Homework #3-Performance Evaluation of C programs

Objectives: To develop a reusable module with timing macros for instrumentation of C programs and to experiment with some of the performance optimization techniques discussed in class.

Setup: Upload the file **hw3_files.tar** (available in MyCourses) to a suitable directory in your cluster account. Then unarchive it (**tar -xvf hw3_files.tar**).

This will create the directory **hw3** with files: **hw3cpp.cpp, hw3c.c, sleep.c, data.txt and Timers.h**.

Note that **Timers.h** is empty, you will write the implementation of timing macros here.

The files **hw3cpp.cpp** and **hw3.c** implement a least-squares linear fitting program, in C++ and C, respectively. These program take their *input from a file* whose name is passed as a *command-line argument* and are designed to accommodate a *variable number of data points*. The file **data.txt** contains the data points to be used as input to the programs *for testing*. Both program read all the data points from the input data file and store them in memory for "batch processing" (The number of data points is not known in advance). This is why the C program uses dynamic arrays to store the data. **Sleep.c** implements a simple 60 second delay program which can be used to test your timing macros.

Preliminary Step: Compile the C and C++ programs and examine the output and the code to understand what they do and how they do it (note the use of dynamic arrays in the C program). Use the following directives for compilation:

a) C++ compilation command: **g++ -Wall -pedantic -O0 hw3cpp.cpp -o hw3cpp**
b) C compilation command: **gcc -Wall -std=c99 -pedantic -O0 hw3c.c -o hw3c**

Note: **g++** and **gcc** are DIFFERENT compilers.

Implementation of Instrumentation Macros:

Implement the **Timers.h** module as described in Chapter 2, Listing 2.1 of Greg Semeraro's Book – (posted in MyCourses **[References]**). You ***must use these macros*** to instrument the programs in this homework and asses their performance. Note that you will not receive any credit if you "hard code" timing statements without using **Timers.h**. In Listing 2.2 of the reference you will find an example that describes how to use these macros.

In addition to the macros described in the given reference you will implement three additional macros:

1. **PRINT_RTIMER(A,R)** This macro takes a timer name **A** and a number **R** representing the number of repetitions (of the code being timed) to print the correct timing info, that is, the time reported by **PRINT_TIMER(A)** divided by **R**
2. The macros **BEGIN_REPEAT_TIMING(R, V)**, **END_REPEAT_TIMING** should be designed to wrap the code begin timed in a for–loop that executes the code **R** times. You will need another macro, **DECLARE_REPEAT_VAR(V)** to allocate a looping variable **V**. Remember that macros are really just text so they can include fragments of "C" code.

These macros should behave in the same way that all the other **Timer.h** macros do, that is, they will be included in the code when **EN_TIME** is set and expand to "nothing" otherwise. Be sure to verify the "nothing" case.

Instrumentation for Timing: For timing of the C and C++ program take into consideration the following *requirements*:

1. Your timing should *measure and report execution time* of ***data input*** and ***calculations*** separately. Use two different timers, one for data input and another for calculations. Your data should include the total elapsed time and the per loop time for data input and calculations.
2. Your C program timing should be "fair" compared to the C++ program, that is, the C timing should include, as much as possible, the same processing as the C++ timing.

   Be careful with the data input portion, you may need to move some code around to get a good measurement. If you read a file that has already been fully read, there is no more data and you will get false, short time values. Manage the fopen()/fclose() sequences to ensure good data.

   Run your input data measurement code multiple times without modification. You will find that data input performance is HIGHLY variable. This variability significantly impacts our ability to "tune" our software because we will have issues identifying the value our changes versus normal system performance noise.

Code Instrumentation: Following these guidelines.

a) Instrument the C++ program using your **Timers.h** macros. Name the instrumented program **hw3t.cpp**
   For timing you will compile the C++ program as follows:

   ```
   g++ -Wall -pedantic -DEN_TIME hw3t.cpp -o hw3t_cpp
   ```

b) Instrument the C program using your **Timers.h** macros. To be fair, make sure that you time the same portions of the code timed in the C++ program. Name the instrumented program **hw3t.c** For timing you will compile the C program as follows:

   ```
   gcc -Wall -std=c99 -pedantic -O0  -DEN_TIME hw3t.c -o hw3t_c
   ```

Analysis and Optimization:

- First analyze the C++ and C programs **hw3cpp.cpp** and **hw3c.c** for ***speed and size***. For speed you will use Timing macros. For size use **ls -l** and **size -A** (refer to the "manual pages" for more details about these commands.) Write your analysis in the text file **analysis1.txt**. Be sure to include the output from your ls –l and size –A in your TAR file as **ls1.txt** and **size1.txt**, respectively.

- Modify the C program in any way you want to make it ***as small and fast as possible***. Call your optimized program **hw3opt.c**. Draw upon your programming skills and knowledge to produce the "best" version of this C program. Note that this is an iterative process, you will likely encounter a trade-off between speed and size. Your analysis should include all the steps and data you took to arrive at your final conclusion. Make sure your optimized C program maintains the following aspects of the original C program:
  a. The same output.
  b. The same command-line arguments.
  c. All data points are read and stored in memory for batch processing.

d. The number of data points is not known in advance.

- • Analyze the optimized C program for ***speed and size***. Write your analysis and conclusions in the text file, **analysis2.txt**. Be sure to include the output from your ls –l and size –A in your TAR file as **ls2.txt** and **size2.txt**, respectively.

Makefile:

You must include a makefile with: all, test, help and clean targets. "all" should build all your timing enabled code (hw3t_c, hw3t_cpp andhw3opt). "test" should run each of your performance test targets, with the data.txt and redirect all the output to a single file called "out.txt". A line indicating what command is being executed must be included in "out.txt" prior to the actual execution of each command.

Once you are done create a **lastName_hw3.tar** (lastName is your last name) file and submit it.

Grading Criteria

1. (50 points)
   a. (12 pts) **Timers.h** module properly implemented.
   b. (14 pts) Correct implementation of **PRINT_RTIMER(A,R)**, **BEGIN_REPEAT_TIMING(R)** and **END_REPEAT_TIMING**
   c. (12 pts) C++ program properly instrumented using **Timers.h**
   d. (12 pts) C program properly instrumented using **Timers.h**

2. (50 points) Analysis
   a. (5 points) The make file contains all the required features and operates properly.
   b. (20points) Comparison of original C++program (**hw3.cpp**)with original C program(**hw3.c**)
      a.1 Explanation of any differences in speed.
      a.2 Explanation of any differences in size.
         i. Includes **ls1.txt**, the output from **ls -l**
         ii. Includes **size1.txt**, the output from **size -A**
         iii. Analyze and explanation differences (**analysis1.txt**), with standard formatting.
      a.3 (4 points) Analysis is clear, concise and direct to the point.
   c. (25 points) Comparison of optimized and original C programs
      b.1 Explanation of differences in speed
      b.2 Explanation of differences in size
         i. Includes **ls2.txt**, the output from **ls -l**
         ii. Includes **size2.txt**, the output from **size –A**
         iii. Analyze and explanation differences (**analysis2.txt**), with standard formatting.
      b.3 (4 points) Analysis is clear, concise and direct to the point.

Hints!
You should verify your timing macros with the sleep.c code. If you later get a "zero" for the execution time, you know the problem must be in the number of loops in your repeat values, not a problem with the macros.

Note: You should expect IO operations to be at the rate of milliseconds and computations at the rate of 100's of nanoseconds.