Lab 0 -- Getting Started
Due Tuesday, January 21st

Objective
Learn about and practice using tools and techniques for performance evaluation of computer programs.

Readings
lab_orientation_2014 and **B&O Chapter 5.0 and 5.2**.

Administration
Deliverables: Hand in the write-ups (answers to questions) and code (as specified below) using the BlackBoard Assignment function. For this assignment, turn in answers to questions in a text file together with the programs that you have written/modified. Combine into a single archive using the format described in the Lab Orientation.
Code for validation by TFs: Code should be compilable and executable on the HP workstations in the HPC lab (PHO 207) or in the 3rd floor Linux lab (PHO 307). Be sure to include compilation instructions (or Make file) and tell which machine(s) you tested your code on. It's OK if you develop code on your own platforms, but it's also your responsibility to make sure that everything works on the common platforms -- this is essential to facilitate grading.

Prerequisites
Basic programming in C and use of Linux programming environments.

Setting up

A. Room Access. Make sure that you have room access to PHO 207 and PHO 307.

B. Machine access. Make sure that you can log in to the machines in both labs (should be immediate with Active Directory, but check!) and that your "ad" stuff comes up.

C. gcc, etc. After logging in, make sure that you have access to gcc and your favorite Linux debugger and/or programming environment. Write and run "Hello World!" (at least) to confirm.

D. icc. Set up your account so that you have access to the icc compiler. Do this by executing the following script (by typing the following at the command line, i.e., type what is after the "$"):

```
$ source /afs/bu.edu/x86_bulnx50/IT/Intel/bin/intel64/iccvars_intel64.sh
```

To confirm, write and run "Hello World!" (at least).

Note: sometimes you get a weird message. Solve with

```
$ export LANG=C
$ set LC_ALL C
```

Assignment

Part 1. Find machine characteristics. Use the following command to find the basic machine characteristics:

```
$ less /proc/cpuinfo
```

- What CPU are you using?
- What is the operating frequency of the cores?

Search the web to find out more machine details:

- how many levels of cache are there and what are the characteristics of each one?
- what is the microarchitecture of the processor core?
- how many cores are there?

Turn in answers to the questions.

Part 2. Timers. Accurate timing is essential to performance evaluation. It is also a (perhaps) surprisingly subtle topic, and getting more so all the time. There are several ways to time program execution which you should explore as follows. Throughout this part, use the -O0 optimization setting. Also note the -lrt switch needed to compile clock_gettime().

- Read notes_timers.txt and test_timers.c. The source file test_timers.c contains three different timing mechanisms. Compile and execute. Observe each method's effectiveness and accuracy.
    - How can you tell whether the timer is accurate? In its deepest form, this is a fundamental question in physics, no need to go that far! First answer this question generally, how do you determine accuracy? Then consider the resolution to which you can easily determine accuracy. What is it?
- The timer that uses RDTSC has some basic problems. Do an internet search to find out what they are.
    - What problems do RDTSC-based methods have? Is it still useful? Note: you can solve these problems yourself, but no need to do so for this lab!
- Each timer requires some calibration which may or may not have been done correctly in the source file.
    - For each timer, what (potentially) needs to be done?
    - As necessary, adjust the constants to give correct timing.
    - Describe how you did this and the modification (if any) you made to test_timers.c.
- Perhaps the best way to do timing nowadays (on our systems) is by using clock_gettime(). The source file test_clock_gettime.c has a partially written testbench for this function.
    - Write dummy code that takes time close to 1 second to execute. How close can you get to it? What is the resolution? What is the error?
    - Write a call to the function “diff” -- this should be simple, but will also test your basic C knowledge.

Turn in the new test_clock_gettime.c and the answers to the questions.

Part 3. Plotting/Graphing Data. Presenting data accurately and concisely is essential in this course. Most of the time simple methods will do, once you figure out what you need to display.

- Read the section on plotting in **lab_orientation_2013**.
- Compile and run the code plotting.c and plot as decribed.

You do not need to turn in anything from this part.

Part 4. Performance evaluation using Cycles-per-Element. As described in B&O Chapter 5, evaluating performance requires establishing a deep understanding of how your code interacts with the CPU and memory hierarchy. Unfortunately, even with accurate timers, program timing measurements can be incredibly noisy. Some of this noise you will understand in the next few weeks, some is understood only by Intel Architects! B&O suggest an excellent way to run performance experiments: find the number of cycles it takes to process each marginal (additional) element in a program. Use the -O1 optimization setting for this part.

- Read **B&O Chapter 5.0** and **5.2**.

- Code and plot the example in Chapter 5.2 (psum1 and psum2) to replicate the graph in Figure 5.2. Start with test_psum.c which has the functions you need already entered. Add timing using clock_gettime() from part B and plot your results as shown in part C.
    - How many cycles per element does it take on your computer for psum1 and psum2? Is it the same as in Figure 5.2? If not, why?

Turn in the modified test_psum.c and your plot.

Part 5. Interacting with the compiler. One of the difficulties in developing an understanding of how programs interact with hardware is making sure that your program actually does what you intend! In particular, the compiler will, by default, try to make your code as efficient as possible. Usually that's a good thing, but not here. If your program is not actually doing anything (e.g., it has only reads with no writes as in test_timers.c), then the compiler may get rid of all of your code leaving nothing to measure. This is why you compiled test_timers.c with the -O0 optimization setting: this turns off all optimization and leaves your code as is, however silly your code may be. But the way to really make sure what you expect has happened is to read the assembly language code, which is what you will now do. Some basics:

- -O0 compiler directive turns off all optimizations. -O1 turns on basic optimization. -O2 and -O3 turn on more complex optimizations. More about all of these later in the semester.
- -S compiler directive generates an assembly language file. While compiler generated assembly language can be opaque, looking at it is essential to confirming what code is being executed.

To do: Observe what happens when -O1 (as opposed to -O0) is applied to the test_timers.c kernel code, i.e., the code that we use as a delay loop.

1. Compile and run test_O_level.c as follows:
   $ gcc -O0 test_O_level test_O_level.c
   $ time ./test_O_level
   What gets printed?
2. Now compile and run test_O_level.c as follows:
   $ gcc -O1 test_O_level test_O_level.c
   $ time ./test_O_level
   What gets printed? You should see that the execution time has gone down nearly to 0s.
3. This time compile test_O_level.c as follows to generate the assembly language code:
   $ gcc -O0 -S test_O_level.c
   Look at test_O_level.s and find the loop. Don't panic! Look for the two "call puts" lines which are the printf statements. Also, the variable "steps" is -8(%rbp) and "i" is -32(%rbp). And the "$" means "treat this number as an immediate" so $1 means 1 and $3 means 3. Rename the file to save it.
4. Now generate the assembly language code of test_O_level.c with basic optimization:
   $ gcc -O1 -S test_O_level.c
   Look at test_O_level.s and find the loop. What do you see? How does it compare with the -O0 version?
5. You should have found something strange. Perhaps using "steps" will solve the problem. Modify test_O_level.c so that "steps" gets printed out.
   Repeat Step 2. How much time does the code take to execute now?
   Repeat Step 4. What is happening? That is, how has the code been optimized?

Turn in answers to the questions.

Part 6: QC

1. Are you missing skills needed to carry out this assignment?
2. How long did this take (hours)?
3. Did any part take and "unreasonable" amount of time for what it is trying to accomplish?
4. Are there problems with the lab?

Turn in answers to questions.