Homeworks 3-4: Sparse Matrix Vector Multiplication, parallelized

Stella Greenvoss
11/24/2025 — CS431
Computation done on ix-dev and ORCA

Function	Runtime	Speedup from serial	Speedup from max
COO Serial	2.05966	1	1
COO Parallel	0.09873	20.86264	20.86264
CSR Serial	1.58293	1	1.30118
CSR Parallel	0.05230	30.26799	39.38398

Table 1: Runtime and speedup for all functions. Computation done on ix-dev.

This paper details my experience attempting to parallelize sparse matrix vector multiplication using different methodologies and different storage formats. The first section describes the process of parallelizing for the CPU using C and OpenMP, with the storage formats COO and CSR. The second section discusses parallelizing for the GPU using CUDA and C++ with the storage formats CSR and ELLPACK. In this particular paper, my focus will be more towards describing my process towards parallelizing these functions (rather than an attempt at describing solid academic findings).

Part of my difficulties in this process involved manually editing the Makefile and SLURM script to run on ORCA, the Oregon Regional Computing Accelerator. Their environment is set up differently to Talapas', so I had to copy some header files from the CUDA repository, change CUDA versions, and rewrite the SLURM script to properly load modules and submit jobs.

1 CPU Parallelism

This section will describe serial and parallel versions of SPMV using both COO and CSR.

1.1 COO and conversion to CSR

COO, or *COOrdinate* format, stores sparse matrices using three columns: row, indices, and data. Each column has a total of nnz elements, where nnz is the number of nonzero elements in the sparse matrix.

Computing the matrix-vector product serially using a matrix stored in COO is simple[1] and runs in $O(nnz)$. Special care must be taken to ensure off-by-one safety, since the arrays stored in COO tend to be 1-indexed. As might be expected, this serial code performed the worst of the four functions. This is because much less work is being done per cycle, since a single thread must do every computation itself (assuming no compiler optimization). The data access pattern is also suboptimal, being linear for the row and col array accesses but without a clear pattern for r/w to and from the result and input, which negatively contributes to the temporal locality and thus the runtime.

The parallel version of COO-stored SPMV was also straight-forward, although there may be a more complicated implementation that performs better. Since I achieved a $20x$ speedup from the serial version, I decided that this would be sufficient for now. The algorithm is essentially the same, but with two parallel regions: one to initialize the result array and one to compute and add values to the result. To prevent simultaneous access, writing to a position in the result array in parallel must be done using an atomic pragma. It could also be done using an omp_lock, but my brief experiments with using that seemed to show that it was much slower.

Converting from COO to CSR was difficult, as parts of it had to be parallel. I had never done a parallel histogram before and had to use (and fix) my previous implementation of parallel prefix sum. For simplicity, I chose to use the simpler version of parallel prefix sum, which likely makes it overall less efficient than it could be.

[1] See *spmv_coo_ser* in homework3/main.c

1.2 CSR

CSR, or *Compressed Sparse Row* format, stores data and col arrays like COO. However, it also stores a third array, called row_ptr, which stores pointers to the first value in every row. This allows it to store rows of varying sizes in a compressed manner, removing the extraneous repeated values from COO's row array.

Implementing the serial version is relatively simple, although the relative complexity (compared to COO) made it so that reasoning about the indices takes a little extra work. The function has a smaller asymptotic complexity than the serial COO function, since the matrix's sparsity means that the inner for loop cannot always be equal to n, the number of columns. This improvement likely accounts for the small speedup ($1.3x$ COO serial). On ix-dev, at least, the overhead from converting COO to CSR is not worth the tiny performance boost from a serial CSR SPMV implementation.

However, parallelized CSR SPMV was by far the best function, with a $30x$ speedup from the serial version. These performance gains were achieved via two simple OpenMP pragmas, parallelizing the initialization loop and the outer loop of the main computation. Schedule(static) seemed to achieve better performance than dynamic scheduling. The huge performance gains might be from the improved temporal locality achieved through the new data format, or as a consequence of the asymptotic improvement described in the serial section. Either way, it is clear that on a CPU with sufficient threads, CSR is the optimal storage format for computing SPMV. This should be (and certainly has been) explored further using different types of matrices.

An interesting final observation for the CPU-based part of this assignment is that I ran the executable on the large cant matrix on both ix-dev and a login node on ORCA. Ix-dev produced the behavior as seen in table 1. On ORCA, the distribution was bizarre, with parallel COO taking 2.5 seconds, almost twice as long as both serial functions. Looking at the output of lscpu on ORCA, the login node only has four CPUs with one thread per core. In contrast, ix-dev has 64, with two threads per core. Therefore, it makes perfect sense that parallel functions run slower on a node that is so ill-equipped to handle parallelism.