

Deliverable 1: Sparse Matrix-Vector Multiplication

Sepa Matteo (243283)

Course: Introduction to Parallel Computing (2025–2026)

University of Trento

matteo.sepam@studenti.unitn.it

github.com/sepatmatteo/PARCO-Computing-2026-243283

Abstract—This report presents a study of shared-memory parallelization of sparse matrix-vector multiplication (SpMV). We implement and benchmark SpMV using Compressed Sparse Row (CSR). The implementations are written in C++ and parallelized using OpenMP. We evaluate their performance on a real-world matrix from the SuiteSparse collection. Our results show that, the parallel CSR implementation highly beats the sequential one. This report details our implementations, benchmarking methodology, and the analysis of the performance results. The project, including source code and scripts for reproducibility, is available on GitHub.

Index Terms—SpMV, sparse matrices, OpenMP, parallel computing, CSR, COO, SIMD, vectorization, performance analysis.

I. INTRODUCTION

Sparse matrix-vector multiplication (SpMV), the operation $y = Ax$ where A is a sparse matrix and x a monodimensional vector, is a fundamental kernel in many scientific and engineering applications. Its performance is critical for a wide range of domains. The efficiency of SpMV is highly dependent on the data structure used to store the sparse matrix, as it affects memory access patterns and computational efficiency [10].

This project focuses on the implementation and performance analysis of SpMV using the popular sparse matrix format **Compressed Sparse Row (CSR)**: A more compact format that enables efficient row-wise access to the matrix elements.

We develop sequential and parallel implementations of SpMV in C++ and use OpenMP for shared-memory parallelization. We conduct a comparative performance analysis of these implementations on a real-world large-scale sparse matrix and compare the result from a regular machine (i7-7700k, 16GB of RAM) and the HPC cluster. This report presents our methodology, experimental setup, performance results, and an in-depth discussion of the observed performance characteristics.

II. METHODOLOGY

The SpMV kernels is implemented in C++11 while we use the ANSI C library for Matrix Market I/O for matrix reading [1]. We use `chrono::steady_clock` from the `chrono` library for highly precise monotonic clock. We also randomly generate the vector to multiply with the matrix using the `random` library

A. Sparse Storage Formats

1) *Coordinate (COO)*: The COO format stores each non-zero element as a triplet (r , c , v), where r is the row index, c is the column index, and v is the value. These triplets are stored in three separate arrays: `row_idx`, `col_idx`, and `values`. The SpMV operation in COO format is implemented by iterating over the non-zero elements and performing the update $y[\text{row_idx}[k]] += \text{values}[k] \cdot x[\text{col_idx}[k]]$.

2) *Compressed Sparse Row (CSR)*: The CSR format uses three arrays: `values`, `col_idx`, and `row_ptr`. The `values` and `col_idx` arrays store the non-zero values and their corresponding column indices, ordered by row. The `row_ptr` array stores the starting index of each row in the `values` and `col_idx` arrays. The SpMV operation in CSR format involves iterating through each row i , and for each row, iterating through its non-zero elements from `row_ptr[i]` to `row_ptr[i+1]-1`.

B. Sequential CSR SpMV

Firstly we implement a standard conversion from COO format to CSR format for a sparse matrix with M **rows** and nz **non-zero elements**. COO stores non-zero entries as separate lists of row indices, column indices, and values, while CSR compresses the data using a row pointer array, column indices array, and values array for efficient row-wise access.

Key Steps in the conversion:

- **Count Non-Zeros per Row**: Initialize a `row_ptr` array of size $M+1$ to zero. For each COO entry, increment `row_ptr[row_coo[i]] + 1` to tally non-zeros in each row.
- **Compute Prefix Sums**: Perform a cumulative sum on `row_ptr` so that `row_ptr[i]` indicates the starting index in the column indices and values arrays for row i and `row_ptr[M]` equals nz .
- **Fill Column Indices and Values**: Create a temporary fill array copying the initial `row_ptr` values, serving as pointers to the current position in each row's segment. Iterate through COO entries in order, placing the column index (`col_coo[i]`) and value (`val_coo[i]`) into `col_idx` and `values` at position `fill[row_coo[i]]`, then increment the pointer for that row. This ensures entries are sorted by row without explicit sorting

This process achieves $O(nz)$ time complexity and preserves the original entry order within rows, enabling fast matrix-vector multiplications in CSR format. The input file uses 1-based indexing, which is adjusted to 0-based for internal use.

```

1  for (int i = 0; i < BENCHMARK_ITERS; ++i) { // ←
2    BENCHMARK_ITERS=10
3      std::fill(y.begin(), y.end(), 0.0); // Reset y
4      auto start = std::chrono::steady_clock::now(); ←
5      // starting measurement
6      for (int j = 0; j < M; j += BLOCK_SIZE) {
7        int j_end = std::min(j + BLOCK_SIZE, M);
8        #pragma omp simd
9        for (int r = j; r < j_end; ++r) {
10          double sum = 0.0;
11          #pragma omp simd reduction(+:sum)
12          for (int k = row_ptr[r]; k < row_ptr[r + ←
13            1]; ++k) {
14            sum += values[k] * x[col_idx[k]];
15          }
16          y[r] = sum;
17        }
18      }
19      auto end = std::chrono::steady_clock::now(); // ←
20      // ending measurement
21      auto elapsed = std::chrono::duration<double, ←
22      std::milli>(end - start);

```

Listing 1: Sequential CSR SpMV Kernel

This code applies **SIMD vectorization** to accelerate the SpMV computation in CSR format. SIMD (Single Instruction Multiple Data) enables simultaneous execution of the same operation on multiple data points, leveraging wide vector registers in modern CPUs.

The reduction clause for SIMD vectorizes accumulation operations by giving each SIMD lane a private copy of the sum variable and combining results at loop end, guaranteeing **correctness**.

C. Parallel CSR SpMV Implementation with OpenMP

We use OpenMP to parallelize the SpMV computation.

- **Parallel CSR-SpMV:** The parallelization is done over the rows of the matrix. We use an OpenMP parallel for loop with a different schedules to distribute the rows among the threads.

```

1 // BLOCK_SIZE=12; BENCHMARK_ITERS=10;
2 for (int i = 0; i < BENCHMARK_ITERS; ++i) {
3   auto start = ←
4   std::chrono::steady_clock::now(); // start timing
5   #pragma omp parallel
6   {
7     #pragma omp for schedule(static) nowait
8     for (int i = 0; i < M; i++) {
9       y[i] = 0.0;
10    }
11    #pragma omp for schedule(guided, ←
12    BLOCK_SIZE) nowait
13    for (int r = 0; r < M; ++r) {
14      double sum = 0.0;
15      #pragma omp simd reduction(+:sum)
16      for (int k = row_ptr[r]; k < ←
17        row_ptr[r+1]; ++k) {
18        sum += values[k] * x[col_idx[k]];
19      }
20      y[r] = sum;
21    }
22  }

```

```

19 } // implicit barrier here
20 auto end = std::chrono::steady_clock::now(); ←
21 // stop timing
22 auto elapsed = std::chrono::duration<double, ←
23 std::milli>(end - start);

```

Listing 2: Parallel SpMV Kernel

III. EXPERIMENTS

A. System Description

The experiments were conducted on a system with the following specifications:

- **Compiler:** GCC (g++) 9.1.0
- **Compiler Flags:** -O3 -std=c++11 -Wall -fopenmp -ffast-math -march=native

B. Dataset

We used the different matrices from the SuiteSparse Matrix Collection [2] with different sizes and Non-zero elements:

- **1138_bus** $1,138 \times 1,138$ matrix with 4,054 **non-zeroes** [3]
- **bcsstk18** $11,948 \times 11,948$ matrix with 149,090 **non-zeroes** [4]
- **cage14** $1,505,785 \times 1,505,785$ matrix with 27,130,349 **non-zeroes** [5]
- **nlpkkt160** $8,345,600 \times 8,345,600$ matrix with 225,422,112 **non-zeroes** [6]
- **Queen_4147** $4,147,110 \times 4,147,110$ matrix with 316,548,962 **non-zeroes** [7]

C. Benchmarking Methodology

The benchmarks were executed using the `src/runner.sh` script. For each implementation (COO, sequential CSR, parallel CSR), the script performs 3 warm-up runs followed by 10 timed runs. The execution times of the timed runs are recorded in text files in the `benchmarks` directory. A Python script, `benchmarks/script.py`, can be used to calculate the average execution time and 90th percentile and generate a plot of the results. We ran the HPC benchmark on the `short_cpuQ`

IV. RESULTS AND DISCUSSION

The 90th-percentile execution times for the CSR-based SpMV kernels are reported in Table I (desktop PC, i7-7700K) and Table II (HPC cluster). Figure 1 visualises both the results on a logarithmic scale.

A. Overall Speed-up Trends

On the **desktop PC**, parallel CSR yields a 2-3x speed-up with 4 threads and reaches approximately 3-4x at 8 threads for the larger matrices (cage14, nlpkkt160, Queen_4147).

The 16-thread configuration occasionally regresses (e.g., 1138_bus jumps from 0.0038 ms at 8 threads to 0.0985 ms) because the matrix is too small to compensate for OpenMP overhead and cache thrashing.

On the **HPC cluster**, the sequential baseline is **markedly slower** than the PC (e.g., 414 ms vs. 123 ms for Queen_4147).

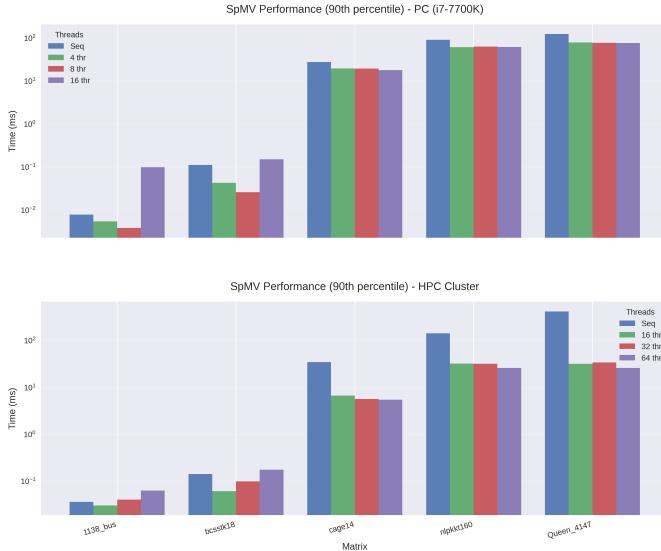


Fig. 1: SpMV 90th percentile execution time on PC and HPC (log scale).

TABLE I: Performance of parallel CSR SpMV on **i7-7700k**(90th-percentile time over 10 runs). GFLOPS and effective bandwidth (GB/s) refer to the best observed configuration

Matrix	Time (ms)				GFLOPs	GB/s
	Seq	4 thr	8 thr	16 thr		
1138_bus	0.0078	0.0054	0.0038	0.0985	0.11	1.090
bcsstk18	0.1112	0.0428	0.0259	0.1506	3.29	23.64
cage14	27.427	19.608	19.368	17.839	2.32	14.98
nlpkkt160	90.554	60.966	63.009	61.798	2.80	18.39
Queen_4147	123.30	78.198	76.841	76.085	3.41	21.16

TABLE II: Performance of parallel CSR SpMV on **HPC Cluster** (90th-percentile time over 10 runs). GFLOPS and effective bandwidth refer to the best observed configuration

Matrix	Time (ms)				GFLOPs	GB/s
	Seq	16 thr	32 thr	64 thr		
1138_bus	0.0362	0.0304	0.0403	0.0629	0.21	2.040
bcsstk18	0.1421	0.0610	0.0985	0.1750	3.58	25.71
cage14	34.461	6.6561	5.6460	5.4590	5.12	33.02
nlpkkt160	141.16	31.910	31.816	25.836	4.54	29.80
Queen_4147	414.30	31.802	33.905	25.780	4.84	30.02

Parallel execution at 16 threads delivers 5-13x speed-up, peaking at **approximately 16x** for Queen_4147 when moving to **64 threads**.

B. Matrix-Specific Scaling Behaviour

- Small, highly sparse matrices** (1138_bus, bcsstk18) benefit modestly up to 8 threads on the PC but degrade sharply beyond that because the work per row is insufficient to amortize thread creation and scheduling costs. This behavior is consistent with findings that static scheduling fails to handle irregular row distributions effectively [9].

- Medium-to-large irregular matrices** (cage14) show excellent scaling on the HPC node (approximately 6x from sequential to 64 threads) once enough threads are available to mask memory latency. On the PC, scaling plateaus after 8 threads owing to limited core count.

- Dense, regular problems** (nlpkkt160, Queen_4147) exhibit the most consistent gains: approximately 1.5x per doubling of threads on the PC up to 8 cores, and a clear progression from 16 → 32 → 64 threads on the HPC, culminating in **approximately 16x** overall speed-up for Queen_4147.

C. Thread Scaling Behavior on HPC

Although tested up to 64 OpenMP threads, **peak performance** occurs at 16 to 32 **threads** for most matrices. The two largest problems show continued improvement to 64 threads, but gains diminish.

This plateau is consistent with **memory-bound behavior** [10]: SpMV performs one FLOP per two irregular memory references. Once available memory bandwidth is saturated, additional threads increase contention without improving throughput.

D. Scheduling

The row-wise parallelisation of CSR-SpMV is inherently **load imbalanced** because the number of non-zero elements per row varies widely in real-world matrices. OpenMP provides several schedule clauses to distribute loop iterations among threads [9].

TABLE III: OpenMP scheduling strategies for CSR-SpMV row parallelisation.

Schedule	Chunk size	Work distribution	Overhead
static (default = $M/\text{threads}$)	fixed at compile time deterministic, no runtime cost	poor for irregular row lengths	
dynamic	user-defined (e.g. 64)	threads request chunks when idle	high (lock + scheduling)
guided	starts large, shrinks exponentially	self-balancing	moderate, adaptive

1) Experimental evaluation of static scheduling: We executed the CSR-SpMV kernel on the Queen_4147 matrix ($4,147,110 \times 4,147,110$, 316 M non-zeros) with the **static** OpenMP schedule (chunk size 64). Ten timed runs were performed after three warm-up iterations on an HPC node. Table III summarises the three classic policies. In our experiments the guided schedule gave the best trade-off between load-balance and overhead.

TABLE IV: 90th-percentile execution time (ms) with static schedule on Queen_4147.

Implementation	90th percentile (ms)
Sequential CSR	279.2
Parallel (16 thr)	71.6
Parallel (32 thr)	114.3
Parallel (64 thr)	535.2

Observation. Static scheduling yields a respectable 3.9x speedup at 16 threads (279.2 ms → 71.6 ms). Increasing the thread count to 32 threads **regresses** performance because

the fixed chunk assignment cannot cope with the highly irregular row lengths of Queen_4147; a few threads finish early while others process rows with >100 k non-zeros. At 64 threads the overhead of thread creation and severe load imbalance cause a **dramatic slowdown**. This confirms that static is unsuitable for large irregular matrices and motivates the use of adaptive policies such as guided [9].

E. Impact of Scheduling on Cache Behavior

We profiled the parallel CSR-SpMV kernel (32 threads, guided schedule) on the cage14 matrix using Valgrind/Callgrind with full cache simulation: To quantify the effect of OpenMP scheduling on cache efficiency, we repeated the Callgrind profiling of one full SpMV iteration on the cage14 matrix using exactly the same binary and 32 threads, differing only in the schedule.

We only profiled the SpMV Kernel using callgrind with the flags : --tool=callgrind --simulate-cache=yes --collect-atstart=no --dump-instr=yes and these callgrind toggles before and right after the SpMV Kernel.

```

1 CALLGRIND_TOGGLE_COLLECT;
2 // .. SpMV Kernel Here
3 CALLGRIND_TOGGLE_COLLECT;
```

Listing 3: Callgrind activation in SpMV Kernel

Key cache statistics:

TABLE V: Callgrind cache statistics for cage14 (16 threads, one SpMV iteration).

Metric	guided	static	Difference
Instruction references (Ir)	95.1 M	63.9 M	-33%
Data references (Dr+Dw)	44.5 M	29.5 M	-34%
D1 misses (total)	3.75 M	2.87 M	-24%
D1 miss rate	8.4%	9.7%	+15%
LL misses (total)	2.79 M	1.94 M	-30%
LL miss rate	6.3%	6.6%	+5%
LL write miss rate	15.3%	15.0%	≈

Key observations:

- The static schedule executes **fewer instructions and memory references** (approximately one-third less) because **severe load imbalance** causes many threads to finish early and sit idle — the effective work is done by only a few heavily loaded threads.
- Despite doing less total work, the **D1 miss rate increases from 8.4% to 9.7%** and the **LL miss rate rises from 6.3% to 6.6%**. This occurs because the same hot rows are repeatedly processed by the same threads, destroying temporal locality in the values and col_idx arrays.
- Write-related false sharing (approximately 15--17% D1 write misses) remains essentially unchanged — both schedules suffer from concurrent writes to nearby elements of y.
- Overall, guided scheduling incurs **approximately 30% more cache misses** but distributes work evenly, yielding **significantly lower wall-clock time**.

Interpretation:

- The kernel executes approximately **95 M instructions** and approximately **44 M data references** per SpMV — consistent with approximately 1.6--2 memory operations per non-zero.
- The **last-level cache miss rate of 6.3%** (approximately 2.8 M compulsory/capacity misses) is the dominant performance limiter. Each LL miss costs approximately 200--250 cycles → approximately 560--700 million cycles spent waiting for memory.
- Arithmetic intensity is extremely low (approximately 0.05--0.06 FLOP/byte), making SpMV **strongly memory-bound** [10] — performance scales with memory bandwidth, not core count, beyond approximately 16--32 threads.

This confirms that static scheduling trades lower absolute miss counts for dramatically worse load balance and higher miss rates per operation, making guided (or dynamic) the clearly superior choice for irregular matrices such as cage14.

V. CONCLUSION

We successfully implemented and benchmarked sequential and parallel SpMV kernels using the CSR format on both a desktop PC (i7-7700K) and an HPC cluster node. The results provide clear insights into the effectiveness of shared-memory parallelization with OpenMP:

- OpenMP parallelization delivers substantial speedups**, achieving up to **4–6x** on well-balanced, large matrices (nlpkkt160, Queen_4147) with 8–16 threads. However, small or highly irregular matrices (1138_bus, cage14) show limited scaling or even performance degradation at high thread counts due to load imbalance and parallel overhead.
- Single-thread performance is critical**: The HPC node exhibits **significantly slower sequential execution** than the desktop CPU, inflating parallel efficiency metrics. This highlights that **speedup alone is misleading** without considering baseline performance.
- Load balancing and matrix structure dominate scalability**: The best scaling is observed on dense, regular problems; irregular sparsity patterns (e.g., cage14 at 32 threads) lead to severe underutilization despite increased parallelism.

Overall, while OpenMP enables effective parallelization of CSR-based SpMV on shared-memory systems, **achieving consistent high performance requires careful consideration of matrix properties, thread scheduling, and underlying hardware characteristics**.

REFERENCES

- [1] Matrix Market I/O routines in ANSI C. <https://math.nist.gov/MatrixMarket/mmio-c.html>.
- [2] T. A. Davis, The SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection). <https://sparse.tamu.edu/>.
- [3] 1138_bus matrix from HB collection. https://sparse.tamu.edu/HB/1138_bus.

- [4] bcsstk18 matrix from HB collection. <https://sparse.tamu.edu/HB/bcsstk18>.
- [5] cage14 matrix from vanHeukelum collection. <https://sparse.tamu.edu/vanHeukelum/cage14>.
- [6] nlpkkt160 matrix from Schenk collection. <https://sparse.tamu.edu/Schenk/nlpkkt160>.
- [7] Queen_4147 matrix from Janna collection. https://sparse.tamu.edu/Janna/Queen_4147.
- [8] Introduction to Parallel Computing — Report and project preparation guidelines, University of Trento, 2025–2026 (course material, accessed Nov 10, 2025).
- [9] O. Asudeh et al., “Is Sparse Matrix Reordering Effective for Sparse Matrix-Vector Multiplication?,” arXiv:2506.10356 [cs.DC], Jun. 2025.
- [10] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, “Bandwidth-Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking,” arXiv:2002.11302 [cs.DC], Feb. 2020.