Debugging and Profiling

Mark Howison

User Services & Support
Debugging: Overview

- What is a bug?
- What is debugging?
- Preventing bugs
- Segfault example with two debugging tools:
  - catchsegv
  - gdb
What is a bug?

- Programs that crash or return unexpected output usually contain a "bug", which can be from:
  - Oversights during coding: a simple typo or syntactic mistake in the source code
  - Mis-communication or mis-understanding among developers: a library function is incorrectly documented
  - A fundamental flaw in the design of a system or algorithm
- Bugs almost always arise due to human factors (it isn't the machine that is broken!)
- They may or may not be easily reproducible
Common bugs

- Off-by-one errors
- Out-of-bounds memory access (or buffer overflow)
  - Can lead to stack corruption
- Assignment (=) where you meant comparison (==)
- Misnamed variables or functions
- Division by zero
- Integer overflow
- Dereferencing a null pointer
- Using an initialized value
What is debugging?

- The art of systemically (or not) determining where a bug is located in your program
  - Test the assumptions/beliefs you have about your program, until you find one that is wrong
  - Many tools exist (printf, catchsegv, gdb, TotalView)
  - Almost all require that you compile your program with “debug symbols” (use -g with most compilers)
    - Some bugs disappear when you recompile with -g (“heisenbugs”)
  - The Binary Search principle:
    - Check half-way through your program for the bug
    - Search again in either the first or second half
    - Repeat recursively until you find the line containing the bug
Preventing bugs

- Usually it is worth investing time in the following software engineering practices to reduce the time you spend debugging:
  - Using consistent and well-accepted programming techniques and styles
  - Pre-planning and drafting program specifications before programming
  - Adding assertions and other debug code as you program
  - Creating testing frameworks (i.e. “unit testing”)

- Standard references:
  - “Writing Solid Code” by Steve Maguire (C language)
  - “Code Complete” by Steve McConnell (C++ language)
catchsegv

- If you program crashes with a segfault:
  - Recompile with `-g`
  - Run your program through `catchsegv`:
    ```
    catchsegv ./myprogram [arg1 arg2 ...]
    ```
  - This preloads a library that will intercept the segfault signal and print out a stack trace
  - Usually this will quickly point you to a line in your source code where you are accessing memory that is out-of-bounds
gdb

- Start your program within gdb then set the arguments:
  
gdb ./myprogram

  This GDB was configured as "x86_64-redhat-linux-gnu"
  (gdb) set args arg1 [arg2 ...]
  (gdb) run

- After a segfault or other crash occurs, print a backtrace with:
  
  (gdb) bt

- Many features exist in gdb to handle other kinds of bugs: setting breakpoints, querying values, etc.
Profiling: Overview

- What is profiling?
- Easy serial profiling
- Memory profiling with Valgrind
- OProfile for batch jobs
- IPM for MPI applications
- Python profiling
- CUDA profiling
What is profiling?

- Analyzing the behavior of a program *while it is running*
- Different methods:
  - *Sampling* means polling the status of the program at regular time intervals
    - Resulting profiles are statistical – not exact
    - Usually has minimal interference with program's runtime
  - *Tracing* or *instrumenting* means interposing profiling calls within your program's regular calls
    - More exact, but also more resource-intensive
- Usually refers to analyzing CPU performance, but can also apply to memory or I/O
Why or when to profile?

- During performance optimization: goal is to find and remove bottlenecks in your program
- You can find performance problems by...
  - Making a wild guess (“Maybe it is X”)
  - Making an educated guess (“Last time it was X, so maybe it is X this time too”)
  - Taking advice from an expert (“Prestigious Author says it is usually X”)
- OR you could profile to collect evidence that helps you narrow down the possibilities
- There is a trade-off between the time it takes to profile vs. for guess and check
  - Utility of profiling usually grows with size/complexity of code
Easy serial profiling

- Simplest way to sample: stop your program where it is taking a long time, and see what it is doing
- Works if your program runs on timescales >1s
- "ezprofile" script on Oscar does just this
- It wraps functionality from the binutils package:
  - "pstack" shows a stack trace of a running program
  - "addr2line" matches a program address with its corresponding line in source code
  - Requires compiling with debug symbols (-g)
Memory profiling with Valgrind

- Lots of functionality, but also high overhead
  - Your program could run many times slower

- Available on Oscar with:
  module load valgrind

- Use on serial programs with:
  valgrind program [args]

- Good for finding memory leaks
  --leak-check=full

- Will also identify the cause of most segfaults
  (debugging more than profiling...)
The Linux kernel comes with a built-in profiler

You can use it to profile an *entire system*

- Can also filter the data to isolate a specific program

Requires root access to load the kernel module and start/stop the daemon

But on Oscar, this can be done *automatically* for your batch job if you qsub with “-T oprofile”

The profile data will be saved to:
/gpfs/scratch/shared/oprofile/〈jobname〉.〈jobid〉/〈node〉
Viewing OProfile data

- “opreport” parses the OProfile output
- Overview of entire system:
  opreport -session-dir=<profile_data>
- By individual programs/libraries:
  opreport -l -session-dir=<profile_data> [/path/to/program]
- “opannotate” can correlate profile results with lines in your source code
IPM for MPI programs

- The Integrated Performance Monitor will show high-level stats about MPI communication in your program
- The MPI library exposes “hooks” for intercepting each MPI call, which IPM uses to start/stop timers
- Scalable, low overhead
- Available on Oscar with: 
  module load ipm
- Once loaded, automatically profiles any mpirun program
  - Uses LD_PRELOAD to load itself before your program
IPM output

- Writes a summary to stdout after your program calls MPI_Finalize()
- Set IPM_REPORT=full to enable more detail in the summary
- Outputs a detailed XML file to: `/gpfs/scratch/shared/ipm/<username>.<id>.xml`
- Use `ipm_parse` tool to parse XML file
IPM output (Cont'd)

- Example of plots that can be generated from XML:
  (see http://ipm-hpc.sourceforge.net/)

![Example of plots](http://ipm-hpc.sourceforge.net/)

- MPI_Barrier
- MPI_Send
- MPI_Waitall
- MPI_Reduce
- MPI_Alltoall
- MPI_Allreduce
- MPI_Irecv
- MPI_Gather
- MPI_Comm_size
- MPI_Comm_rank
Python profiling

- Use the cProfile module
- To profile a function from within a script or the interpreter:
  ```python
  import cProfile
  cProfile.run('func()' [, 'output_file'])
  ```
- To profile an entire script from the command line:
  ```bash
  python -m cProfile [-o output_file] ...
  ```
CUDA Profiling

- Built in profiler will provide information on data transfers and kernel execution
- Simply set the appropriate environment variables:
  - CUDA_PROFILE=1 (turns profiling on)
  - CUDA_PROFILE_CONFIG=file (points to text file that lists performance counters)
  - CUDA_PROFILE_CSV=1 (enables CSV output; easier to import into Excel, etc.)
- For list of counters and other options, see “doc/ComputeProfiling.txt” in the CUDA Toolkit
- Helpful overview from SC10:
Performance tips

Use Existing Optimizations

- Link against optimized libraries when you can (BLAS, LAPACK) instead of reinventing the wheel
  - Although these libraries may be optimized for very large data, so if you have small data, it can still be better to write your own routine
- Use MPI collectives instead of point-to-point communication when possible
  - Usually the collectives have additional optimizations that are specific to the system you are running
Performance tips (Cont'd)

In Your Own Code

- Pay attention to how you order multi-dimensional arrays:
  - C expects the last dimension is sequential in memory
  - Fortran expects the first dimension is sequential
  - If you use the wrong layout, your program will stride through memory very inefficiently!

- Function calls have some overhead (usually worse in C++)
  - Sometimes it can help to force small functions to be “inlined” (meaning copied in place instead of called)

- Conditionals within nested loops can be expensive because of branch mispredictions
In Your Own Code

- Replace expensive operations like division, exp, log, trig functions, etc. with precomputed lookup tables
  - Only works if you are operating on the same set of values over and over again
  - Sometimes you can also find versions that are faster but less precise, if that is acceptable for your computation

- In nested C loops, use the "__restrict__" keyword to indicate when arrays are disjoint (they don't overlap)

- Avoid lots of mallocing/freeing or new/delete operations
  - Also consider using a malloc replacement like Hoard