15. 780.20 Session 15

a. Overview of Session 15

As we wind down through the last week of the quarter, we'll tie off some loose ends and hit some topics that have been omitted so far. Here's the gameplan:

- 3D plots using gnuplot;
- using the gdb debugger;
- taking a look at optimization;
- using the Intel C++ compiler;
- trying out a profiler.

There are separate handouts for the first two items that step you through making some sample 3D plots and step you through a debugging session.

b. Optimization

In an ideal world, optimization of a computer code would be transparent to the user: the compiler would do it for you. In practice, different compilers for the same language on the same machine can provide very different performances. That is why people still pay big bucks for fortran compilers rather than use g77 (the compiler that is part of the gcc/g++ family). In the Linux C++ world, we have g++ and then commercial compilers, but sometimes the latter are freely available for academic use. In this class is the Intel C++ compiler, which we'll explore in this session. It has a strong benchmarking record, although the latest version of g++ seems to have closed the gap between them. The generic options (-O3 for g++ and -O2 for icpc) will generally do most of the useful optimizations. But you should be aware that there are many additional optimization options that can improve particular codes, especially on a known architecture such as a Pentium 4. You should explore the man pages for the compiler to find out potential options.

In this session, we'll take a look at a simple example of how coding the same operation different ways can make a dramatic difference in the execution time. For example, if we need to calculate the value of $x^n$ where $n$ is an integer, using the function pow($x$, $n$) takes much longer than multiplying $x$ together $n$ times. That is because pow is a library function valid for any real value of $n$ (i.e., any double). The operations needed in general (e.g., logarithms) take much more time than floating point multiplies and if the compiler doesn't substitute for the general algorithm, there will be a big difference in times. We'll see this in practice.

Since operations like exp, sin, and cos are also expensive, if they are evaluated repeatedly with the same argument it is often efficient to use a “look-up” table. This is an array that is filled with the needed values once at the beginning of the program, and then just referenced later. An array lookup is much faster, as long as their is memory available for the array.
c. Segmentation Faults

[This discussion is based in large part on the article at http://www.cprogramming.com/debugging/segfaults.html.]

One of the most useful functions of a debugger is to track down the source of “Segmentation fault” error messages. In general, the segmentation fault means you have tried to access memory that doesn’t belong to your program. This is not allowed, and the segmentation fault or “segfault” is the result. One region of memory is called the “stack,” which is where local variables are stored. Another region is called the “heap,” which is dynamically allocated when your program is running if you use a “new” command in C++.

Here are some sources of segmentation faults (not a complete list!):

1. Accessing an array beyond its bounds. When you declare or allocate an array, it has a certain size. If you try to write beyond it, you will get a segmentation fault error if you go outside the memory assigned to your program. In this case, just use gdb; it will identify that your program crashes at a line involving an array access and you can print the argument of the array. The tricky case is if you stay within the memory assigned, which means you will not get an error but will unpredictably write over other variables. This is not good!

2. Dereferencing NULL, uninitialized, or deleted pointers. This code illustrates the first one:

   ```c
   int *my_ptr = 0;
   *my_ptr = 3;
   ```

   This fails because the pointed is initialized to NULL (which is the same as 0) in the declaration, and then in the next line it is accessed. In gdb, this would be revealed by

   ```
   (gdb) print my_ptr
   $1 = 0x0
   ```

   where memory address 0x0 is, in fact, NULL. To avoid problems with uninitialized pointers, you should simply set them to NULL when you declare them (unless you are using them with a “new” statement to allocated memory).

3. Using all of the stack space (such as by a recursive function); this is sometimes reported as a “stack overflow” rather than a segmentation fault.

The use of a debugger such as gdb to track down segmentation faults will be demonstrated in this session.

d. Profiling

The point of “profiling” is to identify areas of code that use the most overall time. There is no point in optimizing sections that use a small fraction of the total time, particularly if it causes the code to be less clear. For example, if a code spends roughly 90% of its time in one function and 10% in another, making the latter run ten times faster (which is an enormous improvement) will
only make the code run 10% faster (e.g., 100 minutes before and 91 minutes after). Focus on the first function!

There is a standard GNU tool for profiling, called `gprof`. We’ll give it a try in this session, although you should be warned that it is not always helpful.

e. Using the Intel Compiler

So far we’ve used the GNU C++ compiler, called `g++`, exclusively. The C++ compiler `icc`, supplied by Intel and optimized for Intel chips, is also available. (You can get it for your own computer from the Intel website.) There are several reasons it is useful to have an alternative compiler available:

- It may produce a faster executable through better optimization.
- There may be a bug in one of the compilers, which leads to incorrect results from your program.
- One compiler may give more useful warning messages than the other for debugging or verifying your code.

In short, any important code should be compiled with more than one compiler. In this session, we’ll learn how to let up the Linux environment variables so we can use the Intel C++ compiler, called `icpc`, in place of `g++`.