**Report Parallel Computing**

Nicola Levorato

**1. Strassen algorithm and the sequential 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 be two square matrices of size where is a power of 2. We want to calculate the matrix product .

We then partition and into equally sized block matrices:

, ,

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.

Finally, is calculated as following:

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

The recurrence relation for the Strassen algorithm is:

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

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

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

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.

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 loop on every element of the matrices. 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 .

**2. The parallel algorithm**

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

1. The first solution exploits every process: at the start the algorithm use one process as master to compute every intermediate sum or difference of the submatrices to multiply. After this step the algorithm compute 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 processes and the second is boadcasted. 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.

After calculated each multiplication the master process can compute the final product matrix .

1. The second is more straightforward: to compute every recursive call the algorithm use different processes.

Since it’s impossible to have so many processes to compute every recursive call I just stopped to the first level of recursion. So, I used one process as master process that compute every intermediate sum or difference of the submatrices to multiply and then send these submatrices to 7 different processors. Each of these processors apply the recursive Strassen algorithm and then send to the master the result to compute the final product.

I also developed an alternative version where there are only 4 processes and 3 of them compute two products and one process compute only one product.

This algorithm 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 first the master process partition into blocks. This requires operation. Then the process compute 10 summation/subtraction that requires .

Then each process compute using the parallel matrix multiplication. Each multiplication requires operations per process.

Finally, to compute the master process 8 summation/subtraction and a operations to loop on the block matrices and compose the matrix.

1. In the second algorithm the first part of the analysis is similar. Then the master process sends the summation/subtraction submatrices to all processes that apply the Strassen algorithm to compute the matrices . In this step each process requires operations. Also, the final part is similar to the first algorithm analysis.

Looking to the two complexity analysis the second one seems to be asymptotically faster but if the number of process is high enough the first algorithm is the best.

**3. Experimental setup and performance metrics**

In order to get uniform results across all the different execution in my experimental setup the matrices and are always the same where and and each value is a float.

Also, since my algorithms use the standard matrix multiplication algorithm (sequential or parallel version) I tried different combination 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 obtain good performance but is memory consuming;
4. Loop unrolling: using loop unrolling can boost the performance of the code;
5. SIMD instruction: trying SIMD instruction as the code in the slides on CAPRI I found out that the performance are similar to the i-k-j loop;
6. Chunked version: partitioning 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 standard matrix multiplication algorithm I was able to boost the performance of my parallel version of the Strassen algorithm a lot instead of using the others code versions.

The performance metrics measured are:

1. Execution time
2. Communication time
3. Speedup factor
4. Scalability

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

I tested all different compiler optimization (-O0, -O1, -O2, -O3, -Ofast) and I found out that regardless the number of processors 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 and the best flag is -O3.

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

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 () 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 splitted between the processes.

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.

In the second chart is compared the communication time percentage calculated as:

where

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 led to a lower computation time but to an higher communication time and consequently the percentage is higher.

The above chart compares the speedup, based on the input size and the number of processes, calculated as: where is the execution time measured on the sequential algorithm and is the execution time measured on the parallel algorithm with 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 processes the best value for the speedup is and so all the graphs based on the number of the processes should be 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.

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,

**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>