Coding Tools

(Lectures on High-performance Computing for Economists VI)

Jesús Fernández-Villaverde,[1] Pablo Guerrón,[2] and David Zarruk Valencia[3]

October 22, 2018

[1]University of Pennsylvania

[2]Boston College

[3]ITAM

Compilers

Compilers

- If you use a compiled language such as `C/C++` or `Fortran`, you have another choice: which compiler to use?

- Huge differences among compilers in:

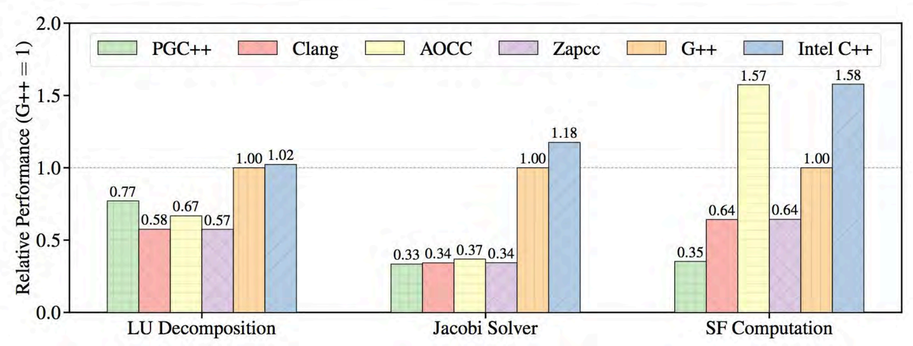
  1. Performance.
  2. Compatibility with standards.
  3. Implementation of new features:

     `http://en.cppreference.com/w/cpp/compiler_support`.

  4. Extra functionality (`MPI`, `OpenMP`, `CUDA`, `OpenACC`. ...).

- High return in learning how to use your compiler proficiently.

- Often you can mix compilers in one project.


PGC++
Clang
AOCC
Zapcc
G++
Intel C++
Relative Performance (G++ = 1)
2.0
1.5
1.0
0.5
0.0
0.77
0.58
0.67
0.57
1.00
1.02
0.33
0.34
0.37
0.34
1.00
1.18
0.35
0.64
1.57
0.64
1.00
1.58
LU Decomposition
Jacobi Solver
SF Computation

Linux/64 on Intel Processor

	Absoft	Absoft(AP)	gfortran	Intel	Intel(AP)	NAG	Oracle	PGI	open64
	17.0	17.0	5.2.0	17.0	17.0	6.1	12.5	16.9	4.5.2.1
AC	5.01	4.96	6.36	4.47	4.44	7.32	18.91	6.84	5.07
AERMOD	11.35	11.51	15.60	11.52	12.26	18.64	11.13	11.98	15.79
AIR	3.50	2.18	3.16	2.60	1.98	4.94	2.78	3.32	3.46
CAPACITA	20.26	17.75	19.21	16.83	17.32	22.17	21.49	15.36	19.33
CHANNEL2	73.53	28.58	83.00	84.76	28.95	105.87	84.26	81.57	103.80
DODUC	19.28	19.34	18.70	15.12	14.97	24.20	15.96	18.11	18.72
FATIGUE2	63.09	66.65	67.08	55.62	55.75	117.37	82.94	89.12	77.66
GAS_DYN2	73.96	49.13	86.23	62.25	38.42	177.37	74.91	111.19	79.02
INDUCT2	83.68	76.03	80.99	71.82	50.96	132.20	138.92	127.38	144.11
LINPK	5.23	5.49	4.93	4.37	4.47	6.22	4.70	5.96	5.73
MDBX	9.68	7.98	8.07	6.47	4.85	8.68	8.54	9.08	9.35
MP_PROP_DESIGN	120.85	13.13	157.61	62.78	10.97	254.30	196.16	88.35	127.22
NF	8.13	8.21	7.30	7.58	7.54	9.00	8.98	8.47	7.96
PROTEIN	21.58	21.16	21.13	23.68	25.02	20.31	22.09	23.22	21.95
RNFLOW	15.55	15.23	13.66	12.52	9.65	16.66	17.34	17.00	21.54
TEST_FPU2	61.24	43.00	50.15	43.44	39.64	82.37	64.90	48.28	57.10
TFFT2	58.66	61.10	46.74	58.95	62.43	60.41	58.91	56.78	58.55
Geometric Mean	22.93	17.39	22.95	19.33	14.96	30.95	26.44	24.23	25.45

The GCC compiler collection

- A good default option: `GNU GCC 8.2` compiler.

  1. Open source.
  2. `C`, `C++`, `Objective-C`, `Java`, `Fortran`, `Ada`, and `Go`.
  3. Integrates well with other tools, such as `JetBrains`' IDEs.
  4. Updated (`C++17`).
  5. Efficient.
  6. *An Introduction to GCC*, by Brian Gough,

     `http://www.network-theory.co.uk/docs/gccintro/`

The LLVM compiler infrastructure

1. LLVM (http://llvm.org/), including Clang.
    1.1 It comes with OS/X and Xcode.
    1.2 Faster for compiling, uses less memory.
    1.3 Run time is slightly worse than GCC.
    1.4 Useful for extensions: Cling (https://github.com/root-project/cling).
    1.5 Architecture of Julia.
2. DragonEgg: uses LLVM as a GCC backend.

Commercial compilers

1. `Intel Parallel Studio XE` (in particular with MKL) for `C`, `C++`, and `Fortran` (plus a highly efficient `Python` distribution). Community edition available.
2. `PGI`. Community edition available. Good for `OpenACC`.
3. `Microsoft Visual Studio` for `C`, `C++`, and other languages less relevant in scientific computation. Community edition available.
4. `C/C++`: `C++Builder`.
5. `Fortran`: Absoft, Lahey, and NAG.

Libraries

Libraries I

- Why libraries?
- Well-tested, state-of-the-art algorithms.
- Save on time.
- Classic ones
  1. `BLAS` (Basic Linear Algebra Subprograms).
  2. `Lapack` (Linear Algebra Package).

Libraries II

- More modern implementations:

  1. `Accelerate Framework` (OS/X).
  2. `ATLAS` (Automatically Tuned Linear Algebra Software).
  3. `MKL` (Math Kernel Library).

- Open source libraries:

  1. `GNU Scientific Library`.
  2. `GNU Multiple Precision Arithmetic Library`.
  3. `Armadillo`.
  4. `Boost`.
  5. `Eigen`.

Build Automation

Build automation

- A build tool automatizes the linking and compilation of code.
- This includes latex and pdf codes!
- Why?
  1. Avoid repetitive task.
  2. Get all the complicated linking and compiling options right (and, if text, graphs, options, etc.).
  3. Avoid errors.
  4. Reproducibility.
- `GNU Make` and `CMake`.

Why Make?

- Programed by Stuart Feldman, when he was a summer intern!
- Open source.
- Well documented.
- Close to Unix.
- Additional tools: `etags`, `cscope`, `ctree`.

Basic idea

- You build a make file: script file with:

  1. Instructions to make a file.
  2. Update dependencies.
  3. Clean old files.

- Daily builds. Continuous integration proposes even more.

- *Managing Projects with GNU Make (3rd Edition)* by Robert Mecklenburg, `http://oreilly.com/catalog/make3/book/`.

Containers

- A container is stand-alone, executable package of some software.

- It should include everything needed to run it: code, system tools, system libraries, settings, ...

- Why? Keep all your environment together and allow for multi-platform development and team coding.

- Easier alternative to VMs. But dockers are not "lightweight VMs."

- Most popular: Docker `https://www.docker.com/`.

- Built around dockerfiles and layers.

Linting

Linting

- Lint was a particular program that flagged suspicious and non-portable constructs in `C` source code.

- Later, it became a generic word for any tool that discovers errors in a code (syntax, typos, incorrect uses) before the code is compiled (or run)⇒static code analyzer.

- It also enforces coding standards.

- Good practice: never submit anything to version control (or exit the text editor) unless your linting tool is satisfied.

- Examples:

  1. Good `IDEs` and `GCC` (and other compilers) have excellent linting tools.
  2. `C/C++`: `clang-tidy` and `ccpcheck`.
  3. `Julia`: `Lint.jl`.
  4. `R`: `lintr`.
  5. `Matlab`: `checkcode` in the `editor`.

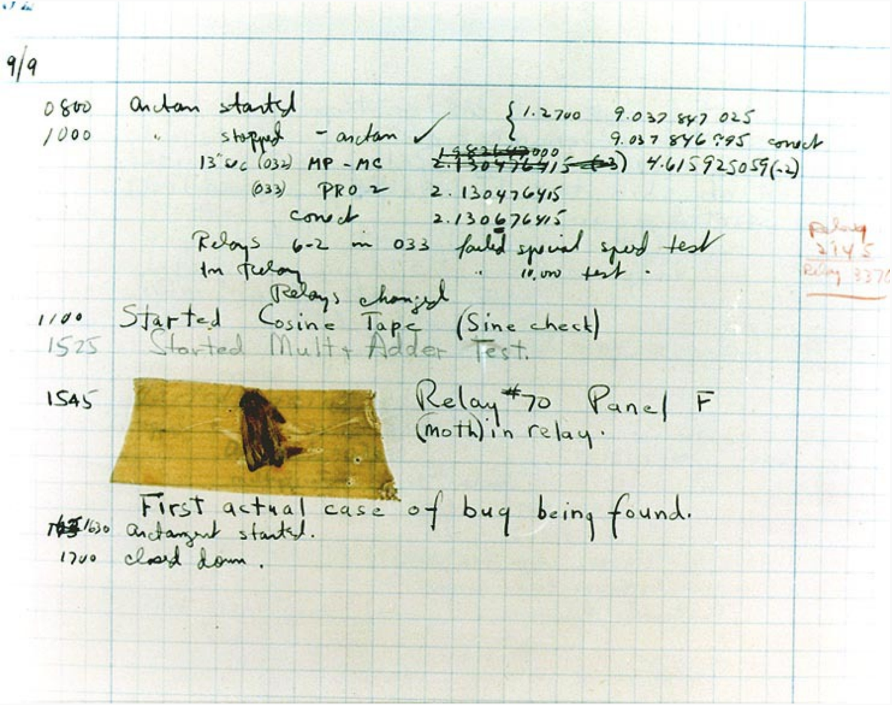
Debugging

Debugging

C. Titus Brown

If you're confident your code works, you're probably wrong. And that should worry you.

- Why bugs? Harvard Mark II, September 9, 1947.
- Find and eliminate mistakes in the code.
- In practice more time is spent debugging than in actual coding.
- Complicated by the interaction with optimization.
- Difference between a bug and a wrong algorithm.



9/9

0800 Anctan started

1000 " stopped - anctan ✓ {1.2700 9.037 847 025
9.037 846 795 correct

13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ ~~(23)~~ 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test
In relay " 10,000 test .

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1630 Antangent started.

1700 closed down.

Relay 2145
Relay 3370

Typical bugs

- Memory overruns.
- Type errors.
- Logic errors.
- Loop errors.
- Conditional errors.
- Conversion errors.
- Allocation/deallocation errors.

How to avoid them

- Techniques of good coding.
- Error handling.
- Strategies of debugging:
  1. Tracing: line by line.
  2. Stepping: breakpoints and stepping over/stepping out commands.
  3. Variable watching.

Debuggers

- Manual inspection of the code. Particularly easy in interpreted languages and short scripts.
- Use `assert`.
- More powerful: debuggers:
    1. Built in your application: `RStudio`, `Matlab` or `IDEs`.
    2. Explicit debugger:
        - 2.1 `GNU Debugger (GDB)`, installed in your Unix machine.
        - 2.2 `Python`: `pdb`.
        - 2.3 `Julia`: `Gallium.jl`.

BEFORE GOING DOWN A STEEP HILL LIKE THIS, ONE SHOULD ALWAYS GIVE HIS SLED A SAFETY CHECK.
RIGHT.
SEAT BELTS?
NONE.
SIGNALS?
NONE.
BRAKES?
NONE.
STEERING?
NONE.
WHEEEEEEE
© 1990 Universal Press Syndicate
1-2

Unit testing

- Idea.
- Tools:

  1. xUnit framework (CppUnit, testthat in `R`, ...).
  2. In `Julia`: Test module.
  3. In `Matlab`: matlab.unittest framework.

- Regression testing.

Profiler

Profiler

- You want to identify the hot spots of performance.
- Often, they are in places you do not suspect and small re-writtings of the code bring large performance improvements.
- Technique:
    1. Sampling.
    2. Instrumentation mode.
- We will come back to code optimization.