OpenFOAM® Code Debugging and Profiling

University of Minho
School of Engineering

INSTITUTE FOR
POLYMERS AND COMPOSITES

Computational
Rheology
@IPC

Bruno Ramoa[1] and J.M.Nóbrega[1]

[1] Institute for Polymers and Composites, University of Minho, Portugal

bruno.ramoa@dep.uminho.pt

Researcher@IPC-UMinho

http://www.ipc.uminho.pt/

https://crheo.org

11/07/2023

Outline

Introduction

This course requires:

- OpenFOAM® installation
- gdb → `sudo apt install gdb`
- vsCode → `sudo apt install code`
- valgrind → `sudo apt-get install valgrind`
- kcachegrind → `sudo apt-get install kcachegrind`
- Basic C++ and OpenFOAM understanding

Introduction

“Debugging is the process of finding and fixing errors or bugs in the source code of any software” [1]

- You coded your OpenFOAM® application/utility.
    - Your program is not working as it should?
        - You don’t know what is wrong?
            - ➢ You need to **debug** it

Within the OpenFOAM® framework you have some debugging approaches:

1. `Info` statements (print variables)
2. Use the `DebugSwitches` in *controlDict*
3. Use a debugger (e.g., GDB, LLDB)

```
    while (simple.loop())
    {
        Info<< T << endl;
…
```

```
DebugSwitches
{
    objectRegistry    1;
    surfaceInterpolation 1;
}
```

[1] - https://aws.amazon.com/what-is/debugging/ accessed on the 06/05/2023

Introduction

3. Use a debugger:

- Code must be compiled in advance with a `debug` flag.
  - There are 2 ways to achieve this:

    2. Top level debugging of application/library

       ➢ Go to the `Make` folder of your application and include in the first line of the `options` file **`-ggdb3 -O0 -DFULLDEBUG`**.

```
├── Make
│    ├── files
│    ├── options
```

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude

EXE_LIBS = \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools
```

→

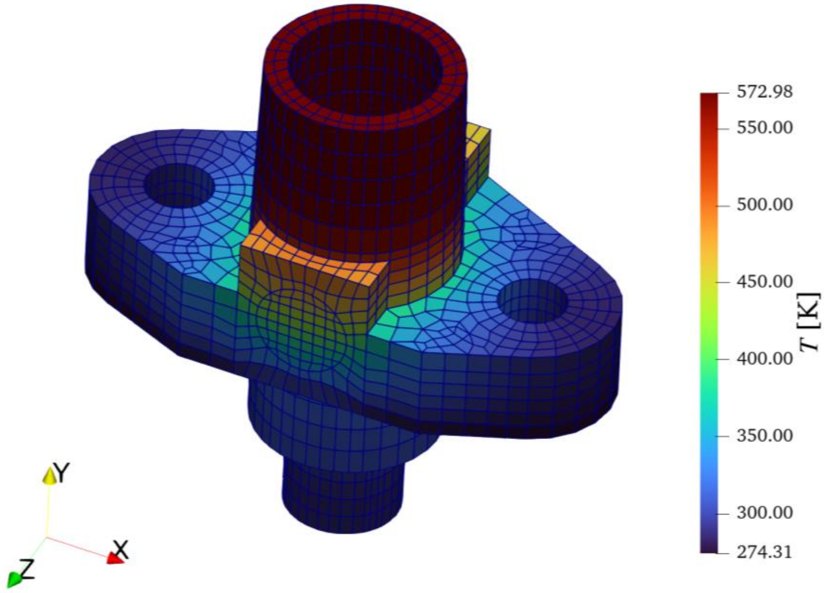
```
EXE_INC = \
    -ggdb3 -O0 -DFULLDEBUG \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \

EXE_LIBS = \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools
```

Training with GDB

laplacianFoam: $\frac{\partial T}{\partial t} - \nabla \cdot \left(D_T \nabla T \right) = S_T$

Let us explore **GDB**
and its functionality



Training with GDB

Launching	
`gdb` **`<exec_file>`**	Make `gdb` launch an executable file
`gdb --args` **`<exe_file> <args…>`**	Make `gdb` launch an executable file and pass arguments
`run`	Run the application in the debugger
`kill`	Kill the running program
`quit`	Exit `gdb`
Note: When in need: `gdb -help`	

Viewing Source	
`list` **`<lineNum>`**	Print lines centered around line number `linenum` in the current source file
`list` **`<func>`**	Print lines centered around the beginning of function **`<func>`**
`list` ***`<first>`***`,` ***`<last>`***	Print lines from ***`<first>`*** to ***`<last>`***
`list` **`<+/->`**	`+`: Print lines just after the lines last printed `-`: Print lines just before the lines last printed.
`set listsize` ***`<count>`***	By default, GDB only prints 10 lines. To change the number, define a `<count>`. For displaying everything set the value to `0` or `unlimited`
`gdb -tui` **`<exec_file>`**	Lunch `gdb` with text user interface (TUI). You can also press `Ctrl+x` followed by `Ctrl+a` to go in and out of TUI mode
Note: you can turn pagination off with `set pagination off` Note: to show the current value of a parameter: `show` **`<parameter>`**, e.g., `show listsize`	

Source of GDB commands and definitions: https://sourceware.org/gdb/documentation/

CRheo@IPC

Training with GDB

Navigation in text user interface (TUI) mode	
`Arrows: up, down, left and right`	Scroll the active window: up, down, left and right
`Ctrl+p and Ctrl+n`	To get previous command. To get the next command
`Ctrl+b and Ctrl+f`	Move backward in the line. Move forward in the line
`Ctrl+l`	Refresh the screen

BreakPoints	
`break` *`filename`*`:`**`<lineNum`**`>`	Set a breakpoint in **`<lineNum>`**`.` If no `filename` is specified, uses the current source file.
`break` *`filename`*`:`**`<func>`**	Set a breakpoint at entry to function **`<func>`**.
`break ... if` *`cond`*	Set a conditional breakpoint. Breaks when `cond` evaluates to true
`info breakpoints` **`<N>`**	Information regarding all breakpoint. If **`<N>`** is supplied, prints information regarding breakpoint **`<N>`**
`delete` **`<N>`**	Delete all breakpoints. If **`<N>`** is supplied, delete breakpoint number `<N>`
`disable/enable` **`<N>`**	Disable/Enable all breakpoints. If **`<N>`** is supplied, disable/enable breakpoint number `<N>`

Execution Control	
`next`	Execute next line of code.
`step`	Step into the next line of code.
`finish`	Step out of the current block of code.
`continue`	Resumes execution of a stopped program, and stops again at the next breakpoint

Source of GDB commands and definitions: https://sourceware.org/gdb/documentation/

CRheo@IPC

Training with GDB

WatchPoints	
`watch/rwatch/awatch` **`<expr>`**	You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen. `watch`: GDB will break when *expr* is written into by the program and its value changes. `rwatch`: Set a watchpoint that will break when the value of *expr* is read by the program. `awatch`: Set a watchpoint that will break when *expr* is either read from or written into by the program.
`info watchpoints`	Get information on available watchpoints
`delete/disable/enable`	Same syntax as in breakpoints

Symbols	
`whatis` **`<var>`**	Print the data type of the variable **`<var>`**
`ptype` **`<var>`**	Print a detailed description of variable **`<var>`**

Printing	
`print` **`<var>`**	Print value stored in variable **`<var>`**. For scalarField use: `print *<var>.v_@<sizeOfVar>` For vector/tensorField use: `print (<var>.v_).v_@<sizeOfVar>`
Note: use `help print` to see additional printing options	

Source of GDB commands and definitions: https://sourceware.org/gdb/documentation/

Training with GDB

Convenience variables	
`set $`**`<nameOfVar>`**` = `**`<something>`**	These variables exist entirely within GDB; they are not part of your program, and setting a convenience variable has no direct effect on further execution of your program.
`show convenience`	Print a list of convenience variables used so far, and their values.

Backtrace	
`backtrace `**`<option>`**	A backtrace is a summary of how your program reached the current location.
`frame `**`<N>`**	Select frame number **`<N>`**
`up/down `**`<N>`**	Move **`<N>`** frames up/down the stack
Note: use `help backtrace` to see the options	

Call program functions	
`call `**`<expr>`**	Execute a function from your program, but without cluttering the output with void returned values. If the result is not void, it is printed and saved in the value history.

User-defined commands	

`define foo`
`    print $arg0 + $arg1`
`end`
 	
`document foo`
`    print sum of two arguments`
`End`
	A *user-defined command* is a sequence of GDB commands to which you assign a new name as a command

Source of GDB commands and definitions: *https://sourceware.org/gdb/documentation/*

CRheo@IPC

Training with GDB

User-defined commands

```
1.define printField
2.     set $fieldSize = $arg0.size_
3.     set $formatType = 0
4.     set $fieldType = $arg0.typeName._M_dataplus._M_p

5.     set max-value-size unlimited
6.     set print elements 0
7.     set print repeats 0
8.     set pagination off

9.     if $argc > 1
10.            set logging file $arg1
11.            set logging redirect on
12.            set logging enabled on
13.    end

14.    if ($_regex($fieldType, ".*[sS]calar.*"))
15.            print *$arg0.v_@$fieldSize
16.    else
17.            print ($arg0.v_).v_@$fieldSize
18.            set $formatType = 1
19.    end
20.
21.    if $argc > 1
22.            set logging enabled off
23.            set max-value-size 65536
24.            set print elements 200
25.            set print repeats 10
26.            set pagination on

27.            if ($formatType == 0)
28.                    shell sed -i -e '/$[0-9]*/s/}.*$//g' -e '/$[0-9]*/s/, /\n/g' -e 's/.*$[0-9]* = {//' $arg1
29.            else
30.                    shell sed -i -e '/$[0-9]*/s/}}.*/)/' -e '/$[0-9]*/s/}, {/)\n(/g' -e 's/.*$[0-9]* = {{/(/' -e 's/,/ /g' $arg1
31.            end
32.    end
33.end
```

Description:

2- Collect size of the field to a Convenience variables

4- Collect typename of the field

5-7: Set maximum memory allocation, display all numbers without repeats.

8: Disable pagination

9-13: If the user supplied more than 1 argument, create a log file to store the data. The log file name is 2nd argument given the user

14-19: print data according to its type (scalar, vector or tensor)

21-33: If the user supplied more than 1 argument, reset values to default and make output number for OF-like

Note: to use this function, save it to a file called `<nameOfFile>.gdb`. When lunching `gdb` use: `gdb -x <nameOfFile>.gdb <executableName>`

Using GDB in vsCode

launch.json

```
1.{
2.     "version": "0.2.0",
3.     "configurations":
4.      [
5.         {
6.             "name": “Debug in OpenFOAM",
7.             "type": "cppdbg",
8.             "request": "launch",
9.             "program": "${env:FOAM_USER_APPBIN}/<solver>",
10.             "args": [],
11.             "stopAtEntry": false,
12.             "cwd": "${workspaceFolder}/<case folder>",
13.             "environment": [],
14.             "externalConsole": false,
15.             "MIMode": "gdb",
16.             "miDebuggerPath": "/usr/bin/gdb",
17.             "miDebuggerArgs": "-x ${workspaceFolder}/.vscode/OF_Funcs.gdb",
18.             "setupCommands":
19.             [
20.                 {
21.                     "description": "Enable pretty-printing for gdb",
22.                     "text": "-enable-pretty-printing",
23.                     "ignoreFailures": true
24.                 }
25.             ]
26.         }
27.     ]
28.}
```

Description:

9: Location of the executable program

10: Arguments to pass to the program

11: Should the program stop at the beginning of the main function?

12: Path to the case folder

13: Environment variables to add to the environment for the program

15: What debugger should vsCode connect to.

16: Path of debugger. Don’t know? Use `which gdb` in the command line

17: Arguments for the debugger

18-25: Array of commands to execute in order to set up GDB

Training with Valgrind: Memory debugging and profiling

- `memcheck` is a memory error detector tool. It is the default utility in Valgrind.
    - The program needs to be compiled with **`-g`** flag to include debugging information.
        - Note: For our cases, we can leave the previous debug flags.

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --log-
file="memCheck.txt" <solver>
```

`--leak-check=full:` Will give details for each *definitely lost* [your program is leaking memory] or *possibly lost* [the program is leaking memory, unless you're doing funny things with pointers] blocks, including where it was allocated.

`--show-leak-kinds=all:` Specifies the leak kinds to show in a full leak search, in one of the following ways: definite, indirect, possible and reachable

`--track-origins=yes:` Generate full information when looking for uninitialized values. This makes `memcheck` run more slowly, but can make it much easier to track down the root causes of uninitialized value errors

"`Memcheck` cannot detect every memory error your program has. For example, it can't detect out-of-range reads or writes to arrays that are allocated statically or on the stack."

Source: *https://valgrind.org/* and *https://valgrind.org/docs/manual/quick-start.html* accessed on 06/05/2023

Training with Valgrind: Memory debugging and profiling

- A **code profiler** is a tool to analyze a program and report on its resource usage.

- "`Callgrind` is a profiling tool that records the call history among functions in a program's run as a call-graph. By default, the collected data consists of the number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the numbers of such calls… The program does not need to be recompiled, it can use shared libraries and plugins, and the profile measurement doesn't influence the memory access behavior." [2]

- Compile with debugging info (**`-g`** flag) and with optimization turned on
  - The code will run considerably slower under `callgrind` .
  - It is good practice to run a representative task that is as small as possible when profiling the code.

[2] - *https://valgrind.org/docs/manual/cl-manual.html* accessed on 06/05/2023.

Training with Valgrind: Memory debugging and profiling

Serial:

```
valgrind --tool=callgrind --simulate-cache=yes <executable>
```

Visualize:

```
callgrind_annotate --auto=yes <nameOfOuputFile>


kcachegrind <nameOfOuputFile>
```

CRheo@IPC

University of Minho
School of Engineering

iPC
INSTITUTE FOR
POLYMERS AND COMPOSITES

Thank you for your attention

This work was funded by National Funds through FCT - Portuguese Foundation for Science and Technology through the PhD grant with reference 2020.07424.BD

Computational
Rheology
@IPC