Project 1: Parallel Addition of Sparse Matrices using PThreads

Let *A* and *B* be two *sparse* *m*×*n* matrices, i.e., matrices most elements of which are zero; let *A* and *B* be stored in the so-called *Yale Sparse Matrix Format (YSMF).* Besides sparsity, there is no specific property or structure to be assumed for the matrices.

Write and test a parallel program using PThreads that computes *C* = *A* + *B*, where *C* is again a sparse *m*×*n* matrix and is to be stored in the YSMF. The number of threads *p* that your parallel program deploys, should be easily configurable.

Hints and explanations:

1. On sparse matrices and YSMF (adapted from: *Patterson, Hennessy. Computer Organization and Design. Fourth Edition. Morgan Kaufmann 2009;* but almost identically also to be found in Wikipedia):

> **Storing a sparse matrix**
>
> Many if not most entries of a sparse matrix are zeros. The basic idea when storing sparse matrices is to store only the non-zero entries as opposed to storing all entries. Depending on the number and distribution of the non-zero entries, different data structures can be used and yield huge savings in memory when compared to a naive approach.
>
> One example of such a sparse matrix format is the Yale Sparse Matrix Format. It stores an initial sparse *m*×*n* matrix, *M*, in row form using three one-dimensional arrays. Let `NNZ` denote the number of nonzero entries of *M*. The first array is `A`, which is of length `NNZ`, and holds all nonzero entries of *M* in left-to-right top-to-bottom order. The second array is `IA`, which is of length *m* + 1 (i.e., one entry per row, plus one). `IA(i)` contains the index in `A` of the first nonzero element of row `i`. Row `i` of the original matrix extends from `A(IA(i))` to `A(IA(i+1)-1)`. The third array, `JA`, contains the column index of each element of A, so it also is of length `NNZ`.
>
> For example, the matrix
>
> ```
> [ 1 2 0 0 ]
> [ 0 3 9 0 ]
> [ 0 1 4 0 ]
> ```
>
> is a three-by-four matrix with six nonzero elements, so
>
> ```
> A  = [ 1 2 3 9 1 4 ]
> IA = [ 1 3 5 7 ]
> JA = [ 1 2 2 3 2 3 ]
> ```

2. Make yourselves familiar with the YSMF, with generating this specific format (from the "obvious" matrix data structure), and with using it (i.e., accessing non-zero matrix elements). Then, think about the manipulation of the YSMF data structure, in particular what can be readily parallelized, what is less or barely suited for parallelization, and *what* needs to be locked *when*, in case of concurrent, modifying accesses (stores) to the data structure. Then, actually go about parallelizing the problem.

3. Start testing and debugging the program on a single-processor/core computer, then move on to a multi-core or multi-processor system. The matrices *A* and *B* should have integer elements and should be randomly initialized, i.e., have random non-zero elements at random positions in the matrices. You should, however, be able to control the degree of density, i.e., the fraction of non-zero elements in the matrices, e.g., 3%. Use very big matrices and YSMF data structures, respectively, up to the limit of storage you can allocate on the test machine; then, parallelization should pay off. The initialization of the matrices and the creation of the YSMF data structures for *A* and *B* need *not* be parallelized and *not* be measured.

4. Experiment with a varying number of threads *p*, e.g., *p* = 1, 2, 4, 6, 8, 12, 16. Determine and compare the *speedup* of the program with varying *p*. The performance metric is *program run time of the core sparse matrix addition*, i.e., without considering and measuring matrix

initialization and YSMF generation. The reference point, i.e., *speedup* = 1, is the program run time of the core sparse matrix addition with $p = 1$.

5. Work as two- or three-person teams. (Organize yourselves into the teams.) Please send me your solution until 12th May 2016, and do prepare a brief presentation of your approach, solution, and results. There should be one or two presentations on the 12th May; each team should present at least once in the course of the semester.

6. Please use your own (or some other available) multi-core machine; it should have at least two cores.