# An Introduction to parallel debugging

Joachim Hein (LUNARC)

1

## Overview

- Introduction to debugging and parallel debugging
- Running the Linaro DDT parallel debugger

2

# INTRODUCTION TO DEBUGGING

3

## Traditional standard way to debug: "`printf` debugging"

- Add extra print statements to the code
  - Indicate whether the code reaches a certain stage
  - Print the values of key variable
- Issues with this approach
  - Need to modify the source code, recompile
  - Iterative approach, frequent recompiles
- Debuggers are more convenient
  - Allows working with unmodified source
  - Allows line by line execution

4

# Debuggers

- Linux system come with **gdb** as a debugger
  - Command line execution
  - GUIs exist and are recommended
  - Often integrated into development platforms

5

# PARALLEL DEBUGGING

6

# Parallel debugging

- Parallel applications offer new levels of complexity
- Before starting, try to simplify the task
  - Problem still there if you **reduce the problem size**?
  - Problem still there if you **reduce the task/thread count**?
- "printf debugging" even more problematic than in serial
  - More output (different tasks/threads printing)
  - Identification of task/thread printing required
  - UNIX `grep` helpful to filter output

7

# Parallel debuggers

- Licenses are expensive
  - Being able to do "`printf`" is an essential skill
- Parallel debuggers became more usable over the years
- I am aware of two products
  - Totalview for HPC (https://totalview.io/)
  - Linaro DDT - part of Linaro Forge
    - Formerly known as ALLINEA DDT/FORGE and ARM DDT/FORGE
    - There is a NAISS wide license

8

# PREPARATIONS AND STARTING DDT

9

# Preparations

- HPC system needs to display the gui on your monitor
  - VNC solution (e.g. LUNARC HPC desktop, ThinLinc)
  - Connect with X-forwarding (`ssh –X ...` )
- Recompile your application with the flags: `-g -O0`

```
mpif90 -g -O0 -o hello_mpi hello_mpi.f90
```

- Comments:
  - Lack of optimisation slows code (in particular C++)
  - Problem might disappear – hint for overrun array
  - You can use optimisation
    - Though match code line to instruction might not work

10

# Start the gui

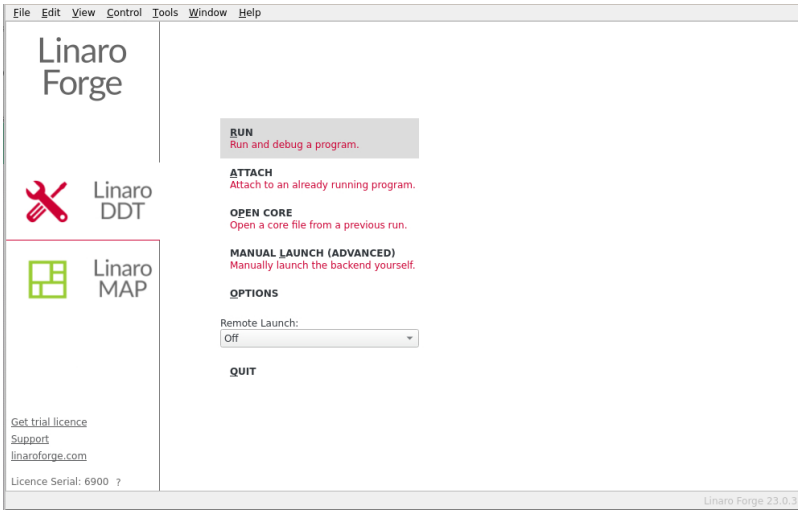
- Best to start the gui on the login node and keep it running

```
module load linaro_forge/23.0.3
ddt &
```

- Alternative use the ARM remote client on your desktop and connect to the (front end of the )HPC service

11

# Linaro Forge gui




12

# Starting code on the compute nodes

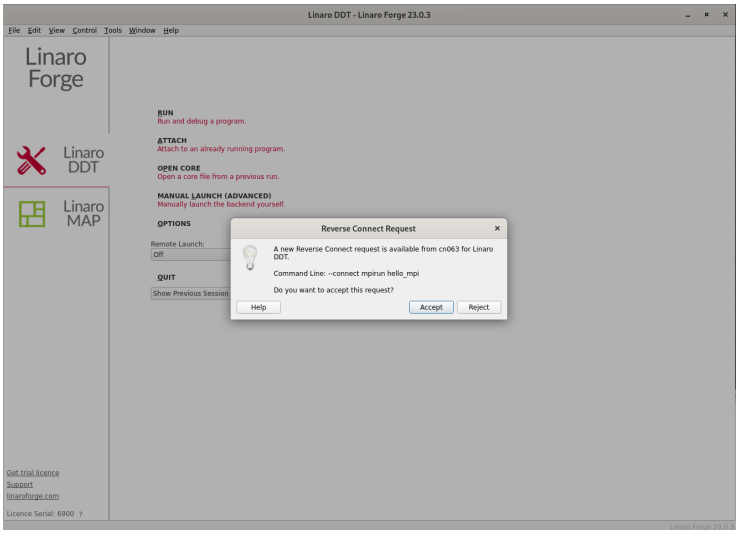
- Transfer to the backend node
  - Jobscript
  - Interactive allocation
- Make sure relevant modules are loaded
  - compiler, MPI lib, other libs, Linaro DDT/Forge
- Prefix job launcher with: `ddt --connect`

```
ddt --connect mpirun mpihello
```

```
ddt --connect mpirun python3 %allinea_python_debug%  hello_mpi.py
```

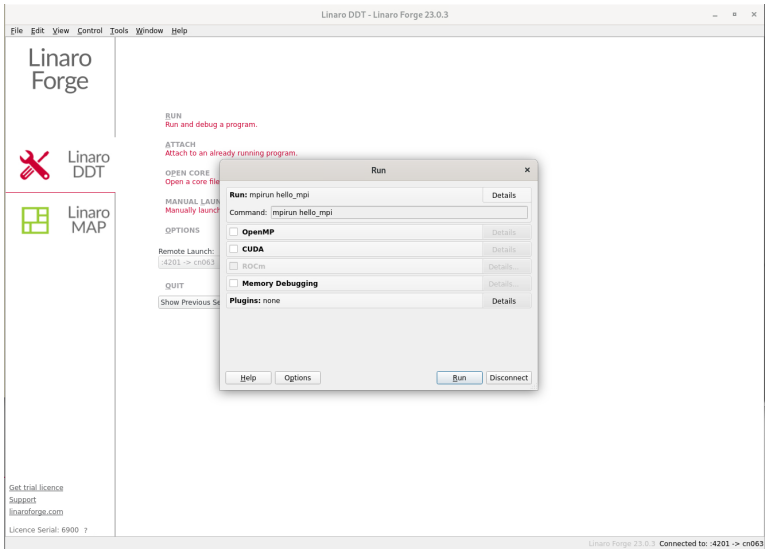
13

# Accept the “Reverse Connect request”



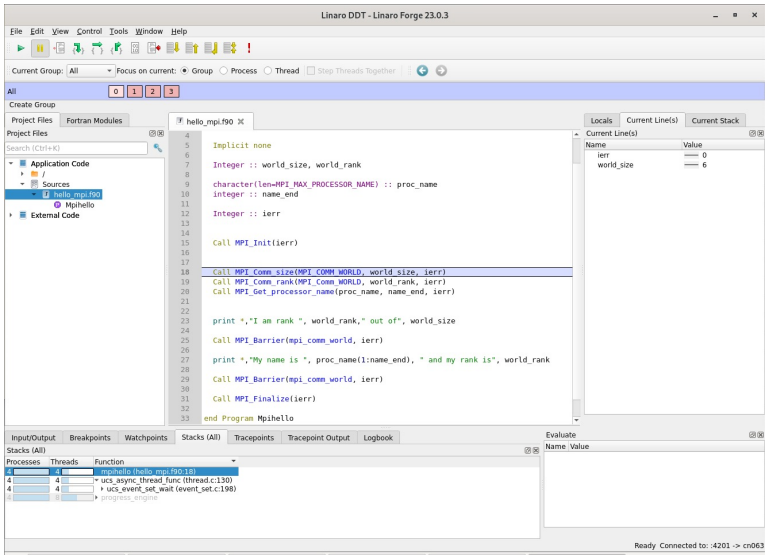

14

# Start running your program



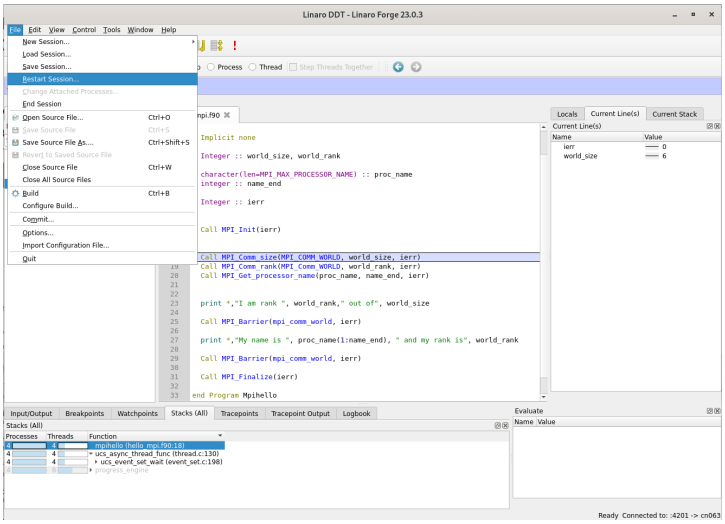

15

# Start debugging in the gui



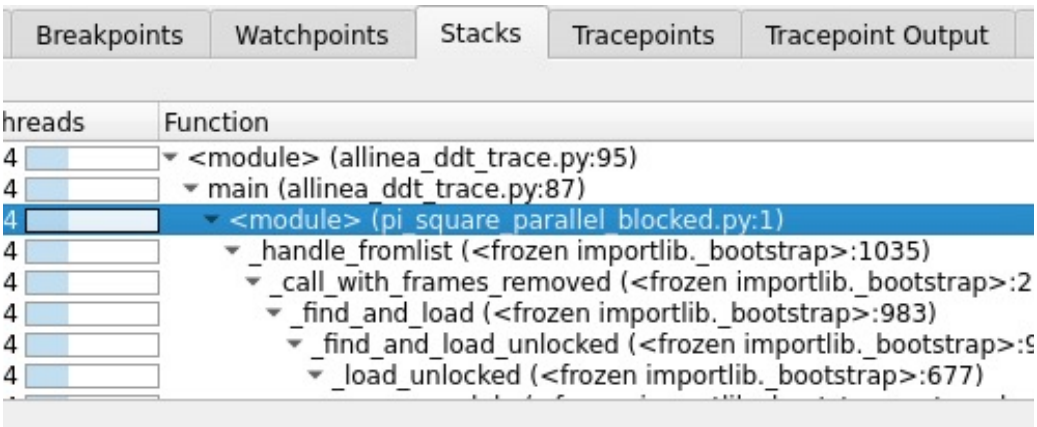

16

# Starting over – frequently required
# Use “Restart session” option




17

# Starting a Python debug session



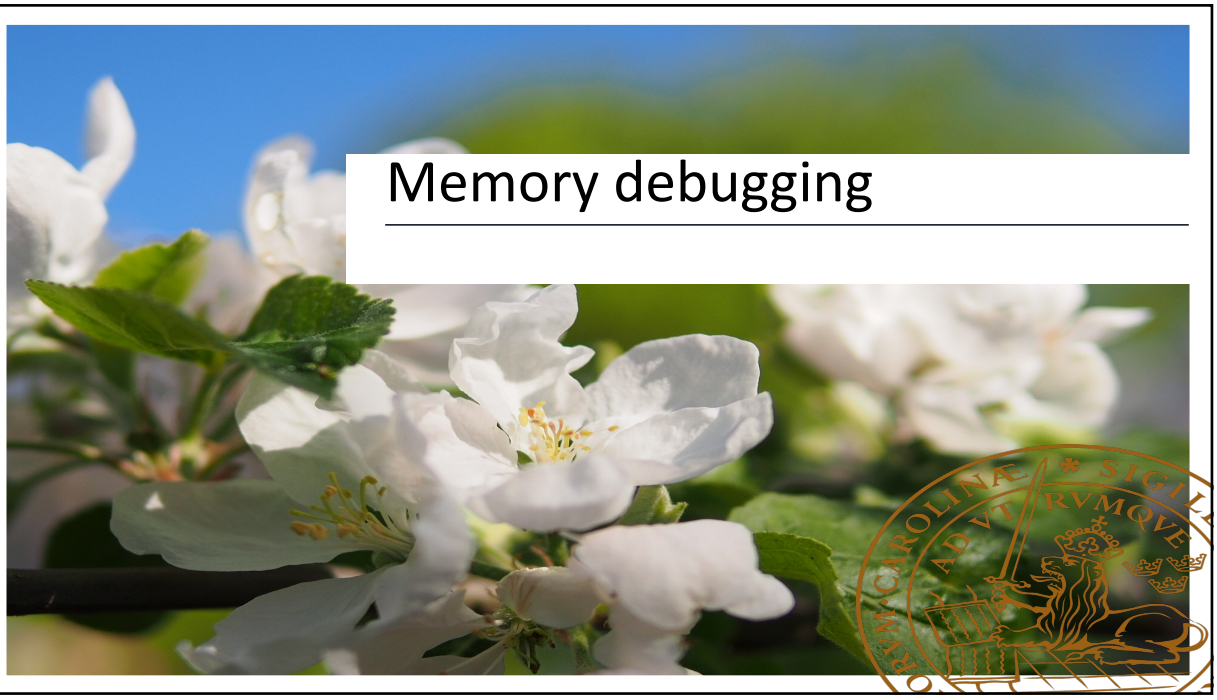

- You might need to locate Python script in “Stacks” window
- Set breakpoint on first line after importing (from) `mpi4py`

18

# Demo

- hello world (Fortran)
- Message on a ring (C)
- Pi-square (Python)

19




20

# Problematic memory access

- Codes often suffer from memory problems
  - Writing in memory locations they shouldn't
  - Illegal deallocation (double, bad pointer position, ...)
  - Memory leaks
- Typical signatures of memory problems
  - Seg-faults
  - Code behaviour changes when:
    - Editing (e.g. printf debugging)
    - Changing compilers or optimisation flags

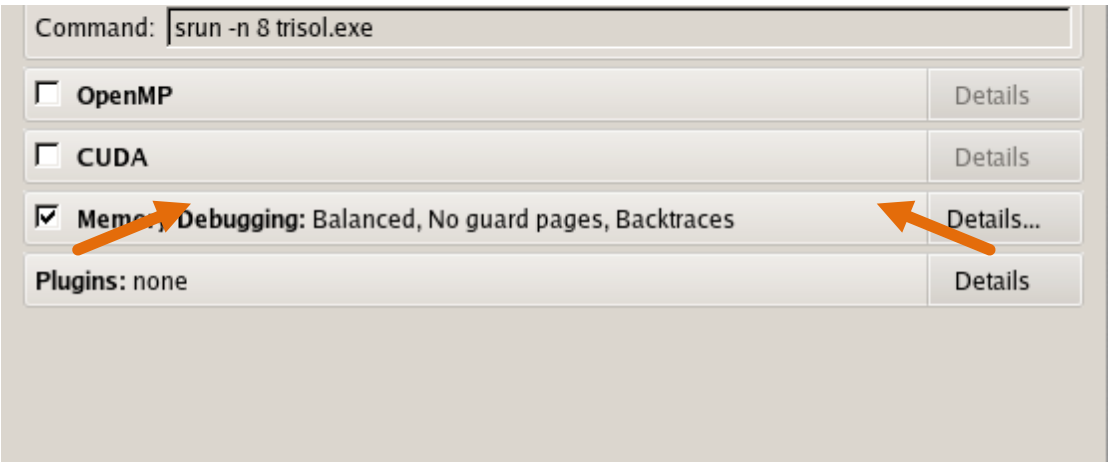
21

# Activating memory debugging in DDT

- Replace the malloc library with ARM's dmalloc
- Comes in 4 versions:
  - C/Fortran no threads
  - C/Fortran threads
  - C++ no threads
  - C++ threads
- Current version seems to prefer "threads"

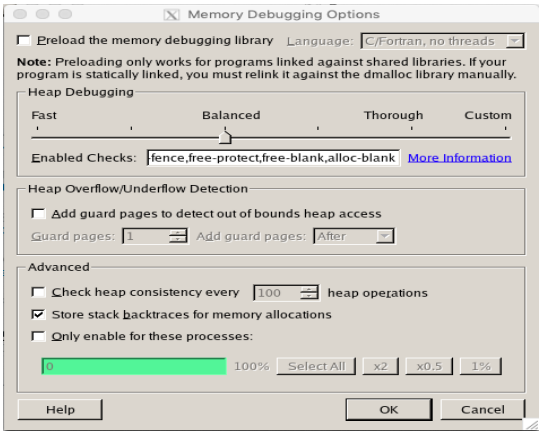
22

# Select memory debugging



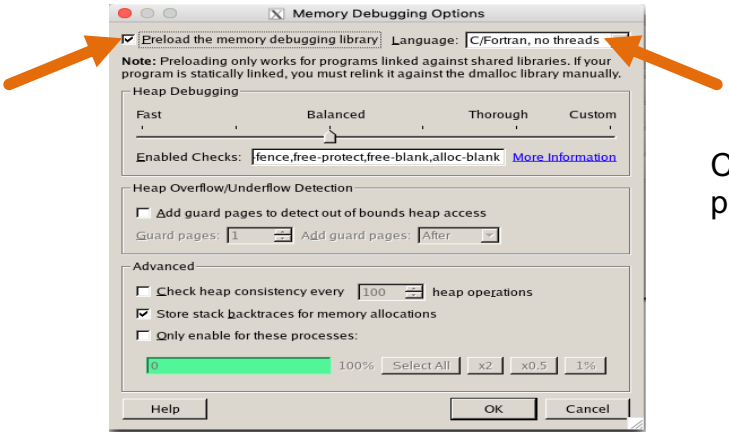

23

# Selecting Memory Debugging Option




24

# Dynamic linking




Current version prefers: threads!

25

# Static linking

- If you link statically or if dynamic linking fails
- Add a line like (check user guide)

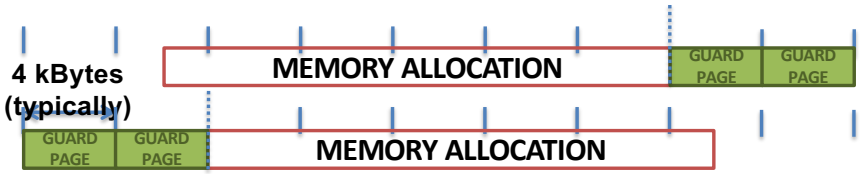
```
-Wl,--allow-multiple-definition,--undefined=malloc /path/lib/64/libdmalloc.a
```

to the link line **<u>before</u>** anything else

– Often required on CRAYs

26

# Guard pages (aka “electric fences”)

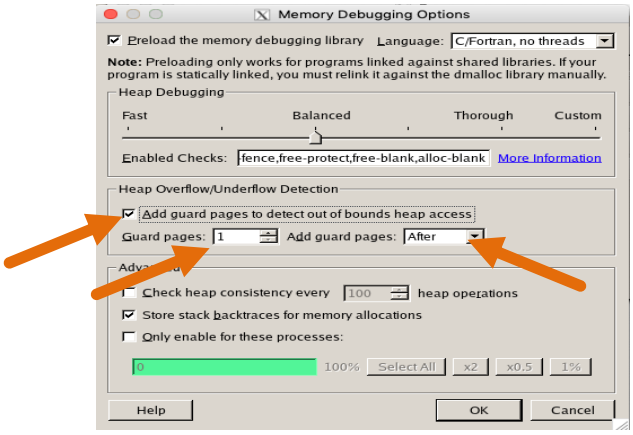



- **A powerful feature…:**
  - **Forbids read/write on guard pages throughout the whole execution**

    ***(because it overrides C Standard Memory Management library)***
- **… to be used carefully:**
  - **Kernel limitation: up to 32k guard pages max**
  - **Beware the additional memory usage cost**

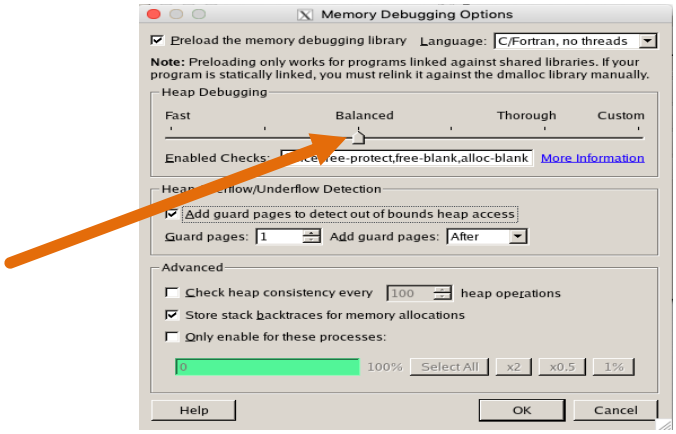
27

# Activate guard pages



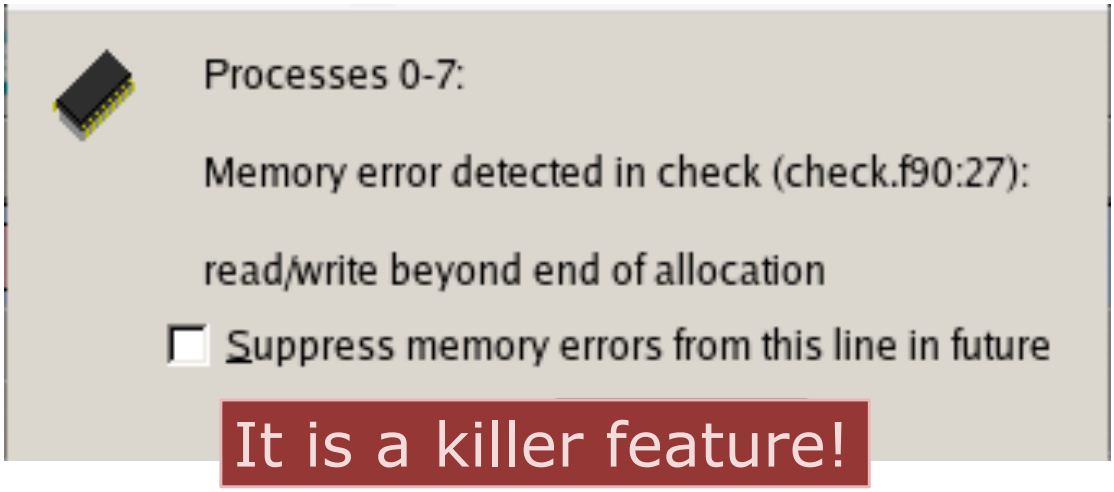

28

# Select the depth of the tests




29

# When it finds something you get:




30

# Demo

- Locating memory issue

31

# Recap/Summary

- Starting the gui
- Demonstrating how to run it
- Memory debugging feature
  - This saved me so much time in the years

32

# Acknowledgements


- Juan Gao (then at ARM)
- Patrick Wohlschlegel (than at ARM)

- Thor Wikfeldt (than at KTH/PDC)


33