Debugging Guide

Debugging Guide (Adapted for Users and Developers in the field of Earth System Modeling)

"*Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*"

Brian Kernighan / Co-author of the first book on C

Debugging Commons

Software debugging …

- is *identifying* and *correcting* existing or potential problems ("bugs") in a program or software system;
- is a *multi-step process* from identifying a problem to fixing it or finding a solution;
- is an *integral part* of the entire software and model development process;
- can be approached in a *systematic* and *predictable way* to maximize effectiveness

Why it is important:

- Testing and debugging takes 50-75% of the time in a software project
- Cost of software bugs to the U.S. economy: nearly $60 billion a year (2002 study); global cost of debugging software: $312 billion annually (2013 study) cost of poor software quality in the U.S. was over $2 trillion/year (2020 report).
- Debugging is a difficult process because of the involvement of humans.

Types of common errors include the following:

- **Syntax or typo errors.** Could be caught by a good code editor or compiler. An error message typically indicates a source of the problem.
- **Runtime errors**. A problem occurs during the model runtime, and could cause a fatal run or unexpected behavior. The source of errors could be a code issue or an input data issue.
- **Implementation errors**. The code may be inefficient, or data could be manipulated incorrectly.
- **Logic errors**. The algorithm is logically flawed.

Types of debugging analysis:

- **Static analysis** - the code is examined without executing the program
- **Tracing analysis** - watching live or recorded printout or log files and monitoring workflow. Log files are text files documenting action that happened in the application
- **Profiling** - analyzing time spent in different parts of a program during a run
- **Remote debugging** - running code on a different system from the one it was developed

Types of software tools that serve debugging needs:

- Text editor, or, preferably, a code editor with language-specific syntax checking. Example: GVIM, Visual Studio
- Compiler flags for compile or runtime code issue reporting
- Command-line debuggers for runtime error catch. Example: GDB, valgrind
- GUI (Graphical User Interface) based debugging and profiling tools for serial and parallel codes. Examples: TotalView, Arm Forge/Allinea (DDT, MAP, PerformaceReview)

Best practices and effective strategies

Most of the strategies could be reduced to the following 8 rules:

1. Understand the System
2. Reproduce the Error
3. Look First, Think Second
4. Divide and Conquer
5. Change One Thing at a Time
6. Keep an Audit Trail
7. Get a Fresh View
8. If you Didn't Fix It, It Is Not Fixed

1. Understand the System

Learn the system, know the fundamentals, verify requirements and prerequisites, read the manual or a user's guide. Know the roadmap of the software system workflow and your implementation of the workflow. If you are not sure how it works, you won't know why it fails.

If the parts of the system are designed rather like "black boxes", e.g., subroutines or modules, find out how they are supposed to interact with the other parts, so you could determine whether the problem is inside the box or outside of the box. Verify data requirements and formats, know that's reasonable behavior or output. Don't guess it, look up the details (see Note below)!

If you assume that parameters to a function are in the correct order, check them. Verifying your assumptions and guesses not only helps prevent skipping past the problem, but avert more confusion because of falsely reassuring information.

parallels

“If you can’t fix the flood in your basement at 2am and decide to call the plumber, don’t guess the number, look it up!”

2. Reproduce the Error

The first thing to do after encountering the issue is to reproduce it, and make it fail again. If the error cannot be reproduced, there is no way to verify that it was fixed. For intermittent errors that cannot be reproduced reliably, you need to know the circumstances when the problem occurred. If possible, artificially recreate the environment and circumstances to stimulate the failure (*simulate* the environment to *stimulate* failure), use known data for testing.

parallels - humor

Q.: How many engineers does it take to fix a lightbulb?

A.: None; they say, “I can’t reproduce it — the lightbulb in *my* office works fine”.

3. Look First, Think Second

See the failure; look at each time it fails. Make sure you see what is actually going wrong, not what you guess might be a problem. Record system output, capture information on every run, so you can look at it after the failure. If the error is intermittent, observe and learn the differences between a good case and a bad case, using as many identical diagnostics as possible. Add in more diagnostics: system state, check variables, examine pointers, check memory allocations, event turning relationships (conditional statements), error flags, and function calls, including and their exit codes, their parameters, and their return values. Make guesses only to focus the search.

Be aware that your debugging activities could require more compute resources, and thus could become a part of the system, known as the “Heisenberg Uncertainty Principle”.

Heisenberg Uncertainty Principle

This principle is named after one of the pioneers in quantum physics, who was working with atomic particles. He noticed that estimation of a particle location affects measurements on where it was going, because measurements probes then become part of the system.

4. Divide and Conquer: The Binary Search

Determine what side of the bug you are on: narrow the search for the error location with successive approximations. A good start is to go halfway through the program and test for the presence of the error. Then go upstream or downstream from the halfway point and set another break point midway through that section of code to see if the error surfaces or not. Continue this process iteratively (see Note below). Start with the bad: check the workflow where it’s broken first, and then work your way up to the cause. Use easy-to-spot test patterns or parameters.
Fix the known bugs first. Know the ranges of the accepted values of your variables.

parallels

In a similar way, you can guess the number from 0 to 100 in 7 guesses or less.

5. Change One Thing at a Time

When testing possible solutions, change one thing or apply one fix at a time. If the fix did not solve the problem, back it up right away. Test another solution. Compare a failed case with a good case, narrow down the differences to only the bug, and look at the log files side-by-side. Large volumes of identical irrelevant information could be more easily skimmed over because of their similarity, while the differences found in the logs should be observed carefully. Simplify the cases to focus on the differences in outcome. In the case where the study or simulation worked before, determine what has changed since then.

6. Keep an Audit Trail

While the value of an audit trail is accepted, the level of details is often under-appreciated. *Write down* what you are doing, in what order, what you see, what happens, any system details. When testing solution hypotheses, write down your current test, what was the expected result and what was the outcome. Why keep *written* log notes? 1) No need to memorize it; you can pick it up and resume at any time. 2) Revise the tests later, so you avoid repeating them. Even if you have a great memory, keeping everything in your head requires extreme concentration (*). This could be okay if this takes 5 minutes, but debugging usually takes longer. The shortest pencil is longer than the longest memory. 3) Seeing details of the tests and written notes repeatedly may reveal a solution at hand. Correlate the events between the error symptoms or debug information.

7. Get a Fresh View

It’s always a good idea to explain the issue to another person, friend, or a family member. By reconstructing, streamlining, and verbalizing a whole story, possible solution(s) often become clear. After all, asking for help does not reflect poorly on you but rather shows your eagerness to solve the problem; by choosing your help wisely, the issues can be fixed faster. Tap into expertise, and listen to the voice of experience. Take pride in getting rid of the bugs faster, not in getting rid of them by yourself!

When describing the problem, report symptoms, not theories. Ask for help to get a fresh insight, without imposing your bias or theories. However: if data looks wrong and you suspect it may be related to the problem, this is worth presenting. You don't have to be sure: this data may not be relevant, but it *is* information.

8. If You Didn't Fix It, It Is Not Fixed

Check that it's really the solution you applied that solved the problem. Did you fix it, or did you get lucky? Apply the "Reproduce the Error" rule to the design, take the fix out and make sure it fails again, then put the fix back in and see it working. This last step is important because often during debugging other things of the work sequence are modified that are not the part of the official "fix". When a random factor affects the outcome, it may actually fix or hide the problem. Only when you cycle from a working solution to the error, and then back to working again, you prove that a correct fix to the problem was indeed found.

Other Practical Debugging Methods

- Add diagnostics reported in output or log files, check variables and states. Check the state of the system by estimating minimum, maximum, sum, or average values of fields.
- Get time profiling of the sections of the code, modules or routines. Time stamp specific locations in the code to report during model run. Could be helpful for debugging parallel programs.
- Explore compile-time debugging options
- Use runtime debugging levels
- Use simplified test data

Debugging Tools (commonly used software packages)

Vi, **VIM** ("vi improved"), Lightweight and efficient text and code editors for the command-line interface (vi, vim, view) and GUI-supported (gview, gvim). These are FREE. Basic versions of vi and vim are included in all Unix/Linux and MacOS X operating systems. These options are freely available to download and install.

Download: https://www.vim.org/. Quick start with vim:
https://www.geeksforgeeks.org/getting-started-with-vim-editor-in-linux/

Emacs, GNU emacs - An extensible, customizable, free/libre text editor. Most GNU/Linux systems include emacs in their repositories; could also be installed on MacOS, Windows, OSX.
https://www.gnu.org/software/emacs/ , https://www.gnu.org/software/emacs/download.html

Visual Studio Code, or VS Code - a free source-code editor built by Microsoft for Windows, macOS, and Linux. It has support for most common programming languages, including syntax highlighting.
Downloads and docs: https://code.visualstudio.com/, Quick start: https://code.visualstudio.com/docs/introvideos/basics

Microsoft VisualStudio - another product by Microsoft, a standalone IDE and source code editor for Windows, macOS, and Linux. https://visualstudio.microsoft.com/ . A free version is available for individuals. Quick start: https://docs.microsoft.com/en-us/visualstudio/ide/quickstart-ide-orientation?view=vs-2022

GDB - GNU Project Debugger - an open-source debugger that uses a command-line interface. It allows you to run a program interactively, and pause it to examine its variables and state. Could be useful for interactive serial programs. Builds and runs on Unix/Linux, Mac OS X. Download a *tar file and build from the source: https://www.sourceware.org/gdb/, or clone from github:
https://sourceware.org/git/binutils-gdb.git. GDB is created to work with GNU compilers.
GDB tutorial: https://www.tutorialspoint.com/gnu_debugger/index.htm

Valgrind - debugger and code profiler for programs for major Linux distributions and is useful for memory management and threading bugs. Valgrind is a command-line tool, but also has GUI options, such as **Valkyrie**. More on https://valgrind.org/info/ . Builds from source and downloads from https://valgrind.org/downloads/current.html.
Quick start: https://www.cprogramming.com/debugging/valgrind.html

LLDB is a part of the LLVM suite of tools for Apple, and is default debugger in XCode on MacOS and built to work with CLang, for programs in C, Objective-C, and C++. Its Linux version is developed and improving. It could also be used as a command-line standalone debugging tool, https://lldb.llvm.org/ . Tutorial: https://lldb.llvm.org/use/tutorial.html

VTune - Intel VTune Profiler - a performance optimization tool for serial and parallel programs and system configurations, including HPC and cloud. VTune is available for free as a stand-alone tool or as part of the Intel oneAPI Base Toolkit; it is usually available along with Intel compilers on HPC systems. It can be run as an application with a graphical interface (vtune-gui), as a command line (vtune). More information:
https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.htm
User's Guide: https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html

TotalView -a powerful debugger for HPC applications written in C, C++, Fortran, Python, and could be used for multithreaded, multi-process and GPU-specific applications.
Free trial is available, https://totalview.io/; it requires a license for the full use. Downloads: https://totalview.io/downloads
Video tutorials: https://totalview.io/support/video-tutorials
It is often installed on HPC systems, system-wide or loadable as a module.

Arm Forge - is a high-end software suite for HPC code optimization and development, consisting of a debugger**, Arm DDT**, a visual performance profiler for code optimization, **Arm MAP**, and a performance analyzer, **Arm Performance Reports**. (Also former: **Allinea DDT, Allinea Map, Allinea Performance Reports, Forge**). A license is needed. Is installed as a separate module on HPC systems.

https://www.arm.com/products/development-tools/server-and-hpc/forge

Debugging software packages on NOAA's HPC systems (RDHPCS):

Cheyenne:

Installed system wide:
VIM - vi (vi, vim, view, gview) - Vi IMproved 7.4.326
GDB: GNU gdb (GDB; SUSE Linux Enterprise 12) 8.3.1
Loadable modules:
vim/8.1
nano/4.3
valgrind/3.17.0
xxdiff/4.0.1 - graphical `diff` utility
ARM Forge: arm-forge/19.1, 20.0.2, 20.1.1, 20.2, 21.0.3, 21.1.1, 22.0.2
(ddt, forge, map, perf-report)
ARM Reports: arm-reports/19.1, arm-reports/20.0.2 (ddt-debugger, perf-report)

Hera:

Installed system wide:
VIM - Vi IMproved 7.4.629
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
Loadable modules:
valgrind/3.16.1
xxdiff/3.2.Z1
Allinea Forge (ddt, map, perf-report): forge/21.0, forge/22.0, forge/22.0.2
VTune Amplifier (Intel's): vtune/2019, vtune/2020, vtune/2020.2

Gaea:

Installed systemwide:
VIM - Vi IMproved 8.0
GDB - GNU gdb 10.1
Loadable modules:
emacs/26.2, emace/26.3
cray-lgdb/3.0.10
Arm Forge DDT: ddt/19.0.4, ddt/20.1, ddt/21.0.3
gdb4hpc/3.0.10, gdb4hpc/4.10.6
valgrind4hpc/1.0.0, valgrind4hpc/2.10.2
pdt/3.25.1 - Program Database Toolkit
perftools/7.0.6, perftools-base/7.0.6, perftools-base/7.1.3, perftools-base/21.05.0, (other versions of perftools-lite, and other perftools-<name>)
iobuf/2.0.8, iobuf/2.0.9, iobuf/2.0.10

Jet:

Installed system wide:
vi, vim - Vi Improved VIM, version 7.4
gdb, GNU gdb for Red Hat Enterprise Linux 7.6.1-120.el7
Loadable modules:
valgrind/3.16.1
xxdiff/3.2.Z1 - graphical diff utility
advisor/2019, advisor/2020, advisor/2021: Intel Advisor is a design assistance and analysis tool for SIMD vectorization, threading, memory use, and GPU offload optimization.
vtune/2019, vtune/2020, vtune/2020.2
inspector/2019, inspector/2020, inspector/2020.2 (previously known as Intel Thread Checker) is a memory and thread checking and debugging tool to increase the reliability, security, and accuracy of C/C++ and Fortran applications.
forge/21.0, forge/22.0

NOAA's PCluster/AWS, Azure, GCluster (Google):

Installed system wide:
vi, vim - Vi Improved VIM, version 7.4.629
gdb, GNU gdb for Red Hat Enterprise Linux 7.6.1-120.el7
Loadable modules:
valgrind/3.16.1
xxdiff/3.2.Z1 - graphical diff utility
inspector/2019, inspector/2020, inspector/2020.2 (previously known as Intel Thread Checker) is a memory and thread checking and debugging tool to increase the reliability, security, and accuracy of C/C++ and Fortran applications.

vtune/2020, vtune/2020, vtune/2020.2
https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.8qteid

advisor/2019, advisor/2020, advisor/2020.2: Intel Advisor ("Advisor XE", "Vectorization Advisor", "Threading Advisor") - is a tool to analyze code and determine the costs and benefits of adding various threading models on Intel processors. It works on code written in C, C++, and Fortran, and can model parallelism using OpenMP, Intel Thread Building Blocks, and Intel Cilk Plus. Advisor can also provide guidance to help codes get better vectorization https://www.intel.com/content/www/us/en/developer/articles/release-notes/advisor-release-notes.html

References

Agans, David. 2002. "Debugging. The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems". Publisher: AMACOM, a division of American Management Association. 183pp.

Zeller, Andreas. "Software Debugging" a course on https://www.udacity.com/, course SC259.

Thomas, David; Hunt, Andrew. 2019. The Pragmatic Programmer. 20th Anniversary Edition. 2nd Edition. ISBN: 9780135957059. Hardcover Publisher: Pearson. Audio book: https://www.audible.com/pd/The-Pragmatic-Programmer-20th-Anniversary-Edition-2nd-Edition-Audiobook/B0833FMYH9