Analysis of OpenMP Scheduling Strategies for Sparse Matrix-Vector Multiplication in CSR Format

Samuele Morandi
Student-ID: 245067
Email: samuele.morandi@studenti.unitn.it
Github Repository: https://github.com/SamuMorandi/PARCO-Computing-2026-245067

***Abstract*—The Sparse Matrix-Vector (SpMV) multiplication is a bandwidth-critical kernel where performance is often severely limited by irregular memory access patterns and inherent load imbalance. This paper presents a performance analysis of OpenMP scheduling strategies (static, dynamic, and guided) for a Compressed Sparse Row (CSR) based SpMV implementation. By benchmarking five diverse matrices from the SuiteSparse collection on a 64-core HPC node, we identify a critical trade-off between synchronization overhead and load balancing. Our results demonstrate that while static scheduling provides a robust baseline for smaller matrices, default dynamic strategies incur prohibitive overheads. We show that explicit chunk-size tuning is mandatory for irregular datasets: specifically, optimizing the chunk size improved the speedup on the largest matrix from a negligible 0.13x (slowdown) to 8.38x (with 32 threads), ultimately outperforming static scheduling. Finally, hardware counter analysis confirms that scalability is bounded by memory bandwidth saturation rather than compute capability, with LLC miss rates rising significantly beyond 32 threads.**

I. Introduction

Sparse Matrix-Vector multiplication (SpMV) is a fundamental computational problem in scientific computing and engineering. Its applications span diverse domains, as reflected by the matrices selected for this study from the SuiteSparse collection. For example, the *bmwcra_1* matrix originates from automotive structural analysis (FEM), *msdoor* from fluid dynamics simulations, and *ML_Geer* from graph analysis in machine learning [1]. As datasets continue to grow in scale, the acceleration of SpMV via parallel processing on multi-core architectures is critical for achieving timely and efficient analysis.

Despite its conceptual simplicity, achieving high performance for SpMV on parallel systems is notoriously difficult. This challenge stems from two primary characteristics of sparse data. First, the matrix sparsity requires specialized storage formats like Compressed Sparse Row (CSR) to avoid wasted computation and memory bandwidth. Second, and most critically, the distribution of non-zero elements (NNZ) is often highly irregular. This irregularity leads to two major performance bottlenecks: 1) Irregular Memory Access, which leads to high cache miss rates and memory bandwidth saturation by defeating hardware prefetchers [2], and 2) Load Imbalance, where threads assigned to dense rows perform significantly more work than threads assigned to sparse rows, causing pipeline stalls [4].

Extensive research has focused on optimizing SpMV. Seminal work by Williams [2] provided a comprehensive analysis, identifying memory bandwidth as the primary bottleneck, rather than computational limits. Subsequent research has confirmed that SpMV performance is highly dependent on both the matrix structure and the underlying hardware, advocating for "matrix-aware" or "architecture-aware" runtime specialization rather than a one-size-fits-all solution [3].

For shared-memory systems, OpenMP is the standard parallelization model. However, the default strategy of statically mapping rows to threads often fails to address the load imbalance inherent in the data. Research has shown that the choice of OpenMP scheduling policy has a considerable effect on SpMV performance [4], validating the investigation of scheduling as a primary optimization method.

While many studies focus on novel data formats, a gap exists in the empirical analysis of how OpenMP's built-in scheduling clauses (static, dynamic, guided) perform when applied to the standard CSR format across different types of matrix sparsity.

This paper addresses this gap by presenting a comparative performance analysis of these OpenMP scheduling strategies for a CSR-based SpMV kernel. We evaluate the trade-off between scheduling overhead and load balancing by benchmarking five matrices with diverse sparsity patterns on a 64-core HPC cluster node. The main contribution of this work is to provide clear empirical evidence to guide the selection of an optimal scheduling strategy based on matrix characteristics.

II. Methodology

A. Data structure and pre-processing

To optimize memory access and avoid wasted computation on zero-value elements, we utilized the Compressed Sparse Row (CSR) format. This format represents the matrix using three one-dimensional arrays: *values*, *cols*, and *rows_ptr*. However, the input .mtx files are provided in a Coordinate (COO) format. We therefore implemented a three-stage pre-processing pipeline to convert the data. This pipeline relies on a temporary structure defined as struct Node int row, int col, double value;.

1) **Parsing and Symmetry Handling**: The .mtx file is read line by line. Each non-zero element is read into a Node object. This object is then added to a *vector<Node>*. If the matrix is defined as "symmetric," a second Node

with swapped row and col indices is also added to the vector to explicitly represent the full matrix.

2) **Sorting**: The *vector<Node>* is sorted in-place using *sort*. The sorting predicate uses the **row as the primary key** and the **col as the secondary key**. This step is essential to group all elements of the same row contiguously, as required by CSR.
3) **Compression (CSR)**: We iterate over the sorted *vector<Node>*. For each Node, its *value* and *col* are appended to the final *values* and *cols* CSR vectors. The *rows_ptr* array, which stores the starting index of each row, is built simultaneously using an implicit prefix-sum by tracking changes in the row field during this iteration. The temporary *vector<Node>* is then discarded.

B. Parallelization Strategy (OpenMP)

The SpMV multiplication $y = A \cdot x$ was parallelized using the OpenMP shared-memory model. The chosen approach is a Row-wise Data Decomposition. The outer for loop, which iterates over the rows of matrix A, was parallelized using the *#pragma omp parallel for* directive. In this strategy, each thread in the OpenMP team executes the computation

$$y_i = \sum_j A_{i,j} \cdot x_j$$

for a distinct subset of rows i. This approach ensures there are no write-based race conditions on the result array (y), as each thread writes to different indices (each thread "owns" its assigned rows).

Algorithm 1 Parallel SpMV Kernel (OpenMP)

Require: $rows\_ptr$, $cols$, $values$ (CSR matrix A), x (vector)
Ensure: $result$ (vector y)
#pragma omp parallel for schedule(...)
for $r \leftarrow 0$ to $N\_ROWS - 1$ **do**
  $sum \leftarrow 0$
  for $idx \leftarrow rows\_ptr[r]$ to $rows\_ptr[r+1] - 1$ **do**
    $col \leftarrow cols[idx]$
    $val \leftarrow values[idx]$
    $sum \leftarrow sum + val \times x[col]$
  end for
  $result[r] \leftarrow sum$
end for

C. Scheduling for Load Balancing

The primary challenge of this parallel strategy is Load Imbalance: sparse matrices often have a highly variable number of NNZ per row. If a static schedule is used, a thread assigned to a "dense" row will work much longer than a thread assigned to a "sparse" row, leading to idle time and low efficiency. To address this problem, we analyzed the impact of several OpenMP scheduling clauses:

1) **Static**: *#pragma omp parallel for schedule(static)*. Divides the rows into similarly-sized chunks and assigns them to threads in a fixed manner before the loop begins. It has low overhead but is inefficient under load imbalance.
2) **Dynamic**: *#pragma omp parallel for schedule(dynamic, C)*. Threads request a chunk of C rows (or 1 if C is unspecified) from the runtime when they become idle. This balances the load but introduces higher scheduling overhead.
3) **Guided**: *#pragma omp parallel for schedule(guided, C)*. A compromise where the runtime assigns large chunks initially and dynamically reduces the chunk size as work proceeds, balancing the load while mitigating overhead.

III. Experimental setup

To ensure the reproducibility and rigor of our performance analysis, this section details the hardware environment, the software toolchain, the dataset used for benchmarking, and the precise metrics collected.

A. Hardware and Software Environment

All benchmarks were executed on a high-performance computing (HPC) cluster. The performance scaling was measured by submitting 15 different PBS scripts, one for each combination of thread count and scheduling type. 10 More PBS scripts were submitted for testing better performance with dynamic and guided scheduling with a chunk size of 100. This approach ensured that for each test (with 4, 8, 16, 32, or 64 threads), the OpenMP program was run on a dedicated node (select=1) provisioned with the corresponding number of CPU cores (ncpus=4, ncpus=8, etc.). Each PBS was requesting the same amount of memory: 25gb.

The software environment was standardized using GCC version 9.1 (module load gcc91) and all C++ source files were compiled with: `g++ -std=c++11 -O3 -march=native -fopenmp`. The -O3 flag enables aggressive optimization, -march=native tailors the binary to the node's specific CPU architecture, and -fopenmp enables the OpenMP parallel runtime.

B. Data set

The performance of SpMV is highly dependent on the sparsity pattern of the input matrix. Therefore, we selected five matrices with diverse characteristics from the SuiteSparse Matrix Collection [1]. The selected matrices are listed in Table I.

The input vector x was initialized with random double-precision values to prevent compiler optimizations from pre-calculating results and to simulate a realistic workload. Its values range from 1 to 10.

Table I summarizes the key properties of the dataset.

C. Benchmarking Procedure and Metrics

A rigorous benchmarking procedure was followed to gather reliable performance data.

1) **Execution Runs**: For each configuration (scheduling strategy and thread count), the application was launched as a **fresh process instance** 10 times for each matrix

TABLE I
CHARACTERISTICS OF TEST MATRICES [1]

Matrix	Rows	NNZ	Density	Domain
ML_Geer	1.5M	110M	4.89×10^{-5}	Machine learning
PFlow_742	0.7M	37M	6.73×10^{-5}	Fluid dynamics
bmwcra_1	0.14M	10M	4.81×10^{-4}	Structural problem
msdoor	0.41M	19M	1.11×10^{-4}	Structural problem
nlpkkt240	27.9M	760M	9.71×10^{-7}	Optimization

M = Millions. Values rounded for brevity.

(round-robin fashion). A single PBS script managed the execution of the fifty consecutive runs (10 iterations over 5 matrices).

Crucially, each process execution involves a full **loading and population of CSR data structures** immediately before to the timed region. Because the SpMV kernel executes right after this data population phase, the measurements reflect a **warm-cache scenario for smaller datasets** (ex., msdoor). This is due to the *write-allocation* policy: the initialization phase writes data into memory, pre-loading it into the Last Level Cache (LLC). Conversely, for larger datasets exceeding cache capacity (e.g., *nlpkkt240*), the initialization evicts earlier data, and the execution effectively simulates a streaming workload from main memory.

2) **Time Measurement**: The core SpMV computation time was measured within the C++ code using *clock_gettime(CLOCK_MONOTONIC)*. The timer starts immediately after the CSR construction and vector generation are completed. We report the 90th percentile of the real-time values from the 10 runs to ensure stable and reliable results, filtering out potential system noise. Also CPU-time values were measured but these were used only for comparisons.

   All the values were printed in seconds and later translated into milliseconds.
3) **Performance Metrics**: We evaluated performance using standard metrics: Real time (ms), Speedup and Efficiency.

   Speedup was calculated as $S_p = T_{seq}/T_p$, where T_{seq} is the 90th percentile time of the sequential (1-thread) version and T_p is the 90th percentile time with p threads. Efficiency was calculated as $E_p = S_p/p$, where p is the number of threads used during the execution.
4) **Hardware profiling**: To identify bottlenecks, we also used *perf stat* to capture hardware performance counters, specifically: *cycles*, *instructions*, *L1-dcache-load-misses*, and *LLC-load-misses*.

IV. RESULTS

This section presents the performance analysis of the OpenMP-based SpMV kernel. We analyze the trade-off between synchronization overhead and load balancing, and identify the hardware limits affecting scalability.

A. The Overhead of Default Scheduling

Table II summarizes the baseline performance. Initially, we compared the sequential runtime (T_{seq}) against the OpenMP schedules using their default chunk sizes.

The results highlight a critical weakness in the default *dynamic* and *guided* strategies. On the largest matrix (nlpkkt240, $T_{seq} = 1848$ms), the default *dynamic* schedule yielded catastrophic performance, with execution times soaring to $13,366$ms at 32 threads (a speedup of 0.13x). Similarly, the default *guided* schedule failed to scale. This confirms that for SpMV operations (which involve very little computation per row) the runtime overhead of fetching work one row at a time (default chunk size = 1) completely dominates the execution time, negating any benefit from parallelization.

In contrast, *static* scheduling, which has near-zero runtime overhead, provided consistent speedups immediately, reaching 6.5x speedup on nlpkkt240 at 32 threads (284ms).

TABLE II
PERFORMANCE SUMMARY (TIME (MS) & BEST SPEEDUP)

Matrix	T_{seq} (ms)	T_{best} (ms)	Speedup (Max)	Best Config. (Threads, Sched.)
ML_Geer	335	31.5	10.6x	32, Dynamic-100
PFlow_742	104	10.7	9.77x	32, Dynamic-100
bmwcra_1	31.9	3.4	9.25x	16, Dynamic-100
msdoor	64.4	5.63	11.43x	8, Guided-100
nlpkkt240	1848.6	220	8.38x	32, Dynamic-100

All times represent the 90th percentile of 10 runs.

B. Impact of Chunk Size Optimization

To mitigate the synchronization overhead, we tuned the *dynamic* and *guided* schedules by setting a fixed chunk size of 100. This value was selected as a representative trade-off point: it is sufficiently large to amortize the atomic fetch overhead of the OpenMP runtime, yet small enough to maintain granular load balancing across the 64 cores.

The impact was substantial, particularly for the largest and most irregular matrix. As shown in Figure 1, the performance of *dynamic* scheduling on nlpkkt240 at 32 threads recovered from the disastrous $13,366$ms (default) to 220ms (chunk 100). Crucially, this optimized dynamic configuration outperformed the best *static* result (284ms) at the same thread count (32). This validates the hypothesis that dynamic scheduling is superior for irregular matrices (Load Balancing), but only if the overhead is amortized via chunking.

C. Scalability Limits

Despite the optimizations, the speedup did not scale linearly to 64 cores. Across all matrices, performance peaked between 16 and 32 threads before degrading. For example, on nlpkkt240 using *dynamic-100*, the execution time increased from 220ms (32 threads) to 8971.9ms (64 threads).

This behavior is characteristic of a **memory-bound** application. SpMV has a low arithmetic intensity (2 floating point operations per non-zero element). Once the memory

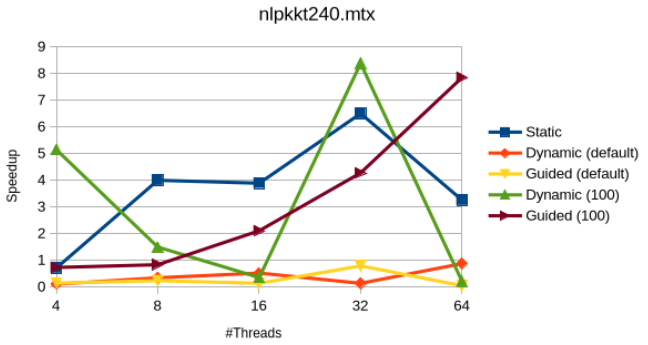

Fig. 1. Comparison of Scheduling Strategies on nlpkkt240.

bandwidth of the node is saturated, adding more threads leads to resource contention rather than increased throughput. Furthermore, the degradation at 64 threads suggests the impact of NUMA (Non-Uniform Memory Access) effects, where the cost of maintaining cache coherence and accessing remote memory banks across sockets outweighs the parallel compute capacity. To pinpoint the cause, we analyzed the hardware performance counters using *perf*. The data reveals a severe memory bandwidth bottleneck. Specifically, for the `nlpkkt240` matrix at peak performance (32 threads), we measured an average **LLC-load-miss rate of 39.9%**.

Scaling to 64 threads, this bottleneck exacerbated significantly. The LLC miss rate increased to **43.5%**, and we observed a massive spike in execution time, indicating that the coherence traffic and thread contention completely saturated the bus. This confirms that the SpMV kernel on this architecture is memory-bound, and adding cores beyond the saturation point yields negative returns.

D. Analysis on performance anomalies

Our experimental results highlight two significant performance anomalies that diverge from ideal scaling behavior.

First, as shown in Figure 2, we observed super-linear speedup for the msdoor matrix using the Dynamic-8 strategy, which achieved a speedup of 9.85x on 8 threads (Efficiency $\approx$ 1.23). This phenomenon is attributable to the cache aggregation effect: by partitioning the matrix among multiple cores, the working set assigned to each thread becomes small enough to fit entirely within the fast L1/L2 caches. As shown in Table III, this hypothesis is corroborated by hardware counter analysis: increasing the thread count from 4 to 8 reduced the LLC Miss Rate from 41.4% to 36.0%. This reduction confirms that the parallel execution benefits from a drastic reduction in memory latency compared to the sequential execution.

Conversely, other configurations exhibited significant performance instability. For instance, the nlpkkt240 matrix at 16 threads showed extreme variance: while the best-case execution time (217ms) demonstrated near-linear scaling, the 90th percentile time degraded to over 5000ms. Crucially, Table III highlights that the IPC remained high ($\approx$ 2.03) even during these slow runs. This paradox indicates that the CPU was not stalled on memory, but rather performing "busy-waiting" due to **scheduling contention**. The overhead of managing the shared work queue creates a bottleneck where threads actively burn CPU cycles waiting for work.

TABLE III
ANALYSIS OF PERFORMANCE ANOMALIES

Matrix and config.	**Key Metrics**	**Identified cause**
msdoor (Dynamic-8, chunk=100)	Speedup: 9.85x Eff: $\approx$ 1.23 LLC Miss: 36%	Cache Aggregation (Working set fits in L1/L2 cache)
nlpkkt240 (Dynamic-16, chunk=100)	Best: 217ms Worst: >5000ms IPC: $\approx$ 2.03	Scheduling Contention (Busy-waiting on shared queue)

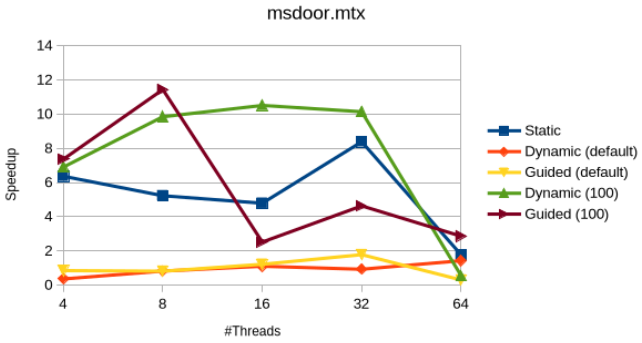

Fig. 2. Comparison of Scheduling Strategies on msdoor.

V. CONCLUSION

This study evaluated the performance of OpenMP scheduling on the SpMV kernel. Our results disprove the naive assumption that *static* scheduling is always inferior due to load imbalance. On the contrary, we demonstrated that *static* is often the most robust default due to its minimal overhead.

Finally, hardware profiling revealed the hierarchical nature of the performance limits. While the Memory Bandwidth (evidenced by high LLC miss rates) limits scalability up to the physical core count, exceeding this limit leads to severe performance degradation due to resource contention and thread oversubscription.

A. Future Works

Future work should focus on addressing the primary bottleneck identified in this study: memory bandwidth saturation. While our optimized static scheduler solved the overhead problem, it is still limited by high LLC cache miss rates from the irregular memory access of the CSR format. A possible better solution would be to explore alternative data formats, such as Compressed Sparse Blocks (CSB), which are designed to improve cache locality. A comparative study between CSR and CSB on this hardware would be a valuable next step [5].

REFERENCES

[1] T. A. Davis and Y. Hu, "The University of Florida SuiteSparse Matrix Collection," ACM Transactions on Mathematical Software (TOMS), vol. 38, no. 1, pp. 1-25, 2011. [Online]. Available: https://sparse.tamu.edu/

[2] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," Parallel Computing, vol. 35, no. 3, pp. 178-194, 2009.

[3] A. Elafrou, G. Goumas, and N. Koziris, "Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 10, pp. 2967-2980, 2017.

[4] T. Iwata, H. Nakashima, T. Fukaya, and K. Kise, "Performance optimization of SpMV using CRS format by considering OpenMP scheduling on CPUs and MIC," in 2014 13th International Conference on Control, Automation, Robotics and Vision (ICARCV), 2014, pp. 1515-1520.

[5] 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. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2009, pp. 233–244.