Remark: the results in this paper are obtained by remote connection to the lab and by using gcc compiler, version 8.2.0.

1 Description and numerical results

We made a parallelization of the lapFusion code using OpenMP, and then performed some experiments to analize the performance of the program and link them to problem-dimensional issues.
In particular we run different parallelizations, respectively with 1,2,4 or 8 threads, each one with different sizes of the matrix (64,128,256,...,16384) and different number of iterations (100, 500, 1000).
For each experiment we collected the CPU execution time using the `perf stat` command and we otained the following results:
In the table the TL value means that the CPU execution time limit was reached with that problem size, and so

100 ITERATIONS									
matrix size ->	64	128	256	512	1024	2048	4096	8192	16384
1 thread	0,017646836	0,049587437	0,165680451	0,578700125	2,250387945	8,913072711	35,556533455	141,802850062	566,690510690
2 threads	0,008552856	0,026161036	0,183704000	0,302037559	1,168144361	4,597471636	18,328129195	73,085288454	292,085743711
4 threads	0,156563596	0,012912229	0,039419116	0,127735653	0,465882781	1,813730031	7,047961770	29,011472490	112,168649772
8 threads	0,008401835	0,017022661	0,058117690	0,160466376	0,840656670	2,409072105	9,497319915	37,905232625	151,427152213

500 ITERATIONS									
matrix size ->	64	128	256	512	1024	2048	4096	8192	16384
1 thread	0,056921053	0,171612273	0,728583338	3,115160593	12,150875150	48,048956477	179,678345496	712,54493587	TL
2 threads	0,032291711	0,065474846	0,294101889	1,217023486	4,519492797	17,674417731	72,554692605	279,10188098	TL
4 threads	0,014459055	0,042497657	0,155533024	0,632850571	2,364323420	9,115204287	35,638634529	140,94794522	TL
8 threads	0,028504550	0,069864905	0,222056875	0,836530172	3,139244307	12,199151581	47,966720139	189,91035504	TL

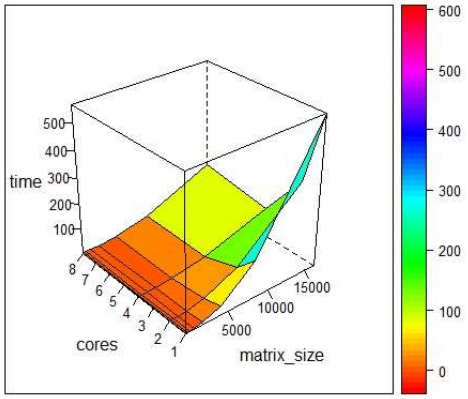
1000 ITERATIONS									
matrix size ->	64	128	256	512	1024	2048	4096	8192	16384
1 thread	0,103055278	0,358439831	1,438762813	6,087067452	23,982172869	92,331784554	362,165904017	TL	TL
2 threads	0,047101965	0,129135605	0,559225661	2,390431182	9,3853396210	35,55040631	144,6324365	TL	TL
4 threads	0,026452767	0,083154924	0,294682876	1,282515598	4,8828218550	18,65187529	72,33401688	TL	TL
8 threads	0,045044124	0,129306052	0,406346365	1,666148324	6,4633953000	24,744486270	96,494986453	TL	TL

Figure 1: Numerical results

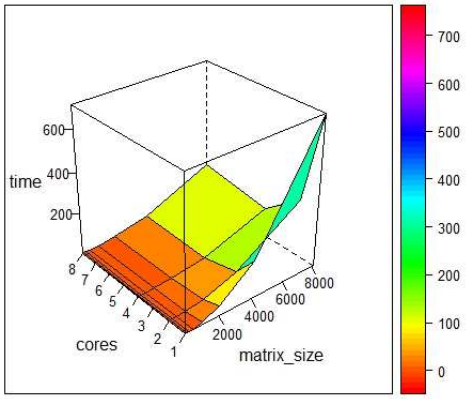
we could not get the proper data.
We made a graph representation of the results obtained, grouping by the number of iterations, obtaining the graphs in the figures.

2 Analysis and comments

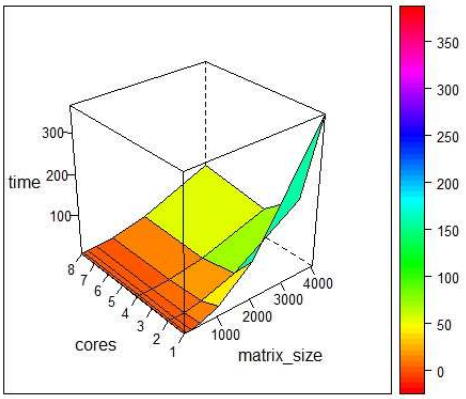
As made clear by the graphs, which display the same surface structure, but with different numerical values, to analyze the problem with 100, 500, or 1000 iterations, is the same if we consider the variability of the other two dimensions (threads and matrix size).
We can observe that, except for the very small matrix sizes, if we fix the number of threads, the execution time grows linearly with the matrix size. Indeed when we increase the matrix size by 4, we obtain a 4 time longer execution time.
Instead, for the very small matrix sizes, we can observe that if we increase by 4 times the matrix size, we always obtain an increasing of the total time which is strictly less than 4 times. This can be explained by the fact that the not-parallelized operations of the program occupy a greater proportion of operations, and so in these cases the matrix size is not a bottle-neck for the execution time (observe that also the creation of the



(a) 100 iterations



(b) 500 iterations



(c) 1000 iterations

threads can be wieved as not-parallelized operations).

If we analyze the results by fixing the matrix-size and varying the number of threads, we can observe that, with the only exception of the case 100 iterations$\times$64 size matrix, as we double the number of threads the execution time becomes an half (or slightly less) as the number of threads is less than or equal to 4, which is reasonable to expect, since by doubling the computational reources the calculations should be done 2 times faster.
But if we double the number of threads from 4 to 8 we observe (very clearly in the graphs) that the execution time increases, instead of diminishing. This, if the matrix dimension is small, could be explained by the fact that the time the CPU needs to create the threads is not compensated by the amount of work the threads can do together, and so it is not optimal to create such a number of threads, for the small sizes. But for the bigger sizes of the matrix this is a very strange result, indeed the cost to create the threads should be compensated by the bigger amount of calculation they can perform together. Indeed, analyzing deeply the data we obtained we have observed that when try to run 8 threads, the CPU only created at most four threads. So the data for the 8 threads cases are not valid, and does not make sense to do analysis based on them.
In any case we tried to explain to ourselves the strange time-increasing which is sistematically observed when trying to run 8 threads as follows: the system tries to create more than 4 threads, but since only 4 are allowed, in doing this, it looses time, and so the program runs slower than the 4-thread cases.

To get some valuable data in the 8 threads case we try to run the program in our PC, and indeed we found out that the more threads we execute, the less execution time we obtain, as we expected at least for the bigger matrix sizes.

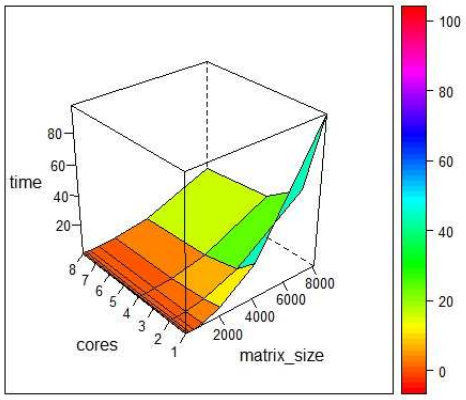
In the following table we show the execution time of the experiments we run on our PC, and in the figures the new graphs obtained.

100 ITERATIONS								
matrix size ->	64	128	256	512	1024	2048	4096	8192
1 thread	0,013	0,031	0,102	0,370	1,532	5,754	23,167	97,225
2 threads	0,009	0,019	0,057	0,195	0,756	2,979	11,840	44,806
4 threads	0,010	0,013	0,038	0,120	0,428	1,631	6,719	28,479
8 threads	0,009	0,013	0,031	0,118	0,405	1,434	5,626	23,741

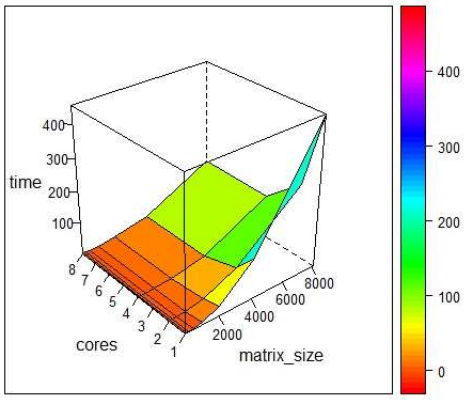
500 ITERATIONS								
matrix size ->	64	128	256	512	1024	2048	4096	8192
1 thread	0,035	0,121	0,534	2,294	8,218	31,733	124,323	454,487
2 threads	0,021	0,062	0,252	1,092	3,976	15,505	59,320	225,264
4 threads	0,015	0,055	0,146	0,567	2,108	7,728	30,090	129,655
8 threads	0,016	0,037	0,135	0,492	1,868	7,741	30,154	136,894

1000 ITERATIONS								
matrix size ->	64	128	256	512	1024	2048	4096	8192
1 thread	0,062	0,241	0,975	4,447	17,274	65,173	233,061	911,156
2 threads	0,035	0,119	0,472	2,163	8,284	31,452	123,197	456,988
4 threads	0,022	0,072	0,261	1,124	4,279	17,093	66,618	286,083
8 threads	0,032	0,066	0,265	0,987	3,906	16,169	59,932	268,345

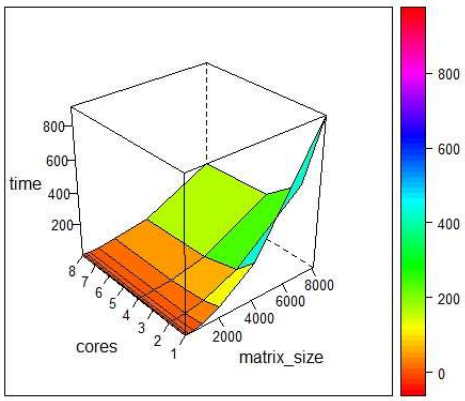
Figure 3: PC configuration: The MacBook Pro, Core i7 2.6GHz processor(3820QM), RAM 16GB 1600MHz



(a) 100 iterations



(b) 500 iterations



(c) 1000 iterations