HPC VT 2013

Testing and Debugging
Scientific Software
Overview

- Testing
  - Definitions
  - Industry VS Academia
  - Methodology
  - Principles

- Localizing Bugs
  - Debugging
  - GDB
  - Valgrind / Memgrind
Two classical software “fails”

- **Cluster / Ariane 5 rocket**
  - Overflow at conversion of double to uint16
  - Rocket went off-course had to self-destroy
  - Cost: 370 million US$

- **Therac 25 medical accelerator**
  - No checking of user input caused counter overflow
  - Interlock selected wrong energy source
  - 6 patients got overdoses 100x the intended dose
A software bug is an error, flaw, mistake, failure, or fault in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways.

Common types: Arithmetic, Logic, Syntax, Resource, Concurrent, Interface, Performance, ...
Software testing is a formal process carried out by a specialized testing team in which a software unit, several integrated software units or an entire software package are examined by running the programs on a computer. All the associated tests are performed according to approved test procedures on approved test cases.
Bill Gates at 17th Conference on Object-Oriented-Programming, Seattle, 2002/11/8:

“... When you look at a big commercial software company like Microsoft, there's actually as much testing that goes in as development. We have as many testers as we have developers. Testers basically test all the time, and developers basically are involved in the testing process about half the time...”

“... We've probably changed the industry we're in. We're not in the software industry; we're in the testing industry, and writing the software is the thing that keeps us busy doing all that testing.”

“...The test cases are unbelievably expensive; in fact, there's more lines of code in the test harness than there is in the program itself. Often that's a ratio of about three to one.”
Testing: Industry

• 20-50% of effort goes into testing depending on the development model

• Example: Test Driven Development (TDD):
  • Exhaustive suite of programmer tests
  • No code goes into production unless it has associated tests
  • You write the tests first
  • The tests determine what code you need to write
Testing: Academia

• Academia is different:
  • Mostly single or small group of researchers involved in development
  • Project are not fully defined at the starting point
  • Requirements change fast
  • Often little time to invest into testing and documentation
• Most scientific programs are poorly tested
  • Tests are written at the end of the development (if..)
  • Asserts are used only to avoid critical passages
  • Often poor exception and error handling when user input is used

• Hear now how to do it better!
Testing levels

• **Unit Testing**
  • Verify specific section of code, component or class

• **Integration Testing**
  • Verify interfaces between components

• **System Testing**
  • Test the whole program (“top-down”)
Designing Tests

- **Fixture**
  - What is the test run on (defined by *function* and *input*)

- **Action**
  - What is done to the fixture (mostly *calling the function*)

- **Expected Result**
  - What *should* happen

- **Actual Result**
  - What *actually* happened

- **Report**
Aim of software tests

• Identify test cases which have the highest probability to detect wrong behaviour of the system.

• A good test has a high functional coverage.

• A test aims to find bugs in programs; it is successful if it found a bug. Think about:
  • How can a correct output be characterized?
  • How can you catch an incorrect result?
Computing the series

\[(X^0)/0! + (X^1)/1! + (X^2)/2! + (X^3)/3! + (X^4)/4! + \ldots + (X^n)/n!\]
A test can be...

- Runnable specification
- Examples & part of documentaiton
- Status report of development

- Therefore, if you give them away try to keep them **readable & reliable**
When to start testing?

It is important to start testing early!
Good functional coverage is important, although exhaustive testing is impossible!
Unit tests: Frameworks C/C++

- MinUtit
  - Minimal and light-weight
- Check
  - More functionality and asserts
- CppTest
  - Very useful: TEST_ASSERT_DELTA(a,b,delta)
- Google C++ Testing Framework
  - Often used in industry
- Boost.Test library
  - Popular in scientific computing
If a test fails..

- Think about the logical cause
- Try to localize bug
- Use assert
  - assert(size <= LIMIT);
- Use printf
  - printf("Iteration %d output %f", it, out);
- Debugging
Print contents of variables and data structures

Write print routines for displaying the contents of your data structures in a readable format.

Print data in condensed form, e.g. the norm of a vector instead of all the vector entries.

Use the more or less tools for examining output, or use a text editor.

Display results graphically.
Localizing bugs

Compare and contrast

A working program in another language, on another system etc is a very valuable resource

BUT: Can the "working program" be trusted?

Subproblems, or simplified problems, can be easily implemented in e.g. Matlab

Use the same test data for both codes!

Regression testing by giving identical input to “known good” stable version
Localizing bugs

Things to try if you can't find a bug:

- Identify code areas that do NOT contain the bug
- Remove chunks of code until bug disappears
- Fix hacks & kluges, restructure code
- Rewrite the buggy code from scratch
- Write code on paper and execute
- Use exceptions and error handling during coding if you're unsure of the correctness!
Debugging

Find causes of errors in your code
Executes your code like an interpreter
  You can step through your program
  Set breakpoints
  Print and set variables
  Catch run-time exceptions

Two standard debuggers
  GNU gdb
  dbx
Compiling for debugging

To be able to refer to names and symbols in your source code it must be stored in the executable file

Standard flag “-g”

A “debug build” can also enable lower optimization (-O0), debug mode in libraries (asserts)

Without “-g” you can only debug using virtual addresses
Computing the series

\( \frac{X^0}{0!} + \frac{X^1}{1!} + \frac{X^2}{2!} + \frac{X^3}{3!} + \frac{X^4}{4!} + \ldots + \frac{X^n}{n!} \)
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>step [count]</td>
<td>Next line of code, step into functions</td>
</tr>
<tr>
<td>next [count]</td>
<td>Next line of code, execute function</td>
</tr>
<tr>
<td>list</td>
<td>Print surrounding source</td>
</tr>
<tr>
<td>where</td>
<td>Print call hierarchy (stack)</td>
</tr>
<tr>
<td>print expr</td>
<td>Print the value of a variable</td>
</tr>
<tr>
<td>break</td>
<td>Set breakpoints (many options)</td>
</tr>
</tbody>
</table>
More on GDB

Type **help** in the GDB prompt

**help breakpoints, help running**

Check out the gdb man page

GDB can also attach to already running processes owned by you

There is a graphical interface to GDB called the “Dynamic Data Debugger” DDD

$ ddd ./a.out

Allows you to visualize data and breakpoints
malloc() and free() – with friends like these, who needs enemies?

Can cause problems “far from” where the bug really is
Common to only occur for special data sets
May occur in a seemingly non-deterministic way
Influence from the environment (system load etc)
May be found in codes that were supposed to be “tested and verified”, e.g. when porting to another system
Memory bugs

Unallocated memory

double *A;
...
for (i=0; i<n; i++)
   A[i] = 0.0;

Can cause the execution to stop when the variable is used or an attempt to deallocate the memory is made

Most compilers can check for unallocated memory
Memory bugs

Overwriting or “overreading” memory
Common error in C and Fortran: Using arrays outside the array bounds

- `double * A = (double*) malloc(N*sizeof(double));`
- `...`
- `for(i=0; i <= N; i++)`
- `Diff[i]=(A[i+1]-A[i])/h;`

May result in erroneous results or program crash (page fault in invalid page)
Program crash may occur later, e.g. when calling memory allocation routine

Many compilers can generate code that checks array bounds during execution (add option for this). NOTE: This reduces performance!

Handled in Java and e.g. Pascal (Runtime error)
Memory bugs

Stack buffer overrun/overflow

Special case of memory overreading/overwriting where you run out of stack space.

Caused by:

- Too many very large stack variables
- Too many (recursive) function calls
Dangling pointers

A pointer to a piece of memory that once was allocated to a variable, but then was deallocated

Common error when copying and deallocating derived types with dynamically allocated fields

Another common error: returning a pointer to a local variable

Hard to debug! Some debugging tools may help you

0xBAADFOOD, 0xDEADBEEF
Memory leaks

Allocated memory is lost without deallocation

The program runs out of memory for large problems, many iterations ...

Hard to detect. Use e.g. `top` to monitor memory usage

Hard to debug. Instrumented system call libraries may give some information; allocation “tagging”.

Valgrind memcheck can help

Handled in e.g. Java, and other languages with built-in garbage collection
Memory bugs and testing

- Tests often miss memory bugs or
- Tests only *sporadically* catch memory bugs
- What to do?
  - Check memory addresses
  - Isolate program components
  - Use Valgrind
Valgrind

• Use of uninitialised memory
• Reading/writing memory after it has been free'd
• Reading/writing off the end of malloc'd blocks
• Reading/writing inappropriate areas on the stack
• Memory leaks -- where pointers to malloc'd blocks are lost forever
• Mismatched use of malloc/new/new [] vs free/delete/delete []
• Overlapping src and dst pointers in memcpy() and related functions
• Some misuses of the POSIX pthreads API

% valgrind --tool=memcheck program_name
...
=18515== malloc/free: in use at exit: 0 bytes in 0 blocks.
==18515== malloc/free: 1 allocs, 1 frees, 10 bytes allocated.
==18515== For a detailed leak analysis, rerun with: --leak-check=yes
Valgrind will NOT perform bounds checking on static arrays (allocated on the stack)

X 2 memory usage

Program will have terrible performance

Valgrind may not find all the errors – especially those that don't happen every time.
Computational bugs

Floating-point problems

Cancellation

Precision (using numbers of very different magnitude)

Accuracy of intrinsic functions and constants (embedded, GPUs)

Strict equality is always dangerous. Always use: \( \text{fabs}(x-y) < \text{EPSILON} \), not \( x == y \)

Ordering is crucial!

Stress testing: Include “extreme inputs” in test data set
Logical bugs and typos

Just about anything…
Mixing up negation
Assigning when intending equality
Misusing a library, or another part of your code

"That case will never arise"

"I use this argument in a clever way to indicate two things"
What to do?

Tedious code is tedious to test
Respect compiler warnings (use -Wall)

*Reusing* code is good, *copying* code is a slippery slope

Use the libraries of others:
Data structures, common routines & operations
e.g. Boost (www.boost.org)
Highly optimized code is ugly code

Remember your priorities!

- Correctness
- Flexibility
- Performance