Optimizing Sparse Matrix Kernels

In this task, you will be optimizing the performance of the sparse triangular solve (and the sparse matrix-vector multiply as a voluntary step). Your code has to be in C or C++, however, if you decide to auto-generate C/C++ code (not easy!) you are free to use Python for the code generation (your final autogenerated code has to be C/C++). In the *Helpful links and documentation* section of this document, we have provided links and material to help you with the task.

Your optimizations can start with improving the performance of the algorithm for a single core and then should extend to multicore with parallelism. Remember to start with optimizing the serial code and make sure it executes correctly. You are most probably familiar with threads and parallelism from courses such as operating systems, computer architecture, or parallel computing. You should use the parallel programming model OpenMP. OpenMP is easy to learn, just see the links in the *Helpful links and documentation* section of this document. Prior experience with OpenMP is **not required**, just read these documents as needed and primarily focus on the algorithmic aspects of exploiting sparsity to better optimize the code.

Follow the below steps in order to write your code and populate the report. We prefer to see a perfect solution to the triangular solve rather than semi-optimized versions of the two algorithms.

1- Download the af_0_k101 matrix in matrix market format obtained from the below repository, all results should be reported for this matrix only, you need to learn how to read these matrices with the help of online resources:

https://sparse.tamu.edu/

2- The matrix is already stored in lower triangular.

3- The matrix has to be stored in the compressed column storage (CSC) format.

2- Use the b vectors b_sparse_af_0_k101 and b_dense_af_0_k101 as the right-hand sides for the triangular system. Some of the optimizations made for parallelism are easier to observe with the dense b vector while serial optimizations are more obvious with the sparse b vector. Any routine you have used for reading the matrix in the matrix market format should work for the two right-hand sides as well.

3- You can use the code in the *starter code* section of this document. These are naive implementations but can get you started. The sparse matrix-vector multiplication is only for the voluntary section of this task.

4- Your code should verify the final output. Do not report any results if the algorithm is not running correctly, your code should print the result of the verification. In the report, discuss the verification strategy used, also add comments to your code that explicitly highlight parts that are related to verification.

5- We want to see your coding style. So write well-commented and well-structured C or C++ code. Preferably use C++ and leverage C++ specific features to write a piece of code that is easy to extend. Think of someone else trying to use your code to add another sparse matrix routine to your library. For

example, use templates to enable extensions to single and double precision, to represent dense or sparse vectors, to support storage formats other than CSC, and so on.

6- You are free to use open source code and online resources but the sources have to be included in your report.

7- Voluntary step: Only if you are super satisfied with your sparse triangular solver optimizations or cannot come up with any other way to optimize it, proceed to optimize the sparse matrix-vector multiply. Use the matrix from the triangular solver section and multiply them with its corresponding b vectors. Optimize for single core and multicore. We still highly recommend that you only focus on the sparse triangular system solve since there are a lot of interesting and important optimizations you can do with that kernel.

Please provide the below:

1) your code with the makefile. Your code will be compiled and executed by us. It will be tested with a second sparse matrix for correctness. The second matrix will not be used to test performance since you are allowed to specifically optimize for performance for the sparsities of the provided matrix. Let us know if you are doing code generation (very impressive!) if so, we will also check for performance against the third matrix.

2) a short report that describes your optimizations, and also shows results with analysis. The running time of the naive code should be compared to the optimized code in speedup. Preferably, separate the speedups based on optimizations applied (or as a stacked bar). Separate the performance improvements and optimizations that you implement for single core from the optimizations that you implement for parallelism. Remember to explain how far you got in optimizing both the serial and the parallel versions. Write a report that explains all of your efforts.

Helpful links and documentation:

Useful links for OpenMP:
Video: https://www.youtube.com/watch?v=PR-98tVHFVg&feature=youtu.be
Slide: https://drive.google.com/file/d/1SmoOqlqIaC73m56l9slDDVUvBI0hokAo/view
More: https://computing.llnl.gov/tutorials/openMP/

Useful links for sparse triangular solve:
Video (up to minute 19): https://www.youtube.com/watch?v=7ph4ZQ9oEIc
Book (page 10): http://faculty.cse.tamu.edu/davis/publications_files/survey_tech_report.pdf

Useful links for compressed storage formats:
Section 3.4 of https://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf

Papers that might help:
http://www.cs.toronto.edu/~mmehride/papers/Parsy.pdf

http://www.cs.toronto.edu/~mmehride/papers/SC17.pdf

Starter code:

```
/*
* Lower triangular solver Lx=b
* L is stored in the compressed column storage format
* Inputs are:
* n : the matrix dimension
* Lp : the column pointer of L
* Li : the row index of L
* Lx : the values of L
* In/Out:
* x : the right hand-side b at start and the solution x at the end.
*/
int lsolve (int n, int* Lp, int* Li, double* Lx, double *x){
 int p, j;
 if (!Lp || !Li || !x) return (0) ;                    /* check inputs */
 for (j = 0 ; j < n ; j++)
 {
  x [j] /= Lx [Lp [j]] ;
  for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
  {
   x [Li [p]] -= Lx [p] * x [j] ;
  }
 }
 return (1) ;
}


/*
* Sparse matrix-vector multiply: y = A*x
* A is stored in the compressed column storage format
* Inputs:
* Ap : the column pointer of A
* Ai : the row index of A
* Ax : the values of A
* x : is a dense vector
* Output:
* y : is a dense vector that stores the result of multiplication
*/
int spmv_csc (int n, size_t *Ap, int *Ai, double *Ax,
              double *x, double *y)
{
 int p, j;
```

```
 if (!Ap || !x || !y) return (0) ;       /* check inputs */
 for (j = 0 ; j < n ; j++)
 {
  for (p = Ap [j] ; p < Ap [j+1] ; p++)
  {
   y [Ai [p]] += Ax [p] * x [j] ;
  }
 }
 return (1) ;
}
```