Parallel Project Report: Performance Analysis of Parallel Sparse Matrix Multiplication

Giovanni Todesco (ID: 244645)
Email: giovanni.todesco@studenti.unitn.it
Course: Introduction to Parallel Computing (2025-2026)
GIT: https://github.com/T00dd/PARCO-Computing-2026-244645.git

***Abstract*—This work presents a performance analysis of OpenMP-parallelized SpMV algorithm (Sparse Matrix-Vector Multiplication). The SpMV operation (y=Ax) remains a critical bottleneck in numerous scientific computing domains. We use the Compressed Sparse Row (CSR) format on a modern Non Uniform Memory Access (NUMA) architecture. The primary objective is to evaluate the impact of OpenMP parallelization (with different schedule policies and chunk size) on execution time and correlate this findings with hardware performance counters. We conduct a strong scaling benchmark with five different matrices using Perf tool to measure L1 data cache misses and Last-Level Cache (LLC) misses to quantify memory-bound inefficiencies.**

The results confirm that SpMV performance is heavily memory bandwidth-bounded limiting speedup after 16 threads. We also observe a performance drop at 64 threads due to scheduling overhead and NUMA architecture. Furthermore, we found that while dynamic scheduling helps with irregular matrices, static scheduling is faster for structured ones. Profiling confirms that cache misses depend on the matrix structure, proving that memory speed is the main bottleneck.

***Index Terms*—OpenMP, SpMV, CSR, NUMA, bandwidth-bounded, schedule**

I. Introduction

Problem. SpMV despite its simplicity has two different problems that lead to weak performances: low arithmetic intensity, making the algorithm memory bandwidth-bounded (as we can see from the roofline model explained by Samuel Williams [1]) and indirect memory access (due to CSR format), which leads to high cache misses.

Objectives. (i) Design and implement a parallel SpMV; (ii) quantify strong scaling; (iii) evaluate scheduling policies and chunk sizes; (iv) analyze bottlenecks; (v) document a reproducible workflow.

II. State of the art

The optimization of SpMV is a very common field in High Performance Computing (HCP).

CPU and storage format. The CSR format is a standard due to its storage efficiency but it suffers from irregular memory access patterns which degrade spatial locality [2]. Different researches proposed others storage formats, like Blocked-CSR or SELL-C-σ, to maximize vectorization and cache reuse. However, these type of formats often introduce lots of preprocessing overhead, making standard CSR the more efficient choice when the matrix structure is unknown or changes dynamically [3].

CPU and storage format. On shared-memory multicore architectures, the primary challenge for SpMV is load imbalance because the majority of the rows has few non-zero values and few rows has lots of non-zero values. For this reason OpenMP static scheduling often fails to distribute work evenly across threads for such matrices. While dynamic or guided scheduling can reduce this imbalance, but they can introduces some runtime overhead that can decrease the performance if not perfectly tuned [4].

III. Contribution and Methodology

A. Contribution

Scheduling Analysis: We make a comparison of OpenMP static, guuided, and dynamic schedules with different types of chunk sizes in order to identify the "sweet spot" between load balancing and syncronization overhead for matrices with different grade of sparsity

Cache Efficiency Correlation: We utilize hardware performance counters (via the perf tool on linux) to correlate L1 and Last Level Cache miss rates with execution time. We demonstrate that matrices with high sparsity or irregular non-zero value patterns degrade spatial locality due to irregular access to the input vector, increasing the number of cache misses.

NUMA Constraints: We make some observations about thread pinning in a multi socket environment, addressing the specific challenges of first-touch memory placement in parallel implementations.

B. Methodology

After we download the matrix (MxN) from Matrix Market, we convert the format from (COO) to Compressed Sparse Row format (CSR), to store more efficiently the non-zero elements. With the CSR format a matrix is compressed in three arrays: val (stores the non zero values), col_ind (stores the column indices corresponding to val) and row_ptr (stores the starting index of each row in val). Than we create the vector with random values using the $rand$ function.

We firstly execute the SpMV algorithm ($y = Ax$) sequentially than we parallelize it with OpenMP. We use the `default(none)` clause to explicitly specify the

shared and private variables with `shared(matrix, x, y, M)` and `private(i, j)` clauses. We need also the `schedule(runtime)` clause to dynamically change different schedules type and chunk sizes.

Algorithm 1 OpenMP Parallel SpMV Kernel

Require: row_ptr, col_ind, val, x
Ensure: $y \leftarrow Ax$

```
#pragma omp parallel for default(none) shared(matrix, x, y, M) private(i, j) schedule(runtime)
for i = 0 to M − 1 do
    sum ← 0
    for j = row_ptr[i] to row_ptr[i + 1] − 1 do
        sum ← sum + val[j] × x[col_ind[j]]
    end for
    y[i] ← sum
end for
```

Then we measure the execution time of only the multiplication between the matrix and the random generated vector.

IV. Experiments and System Description

A. Computing system

The experiments were conducted on the University of Trento's cluster, in a High-Performance Computing (HPC) node, designed with a Non-Uniform Memory Access (NUMA) architecture. The system specifications are:

- **CPU**: 64 logical cores distributed across multiple sockets
- **Memory**: 32GB of RAM
- **Operating System**: CentOS Linux release 7.6.1810

B. Software configuration

The code was compiled using GCC 9.1 with the `-O3` optimization flag and `-fopenmp` flag for OpenMP directives. To analyze hardware behavior, we used the Linux perf tool.

Thread Affinity and Policy. To make the performance more stable and reducing operating system noise due to a NUMA architecture, we use the following OpenMP environment variables:

- `OMP_PLACES=cores` that locks a thread to a specific physical core
- `OMP_PROC_BIND=close` that keeps worker threads physically close to the master thread
- `OMP_WAIT_POLICY=active` that keeps waiting threads active to reduce wake-up latency

On NUMA systems, memory access time depends on the memory location relative to the processor. If threads migrate between sockets, they may need to fetch data from remote memory banks, increasing the latency.

C. Dataset

We select five sparse matrices from the SuiteSparse Matrix Collection with different sizes and sparsity, allowing us to stress our algorithm. We selected only matrices with Real values and Unsymmetric structures: $Twotone$, $Transport$, $Cage14$, $Torso1$ and $Memchip$ as shown in Table I.

D. Benchmarking

To get accurate results, we used two different ways to measure performance:

Execution Time (Warm Cache): To measure how fast the code runs, the program repeats the multiplication 10 times in the C code. When we extract the data we use only the 9th iteration. This ensures that the data is already loaded into the CPU cache ("warm up"), giving us a stable result and ignoring the initial slowness of the first run.

Hardware Profiling (Cold Cache): To count cache misses using the perf tool, we run the entire program 5 separate times using the job scheduler. We take the data from the 4th run. This allows us to see how the program behaves when it starts from scratch ("cold"), but we skip the very first runs to avoid random noise from the operating system.

E. Design of Experiments

We designed our experiments to answer three main questions:

- **Speedup test**: We check how much faster the code runs when we add more threads (from 1 to 64)
- **Finding the best settings**: We test every combination of OpenMP schedules (static, dinamic, guided) and chunk sizes. The goal is to find which settings give the best performance for unbalanced matrices where the work is uneven
- **Memory bottleneck check**: We measure cache misses, if this number is high, the CPU is wasting time waiting for data from the slower memory (RAM). This helps us confirm the "Memory Wall" bottleneck

$$\text{L1 Miss Rate (\%)} = \frac{\text{L1 Load Misses}}{\text{L1 Loads}} \times 100 \tag{1}$$

$$\text{LLC Miss Rate (\%)} = \frac{\text{LLC Misses}}{\text{LLC Loads}} \times 100 \tag{2}$$

TABLE I
DATASET

Matrix	Rows	Columns	Non-zero	Sparsity(%)
twotone	120,750	120,750	1,206,265	0.0083%
Transport	1,602,111	1,602,111	23,487,281	0.00092%
cage14	1,505,785	1,505,785	27,130,349	0.0012%
torso1	116,158	116,158	8,516,500	0.063%
memchip	2,707,524	2,707,524	13,343,948	0.00018%

V. Results and discussion

A. Peak performance and speedup

As shown in Fig. 1, all the matrices show a clear performance improvement as we increase the number of threads (strong scaling). We can see from TABLE II that the parallelized SpMV achieves a maximum speedup between $6.94x$ and $9.28x$, that is far from a linear speedup. Most matrices get the best results with 16 threads, only torso1 continues to get faster with 32 threads. The fact that the performance gradually stops improving at 16 threads is a clear indicator that

the algorithm is memory bandwith-bounded. This because the RAM can not send data fast enough to keep the cores busy.

TABLE II
SPEEDUP

Matrix	Speedup	Threads	Schedule	Chunck	Time (ms)
twotone	9.28x	16	static	1000	0.15
Transport	7.64x	16	guided	1000	3.90
cage14	8.19x	16	dynamic	10000	4.54
torso1	9.06x	32	static	100	1.33
memchip	6.94x	16	static	10000	3.37

B. NUMA observations

We can see from the graphs that pushing the system with more than 16 threads can cause a slow down of performance that create a "U-shape" curve. The reason for this is the NUMA architecture. Our code initializes the matrix sequentially, so the operating system places all the data and the threads in socket 0, but when we use more than 16 threads the cores of socket 1 are not enough and we use more than one socket. Threads on socket 1 must read data from socket 0 and this traffic floods the connection between CPUs creating a bottleneck [5].
We can also see that the "U-curv" graphs are more visible with guided and dynamic types of schedule. The reason is because with higher number of threads (64) there will be more scheduling overhead due to the threads constant asking for the next chunk of work [4].

C. Schedule Analysis and Matrix Properties

cage14. This matrix is unstructure and is very irregular (some rows have many non-zero elements, others have very few). As predicted from theory the dynamic scheduling is the best (at 16 threads) because there is a load imbalance that is fixed by handing out work on demand. The large chunk size (10000) is important because it reduce the time of asking for new work for each thread, finding the sweet spot between load balancing and synchronization cost.

memchip, torso1, twotone. With these matrices, even if they are very different (memchip is extremely sparse and torso1 has a more defined structure), the most efficient schedule is static. The workload is either balanced enough naturally or the computation per row is so small that the overhead of managing a dynamic list of tasks is too expensive.

Transport. This matrix is a physical simulation and it has a mixed structure: some parts of the matrix are "heavy" (many calculations), while others are "light". The best schedule in this case is guided with a chunk size of 1000. This schedule is a compromise between static and dynamic because at the beginning it assigns large chunks of work like static, but when it gets closer to finish it assigns smaller chunks. This helps to balance the load among threads.

D. Profiling Analysis: L1 and LLC Cache Behavior

To understand exactly why the speed stops increasing after 16 threads, we looked at the hardware counters using the perf tool. Figures 2 and 3 show the "Miss Rates" for the L1 Cache (small, fast memory) and the LLC (large, slower memory).

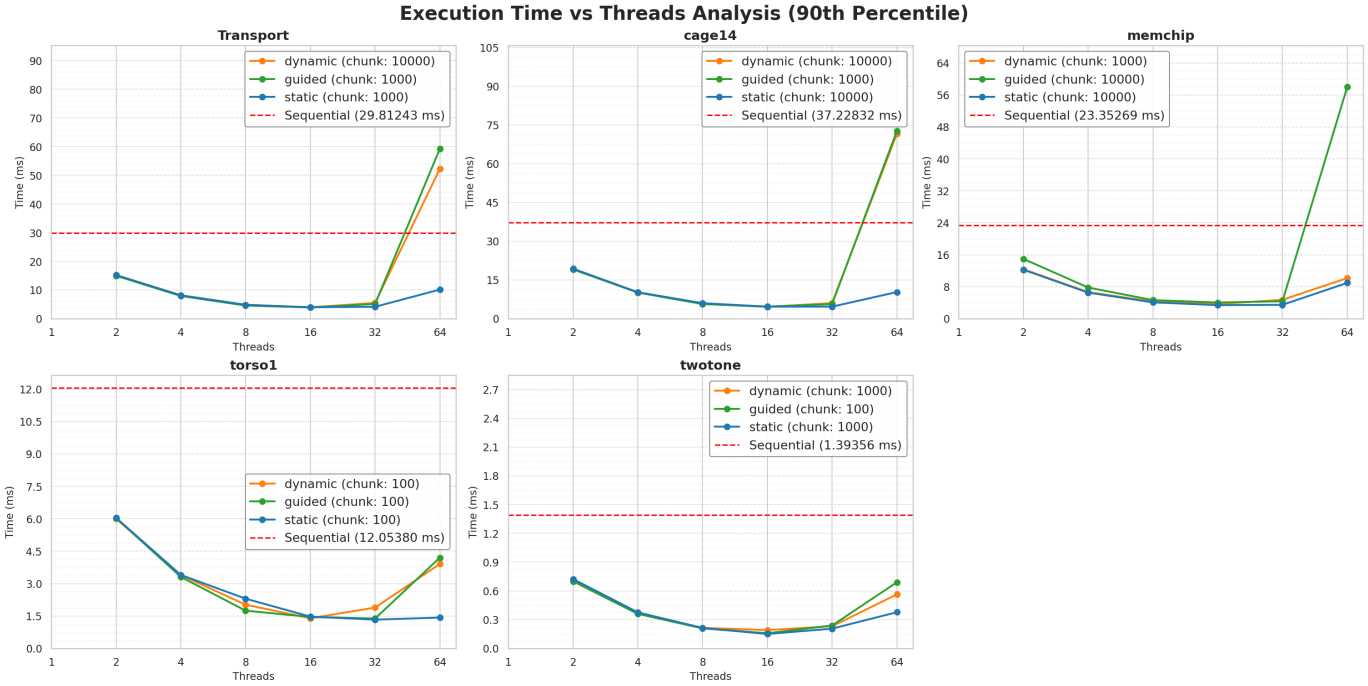

Fig. 1. Strong scaling

L1 Cache: Small Jumps in Memory

The L1 Miss Rate tells us if the CPU struggles to find data in its fastest memory cache. The error rates are very low (between 0.20% and 0.40%). This means the code generally reads data smoothly. Torso1 is the worst ($\sim 0.40\%$) and even though the rate is low, it is double that of the others. This means the column indices (x[col[j]]) jump around a bit more locally, making it harder for the CPU to guess the next value. memchip is the best ($\sim 0.20\%$) because this matrix is very organized locally, so the CPU rarely misses in the L1 cache.

LLC Cache: The Impact of Matrix Structure

The LLC (Last Level Cache) Miss Rate is the most important graph. The huge differences between the lines explain why some matrices are fast and others are slow [6].

torso1 ($\sim 81\%$ Miss Rate): Even if this matrix is not the biggest it has an enormous amount of cache misses because of its irregularity. The access pattern jumps around randomly, the CPU constantly has to throw out old data to make room for new data. 80% of the time, the CPU has to wait for the slow RAM.

cage14 ($\sim 50\%$ Miss Rate): This is a DNA graph. It is also irregular and it doesn't have a geometric shape. Because of this the cache only works half the time.

Transport ($\sim 30\%$ Miss Rate): This matrix even if it has the same size of cage14, it has less LLC cache misses. It is very structured and this helps the cache to keep the right data ready, so we only miss 30% of the time.

memchip ($\sim 26\%$ Miss Rate): This simulates a computer chip. Circuits have groups of components connected together. This "grouping" helps the cache work efficiently, similar to Transport.

twotone ($\sim 13\%$ Miss Rate): This matrix is small and for this reason the entire input vector x fits inside the CPU cache. Once the data is loaded, it stays there. The CPU almost never has to go to the slow RAM, which is why twotone is the fastest matrix ($0.15ms$).

Proving the "Memory Wall":

The most important thing to notice in the graphs is that the lines are flat. The miss rate (%) is the same whether we use 1 thread or 64 threads. Since the percentage of errors is constant, adding more threads just creates more traffic. At 1 thread, there is one stream of requests to memory. At 64 threads, there are 64 streams of requests all at once. By 16 threads, the road to memory (the bandwidth) is completely full. Adding more threads doesn't make the calculation faster; it just creates a longer line of requests waiting to get to the RAM. This is why the speed improvement stops.

VI. Conclusions

A. Summary of key findings

This work presented a performance analysis of OpenMP-parallelized SpMV algorithm on a NUMA architecture. Our experiments lead to three primary conclusions:

The memory wall: the algorithm shows strong scaling up to 16 threads, with speedups between 6.9x and 9.3x. With more than 16 threads performance saturates due to memory bandwidth exhaustion.

NUMA and schedule bottleneck: executing with 64 threads consistently resulted in bad performance. This is caused by the sequential initialization of the matrix, which triggers the OS First-Touch policy, and scheduling overhead.

Structural dependency: hardware profiling revealed that cache efficiency depends from the structures of the matrix. Large, irregular matrices like torso1 sustain high LLC miss rates, while smaller ones like twotone benefit from fitting within the Working Set Size.

B. Impact and future work

Our findings show that there isn't a "one-size-fits-all" approach to SpMV. Because there is no overhead, static scheduling works best for regular or extremely sparse matrices (mem-

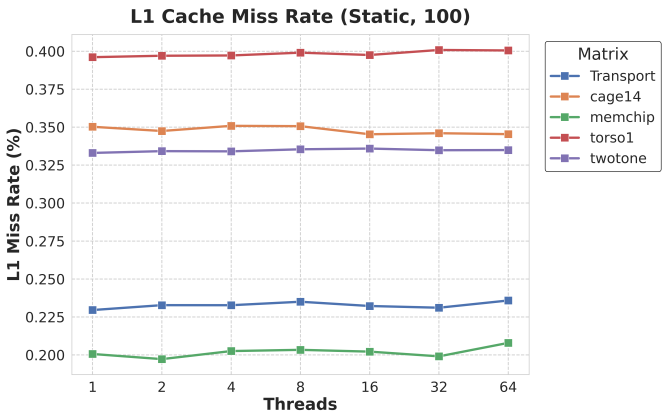

Fig. 2.

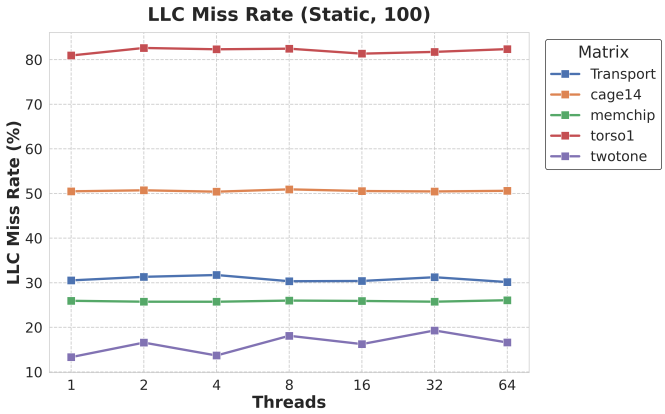

Fig. 3.

chip), while dynamic scheduling is necessary for irregular graphs (cage14) in order to balance the load. Future research will concentrate on using concurrent first-touch initialization to solve the NUMA bottleneck. This could eliminate the performance reduction seen at high concurrency by ensuring appropriate memory distribution across sockets. In order to further optimize the CPU's computation phase, we also intend to investigate SIMD vectorization.

REFERENCES

[1] Samuel Williams, Andrew Waterman, and David Patterson. 2009. "Roofline: an insightful visual performance model for multicore architectures." Commun. ACM 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[2] J. L. Greathouse and M. Daga, "Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format," SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 2014, pp. 769-780, https://doi.org/10.1109/SC.2014.68

[3] Kreutzer, Moritz, Hager, Georg, Wellein, Gerhard, Fehske, Holger, Bishop, Alan R., "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units" SIAM Journal on Scientific Computing, 2014. https://epubs.siam.org/doi/abs/10.1137/130930352

[4] Bull, J.M.: Measuring synchronisation and scheduling overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, pp. 99–105 (1999) https://www.researchgate.net/profile/Mark-Bull/publication/2565983_Measuring_Synchronisation_and_Scheduling_Overheads_in_OpenMP/links/00b7d52cc3ac381d5b000000/Measuring-Synchronisation-and-Scheduling-Overheads-in-OpenMP.pdf

[5] Muddukrishna, Ananya, Jonsson, Peter A., Brorsson, Mats, Locality-Aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors, Scientific Programming, 2015, 981759, 16 pages, 2015. https://doi.org/10.1155/2015/981759

[6] N. Namashivavam, S. Mehta and P. -C. Yew, "Variable-Sized Blocks for Locality-Aware SpMV," 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Seoul, Korea (South), https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9370327&isnumber=9370301