Performance evaluation of OpenMP parallel implementation for Sparse-Matrix Vector Product

Andrea Boarini
MAT. 244129
https://github.com/AndreaBoarini/PARCO-Computing-2026-244129
andrea.boarini@studenti.unitn.it

***Abstract*—The main goal of this research is to determine the impact of parallelization in terms of execution time and cache addresses on the Sparse-Matrix Vector product, a bandwidth-bound kernel extensively adopted in scientific computing and machine learning applications. For this purpose, we tested the results of the algorithm's implementation exploiting the OpenMP library and compared them with the associated results provided by its baseline sequential version. Specifically, we put focus on showing the effect of parameter tuning on time performances and memory access efficiency, further investigating the advantages and trade-offs introduced by the adoption of different OMP parameters configurations. Furthermore by adopting thread pinning to augment cache locality and reducing memory latency, together with identifying the optimal combination, we've been able to reach an up to 7.59x faster execution compared to the baseline sequential results.**

I. Introduction

A. Motivation and real world challenges

In the last decade, the rise of more challenging and complex problems in scientific and engineering realms, rapidly forced computer architectures to adopt an approach that aims to maximise computational power within the limits of the available resources, wisely partitioned and managed. The standard technique to face these type of challenges is parallelization, which means, dividing the workload among different processing units so as to achieve concurrency and, in optimal conditions, a faster task execution [1] [2] [3].

B. Sparse matrix

In numerical analysis and scientific computing, a sparse matrix is a matrix in which most of the elements are zero [4]. Sparse matrices are found pretty much anywhere in the realm of physics simulations, mathematics and engineering when dealing with partial differential equation [5], also covering a fundamental role in Machine Learning and Deep Learning [6].

C. CSR format for sparse matrix storage

When storing and manipulating a sparse matrix it is often necessary to rely on specific algorithms and data structures that optimize the memory resources required. Operations involving standard dense-matrix structures are indeed slower and inefficient when applied to large sparse matrices as most of the memory resources are wasted on the zero elements [7]. For this reason sparse data are found in a compressed format that enables less storage occupation.

There are several existing formats in which such data structures can be stored [14]. For the scope of this research it's chosen *Compressed Sparse Row* (CSR), originally introduced by George and Liu [15], in which only three arrays are saved: the actual non-zero values (nz), the column indexes of the values and the container of row pointers, those values indicating the start and the end of each row in the original two-dimensional format, pointing at the values' vector. This approach discards the zero values [8] [9] [10].

D. Sparse-Matrix Vector multiplication

Common sparse computations can involve simple kernel such as Sparse-Matrix Vector (SpVM), perhaps the most used one, used to obtain $y \leftarrow Ax$ with A being a $n \times n$ sparse matrix and x,y two $n \times 1$ vectors [16].
Irregularity of SpVM's data access patterns and higher frequency of access per operation than its dense counterpart, pose this operation as an interesting study case to inspect its scalability on Multi-core shared-memory architectures [17] [18].

E. OpenMP library

The entire parallelization section of this research was conducted exploiting different functionalities offered by OpenMP. OpenMP is an application programming interface (API) that offers a portable framework for parallel programming on shared-memory systems. More specifically, parallelizing nested loops is facilitated together with the distribution of the workload across the instantiated threads, all achievable by using compiler directives. [11] [12] [13]

II. State of the Art

The extensive literature shows that SpMV is predominantly memory-bound due to its low arithmetic intensity. Therefore, performance is limited to the available memory bandwidth rather than the intrinsic computational throughput. This pushes reasearch and effort towards improving data locality and memory traffic minimization.
The approaches exploring parallel implementations, using both libraries and GPU-based kernels [22], achieve significant results only when the input samples considered are sufficiently large to provide the working thread enough workload to mitigate overheads introduced by synchronization.

III. Contributions and Research Methodology

A. Contributions and achievements

The contributions of this research can be stated as follows:

- We are able to characterize the performance limits of OpenMP based SpMV algorithm considering different matrices as inputs having disparate dimensions.
- We conducted an analysis on the strong scalability of the algorithm considering a changing number of instantiated threads, workload and scheduling policies, precisely identifying the conditions when adopting a similar implementation is beneficial or counterproductive.
- We investigate impactful factors such as thread-memory affinity which contribute to the discrepancy between the theoretical and obtained results.

B. Research and operational methodology

1) COO to CSR: The actual first step taken was to convert the matrix's storage format from its native COO (Coordinate format) to CSR. The choice of working with a specific format rather than another is supported by the following reasons, concerning memory addresses and resources usage: [18]

- CSR has a less storing overhead which leads to a more efficient use of the memory resources. This is due to the fact that only column indexes and row pointers are stored. Distinctively, COO stores both row and column indexes for each nz-element [19].
- In CSR the nz-elements stored positions follow the row order, meaning that, when considering a row, its nz-values and column indexes are contiguous in memory. In the case of a specific partial or full matrix layout (i.e. partial or full matrix simmetry) this approach registers much better performances in terms of cache spatial and temporal locality, helping reducing memory misses [20] [21].

This procedure involves sorting the three arrays, given $a = (i, j)$ and $b = (i', j')$, with the criteria shown in (1).

$$a < b \iff (i < i') \; or \; (i = i' \; and \; j < j') \qquad (1)$$

To sort the values, Quicksort was used, given its computational complexity of $O(n \log(n))$, making it feasible to operate on inputs with an elevate number of nz-values.
The next steps were to count the number of elements for each row (1) and computing the prefix-sum (2). We refer to $rowIndex$ as the array of row indices, originally retrieved from COO format and sorted, and to $rowPtr$ as the output array containing the row pointers. Prefix-sum (2) is the sum, for each elements of $rowPtr$, of the previous ones.

Algorithm 1 Row elements counter

```
for i ← 0 to nnz − 1 do
    rowPtr[rowIndex[i]] ← rowPtr[rowIndex[i]] + 1
end for
```

Algorithm 2 Prefix-sum

```
for i ← 0 to nrows − 1 do
    rowPtr[i + 1] ← rowPtr[i + 1] + rowPtr[i]
end for
```

2) Computing the product: The implementation of the SpMV product involved the use of a random generated vector of size equal to the number of rows, referred to as vec. The main concern was standardize the notation from the 1-based of the COO format to the 0-based used in CSR. Briefly, the only difference is estabilishing which index to assign to the first element of an array, changing from being 1 (in 1-based indexing) to 0 (in 0-based indexing). When extracting the matrix intrinsics, the format it came was using 1-based indexing, forcing us to adapt this condition when considering the single element of vec in the multiplication (3). Note that this condition caused also the counting of the elements in each row to be shifted by 1 in 1.

Algorithm 3 SpVM product

```
for i ← 0 to nrows − 1 do
    for j ← rowPtr[i] to rowPtr[i + 1] − 1 do
        y[i] += val[j] × vec[colIndex[j] − 1]
    end for
end for
```

3) Parallel region: The nested loop section was parallelized exploiting compiler directive `#pragma` together with OpenMP directive `omp parallel for`. This allows the parallel region to be created and, when encountered, a logical team of threads to be formed. Each thread in the team executes all statements within a parallel region except for work-sharing constructs [23]. In our specific implementation each unit operates on a number of rows that depends on the specified chunksize (subsubsection IV-C3). The nested loop is executed sequentially by each assigned thread. Ultimately, this behavior prevents from data races on the i-th element of the result vector y as it is always accessed by a unique thread.
Eventually, i and j have been added to OpenMP's `private` clause, consequently, each thread has its own private copy of the variable, and modifications made by a thread to its copy are not visible to other threads [24]. This was made for the sake of better conceptual clarity since in OpenMP constraints it's enforced that loop iteration variables are private within their loops, so the value of the iteration variable after the loop is the same as if the loop were run sequentially [24].

IV. Experiments and system description

A. System description and target architecture

All simulations and tests were performed on the UniTN HPC cluster, a combination of specialized hardware, including a group of large and powerful computers, and a distributed processing software framework configured to handle massive amounts of data at high speeds with parallel performance and high availability [25]. The system provides both CPU

and GPU computing capabilities, but in the context of this research only CPU resources were exploited. Although total CPU resources available are 142 calculation nodes for 7674 usable cores and 65TB RAM distributed and shared among the nodes, only a single node with up to 96 physical cores (x86_64) was involved (Intel(R) Xeon(R) Gold 6252N CPU @ 2.30GHz), formed by 4 NUMA nodes with 24 cores each, without hyperthreading. LLC cache dimension: 36608KB. OS: Red Hat Enterprise Linux 7 – version 7.9 [26].

B. Libraries and programming toolchains

As mentioned, for the parallelization section of the algorithm, OpenMP library was used. Whenever the `#pragma` directive is encountered, the compiler links the OpenMP runtime library (libgomp for gcc) only if `-fopenmp` flag is specified at compilation.
The algorithm was written using C language and compiled using `gcc-9.1.0`, directly loaded as module in the cluster's environment, with `-O3` optimization.
To evalute cache addresses and statistics, `perf` command was invoked directly on the program execution.

C. Experimental setup and procedures

1) Input matrices: This research was conducted considering different sparse matrices of different dimension and sparsity degree. Each sample has been classified as small, medium or large depending on the number of nz values within a fixed range. In the choice of the samples, different pattern and numerical simmetries were taken into account. I

TABLE I
MATRIX SAMPLES INTRINSICS

Name	rows	cols	nz elements	type*
bayer03	6747	6747	29195	small
memplus	17758	17758	99147	small
cage13	445315	445315	7479343	medium
G3_circuit	1585478	1585478	7660826	medium
rajat31	4690002	4690002	20316253	large

**small*: nz $<$ 1M, *medium*: 1M $<$ nz $<$ 10M, *large*: nz $>$ 10M

2) Methodology: The approach used for evaluating the execution time is different from the cache addresses evaluation. As far as time performance is concerned we let the SpVM product *iterate* 10 times within the same process and then considered the 90 percentile out of the obtained results. This was made with the intent to isolate from the evaluation possible outliers caused mainly by a *cold-start effect*. Since we aim to investigate the algorithm's steady-state time performance, these values are then discarded.
Differently, the cache performance evaluation stage focuses on the *cold* algorithm's behaviour. Thus, the results considered are those returned after a single iteration of the product loop.

3) Test cases: As already mentioned in subsection III-A, our goal is to inspect the strong scaling of the algorithm as resources increases, keeping the problem's size fixed. To make this investigation possible different configurations had been tested tweaking the following parameters:

- **Thread number**: the number of logical thread allocated and managed by OpenMP library. It ranges from 1 to 64.
- **Chunksize**: dimension of each block of iterations assigned to each thread unit, ranging between 1 and 10000.
- **schedule type**: it defines the workload balance among different threads and specifies which policy to adopt when new blocks of iterations are assigned to them [27]. Three different types of schedules had been tested: *static*, *dynamic* and *guided*.
  - *static*: it's the most basic policy where each thread gets approximately the same number of iterations as any other thread, and each thread can independently determine the iterations assigned to it.
  - *dynamic*: in this case no thread waits taskless for longer than it takes another thread to execute its final iteration. This requirement means iterations must be assigned one at a time to threads as they become available, with synchronization for each assignment.
  - *guided*: Like dynamic, the guided schedule guarantees that no thread waits longer than it takes another thread to execute its final iteration, although it requires the fewest synchronizations.

D. Metrics

In order to convey the most information as possible in the results presentation, different metrics are being used.

- **Speedup**: speedup is exploited when analyzing the strong scaling of the implementation. For this purpose it's calculated as the rate of sequental time excution over the one parallelizing with N threads.
- **Cache miss percentage**: we encounter this ratio when evaluating the impact on cache addresses when scaling the number of threads. This is simply calculated as the total number of a specific cache level addresses over the number of misses on it, put in percentage. This can address different cache levels, in this research only *LLC* level was considered.

V. RESULTS

A. Strong scaling for different classes

The results of the strong scaling analysis shows a clear relationship between implementation's scalability and sample's size class.
Large matrices show good strong scaling (Figure 1). This comes from the fact that each parallel thread has been assigned with substantial work due to the elevate number of nz elements. Therefore, the execution benefits from parallelization before saturating the memory bandwidth, reaching a maximum speedup of **7.59x** higher than the sequential execution (*G3_circuit – medium class – best chunksize and schedule*).

Differently, small samples show limited or no speedup at all (i.e. $T_{\text{seq}} < T_{\text{par}}$) (Figure 1). When the number of nz elements is reduced, the overhead generated by thread creation,

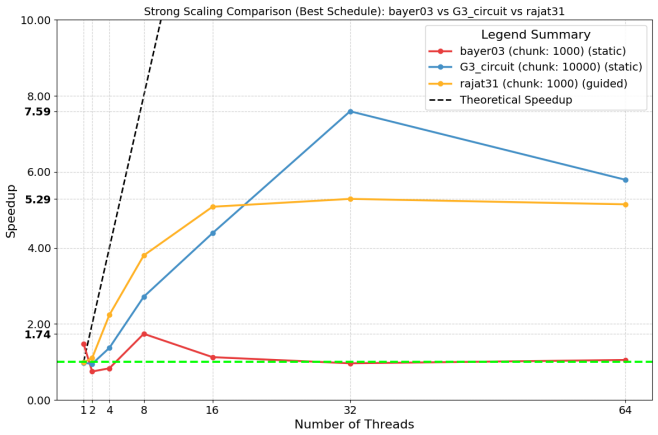

Fig. 1. Strong scaling comparison for classes sample – best chunksize and schedule considered. In limegreen the sequential speedup (1x).

synchronization, scheduling and memory traffic overwhelms the benefits brought by a parallelization approach, in which, in this specific case, each operating thread receive an insignificant amount of work.

B. Parallelization effect on cache memory

Analyzing the last level cache (LLC) misses can further contirbute to explain the poor results obtained in terms of speedup when increasing the number of threads on small samples. In

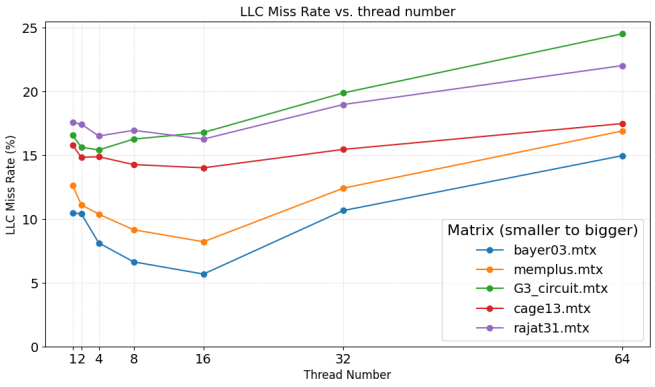

Fig. 2. Comparison of LLC-misses ratio for different matrices - for each matrix-thread number combination the mean over all of the possible schedule and chunksize's results is considered

case of small datasets, almost all the data needed to compute the SpVM can be entirely stored in the cache memory. For this reason, as shown in Figure 2, the behaviour of the LLC-misses curve depends on the matrix size: small ones tends to exhibit lower LLC-miss percentages than larger ones. This also suggests that, when threads number increases, accesses to the main memory from large matrices increases, enabling strong scaling until the bottleneck of memory bandwidth is encountered. From this, SpMV is said to be a *memory-bounded* kernel, not limited by the number of floating point operations (FLOPs) [28].

C. Thread pinning and NUMA effect

The experiments were carried out considering the NUMA (Non-Uniform Memory Access) architecture of the allocated node, where each memory domain is physically closer to a specific group of cores, making accessing to local memory faster than to other remote sections. Thread pinning is therefore crucial: by pinning each thread to a specific core, it prevents thread migration, situation that would degrade cache locality and increase memory latency. This choice results in more stable performances and reduced variability in the measurements. Figure 3 shows the effects of the non-uniform memory access.

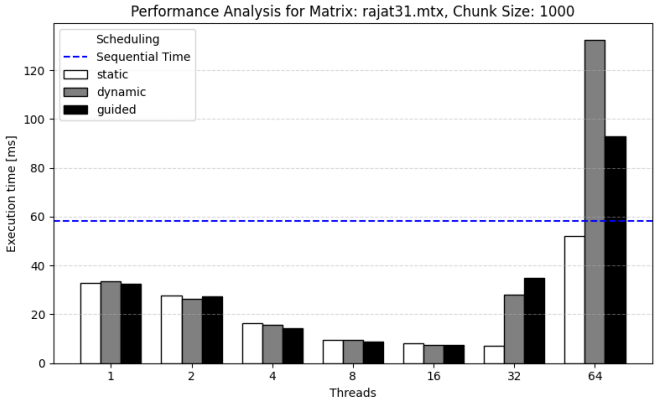

Fig. 3. NUMA effect on large sample with optimal chunksize – **no thread pinning enabled**. Over a certain thread number threshold results show degrading noise

VI. Conclusions and Future Work

In this research we examined OpenMP parallelization of the SpMV kernel across different matrices sizes. The results highlighted good strong scaling for medium-large samples (up to **7.59x** for a specific sample in optimal conditions), until memory-bandwidth limit is hit, while small matrices exhibited poor or even no speedup at all, due to an impactful thread synchronization overhead. Cache misses analysis supports this behaviour with small datasets fitting in cache memory and larger ones stressing main memory (DRAM), clearly demonstrating the conditions in which parallelization of the SpMV kernel is beneficial.

As long as future work is concerned, exploring different storing formats might open to new possibilities to mitigate the effect of the memory bandwidth limit. This approach is already adopted by some implementations in the current literature, some of them with the goal to further optimize CSR storing overhead. In this scope honourable mentions goes to *LSRB-CSR* specifically designed for GPU systems [29], *Tiled-CSR* structured to release the most power as possible from tensor cores in sparse matrix-matrix multiplication [30] and *MdCSR* proposing a new way to optimize column indexes storing [31].

REFERENCES

[1] D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 6th ed., Morgan Kaufmann, 2020.

[2] J. Dongarra, T. Hoefler, D. Keyes, et al., The International Exascale Software Project Roadmap, International Journal of High Performance Computing Applications, vol. 25, no. 1, pp. 3-60, 2011.

[3] J. Fang, C. Huang, T. Tang, and Z. Wang, "Parallel Programming Models for Heterogeneous Many-Cores: A Comprehensive Survey," CCF Trans. High Perf. Comput., 2020.

[4] Yan, Di; Wu, Tao; Liu, Ying; Gao, Yang (2017). "An efficient sparse-dense matrix multiplication on a multicore system". 2017 IEEE 17th International Conference on Communication Technology (ICCT). IEEE. pp. 1880–3.

[5] Scott, J., Tůma, M. (2023). An Introduction to Sparse Matrices. In: Algorithms for Sparse Linear Systems. Nečas Center Series. Birkhäuser, Cham.https://doi.org/10.1007/978-3-031-25820-6_1.

[6] T. Gale, E. Elsen, and S. Hooker, "Sparse GPU Kernels for Deep Learning," in Proc. ACM/IEEE Int. Conf. High Performance Computing (SC '20), 2020. [Online]. Available: https://cs.stanford.edu/~matei/papers/2020/sc_sparse_gpu.pdf

[7] G. Xiao, C. Yin, T. Zhou, X. Li, Y. Chen, and K. Li, "A Survey of Accelerating Parallel Sparse Linear Algebra," ACM Comput. Surv., vol.56, no.1, 2023, doi:10.1145/3604606.

[8] P. A. Lane and J. D. Booth, "Heterogeneous Sparse Matrix-Vector Multiplication via Compressed Sparse Row Format," arXiv preprint arXiv:2203.05096, 2022.

[9] M. Heller and T. Oberhuber, "Adaptive Row-grouped CSR Format for Storing of Sparse Matrices on GPU," arXiv preprint arXiv:1203.5737, 2012.

[10] SciPy Documentation – CSR Matrix (Compressed Sparse Row), v1.14.0, 2024. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

[11] M. J. Voss, Ed., OpenMP Shared Memory Parallel Programming (WOMPAT 2003 Proceedings). Berlin, Germany: Springer, 2003.

[12] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," IEEE Computational Science & Engineering, vol. 5, no. 1, pp. 46–55, 1998.

[13] B. Chapman, G. Jost, and R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming. Cambridge, MA, USA: MIT Press, 2007.

[14] T. Gilray, Z. Huang, L. Bowman, M. D. Johnson, and J. W. Berry, "Dynamic-CSR: A Fast, Dynamic Compressed Sparse Row Library," Univ. of Utah, Tech. Rep., 2023.

[15] A. George and J. Liu, Computer Solution of Large Sparse Positive Definite Systems. Englewood Cliffs, NJ, USA : Prentice Hall, 1981.

[16] H. Kabir, J. D. Booth and P. Raghavan, "A multilevel compressed sparse row format for efficient sparse computations on multicore processors," 2014 21st International Conference on High Performance Computing (HiPC), Goa, India, 2014, pp. 1-10, doi: 10.1109/HiPC.2014.7116882

[17] D. Guo and W. Gropp, "Optimizing sparse data structures for matrix-vector multiply," Int. J. High Perform. Comput. Appl., vol. 25, no. 1, pp. 115–131, Feb. 2011.

[18] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," In Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), ser. Supercomputing 99. New York, NY, USA : ACM, 1999.

[19] M. B. Giles, "Efficient Sparse Matrix-Vector Multiplication," Oxford University Mathematical Institute, Parallel SpMV Course Notes, University of Oxford, 2013.

[20] J. King, Dynamic Sparse-Matrix Allocation on GPUs, Tech. Rep., University of Utah, 2014.

[21] J. B. White and P. Sadayappan, "On improving the performance of sparse matrix-vector multiplication," in Proc. 4th Int. Conf. High Performance Computing (HiPC), 1997, pp. 66–71.

[22] P. Sadayappan, S. Williams, K. Underwood, and E. Saule, "Efficient sparse matrix–vector multiplication on GPUs using the CSR storage format," in Proc. IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2014, pp. 27–34.

[23] https://www.ibm.com/docs/en/xl-c-aix/13.1.2?topic=processing-pragma-omp-parallel

[24] https://www.ibm.com/docs/en/xl-c-and-cpp-aix/13.1.0?topic=programs-shared-private-variables-in-parallel-environment

[25] https://www.hpe.com/emea_europe/en/what-is/hpc-clusters.html

[26] https://servicedesk.unitn.it/sd/en/kb-article/cluster-hpc-architecture?id=unitrento_v2_kb_article&table=kb_knowledge&sys_id=3041500ec30e91102f9f70e4e401312c&recordUrl=%2Fkb_view.do%3Fsys_kb_id%3D3041500ec30e91102f9f70e4e401312c

[27] https://learn.microsoft.com/en-us/cpp/parallel/openmp/d-using-the-schedule-clause?view=msvc-170

[28] S. Dinkins, M. M. Strout and S. Rajopadhye, "A model for predicting the performance of sparse matrix vector multiply (SpMV) using memory bandwidth requirements and data locality". Colorado State University, Technical Report, 2018. - https://doi.org/10.25675/3.020795

[29] L. Liu, M. Liu, C. Wang and J. Wang, "LSRB-CSR: A Low Overhead Storage Format for SpMV on the GPU Systems," 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), Melbourne, VIC, Australia, 2015, pp. 733-741, doi: 10.1109/ICPADS.2015.97.

[30] Z. Xue et al., "Releasing the Potential of Tensor Core for Unstructured SpMM using Tiled-CSR Format," 2023 IEEE 41st International Conference on Computer Design (ICCD), Washington, DC, USA, 2023, pp. 457-464, doi: 10.1109/ICCD58817.2023.00076.

[31] G. Noble, S. Nalesh, S. Kala, S. Ullah and A. Kumar, "MdCSR: A Memory-Efficient Sparse Matrix Compression Format," in IEEE Embedded Systems Letters, vol. 17, no. 5, pp. 289-292, Oct. 2025, doi: 10.1109/LES.2025.3598189.