CMSE/CSE 822: Parallel Computing
Fall 2019, Homework 4
Due 11:59 pm, Monday, October 21st

Important note: *Please use a word processing software (e.g., MS Word, Mac Pages, Latex, etc.) to type your homework. Follow the submission instructions at the end to turn in an electronic copy of your work.*

1) [100 pts] OpenMP Parallel Sparse Matrix Vector Multiplication

The sparse matrix-vector multiplication (SpMV) describes the operation $\boldsymbol{b} = \boldsymbol{Ax}$ where $\boldsymbol{b}$ is the output vector, $\boldsymbol{A}$ is the input sparse matrix and $\boldsymbol{x}$ is the input vector. SpMV is an important kernel for myriad scientific applications, particularly for iterative solvers for sparse linear systems. There are special formats to store and process sparse matrices, the Compressed Sparse Column (CSC) and Compressed Sparse Blocks (CSB) formats are two commonly used ones among several others. Sparse matrix formats focus almost exclusively on efficient schemes to access nonzero elements in the matrix, while ignoring the zero entries. Adopting the right format based on the structure of the sparse matrix and application needs is vital to obtain satisfactory performance. In this problem, you are asked to create OpenMP parallel versions of sequential kernels related to sparse matrices, i.e., conversion of a sparse matrix given in the CSC format to the CSB format and performing the SpMV operation in both formats.

The CSC format stores a matrix with *numrows* rows, *numcols* columns and *nnonzero* non-zero entries using three 1D arrays:

- *irem* is an integer array of length *nnonzero* and contains the row indices of all non-zero entries in column-major order.
- *xrem* is a floating point array, which is also of length *nnonzero* and holds the values of all non-zero entries in column-major order.
- *colptrs* is an integer array size of *numcols + 1* where *colptrs[i]* denotes the starting position of column *i*'s entries on *irem* and *xrem* arrays and by definition, ***colptrs[i]=colptrs[i-1] + number of nonzero entries on column i***. Also note that for this assignment, we will assume that *colptrs[0] = 1* and *colptrs[nnonzero] = nnonzero+1*.

In CSB format, a sparse matrix gets divided into (*numrows/block_width) x (numcols/block_width)* blocks with each block being size of *block_width x block_width*. This formats then uses three integers (*nnz, roffset, coffset*) and three 1D arrays (*rloc, cloc, val*) to store the nonzero entries inside each block:

- *nnz* is the number of nonzeros in a given block, while *roffset* and *coffset* correspond to the row offset (*roffset*) and column offset (*coffset*) of the top left corner of the block in the big sparse matrix.
- *rloc, cloc and val* shows the row and column (*rloc[i]*, *cloc[i]* respectively) of a nonzero within that block along with the value (*val[i]*) of the i^{th} nonzero. As such, the tuple *(rloc[i] + roffset, cloc[i] + coffset, val[i])* would correspond to the position and value of that entry in the original matrix.

To understand CSC and CSB formats, consider the following matrix A:

$$A = \begin{pmatrix} 0 & 59.68 & -4.52 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.68 & 0 & 0 \\ 68.03 & 82.32 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -60.48 & 0 & 0 & 0 & -96.73 \\ -21.12 & -32.95 & 25.77 & 90.44 & -71.67 & 0 \end{pmatrix}$$

If we were to traverse the entries in column-major order and print them as (row, column, value) tuples, we would get:

3 1 68.03
6 1 -21.12
1 2 59.68
3 2 82.32
5 2 -60.48
6 2 -32.95
1 3 -4.52
6 3 25.77
2 4 2.68
6 4 90.44
6 5 -71.67
5 6 -96.73

The corresponding CSC representation of this matrix would be as follows:

irem: [3, 6, 1, 3, 5, 6, 1, 6, 2, 6, 6, 5]
xrem: [68.03, -21.12, 59.68, 82.32, -60.48, -32.95, -4.52, 25.77, 2.68, 90.44, -71.67, -96.73]
colptrs: [1, 3, 7, 9, 11, 12, 13]

Notice that *irem* and *xrem* consist of the numbers from the first and third column respectively, whereas *colptrs* displays where the entries from each column start and end on *irem* and *xrem*.

For CSB format with *block_width = 3*, for example, we are going to have *(6/3) x (6/3) = 4* blocks and the data that is stored by each block would be as follows:

Block[0][0], which is top left 3x3 submatrix:
nnz, roffset, coffset: 4, 0, 0
rloc, vloc, val: [3, 1, 3 1], [1, 2, 2, 3], [68.03, 59.68, 82.32, -4.52]

Block[0][1], which is top right 3x3 submatrix:
nnz, roffset, coffset: 1, 0, 3
rloc, vloc, val: [2], [1], [2.68]

Block[1][0], which is bottom left 3x3 submatrix:
nnz, roffset, coffset: 4, 3, 0
rloc, vloc, val: [3, 2, 3, 3], [1, 2, 2, 3], [-21.12, -60.48, -32.95, 25.77]

Block[1][1], which is bottom right 3x3 submatrix:
nnz, roffset, coffset: 3, 3, 3
rloc, vloc, val: [3, 3, 2], [1, 2, 3], [90.44, -71.67, -96.73]

a. [30 pts] You are given a sequential function for converting a sparse matrix in *CSC* format to *CSB* format. Implement an *efficient* OpenMP parallel version of this function by modifying *parallelMatrixConversion* in *parallelSpMV.c* as necessary.

b. [20 pts] You are given a sequential version of the SpMV operation for matrices in the CSC format. Implement an efficient OpenMP parallel version of this function by modifying *parallelCSC_SpMV* in *parallelSpMV.c* as necessary.

c. [20 pts] You are also given a sequential implementation of the SpMV operation for matrices in the CSB format. Implement an efficient OpenMP parallel version of this function by modifying *parallelCSB_SpMV* in *parallelSpMV.c* as necessary.

d. [30 pts] On the **intel16** cluster, experiment with 3 large sparse matrices, "it-2004", "twitter7" and "sk-2005" from the SuiteSparse Matrix Collection provided under "/mnt/gs18/scratch/users/alperena" with ".cus" extension in binary format. Measure the running time of your parallel CSC SpMV implementation using 1, 2, 4, 8, 14, 28 threads, your CSB matrix conversion and SpMV implementation using the same set of threads along with varying block sizes (2048, 8192, 32768) for each of the matrices. Plot the speedup you obtain over the sequential version of matrix conversion; also plot the speedup you obtain over the sequential version of SpMV vs. the number of threads for your OpenMP parallel CSC and CSB (with the a given block size) SpMV implementations. How scalable are the three OpenMP parallel function you implemented? Analyze your findings and explain your observations in regards to the good/bad scaling of your implementations.

Instructions:

- **Compiling and executing your program:** You can use any compiler with OpenMP support. The default compiler environment at HPCC is GNU, and you need to use the –fopenmp flag to compile OpenMP programs properly. Remember to set the OMP_NUM_THREADS environment variable, as necessary. Note that there is a makefile provided; to compile your code using the makefile, you simply need to use the "make" command, which will create a binary file named "spmv.x". You can later execute your program as follows:

```
./spmv.x matrix_file.cus block_size
./spmv.x /mnt/gs18/scratch/users/alperena/it-2004.cus 4096
```

- **Editing and debugging the code**: You are only supposed to edit the "parallelSpMV.c" file, and NOTHING ELSE. When you run the code, the performance of your implementations will automatically be compared against the corresponding functions' performance found in the "sequentialSpMV.c" file. The output to the stdout will indicate if any of your parallel versions is not producing the correct output (1 means they are correct, otherwise it will print the first position where your output and the expected output differ). Performance and speedup numbers will also be reported to the stdout.
- You are advised to test your code on one of the two small matrices, "sample.cus" which is the sample matrix discussed above and "Tina_AskCog.cus" with the size of 11x11 and 36 nonzero entries until your program is running correctly. Ignore the speedups reported for these two matrices as it might be *NaN* or *infinity*. Keep in mind that under the same directory, "/mnt/gs18/scratch/users/alperena/", you can view all the matrices with the 'txt' extension in ASCII format and they will be in the form of (row, colum, value) tuples in column-major order as in the example shown above whereas '.cus' files are in binary and CSC format. The given code will only read matrices stored in the ".cus" as the ASCII versions take too long to read from disk.

- **Measuring your execution time and performance properly:** The wall-clock time measurement mechanism (based on the gettimeofday() function) implemented in the provided source file will allow you to measure the timings for a particular part of your program (see the skeleton code) precisely. However, while they are convenient to use, the dev-nodes will have several other programs running simultaneously, and your measurements may not be very accurate due to the "noise".
  After making sure that your program is bug-free and executes correctly on the dev-nodes, a good way of getting reliable performance data for various input sizes is to use either the interactive queue or the batch jobs queue. Please refer to instructions in HW2 for guidance on how to use HPCC in these two different modes of execution. **Also, please use the intel16 cluster for all your performance measurement runs!**

  Note that a job script is essentially a set of bash commands that are executed one after the other. Assuming you have enough resources allocated (cores, wallclock time, memory), you should be able to execute multiple jobs in the same job script (and perform file/directory operation as needed in between).

- **Submission:** Your submission will include the following files:
  - A pdf file named "HW4_*yourMSUNetID*.pdf", which contains your answers to the questions in the assignment.
  - A source file "parallelSpMV.c", which contains the parallel implementation of all three functions discussed in part (a), (b) and (c).
  - Place both files directly under homework/4/ directory.

  Further instructions regarding the use of the git system and homework submission is given in the "Homework Instructions" under the "Reference Material" section on D2L. Make sure to strictly follow these instructions; otherwise you may not receive proper credit.