Sparse Matrix-Vector Multiplication sequential vs OpenMP

Devid Troka 239293
Computer science, communications and electronics engineering
University of Trento, Trento, Italy
devid.troka@studenti.unitn.it
https://github.com/Devid663/PARCO-Computing-2026–239293-.git

***Abstract*—In the last few decades, the performance of computers and processing architectures has grown steadily, driven by increases in clock frequencies and the technological evolution of semiconductors. However, in recent years, this growth has shown a significant slowdown, due to the energy and thermal limits of modern processors. Consequently, the only direction to continue improving performance is through the effective exploitation of parallelism, using multi-core architectures and optimizing programs and algorithms. The purpose of this paper is to analyze the performance of executing a multiplication operation between a sparse matrix and a random vector, considered a bottleneck, heavily limited by memory bandwidth, for modern applications and scientific computing in general. This makes it an ideal case study for evaluating the potential and limitations of parallel implementations. The execution of this operation will be analyzed first in a sequential manner and then with parallelism techniques, such as the OpenMP API, to implement an application that performs the operation using multiple threads.**

***Index Terms*—1. Introduction, 2. Methodology, 3. Experiments, 4. Results, 5. Conclusions**

I. Introduction

A **Sparse Matrix Multiplication** (**SpMV**) is a matrix-vector operation, where the matrix operands are sparse (typically more than 50% of the entries are 0). A sparse matrix is a matrix whose number of nonzero elements is much smaller than the total number of elements. It is a fundamental operation that can have a significant impact on the performance of applications and algorithms in general. SpMV is widely used in scientific algorithms, image processing techniques, as well as machine learning and graph analysis. It is therefore an ideal case study. The problem with this sparse matrix is that storing it in a dense format is not sufficient. Effective implementations have been scarce over time, which is why various formats and techniques for storing the matrix have been developed to make this calculation more efficient, although they are very complicated. In fact, the development of parallel codes is a significant challenge. An efficient parallel implementation of a serial program may not be achieved by finding efficient parallelizations of each of its steps. Instead, the best parallelization could be achieved by designing a completely new algorithm.

II. Methodology

A. Problem definition

The Sparse Matrix-vector Multiplication (SpMV) operation computes $y = A \cdot x$, where A is a sparse matrix with nnz non-zero elements. This operation is clearly **memory-bound** and the performance of SpMV **depends on the storage format** used, which should be suitable for the sparsity structure of a given matrix.

The most common data structure used to store a sparse matrix for SpMV calculations is the Compressed Sparse Row (CSR) format [1]. Essentially, each row of matrix A is stacked one after another in a dense array val, along with a corresponding integer array ind that stores the column indices of each stored element. A third integer array, ptr, keeps track of where each row begins in val, ind. In a popular variant, CSR maintains a floating-point array $val[nnz]$ and two integer arrays, $col\_ind[nnz]$ and $row\_ptr[n]$, to store the matrix $A = (aij)$.

Listing 1
CSR SPMV (A) THEORICAL CODE

```
for (int i = 0; i < n; i++) {
    y[i] = 0.0;
    for(int k=row_ptr[i]; k<row_ptr[i+1]; k++){
        y[i] = y[i]+val[k]*x[col_ind[k]];
    }
}
```

The row_ptr array stores the index of each row in val. This means that if $val[k]$ stores the element of the matrix aij, then $row_ptr[i] \leq k < row\_ptr[i+1]$. The col_ind array stores the column indices of the elements in the val array. This means that if the element of the matrix $val[k]$ is at j, then $col\_ind[k] = j$ [2].

B. Algorithms and data structures

This section describes how the algorithm is implemented, both serially and in parallel, highlighting the problems of

each approach from the start.

1) Starting with the sequential implementation, the main function, which actually performs the SpMV calculation, is implemented by $matrix\_vector\_sparse\_csr$.

Listing 2
CSR SPMV (A) CODE

```
for (size_t i = 0; i < csr->nrows; ++i) {
    res[i] = 0.0;
    size_t nz_start = csr->row_ptr[i];
    size_t nz_end = csr->row_ptr[i + 1];
    for(nz_id=nz_start; nz_id<nz_end; ++nz_id){
        size_t j = csr->col_ind[nz_id];
        double val = csr->val[nz_id];
        res[i] = res[i] + val * v[j];
    }
}
```

Function whose algorithm works very similarly to the one presented in point A, a first cycle iterates through the rows of the matrix, setting $y[i]$ (in the code $res[i]$) to zero and defining the start and end of nnz of the i-th row to find the range of non-zero elements, and a second cycle on nnz reads val a_ij ($val[k]$) reads $col\_ind[k] = j$, ultimately $res[i] = a\_ij * v[j]$.

$$y[i] = \sum_{k=row\_ptr[i]}^{row\_ptr[i+1]-1} val[k] \cdot x[col_ind[k]]$$

2) Moving on to parallel implementation, it's best to clarify what OpenMP is first. It's an API for parallelization on shared memory where all threads share the same memory (RAM), working on **different portions of data**. The simplest case of application (which is actually the one used in our case) is the parallelization of a for loop that iterates over independent elements. In fact, in the case of the matrix-vector product, each thread calculates a row of the CSR matrix independently.

Listing 3
CSR SPMV (A) CODE

```
#pragma omp parallel
for (size_t i = 0; i < csr->nrows; ++i) {
    double sum = 0.0;
    for(j=csr->row_ptr[i];j<csr->row_ptr[i+1];
    ++j)
        sum += csr->val[j]*v[csr->col_ind[j]];
    res[i] = sum;
}
```

In our case, the parallelizable part of SpMV is the row loop (the second one). This is because the first cycle can create race conditions, due to the fact that there could be multiple threads within the same line. The cycle on the lines is perfect because:

- Each iteration is written in a different index: $res[i]$
- No thread accesses or modifies the variables of another thread
- The accesses $val[]$ and $col\_ind[]$ are read-only.
- No race conditions

Instead, for the internal cycle (the one that does the product $row \cdot vector$), always update the same variable $res[i]$. If you parallelize it, multiple threads would update the same variable multiple times (**race condition**).

The OpenMP #pragma omp parallel for annotation applied to the row loop allows the rows to be distributed among the threads. This parallel calculation is performed by 3 functions:

- $matrix\_vector\_sparse\_csr\_omp\_static$
- $matrix\_vector\_sparse\_csr\_omp\_dynamic$
- $matrix\_vector\_sparse\_csr\_omp\_guided$

One for each scheduling type:

Static Scheduling (schedule(static))
The iterations are divided into blocks of uniform size before execution. It may be efficient when each iteration costs the same. It works well if each row has an even number of non-zeros, because the workload is similar among the rows.

Dynamic Scheduling (dynamic schedule)
Iterations are assigned as threads finish. It has more overhead, but it balances the load better. It is useful when some lines have many more non-zeros than others. A thread that receives a "heavy" line does not remain blocked while the others have finished. The "light" threads take on new iterations.

Guided scheduling (schedule(guided))
Similar to dynamic, but the initial blocks are large and become smaller over time, combine.It reduces overhead and improves balance, making it great for large and very irregular real matrices, where static would be terrible.

As with the sequential version, the main limitation of this algorithm is the irregular accesses to the vector $v[]$. If the vector v has poorly localized accesses (**very sparse** and **non-consecutive columns**), the operation is **memory-bound**, regardless of whether the algorithm is sequential or parallel, because it depends on the CSR format of the matrix. OpenMP **doesn't change the access to the matrix**, but only the way iterations are distributed among threads.

III. Experiments

High Performance Computing (HPC) clusters use powerful processors that work in parallel to process large multidimensional data sets and solve complex problems at extremely high speeds. Each cluster node operates independently with its own processor and memory, enabling efficient handling of complex workloads. For all experiments, a 16-core computing node with 16 GB RAM available was requested via pbs through the $short\_cpuQ$ queue. Accessible via SSH to the cluster (head node), via external software, such as MobaXterm, and a VPN. The code was entirely written in C language, divided into 3 files (main.c, utility.c, and utility.h). The Matrix Market library was used to read the matrices. For parallelization, the openMP API was used. All of this is then performed by a .pbs script on the node. The toolchain used is:

gcc -std=c99 -O3 -fopenmp main.c utility.c mmio.c -o spmv_exec

The project focuses on analyzing the execution of the operation in different ways, comparing the sequential version to the parallel one, with different types of scheduling. For experimentation, it is therefore necessary to measure various metrics, including: start time, end time (measured with the $omp\_get\_wtime()$ functions) and measuring cache misses. For the benchmarks, 10 runs were performed, preceded by 1 warm-up per execution and a result check, all tested with 1, 2, 4, 8, and 16 threads. From these metrics, you can obtain data such as: time, 90th percentile, speed up, and scalability.

IV. Results and Discussion

Before discussing the benchmark results, we need to discuss what we expect. In general, the performance of any application depends on the ratio of mathematical operations to bytes moved (arithmetic intensity.) If the arithmetic intensity is low compared to the system's capacity, the application is memory-bound. The SpMV in CSR has an extremely low arithmetic intensity: For each nnz, do the following:

- 1 load of val
- 1 load of col_ind
- 1 indirect loading of $v[j]$
- 1 write di $res[i]$
- for only 1 multiply-add.

The limit is memory. This explains why the speedup will never be linear.

A. Sequential

For the tests on the sequential version, 5 matrices with very different characteristics were used:

1) first medium-small matrix with moderate sparsity, test overhead
2) medium-large, fairly dense matrix
3) huge matrix with unbalanced rows
4) Moderate to low density
5) power law matrix (different pattern) high sparsity

The sequential version's performance essentially shows reliable behavior, proportional to the number of non-zero elements (nnz), for each matrix. The smaller matrices perform the operation in a few milliseconds, while the larger matrix (the 3.1Mx1M) takes much longer.

TABLE I
Sequential average times

	Average times		
matrix	version	threads	avg time
matrix1.mtx	seq	1	0.0897058
matrix2.mtx	seq	1	2.46492862
matrix3.mtx	seq	1	22.06580182
matrix4.mtx	seq	1	0.18605654
matrix5.mtx	seq	1	0.40386242

This confirms that the SpMV operation in CSR format is memory-bound: most of the execution time is determined by memory access. To effectively show how much bandwidth

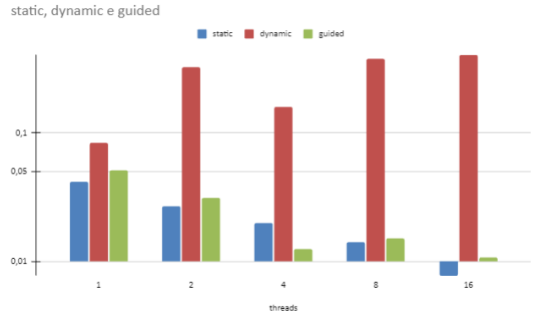

Fig. 1. All 5 matrices avg. execution time

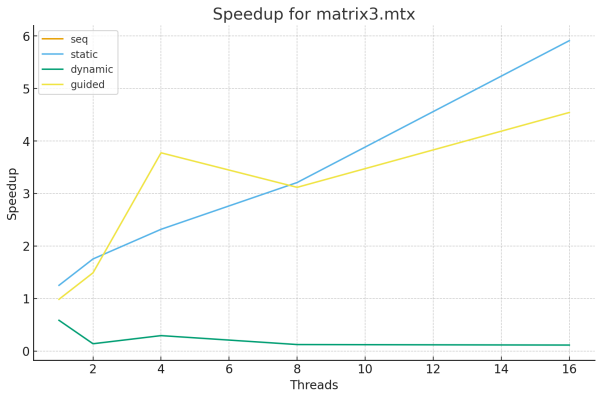

Fig. 2. Biggest matrix (n 3) speedup

the interference-free SpMV "consumes," it was calculated on matrix 1, showing that it only reaches:

$$BW = \frac{20b \cdot nnz}{time} \approx 0.0136 GB/s$$

definitely much lower than the node's capacity. This demonstrates the memory-bound nature of the SpMV operation and shows how, for small matrices, it cannot effectively utilize the available bandwidth.

B. Parallel performance and speedup

The benchmarks for the parallel version were run on the same 5 matrices, after 1 warm-up run, to heat up the cache. From the start, you can see that static is the fastest of the three versions, with zero overhead. The dynamic version, on the other hand, is terrible, especially for 8 and 16 threads, due to memory bandwidth saturation. Finally, Guided is very balanced.

A significant improvement is observed for all matrices when switching from 1 to 4 threads. However, scalability tends to reach saturation beyond 8 threads, with some cases where the time increases again at 16 threads.This behavior is consistent with the memory-bound nature of the operation: increasing the number of threads does not increase the available memory bandwidth, and the system quickly reaches the limit imposed by the DRAM. In the case of the larger matrix, there is a speedup of about 3x (in the static case), this is because it generally varies depending on the structure of the matrix.

So in general OpenMP helps to improve time execution only until a certain thread number. The figure helps to understand that the parallel versions are different:

- Static was the fastest, the load per line is sufficiently uniform, so the static partitioning minimizes overhead and makes the best use of the cache.
- Dynamic is the worst, with much longer times. This is caused by the high overhead assigned by the dynamic strategy.
- Guided is the most balanced, even with large matrices and irregular patterns. However, in most cases, it is slightly slower than static.

For some, 10 times were collected and the 90th percentile was calculated to indicate good and reliable measurements. For example, for the static version of matrix 1, we have:

TABLE II
90 PERCENTILE

	Matrix 1 static		
matrix	version	threads	avg time
matrix1.mtx	static	1	0.0728192
matrix2.mtx	static	2	0.0359896
matrix2.mtx	static	4	0.0210581
matrix2.mtx	static	8	0.0202317
matrix2.mtx	static	16	0.024488199

C. Influence of Matrix Structure

The matrix structure significantly influences performance:

- Small matrices: OpenMP overhead limits scalability;
- Large matrices: parallelism is more effective and speedup increases more;
- Distribution of nnz per row: where the distribution is uniform, static dominates; where some rows are much denser, guided achieves slightly better performance, while dynamic worsens.
- Irregular accesses to the v vector: generate more cache misses, worsening the performance of parallel versions.

D. Cache Behavoir

With the valgrind tool, it was possible to analyze the amount of cache misses in 2 cases:
1. Matrix 1, very small and well-structured:

- l1 miss rate: 0.29%
- D1 miss rate: 3.1%
- LLD miss rate: 2.1%
- Total misses: 4,844%

That's a good result.
2. Matrix 3, the largest, millions of miss:

- D refs: 3,742,541,453
- D1 misses: 70,320,252
- LLd misses: 52,877,140

The analysis of cache behavior reveals significant differences between small and large matrices. This behavior is due to indirect accesses to the vector $v[col\_ind[k]]$, which for very large matrices generate a very high number of cache misses and make it impossible to maintain significant locality. This is the main factor that prevents SpMV from scaling beyond 4–8 threads.

E. Bottleneck

As is now clear from the entire discussion of this paper and the results shown in the previous section, it is clear to identify how the bandwidth is the main bottleneck of the SpMV operation. The combined results show that this version, used with a CSR matrix, is largely limited by memory.

As noted from the beginning, CSR implementation requires indirect access to the vector $v[col\_ind[k]]$, which therefore does not allow for effective reuse of the vector elements in the cache.

The measurements clearly show that for small matrices, the number of cache misses is low, but for large matrices, the number is extremely high. This results in increased DRAM usage and a significant increase in execution time. Following our measures, there is a low utilization of available cores. In general, combining all the results, it results in maximum access to DRAM, indirect accesses prevent spatial locality, DRAM is the limiting factor.

V. CONCLUSIONS

Finally, this paper allowed for the analysis of the implementation of the SpMV operation in different modes, in a sequential version and through parallelization strategies using OpenMp. It was immediately possible to understand how the limit of this operation is the memory bandwidth and its use, regardless of the implementation, sequential or parallel. SpMV is a memory-bound operation, which means that the time it takes to complete it depends on memory traffic. This results in a scalability limit. In parallel execution, the static version works best, guided can be a good alternative, but it is less effective, for better calculation management, while the dynamic one performs worse, due to an increase in overhead.

These results allow us to understand that **SpMV has a low arithmetic intensity** and falls within the **memory-bound region** of the **Roofline model**.

Future implementations and improvements may involve using the BSR storage format, rather than using the NVIDIA cuSPARSE libraries, which provide SpMV implementations for CPUs and GPUs, or using the OpenACC API to verify how different implementations can optimize and help improve the performance of this operation.

VI. References

References

[1] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC '07). Association for Computing Machinery, New York, NY, USA, Article 38, 1–12. https://doi.org/10.1145/1362622.1362674

[2] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (SPAA '09). Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/1583991.1584053

[3] Devid Troka. Sparse Matrix-Vector Multiplication sequential vs OpenMP

[4] Performance portability of sparse matrix–vector multiplication implemented using OpenMP, OpenACC and SYCL Kinga Stec, Przemysław Stpiczyński Maria Curie-Skłodowska University, Institute of Computer Science, ul. Akademicka 9, 20-033, Lublin Poland

[5] Flavio Vella (2025). Introduction to parallel computing. University of Trento. Lecture 9.

[6] Flavio Vella (2025). Introduction to parallel computing. University of Trento. Lecture 10.

[7] Flavio Vella (2025). Introduction to parallel computing. University of Trento. Lecture 9.