, compile, linting, static checks)
 - Debug (GDB, LLDB)
- Extensions (add languages, debuggers, and tools)



Visual Studio Code - Debugging

- Extensions for debugging support:
 - C/C++: C/C++ for Visual Studio Code [↗](#)
 - Fortran: Modern Fortran [↗](#)
- Documentation
 - Visual Studio Code - Debugging [↗](#)
 - Debug C++ in Visual Studio Code [↗](#)
- Features
 - (Un)conditional breaking / logging points
 - Expression evaluation
 - Multithreaded debugging (OpenMP)
 - Call stack
 - Stepping
 - Watch window

Visual Studio Code - Debugging ...

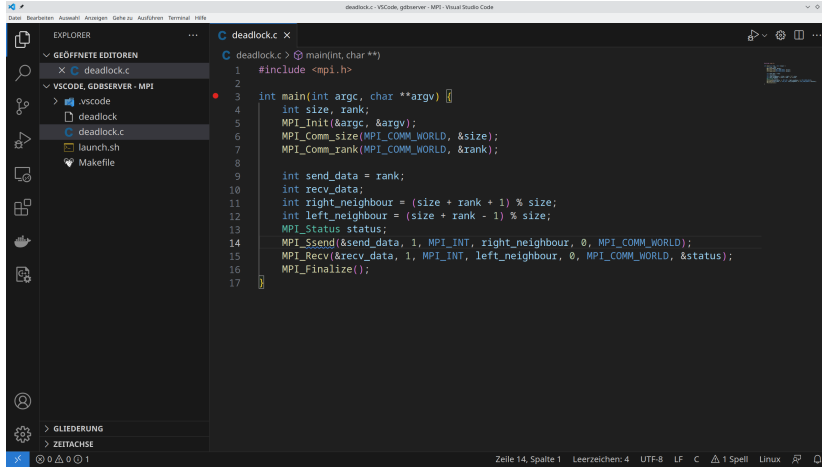
- Debug modes:
 - Config: `.vscode/launch.json`
 - Attach to running application
 - Launch application
 - Connect to running debug server (gdb Machine Interface)
- Debug actions
 - F5 Continue / Pause
 - F10 Step Over
 - F11 Step Into
 - Shift + F11 Step Out
 - Shift + F5 Stop

Visual Studio Code - Debugging ...

Example config `.vscode/launch.json` (get config options: Ctrl + Space):

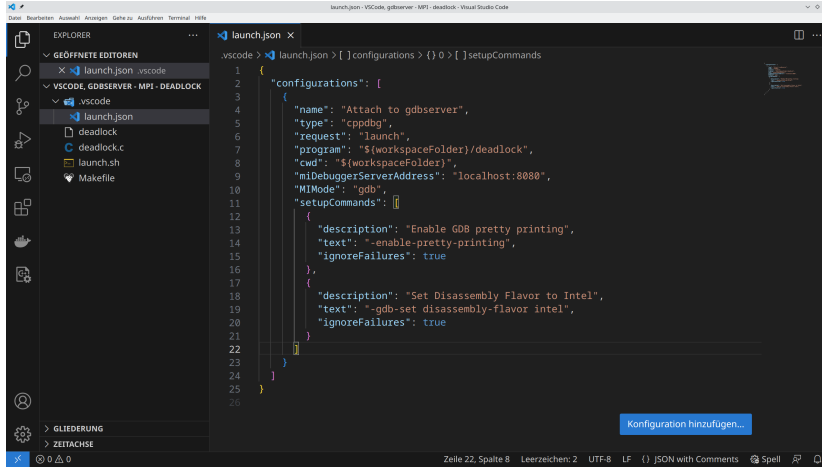
```
{
  "configurations": [
    {
      "name": "Attach to gdbserver",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/division_by_zero",
      "cwd": "${workspaceFolder}",
      "miDebuggerServerAddress": "localhost:8080",
      "MIMode": "gdb",
      "setupCommands": [ { ... } ]
    }
  ]
}
```

Visual Studio Code ...



```
1 #include <mpi.h>
2
3 int main(int argc, char **argv) {
4     int size, rank;
5     MPI_Init(&argc, &argv);
6     MPI_Comm_size(MPI_COMM_WORLD, &size);
7     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8
9     int send_data = rank;
10    int recv_data;
11    int right_neighbour = (size + rank + 1) % size;
12    int left_neighbour = (size + rank - 1) % size;
13    MPI_Status status;
14    MPI_Send(&send_data, 1, MPI_INT, right_neighbour, 0, MPI_COMM_WORLD);
15    MPI_Recv(&recv_data, 1, MPI_INT, left_neighbour, 0, MPI_COMM_WORLD, &status);
16    MPI_Finalize();
17 }
```

Visual Studio Code ...



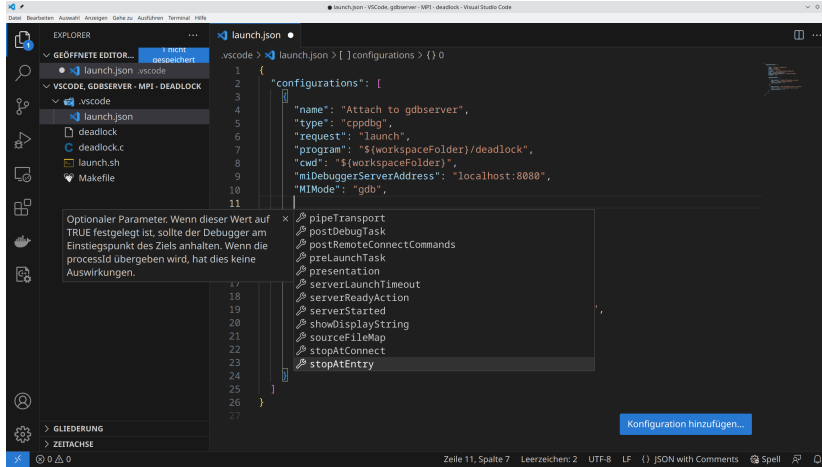
The screenshot shows the Visual Studio Code interface with a dark theme. The Explorer sidebar on the left shows a project structure with folders for 'GEÖFFNETE EDITOREN', 'VS CODE, GDBSERVER - MPI - DEADLOCK', and '.vscode'. The file 'launch.json' is selected. The main editor displays the content of 'launch.json', which is a JSON configuration for launching a debugger. The configuration includes a single entry for 'Attach to gdbserver' with various settings like 'type', 'request', 'program', 'cwd', 'miDebuggerServerAddress', and 'MIMode'. It also includes two 'setupCommands' to enable GDB pretty printing and set the disassembly flavor to Intel. A status bar at the bottom indicates the current cursor position (Zeile 22, Spalte 8) and other details like 'Leerzeichen: 2', 'UTF-8', 'LF', and 'JSON with Comments'.

```
1 {
2   "configurations": [
3     {
4       "name": "Attach to gdbserver",
5       "type": "cppdbg",
6       "request": "launch",
7       "program": "${workspaceFolder}/deadlock",
8       "cwd": "${workspaceFolder}",
9       "miDebuggerServerAddress": "localhost:8080",
10      "MIMode": "gdb",
11      "setupCommands": [
12        {
13          "description": "Enable GDB pretty printing",
14          "text": "-enable-pretty-printing",
15          "ignoreFailures": true
16        },
17        {
18          "description": "Set Disassembly Flavor to Intel",
19          "text": "-gdb-set disassembly-flavor intel",
20          "ignoreFailures": true
21        }
22      ]
23    }
24  ]
25 }
```

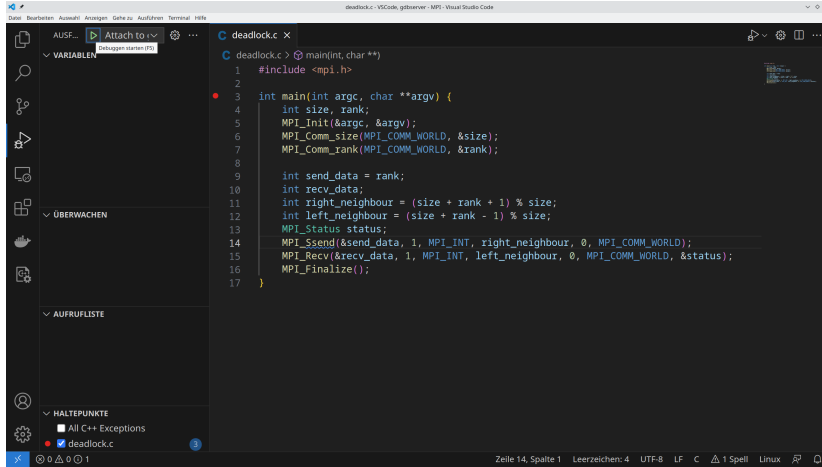
Konfiguration hinzufügen...

Zeile 22, Spalte 8 Leerzeichen: 2 UTF-8 LF {} JSON with Comments Spell

Visual Studio Code ...



Visual Studio Code ...

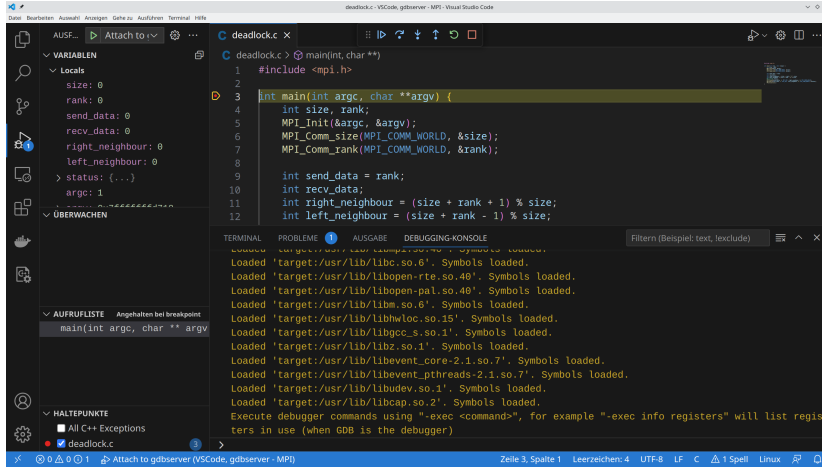


The screenshot shows the Visual Studio Code editor with a C++ file named `deadlock.c`. The code is as follows:

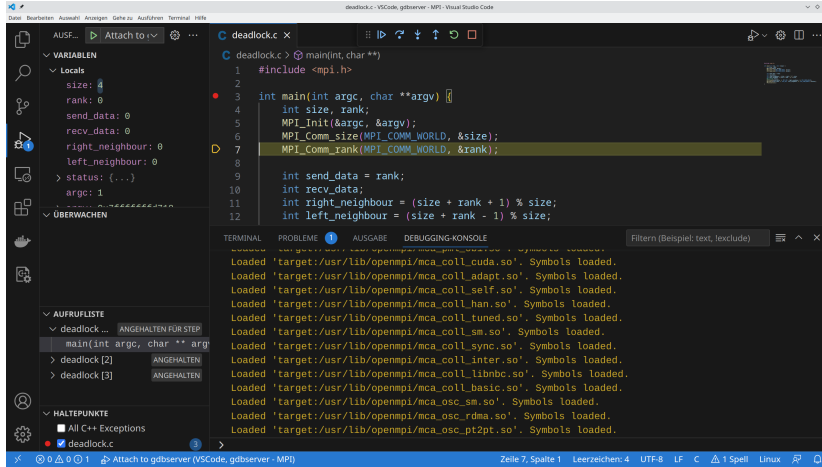
```
1 #include <mpi.h>
2
3 int main(int argc, char **argv) {
4     int size, rank;
5     MPI_Init(&argc, &argv);
6     MPI_Comm_size(MPI_COMM_WORLD, &size);
7     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8
9     int send_data = rank;
10    int recv_data;
11    int right_neighbour = (size + rank + 1) % size;
12    int left_neighbour = (size + rank - 1) % size;
13    MPI_Status status;
14    MPI_Ssend(&send_data, 1, MPI_INT, right_neighbour, 0, MPI_COMM_WORLD);
15    MPI_Recv(&recv_data, 1, MPI_INT, left_neighbour, 0, MPI_COMM_WORLD, &status);
16    MPI_Finalize();
17 }
```

The left sidebar shows the Explorer view with a folder named `deadlock.c`. The bottom status bar indicates the current position is `Zeile 14, Spalte 1` (Line 14, Column 1) and the file encoding is `UTF-8`.

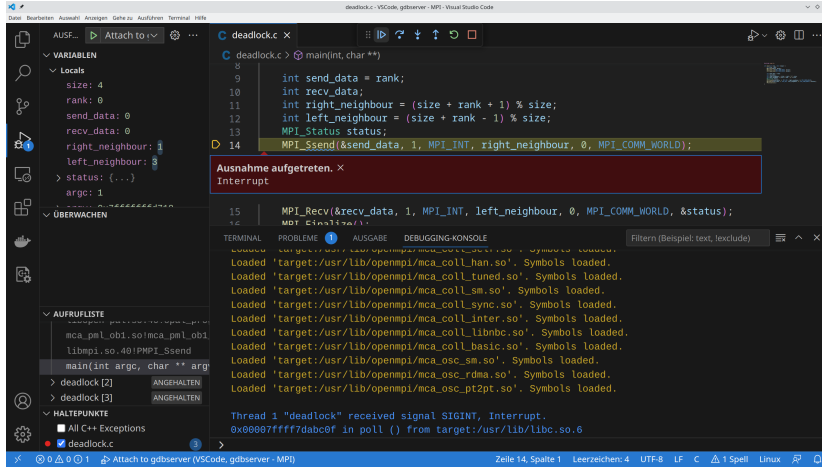
Visual Studio Code ...



Visual Studio Code ...



Visual Studio Code ...

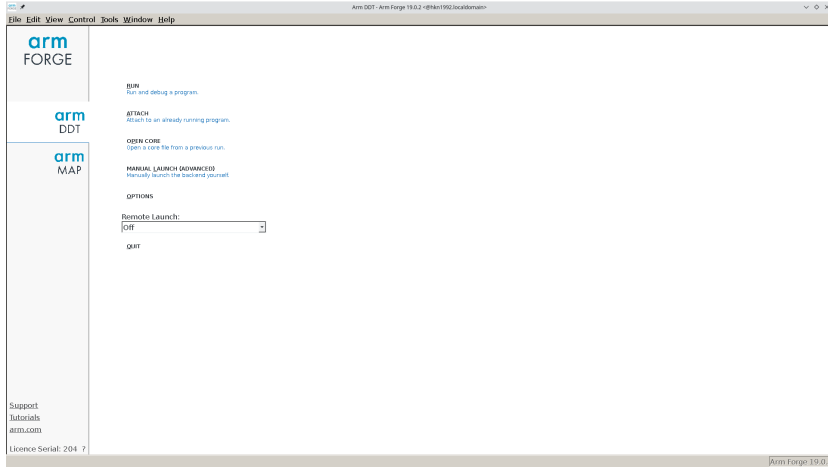


ARM Forge DDT (Distributed Debugging Tool)

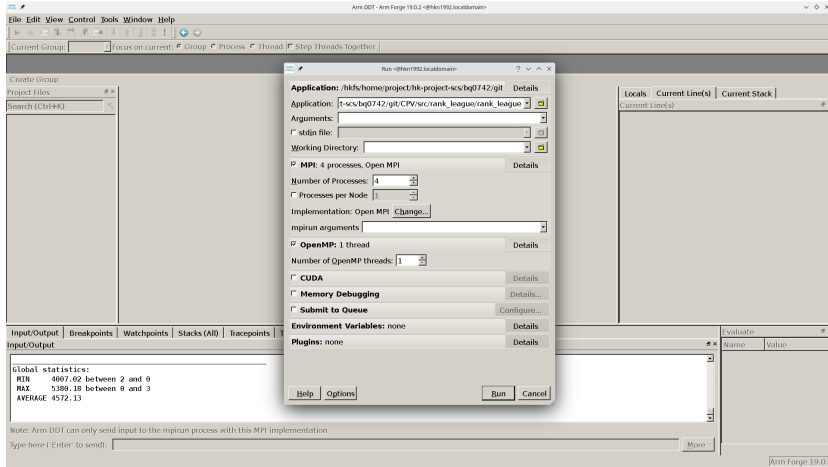
- Graphical debugger
- Supports OpenMP, MPI and GPUs
- Commercial license available on HoreKa
- Arm DDT [↗](#)
- Get started [↗](#), User Guide [↗](#), Tutorials [↗](#)
- Basic usage

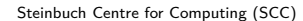
```
module add devel/ddt/  
ddt
```

ARM Forge DDT (Distributed Debugging Tool) ...



ARM Forge DDT (Distributed Debugging Tool) ...





ARM Forge DDT (Distributed Debugging Tool) ...

