02635 Fall 2016 — Module 6

Homework

- Read chapter 11, sections 12.1-12.7 (pp. 156-170), and chapter 13 in "Writing Scientific Software"
- Catch up on unfinished exercises from previous weeks

Exercises

I. In this first exercise today, you should write a program that times the function `datasize1()` from the lecture. The goal is to produce a performance graph, as the one shown in the lecture, that illustrates the performance dependency on the different cache levels in your computer.

1. Write a `main()` function, that calls `datasize1()` for different numbers of elements, and time the execution. To get realiable results, you might need to time several function calls, and then take the average, e.g. by adding a loop around the call. Use `clock()` to measure the CPU time.

2. Instead of looking at the timings, you can look at the performance, measured in Flop/s (floating point operations/second). This can be easily calculated from the timings - how? **Hint:** the loop in `datasize1()` has one floating point operation (multiplication) per loop element!

3. Plot your performance numbers over the memory size of the array used, for different array sizes, ranging from a few kilobytes (kB) to some Megabytes (MB). Can you observe the 'stepping' behaviour, and do the different steps correspond to the cache size levels in your computer? Use the commands shown in the lectures to get the specifications of your computer. **Hint:** you might need to compile with full optimization, to see this!

Why is looking at the performance a better measure than using the raw timings in this case?

Hint for plotting the performance data: use a log-scale with base 2 for the x-axis in the plot!

II. In this exercise, we will implement our own **dgemv** routine, i.e., a function that computes the matrix-vector product

$$y \leftarrow \alpha A x + \beta y$$

where A is a matrix of order $m \times n$, x is a vector of length n, y is a vector of length m, and α and β are scalars. We will compare different implementations by measuring the CPU time with the `clock()` funtion for different values of m and n. You can re-use the framework of the first exercise to measure the timings. Next week we will parallelize the code using OpenMP.

1. Write a function `my_dgemv_v1` with the following prototype:

   ```
   void my_dgemv_v1(
     int m,          /* number of rows                    */
     int n,          /* number of columns                 */
     double alpha,   /* scalar                            */
     double ** A,    /* two-dim. array A of size m-by-n   */
     double * x,     /* one-dim. array x of length n      */
     double beta,    /* scalar                            */
     double * y      /* one-dim. array x of length m      */
   );
   ```

   The matrix A should be a (C-style) **dynamically allocated** two-dimensional array, and the function should compute

   $$y_i = \alpha \sum_{j=1}^{n} A_{ij} x_j + \beta y_i, \quad i = 1, \ldots, m.$$

   Use two nested loops: the outer loop should loop over i (corresponding to the m elements of y) and the inner loop should loop over j (corresponding to the sum over j).

2. Write a program that measures the CPU time required by `my_dgemv_v1()` for $m = n$ and $n \in \{100, 200, \ldots, 1000\}$.

3. Write a function `my_dgemv_v2()` that has the same prototype as `my_dgemv_v1()`, but instead of looping over i in the outer loop, loop over j in the outer loop and i in the inner loop. In other words, `my_dgemv_v2()` should compute y by accessing A column-by-column, i.e.,

   $$y = \sum_{j=1}^{n} \alpha a_j x_j + \beta y$$

   where a_j denotes the jth column of A.

4. Measures the CPU time required by `my_dgemv_v2()` for $m = n$ and $n \in \{100, 200, \ldots, 500\}$. Compare with the CPU time required by `my_dgemv_v1()`. Which of the two methods is faster and why?

5. Compile your code with compiler optimization flags and repeat the two timing experiments (i.e., measure the CPU time required by `my_dgemv_v1()` and `my_dgemv_v2()`). Try with each of the following flags:

  - `-O0` — no optimization
  - `-O1` — enables basic optimizations; no speed-space trade-offs
  - `-O2` — enables further optimization; no speed-space trade-offs
  - `-O3` — most expensive optimizations; may increase size of executable

  How much does compiler optimization affect the results? If you are using GCC, you may also try to enable loop-unrolling with the `-funroll-loops` compiler flag.

  Remark: The optimization flags do not work the same for all C compilers. For example, `-funroll-loop` is recognized by GCC and works independently of the `-O` compiler options, but Clang enables loop-unrolling with the `-O1` option. For more information about GCC and optimizations, see Options That Control Optimization.

Optional

Instead of looking at the timings, you can also measure the performance in Flop/s of your `my_dgemv...()` functions. How can this be achieved? **Hint:** count the number of floating point operations in the loop body. The total number of operations is then dependent on m and n.