02635 Fall 2018 — Module 9

Homework

- Catch up on unfinished exercises from previous weeks, especially last week. We will need the matrix times vector code this week again.
- Read the two Wikipedia articles about CPU time and Wall-clock time, and make yourself clear what the difference between the two is.

Exercises

I. To get started, implement the OpenMP "Hello world v2" example from the lecture.

1. Compile the code as you are used to. Does it work, or does the compiler give you any errors or warnings?
2. Compile the code again, now enabling OpenMP support in your compiler. Check your compiler documentation for the necessary options.
3. Run the example for different numbers of threads, using the environment variable `OMP_NUM_THREADS`. Make yourself familiar with, how to use this variable in your environment.

More information on how to enable OpenMP in your compiler has been published on Piazza.

II. In today's second exercise, we will implement our own, parallel version of the **dgemv** routines from last week. Remember: **dgemv** is 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.

1. It is very easy to introduce errors when parallelizing a program, e.g. if one is not aware of data dependencies, and thus introduces data corruption in the code. Therefore it is a good idea, to compare the obtained results against a reference solution, e.g the results obtained in a serial version of the program. To make your life easier in this week's exercise, you can find a C source (and the corresponding header file), with some helper functions, in a ZIP file on CampusNet. Here are the prototypes:

```
void init_data(int m,         /* number of rows               */
               int n,       /* number of columns            */
               double *a,   /* result vector of length m    */
               double **B,  /* two-dim array of size m-by-n */
               double *c,   /* input vector of length n     */
               double *ref  /* reference vector of length m */
              );

int check_results(char *comment, /* comment string              */
                  int m,         /* number of rows              */
                  int n,         /* number of columns           */
                  double *a,     /* vector of length m          */
                  double *ref    /* reference vector of length m */
                 );
```

Notes:

- `init_data()` assumes that $\alpha = 1.0$ and $\beta = 0.0$! You should call the function before you call any of your own functions, and you need to allocate a vector for the reference solution as well (same length as vector y)!
- After the call to your function(s), you can compare your result to the reference, using `check_results()`. The first argument is a string, that allows you to label your check, i.e. "test v1" or "test v2".
- The C source contains an updated version of `malloc_2d()`, and a corresponding `free_2d()`, as well!

2. Copy your function `my_dgemv_v1` from last week to a new function `omp_dgemv_v1` with the same prototype:

```
void omp_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) two-dimensional array, and the function should compute

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

There are 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).

Make the code parallel with OpenMP, by turning one of the for-loops into a parallel loop, using

`#pragma omp parallel for`

3. Take your program to measure the CPU time from last week, and create a version that measures wallclock time, instead. Repeat the experiments from last week for the parallel version, i.e. take timings of the `omp_dgemv_v1()` for $m = n$ and $n \in 100, 300, \ldots, 3000$, and different numbers of threads, from 1 to the max. no of cores available on your computer.

   What can say about the speed-up of the code, e.g. for different values of n?

   Notes:

   - We will compare different implementations by measuring the wallclock time with the OpenMP library function `omp_get_wtime()`. The function can be used like `clock()`, but it returns the time in seconds (as a double precision number).
   - Control the number of threads by the environment variable `OMP_NUM_THREADS`.
   - To get the best parallel performance, use optimization as well, i.e. the compiler option `-O3`.

4. Write a function `omp_dgemv_v2()` that has the same prototype as `omp_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, `omp_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.

   Make the code parallel with OpenMP (see above). What is different here, compared to the parallelization of `omp_dgemv_v1()`?

5. Measures the wallclock time required by `omp_dgemv_v2()` for $m = n$ and $n \in 100, 300, \ldots, 2500$, and different numbers of threads. Compare with the obtained for `omp_dgemv_v1()`. Which of the two methods has a better speed-up, and why?