EPFL

Parallel and High Performance Computing

Dr. Pablo Antolín

Solution Series 5

April 10 2025

MPI - Graded exercise

1 Profiling

Performance profiling, using tools such as `gprof` or `perf`, is essential for identifying computational bottlenecks in the algorithm. However, obtaining clear insights can be challenging, particularly when analyzing code compiled with high optimization levels (e.g., `-O3`) applied to small input matrices, as the resulting efficiency leads to very short execution times. To mitigate this and achieve more meaningful results, one can either increase the problem size substantially by using a larger input matrix or compile the code with lower optimization levels (like `-O0`). While compiling with lower optimization helps correlate runtime more directly with the source code structure, it is important to remember that this does not represent the performance characteristics of the final, optimized application.
In this particular project, profiling analysis consistently reveals that the matrix-vector product is the most computationally dominant part of the algorithm. When compiled without optimization (`-O0`), this operation accounts for approximately 76% of the total execution time, allowing for a conservative estimate of the parallel fraction, $\alpha \approx 0.76$ if one only parallelizes the matrix-vector product. In contrast, if one parallelizes the entire loop including the scalar products, the parallel fraction is significantly higher, estimated to be $\alpha \geq 0.99$. This is due to the fact that even at the highest sampling frequency, `perf` does not detect any other operations apart from the matrix-vector product or the `daxpy` calls, which are used for the scalar products in the algorithm. The same holds for higher compiler optimization settings.

2 Parallelization Strategy

From the profiling results, we conclude that the matrix-vector product is the most time-consuming part of the algorithm and therefore the first candidate for parallelization. The matrix-vector product in COO format can be parallelized in two ways: Either, the matrix is partitioned in blocks of entries (variant 1) or the matrix is partitioned in blocks of rows (variant 2). In a more refined approach, one can combine the row-wise partitioning of the matrix-vector product with the row-wise partitioning of all the scalar products (variant 3).

2.1 Variant 1: Entry-wise partitioning of the matrix

In this variant (folder `solution_1`), the matrix is partitioned in blocks of entries, when it is parsed in COO format; see `MatrixCOO::read(const std::string & fn)` in `matrix_coo.cc`. More precisely, the three vectors representing the matrix A in COO format are partitioned into p blocks, where p is the number of processors. Let M denote the number of non-zero entries in the matrix A. If M/p is not an integer, the last processor will be assigned $M - (p-1)(M/p)$ entries. This allows for an even distribution of the work among the processors, as each processor is assigned a block of entries to process. Mathematically, this corresponds to decomposing the matrix A into a sum of p matrices A_1, A_2, ..., A_p such that

$$A = A_1 + A_2 + \ldots + A_p \tag{1}$$

where A_i is the submatrix assigned to processor i. The matrix-vector product can then be computed as

$$y = Ap_k = A_1p_k + A_2p_k + \ldots + A_pp_k = y_1 + y_2 + \ldots + y_p \tag{2}$$

where p_k is the search direction in the k-th iteration, y is the output vector and y_i is the output vector computed by processor i. The latter are computed on each processor in parallel with the same code as in the serial version. However, note that the vectors y_i may contain non-zero entries at the same index. Therefore, the final result must be computed by summing the vectors y_i on each processor. This can be done using the `MPI_Allreduce` function, which computes the sum of the vectors y_i and stores the result in the vector y on each processor. The rest of the code remains in serial form.

2.2 Variant 2: Row-wise partitioning of the matrix

This variant (folder `solution_2`) is similar to the previous one, but the matrix is partitioned in blocks of rows (see `MatrixCOO::read(const std::string & fn)` in `matrix_coo.cc`). More precisely, the matrix A is partitioned into p blocks of rows, where p is the number of processors. This also leads to a decomposition of the matrix A into a sum of p matrices. However, this time, the local vectors y_i are disjoint in the sense that they do not contain non-zero entries at the same index. Therefore, the final result can be computed by an `MPI_Allgatherv` operation that simply stacks the resulting vectors and without actually summing them. The latter improves the scalability of the algorithm.

2.3 Variant 3: Row-wise partitioning of the matrix and the scalar products

This variant (folder `solution_3`) is similar to variant 2. But instead of gathering the local vectors $y_i = A_ip_k$ directly, we continue to compute the subsequent scalar products $p_k \cdot A_ip_k$ and $r_{k+1} \cdot r_{k+1}$ locally too. This improves the parallel efficiency since we use the computational resources for a maximum of operations. Nevertheless, we still must reduce the scalar products to all processors and communicate the local search directions to all processors at the end of each iteration.

2.4 Algorithmic complexities

In all the three variants, the main bottleneck to parallelize is the matrix-vector product Ap_k. For the particular case of sparse matrices with bandwidth k, as the ones provided for testing, the computation complexity of the parallelized matrix-vector product is $\mathcal{O}(c_1kN/p)$ where N is the number of rows in the matrix (for the Variant 1 we have that $M = Nk$ and therefore the cost is $\mathcal{O}(c_1M/p)$); and c_1 is a constant independent from N or p, but that depends strongly on the implementation and the optimization level.

In the same way, the complexity of the scalar products $p_k \cdot A_i p_k$ and $r_{k+1} \cdot r_{k+1}$ is $\mathcal{O}(c_2 N)$. That becomes $\mathcal{O}(c_2 N/p)$ in the parallel case. Discarding the involved constants, for the sake of simplicity, we can estimate the total computation cost of the algorithm as

$$\begin{array}{ll} \mathcal{O}(N/p + N) & \text{for Variants 1 and 2} \\ \mathcal{O}(N/p) & \text{for Variant 3} \end{array} . \tag{3}$$

On the communication side, the most costly operation are the `MPI_Allgatherv` calls (in Variants 2 and 3) and the `MPI_Allreduce` calls (in Variant 1), whose complexities are[1] $\mathcal{O}(c_2 \log p + c_3 N)$, where c_2 and c_3 are constants that depend on the network topology and the implementation of the MPI library, but are independent from N or p. Note that in the `MPI_Allreduce` calls in Variant 3 the size of transferred data is $N = 1$, the result of scalar product, hence their complexity can be estimated as $\mathcal{O}(c_2 \log p)$. Discarding the involved constants as before, we can estimate the total communication cost of the algorithm as

$$\mathcal{O}(\log p + N) \quad \text{for all the Variants} . \tag{4}$$

2.5 Strong and weak scaling

2.5.1 Strong scaling

The idea behind strong scaling is to keep the size of the problem constant and increase the computational power (the number of processors). In our problem this translates into keeping the matrix size N constant and increasing the number of processors p. Thus, as p grows the computation cost decreases as $\mathcal{O}(1/p)$, while the communication cost grows as $\mathcal{O}(\log p)$.

We recall that, according to Amdahl's law, if α denotes the parallel fraction of the code, then the ideal speedup is given by

$$S = \frac{1}{(1-\alpha) + \frac{\alpha}{p}}.$$

This constitutes the theoretical upper bound for the speedup of the algorithm.

According to these theoretical observations, the trend we expect to observe (independently of the considered variant) is that initially the parallelization pays off and the total time (computation plus communication) decreases (speedup increases) when the number of processors grows, while at certain point the communication cost starts to dominate over the computation cost and the total time starts to increase again (speedup decreases), deviating from Amdahl's law. Due to its lower computation cost, variant 3 is expected to be the one that scales better.

The point at which the communication cost starts to dominate over the computation cost depends of course on N, but also on the complexity constants involved, the considered variant, the optimization level, etc.

On the other hand, considering growing matrix sizes N should lead to larger communication (see Equation (4)), and computation costs (specially for variants 1 and 2 due to the constant term N in Equation (3)), what will lead to a deterioration of the speedup.

[1] See for example: https://who.rocq.inria.fr/Laura.Grigori/TeachingDocs/UPMC_Master2/Slides_HPC_MN_DA/costMPIRoutines.pdf

2.5.2 Weak scaling

In the weak scaling analysis, the idea is to keep the work per processor constant when increasing the number of processors. To that end, we will use an initial matrix with size N_0 for $p = 1$ and scale it as $N_p = N_0 p$ for an increasing number of processors. In this way, the computation cost per processor will be

$$\begin{array}{ll} \mathcal{O}(N_0 + N_0 p) & \text{for Variants 1 and 2} \\ \mathcal{O}(N_0) & \text{for Variant 3} \end{array} . \tag{5}$$

I.e., for the first two variants, the computation cost per processor grows with p, what contradicts the definition of weak scaling (the work per processor must stay constant), but we will keep it for the sake of the analysis. For variant 3, the computation cost per processor stays constant.
On the other hand, the communication cost will grow as $\mathcal{O}(\log p + N_p) = \mathcal{O}(\log p + N_0 p)$ in all three cases. Therefore, the total cost (computation + communication) will grow linearly with p. Then, we expect to observe a descending trend that will be more pronounced for the first two variants (as the constant in front of the linear term p has contributions from both computation and communication), than for variant 3.

3 Results

In the following, we will present and analyze the strong and weak scaling results for the parallel CG implementation. All the results presented in the following are obtained on the `jed` cluster, using `gcc`, `openmpi`, and `openblas`, and considering both `-O0` and `-O3` optimization flags.

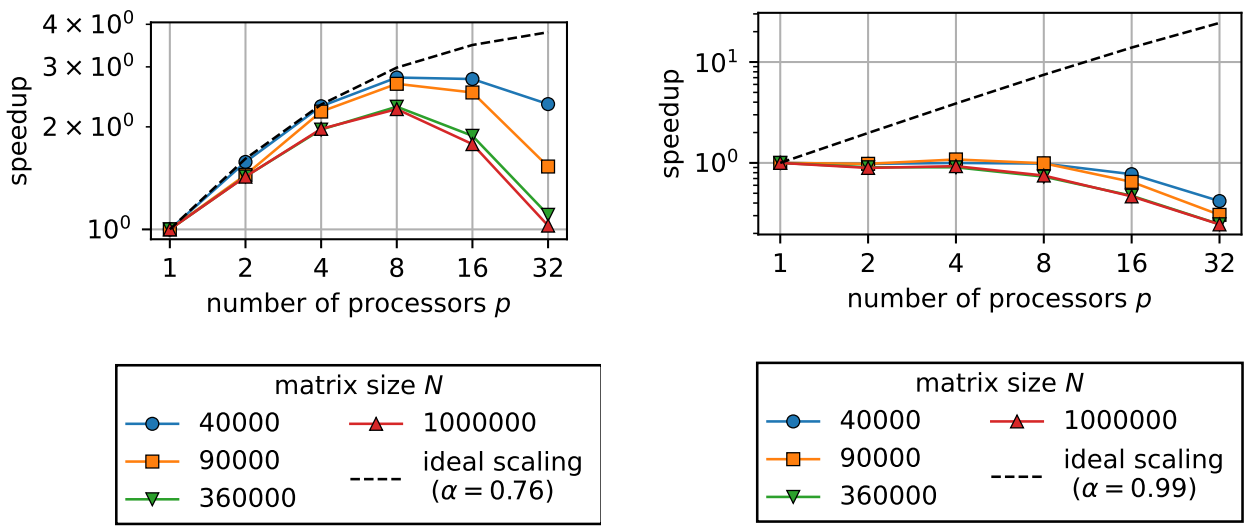
In order to analyze the strong scaling, we generated matrices of increasing sizes $N \in \{200^2, 300^2, 600^2, 1000^2\}$ and computed the obtained speedups for an increasing number of processors $p = 1, 2, \ldots, 32$. Results are shown in Sections 3.1 to 3.3. While for the weak scaling analysis, we will use initial sizes $N_0 \in \{256, 1024, 4096\}$ to see the influence of the matrix size.
Finally, we stress that, in the weak scaling analysis, to keep the work per processor constant, we also have to normalize the runtime by the number of iterations of the CG algorithm (that depends on N).

Let us now analyze the results for the three variants.

3.1 Variant 1

In Figure 1, the strong scalings are shown. For the lower optimization setting (a), the strong scaling closely follows Amdahl's law up to 16 processors. Then the speedup starts to deteriorate significantly. This suggests that for 32 processors, the communication cost starts to dominate over the computations on each processor. Furthermore, we observe that the scaling is worse for increasing problem sizes. Both observations are in line with the theoretical analysis presented in Section 2.5. For the higher optimization setting (b), the strong scaling is terrible. The reason for this may be the fact that, due to the compiler optimization, the constant c_1 in the computation cost gets significantly reduced, making the computation to communication ratio even further unbalanced.

Similarly, the weak scaling shown in Figure 2 shows that the parallel efficiency deteriorates rapidly for increasing numbers of processors. In addition, the weak scaling is worse for larger problem sizes, which is not surprising since both computation and communication costs grow linearly with p and the initial matrix size N_0. Both effects were already anticipated in Section 2.5.2.



(a) `-O0` optimization

(b) `-O3` optimization

Figure 1: Variant 1: Strong scaling results for optimization flag `-O0` (a) and `-O3` (b).

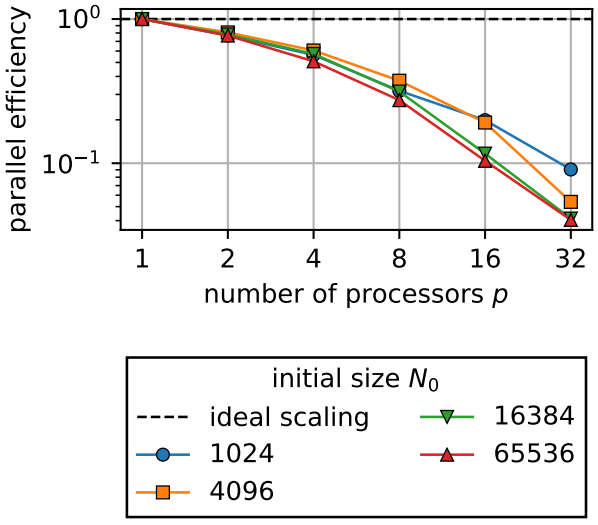
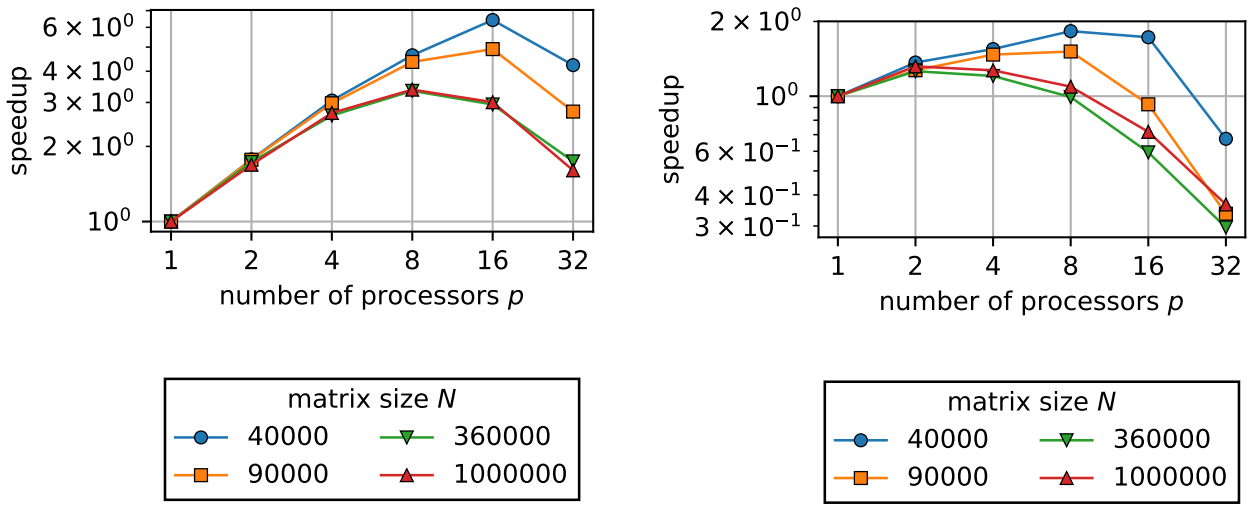


Figure 2: Variant 1: Weak scaling results for optimization flag `-O0`.



(a) `-O0` optimization

(b) `-O3` optimization

Figure 3: Variant 2: Strong scaling results for optimization flag `-O0` (a) and `-O3` (b).

3.2 Variant 2

The results for variant 2 are very similar to the one for variant 1 (see Figures 3 and 4). Note, however, that we are not parallelizing the exact same operations from the serial code since we avoid having to sum the different vectors $y_i, i = 1, \ldots, p$, we just stack them with `MPI_Allgatherv`. Consequently, the parallel fraction cannot be estimated by 76% anymore. Therefore, we dropped the ideal scaling from Figure 3 to avoid confusion. In particular, we observe that the strong scaling is better compared to variant 1 for the same reason. Beyond that, the same appreciations that were made for the strong and weak scaling of variant 1 hold here.

3.3 Variant 3

For variant 3, we note that for both optimization levels, we obtain a better speedup according to Amdahl's law (see Figure 5). In particular, this variant successfully parallelizes a much larger fraction of the code (see Equation 3). And, even if still low, the parallel efficiency of the weak scaling analysis (see Figure 6) is slightly better when compared with the ones obtained for variants 1 and 3 (cf. Figures 2 and 4). This may be due to the *actual* constant computation cost per processor considered in this case (recall estimation (5)). Beyond these particularities, the same observations made for the strong and weak scalings of variants 1 and 2 hold here.

4 Bonus: Dense matrices

The results above show that for sparse matrices in COO format, while the computation cost per processor descreases with p according to the estimate (3), the communication cost grows as in (4). Therefore, the algorithm total cost grows logarithmically with p. However, even if this holds true when applying the *same code* to dense matrices (same COO format but containing all the zero entries), slighlty different results can be observed.

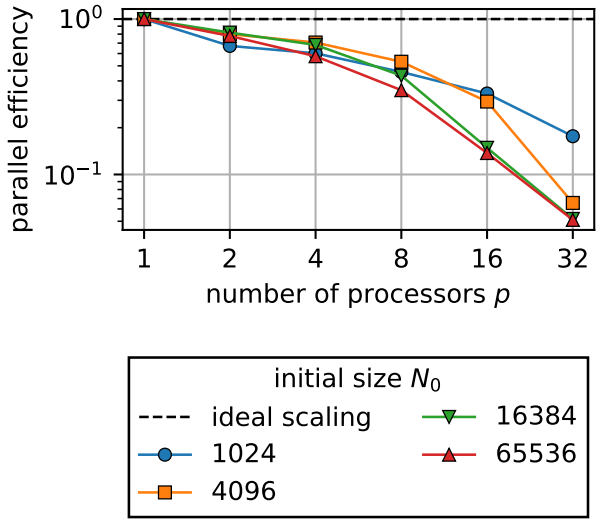
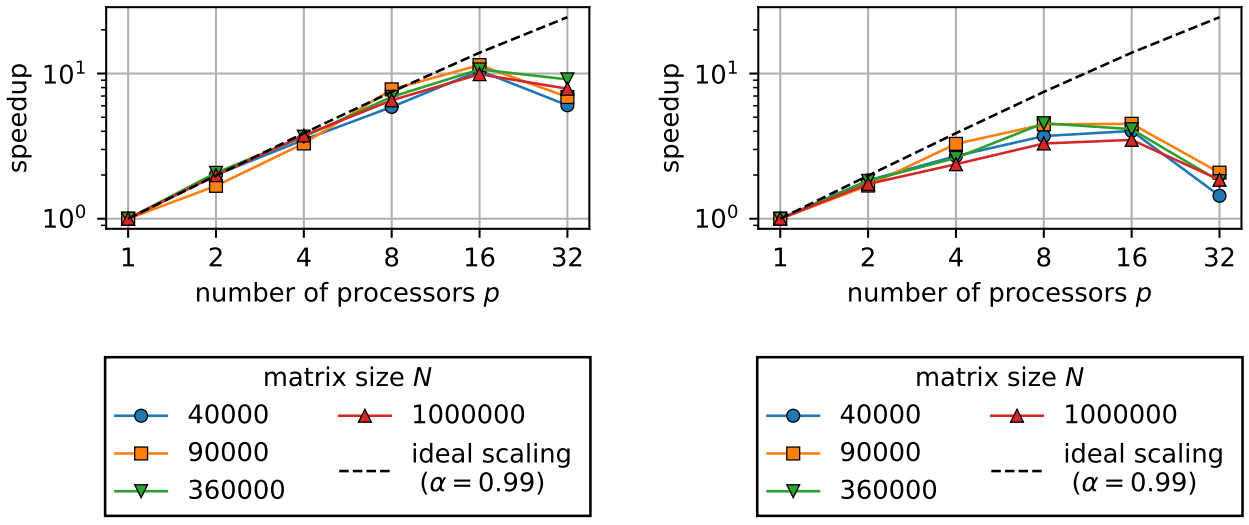


Figure 4: Variant 2: Weak scaling results for optimization flag `-O0`.



(a) `-O0` optimization

(b) `-O3` optimization

Figure 5: Variant 3: Strong scaling results for optimization flag `-O0` (a) and `-O3` (b).

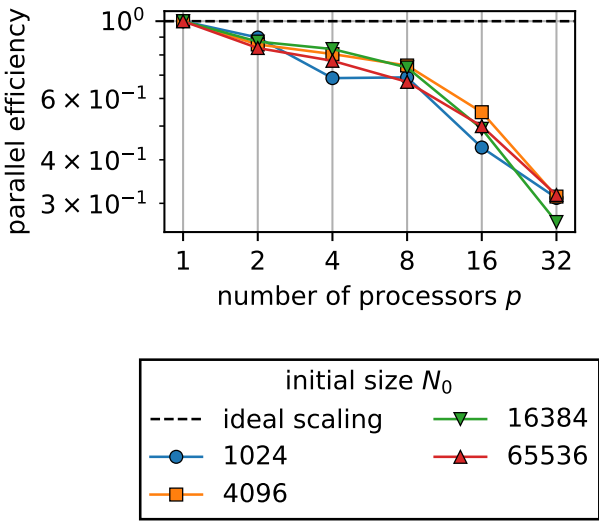


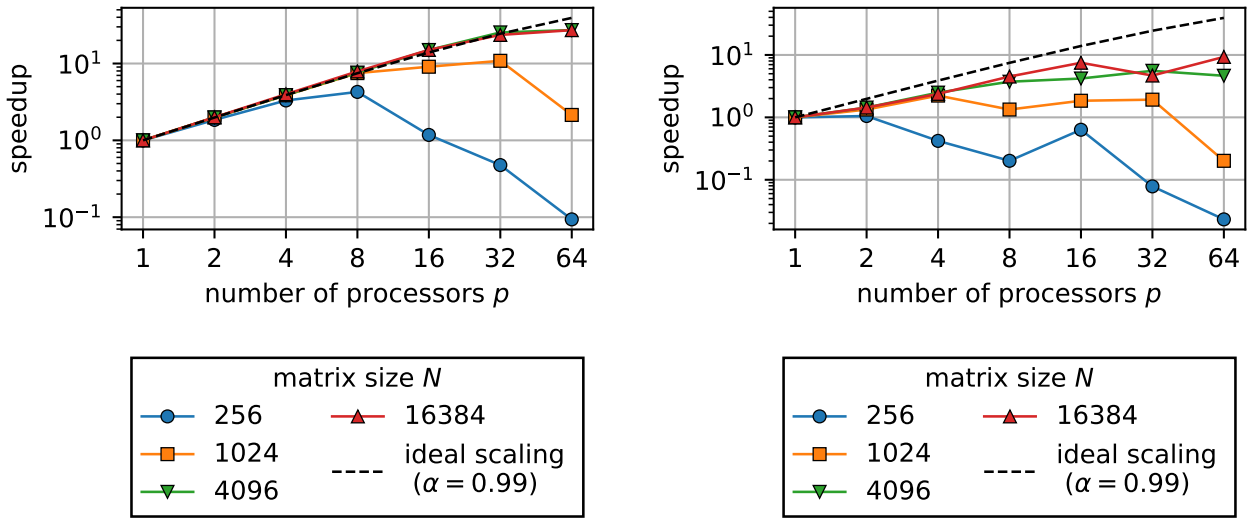
Figure 6: Variant 3: Weak scaling results for optimization flag `-O0`.

Indeed, for the dense case the computational cost per processor is now of order $\mathcal{O}(N^2/p)$ (bigger than the original cost $\mathcal{O}(N/p)$ for sparse matrices), while the communication cost still is $\mathcal{O}(\log p + N)$. Both, dense and sparse cases present identical asymptotic trend $\mathcal{O}(\log p)$ in the case of the strong scaling, however, the dense one presents a better behavior in the pre-asymptotic regime due to its larger computation cost constant term. This effect can be seen in the results in Figure 7, where a better speedup compared to the sparse case is observed for growing matrix sizes. Inevitably, as the theory predicts, and already observed in previous sections, for large enough p, the communication cost will eventually dominate over the computation cost and the speedup will start to deteriorate for large enough p.

The same reasoning developed for the strong scaling of the sparse matrices in Section 2.5 applies in the case: Keeping N constant and growing p, the computation cost decreases as $\mathcal{O}(1/p)$ while the communication cost increases as $\mathcal{O}(\log p)$. However, the constant in front of the computation cost is larger in this case, as it depends on N^2 instead of N, what increases the computation cost, delaying the inflection point. This effect can be seen in the strong scaling results for the dense matrices in Figure 7, where we observe that the speedup is much better than for the sparse matrices for large matrices, being its inflection point delayed respect to p.

Regarding the weak scaling, the analysis is slightly different in this case. In order to keep the computation cost per processor constant, i.e., $\mathcal{O}(N_p^2/p) = \mathcal{O}(N_0^2)$, we have to increase the size of the matrix as $N_p = N_0\sqrt{p}$. In that setting, the communication cost becomes $\mathcal{O}(\log p + N_p) = \mathcal{O}(\log p + N_0\sqrt{p})$, while the computation cost $\mathcal{O}(N_0^2)$ stays constant. Note that in this case the communication cost is smaller as it grows as $\mathcal{O}(\sqrt{p})$, instead of $\mathcal{O}(p)$ for the sparse one (recall Section 2.5.2). This explains why the weak scaling, even if still decaying (as total cost grows with $\sqrt{p}$), presents a significantly better behavior compared to the one observed for sparse matrices (see Figure 8).

Thus, the strong and weak scaling results for dense matrices seem to suggest that, given a large enough N, it seems possible to mask longer the (inevitably) increasing communication costs with sufficiently large computational costs.



(a) `-O0` optimization

(b) `-O3` optimization

Figure 7: Variant 3: Strong scaling results for optimization flag `-O0` (a) and `-O3` (b) for dense matrices.

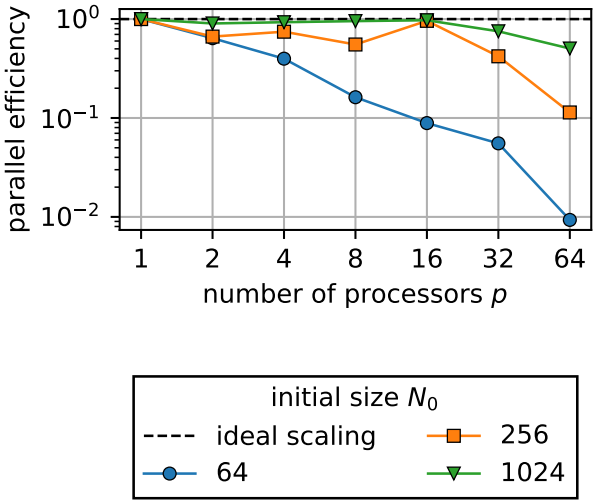


Figure 8: Variant 3: Weak scaling results for optimization flag `-O0` for dense matrices.

5 Conclusion

We conclude that the parallelization of the CG algorithm for sparse matrices in COO format is challenging due to the bad scaling of the communication cost. In particular, we point out that to achieve a good parallel efficiency, the communication costs must stay small compared to the computation cost. This could be potentially achieved by reducing the amount of the data to be communicated, for example by using a more refined partitioning of the matrix and the vectors.