# Homework Report: Parallel Sparse Matrix-Vector Multiply (SpMV) using OpenMP

Ethan Reinhart

November 14, 2024

## Contents

# 1 Introduction

Sparse matrix-vector multiplication (SpMV) is a fundamental operation in scientific computing and various engineering applications. It involves multiplying a sparse matrix, which contains a significant number of zero entries, by a dense vector. Efficient implementation of SpMV is crucial for performance, particularly when handling large datasets.

In this project, we focus on implementing SpMV in both serial and parallel environments using OpenMP. The sparse matrices are represented using two popular formats:

- **COO (Coordinate)**: A format that stores non-zero entries using a triplet structure consisting of row indices, column indices, and values.
- **CSR (Compressed Sparse Row)**: A more compact format that organizes non-zero values by rows, allowing for faster arithmetic operations.

The goal is to implement the following:

1. Conversion from COO to CSR format.
2. Serial and parallel SpMV implementations using COO and CSR formats.
3. Performance evaluation of the parallel implementation compared to the serial versions.

The target matrix for testing is the 'cant' matrix, a sparse matrix. This report documents the implemented algorithms' methodology, results, and analysis.

# 2 Methodology

## 2.1 COO and CSR Formats

To efficiently perform SpMV, understanding the data formats for storing sparse matrices is crucial:

- **COO Format**: Each non-zero element is stored as a tuple of (row index, column index, value). This format is simple to understand and easy to construct but lacks efficiency for matrix-vector operations.
- **CSR Format**: Non-zero elements are stored row-by-row, using three arrays: 'csr_vals' for non-zero values, 'csr_col_ind' for column indices of these values, and 'csr_row_ptr' for indexing the start of each row in the 'csr_vals' array.

## 2.2 Implemented Functions

The project involved the implementation of the following key functions:

1. `convert_coo_to_csr` - A function to convert matrices from COO format to CSR format.
2. `spmv_coo` - A parallel version of SpMV using COO format with OpenMP.

3. `spmv` - A parallel version of SpMV using CSR format with OpenMP.
4. `spmv_coo_serial` - A serial version of SpMV using COO format.
5. `spmv_serial` - A serial version of SpMV using CSR format.

### 2.3 Testing and Validation

The implementations were validated using the 'cant' matrix:

- The output of each function was compared to a reference answer using a user-created script, printing the average difference between predicted and ground truth values.
- The COO and CSR implementations were adjusted to work with 0-indexed arrays, even though input matrices are 1-indexed.
- Numerical accuracy was checked, allowing for small floating-point errors ($< 10^{-6}$).

## 3 Results

### 3.1 Performance Analysis

The performance results of the sparse matrix-vector multiplication (SpMV) implementations are summarized in Table 1.

Table 1: Performance results for SpMV implementations

| **Module** | **Time (seconds)** |
|---|---|
| Load | 0.856103 |
| Convert | 0.042845 |
| Lock Init | 0.000232 |
| COO SpMV | 0.311959 |
| CSR SpMV | 0.085028 |
| COO SpMV Serial | 1.892937 |
| CSR SpMV Serial | 1.850571 |
| Store | 0.029387 |

### 3.2 Key Observations

- **Loading and Conversion**: The matrix was successfully loaded from the `cant/cant.mtx` file with a size of 62,451 x 62,451 and 4,007,383 non-zero elements, meaning the matrix is 0.10275% dense. The conversion from COO format to CSR format was efficient, taking only 0.042845 seconds, indicating that the conversion step does not impose a significant overhead.

- **Parallel vs. Serial Performance**: The parallel implementations (`Parallel COO SpMV` and `Parallel CSR SpMV`) significantly outperform the serial versions. In particular, the `Parallel CSR SpMV` implementation achieved the fastest execution time of 0.085028 seconds, while the serial version of CSR took 1.850571 seconds, indicating a significant speedup due to parallelization. For the COO format, the parallel implementation ran in 0.311959 seconds, while the serial version took 1.892937 seconds. This highlights that parallelization benefits both formats, but CSR shows the most improvement.

- **CSR vs. COO Formats**: The CSR format consistently showed better performance compared to the COO format, both in parallel and serial cases. This improvement is due to the CSR format's better memory access patterns and cache efficiency, as it stores row pointers which help in accessing data in a more structured way. The difference is particularly notable in the parallel implementation, where `Parallel CSR SpMV` is approximately 3.7 times faster than `Parallel COO SpMV`.

- **Storage and Lock Initialization**: Storing the results in `my.ans` took a negligible amount of time (0.029387 seconds). Lock initialization time for handling parallelism was minimal (0.000232 seconds), indicating that synchronization overhead did not significantly affect performance.

## 3.3 Speedup Analysis

- **Speedup for COO Format**:

$$\text{Speedup (COO)} = \frac{1.892937}{0.311959} \approx 6.06 \tag{1}$$

  This demonstrates a speedup factor of approximately 6.06 times when using the parallel version of the COO format.

- **Speedup for CSR Format**:

$$\text{Speedup (CSR)} = \frac{1.850571}{0.085028} \approx 21.77 \tag{2}$$

  This highlights a speedup factor of approximately 21.77 times with the parallel CSR implementation.

## 3.4 Conclusion

The experiment clearly shows that:

- **Parallelization** is crucial for optimizing SpMV performance, with CSR format exhibiting the most notable gains.

- The **CSR format** outperforms COO in both parallel and serial contexts due to its structured representation, which enhances memory locality and reduces computational overhead.

- In high-performance applications involving large, sparse matrices, leveraging the **CSR format in combination with parallel processing** yields substantial performance benefits.

These results align with the expected efficiency of CSR for sparse matrix operations and illustrate the advantages of parallel computing for large-scale data processing tasks.

# 4 Discussion

## 4.1 Effectiveness of Parallelization

The parallel implementations demonstrated clear advantages over the serial versions, particularly for the CSR format. OpenMP efficiently utilized multiple CPU cores, resulting in a significant reduction in computation time for larger matrices. The parallel CSR implementation showed an impressive speedup factor of approximately 21.77 compared to its serial counterpart, demonstrating the substantial benefits of parallelization.

## 4.2 CSR vs. COO Performance

The CSR format consistently outperformed the COO format in both serial and parallel settings. The row-based structure of CSR allows for reduced memory accesses and better data locality, which are crucial for improving performance during matrix-vector operations. In parallel implementations, CSR showed the most notable speedup, with the parallel CSR version running approximately 3.7 times faster than the parallel COO version. This highlights the efficiency of CSR, especially in large-scale sparse matrix computations.

## 4.3 Accuracy Considerations

The ground truth and the parallel CSR results were compared using a Python script, ‘avg.py‘, which computed the average difference between the ground truth and the parallel CSR output. Remarkably, the average difference was 0.0 in each case, indicating that the parallel CSR implementation produced results that were virtually identical to the ground truth. The same method was used to test on serial CSR implementation as well as parallelized and serial COO implementation and yielded the same results.

# 5 Conclusion

This project successfully implemented parallel SpMV operations using both COO and CSR formats with OpenMP. The CSR format was found to be more efficient for both serial and parallel implementations due to its optimized structure for matrix-vector operations. The parallel CSR implementation, in particular, demonstrated a substantial speedup, reinforcing the advantages of utilizing multi-core architectures for sparse matrix computations. Additionally, the accuracy of the parallel CSR implementation was verified to be consistent with the ground truth, further validating the correctness of the parallelization.

# 6 References


- NVIDIA, *Efficient Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors*, `https://www.nvidia.com/docs/IO/77944/sc09-spmv-throughput.pdf`.
- Lecture notes on Sparse Matrix Formats and Parallel Computing.