Tutorial on Parallel Debugging
Victor Eijkhout
TACC HPC Training 2021
Better than finding errors is preventing them: defensive programming.

One possibility: Use ‘assertions’ about things that have to be true.

```c
#include <assert.h>
// for C++: #include <cassert>
assert( x >= 0 );
y = sqrt(x)
```

Program will terminate if the assertion fails.

Disable assertions in production by defining `NDEBUG`
Compiling for debug

Enable debug mode with `-g` option:

```
mpicc -g -O2 yourprogram
```

Debug option can be used with any optimization level, but sometimes good to start at `-O0`:

```
mpicc -g -O0 yourprogram
```

Compiler optimizations may confuse you otherwise.
Important! Note! About! Exercises!

1. You should have a directory exercises_ddt_c (or maybe f). Go there.
2. Start an interactive session: idev
3. Exercise slides will have a program name at the top: [roots]. This means you compile with make roots
4. Run your program with ./roots if sequential or ibrun roots for parallel.
Traditional sequential debugging
Debugging approaches

- Print statements:
  - can be effective, but they often perturb the behaviour: crashing code mysteriously works with print statements.
  - Also: the error is often not where you think it is.
  - Lots of recompilation.

- Interactive debuggers, different approaches:
  1. Start program in debugger
  2. Attach debugger to running program
  3. Do ‘post mortem’ analysis on ‘core dump’.
Interactive debuggers

- Commandline based tools:
  - *gdb* comes free with Gnu compilers; other debuggers are very similar (Apple has switched to *lldb*, which has different commands)

- Graphic frontends: Visual Studio, CLion, Eclipse, Xcode, ...

- Catch interrupts and inspect state of the program
- Interrupt a run yourself to inspect variables (breakpoints)
- Step through a program.
Example

- **Compile** `roots.c`: `make roots`
- Run the program, first on the commandline. Output?
- Execute this sequence of commands:
  - `gdb root`
  - `run`, **observe the output**
  - `quit`
Diagnosing the problem

- Floating point errors do not stop your program!
- In the debugger type:
  - `break roots.c:32` or whatever the first line of the root function is
  - `run` and note that it stops at the break point.
  - `where` displays the ‘stack frames’; `frame 3` to go there
  - `list` shows you the sources around the breakpoint
  - `print n` to show your the current value
  - `cont` to continue execution.

- **Better**: `break roots.c:32` if (n<0)
## More gdb

<table>
<thead>
<tr>
<th>command</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>run / cont</td>
<td>start / continue</td>
</tr>
<tr>
<td>break file.c:123</td>
<td>breakpoint at line</td>
</tr>
<tr>
<td>break &lt;location&gt; if &lt;condition&gt;</td>
<td>conditional stop</td>
</tr>
<tr>
<td>delete 1 / enable 2 / disable 3</td>
<td>break point manipulation</td>
</tr>
<tr>
<td>where</td>
<td>show call stack</td>
</tr>
<tr>
<td>frame 2</td>
<td>specific frame</td>
</tr>
</tbody>
</table>

For more commands see the cheat sheet in the course package.
Exercise 1 (roots)

You can force your execution to stop at floating point errors:

\texttt{feenableexcept}

Uncomment that line in the source, compile and run program, both commandline and debugger.
In the debugger, inspect the offending line in all frames.
Everyone’s favourite error: memory problems

- Write outside the bounds of an array (runtime checks are too expensive)
- Write to unallocated memory
- Read from uninitialized memory.

First two can usually be caught with a debugger; third one: use a memory tool like `valgrind`

```
module load valgrind

valgrind myprogram # sequential
ibrun valgrind myprogram # parallel
```
Exercise 2 \texttt{(array1)}

Compile and run \texttt{array1.c}.
(Look in the source to see the problem.)
If the program does not crash, recompile:

```
make clean array1 EXTRA_OPTIONS=5000
```

or even more.
Memory tools: valgrind

- At TACC: module load valgrind
- run with `valgrind array1`
- Look at the diagnostics. Do you understand them?
Same program in the debugger

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400b31 in main (argc=1, argv=0x7fffffff95a8) at array1.c:33
   squares[i] = 1./(i*i);
Missing separate debuginfos, use: debuginfo-install glibc-2.17-260.el7.x86_64
(gdb) where
#0 0x00000000000400b31 in main (argc=1, argv=0x7fffffff95a8) at array1.c:33
(gdb) print i
$1 = 5784
(gdb) print squares
$2 = (float *) 0x7fffffff95a0

After a while you ‘get a feel’ for what is a legitimate address and what is not. This is not.
Exercise 3 (array2)

Access out of bounds. Can you find the problem with the debugger or with valgrind?

Bonus exercise: what does valgrind say if you remove the initialization of sum?
Parallel debugging
Your minimal parallel debugger

mpirun -np 4 xterm -e gdb yourprogram

Pops up 4 xterms.
Great for debugging on your laptop.
Not great at scale.
The DDT debugger

Originally by Allinea, now bought by ARM.

- Graphical front-end to gdb-like and valgrind-like capabilities
- Some specifically parallel features
- Commercial, and with very few open source alternatives (Eclipse with PTP)
- An absolute life-saver!
Using the DDT debugger

Load the module:

```
module load ddt
```

Call the debugger:

```
ddt yourprogram
```
Graphics on a TACC cluster

- Through an X forwarding connection:
  
  ```bash
  ssh -X you@stampede.tacc.utexas.edu
  ```

- use VNC.

- use DCV [https://portal.tacc.utexas.edu/tutorials/remote-desktop-access](https://portal.tacc.utexas.edu/tutorials/remote-desktop-access):
  
  ```bash
  # submit DCV job:
  sbatch /share/doc/slurm/job.dcv
  # when the job is running:
  cat dcvserver.out
  ```

  The `dcvserver.out` file contains a URL: this gives a graphical terminal session in your browser.
DDT modes

- Start on login node, let DDT submit to queue you may need to wait a little while
- Start on compute node, DDT runs directly, not through queue
- Also ‘reverse connect’ and batch mode, see https://portal.tacc.utexas.edu/tutorials/ddt
Run parameters

- MPI or OpenMP? Processes, nodes, threads.
- Memory debugging
- Commandline arguments
- Check ‘submit’ when running on a login node: it submits to the queue for you; uncheck if starting from idev session.
Submission setup

- Project: your own, or one for this class
- Queue: development often quickest
Program starts at **MPI_Init**

- Use run controls
Hanging processes

- Red: stopped at an interrupt or breakpoint
- Green: still running.
  - All green but ‘nothing happening’: probably hanging program.
- Combination: some processes are not getting to the breakpoint: probably deadlocked.
Call stacks

- Hit the pause button, go to ‘stacks’ panel.
- Not every process is in the same source line.
- Click on process number to see what it’s doing.
Breakpoints

Set breakpoint by clicking left of the line
when you run, it will stop at the breakpoint.

Values display: everyone the same it
value of mytid linearly increasing
value of randomnumber all over the place.
Exercise 4 (finalize)

Compile and run finalize.c.
Every process completes the run, yet the program is incorrect.

- Uncomment the barrier command and rerun. What do you observe?
- Set a breakpoint inside the conditional. Do all processes reach it?
Exercise 5 (bcast)

Compile and run `bcast.c`. The program finishes, yet it is not correct. (Why?)

Recompile:

```
make clean
make bcast EXTRA_OPTIONS=-DN=100000
```

Does the program still complete?
Exercise 6 (sendrecvv1)

Another program that is incorrect, but that finishes because small messages slip through the network.

Replace MPI_Send with MPI_Ssend which enforces blocking behavior. Now what happens?
Exercise 7 (sendrecv2)

This code fixes the problem with sendrecv1. But is this sensible?

- module load tau
- Compile with TAU:
  make clean; make sendrecv2
- Run and generate trace files:
  make taurun PROGRAM=sendrecv2
- Postprocess:
  make tau PROGRAM=sendrecv2
- Somewhere with X windows:
  jumpshot tautrace_sendrecv2.slog2
Exercise 8 (isendrecv)

The proper solution is of course the use of MPI_Irecv.

Make a TAU visualization of a run of isendrecv.c. Is this optimal?