Parallel Sparse Matrix-Vector Multiplication (SpMV) on Shared Memory: CSR Baseline, OpenMP Scheduling and SIMD

Francesco Lionello
Department of Information Engineering and Computer Science
University of Trento, Italy
https://github.com/francesco-lionello/parco_d1

***Abstract*—Sparse Matrix-Vector Multiplication (SpMV) is a fundamental kernel in scientific computing and learning applications. Despite its apparent simplicity, SpMV is typically memory-bound due to irregular access patterns. This work implements and benchmarks both sequential and parallel CSR-based SpMV using OpenMP on a shared-memory system. The evaluation focuses on execution time (in milliseconds, 90th percentile over $\geq$10 runs), scaling with threads and impact of OpenMP scheduling policies (static, dynamic, guided) across five matrices with different sparsity levels. The results confirm that memory bandwidth and load imbalance remain the primary scalability limits in CSR-based SpMV.**

I. Introduction

Sparse Matrix-Vector Multiplication (SpMV) is one of the most important operations in high-performance computing, scientific simulations, and machine learning. It appears in linear solvers, graph processing, and iterative algorithms where sparse structures arise naturally. Despite its apparent simplicity, SpMV is among the most performance-critical kernels in modern architectures.

The core challenge lies in the memory-bound nature of the operation. In the computation $y = A \cdot x$, the sparse matrix A typically contains only a small fraction of non-zero elements, stored using compressed formats such as *Compressed Sparse Row* (CSR). Each multiplication involves irregular memory accesses to vector x through indirect indexing, which limits cache efficiency and makes the operation memory-bound rather than compute-bound.

On modern multi-core processors, parallel programming models such as OpenMP enable shared-memory parallelization of SpMV by distributing the computation of rows across multiple threads. However, scalability is constrained by several factors:

1) irregular sparsity pattern leading to load imbalance among threads;
2) memory access contention when multiple threads fetch elements of x simultaneously;
3) saturation of main memory bandwidth beyond a certain number of threads.

The objective of this work is to analyze the performance of CSR-based SpMV in both sequential and parallel implementations. Starting from a sequential baseline, develop and evaluate an OpenMP version using different scheduling policies (static, dynamic, and guided) to investigate their effect on load balancing and performance. Additional experiments include a SIMD-enhanced variant to exploit vectorization where possible.

The main contributions of this work are:

- a systematic benchmark comparing sequential and OpenMP implementations on a shared-memory architecture, evaluating execution time (milliseconds), speedup, and bandwidth efficiency;
- the identification of the main performance bottlenecks, particularly memory bandwidth saturation and scheduling overhead, with discussion of potential optimizations.

The results demonstrate that SpMV achieves near-linear speedup up to a limited number of threads, after which performance flattens due to bandwidth limitations. These finding align with previous studies [1], [2], confirming that the key challenge in parallel SpMV lies in improving data locality rather than computation itself.

II. Related Work

Sparse Matrix-Vector Multiplication (SpMV) has been extensively studied as one of the most memory-bound operations in scientific and engineering applications. Many efforts aim to improve its performance through architectural optimizations, new data structures, and parallelization techniques.

Williams et al. [1] conducted one of the most comprehensive analysis of SpMV performance on emerging multicore architectures. They demonstrated that the operation is fundamentally limited by memory bandwidth rather than floating-point throughput, and the performance is strongly dependent on data layout and cache behaviour.

Buluç et al. [2] proposed a novel data format called *Compressed Sparse Blocks* (CSB), designed to enabling fine grained parallelism and efficient execution of both Ax and $A^{\mathsf{T}}x$. CSB partitions the sparse matrix into tiles stored in Z-Morton order, improving data locality and reducing synchronization overhead. They showed $\Theta(\text{nnz})$work and $\Theta(\sqrt{n}\log n)$ span, offering near-linear speedup on multicore systems until memory bandwidth saturation occurs.

These studies provide a strong foundation for understanding the trade-offs between data layout, memory hierarchy and parallel execution. While the CSR format remains the most widely adopted due to its simplicity and compatibility with row-parallel OpenMP implementations, recent research emphasizes that further gains can be achieved through storage schemes and dynamic scheduling strategies. Building on these insights, this work adopts CSR as a baseline representation and focuses on evaluating thread-level scalability and scheduling performance in a shared-memory environment using OpenMP.

III. Methodology

This section describes the implementation strategy adopted for the Sparse Matrix-Vector Multiplication (SpMV) on a shared-memory system. The focus is on the conversion of Matrix Market files into an efficient memory layout, the design of sequential and parallel kernels , and the benchmarking methodology used to assess performance.

A. Matrix Format and Conversion

Input matrices are provided in the Matrix Market (.mtx) format, which stores data in Coordinate (COO) representation through triplets (i, j, v). The program first reads these files and converts them into the *Compressed Sparse Row* (CSR) format, chosen for its compactness and suitability for row-parallel computation. The CSR format consists of three arrays: `row_ptr`, which marks the start of each row; `col_idx`, which stores column indices; `val`, which contains the non zero values.

B. Sequential Implementation

The sequential baseline computes $y = A \cdot x$ by iterating over each row i and accumulating the product of nonzero elements:

$$y_i = \sum_{p=row\_ptr[i]}^{row\_ptr[i+1]-1} val[p] \cdot x[col\_idx[p]]$$

The code is implemented in C with careful attention to memory access locality and floating-point precision. Each run initializes the input vector x with random values and measures execution time over multiple iterations to reduce noise. The sequential kernel serves as a reference for evaluating the scalability of parallel implementations.

C. OpenMP Parallelization

The parallel version exploit OpenMP to distribute the computation of independent rows across multiple threads. A parallelization is achieved using the directive:

```
#pragma omp parallel for schedule(runtime)
```

This approach avoids race conditions because each thread writes to a distinct portion of the output vector y. Different scheduling strategies (*static*,*dynamic*,*guided*) are evaluated to explore trade-offs between load balancing and scheduling overhead. These policies follow the guidelines discussed by Pacheco and Malensek [3], where static scheduling minimizes runtime overhead for regular workloads, while dynamic and guided scheduling improve load balance for irregular sparsity patterns at the cost of higher management overhead.

D. SIMD Optimization

A vectorized variant is implemented using compiler assistant SIMD instructions through the directive:

```
#pragma omp simd reduction(+:s)
```

This hints to the compiler that independent iterations of the inner loop can be executed in parallel using vector registers, potentially improving performance when data alignment and sparsity patterns at the cost of higher management overhead.

E. Benchmarking Setup

All versions (sequential, OpenMP, and OpenMP+SIMD) are benchmarked using twenty independent runs per matrix. For each kernel, both the average and the 90th percentile execution times are reported. Five matrices with varying sparsity levels are selected to analyze how matrix density and structure influence the performance.

The experiments measure:

- Execution time (milliseconds)
- Effective memory bandwidth (GB)
- Speedup and parallel efficiency

To ensure fairness, memory allocations are aligned to 64-byte boundaries using `posix_memalign`, and each benchmark includes a warm-up phase to mitigate cache effects. The same random vector x is reused across all runs to maintain consistency.

F. Summary

The implementation design follows a bottom-up design: to starting from a scalar reference kernel, parallelism is progressively introduced through OpenMP and SIMD vectorization. By combining multiple scheduling strategies, the methodology aims to characterize how architectural and algorithmic factors influence the scalability of SpMV on shared-memory systems.

IV. Experiments and Results

A. Experimental Setup

All experiments were executed on an HPC node equipped with a multi-core CPU supporting OpenMP. Each run was repeated twenty times, and the 90th percentile execution time was reported to minimize variance caused by system noise. Memory allocations were aligned to 64-byte boundaries using `posix_memalign`.

Table I summarizes the five matrices selected from the SuiteSparse Matrix Collection, which different in dimension and sparsity level.

TABLE I
Test Matrices Used in the Benchmark

Matrix	**Size(M×N)**	**NNZ**	**Density(%)**
iris_dataset_30NN	150×150	5518	24.5244
bfwa62	62×62	450	11.7066
1138_bus	1138×1138	4054	0.3133
mnist_test_norm_10NN	10000×10000	145600	0.1456
Spielman_k200	2,686,802×2,686,802	8,100,804	0.0001

B. *Sequential vs. Parallel Performance*

Figure 1 compares the sequential and OpenMP implementations across all matrices. As expected, small matrices (e.g., `bfwa62`, `1138_bus`) are dominated by parallel overheads, and the sequential kernel remains faster. As the matrix size increases, parallelization provides significant gains: `mnist_test_norm_10NN` achieves nearly one order of magnitude improvement, while `Spielman_k200` benefits the most from OpenMP parallelism.

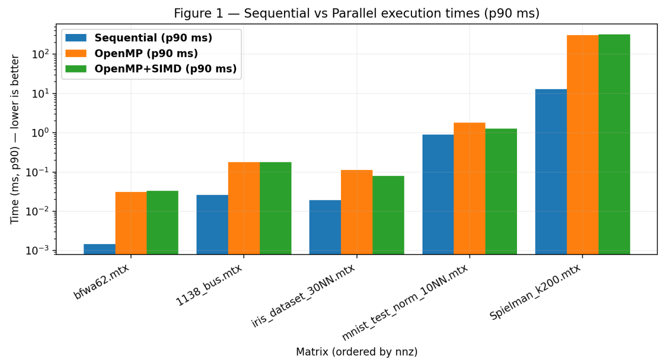

Fig. 1. Sequential vs. OpenMP and OpenMP+SIMD (p90 ms, log-scale). Lower is better.

C. *Speedup and Scalability*

Figure 2 illustrates the speedup obtained by increasing the number of threads. The scaling is nearly linear up to 16 threads and begins to flatten beyond that due to memory bandwidth saturation. This behavior is consistent with the findings of Buluç [2] and Williams [1], who also identified memory throughput as the dominant bottleneck in SpMV. This observation motivates the following analysis of scheduling policies and bandwidth utilization.

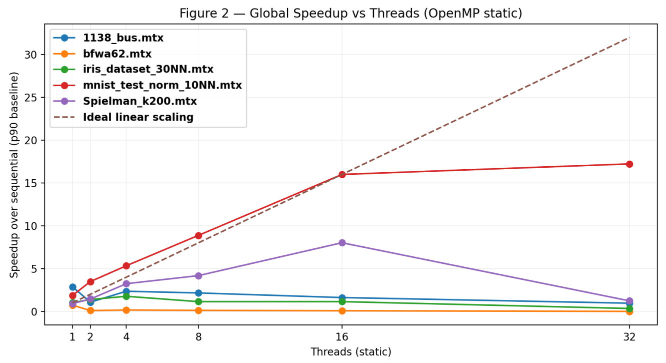

Fig. 2. Global speedup over the sequential p90 baseline vs. number of threads (OpenMP static).

D. *Effect of Scheduling Strategies*

Figure 3 compares static, dynamic, and guided scheduling policies at 16 threads (p90 ms, log-scale). Static scheduling achieves the lowest runtime overall, confirming its efficiency for balanced workloads due to minimal synchronization overhead. Dynamic and guided scheduling better handle irregular sparsity by redistributing work at runtime, but their management overhead dominates, especially on small or moderately sized matrices.

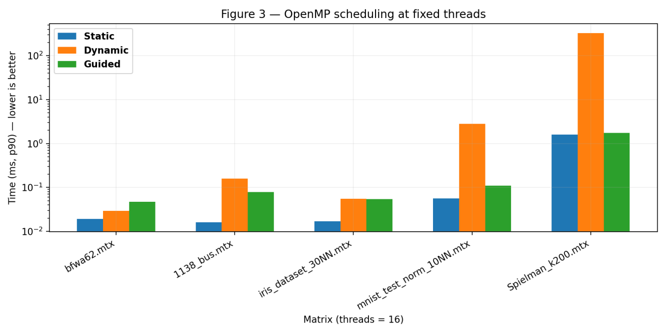

Fig. 3. OpenMP scheduling policies at 16 threads (p90 ms, log-scale).

E. *Bandwidth and Density Analysis*

Figure 4 illustrates the effective memory bandwidth achieved with static scheduling across varying thread counts. For small and moderately sized matrices, the bandwidth remains low due to limited parallel work and high synchronization cost. Only the largest dataset (`Spielman_k200`) scales significantly, reaching over 250 GB/s and approaching the observed peak bandwidth (271.6 GB/s). All curves flatten beyond 8–16 threads, confirming a transition to a memory-bound regime where performance no longer improves with additional parallelism. This behavior indicates that SpMV throughput is ultimately constrained by main-memory bandwidth rather than computational capacity.

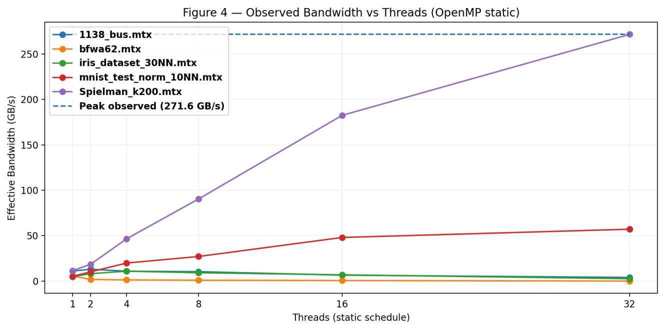

Fig. 4. Observed effective bandwidth vs. threads (OpenMP static). The dashed line marks the highest observed GB/s.

V. Discussion and Conclusion

A. Discussion

Across Figures 1–4, the results consistently show that CSR-based SpMV on a shared-memory node is constrained by data movement rather than computation. In Figure 1, small inputs (`bfwa62`, `1138_bus`) remain faster in the sequential baseline due to parallel overheads (thread start-up, synchronization) that dominate at low `nnz`. As the problem size grows, OpenMP becomes effective: `mnist_test_norm_10NN` improves by nearly one order of magnitude, and `Spielman_k200` benefits the most from parallel execution.

Figure 2 highlights scalability limits: speedup increases almost linearly up to 8–16 threads and then flattens or degrades at 32 threads. This trend correlates with the bandwidth curves in Figure 4: only the largest matrix (`Spielman_k200`) approaches the observed peak (271.6 GB/s), while the others saturate earlier, indicating that the memory subsystem becomes the bottleneck once a moderate level of parallelism is reached.

Scheduling policies (Figure 3) further clarify the trade-off between balance and overhead. Static scheduling is generally best because it minimizes runtime management costs on CSR row-parallel loops whose row-length distribution is not extremely skewed. Dynamic and guided can reduce tail latency on irregular sparsity but their queueing/atomic overheads outweigh the benefits on small and medium matrices; guided tends to be a slightly better compromise than dynamic. Finally, SIMD offers modest gains (5–10%) because indirect indexing on `col_idx` limits vector utilization and prefetch effectiveness; benefits appear mainly when contiguous nonzeros occur within rows.

Overall, the evidence from all figures is consistent:

1) performance scales with `nnz` and working-set size until the memory interface saturates;
2) static scheduling is the safest default for CSR on CPUs;
3) additional threads beyond 8–16 do not translate into proportional speedups due to bandwidth saturation and likely NUMA contention.

B. Conclusion

This study implemented and evaluated sequential, OpenMP, and OpenMP+SIMD versions of CSR SpMV, analyzing execution time (p90), speedup, scheduling, and effective bandwidth on five matrices of varying sparsity. The main conclusions are:

- SpMV is memory-bound: speedup is near-linear only up to the point where memory bandwidth saturates (typically 8–16 threads).
- Static scheduling achieves the lowest runtime across most cases; dynamic/guided help only for pronounced irregularity, but usually incur higher overheads.
- SIMD yields limited improvements because indirect accesses hamper vector efficiency in CSR.

These results demonstrate that SpMV performance on shared-memory systems is ultimately constrained by memory bandwidth rather than computation. Static scheduling provides the best trade-off between simplicity and efficiency, while SIMD acceleration remains limited by indirect accesses in CSR.

VI. References

References

[1] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix–Vector Multiplication on Emerging Multicore Platforms," in *Proc. of the ACM/IEEE Conference on High Performance Computing (SC)*, 2007.

[2] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel Sparse Matrix–Vector and Matrix-Transpose–Vector Multiplication Using Compressed Sparse Blocks," in *Proc. of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

[3] P. S. Pacheco and M. Malensek, *An Introduction to Parallel Programming*, 2nd ed., Morgan Kaufmann, 2022.