Scientific Programming with a Programmer's Tools

Vale Cofer-Shabica
Stratt Group, Brown University

Modified: May 29, 2014 at 12:05

Motivation

Humans have been developing ways of unloading computation onto machines for centuries. The languages our group uses, (C or it's close relative C++) appeared over 40 years ago in 1972. Since that time, computer scientists have spent a lot of time developing tools to make interaction with computers straightforward and to ensure the correctness of programs they write.

The following is a collection of notes I've compiled to review some of these tools. Most of these topics could consume a book by themselves. The discussion is not at all exhaustive, but should serve as an introduction to the key ideas. Suggestions and tips or tricks will be gratefully incorporated.

One more thing before we get started: I'm reading *The Practice of Programming*[1] and it's an excellent reference/style guide/manual for how to be a better programmer—it has already made it easier for me to re-read and debug code that I write. It was written by one of the developers of the C language and one of the developers of the Unix operating system; I cannot recommend it highly enough. I have a copy that you're welcome to borrow when I'm finished.

1 Compiler Warnings and Options

Compilers (gcc, pgcc, g++, *et cetera*) contain the accumulated wisdom [1] of decades of vetted academic research. Most of the time, when I compile my code, I use the following command:

```
% gcc -std=c99 -Werror -pedantic -Wall -Wmissing-prototypes \
      -Wstrict-prototypes -Wconversion -Wshadow \
      -Wpointer-arith -Wcast-qual -Wcast-align \
      -Wwrite-strings -Wnested-externs -g -O2 -c myProgram.c
%
```

[1] see: `http://stackoverflow.com/a/2685541`

Setting all of these warnings is important because they cause the compiler to check more than just syntax (or "grammar" of the program). The compiler can also check for inconsistencies or expressions that result in undefined behavior. It can perform optimizations on your code to make it run faster.

Flag	Description
-std=c99	specify use of the C99 standard (as opposed to ANSI C or C89)
-Werror	treat warnings as errors; don't generate an executable if there are warnings
-pedantic	check for conformance to standard
-Wall	turn on a broad class of warnings to check consistency
-W...	turn on a specific warning
-g	include debugging information in the generated binary
-O2	select optimization level from {0,1,2,3}

Notes and Resources

- See `http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html` for an exhaustive list of warnings available for gcc (and g++).
- See `http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html` for a description of debugging options.
- Once you've verified that your programs works as expected and are ready to run at full speed, you should switch to -O3, which turns on all the optimizations the compiler has available to it; see `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` for a description of optimization levels. See `http://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html` for a list of code generation options.
- The C99 standard admits variable-length arrays, allowing you to write:

```
unsigned int N = getNumber();
double x_vla[N];
// instead of
double * x_malloc = malloc (N * sizeof(double));
```

2 Debugging with GDB

While compilers have a great power to check the consistency of our programs, there are large classes of errors, which they cannot detect. One of the simplest cases is trying to access memory your program is forbidden from accessing. Debuggers are programs for finding these and many other kinds of errors.

Consider the following complete program:

```c
/* segv.c */

int main(void){
  double x[1000];
  double y = 0;
  int i;

  for (i = 0; i < 1000 ; i++){
    y += 0.01;
    x[i]=0;
  }

  //segmentation fault issued here
  x[10000]=0;
  return 0;
}
```

We can compile and execute the program:

```
% gcc -g segv.c
% ./a.out
[1]    5541 segmentation fault  ./a.out
%
```

and see a segfault is thrown when accessing the improper region. We can run the program in `gdb` and inspect the program as it runs:

```
%gdb a.out
GNU gdb 6.6
# start-up elided
(gdb) run
Starting program: /home/vale/src/examples/a.out
Program received signal SIGSEGV, Segmentation fault.
0x00000000004004a0 in main () at segv.c:14
# the last line indicates where the error occurred
14          x[10000]=0;

# we can print the type of any variable with the ptype command
(gdb) ptype x
type = double [1000]
(gdb)
```

and we can get a look at what went wrong. In a non-trivial example, you 'd be able to read out (or modify) the values of all your variables, and watch the execution as it continued. The following

table list some helpful commands:

Command	Description	Example
`help`	list of commands, or help on a topic	`help break`
`apropos`	search for commands relating to a topic	`apropos run`
`run`	runs the program	
`break`	set breakpoints	`break 10 if y > 1.0`
`watch`	break the program when a variable changes	`watch y`
`condition`	set a condition for a breakpoint	`condition 1 y > 1.0`
`set args`	sets the arguments of the program	`set args 1000 2`
`set var`	change the value of a variable	`set var y=5`
`print`	prints the value of an expression	`print x[0]`
`call`	evaluate a function call	`call main()`
`ptype`	prints the type of an expression	`ptype x`
`list`	prints the source code in the vicinity of a line	`list 10`
`bt`	prints backtrace of all stack frames	`bt`
`info`	Many kinds of information; see `help info`	`info scope`
`info locals`	prints a list of local variables	`info locals`
`frame`	selects a frame in the stack (see `bt`)	`frame 0`

GDB Resources

- Quickstart : `http://web.eecs.umich.edu/~sugih/pointers/gdbQS.html`
- Slower Quickstart `http://beej.us/guide/bggdb/`
- Tutorial : `http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html`
- Manual: `http://www.gnu.org/software/gdb/documentation/`

3 Optimization by Profiling

The general rule about optimization is that it should be the last thing you do to your program. Along the way, you should be much more concerned with correctness of your code and of the results it produces. Once you're sure that your program produces mathematically correct results, then you can work on optimizing it for speed.

The first step is changing the `-O2` flag to `-O3` and remove the `-g` flag (see sec 1). This instructs the compiler to exhaustively search for optimizations to your code and to leave out debugging information (which is a waste of space, since your program is correct).

Now if your program still seems slow, the fastest way to improve it's execution time is speeding up the slowest (or longest running) parts. Often that means looking for a better algorithm (bubble

sort [$\mathcal{O}(n^2)$] *vs.* merge sort [$\mathcal{O}(n \log n)$]), but if that's not an option or it's too late for that kind of change, you can profile your code to identify the slowest portions.

`gprof` is the program of choice for identifying slow regions of your code. Using `gprof` is as simple as:

1. Compile your program with the the flag `-pg`
2. Run your program with a *typical* workload; this produces the file `gmon.out`
3. execute `gprof a.out gmon.out > outfile`, where `a.out` is the name of your program
4. view the contents of `outfile`; *e.g.*: `less outfile`

The contents of `outfile` will give you a good idea of where your program spends most of its time; it is divided into 2 sections, the flat profile and a call graph. The flat profile contains functions sorted by time spent in execution (the higher ones re the biggest fish to fry). The call graph gives information about the routes by which functions were called: how much time was spent by each function and its children (functions called from within the first function).

Using the flat profile and the call graph, you can determine where your program spends most of its time and hence where you stand to gain the most for improving speed.

Beware: If you try to do this locally on a mac, you may run into trouble and find no `gmon.out` file is produced. Daniel J. has confirmed this failure under OS X v.10.8.5. If anyone knows why or how to fix it, let me know.

Resources for gprof

- Quickstart : `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`
- Tutorial : `http://www.linuxforu.com/2011/06/code-profiling-in-linux-using-gp`
- Manual: `https://sourceware.org/binutils/docs/gprof/`

4 Missing Topics

- Using version control to track changes to a program
- Makefiles: automating compilation
- Using the queuing system for data parallelism instead of MPI
- finding memory leaks with valgrind
- shell scripting
- ???

References

[1] Kernighan, B. W.; Pike, R. *The Practice of Programming*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1999.