Parallel Computing Project: Parallel Matrix Multiplication*

Massimiliano Viola, 2075219 Leonardo Gemin, 2023860

1 Introduction

Matrix multiplication is a fundamental operation in many scientific and engineering applications, like machine learning, data analysis, and signal processing to name a few. However, it is also a computationally intensive task, and parallelizing it over P processors can greatly improve its performance and scalability. In this project, we study the speedup and efficiency that we can achieve with a simple parallelization strategy to multiply two $N \times N$ matrices based on scattering and broadcasting using MPI. In addition, we also evaluate the effect of further optimizations on the parallel algorithm to try to achieve even greater performance.

2 Problem definition

Given the two matrices $\mathbf{A}$ of size $L \times M$

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,M-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,M-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{L-1,0} & a_{L-1,1} & \cdots & a_{L-1,M-1} \end{pmatrix}$$

and $\mathbf{B}$ of size $M \times N$

$$\mathbf{B} = \begin{pmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,N-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{2,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{M-1,0} & b_{M-1,1} & \cdots & b_{M-1,N-1} \end{pmatrix}$$

the matrix product $\mathbf{C} = \mathbf{AB}$ is defined to be the $L \times N$ matrix

$$\mathbf{C} = \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,N-1} \\ c_{2,0} & c_{1,1} & \cdots & c_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{L-1,0} & c_{L-1,1} & \cdots & c_{L-1,N-1} \end{pmatrix}$$

*Link to GitHub repository: `https://github.com/leonardoGemin/ParallelComputing_UniPd`

such that

$$
\begin{aligned}
c_{i,j} &= a_{i,0}b_{0,j} + a_{i,1} + b_{1,j} + \cdots + a_{i,M-1}b_{M-1,j} \\
&= \sum_{k=0}^{M-1} a_{i,k}b_{k,j} \qquad \begin{cases} \forall i \in \{0, 1, \ldots, L-1\} \\ \forall j \in \{0, 1, \ldots, N-1\} \end{cases}
\end{aligned}
$$

2.1 Naive approach

The naive approach to the matrix multiplication algorithm follows straight from the definition:

Algorithm 1: Naive matrix multiplication

Input: $\mathbf{A}$, $L \times M$ matrix, and $\mathbf{B}$, $M \times N$ matrix
Output: $\mathbf{C} \leftarrow \mathbf{AB}$, $L \times N$ matrix
begin
  for $i \leftarrow 0$ **to** L **do**
    for $j \leftarrow 0$ **to** N **do**
      $c_{i,j} \leftarrow 0$
      for $k \leftarrow 0$ **to** M **do**
        $c_{i,j} \leftarrow c_{i,j} + (a_{i,k} \cdot b_{k,j})$
  return $\mathbf{C}$

It requires $L \cdot M \cdot N$ multiplications and $(M-1) \cdot L \cdot N$ additions of scalars to compute the product of two matrices. In the worst case, when both factor matrices have sizes $N \times N$, its computational complexity is $\mathcal{O}(N^3)$.

2.2 Just a simplification

From now on we will only deal with square matrices of sizes $N \times N$. We will also assume for the description of the algorithm (not in the code!) that P divides N to avoid unnecessary heavy notation.

3 Parallelization strategy

The approach we choose to make these computations faster by using P processors is based on the following strategy:

1. One process is designated as master (e.g., process 0 of `MPI_COMM_WORLD`).
2. The master loads the matrices $\mathbf{A}$ and $\mathbf{B}$, both of size $N \times N$.
3. The master distributes (`MPI_Scatterv`) the rows of the matrix A almost equally into P groups of size N/P (some might have one extra row if N is not a multiple of P) to the other processes while the entire B matrix is replicated (`MPI_Bcast`) in all the other processes.
4. Each process $p \in \{0, 1, \ldots, P-1\}$ computes the product of a stripe of $\mathbf{A}$ of size $N/P \times N$, denoted as $\mathbf{A}(p)$ with $\mathbf{B}$, effectively computing N/P rows of the output matrix $\mathbf{C}(p)$.

5. The master gathers (`MPI_Gatherv`) all the rows from the P processors and fills in the output matrix $\mathbf{C}$, which is returned.

A simple table can help to understand better the phases of the algorithm from the perspective of the master and the worker:

MASTER ($p = 0$)		($p > 0$) WORKER
reads $\mathbf{A}$ and $\mathbf{B}$	1	-
sends $\mathbf{A}(p)$ to the process p	2	obtains $\mathbf{A}(p)$ from the master
replicates $\mathbf{B}$ on each process	3	obtains $\mathbf{B}$ from the master
computes $\mathbf{C}(0) \leftarrow \mathbf{A}(0) \times \mathbf{B}$	4	computes $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$
gathers $\mathbf{C}(p)$ from process p	5	sends $\mathbf{C}(p)$ to the master
writes $\mathbf{C}$	6	-

3.1 Complexity analysis

Differently from the sequential algorithm, now each process computes in parallel only a product of two matrices of size (roughly) $N/P \times N$, which is P times faster than a $N \times N$ product given the observations made in **??**. Therefore, excluding the effect of communication (which will be analyzed in **??**), we expect a complexity of the order $\mathcal{O}(N^3/P)$ for the parallel algorithm, with a speedup of the order of magnitude of P.

3.2 MPI functions

There are three major MPI-related commands in our code: `MPI_Scatterv`, `MPI_Bcast`, and `MPI_Gatherv`. In this subsection, we explain them more in detail.

- The `MPI_Scatterv` function distributes data from one member across all members of a group; it extends the functionality of the `MPI_Scatter` function by allowing a varying count of data, as specified in the `sendcounts` array, to be sent to each process. This makes it possible to handle matrices whose size is not necessarily a multiple of the number of processors, by sending blocks of $\lfloor N/P \rfloor$ or $\lfloor N/P \rfloor + 1$ rows.

```
MPI_Scatterv(*sendbuf, *sendcounts, *displs, sendtype,
             *recvbuf, recvcount, recvtype,
             root, comm);
```

- The `MPI_Gatherv` function gathers variable data from all members of a group to one member; it extends the functionality of the `MPI_Gather` function by allowing a varying count of data from each process. Again, this makes it possible to handle matrices whose size is not necessarily a multiple of the number of processors.

```
MPI_Gatherv(*sendbuf, sendcount, sendtype,
            *recvbuf, *recvcounts, *displs, recvtype,
            root, comm);
```

- The `MPI_Bcast` function replicates data from one member of a group (in our case, the master process) to all members of the group.

```
MPI_Bcast(*buffer, count, datatype,
          root, comm);
```

4 Experiments

To test the goodness of our algorithm, we performed various tests, measuring the performances of the code as the size of the matrices and the number of processors varied. To obtain coherent measurements, all the tests were carried out on the CAPRI[1] server, whose access to computing resources is regulated via the SLURM[2] scheduler.

The variables we have taken into account are

- the size of the matrices: 256, 512, 1024, 2048;
- the number of processors: 1, 2, 4, 8, 16.

For each setup (matrices size and number of processors in use), 10 runs were performed in order to obtain a reliable average of the measurements made. Furthermore, again for each setup, 3 variants of the code have been introduced which we will illustrate below. Finally, each of these setups has been compiled with `-O3` optimization option of `mpicc`. This increases the compilation time but gives important performance gains to the generated code. For consistency, also the sequential baseline has been compiled with the `-O3` of `gcc`.

4.1 Three variants of the same algorithm

Initially, the code was written using matrices (i.e. `M_TYPE a[SIZE][SIZE];`[3]). Subsequently, during the testing phase, we realized that the matrices created in this way could not run at the 1024×1024 size and larger, generating a `segmentation fault` exception. So we have changed the way we handle matrices: we have converted matrices into mono-dimensional arrays in row-major order (for both the sequential baseline and parallel MPI code).

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,SIZE-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,SIZE-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{SIZE-1,0} & a_{SIZE-1,1} & \cdots & a_{SIZE-1,SIZE-1} \end{pmatrix}$$

$$\Downarrow$$

$$\mathbf{A} = (a_{0,0}, \ldots, a_{0,SIZE-1}, a_{1,0}, \ldots, a_{1,SIZE-1}, \ldots, a_{SIZE-1,0}, \ldots, a_{SIZE-1,SIZE-1})$$

This was made through `M_TYPE *a = malloc(SIZE * SIZE * sizeof(M_TYPE))`; in this way, `a[i][j]` is now accessible through `a[(i * SIZE) + j]`. Note that `malloc()` returns an initially null array, so it is not necessary to enter `c[i][j] = 0` before performing the matrix multiplication.

[1] https://capri.dei.unipd.it/

[2] https://slurm.schedmd.com/overview.html

[3] `M_TYPE` is defined in our source code as `unsigned int`. This was done to be able to quickly change the `DATA_TYPE` of all the variables involved. In the header of our source code there are other definitions in order to make the code more flexible to `DATA_TYPE` changes.

4.1.1 First variant

The first variant, called "mono" in the figures, is the parallelized version of the naive approach.

```
for (i = 0; i < sendcount / SIZE; i++)
    for (j = 0; j < SIZE; j++)
        for (k = 0; k < SIZE; k++)
            sendbuf[(i * SIZE) + j] += recvbuf[(i * SIZE) + k]
                                     * b[(k * SIZE) + j];
```

where `recvbuf` and `sendbuf` are, respectively, the chunks of matrices **A** and **C** destined to the current process, while `sendcount / SIZE` is the number of rows of the these chunks.

4.1.2 Second variant

The first improvement was to transpose the matrix **B** before making it available to all processes via the `MPI_Bcast()` instruction. In this way, it is possible to access both the **A** and **B** matrices row-wise and to decrease the cache misses due to the column-wise reading of the **B** matrix. In the figures, this variant has been called "mono + transposed".

```
for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
        bT[(j * SIZE) + i] = b[(i * SIZE) + j];
[...]
for (i = 0; i < sendcount / SIZE; i++)
    for (j = 0; j < SIZE; j++)
        for (k = 0; k < SIZE; k++)
            sendbuf[(i * SIZE) + j] += recvbuf[(i * SIZE) + k]
                                     * bT[(j * SIZE) + k];
```

4.1.3 Third variant

The last improvement, called "mono + $k - i - j$" in the figures, was to change the order of the three loops: instead of $i - j - k$ we adopt the $k - i - j$ order. In this way $a_{i,k}$ (better to say `recvbuf[(i * SIZE) + k]`) is constant in the inner loop, while **B** and **C** (better to say `sendbuf`) are scanned in row-wise.

```
for (k = 0; k < SIZE; k++)
    for (i = 0; i < sendcount / SIZE; i++)
        for (j = 0; j < SIZE; j++)
            sendbuf[(i * SIZE) + j] += recvbuf[(i * SIZE) + k]
                                     * b[(k * SIZE) + j];
```

Note that with this method it is not necessary to transpose matrix **B**.

5 Results

We analyze and plot four important performance metrics: execution time, speedup, scalability, and computing over communicating ratio. This will give a general overview of the behavior of our proposed solutions.

5.1 Execution time

We run the sequential baseline and the first variant of the parallel algorithm (equal in nature for comparison, with the latter just executed in parallel) on matrices of sizes 256, 512, 1024, and 2048. We observe in Fig. **??** that once the algorithm and the number of processors are fixed, both the sequential and parallel algorithm scale compatibly to a cubic law, as expected. Similar patterns hold for the other algorithms as well. The line with label N^3 represents the reference trend in the log scale graph.

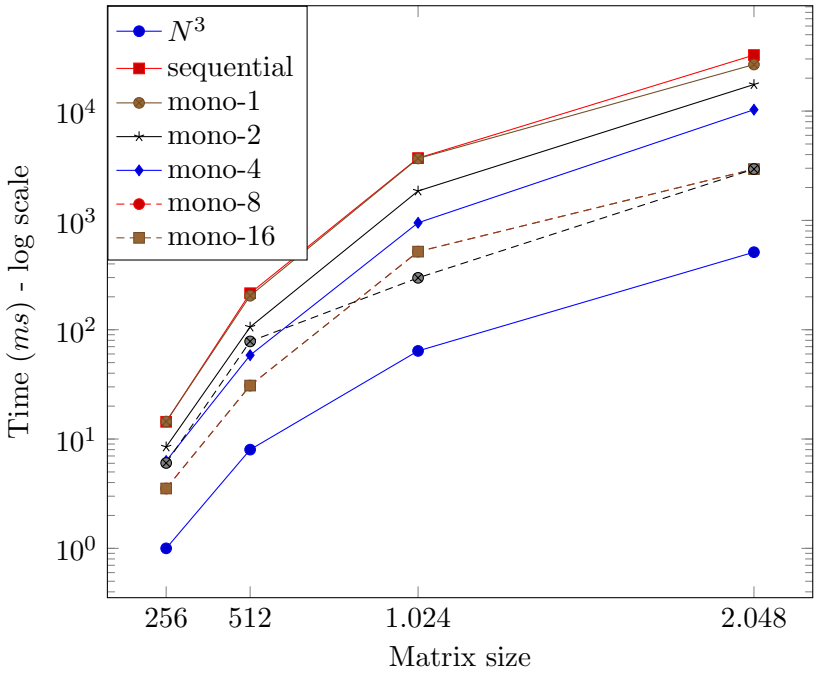


Figure 1: Matrix size vs execution time for the sequential and parallel matrix multiplication algorithms (with an increasing number of processors).

5.2 Speedup

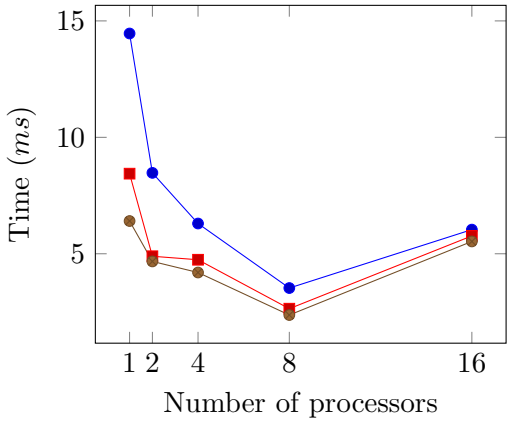
The basic complexity of the algorithm, that is by executing the program without parallelizing the code ($P = 1$), is proportional to the cube of the side of the matrices: $\mathcal{O}(N^3)$. We can therefore say, up to a constant factor, that $T_{base} = N^3$.

Taking advantage of parallelization, we can increase the number of processors that will share the work to lower the execution time. With P processors, as mentioned in section **??**, the complexity per worker is reduced to $\mathcal{O}(N^3/P)$, from which $T(P) = N^3/P$ roughy derives.

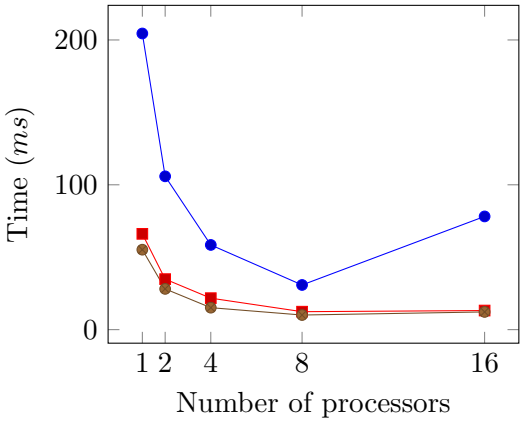
We are now able to calculate the speedup factor:

$$S(P) = \frac{T_{base}}{T(P)} = \frac{N^3}{N^3/P} = P$$

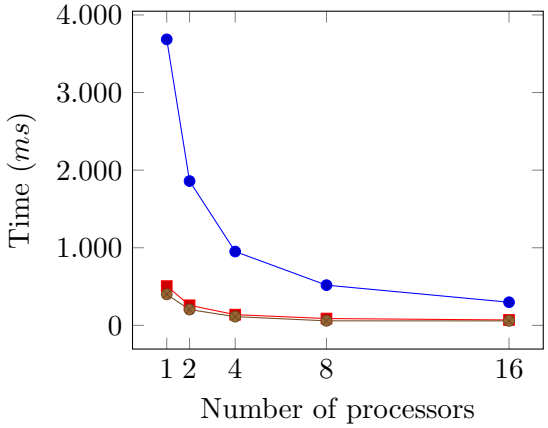
which is optimal in theory. But let's see the behavior in practice. In Fig. **??** we see the execution time for all three parallel algorithms on all input sizes. We should see the execution time decreasing by a factor of 2 every time we double the number of processors, independently of the input size. This doesn't happen for the smaller sizes, but the pattern seems to improve the more we increase the size and give meaningful work to more processors.



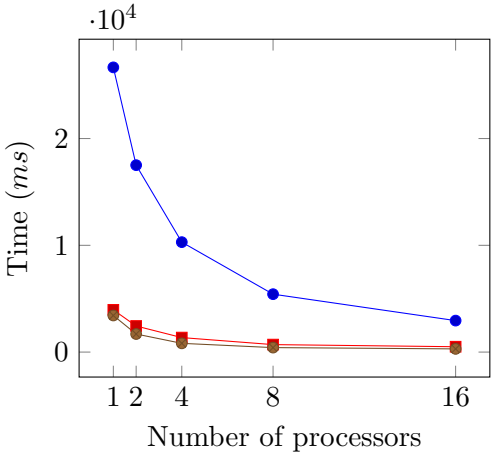
(a) Matrices of sizes 256×256



(b) Matrices of sizes 512×512



(c) Matrices of sizes 1024×1024



(d) Matrices of sizes 2048×2048

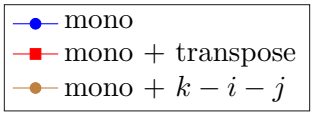
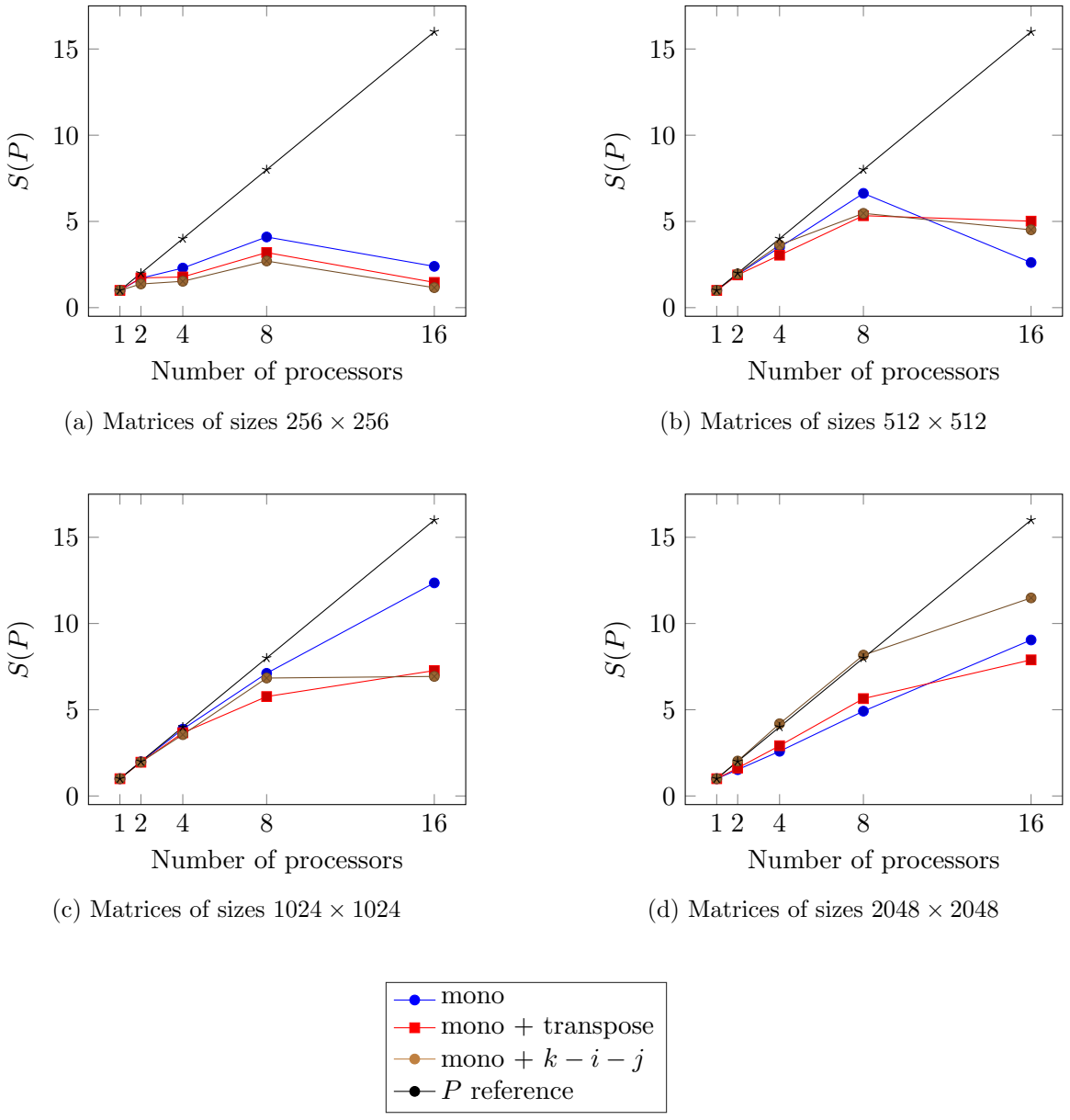


Figure 2: Execution time of parallel matrix multiplication algorithm, divided by size and with increasing number of processors. Lower is better.

By changing the point of view and normalizing each execution time by the base time given by 1 processor for the specific algorithm according to the formula $S(P) = \frac{T_{base}}{T(P)}$, we can directly plot the speedup factors and the reference theoretical linear speedup of P.



(a) Matrices of sizes 256×256

(b) Matrices of sizes 512×512

(c) Matrices of sizes 1024×1024

(d) Matrices of sizes 2048×2048

Figure 3: Empirical speedup of parallel matrix multiplication algorithm, divided by size and with increasing number of processors, versus theoretical speedup of P (black line). Closer to the line is better.

5.3 Scalability

As far as it concerns the scalability of our algorithms, we want to determine whether or not the speedup S(P) given by P processors always increases according to the linear law that

was mentioned before. The answer as we saw is no, and not only the scalability is sublinear, but also the trend is inverted for smaller matrices and a high number of processors (e.g. 256 and 16, or 512 and 16). We believe there is a threshold after which the work assigned to each processor is not meaningful enough to justify their use, or in other words better to use fewer processors at full power rather than executing a small number of operations on a higher number of processors.

5.4 Computing over Communicating ratio

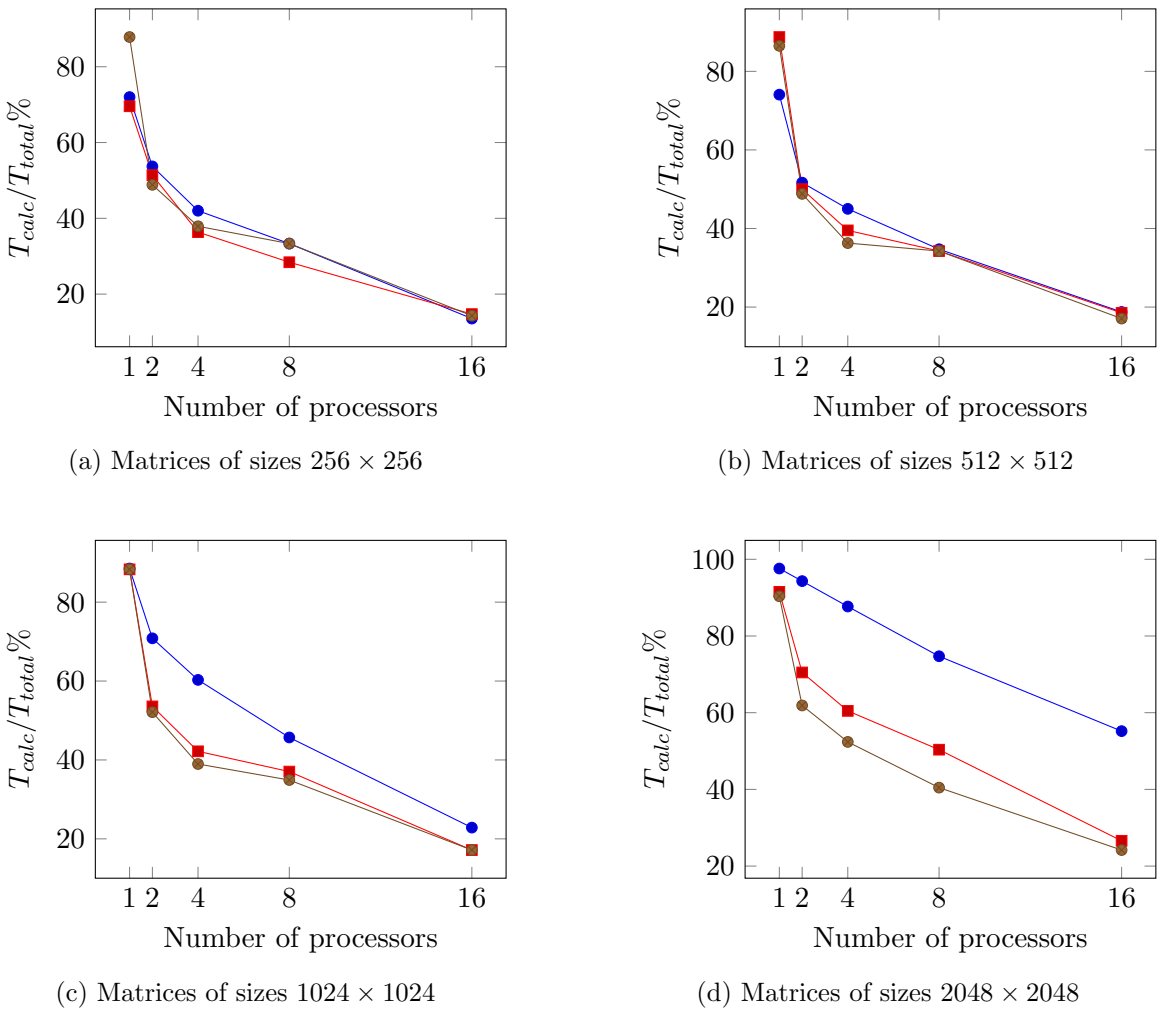
To analyze what could be a reason for the sublinear or even harmful speedup given by a higher number of processors, we can look at the time spent calculating vs communicating. It is clear that with more processors, each of them spends less time calculating but the communication overhead increases and impacts the performance, thus this could explain the degradation after a certain threshold.

We can divide the total execution time of our parallel algorithm into the following parts: *Load*, *Scatter* **A**, *Broadcast* **B**, *Compute*, *Gather* **C**, and *Save*. The communication time is made up of the *Scatter* **A**, *Broadcast* **B**, and *Gather* **C** terms, while the computing part is present in the *Compute* term. Since it is hard to measure accurately all of these quantities, we employ a simpler proxy measure given by the CPU efficiency offered by Slurm, i.e. $T_{calc}/T_{total}\%$.

As we see in Fig. **??**, the effective computing percentage of the algorithms decreases the more processors we use, suggesting communications assume a more important role. We can notice a big waste in CPU utilization for lower sizes and a big number of processors, which is not present as severely for the largest size.

6 Conclusion

In this project, we coded and analyzed different variants of parallel matrix multiplication algorithms, leveraging different levels of optimization from pure parallelism over P processors to reducing cache misses and doing faster loops. We also learned the importance of compiler optimization, which gave huge performance improvements just by adding the `-O3` flag. As for the results, they are kind of in line with expectations: we see that for large enough matrices, increasing the number of processors leads to better execution times; for smaller matrices, using a lot of processors downgrades the performances; using more processors increases the communication time and reduces the CPU utilization.



(a) Matrices of sizes 256×256

(b) Matrices of sizes 512×512

(c) Matrices of sizes 1024×1024

(d) Matrices of sizes 2048×2048

Figure 4: CPU Efficiency of parallel matrix multiplication algorithm, divided by size and with increasing number of processors. Higher is better.