SpMV optimization with OpenMP

Project Report

De Toffoli Alessandro (ID: 243644)
Course ”Introduction to Parallel Computing”
Department of information engineering and computer science - DISI
University of Trento
Email: alessandro.detoffoli@studenti.unitn.it

***Abstract*—I present a parallel implementation of Sparse Matrix–Vector Multiplication (SpMV) on a multicore, shared-memory system using OpenMP. The main objective of the project was to benchmark the performance of the SpMV algorithm by applying parallelization techniques and to assess throughput and scalability improvements over a sequential baseline. Concretely, I evaluate different combinations of thread counts and scheduling policies across five symmetric matrices from the SuiteSparse, collecting both execution times and cache statistics. The experiments show that performance improvements emerge only under specific combinations of thread counts and scheduling, and even in those cases the achievable speedup is ultimately limited by a memory-access bottleneck.**

Artifacts. Source code, scripts, and instructions for full reproducibility are provided (see Sec. VII).

***Index Terms*—Sparse Matrix–Vector Multiplication (SpMV), OpenMP, Performance Benchmarking, Memory Bottleneck**

I. Introduction

Problem. The Sparse Matrix–Vector Multiplication (SpMV) is an essential operation in numerous scientific, engineering, and data-intensive applications. Its efficiency is critical, as SpMV is often invoked within contexts where computational cost and memory behavior directly influence overall performance. Despite its conceptual simplicity, SpMV is well known for being memory-bound since the irregular memory access patterns make it challenging to achieve high performance on modern architectures.

Importance. In the context of parallel computing, SpMV represents an important benchmark for evaluating the effectiveness of shared-memory parallelization strategies. Multicore processors provide opportunities to accelerate this kernel, but performance gains strongly depend on workload distribution, thread scheduling, and the ability of the memory subsystem to keep pace with increased parallel demand. As a result, analyzing how SpMV behaves under different parallel configurations offers valuable insights into both algorithmic scalability and architectural limitations.

Objectives. (i) Design and implement a parallel SpMV algorithm; (ii) evaluate its performance under different configurations; (iii) compare it against a sequential baseline; (iv) document the algorithm's efficiency and the system's limitations.

II. State of the art

Sparse Matrix–Vector Multiplication (SpMV) has been extensively studied as a representative memory-bound kernel: its low arithmetic intensity and irregular memory access patterns make performance primarily limited by memory bandwidth rather than computational throughput. Previous work has focused on:

- **storage formats**, as CSR, to improve locality and reduce indexing overhead [1]
- **parallelization strategies** where OpenMP scheduling significantly impacts performance [2]
- **architecture-aware optimizations** (e.g., NUMA placement, software prefetching, cache blocking) and **high-performance libraries** that provide highly tuned implementations but remain constrained by the memory subsystem and matrix sparsity structure [3]

Despite extensive optimization efforts, SpMV implementations typically achieve only a fraction of the theoretical peak due to high LLC miss rates, limited data reuse, and bandwidth saturation. Performance is highly matrix-dependent, and the most effective scheduling or storage strategy varies with both sparsity pattern and hardware characteristics.

III. Contribution and Methodology

Contributions. I propose a simple performance analysis of both sequential and parallel implementations of the Sparse Matrix–Vector Multiplication (SpMV) algorithm on a multicore shared-memory system. I examine how different OpenMP scheduling policies and thread counts affect execution time, cache behavior, and scalability across a diverse set of real symmetric matrices from the SuiteSparse collection [4] with an increasing number of non-zero (NNZ) elements.

Methodology. This section describes the methodological approach used to implement, parallelize, and evaluate the Sparse Matrix–Vector Multiplication (SpMV) program.

A. Program structure

The SpMV algorithm is implemented across three main `C++` source files:

- `main.cpp`: manages the overall program workflow, including matrix parsing and compression, vector generation, execution of the SpMV operation, and timing measurements.

- `csr.h / csr.cpp`: implement the Compressed Sparse Row (CSR) data structure and the matrix-vector multiplication routine.

B. Data structure

The matrix is stored in Compressed Sparse Row (CSR) format using a `CompressedSparseRow` object. The object contains the following attributes:

- `csr_row`: an array of size (n_{rows}+1) storing the starting index of each row in the `csr_col` and `csr_val` arrays.
- `csr_col`: an array containing the column indices of all non-zero elements in the matrix.
- `csr_val`: an array storing the values of all non-zero elements corresponding to the indices in `csr_col`.

This structure allows efficient row-wise iteration and is well-suited for parallelization of the SpMV operation [1].

C. Algorithm overview

For each row, the algorithm identifies the range of non-zero elements via the `csr_row` array and retrieves their column indices and values from `csr_col` and `csr_val`. Each value is multiplied by the corresponding entry in the input vector, and the products are accumulated to produce the output for that row. This procedure is repeated across all rows, yielding the final dense result vector.

Algorithm 1 SpMV (CSR)

```
procedure SpMV($vec$)
    $result \leftarrow$ vector of size $n_{rows}$
    for $i = 0$ to $n_{rows} - 1$ do
        $sum \leftarrow 0$
        for $j = csr\_row[i]$ to $csr\_row[i+1] - 1$ do
            $sum \leftarrow sum + csr\_val[j] \cdot vec[csr\_col[j]]$
        end for
        $result[i] \leftarrow sum$
    end for
    return $result$
end procedure
```

D. Parallelization (OpenMP)

The parallel implementation leverages a row-level parallelization strategy employing `#pragma omp parallel for schedule(runtime)`, allowing the use of different scheduling policies such as `static`, `dynamic`, and `guided`. These scheduling strategies are evaluated to identify the optimal load-balancing configuration, given that the computational cost of each row may vary depending on the number of non-zero elements. Furthermore, `#pragma omp simd` is applied to the inner `for` loop to enable vectorization and enhance data-level parallelism.

E. Complexity

Work is proportional to the number of non-zero elements (NNZ) rather than the matrix dimension. As a consequence, the overall performance is primarily influenced by memory traffic and spatial locality effects, which dominate the execution time due to the irregular access patterns typical of sparse matrices.

IV. Experiments and System Description

A. Platform

Architecture. Experiments were conducted on the University of Trento's HPC cluster, a distributed–memory system. All tests ran on a single compute node equipped with up to 64 CPU cores and 32 GB of RAM.

OS & Toolchain. The node runs CentOS Linux 7. The software stack includes `gcc/9.1` with OpenMP support for compiling and executing the parallel implementations.

Datasets. The evaluation employs five symmetric matrices in the Matrix Market format from the SuiteSparse Matrix Collection [4], characterized by an exponential growth in size and number of NNZ, enabling performance assessment under increasing computational loads.

TABLE I: Overview of the matrices used for benchmarking.

Matrix	Alias	Rows	Non-zeros
mawi_201512012345	little	18,571,154	38,040,320
mawi_201512020000	small	35,991,342	74,485,420
mawi_201512020030	medium	68,863,315	143,414,960
mawi_201512020130	big	128,568,730	270,234,840
mawi_201512020330	huge	226,196,185	480,047,894

System Instability and Performance Variance. Different testing sessions revealed significant performance fluctuations: execution times reported high variance, heavily influenced by the instantaneous state of the cluster. Consequently, the absolute performance metrics proved to be highly sensitive to environmental conditions, limiting the universality of the measurement outside the specific testing window adopted for the comparison.

B. Metrics and Methodology

Algorithm has been executed 10 times for every combination of scheduling policy, SIMD usage, and number of threads for each matrix, more specifically:

- **Scheduling policy**: `static`, `dynamic`, `guided`
- **SIMD option**: with and without `# pragma omp simd`
- **Number of OMP threads**: 4, 8, 16, 32, 64

For each execution, CPU time, real time, and the cache-related metrics obtained from `perf stat` were collected. All jobs were submitted and executed concurrently through `PBS` scripts to ensure consistent cluster conditions and minimize performance variability. Moreover, for each configuration, matrices have been involved in alternating order so that each run started from a cold-cache state, avoiding bias from residual data locality.

C. Baselines

The sequential version of the SpMV algorithm was executed for all matrices to establish baseline performance metrics. The same data—CPU time, real time, and cache statistics—were collected for comparison with the parallel implementations.

D. Statistical Analysis: Speedup and Efficiency

For each configuration, speedup and efficiency were computed to provide a clear assessment of parallel performance across different thread counts and scheduling policies.

- **Speedup** was calculated as

$$\text{Speedup} = \frac{T_{90}^{\text{sequential}}}{T_{90}^{\text{parallel}}}$$

where T_{90} is the 90th percentile real time.

- **Efficiency** was computed as

$$\text{Efficiency} = \frac{\text{Speedup}}{N_{\text{threads}}}$$

V. Results and Discussion

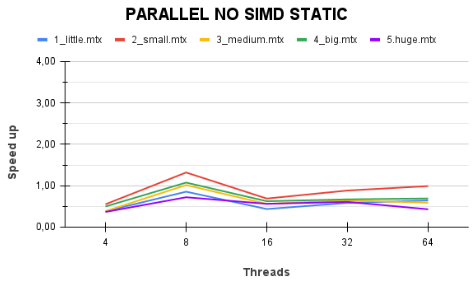
A. Results

The experimental time measurements indicate that performance improvements were achieved only under specific configurations of thread counts and scheduling policies.

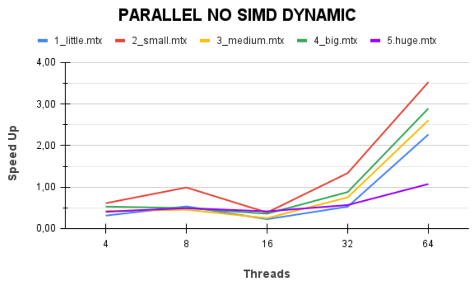
- **Static scheduling** (Figure 1a). Proved to be the least effective, yielding a negligible speedup (max `1.3x` at 8 threads) and degrading performance in other configurations.
- **Dynamic scheduling** (Figure 1b). At 64 threads it emerged as the most efficient strategy, achieving speedups between 2.5x and 3.5x for most datasets; however, it failed to provide significant gains for the largest matrix.
- **Guided scheduling** (Figure 1c). Offered a middle ground: it outperformed static but generally lagged behind dynamic. It showed optimal results at 8 and 16 threads, achieving the peak speedup for the largest matrix in the 8-thread configuration.

Despite these gains, the overall parallel efficiency remains critically low and rarely exceeds 5%.

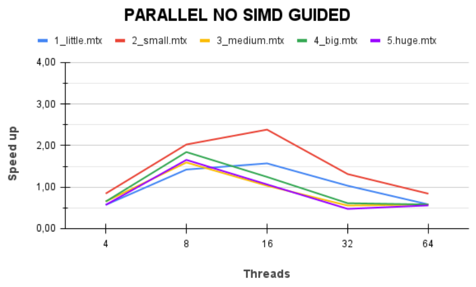
SIMD. The integration of SIMD vectorization resulted in marginal performance improvements across most scenarios in both dynamic and guided strategies, showing the most impact on the weakest configurations (Figure 2a, Figure 2b). While parallel efficiency occasionally reached 10% and execution times decreased in sub-optimal setups, the overall speedup profile remained largely consistent with the non-SIMD implementation. The observed gains typically represent a recovery from critically low baselines rather than a substantial increase in peak throughput. Consistent with previous tests, dynamic scheduling proved to be the optimal choice for most datasets. The notable exception was once again the largest matrix, where guided scheduling achieved a significant improvement, increasing the speedup from 1.6x to 2.1x.



(a) static



(b) dynamic



(c) guided

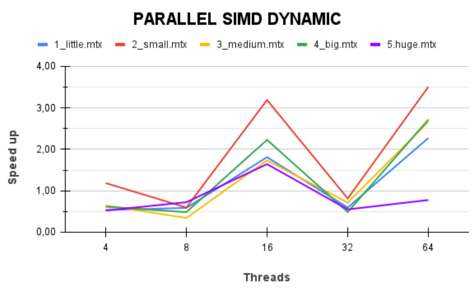
Fig. 1: Scheduling speedups

B. Discussions

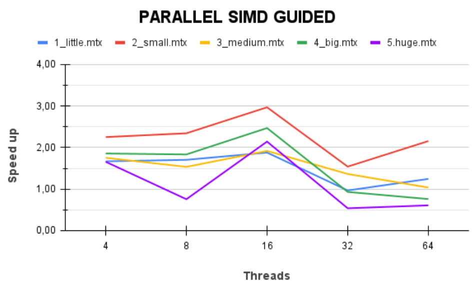
Although the IPC and L1 cache miss rates exhibit negligible variation across the majority of configurations, significant performance disparities are attributed to the Last Level Cache (LLC) miss rate. Specifically, the LLC miss rate was observed to even double in sub-optimal configurations compared to the optimal ones. This metric closely follows the scheduling trends discussed previously, strongly indicating that the primary system bottleneck is memory latency rather than computational throughput or parallelization overhead. Furthermore, the LLC miss rate was observed to decrease in SIMD-enabled configurations. This reduction suggests that vectorized memory instructions exploit spatial locality more effectively, causing the little performance improvements cited above.

The performance disparity between scheduling policies reveals a critical interplay between load balancing and synchronization overhead:

- **Failure of Static Scheduling**: The consistently poor performance of static scheduling confirms severe load



(a) dynamic



(b) guided

Fig. 2: Scheduling speedups with SIMD

imbalance [5].

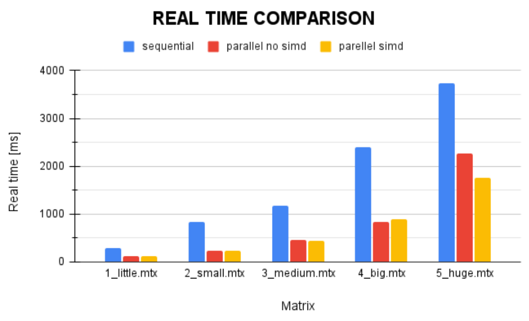
- **Success of Guided at Intermediate Scales**: The guided policy emerged as the optimal strategy for moderate thread counts. By dynamically reducing chunk sizes, it effectively mitigates the straggler effect without incurring the prohibitive synchronization costs associated with fine-grained dynamic scheduling.
- **Dynamic Scheduling and High Concurrency**: The behavior of dynamic scheduling, indicates a high synchronization overhead per task. For lower thread counts, the cost of atomic operations to fetch individual iterations outweighs the computational benefit for the typically sparse rows. However, at 64 threads, the massive parallelism allows the system to amortize this overhead and effectively hide memory latency, finally enabling the dynamic load balancing to yield a net positive speedup. In the largest matrix, this pattern changes since with a working set far exceeding LLC capacity, memory traffic becomes extremely high, and the frequent atomic operations required by dynamic scheduling introduce additional contention on the shared work queue. At 64 threads, this synchronization overhead competes with the SpMV computation and amplifies the impact of LLC misses, cancelling any parallel benefit.

VI. Conclusion and Future Works

The project successfully demonstrated the feasibility of parallelizing the SpMV algorithm and achieving measurable speedup, though performance gains were strictly dependent on the configuration utilized.

The optimal general-purpose setup was identified as Dynamic scheduling utilizing 64 threads, a configuration which yielded supplementary gains upon the integration of SIMD vectorization. Crucially, scalability was fundamentally restricted by memory access latency. This memory-bound characteristic dominated the execution time, establishing memory bandwidth as the primary limiting factor for further speedup.

The overall execution time was reduced by approximately 65% through parallelization. Furthermore, the runtime across the benchmarked matrix set exhibited an exponential scaling trend, which directly correlates with the exponential growth in the non-zero count (NNZ) of the individual datasets (Figure 3a).



(a) best real time comparison

Fig. 3: Final results

Future work will focus on refined chunk-size tuning and on strategies to mitigate memory-access latency and cluster variability. A warm-cache testing protocol is also planned to assess sustained throughput. Nonetheless, preliminary observations indicate that its impact will be limited, as all matrices far exceed LLC capacity and the computation remains fundamentally memory-bound.

VII. Reproducibility and Artifact Availability

The source code, raw experimental data, and complete instructions to run the experiment are publicly hosted at the GitHub repository: https://github.com/Ale-Deto04/PARCO-Computing-2026-243644/tree/main/openmp/.

References

[1] A. A. K. Al-Saeedi, S. O. Yurievna and T. W. Khairi, *Improving the efficiency of sparse matrix class processing by using the SPM-CSR parallel algorithm and OpenMP technology*, 2022 4th International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE), Moscow, Russian Federation, 2022

[2] S. Ohshima, T. Katagiri and M. Matsumoto, *Performance Optimization of SpMV Using CRS Format by Considering OpenMP Scheduling on CPUs and MIC*, 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, Aizu-Wakamatsu, Japan, 2014

[3] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, *Optimization of sparse matrix–vector multiplication on emerging multicore platforms*, SC, 2007 / J. Parallel and Distributed Computing

[4] SuiteSparse Matrix Collection, https://sparse.tamu.edu/, accessed October 2025.

[5] P. S. Pacheco, M. Malensek, *An Introduction to Parallel Programming*, 2nd Edition, Morgan Kaufmann, 2011.