Lecture 24 – Debugging and Profiling Tools

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F22)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

- Overview of Debugging
- Debugging Tools
  - Help from the Compiler
  - From the Command Line (GDB)
  - Memory Debugging (Valgrind)
  - Fancy Debugging Tools (ARM DDT)
- Overview of Profiling
  - Types of Performance Data
  - Collecting Performance Data
  - Analyzing Performance Data
- Performance Analysis Tools

Learning Objectives: By the end of Today's Lecture you should be able to

- (*Knowledge*) Classes of bugs and observed behaviors
- (*Skill*) Debugging at the most basic level
- (*Knowledge*) What types of tools are available for debugging
- (*Skill*) debugging with gdb, valgrind, and ddt

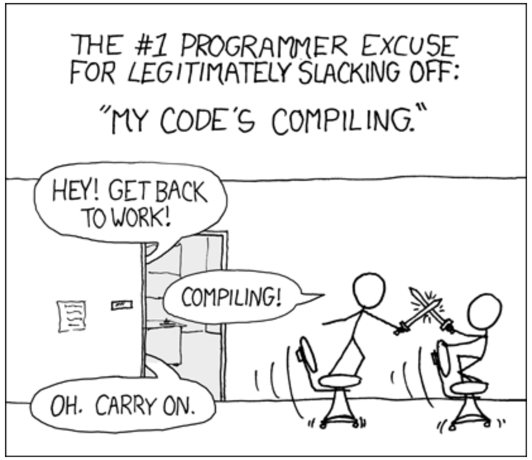
Overview of Debugging

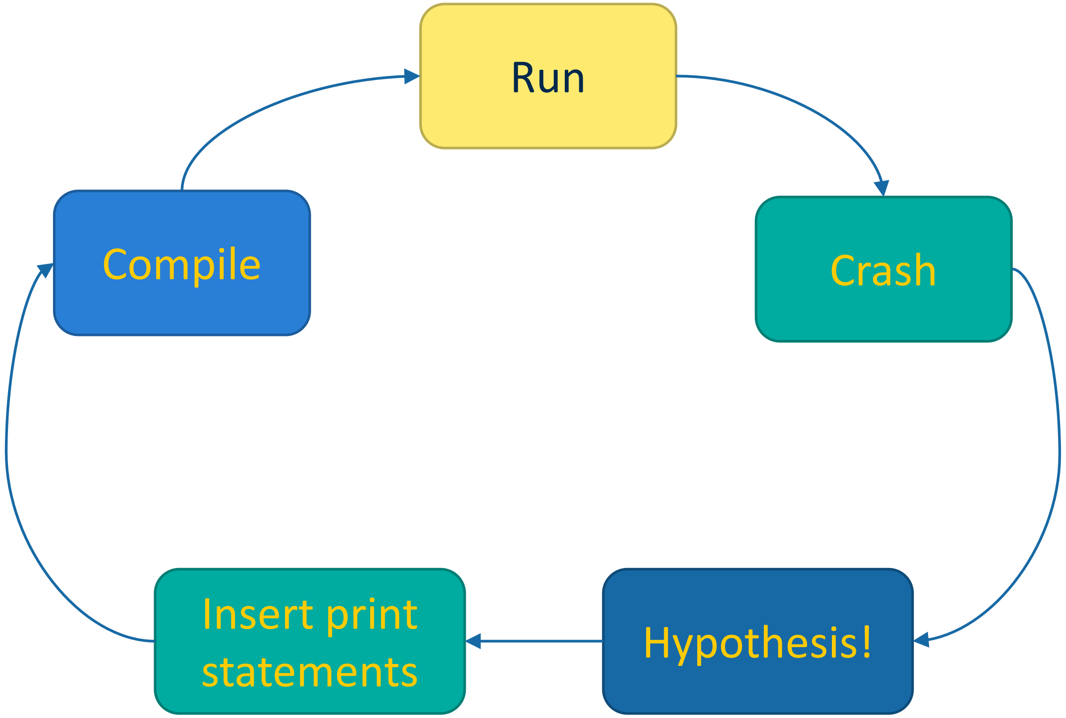
What is debugging?

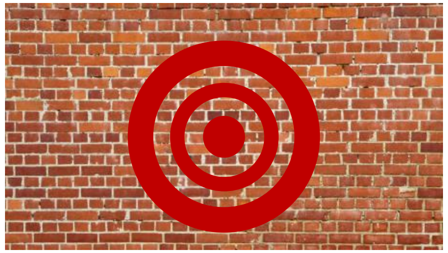
- The art of transforming a broken program to a working one

- Debugging requires thought – and discipline:

Debugging in Practice...







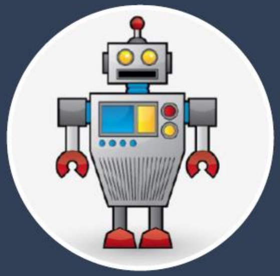
Bang Head Here

A Taxonomy of Bugs

Name	Description
Bohrbug	Steady, dependable bug
Heisenbug	Vanishes when you try to debug (observe)
Mandelbug	Complexity and obscurity of the cause is so great that it appears chaotic
Schroedinbug	First occurs after someone reads the source file and deduces that it never worked, after which the program ceases to work

From presentation: David Lecomber, “Debugging and Profiling your HPC Applications”

Common Approaches to Debugging



Automation
- Test cases
- Bisection via version control

Observation
- Print statements
- Debuggers

Inspiration
- Explaining the source code to a duck

Magic
- Static analysis
- Memory debugging

From presentation: David Lecomber, “Debugging and Profiling your HPC Applications”

Compiler Supported Debugging

What compilers are good for in debugging

- Catching typos
  - Gives a compile time error
- Enforcing variable definitions/type consistency

- Adding instrumentation
  - Checking array bounds
- Providing callstack information
  - backtrace
- Can add warnings, and make these errors

- Integrating source code information for other debugging tools

Summary of Options

GCC compiler option	Meaning
`-g`	Produce debugging information in the operating system's native format.
`-fbounds-check`	Generate code to check that indices used to access arrays are within the declared range during run time.
`-fstack-check`	Generate code to verify that you do not go beyond the boundary of the stack.
`-fbacktrace`	Prints the call stack when there are run time errors
`-fcheck-pointer-bounds`	Each memory reference is instrumented with checks of the pointer used for memory access against bounds associated with that pointer.
`-fsanitize=<opt>`	Enable AddressSanitizer, a fast memory error detector.
`-Wall`	Enable all warnings
`-Wpedantic`	Issue all the warnings demanded by strict ISO C and ISO C++
`-Werror`	Treat warnings as compiler errors

GNU Debugger (GDB)

Debugging with GDB

- Invoked with `$ gdb <exec> [<exec_args>...]`
  - Generates a new command line interface
- A few basic commands…

Command	Description
help	Display help
break	Add a breakpoint
delete	Remove breakpoint(s)
continue	Run program if stopped (till next breakpoint)
step	Executes one line of source
next	Like step, but does not enter functions
list <linenumber>	Print source near line number
print <expression>	Print the result of an expression

Memory Debugging

Memory Debugging with Valgrind

- Essentially a “virtual machine” that simulates all memory traffic and catches errors.
    - Actually a collection of tools, but we’ll focus on memcheck
- Primarily used for serial code
    - Difficult to scale to large applications
- http://www.valgrind.org

<u>Basic Usage</u>

1. Compile your software with debug symbols (e.g. `-g`)
2. Rerun your program with the command:

```
$ valgrind <myexe> [<myexe_arguments>...]
```

Types errors detected by Valgrind

- Memory leaks
  - Memory is allocated but not deallocated
  - Primarily happens with pointers
  - Easier to do in C/C++
  - Serious problem if performed in a loop
- Illegal read/writes
  - When program accesses illegal memory addresses (e.g. beyond the bounds of an array).
- Use of uninitialized values
  - Serious if this happens to be in a branching statement

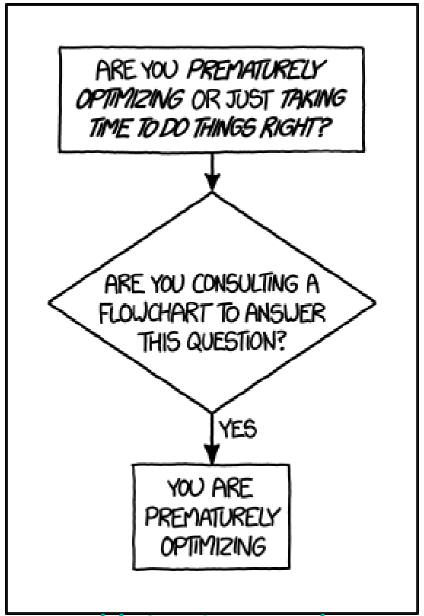
Fancy Debuggers

List of "Fancy" Debuggers

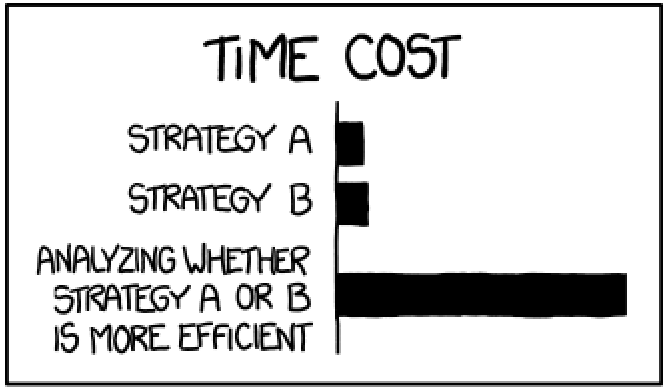
- Various GDB "front ends"
  - atom, vscode, gdbgui, Eclipse, Kdevelop
- DDT – Best option for leadership class computers
  - This is the best, but costs $$$. Fortunately, most (good) computing centers have this installed
- Totalview – Used to be best tool, but likely surpassed by DDT
  - Costs $$$ some computing centers have it
- STAT – Stack Trace Analysis Tool
  - Free! https://github.com/LLNL/STAT

Overview of Profiling

Why do Profiling?



https://xkcd.com/1691/



THE REASON I AM SO INEFFICIENT

https://xkcd.com/1445

Types of Performance Data

Metrics (Schmetrics)

Primitive

Derived

Profiling vs Tracing

Profile

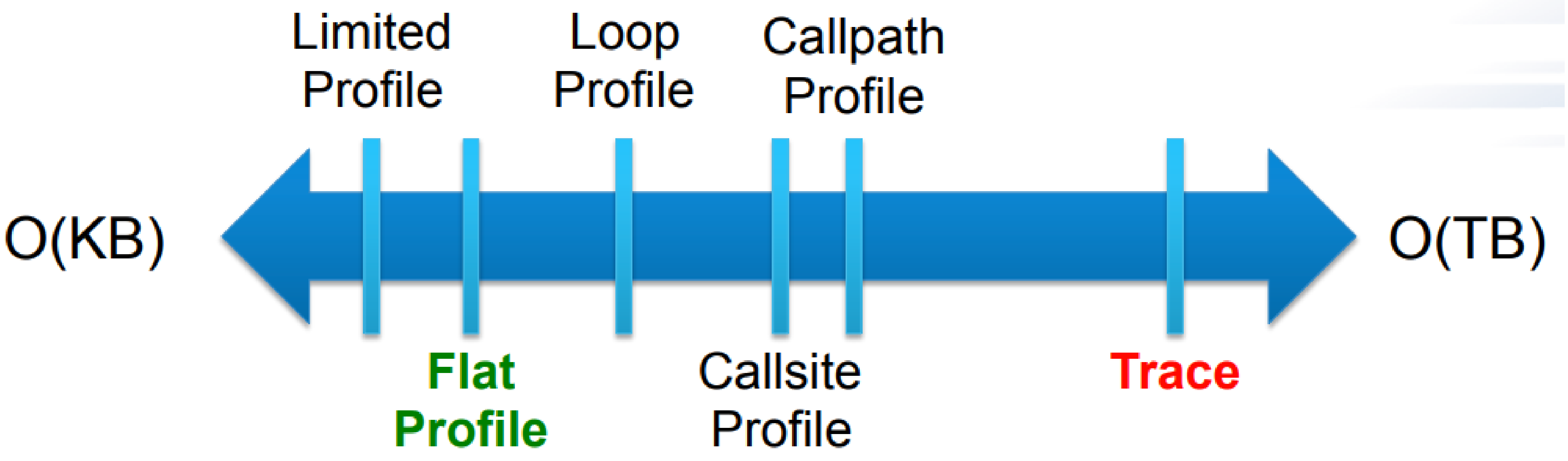
Trace

Inclusive vs Exclusive

ALGORITHM 6.9 *GMRES*

1. Compute $r_0 = b - Ax_0$, $\beta := \|r_0\|_2$, and $v_1 := r_0/\beta$
2. For $j = 1, 2, \ldots, m$ Do:
3. Compute $w_j := Av_j$
4. For $i = 1, \ldots, j$ Do:
5. $h_{ij} := (w_j, v_i)$
6. $w_j := w_j - h_{ij}v_i$
7. EndDo
8. $h_{j+1,j} = \|w_j\|_2$. If $h_{j+1,j} = 0$ set $m := j$ and go to 11
9. $v_{j+1} = w_j/h_{j+1,j}$
10. EndDo
11. Define the $(m+1) \times m$ Hessenberg matrix $\bar{H}_m = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$.
12. Compute y_m the minimizer of $\|\beta e_1 - \bar{H}_m y\|_2$ and $x_m = x_0 + V_m y_m$.

How much data do you want?



Collecting Performance Data

Instrumenting Code

ALGORITHM 6.9 *GMRES*

1. Compute $r_0 = b - Ax_0$, $\beta := \|r_0\|_2$, and $v_1 := r_0/\beta$
2. For $j = 1, 2, \ldots, m$ Do:
3. Compute $w_j := Av_j$
4. For $i = 1, \ldots, j$ Do:
5. $h_{ij} := (w_j, v_i)$
6. $w_j := w_j - h_{ij}v_i$
7. EndDo
8. $h_{j+1,j} = \|w_j\|_2$. If $h_{j+1,j} = 0$ set $m := j$ and go to 11
9. $v_{j+1} = w_j/h_{j+1,j}$
10. EndDo
11. Define the $(m+1) \times m$ Hessenberg matrix $\bar{H}_m = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$.
12. Compute y_m the minimizer of $\|\beta e_1 - \bar{H}_m y\|_2$ and $x_m = x_0 + V_m y_m$.

Overhead from Instrumentation

ALGORITHM 6.9 *GMRES*

1. Compute $r_0 = b - Ax_0$, $\beta := \|r_0\|_2$, and $v_1 := r_0/\beta$
2. For $j = 1, 2, \ldots, m$ Do:
3. Compute $w_j := Av_j$
4. For $i = 1, \ldots, j$ Do:
5. $h_{ij} := (w_j, v_i)$
6. $w_j := w_j - h_{ij}v_i$
7. EndDo
8. $h_{j+1,j} = \|w_j\|_2$. If $h_{j+1,j} = 0$ set $m := j$ and go to 11
9. $v_{j+1} = w_j/h_{j+1,j}$
10. EndDo
11. Define the $(m+1) \times m$ Hessenberg matrix $\bar{H}_m = \{h_{ij}\}_{1\le i\le m+1, 1\le j\le m}$.
12. Compute y_m the minimizer of $\|\beta e_1 - \bar{H}_m y\|_2$ and $x_m = x_0 + V_m y_m$.

Automatic Instrumentation and Sampling

Scaling Studies

- Strong Scaling: fixes problem size and increases number of processors.
  - Provides insight into how finely grained an algorithm can be parallelized and how much parallel overhead there is relative to useful computation
- Weak Scaling: fixes problem size *per process* and increases number of processors.
  - Provides insight into whether the parallel overhead varies faster or slower than the amount of work as the problem size is increased.
- Speedup and Efficiency:

$$S\left(P_{size}, N_p\right) \equiv \frac{T\left(P_{size}, 1\right)}{T\left(P_{size}, N_p\right)}$$

$$E_{strong}\left(P_{size}, N_p\right) \equiv \frac{T\left(P_{size}, 1\right)}{N_p \times T\left(P_{size}, N_p\right)}$$

- Good efficiency does not necessarily mean you have fast code
  - It could mean you have terrible serial performance

$$E_{weak}\left(P_{size}, N_p\right) \equiv \frac{T\left(P_{size}, 1\right)}{T\left(P_{size} \times N_p, N_p\right)}$$

Performance Analysis

How to interpret your data?

Low Computational Intensity

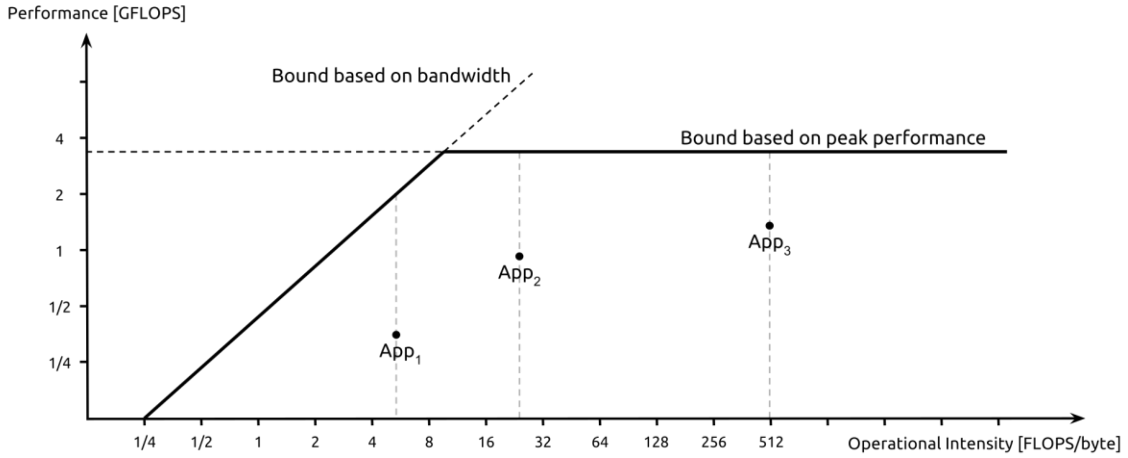
Simple Performance Model

$$T = Ft_F\left(1 + \frac{t_M}{t_F}\frac{L}{F}\right)$$

F= # of FLOPs
L=# of loads and stores
t_F = time for flop
t_M= time for memory load/store
T = execution time

$$\frac{L}{F} = \frac{1}{q}$$

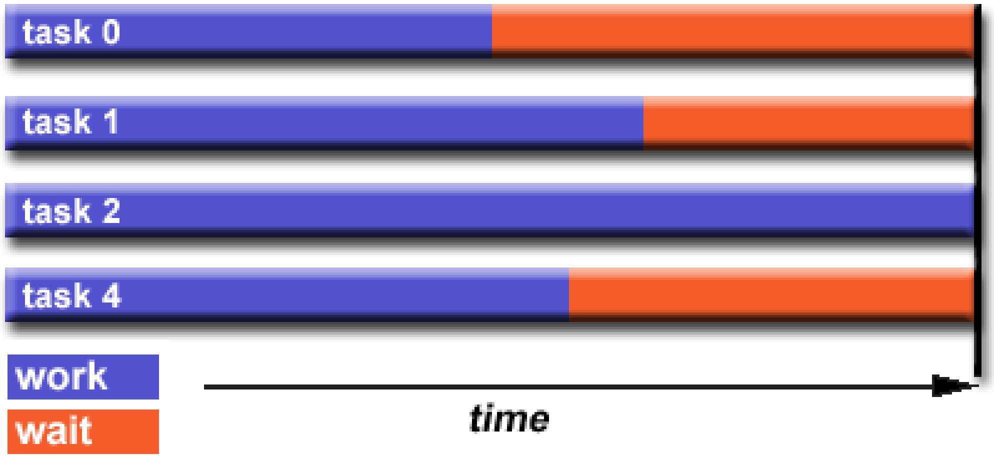
Roofline Model



Load Balance

Problem

- Poor strong scaling
- Poor parallel efficiency
- Increase cores by factor of 2x, do not observe 2x speedup.



Solution

- Determine what the load balance/imbalance is
  - Need to assign a value of "work" to each subdomain.
  - What is the maximum to minimum workload for all domains.
- Change partitioning to improve load balancing
- Change parallel algorithm

Parallel Execution Time Models

Moving from serial to parallel

- Serial Latency based model

$$T_{serial} = Ft_F + \alpha_1 L + \sum_{j=1}^{\kappa-1} \left(\alpha_{j+1} - \alpha_j\right) M_j + \left(\alpha_{mem} - \alpha_\kappa\right) M_\kappa$$

- Parallel Model

$$T_{parallel}\left(N_p\right) = \frac{T_{serial}}{N_p} + T_{overhead}\left(N_p\right)$$

- Difficult to develop exact expressions,
  - Alternatively measure realistic average values based on microbenchmarks.

Canonical Execution Time Models

- Distributed Memory Computing
  - Point-to-Point Communication Time

$$T_{comm} = \underbrace{\alpha_{network}}_{\text{Latency}} + \underbrace{\beta_{network}}_{\text{1/Bandwidth}} \underbrace{N}_{\text{Amount of data}}$$

  - Collective operations have their own (depends on algorithm implemented in library)

$$T_{\text{All\_reduce,small}} = \lceil \log p \rceil \left(\alpha_{network} + \beta_{network} \times N + \gamma \times N\right)$$

$$T_{\text{All\_reduce,large}} = 2 \log p \, \alpha_{network} + \frac{p-1}{p}\left(2\beta_{network} \times N + \underbrace{\gamma}_{\substack{\text{Time to perform reduce operation} \\ \text{(e.g. sum, max, multiply, etc.)}}} \times N\right)$$

Fundamentals of getting good parallel performance

- Maximize amount of work that can be parallelized.
- Minimize overhead.
- Usually this means
  - Balance work loads among processors
  - Avoid synchronization
    - Especially for shared memory
  - use non-blocking communication
    - Primarily in distributed memory models
- Make sure the serial code is optimized.

$$T_{parallel}\left(N_p\right) = \underbrace{T_{non-parallel}}_{\text{Minimize}} + \overbrace{\frac{T_{serial}}{N_p}}^{\text{Assumes perfect load balance}} + \underbrace{T_{overhead}\left(N_p\right)}_{\text{Minimize}}$$

Performance Analysis Tools

Performance Utilities and Tools