Report Parallel Computing

Nicola Levorato

1. Strassen algorithm

The Strassen algorithm is an algorithm for matrix multiplication. It is faster than the standard matrix multiplication algorithm for large matrices, with a better asymptotic complexity, although the naive algorithm is often better for smaller matrices.

Let A, B be two square matrices of size $n \times n$ where n is a power of 2. We want to calculate the matrix product $C = AB$.

We then partition A, B and C into equally sized block matrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

The difference between the Strassen algorithm and the standard matrix multiplication algorithm is to define new matrices to calculate only 7 products instead of 8.

$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$

$P_2 = (A_{21} + A_{22})B_{11}$

$P_3 = A_{11}(B_{12} - B_{22})$

$P_4 = A_{22}(B_{21} - B_{11})$

$P_5 = (A_{11} + A_{12})B_{22}$

$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$

$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

Finally, C is calculated as following:

$C_{11} = P_1 + P_4 - P_5 + P_7$

$C_{12} = P_3 + P_5$

$C_{21} = P_2 + P_4$

$C_{22} = P_1 - P_2 + P_3 + P_6$

We recursively iterate this division process until the submatrices degenerate into numbers.

The recurrence relation for the Strassen algorithm is:

$$T(n) = \begin{cases} 1 & n = 1 \\ 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 & n > 1 \end{cases}$$

Resolving the recurrence, the time complexity for the Strassen algorithm is:

$T(n) = 7n^{\log_2 7} - 6n^2 = \theta(n^{\log_2 7})$

Instead, the time complexity for the standard matrix multiplication algorithm is:

$T(n) = 2n^3 - n^2 = \theta(n^3)$

From this we can conclude that the Strassen algorithm is asymptotically better than the naive algorithm.

But, due to the lower order term (caused by addition and subtraction of the submatrices) the Strassen algorithm for smaller size of the matrices is slower than the standard algorithm.

1.1 The sequential algorithm

I implemented the sequential algorithm using a recursive function in C.

The algorithm follows the procedure described before and allocate memory for every matrix used in the process and use *free(void *)* function to avoid memory leaks.

To calculate the addition and subtraction between matrices I implemented two different functions that return a pointer to the memory that contain the result of the operation.

Then the products are calculated recursively calling the Strassen function.

The only difference between the Strassen procedure described before is that in my implementation the recursion doesn't proceed until the submatrices degenerate into numbers.

I decide to implement a more optimal version of the Strassen algorithm where the base case apply directly the standard matrix multiplication algorithm only for submatrices of certain size.

To determine the best size for the base case I have executed the Strassen algorithm with different size for the base case and found out that the best value is for $n = 256$.

2. The parallel algorithm

During the development of the parallel Strassen algorithm, I came out with two different ideas:

1. The first exploits every process: the 7 Strassen products are computed by every process using the parallel standard matrix multiplication.
    - At the start the algorithm use one process as master (rank = 0) to compute every intermediate sum or difference of the submatrices to multiply.
    - Then the algorithm computes the 7 products using the parallel standard matrix multiplication algorithm exploiting every process allocated.
    - In the parallel standard matrix multiplication algorithm, the first matrix to multiply is scattered between every process and the second is broadcasted. Then each process executes the standard optimized matrix multiplication algorithm to compute a piece of the result matrix that is gathered into the master process.
    - Finally, the master process can compute the final product matrix C.

2. The second is more straightforward: using different processes to compute the recursive calls of the Strassen algorithm.

Since it's impossible to have so many processes to compute every recursive call I just stopped to the first level of recursion.

- At the start the algorithm use one process as master (rank = 0) to compute every intermediate sum or difference of the submatrices to multiply.
- Then the master sends these submatrices to 7 different slave processes. Each of these processes apply the recursive Strassen algorithm to compute one product.
- Then each slave process sends to the master the result.
- Finally, the master can compute the final C product.

I also developed an alternative version where there are only 4 processes and 3 of them (included the master) compute two products and one process compute only one product.
The second idea suffers of one major problem: if the number of processes is more than 8 some processes are not used during the computation and so they are wasted. One could develop a version that go further in the recursion tree and compute 49 matrix multiplication but if I have more processes the problem persists.

2.1 Complexity analysis

The complexity analysis is done based on the work of the master process.

1. In the first algorithm:
    - The master process partition A, B into 4 blocks each. This requires $(\frac{n}{2})^2$ operations.
    - Then the process compute 10 summation/subtraction that requires $10(\frac{n}{2})^2$ operations.
    - Then each process compute $P_1, P_2, P_3, P_4, P_5, P_6, P_7$ using the parallel matrix multiplication. Each multiplication requires $\frac{(\frac{n}{2})^3}{p}$ operations per process.
    - Finally, the master computes 8 summation/subtraction and executes $(\frac{n}{2})^2$ operations to loop on the block submatrices to compose the C matrix.

$$(\frac{n}{2})^2 + 10(\frac{n}{2})^2 + 7\frac{(\frac{n}{2})^3}{p} + 8(\frac{n}{2})^2 + (\frac{n}{2})^2 = \frac{7}{p}(\frac{n}{2})^3 + 20(\frac{n}{2})^2 = \frac{7}{8p}n^3 + 5n^2$$

2. In the second algorithm:
    - The first two points of the analysis are the same.
    - Then the master process sends the summation/subtraction submatrices to all processes that apply the Strassen algorithm to compute the matrices $P_1, P_2, P_3, P_4, P_5, P_6, P_7$. In this step each process requires $O((\frac{n}{2})^{\log_2 7})$ operations.
    - The final point is the same to the first algorithm analysis.

$$(\frac{n}{2})^2 + 10(\frac{n}{2})^2 + O((\frac{n}{2})^{\log_2 7}) + 8(\frac{n}{2})^2 + (\frac{n}{2})^2 = O((\frac{n}{2})^{\log_2 7}) + 5n^2$$

Looking at the two analysis the second one seems to be asymptotically faster but if the number of processes p is high enough, the first algorithm is the best.

The two ideas have some pros and cons:

First idea:

- **PROS**: exploits every process.

- **CONS**: with the same number of processes is slower than the second idea.

Second idea:

- **PROS**: faster using a fixed number of processes (like 8 or 49).
- **CONS**: using more processes some of them can be wasted.

3. Experimental setup and results

In order to get uniform results across all the different execution in my experimental setup the matrices A and B are always the same where $A[i,j] = i + j$ and $B[i,j] = i - j$ and each value is a float.

Also, since my algorithms use the naive multiplication algorithm (sequential or parallel version) I tried different optimized versions to obtain the fastest possible algorithm since the performance of the final result depends a lot on the performance of the standard algorithm.

Between all the possible combination of the standard matrix multiplication algorithm I tried:

1. i-j-k loop: this is the standard loop and also the slowest.
2. i-k-j loop: this combination is cache friendly and also one of the fastest.
3. Transposition: transposing the matrix B can obtain good performance but is memory consuming.
4. Loop unrolling: using loop unrolling can boost the performance of the code.
5. SIMD instructions: I tried SIMD instructions (code in the slide 42 of pdf CP05_Matrix_multiplication_and_Optimizations) on CAPRI and found out that the performance is similar to the i-k-j loop.
6. Chunked version: partitioning A, B, C into sub-matrices for better use of the L1 cache is the fastest version of the standard matrix multiplication algorithm that I was able to find.

Using the chunked version of the naive multiplication algorithm I was able to boost the performance of my parallel version of the Strassen algorithm a lot.

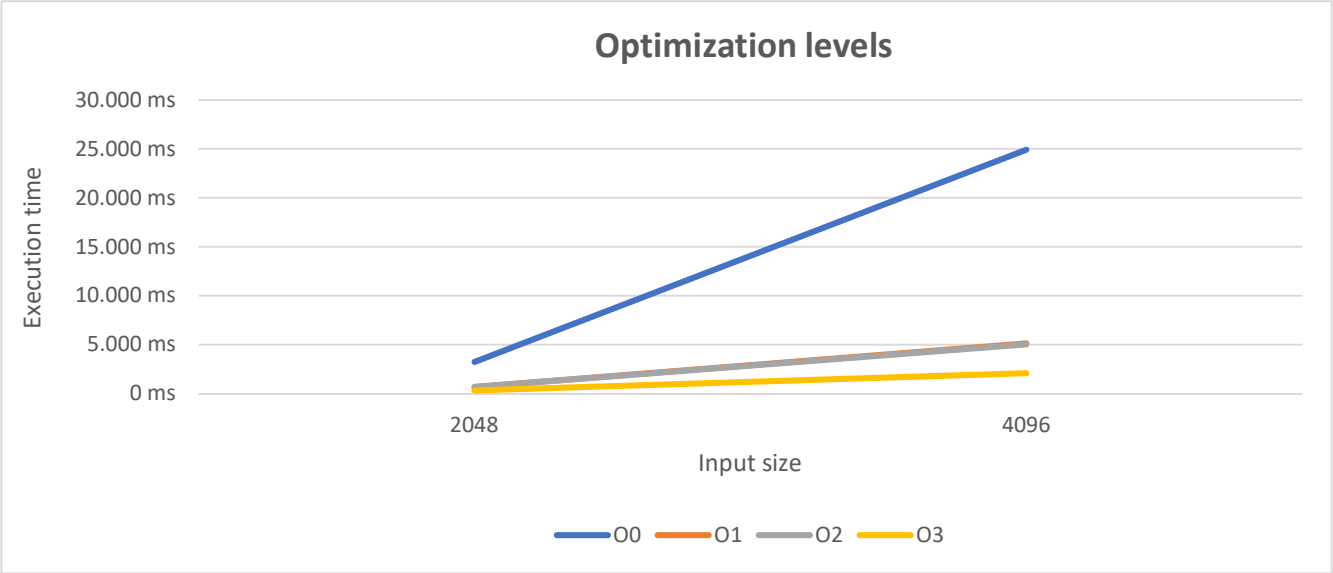
The performance metrics measured are:

1. **Execution time**: time to execute the whole task.
2. **Communication time:** time to exchange data between processes.
3. **Speedup factor:** ratio between execution time of the sequential algorithm and parallel algorithm using p processes.
4. **Scalability:** execution time as input size increases.

To evaluate the performance on the CAPRI infrastructure I tested the parallel algorithm (mainly the first version between the two described in the section 2) varying the number of processes (4, 8, 16, 32 and 64 processes) and the size n of the matrices (where n is the number of elements for each row/column and a power of 2).

3.1 Optimization levels

I tested all different compiler optimizations (-O0, -O1, -O2, -O3, -Ofast) and I found out that regardless the number of processes or the size of the input the best flag is -O3.



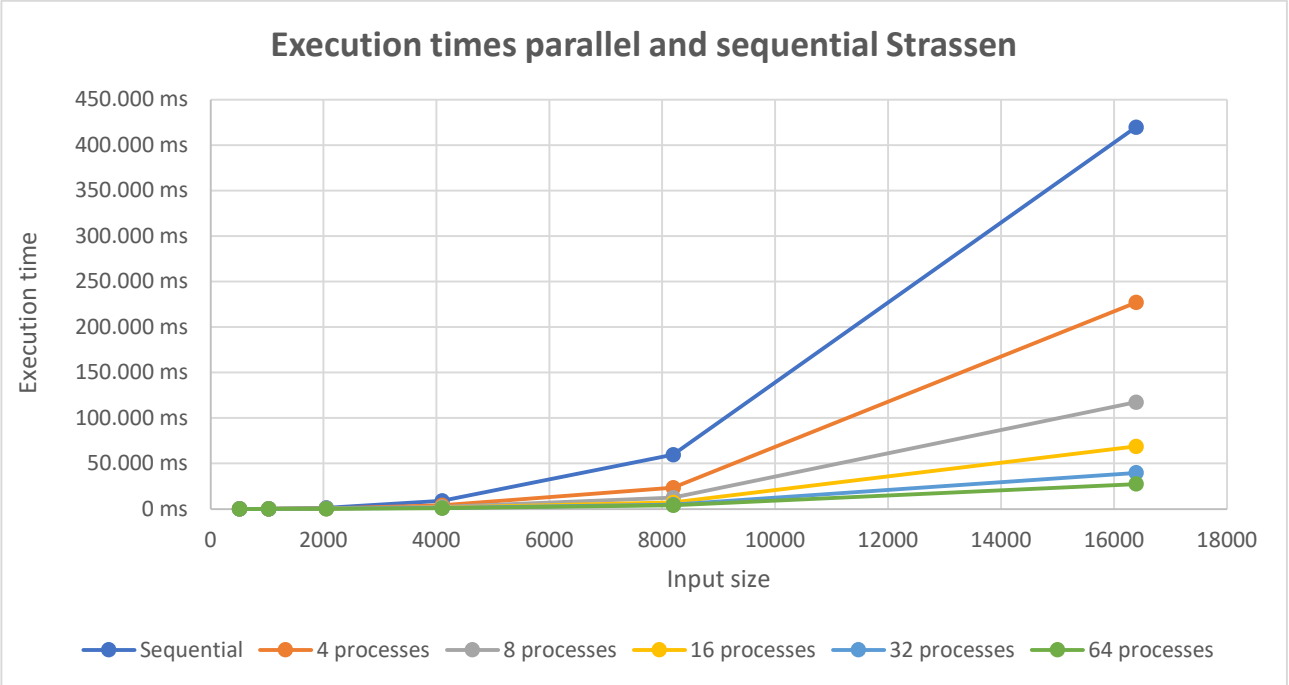
This chart shows that applying all different compiler optimization on the parallel Strassen optimized algorithm using 8 processes for both sizes $n = 2048$ and $n = 4096$ the best flag is -O3.

The flags -O1 and -O2 have similar performance and the flag -O0 is the one that perform the worse.

3.2 Execution time

The first metric I have measured is the execution time.

The following chart compares the execution times of the parallel Strassen algorithm varying the number of processes and the input size of the matrices.



Looking at the chart, the algorithm has similar performance on small sizes of the input ($n \leq 2048$) but on larger instances the algorithm performs better using more processes.

This happens because for small sizes the computation is faster, and the execution time of the parallel algorithm is mostly dominated by the communication time. Therefore, for small sizes is almost always better just use the sequential version of the algorithm or also is good to use the standard matrix multiplication optimized algorithm that is a lot less memory consuming.

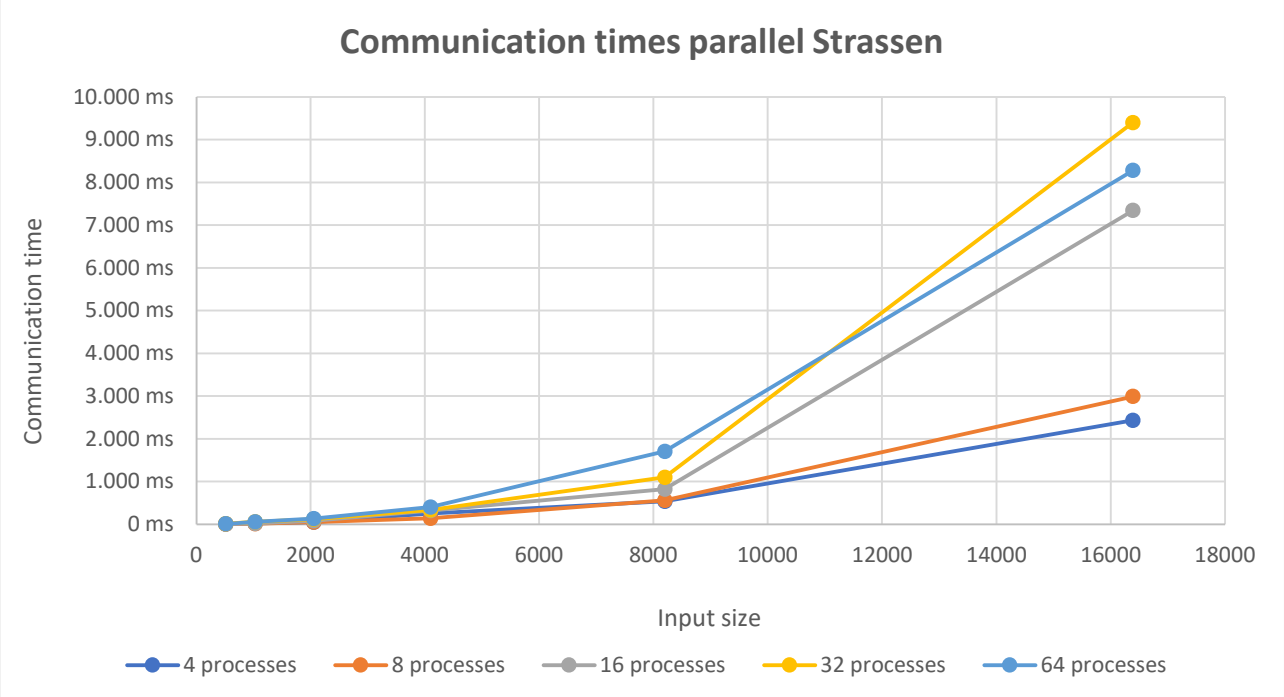
Increasing the size of the input, the parallel version is much faster than the sequential version since the execution time is dominated by the computation time. In fact, in this case the computation is split between the processes.

3.2 Communication time

The following charts compare the communication times of the parallel Strassen algorithm varying the number of processes and the input size of the matrices.

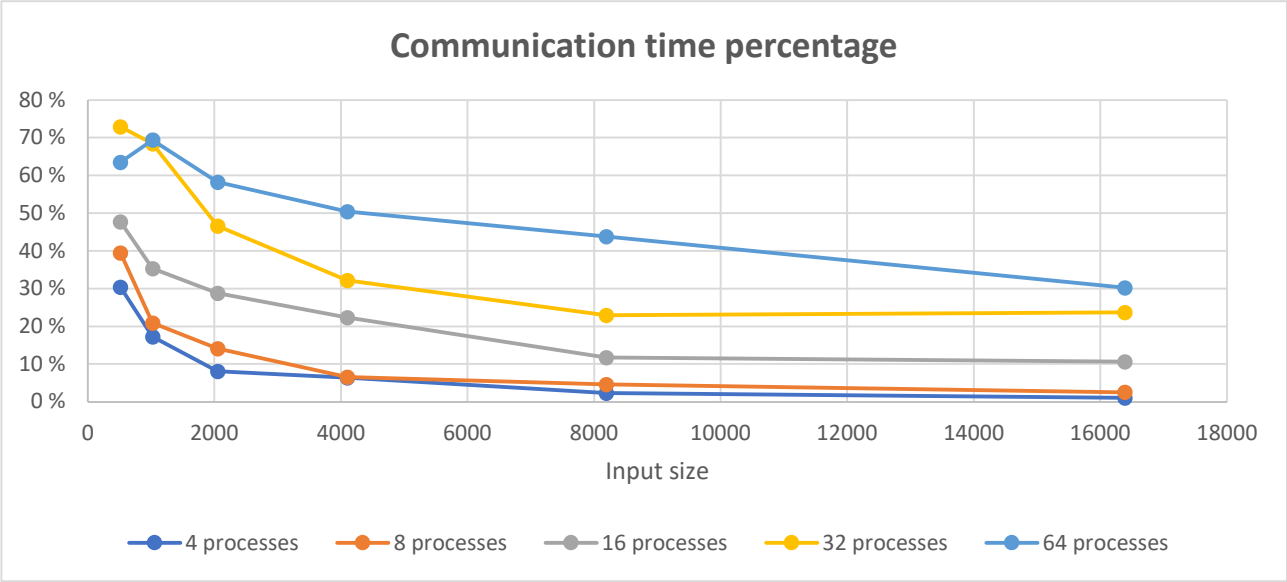
In the first chart increasing the input size and the number of processes the time took to exchange data between all processes is higher.

This happens because if the size of the input increase, there are more data to exchange, and if the number of the processes increase there are more exchanges.



The second chart compares the communication time percentage calculated as:

$$\%\, Comm = \frac{communication}{Useful\ work} \qquad \text{where} \quad useful\ work = communication + computation$$



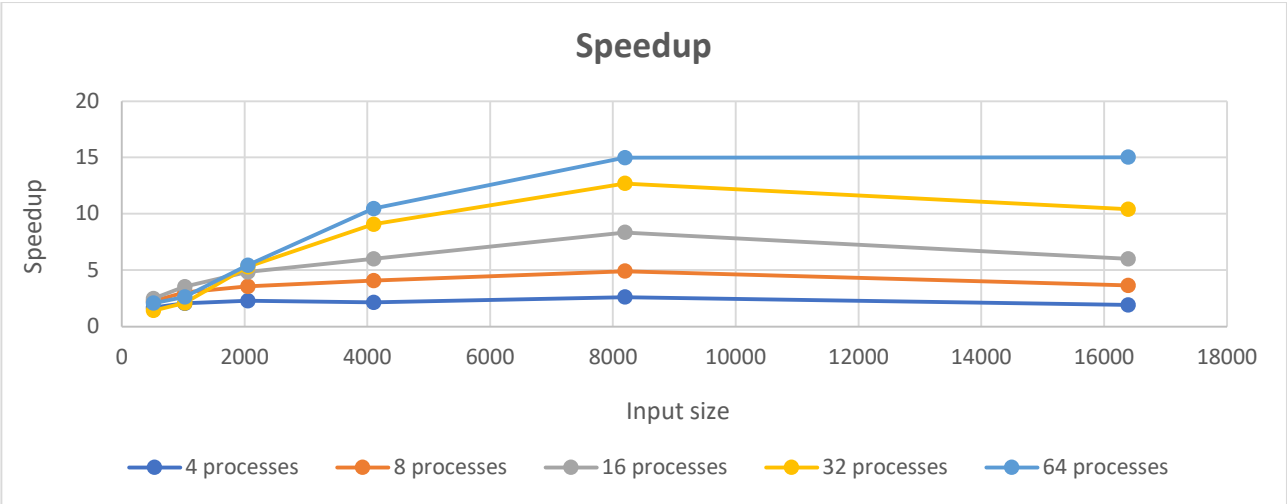
Looking at the chart there is a bottleneck due to communication.

For small sizes, as said before, since the computation is faster if the number of processes is high the execution time is mostly dominated by the exchange of data.

For largest sizes the bottleneck is reduced but using more processes lead to a high percentage of the communication time.

This happens because there are more processes that exchange data and since the matrices to multiply have a smaller size the computation time is smaller. So, this lead to a lower computation time but to a higher communication time and consequently the percentage is higher.

3.3 Speedup



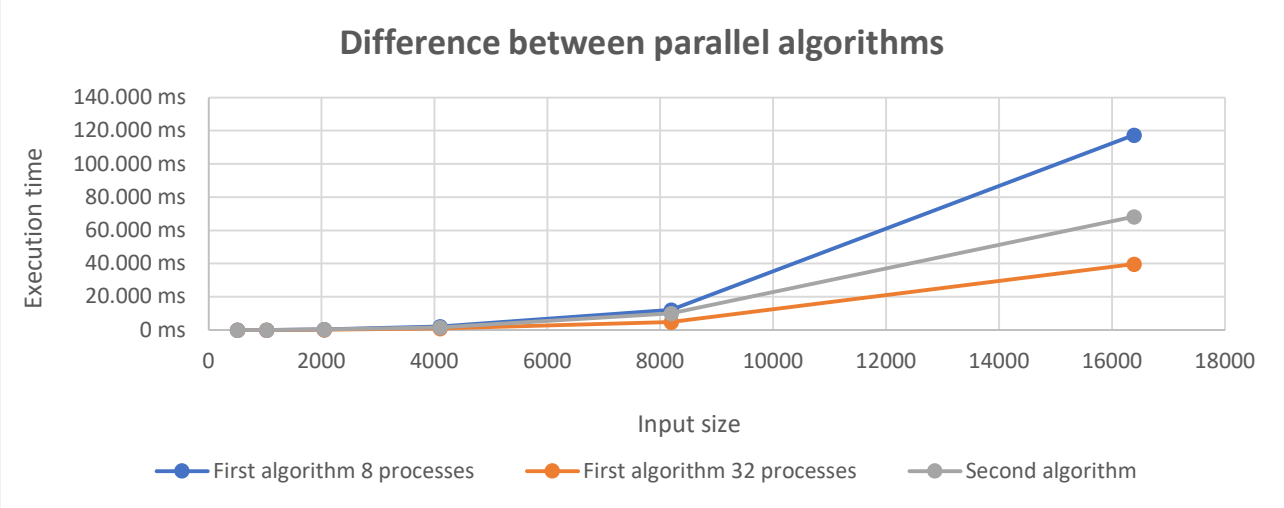
The above chart compares the speedup, based on the input size and the number of processes, calculated as: $S_{up}(p) = \frac{T(1)}{T(p)}$ where $T(1)$ is the execution time measured on the sequential algorithm and $T(p)$ is the execution time measured on the parallel algorithm with p processes.

Looking at the chart, for the same input size, using more processes the speedup increase since the execution of the parallel algorithm is faster.

Ideally using p processes the best value for the speedup is p and so all the graphs based on the number of the processes should be a straight line parallel to the abscissa axis.

This doesn't happen because for small sizes of the input the sequential algorithm is really fast, and the parallel algorithm is dominated by the communication time and so the speedup is really low. Increasing the size, the speedup increases but it's still lower than the ideal value mainly due to the communication bottleneck.

3.4 Differences between the two solutions



Finally, in this chart I compared the two algorithms described in section 2.

Asymptotically the second algorithm is better and in fact using the same number of processes the algorithm performs better than the first one but increasing the number of processes the first algorithm become the fastest.

4. Conclusions

In conclusion, parallelizing the Strassen algorithm is a hard task since the problem require different summation/subtraction of submatrices to be exchanged between the processes. Based on the number of processes or the input size can be better to use one implementation instead of another.

In my implementations the results obtained are good using a lower number of processes but with a higher number of processes ($p \geq 32$) and the same input size the speedup tends to increase slower due to the communication bottleneck.

One further step can be to implement a better data exchange strategy to reduce the bottleneck.

5. Bibliography

https://it.wikipedia.org/wiki/Algoritmo_di_Strassen

https://en.wikipedia.org/wiki/Strassen_algorithm

https://medium.com/swlh/strassens-matrix-multiplication-algorithm-936f42c2b344

https://vaibhaw-vipul.medium.com/matrix-multiplication-optimizing-the-code-from-6-hours-to-1-sec-70889d33dcfa

Repository GitHub with the source code: https://github.com/nico9779/Progetto-Calcolo-Parallelo