Octopus: Debugging and Profiling

Martin Lüders and the Octopus developers

Octopus Advanced Course 2023, MPSD Hamburg

Bugs

Bugs come in all shapes and forms:

- compile errors
  mostly trivial syntax errors
  Compiler will tell you the location
- segmentation faults
  access 'illegal' memory
  use internal or external debugger
- floating exceptions
  mostly division by zero
  use debugger (variable inspection)
- random failures
  mostly uninitialized memory
  valgrind can help
- wrong results
  tough one...

Tools to catch the bug

- Octopus internal debugging:
  - can give hints without recompiling the code
- Compiler flags:
  - can produce more detailed information on location of error
  - can stop te code on errors (rather than crashing at later point)
  - enable warnings on potential problems
- Debugger:
  - define breakpoints
  - investigate variable content at breakpoints
- Valgrind:
  - information on uninitialized memory
  - information on memory leaks

The Internal Debugger: Debug Options

- `Debug`

no	default: no debug output
info	extra information on terminal
trace	backtrace in case of crash
trace_term	full trace on terminal (cluttered!)
trace_file	separate trace files for each processor

- `DebugTrapSignals`

yes	default: Octopus handles exceptions and produces an error message good for production runs. Not good for debugging.
no	floating point exceptions (and other signals) are passed to the system and can be used by a debugger (e.g. `gdb`)

The Internal Debugger: What does it tell me?

- `Debug = trace`
  Backtrace of last routine (not necessarily complete, remember `PUSH_SUB` / `POP_SUB`)
  This information can also be produced with compiler options (`-g -backtrace`)
- `Debug = trace_file`
  Full trace for each processor.
  Interesting to see whether all tasks crash in the same routine.

Sometimes, this is enough to find the buggy routine, and find the error by inspection.
If not, a proper debugger might give more information.

Compiler flags (gfortran)

-g	enable debugging
-O0	disable optimization
-Og	optimization compatible with debugging
-backtrace	produce backtrace when a deadly signal is emitted
-ffpe-trap=invalid,zero,overflow	check floating point exceptions
-fbounds-check	check indices are within declared array bounds
-finit-real=snan	initialize all floating point numbers to signalling NaN
-fcheck=all,no-array-temps	Enable all run-time test of -fcheck (checks for pointers, allocatables, and memory allocation, bounds checks, modification of loop iteration variables), disable checks for temporary array creation
-frounding-math	disable tansformations that assume default floating point-rounging behaviour
-fstack-protector-all	Create extra code to check for buffer overflows
-Wall -Wextra -Wuninitialized	Make the compiler verbose and enable most of the warnings

Compiler flags

Recommended flags for development:

- All of the above!
- Catch possible errors as soon as possible!
- Much easier than finding a but after a big change or development.

The gnu debugger gdb

Need to compile with debug options

- `-g`: leave name symbols in the binary
  this includes: source line numbers, function names, variable names
- `-Og` or lower: avoid too high optimization
  optimization destroys mapping to source code

Set Octopus input variable: `DebugTrapSignals=No`, otherwise Octopus absorbs signals.

Start with sequential code:

```
gdb octopus
```

Post-mortem debugging:

```
gdb octopus core
```

gdb: Basic usage

gdb is commandline driven.

```
> gdb
...
(gdb) help
List of classes of commands:
aliases -- User-defined aliases of other commands.
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
status -- Status inquiries.
support -- Support facilities.
text-user-interface -- TUI is the GDB text based interface.
tracepoints -- Tracing of program execution without stopping the program.
user-defined -- User-defined commands.
Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Useful gdb commands

h[elp]	get help on gdb commands
h[elp] cmd	get help on a specific gdb command
start	start program and pause on first executable line
r[un]	run to next breakpoint or to end
s[tep]	single-step, descending into functions
n[ext]	single-step without descending into functions
fin[ish]	finish current function, loop, etc. (useful!)
c[ontinue]	continue to next breakpoint or end
f[rame] nr	change to another stack frame
up	go up one context level on stack (to caller)
do[wn]	go down one level (only possible after up)
l[ist]	show lines of code surrounding the current point
p[rint] name	print value of variable called name
b[reak] name	set a breakpoint at function name
h[elp] b	documentation for setting breakpoints
i[nfo] b	list breakpoints
i	list all info commands
dis[able] 1	disable breakpoint 1
en[able] 1	enable breakpoint 1
d[elete] 1	delete breakpoint 1
d	delete all breakpoints

gdb: Parallel code

```
mpirun -n 2 xterm -e gdb octopus
```

- This opens 2 terminal, each running one `gdb octopus` task.
- Each needs to be started.
- Each gdb is independent, but octopus communicates via MPI.

Memory debugging

- Many bugs are due to uninitialized memory
- This can lead to random failures.

Several tools to debug that:

- Compiler options:
  - -finit-real=snan -ffpe-trap=invalid:
    uninitialized variables will cause floating exception.
  - -finit-integer=:
    choose different values and see whether the behaviour changes
  - -fsanitize=address: check memory access out of bounds.
- `valgrind`:
  - Slow! Valgrind simulates the processor and monitors each memory access.
  - Compile without -finit-*, as this hides uninitialized memory from valgrind.
  - Valgrind needs some experience: Flags up some (many) false positives.

Running valgrind

Invoke valgrind with

```
valgrind --leak-check=full \
     --show-leak-kinds=all \
     --track-origins=yes \
     --verbose \
     --log-file=valgrind-out.txt \
     ./executable
```

Code performance

- Even though speed is not the only goal, we like to have the code as fast as possible
- Performance optimizations can be a big effort
- Important to find the relevant places in the code: where does the code spend most of the time?
- Pareto rule: "80% of the gains generally come from focusing on 20% of the code"
- Use profiling tools to investigate:

Tools to investigate:

- internal profiler:
  Octopus profiler: how much time spent in each function
- external profilers:
  - likwid: FLOPS, memory bandwidth, ... for functions
  - Intel vtune: time and other metrics on loop level
  - Advisor: roof line metrics on loop level
  - Nvidia Nsight systems: GPU profiling, data transfers, kernel launches

Internal profiling

- Set input variable: `ProfilingMode = prof_time`
- Output in `profiling/time.000000`
- Contains timings for regions in the code
- Self-time and cumulative times
- `ProfilingAllNodes = yes` to generate output for each MPI task
- There will be differences due to load imbalance

Example output

		CUMULATIVE TIME						SELF TIME				
TAG	NUM_CALLS	TOTAL_TIME	TIME_PER_CALL	MIN_TIME	MFLOPS	MBYTES/S	%TIME	TOTAL_TIME	TIME_PER_CALL	MFLOPS	MBYTES/S	%TIME
zNL_OPERATOR_BATCH	4060	3.258858	0.000803	0.000209	2591.9	0.0	35.3	3.258858	0.000803	2591.9	0.0	35.3
PS_FILTER	4	1.889388	0.472347	0.327342	0.0	0.0	20.5	1.889388	0.472347	0.0	0.0	20.5
SG_PCONV	101	0.954465	0.009450	0.009319	0.0	0.0	10.3	0.949822	0.009404	0.0	0.0	10.3
zVLPSI	2030	0.619451	0.000305	0.000166	563.1	1223.1	6.7	0.619451	0.000305	563.1	1223.1	6.7
zSET_BC	2030	0.365790	0.000180	0.000098	0.0	0.0	4.0	0.365790	0.000180	0.0	0.0	4.0
MESH_INIT	3	0.357447	0.119149	0.000002	0.0	0.0	3.9	0.344014	0.114671	0.0	0.0	3.7
zGHOST_UPDATE_START	2030	0.264002	0.000130	0.000069	0.0	0.0	2.9	0.259105	0.000128	0.0	0.0	2.8
CALC_DENSITY	505	0.222086	0.000440	0.000232	0.0	0.0	2.4	0.222086	0.000440	0.0	0.0	2.4
ELECTRONS_CONSTRUCTOR	1	0.383280	0.383280	0.383280	0.0	0.0	4.2	0.208536	0.208536	0.0	0.0	2.3
zPROJ_MAT_SCATTER	12180	0.159651	0.000013	0.000002	73.0	0.0	1.7	0.159651	0.000013	73.0	0.0	1.7
COMPLETE_RUN	1	9.226912	9.226912	9.226912	1051.6	84.5	100.0	0.144164	0.144164	0.0	0.0	1.6
HAMILTONIAN_ELEC_INIT	1	2.000087	2.000087	2.000087	0.0	0.0	21.7	0.110699	0.110699	0.0	0.0	1.2
zVNLPSI_MAT_BRA	2030	0.107823	0.000053	0.000034	1009.6	0.0	1.2	0.107823	0.000053	1009.6	0.0	1.2
LIBXC	202	0.073882	0.000366	0.000228	0.0	0.0	0.8	0.073882	0.000366	0.0	0.0	0.8
zVNLPSI_MAT_REDUCE	2030	0.065831	0.000032	0.000004	0.0	0.0	0.7	0.065831	0.000032	0.0	0.0	0.7
BLAS_AXPY_4	3000	0.064741	0.000022	0.000008	5971.5	0.0	0.7	0.064741	0.000022	5971.5	0.0	0.7
zGHOST_UPDATE_WAIT	2030	0.043132	0.000021	0.000011	0.0	0.0	0.5	0.043132	0.000021	0.0	0.0	0.5
BATCH_COPY_DATA_TO	1750	0.042083	0.000024	0.000009	0.0	0.0	0.5	0.042083	0.000024	0.0	0.0	0.5
zVNLPSI_MAT_KET	2030	0.248724	0.000123	0.000058	327.8	0.0	2.7	0.023242	0.000011	3007.3	0.0	0.3
dCUBE_TO_MESH	101	0.020717	0.000205	0.000191	0.0	443.9	0.2	0.020717	0.000205	0.0	443.9	0.2
XC_LOCAL	101	0.093789	0.000929	0.000875	12.8	0.0	1.0	0.019675	0.000195	0.0	0.0	0.2
zHAMILTONIAN	2030	4.943666	0.002435	0.001494	1817.6	153.3	53.6	0.018308	0.000009	0.0	0.0	0.2
EXP_TAYLOR_BATCH	500	5.000877	0.010002	0.006267	1912.0	149.3	54.2	0.016696	0.000033	0.0	0.0	0.2
dMESH_TO_CUBE	101	0.024494	0.000243	0.000231	0.0	375.4	0.3	0.015987	0.000158	0.0	575.2	0.2
POISSON_SOLVE	101	1.017069	0.010070	0.009940	0.0	18.1	11.0	0.015378	0.000152	0.0	0.0	0.2

Internal profiling: implmentation

- Define a profiling object and call `progfiling_in/profiling_out`

```
use profiling_oct_m
...
subroutine ...
  type(profile_t), save :: exp_prof

  call profiling_in(exp_prof, "EXPONENTIAL")
  ...
  call profiling_out(exp_prof)
end subroutine
```

Internal profiling: Other options

- `prof_io`: count number of file open/close operations
- `prof_mem`: summary of memory used and largest array
  Note: requires `SAFE_ALLOCATE`
- `prof_mem_full`: log every allocation / deallocation