Report Homework 2

Alex Pegoraro 227642
alex.pegoraro@studenti.unitn.it

November 22, 2023

Link to Git repository: `https://github.com/AlphaNightLight/parallel_homework2`

1 Parallel matrix multiplication

1.1 Introduction

The first exercise of the homework asks us to implement an algorithm to multiply two compatible matrices while measuring the time spent to perform such an operation. There is the requirement to be able to handle efficiently both dense and sparse matrices, so I implemented two algorithms: one for the general case of a dense matrix, and another one for the special case of a sparse matrix.

The next step is to use OpenMP to parallelize the codes in a shared memory fashion. To measure the performance I extracted information such as the strong and weak scaling, the speedup, the efficiency and the FLOPS.

1.2 Description and Methodology

1.2.1 The sequential program for dense matrices

When talking about matrix product, the first solution that comes to mind is the classical triple `for`, and that's exactly the solution I implemented. However, to facilitate the benchmark and the code reusability, `matMul_dense.cpp` is much more than this.

I opted for an object oriented approach in which matrices are struct that contain their values as well as other useful information like the number of rows and columns. I also created a struct to return a matrix altogether with the time needed to compute it, to bypass the restriction that C++ functions can return a single value. Furthermore I created service functions for allocation, deallocation and debug print of matrices.

The `main` function has the duty to produce the report file. To do this, it loops over three possible matrix configurations whose parameters are set with a `switch` of the loop index. To reduce fluctuations of performance, I also decided to retest the same configuration many time and perform an average.

That said, the `matMul` function consist of an `if` statement that checks the compatibility of its input matrices and in the case perform the multiplication with a triple `for`. To measure the time I used the `chrono` library of C++. The time samplers are put right before and right after the loops, to follow the golden rule that prints and allocations should be kept outside of the measurement.

1.2.2 The sequential program for sparse matrices

To create a program able to efficiently manage sparse matrices, I browse the internet looking for sparse matrix algorithms. I found an interesting representation standard on Wikipedia called CSR [1]. It represent matrices as a three vectors: one holding the nonzero values, another one for their column index, and a third vector that identifies row slices over the first[1].

Thanks to the object oriented programming, the `main` function is exactly the same as before[2], I just had to change the struct and the functions. Unfortunately, I didn't find an algorithm for matrix multiplication in CSR format, but I found a solution in this site [2] for sparse matrices represented in a quite similar standard, that helped me figuring out an implementation.

The keypoint, is to convert the second matrix in CSC format, i.e. a format that store column slices instead of row slices. To easily achieve such a conversion, it's crucial to observe that the CSC representation of a matrix A, is equivalent to the CSR representation of A transposed. Hence, the algorithm is essentially the same as the one for transposition showed in section 2.2.2, that's why I won't talk about it in this section.

Once we have the second matrix stored by columns, the approach is quite similar to the dense case. We iterate over the cells of the result matrix, and calculate a vector product based on the current indices. The main difference, is that in this case the product will be between sparse vectors, and so the innermost loop will be a while instead of a for. Additionally, we have to push the resulting value in the result if and only if it's different from zero.

1.2.3 The parallel program for dense matrices

For the parallel code, it's not enough to create a single report file, but we need a file for strong scaling and a different one for weak scaling (see section 1.3.2). This is not the only modification to our `main` function, as OpenMP needs to be compiled with `-fopenmp` flag to work properly. We can verify this condition using an `#ifdef` statement that encloses the sensible parts of our code, like the library inclusion and the main `for` loop. I passed form the `chrono` library to `omp_get_wtime` for time measurements, to ensure we're measuring the wall time and not the CPU time.

One last preliminary consideration is that we need to know the number of threads we are working with. I noticed that the function `omp_get_num_threads` returns the number of active threads, not the number of available ones. As at the begin of the code we are in an inactive parallel region, this number is useless. So, I bypassed this problem simply accessing the environment variable `OMP_NUM_THREADS` thanks to the `getenv` function.

The part of the code we want to parallelize is of course the `for`. With OpenMP this can be very easily achieved with `#pragma omp parallel for`. Furthermore, the two loops that iterate over the rows and the columns of the result are independent from each other, that mean we can parallelize both adding the clause `collapse(2)`. The innermost loop could be parllelized, but doing so would lead to a too fined grained data partitioning, that would cause a degradation in the performance as the overhead required to synchronize all the thread in the `reduction` of the value would be greater than the achievement in performance, so I decided to leave it sequential.

A crucial decision is the visibility of the variables. i, j and k must be private as the formers are cycle variables and the latter is modified by every thread. A, B and depth can be shared, as they are only read[3], and C can be shared as all the threads access it in different cells.

[1] I decided to implement the first two as `std::vector` as we don't know their size a priori, while the last vector can be implemented as a normal array as its size is equal to the row of the matrix plus one.

[2] Except for the fact that we also iterate over different values of density

[3] Read a variable causes problems only if another thread is trying to write it at the same time, but if we have only

I also added the `schedule(static)` clause. It would have been really nice to try both the static and the dynamic schedule for all the codes, but I realized that I already had too many plots, and so I took the decision to select a particular schedule for every parallel code instead of systematically benchmark both the options. in this case, I opted for a static schedule, as for a dense matrix every block will take almost the same time to be calculated.

1.2.4 The prallel program for sparse matrices

Unlike the code for the dense case, with sparse matrices the situation is not embarrassingly parallelizable. In particular remind that the code is composed of two main tasks: convert the second matrix to CSC, and make the effective product. Again, the former is the same task as the matrix transposition, and so I redirect to section 2.2.4 for its parallelization.

The most obvious thing to notice, is that the innermost loop can't be parallelized, as it is a while. However, also the middle loop can't. The reason relies in the CSR format. When we calculate a row of C, we examine each of its element, and then decide whether push it in the row or not. Hence, parallelize that loop would mean to concurrently push elements into the same row, that would result at least in a scramble.

On the other hand, in CSR we can access different rows simultaneously, as for them we have indexes instead of a push. The problem, is that this would be the case if we could be able to clearly identify each row into `vals`, but in a matrix product we don't know a priori how many elements will the previous lines have. So, it seems we are again back to a sequential constraint.

In fact we are, but I decided to try the following experiment: each thread calculates in parallel a row of C in its own `local_C_vals`, and after that we impose an `ordered` clause to impose a sequential pushing of the rows. The hint is that calculate an entire row is intuitively slower that push it in the common vector, and so it can be possible that the overhead due to the ordered clause is inferior to the gain of have multiple lines calculated in parallel.

The schedule for such a `for` will be dynamic, as for sparse matrices we have no Idea of how the work is distributed among the rows. The logic to decide if a variable must be shared or private is almost the same as for the dense case.

1.3 Experiments and Benchmarks

1.3.1 The Automation Scripts

To efficiently handle the benchmark process, I designed some automation scripts. First of all, there is the `compile.sh` scripts. It creates two "bin" folders, one for parallel and one serial codes, and puts into them the executable files generated by the compilations. Of course, parallel OMP codes are compiled with the `-fopenmp` flag. I also added the `-std=gnu++11` flag to allow the usage of C++11. In my local machine it was not necessary, but the cluster compiler complained, so I had to add it.

Then, there is the `benchmark.sh` script, that first of all creates the "report" directory with its subfolders. After that, it takes the duty to run every single code, after it has initialized the correspondent report files. Furthermore, for the parallel algorithms it does not just execute them once, but it re-execute them several times increasing the number of cores.

Then, there's the `prepare_plot.sh` script. It does nothing more than create a directory hierarchy rooted in "plots". It is needed because the next script, `plot_results.m` require such a

reads there are no problems, even if they are concurrent.

structure to be able to put the plots in the right place. This last is a GNU Octave script that reads the `.csv` report files, and convert them into a graphical format.

`clean.sh` is a script to remove the report files as well as the binaries. It has the purpose of clean your workspace after you have created the plots (that in fact are not canceled). The last script, is `run.pbs`, that is needed when submitting a job to the cluster. Apart from the cluster setting operations, what it actually do is simply to run the sripts for compilation and benchmarking. I also used it to get some information about the node in which the program was running, through a `lscpu` command. You check these characteristics having a look in the `outputs.o` file.

Summarizing, the expected workflow is: run `compile.sh` and `benchmark.sh` (by hand or via `run.pbs`, depending if you are on your local machine or the cluster) to get the report files, then run `prepare_plot.sh` and `plot_results.m` to get the plots, and finally run `clean.sh` when you no more need the `.csv` files.

1.3.2 The Plots

When aggregating the result into the plots, I took care of mostly four aspects: Strong Scaling, Weak Scaling, Speedup and Efficiency. The Strong scaling consist in running the same parallel program many times, with a fixed input size but an increasing number of threads. What we expect to see is that the amount of time required to complete the task decreases almost linearly while the thread count increases. In the plot, I showed on the x axis the number of available cores in a logarithmic scale, and on the y axis the execution times. I also added to the plot the time of the serial code as an horizontal line. This of course doesn't mean that the serial program uses more threads, I plotted it like this just to have a wide comparison.

A quite similar concept is the one of weak scaling, in which we run the same program many times with an increasing number of threads, but unlike the previous case we also increase the dimension of the input. Again, as a comparison I took the serial code run with the dimension used for a single thread, plotted as an horizontal line.

The other two concepts are not extracted from the raw data, but are obtained performing some analysis of the Strong Scaling Data. Speedup, is defined as the ratio between the time of the parallel code run with one thread and the time of the parallel code run with N threads. This time, as a comparison I reported the ideal speedup: the line with slope 1, that in the plots looks as an exponential as we are in a logarithmic scale plot.

The last concept, is efficiency. We can obtain it from the previous concept, dividing the speedup of a certain run by the number of thread used in such a run. It is in a sense a normalized version of speedup. I decided to represent it in percentage, and I reported also the ideal efficiency: the 100% horizontal line.

1.3.3 The Chosen Parameters

The asymptotic cost required to multiply a $M \times N$ matrix to a $N \times K$ matrix, is $O(MNK)$. That's why in my tests I decided to let the product of this three number to be a constant. What I changed, is how the elements are arranged. In the first case, I computed a matrix that is smaller than the two inputs, but each of its cells requires a large vector product to be computed. The second case is exactly the opposite: the output matrix is bigger but it requires products of small vectors. The Third case, is a middle way, as I selected an arrangement that is as similar as possible to a square matrix. I called this cases "deep", "shallow" and "square" respectively.

For the Weak Scaling It's also required to increase the dimension of te matrices. I implemented this requirement simply adding an `if` statement that in the case of the Weak Scaling iterations

multiplies the row dimension by the number of threads, obtained with `getenv`.

1.4 Observations and Conclusions

1.4.1 Plot Analysis

If you have look in the plot folder, you will see that I computed tons of plots. So, for the sake of simplicity I inserted in this report only one representative input shape per algorithm[4], that in this case happens to be the "square".

Having a look at the Strong Scaling plot for dense matrices in figure 1, we can observe that the parallel algorithm start a little bit slower than the serial one, but then becomes faster and faster as expected. The reason fo this is that parallelism comes with an overhead, and in the case of a single thread it is not balanced by the benefits of having more cores, and so it's perfectly reasonable to have the mono-thread parallel code slower than the serial one.

The same reasoning applies to the Weak scaling. It's true that the parallel program is slower than the serial one, but remember that in such a plot we are more interested in discover how the algorithm deals with bigger inputs, and so the important fact is that the line is almost horizontal as expected. Also remember that the serial result plotted has been computed with an input size equal to the smallest among the ones faced by the parallel code.

Speedup and efficiency, allow us to understand how much we gain when we parallelize a code. It is in particularly easy to see that the more threads we add the more the efficiency decrease. This means that we are obtaining smaller times, but not so small as the ideal ones. This behavior is absolutely normal in real world programs.

For sparse matrices (figure 2), the first thing that catches the eyes, is the obvious fact that the denser is the matrix the slower is the timing. Furthermore, we observe from Strong Scaling that after a certain point the performance decreases! The reason probably lies in the use of the `ordered` clause. When we have few threads, they rarely conflicts with each other, but when the number increases so does the probability to have to wait a not negligible time to proceed.

For the same reason, we observe that with more than 8 threads the weak scaling is far from being horizontal, the speedup is way distant from the ideal and the efficiency drops down towards zero.

1.4.2 FLOPS

Floating Point Operations (FLOPS) are defined as the total number of operations involving floating point variables performed by a program. I will for simplicity calculate It only in the dense algorithm case. We have three cycles that iterates over `ROW_N_A`, `COL_N_B` and `COL_N_A` respectively. The initialization of the value does not consume FLOPS, as it is only an assignment not a computation. On the other hand, the update of each value consist of two operations: a product and a sum. Hence, we can compute the FLOPS:

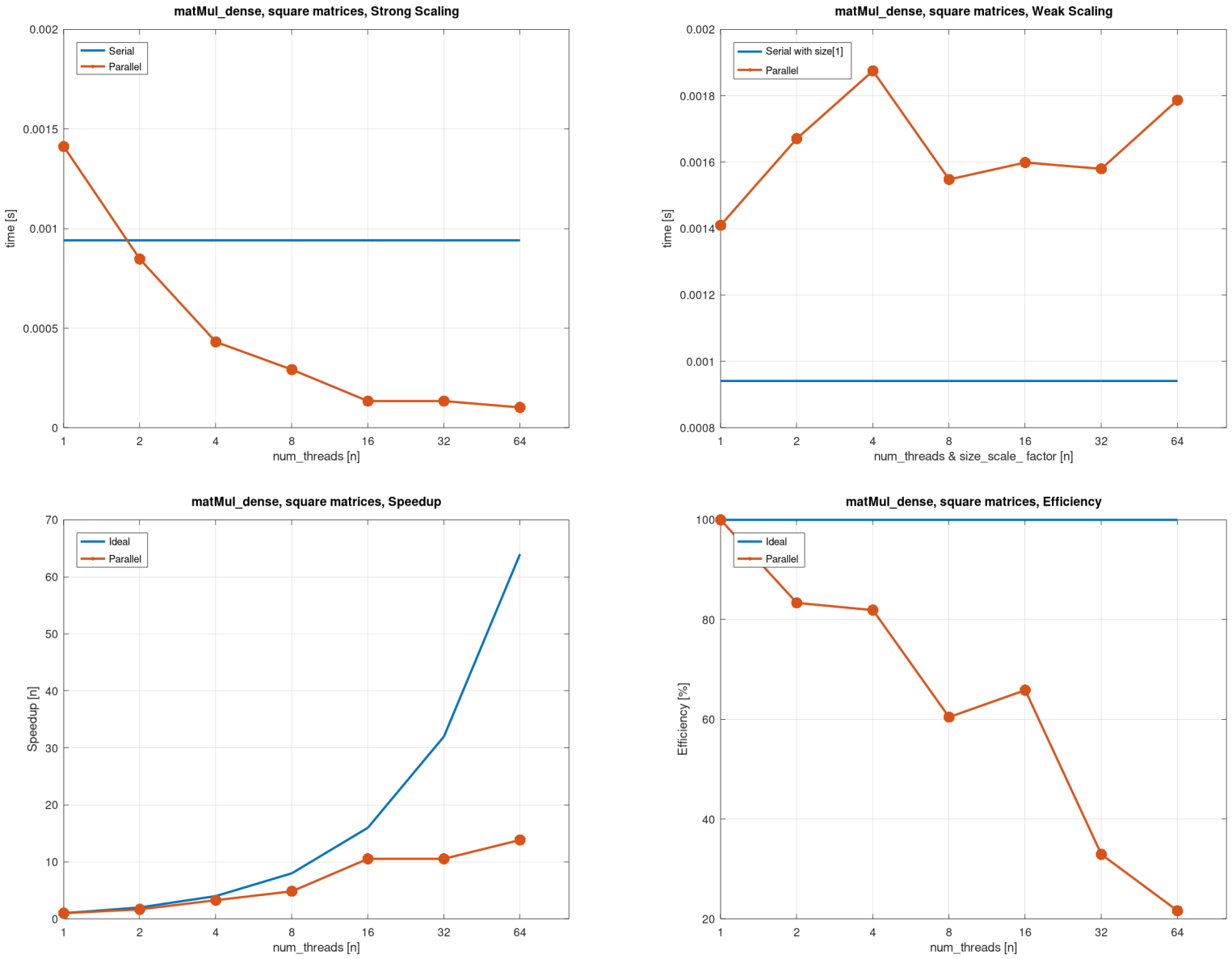
$$FLOPS = ROWN_A \times COLN_B \times COLN_A \times 2$$

That declined in the case I called "square" has the following valuation:

$$FLOPS = 64 \times 64 \times 64 \times 2 = 524288$$

[4]But of course I invite you to have a look to all the plots, to see that the considerations I made here are applicable to them as well.

Figure 1: Strong Scaling, Weak Scaling, Speedup and Efficiency of the dense matrix multiplication algorithm, "square" case.



This value can divided by the amount of time taken by the program to be executed, obtaining the FLOP/s, a measure to estimate how fast is the computer in arithmetic tasks. Looking at the time values of the sequential program and the parallel with 1 and 64 threads, we obtain:

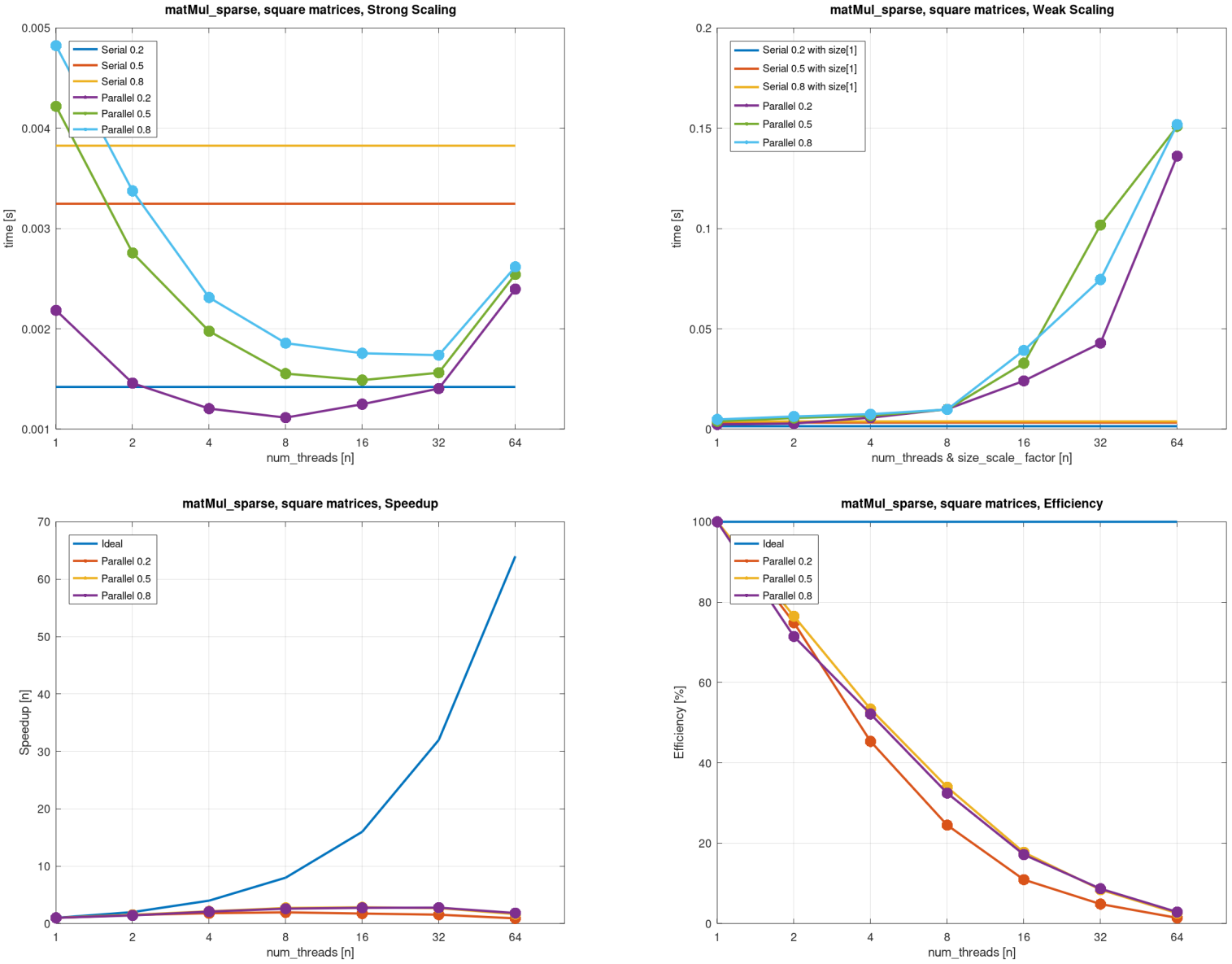
$$serial : FLOP/s = 524288 \div 0.000941 \approx 557\ MFLOP/s$$

$$parallel_1 : FLOP/s = 524288 \div 0.001412 \approx 371\ MFLOP/s$$

$$parallel_{64} : FLOP/s = 524288 \div 0.000102 \approx 5.14\ GFLOP/s$$

Coherently with the plot, we can observe that the speed of the parallel problem with only one thread is inferior to the sequential program, but when the number of threads increases we observe an improvement also in the number of operations per seconds.

Figure 2: Strong Scaling, Weak Scaling, Speedup and Efficiency of the sparse matrix multiplication algorithm, "square" case.



1.4.3 Conclusions

From this experiment I learned that OpenMP clauses can have a strong impact over the parallelization process. In particular, synchronization clauses like `ordered` can allow us to parallelize part of the code that are not obviously parallel, but we need to take into account that after a certain quantity of thread they may drop the performance. So, another important thing this exercise taught to me is that the presence of too many threads can sometimes degrade the execution times, and so it is essential to choose the right number.

2 Parallel matrix transposition

2.1 Introduction

In the second exercise, we are asked to perform a matrix transposition in two different ways: first in the classical element-by-element fashion, and then throw a block access. For explanation simplicity, I put the description of these two approaches in two different sections of this report, and I compared them in the conclusion of the second approach (section 3.4.3).

Again, the code will need to work for both dense and sparse matrices and will finally be parallelized with OpenMP. The information I extracted form the tests are strong scaling, weak scaling, speedup, efficiency and bandwidth.

2.2 Description and Methodology

2.2.1 The sequential program for dense matrices

The sequential dense program is very simple. Just like in the previous exercise, I implemented an object orientated code and I iterated over many parameters and over many trials.

The actual code that solves the problem (and that we are interested to measure) is nothing more than a double `for` that copies each element of `A` into the correspondent cell of `AT`.

2.2.2 The sequential program for sparse matrices

To be able to transpose a sparse matrix in the CSR format, I used [2] as my main source of inspiration. The key point, is to prepare the output matrix so that we know how much space will each row occupy. To do so, we start from `AT.row_index` initialized with zeroes, and then iterate throw `A`, incrementing such an index every time we meet an element of the corresponding column. Like, this `AT.row_index` contains the number of elements the output matrix will have in each row. If we perform an accumulation over this array, we will get the `AT.row_index` as specified by the CSR standard.

After that, we can iterate throw the elements of `A`: we extract its rows, and each element is pushed int the row of `AT` correspondent to its column index. Note that in order to do this we need to know how the rows of `AT` are distributed over `AT.vals`, i.e. we need to know where each row starts, and that's exactly the reason of the preprocess we have done before. This algorithm has the side effect to shift right `AT.row_index`, that's why I added an additional loop to shift it left back to its original configuration.

2.2.3 The parallel program for dense matrices

The parallelization approach is very similar to the one used in matrix multiplication. I put the sensitive code into an `#ifdef` and compiled with the `-fopenmp` flag. The core loop is parallelized with a `#pragma omp parallel for collapse(2)`, as the the nested iterations are independent. I would like to point out again that `AT` is shared because the threads access it always in different cells, and so we have no conflicts. I opted for a static schedule, as the operation of read-write a value is so simple that I don't expect it to require a sensible time difference among the threads, and so I don't expect to see slower or faster threads.

2.2.4 The parallel program for sparse matrices

This has been one of the most complicated code to parallelize. The logic behind the choice that I made is due to the features of the CSR standard. Recall that in such a representation the matrix is stored as many row slices, whose starting point is stored by `A.row_index`. This lead to the fact that rows can be indexed, while columns need to be accessed with a sequential scan.

From this premise it follows the conclusion that the outer loop can't be parallelized. To understand why, consider the following example. Two thread are scanning two different rows of `A`. Each one finds an element in the column `x`. They would both try to push their vaule in the `x` row of `AT`, that would result in a conflict. Moreover, even if we solve this conflict we have the necessity for all the elements in a row to be ordered by their column index, and a parallel execution cannot guarantee this. In simple words, parallelize the outer loop leads to conflicts and to the scramble of the columns of `AT`.

On the other hand, the inner loop *can* be parallelize. This is because when we consider a single row of `A`, its elements can be independently pushed in the rows of `AT` as, I repeat it again, rows can be freely and independently accessed in the CSR.

Unlike the previous sparse-parallel code I opted for a static schedule. The reason is that even thought we're talking about sparse matrices, when we extract a line we know how many elements it has, so it is possible to equally split the work a priori.

2.3 Experiments and Benchmarks

2.3.1 The Chosen Parameters

When I decided how to set the dimension of the matrix in the various cases, I was guided by the idea that the total number of elements should be constant if we want a fair comparison. The ways in which I arranged them are the following: A matrix with many columns and few rows, a matrix that has many rows and few columns, and a square matrix. I called this cases "fat", "thin" and "square" respectively.

For the sparse matrices I also played around with the density, making it vary from 0.2 to 0.5 to 0.8. The reason for this numbers, is to be able to experiment with a very sparse matrix, a matrix that is sparse but not so much, and a matrix that probably is better to handle as a dense one.

To test the programs, I simply added the instructions for compilation and execution in the `compile.sh` and `benchmark.sh` scripts respectively.

2.4 Observations and Conclusions

2.4.1 Plot Analysis

For the matrix transposition, the case I decided to analyzed in figure 3 is the one that deals with square matrices. Just like the multiplication, in the Strong Scaling we have an initial slowness of the parallel code that is then recovered. However the weak scaling is much worse, as it is very evident by the peak at 64 threads. However if we restrict our attention to the first values, it looks quite linear.

The Efficiency plot allowes for a more precise elaboration of this concept: It is quite stable until 16 threads, and then it starts to fall down. So, we can infer that the threads begin to be too crowded when they are more that 16.

The extreme case, is the "fat" case [5], in which the execution time doesn't just increase, but return worse than the serial algorithm. This can be explained with a worse access pattern than the other input configurations. In fact, in C++ matrices are row-major. In the fat case, input rows are much bigger than columns, and so the output matrix will have tons of columns. This means the threads will repeatedly jump around the memory, as the elements they need to push can be very far away from each other.

In figure 4 we can observe the data of sparse matrix transposition. It is absolutely not parallel friendly: the times of strong scaling are always bigger than the serial code, the weak scaling looks inexistent and the efficiency drops exponentially. If I had to speculate about the possible reasons, I would say that in sparse matrices rows can be very short. This means it is very easy to be in a situation in winch threads are overcrowded. In fact, the only case in which the parallel code beats the serial is the "fat", that is exactly the case with the longest lines, while in the opposite "thin" case we have the worst Strong Scaling. Furthermore, the implicit barrier at the end of the `pragma omp for`, is in the inner loop, and so a single slow thread can prevent all the others from going on, limiting the improvements of parallelization.

2.4.2 Bandwidth

Unlike the matrix multiplication case, the matrix transposition problem has a low arithmetic intensity, and so it will be memory bounded. hence, the metric we are interested is the bandwidth. We can calculate it taking into account the number of bytes read and written by the program, as well as its execution time:

$$Bandwidth_{effective} = ((Br + Bw) \div 10^9) \div t_{routine}$$

Looking at the code, we see that the matrix is entirely read once and written once. Considering that a float occupies 4 bytes, we can compute the first term, considering to be in the "square" case:

$$(Br + Bw) \div 10^9 = 2 \times 4 \times 512 \times 512 \div 10^9 \approx 0.002$$

Now, we should only read some times in the report files, and plug them into the formula. As for the first exercise, I will report the values for the sequential program, the parallel with one thread and the parallel with 64 threads:

$$serial : Bandwidth_{effective} = 0.002 \div 0.000758 \approx 2.638\, GB/s$$
$$parallel_1 : Bandwidth_{effective} = 0.002 \div 0.000963 \approx 2.077\, GB/s$$
$$parallel_{64} : Bandwidth_{effective} = 0.002 \div 0.000104 \approx 19.231\, GB/s$$

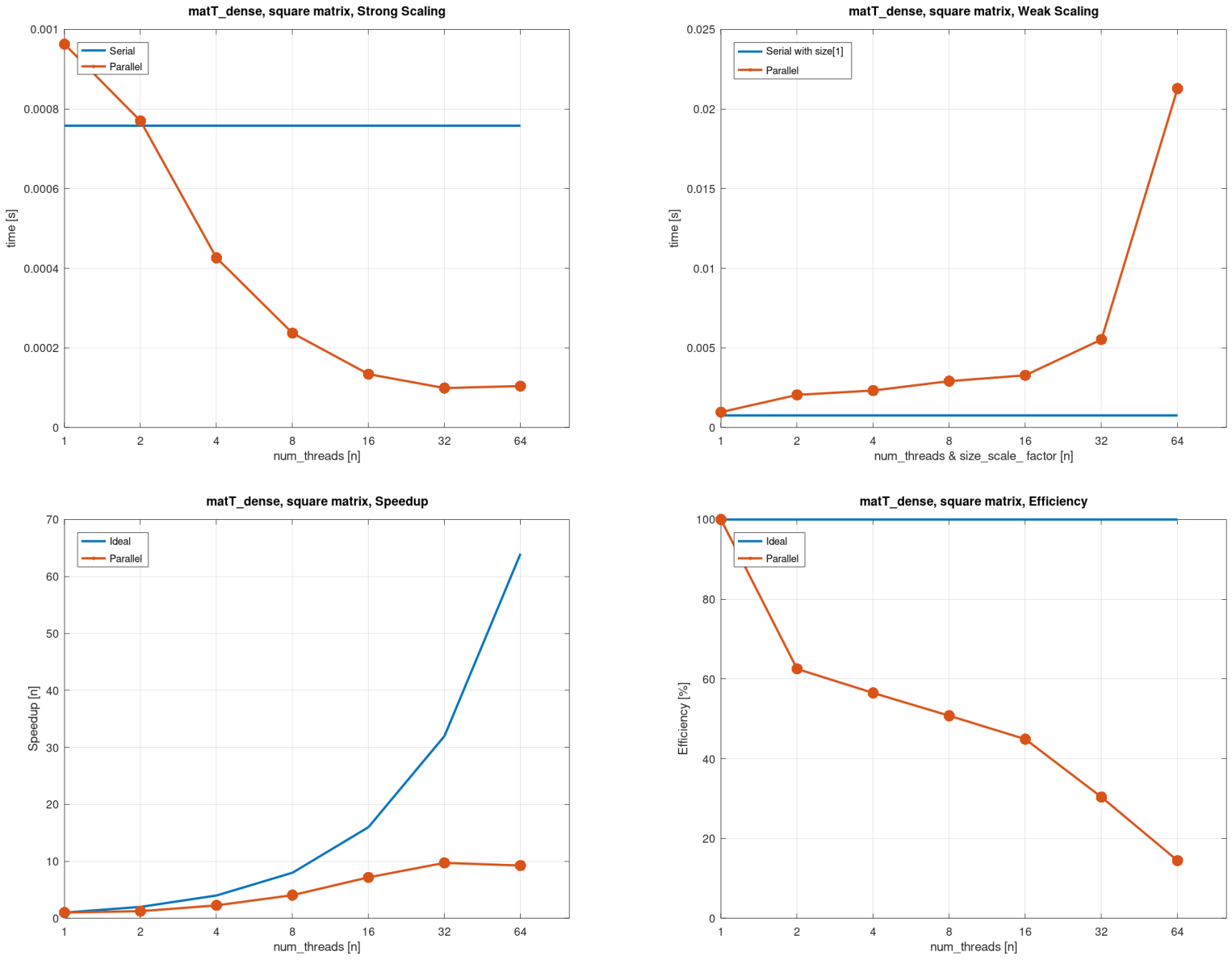
Again, the results are coherent with the plots, as the best one is the parallel code with 64 threads, whose value is way higher that the other two, followed by the serial code and finally the parallel with one thread.

2.4.3 Conclusions

I think that the most important takeaway of this exercise is that parallel code performance really depends over a huge quantity of factors. We can have a program that performs well in a certain input shape, but that turns out to be inefficient with input structured in a different way, exactly

[5] It's not in this report, but you can look at it in te plot folder.

Figure 3: Strong Scaling, Weak Scaling, Speedup and Efficiency of the dense matrix transposition algorithm, "square" case.



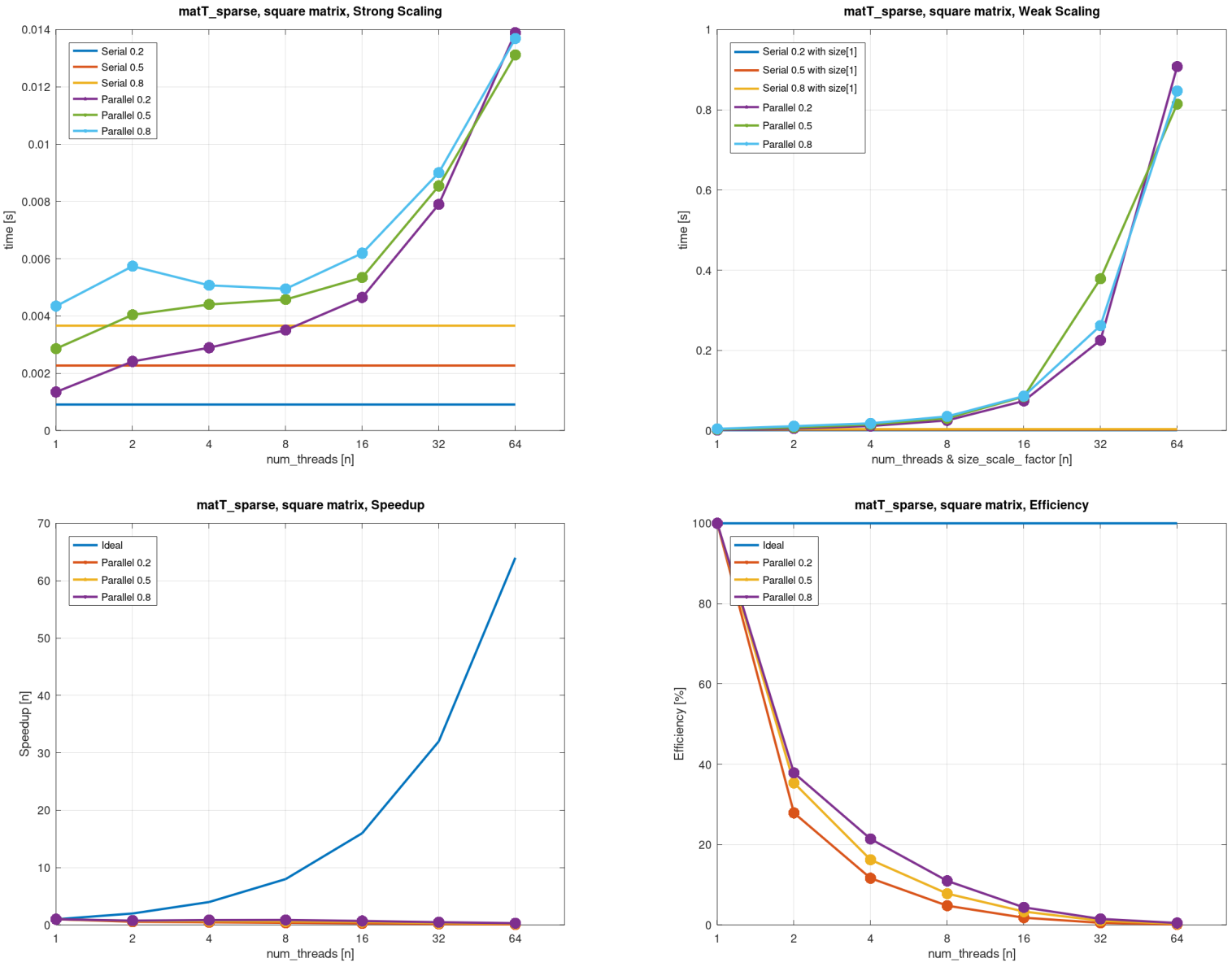
as it happens with `matT_sparse` applied to thin or fat matrices. This also highlight the necessity to test a program in order to see if the parallelization is helpful in the problem we are dealing with or it is turns out to be a burden.

3 Parallel matrix block transposition

3.1 Introduction

The second exercise is not over, as we are also required to transpose the input matrix operating by blocks. I decided to analyze this case in a different section just to have a better structure of the

Figure 4: Strong Scaling, Weak Scaling, Speedup and Efficiency of the sparse matrix transposition algorithm, "square" case.



report, but all the consideration made for the classical transposition are still valid.

3.2 Description and Methodology

3.2.1 The sequential program for dense matrices

Compared with the classical transposition algorithm seen before, operate by blocks makes this code way bigger. First of all, it's necessary to check whether the dimensions of our matrix is compatible or not with the desired block division. If it is, we calculate the number of blocks per row/column. This number is required to be able to iterate throw the blocks.

The main loop consist of two nested `for` loops that identify the target block. Inside it, we find

other three couples of nested cycles: the first one copies the values of the considered block into a support matrix `B`, the second one transpose such a matrix, and the third one put the result into the right block of `AT`.

I also created another version of the algorithm that implies just four nested loops and one assignment. The "block" fashion is in this case achieved throw a proper indexing of the input and output matrices, no middle matrix involved. Such a version is more efficient, however I didn't know if we have as a requirement to explicitly extract the block, transpose it and reinsert it, so I assumed the worst case and implemented the former version, leaving the faster one just as a commented code.

3.2.2 The sequential program for sparse matrices

Apply the block transposition idea to sparse matrices has not been easy. The general workflow is the same of the dense one, as we need to first check the compatibility between blocks and dimensions, then execute two `for` loops to select a block, extract it, transpose it, and insert it in the right position of the output matrix.

To transpose the local block we can simply reuse the sparse algorithm analyzed in section 2.2.2. Also the extraction of the block is not so difficult, in fact we simply have to extract the proper rows of `A` and iterate them. When we encounter an element whose column belongs to the block, we push it into the proper line of `B`.

The complicated part is how to reinsert the block. The problem is that it spans over many rows of the matrix, and so to insert even just the second line we would need to know how many nonzero elements there will be in the full first line of the output matrix, and leave the correspondent gap in `AT.vals`. Unfortunately, we don't have this information. However, if we deal with a matrix represented as an array of lists instead of a CSR, we would simply push each line of the block into a line of the output with no requirements.

For this reason, I decided to work with a support output matrix represented like this, and to convert it into the proper CSR format only after the transposition is concluded (i.e. into a for that is not nested into the previous ones). This conversion can be done via a simple sequential push of the row lists into the `AT.vals` vector.

3.2.3 The parallel program for dense matrices

It's time to parallelize also this code with OpenMP. The most important decision we have to take, is if we want to improve the inner or the outer loop. Parallelize both is useless, as with a 512×512 matrix and only 64 threads probably the load distribution would be just one thread per block, and so we would end up un a case almost equal to the outer one.

If we parallelize the inner cycles, we are in a situation in which we consider one block at a time and transpose it in parallel. In such a case it may be possible for a single slow thread to degrade the performance, as if we are waiting for its job to finish the transposition, other threads cannot move on. Furthermore, if the block is very small, or we are using a big number of threads, there can be more processes than cells, and so we are not exploiting at best the quantity of cores we have. In the extreme case of a 1×1 block, we degenerate to the serial algorithm.

On the other hand, if we parallelize the outer loop each thread will need to have its own copy of B and BT, increasing a lot the memory space we are occupying. As the space we can use in the cluster is quite big for a simple program like this, I opted for this second solution. As in all the other cases, I used a `#pragma omp parallel for collapse(2)`, but this time it can be seen a new clause: `firstprivate`. It works like a normal `private`, but it initialize the variables with the value

they had in the inactive parallel region. We need this initialization as otherwise we would lose two very important fields of the support matrices, namely `rows` and `cols`.

3.2.4 The parallel program for sparse matrices

When we approach the parallelization of the sparse code, we again face the fact that the CSR format comes with a price. If we are working with blocks located on the same block column of the input, two or more threads may try to push them in the same lines of the output, resulting first in a conflict and then in a scramble of the elements.

For that reason, I parallelized instead the transposition at the level of the single block, a task that can be achieved with the exact same code presented in section 2.2.4 as such a block is in every sense a sparse matrix.

3.3 Experiments and Benchmarks

3.3.1 The Chosen Parameters

In order to compare the block transposition with the classic transposition, we need to test the algorithms on matrices of the same size. The parameter I tuned in this experiment was the size of the blocks. Leaving the total size of the matrix unchanged, I started with many small blocks and gradually moved to a few beg ones. The names I decided are "small", "medium" and "large". For the sparse cases I also tried many densities exactly as in the previous exercise.

3.4 Observations and Conclusions

3.4.1 Plot Analysis

It's time to analyze the plots of the final exercise, presented in figure 5. For the "medium" case, we don't see a nice Strong Scalability, but the Weak Scalability doesn't seem bad at all in its initial points. The efficiency drops quickly and the execution times are much worse than the classical transposition.

A hint to justify this bad stuff, can be obtained looking at other input cases `They're not reported here, look them in the plot folder.`. We see that in the "small" case the situation is even worse, but in the "big" case we have good performances, as the parallel code performs better than the serial one and its execution times are quite similar to the ones of classical transposition. Hence, the bigger the block the better it is. This is perfectly explainable with the access pattern: with bigger blocks the memory accessed by each thread will be almost contiguous, while with smaller blocks they jump around the memory more often, as they frequently change block.

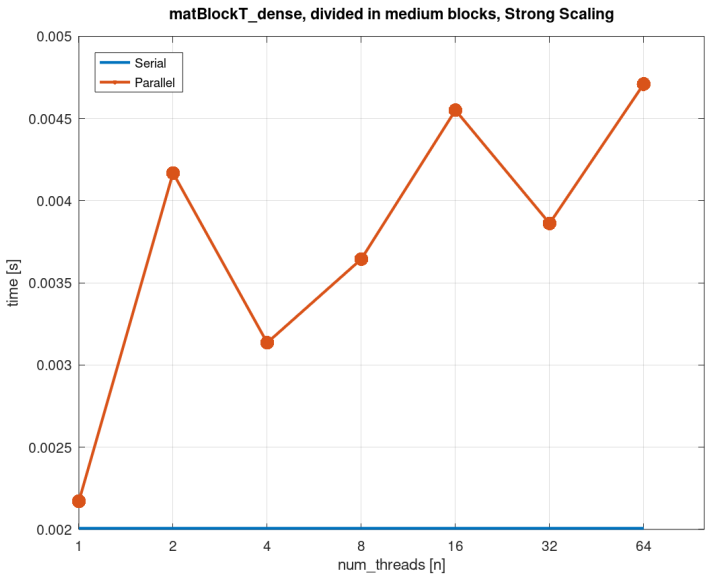
The last plot, is figure 6. It mixes the already bad access pattern of block transposition with the not so good parallelization algorithm already encountered in the matrix transposition. The result is disruptive in terms of performance.

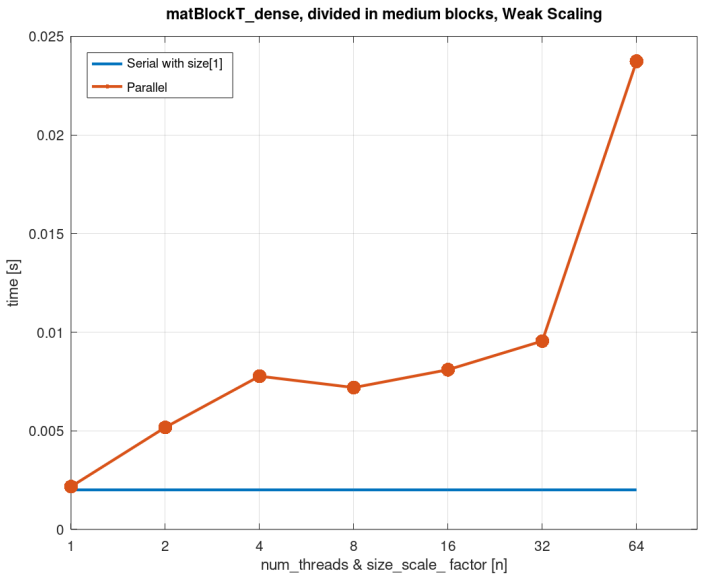
The only positive note, is that its plots confirms some of our previous intuitions: sparser CSR matrices perform better than denser ones, bigger blocks are better, the classical transposition is much better than the block one.

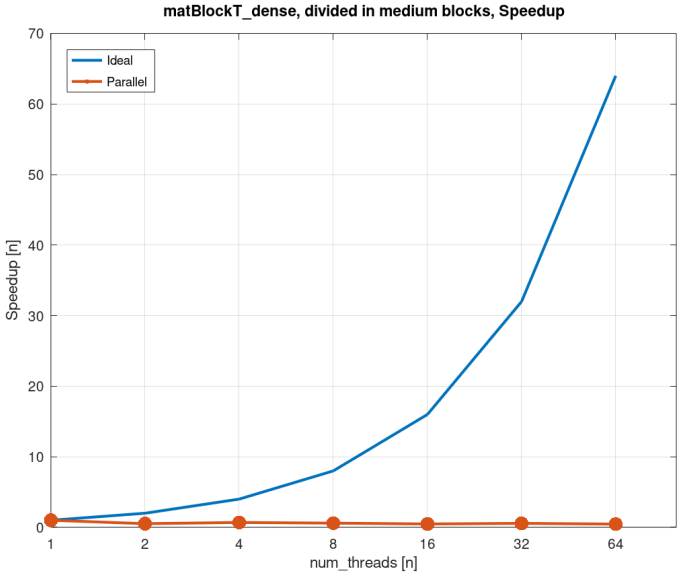
3.4.2 Bandwidth

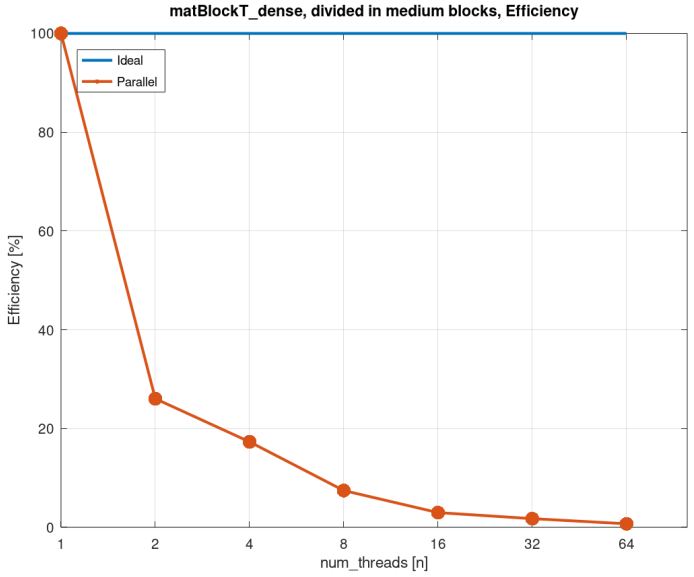
Finally, I will show how to calculate the bandwidth for the matrix block transposition. The formula, is the same as before:

Figure 5: Strong Scaling, Weak Scaling, Speedup and Efficiency of the dense matrix block transposition algorithm, "medium" case.









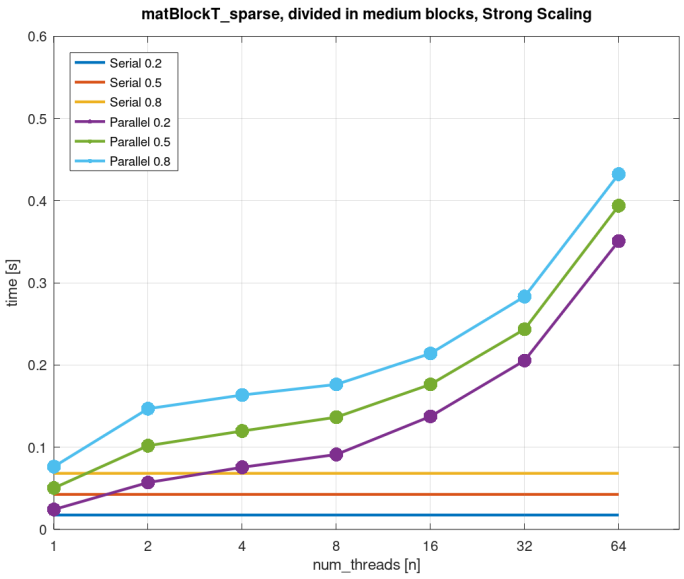
$$Bandwidth_{effective} = ((Br + Bw) \div 10^9) \div t_{routine}$$

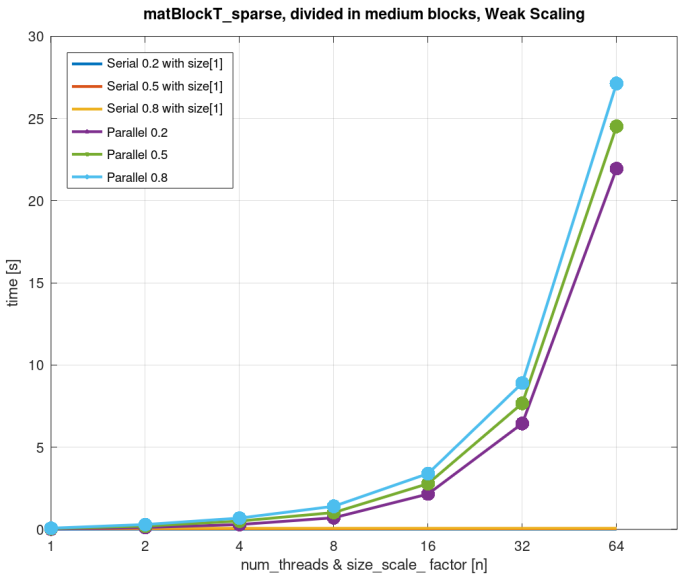
However, we have a crucial difference compared to the classical transposition: each element is first moved from the input matrix to the local one, then it is transposed, and finally copied back. Hence, we have six read/write operations instead of only two. This impacts in the computation of the first term:

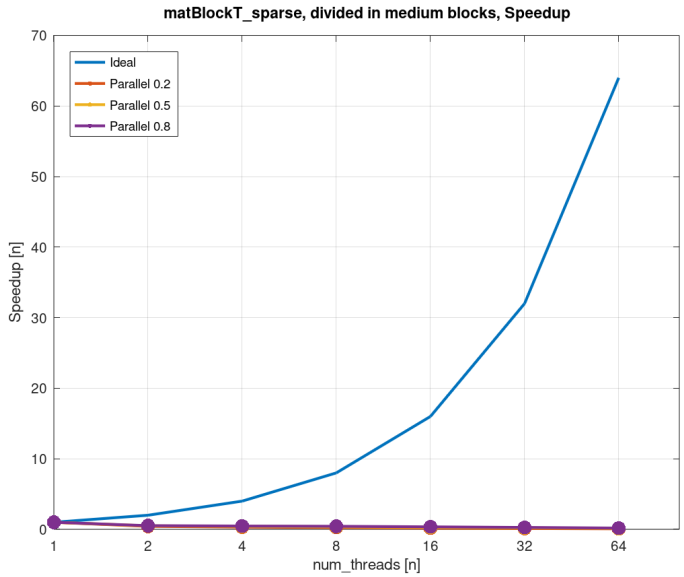
$$(Br + Bw) \div 10^9 = 6 \times 4 \times 512 \times 512 \div 10^9 \approx 0.006$$

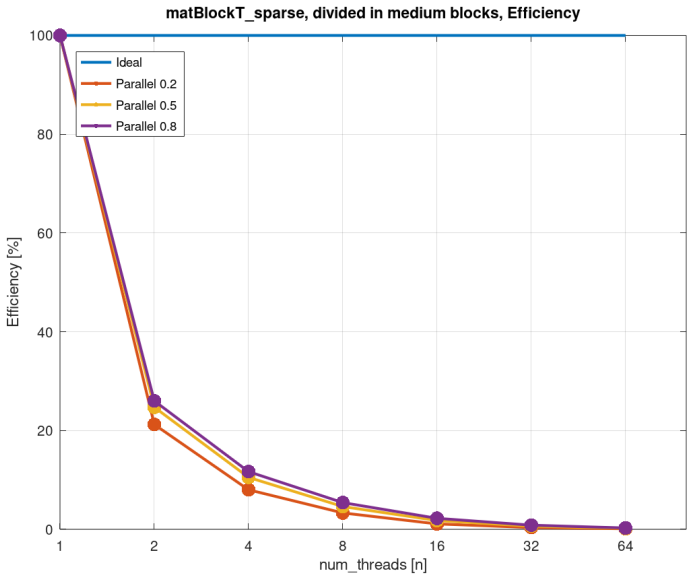
Once this value is computed, we are again in the situation in which we have to read some values and put them in the formula to see what happens. I report the usual three cases I have analyzed

Figure 6: Strong Scaling, Weak Scaling, Speedup and Efficiency of the sparse matrix block transposition algorithm, "medium" case.









also in the previous sections:

$$serial : Bandwidth_{effective} = 0.006 \div 0.002007 \approx 2.989\, GB/s$$

$$parallel_1 : Bandwidth_{effective} = 0.006 \div 0.002172 \approx 2.762\, GB/s$$

$$parallel_{64} : Bandwidth_{effective} = 0.006 \div 0.004710 \approx 1.274\, GB/s$$

We can see that the values of the first two cases are almost similar to each other and to the values obtained for the classical transposition. The bandwidth for 64 threads is instead completely different than its counterpart (in worse), for the same reasons I have already highlighted when talking about the plots.

3.4.3 Conclusions

This experiment taught me the importance of access pattern. Two problems that may look very similar as transposition and block transposition can turn out to be completely different in term of performance. So, it is very important to exploit te cache as much as possible.

4 Final remarks

This homework dealt with parallel matrix operations, highlighting aspects like the choice of omp clauses, the performance metrics and the cache usage. I think that the most important message is to test your program on your data on your computer, as the variables in play are so many that there's no general rule to parallelize a code, but we have to find the solution that best fits the situation in which we are.

However, to be completely honest most of the time I spent over the homework was due to the research about sparse matrices an how to use them. So, these exercises taught me both stuff about parallelization as well as how to deal with sparse matrices.

References

[1] I learned about the CSR standard for sparse matrix on Wikipedia:

`https://en.wikipedia.org/wiki/Sparse_matrix`

[2] I had the intuition on how to perform operations in such a format looking at this site:

`https://www.geeksforgeeks.org/operations-sparse-matrices/`

[3] For the OMP part, I found everything that I needed in the slides on the course.