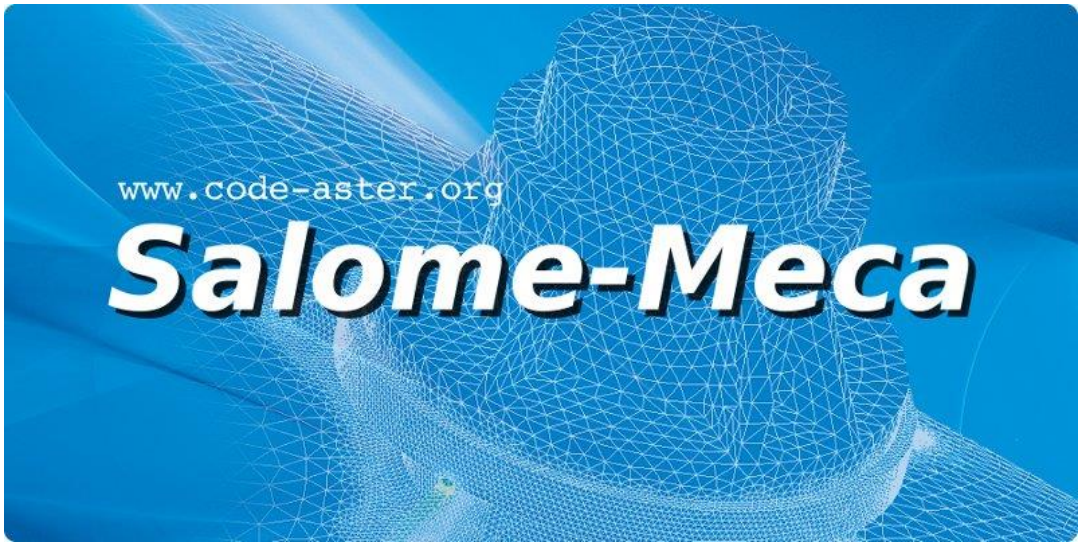
Development in code_aster
Debugging a code_aster execution



Code_Aster, Salome-Meca course material

EDF

How to debug a code_aster execution?

Requirements

Compile Code_Aster with debugging symbols.

Post-mortem analysis

How to execute and read the relevant informations.

Interactive debugging

How a debugger works, general principles.

Debug from the console.

Use a graphic debugger.

Debug a parallel execution

Attach to running session.

Debug Python code

Memory debugging

Other tools…

Requirements for debugging

Enable compiler option (‘-g’)

Produce debugging information that is embedded in the object files.

Read by the debugger to execute code line by line, show variables content…

For code_aster

```
waf install_debug
```

Build the `bin/asterd` executable in the installation directory, used by astk in the `debug` mode.

Disable code optimization (‘-O0’) to make debugging produce expected results but may return different results than the executable in production (`bin/aster`).

code_aster now uses elapsed time. Increase time_limit in .export if necessary.

Post-mortem analysis

Coredump file

When a program fails the OS writes a file containing an image of the process's memory at the time of termination.

The maximum size of the core files is 0 by default. Set unlimited size using:

```
ulimit -c unlimited
```

Study vs testcase

In case of error <F>, studies emit an exception (to return the database results) but testcases raise the SIGABRT signal that produces a coredump file.

Add `ERREUR=_F(ERREUR_F='ABORT')` in `DEBUT` to make a study write a core file.

In case of failure of code_aster with a core file

A debugger is automatically called in the post-mortem mode.

Just call `where + quit` commands to show where the error occurred.

Use the as_run parameter named `cmd_post` (see [d1.05.01], §1.5).

Post-mortem analysis: example

In the output

```
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Core was generated by `/home/user/dev/codeaster/install/std/bin/asterd /home/user/dev/codeaster/in'.

Program terminated with signal SIGABRT, Aborted.

#0  […]

#3  0x00007fcc205b8dc9 in _gfortran_abort () from /usr/lib/x86_64-linux-gnu/libgfortran.so.3

#4  0x0000000001bdfed6 in abortf () at /home/user/dev/codeaster/src/bibfor/utilifor/abortf.F90:21

#5  0x00000000004b24ab in asabrt_ (iret=0x236fb68) at ../../bibc/supervis/aster_mpi.c:688

#6  0x00000000011f1696 in jefini (cond=?, _cond=6) at
    /home/user/dev/codeaster/src/bibfor/jeveux/jefini.F90:115

#7  0x0000000001c1211d in utmess_core (typ=?, idmess=?, nk=1, valk=..., ni=1, vali=..., nr=1, valr=...,
    fname=?, _typ=2, _idmess=12, _valk=256, _fname=256) at
    /home/user/dev/codeaster/src/bibfor/utilifor/utmess_core.F90:172

#8  0x0000000001c11b45 in utmess (typ=?, idmess=?, nk=?, valk=..., sk=?, ni=?, vali=..., si=?, nr=?, valr=...,
    sr=?, num_except=?, fname=?, _typ=1, _idmess=12, _valk=0, _sk=0, _fname=0) at
    /home/user/dev/codeaster/src/bibfor/utilifor/utmess.F90:155

#9  0x00000000015c91cd in op0003 () at /home/user/dev/codeaster/src/bibfor/op/op0003.F90:111

#10 0x00000000019cc1e0 in ex0000 (nuoper=3) at /home/user/dev/codeaster/src/bibfor/supervis/ex0000.F90:232

#11 0x00000000019cca9b in execop () at /home/user/dev/codeaster/src/bibfor/supervis/execop.F90:65

#12 0x00000000019ccc22 in expass (jxvrf=1) at /home/user/dev/codeaster/src/bibfor/supervis/expass.F90:39

#13 0x00000000004af63d in aster_oper (self=0x0, args=0x7fcc103be128) at
    ../../bibc/supervis/aster_module.c:1710
```

Interactive debugging

Usage

Using astk with ‘Run / dbg’.

Using waf:

```
waf test_debug -n zzzz100f --exectool=debugger
```

Use the as_run parameter named `cmd_dbg` (see [d1.05.01], §1.5).

Manually after preparing the environment:

- Using astk with ‘Run / pre’
- or using waf:

```
waf test_debug -n zzzz100f --exectool=env
```

And read instructions from the output

→ Demo

Interactive debugging with gdb

Demo from the temporary directory

```
gdb /path/to/asterd
(gdb) break main
(gdb) run < arguments >
```

That's what as_run does with 'Run / dbg' or exectool=debugger.

Features

Execute the code line by line, show variables content…

Main commands (gdb syntax):

- Move in the stack: `where`, `up`, `down`
- Add a breakpoint: `break` *`subroutine`* / `break #`*`linenumber`*
- List breakpoints: `info breakpoints` (or `status`)
- Disable a breakpoint: `disable #`*`id`*
- Continue up to the next breakpoint: `continue`
- Show source code: `list [`*`#linenumber`*`]`
- Print variable content: `print` *`variable`*
- Assign a value to a variable: `set` *`variable`*`=value`
- Save current breakpoints: `save breakpoints` *`filename`*

Interactive debugging with Allinea DDT

Available on EDF clusters.

Execute code_aster with:

```
ddt /path/to/asterd <arguments>
```

Features

Nice graphical user interface!

→ Demo

Debugging a parallel execution

Prepare the environment as for interactive debugging

'Run / pre' or 'waf test -n testname -exectool=env' with only one processor.

Request resources on the server (number of processors and time limit):

```
salloc -N 4 -t <hour:minute:second>
ssh -X <allocated node>
```

use 'squeue' to know the allocated node

Set the environment

Read the instructions from the output of 'pre'.

Initialize MPI daemon

```
mpdboot
```

Start Totalview parallel debugger

```
module load totalview_debugger
totalview
```

Choose 'new parallel program', select Intel MPI-MPD as 'parallel system' and the number of processors.

Fill 'Program details' with executable and arguments as written in 'mpi_script.sh'.

Then 'Start session'.

Debugging Python code

Create the environment

- Using astk with 'Run / pre'
- or using waf:

```
waf test_debug -n zzzz102a --exectool=env
```

And read instructions from the output

Features of pdb are similar to gdb

→ Demo

```
/path/to/asterd /usr/lib/python2.7/pdb.py <arguments>
(Pdb) break /path/to/lib/aster/Macro/lire_fonction_ops:178
(Pdb) continue
(Pdb) next
(Pdb) print variable
```

Nicer TUI with pudb

Detecting memory errors using Valgrind

Common errors

- Uninitialized variables
- Read or write out of array bounds

Usage

- Using astk with 'exectool=memcheck' in Options menu
- or using waf:

```
waf test_debug -n zzzz100f --exectool=memcheck
```

Needs debugging symbols; 20-30 times slower than normal execution.

Configuration

In ~/.astkrc/prefs:

```
memcheck : valgrind --tool=memcheck --leak-check=full --error-limit=no --
track-origins=yes --max-stackframe=8000000
```

Valgrind: example

Report of valgrind in written in output (see '==PID==' lines):

```
==30155== Memcheck, a memory error detector
==30155== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==30155== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==30155== Command: /path/to/asterd
```

Common errors reports:

'Use of unitialized value'

'Invalid read' / 'Invalid write'

'Conditional jump depends on unitialized value'

May be ignored:

Errors in the Python library (especially before code_aster startup).

Some optimized functions (operations on strings).

Other tools

Debug jeveux

Force write of jeveux objects onto disk after each free (jelibe).

Detect write attempt in read-only objects.

Usage: from astk ‘Options / dbgjeveux’

JXVERI

Detect write after the bounds of a vector: ‘Ecrasement amont/aval’ reported by Jeveux.

Usage: add ‘call jxveri()’ around block of code.

Debug elementary computations

To compare different executions (on two machines or between two revisions, debug/nodebug).

Print a checksum of input and output fields at each call of ‘calcul’.

Compare outputs: check if input fields are identical but output changed.

Usage: set ‘dbg = .true.’ in calcul.F90.

(material field contains addresses, ignore changes)

End of presentation

Is something missing or unclear in this document?

Or feeling happy to have read such a clear tutorial?

Please, we welcome any feedbacks about Code_Aster training materials.

Do not hesitate to share with us your comments on the Code_Aster forum dedicated thread.