Otto-von-Guericke-Universität Magdeburg
Max-Planck-Institut for Dynamics of Complex Technical Systeme
Computational Methods in Systems and Control Theory

Dr. Jens Saak, Dipl.-Math. Martin Köhler
Website: http://www.mpi-magdeburg.mpg.de/2896761/sc2

summer term 2015

Scientific Computing 2
Homework 2

Handout: 04/28/2015 **Return:** 05/05/2015

Attention. If you use the virtual machine from the winter term it can happen that runtime measurements show a "wrong picture" of what is going on. If you have the possibility please use a native Linux setup for testing and evaluating your codes.

Exercise 1: (14 Points)

We conside the matrix-vector product $x = Ay$ ($x \in \mathbb{R}^n$, $y \in \mathbb{R}^n$, and $A \in \mathbb{R}^{n\times n}$). The matrix A is assumed to be dense and not to have any interesting structure that can be exploited in implementations.

a.) Write a C-function

```
void mvp(struct my_matrix_st *A, double *y, double *x),
```

which computes $x = Ay$. The matrix is stored column-wise (as in Fortran). For easier handling of the matrix it is packaged in the structure known from the winter term:

```
struct my_matrix_st {
        int cols;
        int rows;
        int LD;
        double * values;
        char structure;
};
```

b.) Write a C-function

```
void mvpt(struct my_matrix_st *A, double *y, double *x),
```

which computes $x = A^T y$. The matrix A is stored column-wise again.

c.) Parallelize both function form a.) and b.) using PThreads. The parallel implementations should have an additional parameter `T`, which specifies the number of threads to be created. The parallelization should distribute the work in a way such that every thread computes approximately $\frac{n}{T}$ elements of the result vector x. The functions should have the following signatures

```
void mvp_thread(int T, struct my_matrix_st *A, double *y, double *x),
```

and

```
void mvpt_thread(int T, struct my_matrix_st *A, double *y, double *x).
```

d.) Create a second variant of the parallelized matrix-vector product from Exercise a.). But in contrast to the implementation from Exercise c.) create a single thread for each element in the result vector x, i.e., you have to create n threads. The function should have the following signature:

```
void mvp_allthreaded(struct my_matrix_st *A, double *y, double *x).
```

Measure the runtime and the cpu-time of the different implementations. Use random matrices from $n = 100$ to $n = 10\,000$ and compare the achieved flop rate.
Is there a difference between the normal and the transposed matrix-vector product?
Which is the recommended of the two parallelization strategies and why?
A skeleton code is available at: http://www.mpi-magdeburg.mpg.de/mpcsc/lehre/2015_SS_SCII/tutorial/mvp_skeleton.tar.gz

Exercise 2: **(8 Points)**

We consider the Simpson rule to approximate the value of an integral:

$$s = \int_a^b f(x)\,\mathrm{d}x.$$

Thereby, we split the integration interval $[a, b]$ into n equally sized intervals

$$[x_i, x_{i+1}], \qquad x_i = a + ih, \quad i = 0, \ldots, n,$$

with $h = \frac{(b-a)}{n}$. The middle-point of each interval is given by

$$x_{i+\frac{1}{2}} = \frac{x_i + x_{i+1}}{2}.$$

Now, we approximate the integral by

$$s = \frac{h}{6} \sum_{i=0}^{n-1} \left(f(x_i) + 4f(x_{i+\frac{1}{2}}) + f(x_{i+1}) \right).$$

Implement the Simson rule using T Pthreads. The work of the intervals $[x_i, x_{i+1}]$ should be distributed equally across all participating threads. Find a clever way to return the value of the partial sums to the main program.
Use the function

$$f(x) = e^{x^2} - 1.$$

on the interval $[0, 2]$ for testing purpose. Fill up the following runtime table:

Teilintervalle	$T = 1$	$T = 2$	$T = 4$
$n = 2^9$			
$n = 2^{13}$			
$n = 2^{16}$			
$n = 2^{20}$			

Exercise 3: **(8 Points)**

One of the biggest problems in parallel environments are race-conditions that occur because of missing synchronization between the parallel workers. Pthreads provide two concepts to avoid such problems and guarantee a proper interaction between the different threads: the `pthread_mutex` and the `pthread_cond` mechanisms.
Realize the following workflow with the help of Pthreads:

- The main program creates 3 threads.
- Afterwards, the main program reads an integer value from the standard input and passes it with help of a global variable to the first thread.
- Now, the first thread takes this number. If the number is even it is directly passed to the third thread via a global variable. If the number is odd it is passed to the the second thread via a global variable.
- The second thread takes the value it got from the global variable, multiplies it by 4 and passes it to the third thread.
- The third thread takes the integer it got from the first or the second thread and computes the square root of it as a double precision number. The result is passed back to the main program.
- The main program waits until the third thread sends the notification that the results is ready, picks it up, and prints it to the screen. Afterwards it waits for the successful termination of all threads.

Each thread should print some small status messages to the screen, whenever it waits for data, or passes its data to the next thread. Because the standard output on the screen is buffered, in order to avoid this please use `fflush(stdout);` after each print operation to flush the output buffer and to guarantee that everything is written to the screen.

Overall Points: 30