CS 403: Lab 6: Profiling and Tuning

Getting Started

1. Boot into Linux.

2. Get a copy of RAD1D from your CVS repository (cvs co RAD1D) or download a fresh copy of the tar file from the course website.

3. Go to the course website[1], and download UPDATE.tar. This is file has three sample problems of various sizes (C.txt, C2.txt, C3.txt, small.cmnd, medium.cmnd, and large.cmnd) and a new linearalg.c file. In all cases, the initial concentration at each gridpoint was determined by the function:

$$C(x, 0) = \sin^2(\pi x/L)$$

where $L = 10$. The only difference between the initial conditions is the number of grid points ($m = 20$, 100, and 1000, respectively). Diffusion is the only dynamics specified by the .cmnd files, and $k = 0.1$ for each. Other than the initial condition file, the only difference between the .cmnd files is the time step. These values of dt were chosen so that $\sigma = k * dt/dx^2$ is the same for each problem. This ensures that the numerical properties of the solution are the same. The new linearalg.c file identical to the one from last week, except I've added two alternatives to the conjugate gradient routine (pcgm). Place these files in the same directory with your source code.

Profiling with gprof

1. Today, we'll examine the performance of the model problem using the profiling tool gprof. A complete description of gprof can be found at the GNU website[2]. As with gdb, we need to compile our code in a special way to allow gprof to see inside. Specifically, we need to use the -pg flag. Add this flag to your Makefile and recompile (you should delete all object files and the executable rad1d to make sure all files are updated.

[1] www.cs.cornell.edu/Courses/cs403/2002sp

[2] www.gnu.org/manual/gprof-2.9.1/html_chapter/gprof_toc.html

2. Run the program using small.cmnd. In addition to the usual output file, the program will produce a file called "gmon.out." This file contains the information that gprof needs, stored in an incomprehensible binary format. gprof sole reason for existence is to convert this data into a comprehensible format. To do this, type

   gprof rad1d gmon.out > prof_20.txt

   This command tells gprof to interpret the information in gmon.out as if it came from rad1d (which it did) and to save the output in a file called prof_20.txt. Open prof_20.txt in an editor and look at it. The file is divided into two pieces: the "call graph profile" and the "flat profile." Each piece begins by defining the terminology used in the corresponding profile (basically, a table). Following the first set of definitions is the call graph table. This table is divided into sections separated by a line of dashes. Each section corresponds to a subroutine in the program. The first section should contain the information for "main" and it should look something like this:

index	%time	self	descendants	called	name
		0.06	0.19	1/1	__start [2]
[1]	96.2	0.06	0.19	1	_main [1]
		0.01	0.18	1001/1001	_SolveA [3]
		0.00	0.00	9/19	_newarray_double [23]
		0.00	0.00	4/4	_GetData [25]
		0.00	0.00	2/2	_newarray_int [27]
		0.00	0.00	1/1	_ReadComm [31]
		0.00	0.00	1/1	_ReportParams [32]
		0.00	0.00	1/1	_GetInitialCond [29]
		0.00	0.00	1/1	_BuildA [28]
		0.00	0.00	1/1	_PrintArray [30]

   The second line (the only one with information in the first two columns) tells which subroutine this section describes. For clarity, I'll call it the SUB line. Each subroutine is given a unique number called its index (a key is at the very bottom of the file, but the indices are also given

in brackets next to the names on the right). The second column tells the percent of time spent in main and its descendants. The third and fourth columns break down the time (now absolute time in seconds) between the time spent in main (self) and the time spent in direct calls from main (descendants). The fifth column indicates the number of times the routine was called. The line (or lines) above the SUB line describe the subroutine that called this routine (in this case, the system subroutine "start" that actually starts the program). The self column now refers to the amount of time main itself (not its descendants) spent in the service of the caller, while the descendants column refers to the amount of time main's descendants spent in the service of the caller. If the subroutine on the SUB line is only called from one other routine, then the self and descendent times on line 1 will be identical to the SUB line. The called column has two numbers: a/b where a is the number of times start called main, and b is the total number of times that any routine called main. This information is pretty boring for a routine like main that is only called once, but it is more interesting for routines like newarray_double that are called many times by several functions. Finally, the lines below the SUB line describe the subroutines which main called. From this, we can see that most of the time is used in the routine SolveA which is called 1001 times; however, relatively little time is spent in SolveA itself, but is spent in its descendants. Keep looking through the call graph and figure out where rad1d is spending its time.

3. The second piece of prof_20.txt is much easier to interpret, and for a simple program like ours, it contains almost all of the information. The first few lines of the table looks something like

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
53.8	0.14	0.14	1001	0.14	0.18	_pcgm [4]
23.1	0.20	0.06	1	60.00	250.00	_main [1]
15.4	0.24	0.04	2474	0.02	0.02	_sprsmlt [5]
3.8	0.25	0.01	1001	0.01	0.19	_SolveA [3]
3.8	0.26	0.01				_memchr [6]
0.0	0.26	0.00	19	0.00	0.00	_newarray_double [23]
0.0	0.26	0.00	4	0.00	0.00	_CheckCase [24]
0.0	0.26	0.00	4	0.00	0.00	_GetData [25]

Basically, this table breaks down the time among the subroutines and gives some information on the number of times called (but nothing about who did the calling). Does this agree with your conclusion from the call graph?

4. gprof gets its timing data by periodically checking to see which subroutine is being executed. Any sampling procedure, whether its for a computer program or a scientific experiment will only produce an approximation to the truth. One way to find out what our program is really doing is to make several runs and average them. We can also get a better view of our code's performance by using a larger problem. A larger problem means we should spend more time in each subroutine, and this should result in more samples in each routine and a more accurate measure of performance. Run rad1d using large.cmnd. Run gprof and save the output as prof_1000.txt. Examine the file, does it change your conclusions? For our program, the flat profile contains all of the interesting information. If you want to look at it without having to page through the other info, type

   ```
   tail -30 prof_1000.txt
   ```

   (tail is a UNIX command that displays the last few lines of a file, in this case, the last 30).

5. By looking at the code, it shouldn't be surprising that pcgm is limiting the performance of our program. This routine solves our matrix problem $A * C = RHS$ in a somewhat unusual way. Rather than factoring A, pcgm makes a guess at what C should be, multiplies the guess by A

(in a routine called sprsmlt), and based on the difference between the answer and RHS computes a new guess. If A has some key properties (if it is symmetric and positive definite), then pcgm will find C after only a few iterations. In the new linearalg.c, there are two additional implementations of pcgm that I hypothesize will have better performance. The routine pcgmI has the code for sprsmlt pasted inside. My hypothesis is that inlining sprsmlt will avoid the overhead associated with calling it, thereby improving performance. To use pcgmI, just add change the name of the subroutine call inside SolveA and recompile. Run the large problem and then gprof and save your answer in profI_1000.txt. Does inlining improve the performance?

6. Another alternative to pcgm is pcgmL. In pcgmL, I merged some of the loops to try and maximize cache re-use. For example, rather than computing r[n]=r[n]-alpha*w[n] in one loop and then computing z[n]=c[n]*r[n] in a second loop, I merged the two loops so that r[n] can remain in the cache. Is pcgmL better than the original or inlined versions?

7. Last, but not least, let's see what the compiler can do. If you type man gcc, you'll see that there are man options for our compiler, many of them tell the compiler to try various tricks to make the code run faster. Fortunately, most compilers allow you to apply several tricks by setting only one option. For every UNIX compiler I've ever seen, including gcc, there are three optimization levels: -O, -O2, and -O3, with the higher number indicating more aggressive optimizations and hopefully faster code. CAUTION: on some systems, -O3 may result in incorrect results. Check your compiler's manual and your answers! To try the -O2 option: add this option to CFLAGS in Makefile. Delete all of the .o files and rad1d and recompile. As time permits, compare the performance of the three versions of pcgm using this option. Which version is faster with -O2?