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/csc/teaching/19ss/sc2

Summer Term 2019

Scientific Computing 2
Homework 2

Handout: 05/15/2019 **Return:** 05/21/2019 – 12 a.m. via e-mail

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: (8 Points)

We consider the matrix-vector product $x = Ay$ ($x \in \mathbb{R}^n$, $y \in \mathbb{R}^n$, and $A \in \mathbb{R}^{n\times n}$), again. This time the matrix A is assumed to be dense and symmetric. Due to the symmetry, only the entries in the lower left triangle should be used to compute the matrix-vector product. The entries in the upper right part of the matrix A should not be accesses neither read-only nor to store intermediate results.

a.) Implement a C-function

```
void mvp_openmp1(struct my_matrix_st *A, double *y, double *x)
```

which computes the symmetric matrix-vector product in parallel using OpenMP. If race conditions occur, protect the affected operations by using proper synchronization statements from OpenMP.

b.) Create a second C-function

```
void mvp_openmp2(struct my_matrix_st *A, double *y, double *x)
```

which computes the symmetric matrix-vector product without using any of the OpenMP synchronization statements. This implementation might require some extra memory.

c.) Develop a benchmark program which determines the break even point between single- and multi-thread execution of matrix-vector product functions. Use the OpenMP `if`-clause to enable/disable the parallel execution of the algorithm at this point.

Compare the runtime of both implementations and explain the differences. Use matrices of dimension $n = 100$, $n = 1\,000$, $n = 5\,000$, and $n = 10\,000$ for this purpose.
The skeleton code from: http://www2.mpi-magdeburg.mpg.de/mpcsc/lehre/2019_SS_SCII/tutorial/mvp_skeleton.tar.gz can be used again.

Exercise 2: (6 Points)

Monte-Carlo simulations are an important tool in scientific computing with a special interest on stochastic process. These simulations are parallel by design and can be implemented easily using OpenMP.
All basic reoccuring problems of such parallel implementations are covered by the Monte-Carlo computation of π, which works as follows:

- We consider the upper right quarter of a unit circle around $(0, 0)$, i.e. the radius $r = 1$, and a unit square surrounding it.
- The algorithm now generates n random points (x, y), $0 \leq x, y \leq 1$ and checks if these points lie inside the the unit circle or not. The ratio P between the number of points lying inside the unit circle and the overall number of chosen points is now used to approximate π via

$$\pi \approx 4P.$$

Implement this algorithm using OpenMP and a `reduction`-clause to collect the results from the single worker-threads. Determine the critical number of random points, where the OpenMP overhead is vanished by the parallel execution of the algorithm. Please denote what processor you are using.
Hints: Read the `random` manpage to figure out how to generate random numbers.

Exercise 3: (6 Points)

Implement the *tree-reduction* for

$$S := \sum_{i=0}^{n} x_i$$

using OpenMP as a C-function called

```
double treesum (int n, double *x)
```

The implementation has to meet the following requirements:

- The length $n \geq 0$ of the vector x can be arbitrary.
- The vector x can be overwritten during the calculations.
- During each level of the tree reduction two elements should be combined. In this way the overall number of elements halves in every step.

Exercise 4: (10 Points)

Beginning with OpenMP 4 tasks can be scheduled by the use of their data dependencies. This is done using the `depend`-clause while defining a task. Let $A \in \mathbb{R}^{m \times n}$ be an arbitrary matrix.

a.) Create a task-parallel program (using OpenMP 4 dependencies) which fills the matrix A using the following rules:
- $A_{m,1} = \pi$,
- $A_{i,1} = \sin A_{i+1,1}$ for $i = 1 \ldots m-1$,
- $A_{m,j} = \cos A_{m,j-1}$ for $j = 2 \ldots n$, and
- $A_{i,j} = 2A_{i+1,j} + A_{i,j-1}$ otherwise.

b.) At the moment not all compilers support OpenMP 4 and higher on all platforms. For this reason it can be necessary to create an alternative parallel implementation using only the `parallel for` construct which is supported by all OpenMP aware compilers. Develop such an implementation for the filling strategy from a.) which only relies on the parallelization of loops.

Compare the results of both parallel implementations with a purely sequential implementation of the filling strategy to ensure correctness.

Overall Points: 30