

# CS406: Parallel Computing

## Homework 1

### Sparse Matrix–Vector Multiply (SpMV) with OpenMP

Sabanci University

October 30, 2025

## Goal

Implement and evaluate a high-performance Sparse Matrix  $\times$  Dense Vector multiplication (SpMV) for large, real-world sparse matrices using OpenMP. You will measure performance under different thread counts and experiment with preprocessing that improves locality (e.g., reordering).

## Problem Description

You are expected to implement iterative multiplication of sparse matrix-dense vectors. We consider a matrix 'sparse' if the number of elements is much less than the total number of elements (so the idea is that it has few non-zero compared to the size of the matrix). When dealing with sparse matrices, it is generally better to use compressed storage formats like CRS (Compressed Row Storage) and CCS (Compressed Column Storage). If you do not know these formats, you can search for these formats online (i.e., Sparse Matrix), but for convenience, we explain the CRS format briefly.

**CRS (Compressed Row Storage) arrays.** Storing a sparse matrix in CRS uses three arrays: `crs_ptrs`, `crs_colids`, and `crs_values`. For an  $n \times m$  sparse matrix:

- **`crs_ptrs`:** This array has size  $n + 1$ . The first element is `crs_ptrs[0] = 0`. For  $i \geq 1$ ,

$$\text{crs\_ptrs}[i] = \text{crs\_ptrs}[i - 1] + \text{nnz}(\text{row } i - 1),$$

i.e., it stores the cumulative number of nonzeros up to (but not including) row  $i$ . Equivalently,  $\text{crs\_ptrs}[i] = \sum_{r=0}^{i-1} \text{nnz}(\text{row } r)$ . Thus, if you traverse nonzeros row-wise, `crs_ptrs[i]` is the count of nonzeros until row  $i$ . (Note: entries for row  $i$  live in indices  $k \in [\text{crs\_ptrs}[i], \text{crs\_ptrs}[i+1])$ .)

- **`crs_values`:** This array has length `nnz` (the total number of nonzeros). Scanning the matrix row-wise, store each nonzero value in order into `crs_values`.
- **`crs_colids`:** This array also has length `nnz`. For every stored nonzero value, record its column index in `crs_colids` at the same position, giving a one-to-one correspondence between `crs_values` and `crs_colids`.

## Datasets / Matrices

You will use the following matrices from the SuiteSparse Matrix Collection (Matrix Market `.mtx` format) for testing / evaluating your implementation:

- DIMACS10/netherlands.osm (road network; low degree, planar-like)
- Delaunay\_n21 (geometric synthetic graph)
- SNAP/soc-LiveJournal1 (social network; heavy-tailed degree)

## Provided Baseline Code

We provide `spmv.cpp` that:

1. Parses a Matrix Market file into a global CSR structure: `G_row_ptr`, `G_col_idx`, `G_vals`.

```
std::vector<int>    G_row_ptr; // length: G_N+1
std::vector<int>    G_col_idx; // length: nnz
std::vector<double> G_vals;    // length: nnz

std::cerr << "[load] " << path << "\n";
read_matrix_market_to_global_csr(path); // reads into variables above
```

2. A placeholder function that we expect you to implement:

```
void spmv(const std::vector<double>& x, std::vector<double>& y);
```

3. Performs exactly **50 iterations** of SpMV with vector swapping, measures **kernel time**, computes a checksum, and prints: `time_sec`, `gflops`, `checksum`.

Your task is to implement and optimize `spmv`. You may add pre-processing (data structure changes, re-ordering, etc.) as long as you document it in your report. You will use OpenMP.

### Compiling:

```
g++ -O3 -std=c++17 -fopenmp spmv.cpp -o spmv
```

### Running:

```
./spmv path/to/matrix.mtx
```

## Requirements

- Use exactly **50** SpMV iterations (the code already does this).
- Measure and report performance for thread counts **1, 2, 4, 8, 16** You may control threads using OpenMP (`OMP_NUM_THREADS`) or in-code configuration.

- For each matrix, report: **time (seconds)** and **GFLOPS** computed as

$$\text{GFLOPS} = \frac{\text{Number of Floating-Point Operations}}{\text{Execution Time (s)} \cdot 10^9}$$

- Exclude file I/O from the kernel timing (the baseline does this), and clearly state if you include/exclude preprocessing time.

You may attempt *heavy pre-processing* to improve spatial/temporal locality or balance work:

- Alternative data structures (e.g., row reordering, blocking).
- Scheduling policies (OpenMP **static/dynamic/guided** with chunk sizes).
- Matrix reordering: RCM (Reverse Cuthill–McKee), AMD (Approx. Minimum Degree), or tools such as **SparseViz** (<https://github.com/sparcityeu>).

If you use a library or external code, cite it and briefly describe what it does.

## Deliverables

Submit a single zip (`surname_name.zip`) containing:

1. Your source code(s).
2. A PDF report that complies the following instructions:
  - It should include the pre-processing and implementation tricks you used (at least a few sentences about how you attacked the parallelization problem). Please notice that your code will be graded considering correctness, scalability and speed.
  - It needs to contain build and execution details – e.g., how did you compile the code (which instruction set, optimization parameter etc.) and how I can reproduce your results.
  - 1-2 pages reports are usually not OK.
  - Do not increase the number of pages by leaving more spaces in between lines.
  - Perform multiple experiments and report both average execution times and GFLOPS for 1-2-4-8-16 threads. You need to compute the GFLOPS correctly (take every arithmetic operation into account).
  - Charts and tables must have correct and appropriate captions.

GOOD LUCK!