Computing Lab #11: The last lab (for bonus points)

Due: Whenever, but before final grades are submitted (12/19).

Summary: Confirm facts about OpenMP with preprocessor and assembly code. Profile and optimize an existing code. Explore memory layout.

Programming language: Write, compile, and run a Fortran, C or C++ program. Your choice! This lab assumes the C programming language.

Points: Each task carries 2 bonus point towards homework 3. Completing this lab will give you **6 bonus points**. To collect your bonus points, you must show me your results. I won't be checking bitbucket for optional labs (if you do upload to bitcucket, please let me know by email so I can inspect your work).

I. TASK #1: WHAT GCC DOES *BEFORE* COMPILING YOUR CODE (+2 POINTS)

Task: Export and inspect some pre-preprocessor and assembly code.

- (**Part a**) Suppose you want to know the *exact* omp.h file being included in your code. One way would be to check the pre-preprocessor file.

    1. Find some openMP code you have written. Use the preprocessor and inspect its output to find the full path to the header file omp.h. Open the header file and confirm that the function omp_get_thread_num() (and the other ones) are defined but the pragmas are not.

- (**Part b**) Suppose you are unsure whether or not your program is using OpenMP. One way to check would be to look at the assembly code.

    1. Produce two versions of the assembly code with and without the flag "-fopenmp". Use the unix program "diff" to inspect the differences between these two files. Check that when "-fopenmp" is not used the assembly code has no OpenMP in it.

II. TASK #2: LET THE COMPILER OPTIMIZER YOUR CODE (+2 BONUS POINTS)

Task: Try out different compiler optimization flags. Monitor the results.

- (**Part a**) Find some non-trivial code you or I have written and optimize it with compiler flags. The best code to use will have user-defined functions and perform lots of floating-point operations (e.g. homework 2, problem 2) .

    1. Use a profiler (probably gprof) to generate timing data for the un-optimized code. Make sure your program runs long enough (at least a few seconds) to get good data.
    2. Try compiling the code with many different optimization flags (all three optimization levels, fast-math, and vectorization), each time running the code and monitoring its time with the unix program "time".
    3. Using the optimization flags which result in the fastest program, use the profiler again to see which parts of the code were most affected by optimization.

III. TASK #3: MEMORY LAYOUT (+2 BONUS POINTS)

Task: How is your matrix stored in memory? Whats the fastest way to access the entries of a matrix? For this problem do *not* use a compiled language like Fortran or C.

Suppose you want to know how Matlab, Python, R, or any other high-level language stores a matrix in memory. Recall that a matrix

```
0,0 | 0,1 | 0,2 | 0,3
----+-----+-----+----
1,0 | 1,1 | 1,2 | 1,3
----+-----+-----+----
2,0 | 2,1 | 2,2 | 2,3
```

is stored in a particular order, with the two options being

```
OPTION 1:   0,0 | 0,1 | 0,2 | 0,3 | 1,0 | 1,1 | 1,2 | 1,3 | 2,0 | 2,1 | 2,2 | 2,3

OPTION 2:   0,0 | 1,0 | 2,0 | 0,1 | 1,1 | 2,1 | 0,2 | 1,2 | 2,2 | 0,3 | 1,3 | 2,3
```

One way to empirically check would be to write code using two different access patterns and measure the time.

- (**part a**) Lets perform the above numerical experiment. Since your computer will be using cache, the access pattern which utilizes *spatial locality* (multiple accesses of data next to each other) will be faster.

  1. Consider this C code written below. The code fills up a 4000-by-4000 matrix with entries by accessing the matrix elements using two different orderings. Impliment this code I wrote below in Python, Matlab, R or another scripting language. Make sure to explicitly write the for-loops just as I've done below.

```c
int main() {

  double tic = omp_get_wtime();
  static int x[4000][4000];
  for (int i = 0; i < 4000; i++) {
    for (int j = 0; j < 4000; j++) {
      x[j][i] = i + j;
    }
  }
  double toc = omp_get_wtime();
  double time = toc - tic;
  printf("%f\n",time);

  double tic1 = omp_get_wtime();
  for (int j = 0; j < 4000; j++) {
    for (int i = 0; i < 4000; i++) {
      x[j][i] = i + j;
    }
  }
  double toc1 = omp_get_wtime();
  double time1 = toc1 - tic1;
  printf("%f\n",time1);

}
```

  2. Run your code to make sure its working
  3. Run the code and monitor the timing. Do you see a difference? How is the matrix being stored in memory? Option 1 or Option 2?
  4. Use the internet to confirm or refute your findings.

IV. UPLOAD YOUR WORK TO GIT

Congratulations! You've completed the lab. To collect your bonus points either show me your work in class or upload to bitbucket and send me an email to look it over.