Introduction to Debugging

Stéphane Ethier
(ethier@pppl.gov)
http://w3.pppl.gov/~ethier/PICSCIE/DEBUGGING

Princeton University Research Computing Bootcamp 2018
October 31, 2018

Goals of this tutorial

- Give an overview of the debugging process
- Learn tips and tricks from the trenches
- Gain practical knowledge of
  - the debugging process
  - Simple debuggers for C/C++, Fortran, and Python
  - Advanced graphical debuggers

Debugging is not easy…

- When a code crashes it usually writes out a cryptic error message
  - The cpu processes machine instructions, not the C or Fortran source code that you wrote…
  - Instructions in the final executable program have probably been rearranged for optimization purposes
- You are in luck if the error happens at the beginning of the execution… usually it does not!
- In a large parallel code, how many processes triggered the problem? 1, 2, all of them? Was it related to inter-process communication?

Typical error messages

- **SIGFPE**: floating point exception
  - Often controlled be compiler options
  - division by zero
  - square root of a negative number
  - log of a number less or equal to zero
  - Does not always crash your code! (unless compiled to do so…)
    - The code can keep going for a long time with “NaN” values
- **SIGSEGV**: segmentation violation or fault (see “man 7 signal”)
  - invalid memory reference → e.g. trying to access an array element outside the dimensions of an array
  - double x[100]; x[345] = SIGSEGV!

Typical error messages

- **SIGSEGV** can also mean that you ran out of **memory**
  - Allocated memory using “malloc()” goes on the **heap** memory
  - “Local” arrays to a function are put on the **“stack”**
  - Make sure your shell **“stack size limit”** is set to “unlimited”
    - BASH: ulimit -s unlimited (put in .bashrc)
    - CSH: limit stacksize unlimited (put in .cshrc)
- I/O errors
- NO ERROR MESSAGE AT ALL!
  - The code just hangs
  - Usually points to a communication deadlock in a parallel code
  - The results are just plain wrong…

Take advantage of the compiler options

- Take the time to go through **all** the options of the compiler that you use
- Pay particular attention to the diagnostics options under sections with names such as “**debugging**”, “**optimization**”, “**target-specific**”, “**warnings**”
- Using “**man compiler_name**” is a good start although most compilers now have detailed online documentation → Just check the company’s web site under “support” or “documentation”
- (e.g. https://software.intel.com/en-us/intel-software-technical-documentation)

The **-g** compiler option

- All compilers accept the **-g** option
- It links the source code to the executed machine language code.
- The **-g** option is necessary when using a debugger unless you are REALLY good at deciphering machine language.
- However, using -g slows down the code significantly
  - Removes optimizations (unless one uses –gopt or –g –O3)
  - Start with "-g –O0" (no optimization) for most accurate correspondence between executable instructions and source code line
  - Inserts a lot of extra information in the executable to help the debugging process
- Running with "**-g**" is sometimes sufficient to find a bug. The code crashes and indicates where the error occurred

“-g” makes the bug go away!

- Sometimes, the fact of using the -g option makes the bug go away
- This does not necessarily mean that the optimized code generated by the compiler is wrong, although it could be…
- Can point to a memory issue, such as a pointer accessing a bad memory address when the optimized code is executed
- Look at your compiler’s documentation for how you can use the -g option while keeping most of the optimizations intact, such as -gopt for the PGI compiler (Portland Group), or simply “-g –O2” for Intel
  - caveat: it can point you to the wrong location in the source code

Examples of useful compiler options

- All compilers have options that try to detect potential bugs in the code
  - Array bounds check (Fortran: –C, -Mbounds, -check bounds)
    - Check for array subscripts and substrings out of bounds
    - Should be done on unoptimized code (-O0)
  - Easier for Fortran than C/C++ due to the way pointers are treated
    - In C it is the responsibility of the programmer to make sure that a pointer always points to a valid address and number

Examples of useful compiler options

- Enable runtime error traceback capability
  - --trace, -trace, -traceback
- Make sure that Floating Point Exceptions (FPE) are turned on (e.g. -fpe0 for Intel and -Ktrap=fp for PGI compiler)

Warning options in gcc

- The gcc compiler has a large number of options that will produce a warning at compile time
  - The all start with -W…
  - Example: **-Wuninitialized** warns if an automatic variable is used before being initialized
  - **-Wall** turns on most of the gcc warning options
  - **-Werror** makes all warnings into errors
  - Different levels of debugging information with **–g1**, **–g2**, and **–g3**
  - See “**man gcc**” or “**info gcc**”

Try different compilers if you can

- Whenever you can, it is always a good idea to try different compilers if you have access to different platforms or different compilers on the same platform
- Some compilers are a lot stricter than others and can catch potential problems at compile time
- See, for example, the following page for comparison between Fortran compilers in terms of available diagnostics:
  http://www.fortran.uk/fortran-compiler-comparisons-2015/intellinux-fortran-compiler-diagnostic-capabilities/

The code crashes… now what?!

- The first thing that you need to know is **where** the code stopped and **how** it got there
- Each time a program performs a function call, information about the call is generated. That information includes the location of the call in the program, the arguments of the call, and the local variables of the function being called
- This information is saved in a block of data called **stack frame**
- The stack frames are stored in a region of memory called the **“call stack”**

Saving the *stack* in "core" files

- How can one view the stack if the code crashed??
- When a code crashes, the system (normally) saves the last call stack of the code in files named "**core**" (or "core.#", where # is the rank number of an MPI task in a parallel code)
- No core file?
  - Check your shell limits: "ulimit –a" (bash) or "limit" (csh)
  - Look for "core file size" (bash) or "coredumpsize" (csh) > 0
- On most systems, these files are binary files meant to be read by debuggers
  - Exception: IBM Blue Gene saves the information in text form
- Of limited use if the code was not compiled with –g option that links machine language code to high-level source code

Examining the ***call stack***

- All debuggers should allow you to view the *call stack*, or simply *stack*
- Commands to look for in the debuggers
  - backtrace (gdb)
  - where
  - info stack
- Use the "apropos debug" command on your UNIX-based platform to find out which debugger is available
- If working on Linux (most cases), "gdb" should be available

The gdb debugger

- Official GNU debugger available under Linux
- Widely used for C and C++ code debugging
- Can also be used with Fortran codes
- Online manual for gdb at “info gdb”
- Can be used within the emacs editor
  - Can run gdb commands within the emacs source code window (e.g. C-x SPC to set a breakpoint)
- Online manual at http://www.gnu.org/software/gdb/current
- Search for “gdb tutorial”
- On MacOSX, the C compiler is LLVM (Clang) and the associated debugger is “**lldb**”, which is similar to “gdb” but better used via the **Xcode** GUI

Reading core files with gdb

- If code was compiled with –g and “dumped” core files when it crashed, the first thing to try is the following:

  gdb executable core.#

  (gdb) where (or backtrace, or bt)

- The “where” command prints out the *call stack*
- You can also use the DDT advanced debugger to open the core file and view the *call stack*…

Using “gdb”

- Compile with “-g -O0” to get accurate binary-to-source correspondence
- Start gdb: “gdb executable”
- You get the (gdb) prompt, where you can type the commands:

Command	Abbrev.	Description
help		List gdb command topics
run	r	Start program execution
break		Suspend execution at specified location (line number, function, instruction address, etc.)
step	s	Step to next line of code. Will step “into” a function.
next	n	Execute next line of code. Will NOT enter functions
until		Continue processing until it reaches a specified line
list	l	List source code with current position of execution
print	p	Print value stored in a variable

I know *where* the code crashes… what's next?

- Detective work starts
- Try reducing the problem size and see if the error is still there
  - The smaller the better
  - Running with only 2 processes is ideal if your code is parallel
- Start your code in a debugger (gdb or other) and set a breakpoint on a line executed before the crash
- Examine the values of variables and arrays by printing them out or visualizing them (advanced debuggers)
- Step through your code line by line until you find the problem
- Set other breakpoints to jump over long sections of code, such as loops
- If you know which variable goes bad, use a conditional breakpoint to run **"until" (gdb)** the variable changes to a given value
- Visualizing the results coming out of the code may help detect problems
  - Grid problems are often detected by visual inspection

Python debugger

- The “pdb” debugger is part of Python
- Just insert the following at any point in your Python code:

  import pdb
  pdb.set_trace()
- The execution will stop after these lines and will put you under pdb (you will have the **(Pdb)** prompt)
- Use “help” to see the commands

```
(Pdb) help

Documented commands (type help <topic>):
========================================
EOF     bt          cont      enable  jump  pp       run      unt
a       c           continue  exit    l     q        s        until
alias   cl          d         h       list  quit     step     up
args    clear       debug     help    n     r        tbreak   w
b       commands    disable   ignore  next  restart  u        whatis
break   condition   down      j       p     return   unalias  where
```

Please use checkpoint-restart

- Checkpoint = write out to files all the information that you need to restart a simulation from that point
- Extremely important for codes that have long runtimes (> 1 hour)
  - Allows you to restart your simulation at the point of the latest checkpoint
  - Avoid losing hours of precious computer time
  - Especially important for parallel codes ($$$)
- Extremely important when you need to debug a code that crashes after a few hours!!
  - You can recompile the code with –g and start from the last checkpoint
  - Remember… “-g” slows down the code dramatically so you want to be as close to the crash as possible

Using restart files

- When restarting a simulation from a checkpointed state, reproducibility is very important
  - Test by running the code to a certain point and saving its state at that point
  - Rerun the same case but split in 2 steps where the 2nd step uses restart files generated by the first step
  - Compare the end results of the 2 simulations → they should be identical
- The restart files need to be BINARY files
- When dealing with random numbers, use a reproducible random number generator for which you can save the state for restart purposes (e.g. SPRNG http://sprng.cs.fsu.edu/)

“printf” for monitoring and debugging

- Many developers still use “printf” statements to monitor and debug their codes
- May be the only recourse when running a code at very large concurrencies (100,000+ processors)
- The idea is simple:
  - Insert printf statements at strategic locations in the code to gather information and try to pinpoint the faulty code line
- Advantages over other forms of debugging
  - Easy to use and always works
  - Low overhead
  - Works on OPTIMIZED code! (CAVEAT: may change or prevent the optimization of a section of code. DON’T put in loops!)

printf debugging tips: write to stderr

- For optimization purposes, all code output is buffered before being written to disk unless directed otherwise
- If the code crashes before the memory buffers get written to disk, the information is lost
- It makes it difficult to pinpoint the exact location of the failing statement
- This is the case when using “printf” or “write(6,*)”
- Write to **standard error** as much as possible since it is **not buffered**
  - fprintf(stderr,…) in C/C++
  - “cerr <<“ in C++
  - write(0,*) in Fortran
  - redirect output: mpirun –np 1024 ./a.out >& output.out
- Explicit flushing of I/O buffers with “fflush()” (C) or “call flush(unit)” (Fortran)

A better printf: **ASSERT!**

- "assert(condition)" is a way to replace "if(condition)abort();"
- If "condition" is false, the code aborts and outputs the location (line number) of the failed test
- Good way to run "sanity checks" every time the code is executed → Helps catch bugs in the code
- Is part of C/C++ and Python
- Check for successful allocation: **assert(pointer != NULL)**
- Use functions "isnan()", "isinf()", "isnormal()" to catch **NaN** or **Inf** numbers (in math.h)
  - In FORTRAN, "**if (x /= x)**" is **TRUE if x = NaN**
- "assert" is equivalent to an "if" statement so it probably kills optimization in certain cases (may prevent vectorization on modern processors). Better to avoid inserting in important loops
- Watch out for truncation error. Do not use **assert(x==y)** if both "x" and "y" are floating point numbers

Extremely useful tool: **VALGRIND** (a bit slow though…)

- **Valgrind** (http://valgrind.org/)
  - Includes tools that can automatically detect many memory management and threading bugs
  - Profiles your programs in detail
- Currently includes six production-quality tools:
  - memory error detector
  - two thread error detectors
  - cache and branch-prediction profiler
  - call-graph generating cache and branch-prediction profilers
  - heap profiler
- Compile your program with **-g** to include debugging info
  - Using -O0 is also a good idea, if you can tolerate the slowdown
  - With -O1 line numbers in error messages can be inaccurate
  - Use of -O2 and above is not recommended as "**Memcheck**" occasionally reports uninitialised-value errors which don't really exist.

Valgrind example

```
1   #include <stdlib.h>
2
3   void f(void)
4   {
5   int* x = malloc(10 * sizeof(int));
6   x[10] = 0;          // problem 1: heap block overrun
7   }                      // problem 2: memory leak -- x not freed
8
9   int main(void)
10  {
11  f();
12  return 0;
13  }

$ cc -g -O0 -o test1 test1.c
$ valgrind --leak-check=full ./test1
```

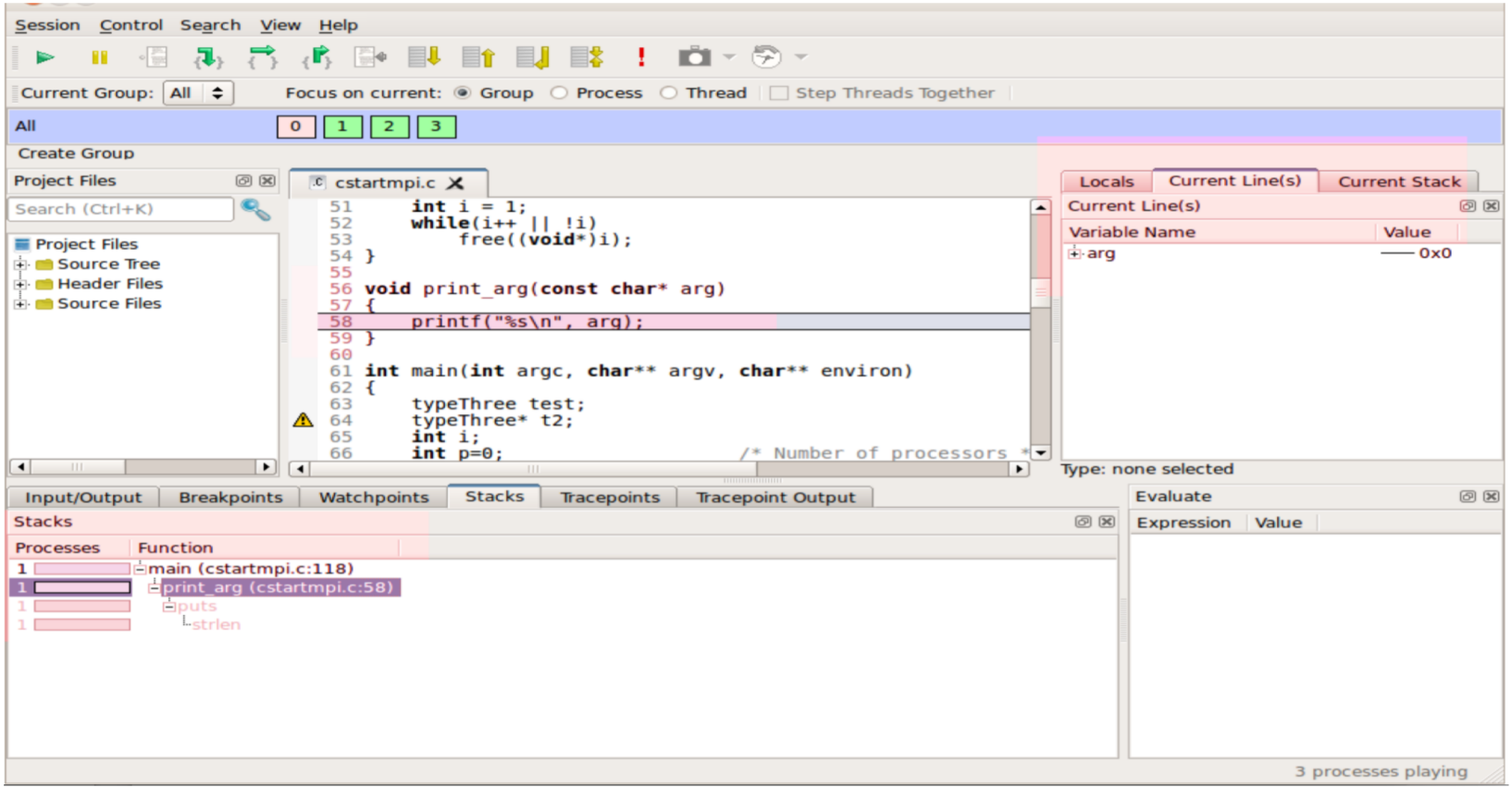
Valgrind example: results

```
==1519== Memcheck, a memory error detector
==1519== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==1519== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==1519== Command: ./test1
==1519==
==1519== Invalid write of size 4
==1519==    at 0x4005B6: f (test1.c:6)
==1519==    by 0x4005C6: main (test1.c:11)
==1519==  Address 0x5183068 is 0 bytes after a block of size 40 alloc'd
==1519==    at 0x4C23938: malloc (vg_replace_malloc.c:270)
==1519==    by 0x4005A9: f (test1.c:5)
==1519==    by 0x4005C6: main (test1.c:11)
==1519==
==1519==
==1519== HEAP SUMMARY:
==1519==     in use at exit: 40 bytes in 1 blocks
==1519==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==1519==
==1519== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1519==    at 0x4C23938: malloc (vg_replace_malloc.c:270)
==1519==    by 0x4005A9: f (test1.c:5)
==1519==    by 0x4005C6: main (test1.c:11)
==1519==
==1519== LEAK SUMMARY:
==1519==    definitely lost: 40 bytes in 1 blocks
==1519==    indirectly lost: 0 bytes in 0 blocks
==1519==      possibly lost: 0 bytes in 0 blocks
==1519==    still reachable: 0 bytes in 0 blocks
==1519==         suppressed: 0 bytes in 0 blocks
==1519==
==1519== For counts of detected and suppressed errors, rerun with: -v
==1519== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 7 from 7)
```

Advanced Graphical Debuggers

- For Python, the Eclipse developer environment with PyDev gives a nice GUI for debugging
  - See http://www.pydev.org/manual_adv_debugger.html
- The 2 most advanced debuggers for C/C++ and Fortran are:
  - Totalview
  - **ARM DDT** (ARM Forge = DDT + MAP)
- Both of them are designed to debug large-scale parallel codes
- All the HPC systems on campus have DDT
  - Type "/usr/licensed/bin/ddt &" ("&" is to run in background)
  - Or "/usr/licensed/bin/ddt –start *executable*"
  - Same idea as gdb: step through your code and examine variables
  - See https://developer.arm.com/docs/101136/latest/ddt/getting-started

ARM DDT window



Important links

- https://developer.arm.com/docs/101136/latest/ddt
- https://www.princeton.edu/researchcomputing/
- https://www.princeton.edu/researchcomputing/faq/debugging-with-ddt-on-the/

- https://w3.pppl.gov/~ethier/PICSCIE/DEBUGGING/training_ddt.tar.gz
  - Can use “wget” on Linux or “curl –O” on MacOS
  - tar –zxvf training_ddt.tar.gz to unpack the files. Will creat a directory “training_ddt”

Final remark

Don’t waste your time… use a debugger!