Otto-von-Guericke-Universität Magdeburg
Max-Planck-Institut für Dynamik komplexer technischer 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 3
Solution

Handout: 05/12/2015 **Return:** 05/19/2015

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: (10 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 assume 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 with OpenMP. If race conditions occur, protect the affected operation 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 specify 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://www.mpi-magdeburg.mpg.de/mpcsc/lehre/2015_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. $r = 1$, 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: **(8 Points)**

The sparse-approximate inverse presented in the lecture is a naturally parallel to compute preconditioner. For a given sparse matrix $A \in \mathbb{R}^{n \times n}$ it is computed following

$$\min ||AP - I||_F^2 = \sum_{j=1}^{n} \min ||Ap_j - e_j||_2^2,$$

where P has the same pattern as A, p_j is the j-th column of P, and e_j is the vector with only a unit entry at position j. The key ingredient to an efficient parallel computation of the sparse approximate inverse is the fast solution of the appearing least-squares problems

$$\min ||Ap_j - e_j||_2^2.$$

a.) Derive a pseudo-code algorithm (MATLAB®-like notation), which solves this problem with respect to the fixed pattern of p_j. Furthermore, the algorithm has to avoid the solution of least-squares problems with the large original matrix A.

b.) Assume that the matrix $A \in \mathbb{R}^{n \times n}$ is symmetric with 10 entries in each row/column. Compute the number of flops depending on n to compute $P \in \mathbb{R}^{n \times n}$

Overall Points: 30