Parallel Project Report:(Sparse Matrix Vector Multiplication)

Andrea Dalla Villa (ID: 242637)
Email: andrea.dallavilla@studenti.unitn.it
Course: Introduction to Parallel Computing (2025–2026)

***Abstract*—Sparse matrix vector multiplication (SpMV) is one of the most computational kernel used in different fields and analyzed more times to obtain the best possible performance solution. In this paper I am going to explain speedup and memory access differences in a NUMA architecture between sequential,with its compiler optimization, and parallel approach, e.g. openMP parallelization and SIMD vectorization; furthermore, I am going to use the CSR format, useful for considering only non zero values (nz) and reducing the number of iterations.**

***Index Terms*—SpMV, CSR, openMP, threads**

I. Introduction

I will start talking about the reasons why some scientific papers are careful about the sparse matrix vector multiplication; we just need to think how redundant the computation is and how much time we are wasting while doing it. CPUs do these kinds of operations every time, and often they are more complex than SpMV; furthermore, whether we chose a sparse matrix instead of a dense matrix we risk to compute some useless operations with a great number of zero values. The main topic that I explain is how we can improve the performances and how its change using a sequential compared to a parallel approach. To begin with, I would like to specify many considerations, for instance:

- **Sparsity**,since values are not allocated contiguously, a high level of it does not exploit the spatial locality if we want to consider only non zero values.
- **The structure of the matrix**; the results change whether you use a diagonal, symmetrical or irregular sparse matrices and another important consideration is the number of non zero values.
- **Architecture used**, when you increase the number of computations, you should use an appropriate architecture for measuring better performances. For instance, performance can have different results for an architecture with fewer cores or if it is a NUMA or an UMA architecture.

II. State of the art

Trying to improve performance on SpMV is very useful, given that we live in a multidimensional world where everything is represented by 2D/3D matrices and a great SpMV algorithm can change performance for many engineering fields like machine learning with deep neural networks [6], where SpMV plays an important role. Another field is when we have spoken about images or videos [7] and always new approaches for this computational kernel are implemented to maximize performance, like CNN [8] or one of the new features from OpenMP 4.0 version like the clause *for+simd* [5].

III. Contribution and Methodology

A. SpMV algorithm

for mitigating the issue of not contiguous data I have used the CSR format that consists in storing in an array the start index of the non zero elements related to a specific row, but this does not fix the entire downside.

Having said that, I can start showing the CSR SpMV algorithm that I have used, exposing the differences between sequential and parallel code by increasing the number of threads, evaluating different scheduling strategies and identifying a possible bottleneck. Writing the sequential code I could consider different optimization techniques for instance, whether we take a simple nested ”for” loop for the matrix-vector product, we have to consider the access pattern. If we access the memory by column instead of by rows, memory addresses are not aligning and this is not ”cache friendly” and it may causes performance degradation with many strides. When we use a sparse matrix to compute the entire matrix vector multiplication is useless and expensive, for this reason we can use COO or CSR formats. Suitsparse [1] just prepare matrix files in COO format but sorted by columns. I have used a quick sort algorithm for sorting the mtx files by row and after I have converted them to CSR format. I have chosen quick sort for its log(n) complexity. This is the multiplication algorithm that I have taken from [2] and implemented in C++.

The CSR Format Sparse Matrix-Vector Multiplication Algorithm

Require: N : the total number of rows in matrix $\mathcal{A}$
val : nonzero values in matrix $\mathcal{A}$
col : column indices of values in matrix $\mathcal{A}$
row : pointers to row starts in matrix $\mathcal{A}$
Input : X : the vector need to calculate
Output : Y : the vector after calculate

```
for i ← 0 to N − 1 do
    for j ← row_i to row_{i+1}-1 do
        Y_i ← Y_i + val_j * X_{col_j}
    end
end
```

Fig. 1. Pseudo code of the algorithm

We can identify a possible bottleneck just from this algorithm in the equation in Fig.1:

$$Y[i] += val[j] * X[col[j]] \quad (1)$$

where the memory access from X[col[j]] makes unstable strides. Due to this bottleneck, the matrix structure influences several data layout [3] and speedups since a diagonal dense matrix exploits the spatial locality better than other sparse matrix when it tries to access data near diagonal, and this reduces the bottleneck; same thing if the matrix is pretty dense or it has many nearest values in a row. CSR stores data in a vector and this way is contiguous but when it loads data for SpMV it depends on column vector which loses the contiguousness. This condition will become clear when we analyze the results from the experiments.

B. Parallelization (openMP)

I implemented row-level parallelism with #pragma omp parallel for num_threads(thread_count), and I used different scheduling types and chunk sizes.

IV. Experiments and System Description

A. Platform

I have done my project on the HPC cluster, an Intel x86_64 architecture with 2 head nodes and 142 CPU for a total of 7674 cores and 10 GPU for a total of 48128 CUDA cores inside the computing nodes. Any node has a NUMA behavior with 4 blocks and each one has 18 CPUs I have run my program on a CPU with 64 cores.
OS: x86_64 GNU LINUX
compiler: g++ version 9.1.0.
Dataset: I downloaded different structures with different sparsity level on "Suitsparse" website [1] e.g. pretty diagonal matrices like "1138_bus" and "bccsstk35", dense diagonal matrix like "Zd_Jac6", symmetrical matrices like "smt" and "ship_001" and an irregular sparse matrix like "adder_dcop_11".

matrix	*rows number*	*columns number*	*non zeros*
1138_bus	1138	1138	2596
adder_dcop_11	1813	1813	11243
bccsstk35	30237	30237	740220
Zd_Jac6	22835	22835	1711983
smt	25710	25710	1889447
ship_001	34920	34920	2339575

matrices used sorted by non-zeros number.

B. Metrics & Methodology

I report:

- **Speedup**, average of 10 runs and report the 90% percentile;
- **percentage of cache miss**, average of 10 runs and report the 90% percentile;

In these experiments I have compared sequential code with parallel code and I have collected the results about speedup even using different thread numbers, scheduling strategies e.g. static, dynamic and guided with different chunk sizes like 10,100,1000 chunks and the percentage of cache miss for every thread. I have used openMP for parallel code. Furthermore, I tested the clause *for+simd* where the iterations are divided for thread and into each thread iterations are vectorized by the compiler with SIMD techniques. I have compared scheduling strategies even for this last feature, obtaining interesting results.

V. Results and Discussion

A. Sequential code vs Parallel Code

Before analyzing the graphs below, I want to underline that I have used O1,O2,O3,Ofast optimizations only for sequential code. These optimizations are made by the compiler which generates different Assembly codes with a fast logic in comparison with the original logic and these operations minimize the number of possible hazards. I have used different number of threads, Nevertheless I arrived to a maximum of 64 threads since I could use one thread per core. Graphs are sorted by number of non zero values on ascending order, equal for the other graphs below.

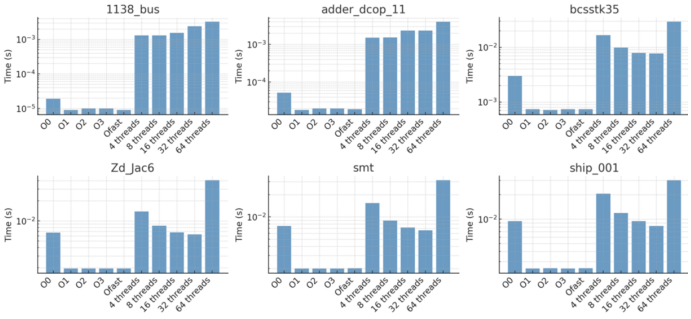

Fig. 2. speedup for SpMV, on the y time is in logarithmic scale, on the x we have sequential code with its compiler optimizations on the right and parallel code sorted by number of threads on the left

In the three upper graphs the sequential code is better than the parallel code since the number of non zero values is small and any thread has a less processing load and parallelizing does not make sense since they waste resources. Parallelizing an algorithm is useful when a thread has more or less as much work as every other thread, in our case when they have a sum with a fair number of multiplications. Much interesting is what happens in the others, we can see that parallelizing with 16/32 threads performs better than using a serial code with O0 optimization. However, why have not we obtained good results with few threads or even with 64 threads?
The idea is that we would like to increase the number of threads and obtain less speedup, this it is called linear speedup; Nevertheless, it is only a theoretical speedup as due to the fact that data dependencies, overhead for threads synchronization and shared memory data. There are different mathematic ways to determinate the better number of threads for our program, for instance, a pessimistic approach like Amdahl's law or other approaches that consider the number of cores like Gustafson's Law where if we increase the problem size,the "inherently serial" fraction of the program decreases in size. [4] In our case we can see that a good number of threads is 32, with a thread per core the speedup is twice as high as before. Afterwards, we will see that this idea is visible with scheduling and cache access where the best number of threads reduces the percentage of cache miss.

B. scheduling strategies

I will clarify about differences between scheduling strategies, I have omitted runtime schedule type since I have not used it. However, you can find it here [4].

- *static*, the iterations are assigned to the threads before the loop is executed, the system assigns chunks to each thread in a round-robin fashion;static type is useful when the number of iterations is equal for each thread. Furthermore, if the cost of iterations changes linearly as the loop executes, choosing a static strategy with a small chunk size is the best solution.
- *dynamic*, The iterations are assigned to the threads while the loop is executing and after each set of iterations is executed by its thread. This strategy requires more run-time operations compared to the static strategy. Iterations are divided into chunks and when a thread finishes a chunk the system gives another one to it. With different sizes of iterations set, dynamic is better than static as due to the fact that the number of iterations can not be equal for each thread.
- *guided*, is similar to dynamic, the main difference is that when a chunk is completed the size of the next chunk decrease. When the later iterations are intense, guided schedule can decrease the computation load.

We are ready to analyze the results that i obtained:

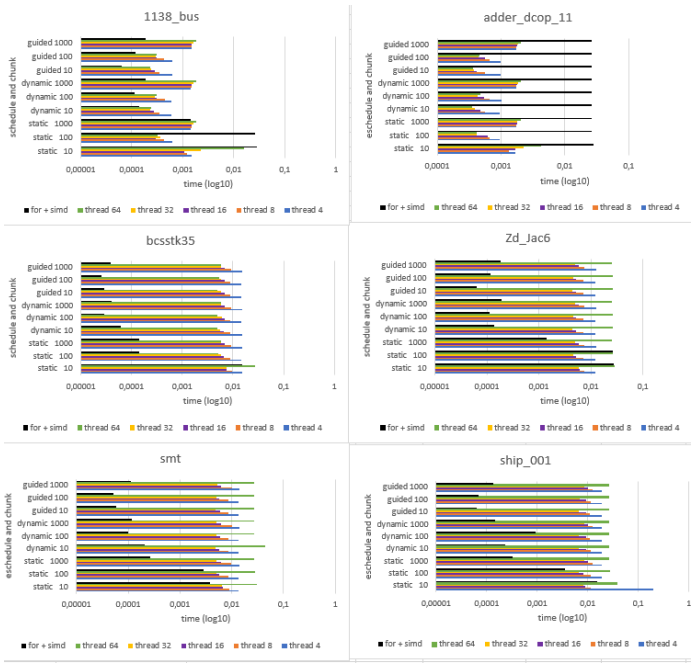

Fig. 3. speedup for SpMV, on the x time is in logarithmic scale, on the y we have different scheduling strategies with different chunk size for several number of threads and for+simd approach. graphs are ordered by number of non zeros in ascending mode.

we can see that we have good performance in dynamic and guided compared to static schedule when we use irregular sparse matrix like "adder_dcop_11" since the number of iterations per thread is not fair. indeed, other matrices that I have used are symmetric or diagonal and when you have a regular pattern like these the number of iterations per thread is more or less equal and static schedule performs better than the others. Furthermore, we can observe like in Fig.2. that 32 threads are the better quantity of threads than the others, obviously I am talking about the last three graphs since in the previous threes the sequential code is better than the parallel one.
As depicted in the graphs I have tried to use a small, a medium and a large chunk size for showing the effects and we have a very small difference between static with 100 chunk and dynamic/guided with 10 chunks, but very close to dynamic/guided with 100 chunks. a large chunk size can decrease the load balance between threads since several threads will finish before others and they will wait them. In the opposite case, for dynamic and guided schedules, a small chunk size can cause busy-waiting while threads wait their chunk size from the run-time system. The main reason why we achieve better speedup with a static schedule using 100 chunks, and with a dynamic schedule using less chunks, is the sparsity of the matrix. the load balance is maintained fair for threads which it is not established before. Now we consider the case of *for+simd* announced before; in this case I have obtained better speedup on all SpMV except one, so far in

cases with few non zero values sequential code obtains better speedup results than parallel code but with *for+simd* things are a bit different. Furthermore, this approach performs better on dynamic and guided schedules exploit the sparsity and SIMD maximizes the work load for each thread. The speedup issue with *for+simd* in the SpMV with "adder_dcop_11" matrix is caused by the matrix pattern. SIMD performs better when we have contiguous data access, it exploits spatial locality and works with contiguous data in one single instruction (Single Instruction Multiple Data). For this reason using diagonal and symmetric matrices with SIMD is a great choice.

C. Cache miss

We have arrived at the last, but not least, part of my experiments with SpMV. What remains to be discussed is cache miss that is the percentage of times that the CPU does not find the data. If data are not available in cache, CPU has to find them in the RAM and speedup increases. The graphs below show the cache miss percentages for each SpMV:

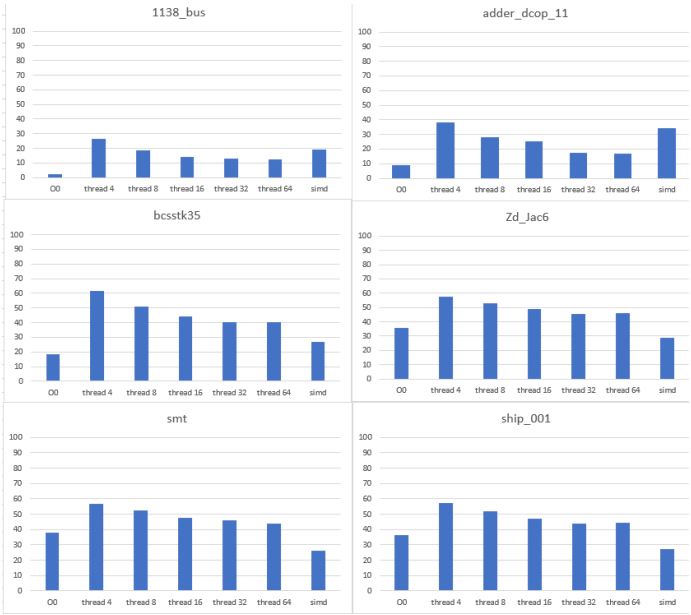

Fig. 4. percentage of cache miss for SpMV, on the x there are the measure of the sequential code compared to measures of the parallel code with different thread numbers and *for+simd* approach, on the y we have the percentage. graphs are ordered by number of non zeros in ascending mode.

It is trivial that the number of cache misses rises when we increase the number of non zero values but it is important to consider the organization of the cache. Every node has three cache levels, L1,L2,L3; L1 and L2 are pretty small(32K and 1024K) but L3 level is large compared to the others (36608K),I can check these on the HPC cluster using the command *lscpu*. The L3 level is shared across all cores. A sequential code uses only one core and data are stored in L1 and L2 levels and data are stored contiguously and this is the reason why its cache miss is low; however, this is true when the number of data stored in the cache is relatively small, we can see that when the number of elements rises small cache level like the first and the second can not store a large amount of data, data in the blocks are replaced and retrieved, generating cache misses. If we use a small number of threads, few cores work and for sharing data they use L3 level, often threads overwrite data that they cause cache misses. Furthermore, the bottleneck discussed earlier becomes visible when a replaced block is requested again by another thread. we can see here the effects of the dependence on column indexes (1). With *for+simd* each core works faster than without *simd* and each thread merges several data for computation leading to small speedups in comparison with sequential code speedups despite a bit high percentage of cache miss; Furthermore, I have just discussed the fact that SIMD exploits the spatial locality and the reason why in "adder_dcop_11" performs are bad.

VI. CONCLUSIONS

In this paper we have observed performance changes between different matrix formats and number of non zero values in a straightforward algorithm such as matrix-vector multiplication; nevertheless, considerations that emerged are not trivial. SpMV is an example of how an algorithm can be optimized and how much effort architectures put into computing these tasks. We have to specify that not all tasks are parallelizable, for instance it is worthless when we have few iterations. Furthermore, we have discovered some features of openMP for implementing to make the best choice for our task.

REFERENCES

[1] https://sparse.tamu.edu/

[2] J. -L. Zhang, L. Zhuang, J. Wan, X. -H. Xu, C. -F. Jiang and Y. -J. Ren, "COSC: Combine Optimized Sparse Matrix-Vector Multiplication for CSR Format," 2011 Sixth Annual Chinagrid Conference, Dalian, China, 2011, pp. 124-129, doi: 10.1109/ChinaGrid.2011.39. keywords: Optimization;Sparse matrices;Libraries;Distribution strategy;Indexes;Pipeline processing;Arrays;COSC;SpMV;CSR;optimization,

[3] O. Kislal, W. Ding, M. Kandemir and I. Demirkiran, "Optimizing sparse matrix vector multiplication on emerging multicores," 2013 IEEE 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS), Edinburgh, UK, 2013, pp. 1-10, doi: 10.1109/MuCoCoS.2013.6633600. keywords: Multicore processing;Layout;Sparse matrices;Processor scheduling;Vectors;Optimization;Compiler Optimization;Data Locality;Data Transformation;Loop Transformation;Sparse Matrix-Vector Multiplication;Multicore,

[4] Peter S. Pacheco, Matthew Malensek, An Introduction to Parallel Programming, Second Edition, Morgan Kaufmann, 2021. ISBN: 978-0-12-804605-0

[5] C. Ponte, J. González-Domínguez and M. J. Martín, "Evaluation of OpenMP SIMD Directives on Xeon Phi Coprocessors," 2017 International Conference on High Performance Computing & Simulation (HPCS), Genoa, Italy, 2017, pp. 389-395, doi: 10.1109/HPCS.2017.65. keywords: Coprocessors;Instruction sets;Graphics processing units;Benchmark testing;Poisson equations;Heuristic algorithms;Parallel processing;Vectorization;OpenMP;SIMD directives;Intel Xeon Phi;Intel MIC,

[6] M. Mahesh, S. Nalesh and S. Kala, "Hardware Acceleration of SpMV Multiplier for Deep Learning," 2021 25th International Symposium on VLSI Design and Test (VDAT), Surat, India, 2021, pp. 1-6, doi: 10.1109/VDAT53777.2021.9600988. keywords: Deep learning;Machine learning algorithms;Quantization (signal);Power demand;Very large scale integration;Performance gain;Sparse matrices;Deep learning;FPGA;Sparse matrix;Performance,

[7] A. C. Shaji and K. Khare, ”Analysis of Optimization on Sparse Matrix Vector Multiplication Model Application in Digital Signal Processing,” 2024 IEEE 3rd International Conference on Electrical Power and Energy Systems (ICEPES), Bhopal, India, 2024, pp. 1-7, doi: 10.1109/ICEPES60647.2024.10653621. keywords: Energy consumption;Signal processing algorithms;Digital signal processing;Vectors;Energy efficiency;Resource management;Sparse matrices;SpMV;FPGA;HLS;OpenCL;High-Performance Computing;Parallelism;DSP,

[8] P. Guo and C. Zhang, ”Sparse Matrix Selection for CSR-Based SpMV Using Deep Learning,” 2019 IEEE 5th International Conference on Computer and Communications (ICCC), Chengdu, China, 2019, pp. 2097-2101, doi: 10.1109/ICCC47050.2019.9064309. keywords: Sparse matrices;Training;Graphics processing units;Measurement;Machine learning;Feature extraction;Convolution;Convolutional Neural Network (CNN);Deep Learning;SpMV;GPU,