Report Lab Work 2 - Group 3E1

Names:

- Jose Collado San Pedro
- Jorge López Ribes

Description of each exercise.

- Exercise 1: In this exercise we are asked to compute the execution time of the function encaja, to obtain it we have used two omp_get_wtime() calls, one at the beginning of the function and one at the end.

- Exercise 2: Here we have to parallelize the loops of the function encaja separately:

    - loop i : It can't be parallelized since each line depends on the one before

    - loop j : It can be parallelized using a pragma omp clause, making variables x and distancia private. We also have to add a pragma omp critical in order to avoid the dependencies between the threads when accessing to the variable distancia_minima

    - loop x : It can be parallelized using a pragma omp clause, we have to do a reduction with the variable distance, in order to compute the total value of the sum.

- Exercise 3: We have to modify the sequential version, making loop x to end as soon as the partial sum of the current distance takes a value greater than the minimum distance computed so far. To do it, we just need to add an if, that calls to a break clause whenever the condition is accomplished.

- Exercise 4: Here we have to parallelize the loops of the new function encaja separately:
    - loop i : It can't be parallelized since each line depends on the one before

    - loop j : The modification in loop x doesn't affect to the parallelization of loop j, so we can use the same clauses as before.

    - loop x : To parallelized it correctly, we have modify the for clause, the first iteration will be the id of the thread, and the increment will be the number of threads that are executing the program.

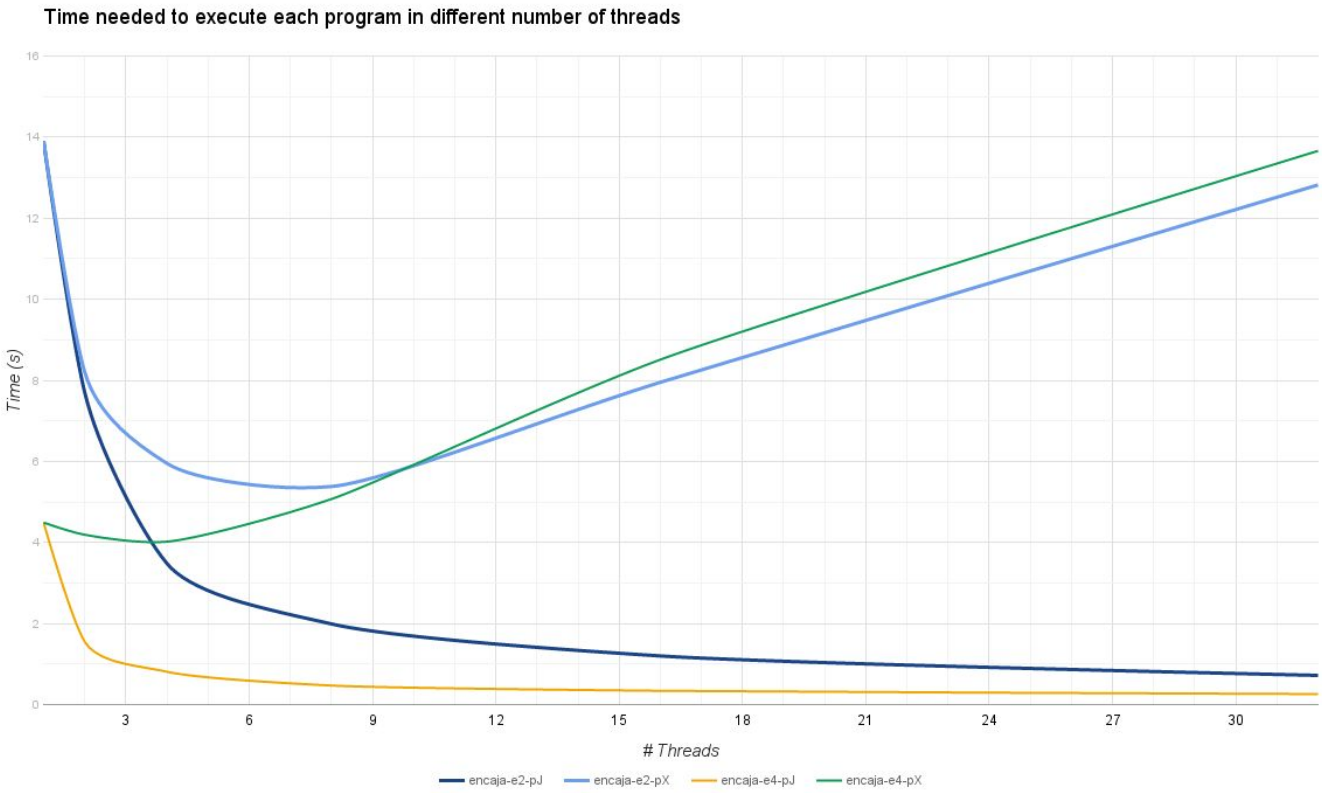
Testing each version of encaja

In this part we executed and tested each parallelized version of encaja in the cluster, changing for each one the number of threads up to 32, which is the maximum of threads that the cluster can support. In order to do that in a easy way and handle the results properly, we did a .sh file called run.sh that executes the 4 programs with different number of threads and prints the execution time.

- *Execution time in seconds for each program in different number of threads*

Program	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
encaja-e2-pJ	13,90	7,70	3,48	1,99	1,2	0,72
encaja-e2-pX		8,20	5,95	5,38	7,95	12,82
encaja-e4-pJ	4,49	1,55	0,81	0,47	0,34	0,26
encaja-e4-pX		4,19	4,02	5,07	8,51	13,66

The time of execution of encaja-e1 is the same as encaja-e2-pJ and encaja-e2-pX with one thread and time of execution of encaja-e3 is the same as encaja-e4-pJ and encaja-e4-pJX with one thread because executing a program with one thread is the same as executing it sequentially.



Speedup

To compute the speedup we have divide the execution time of encaja-e1 and encaja-e3, between the time we have obtained by parallelizing both loops (formula below). All the results are written in the following tables:

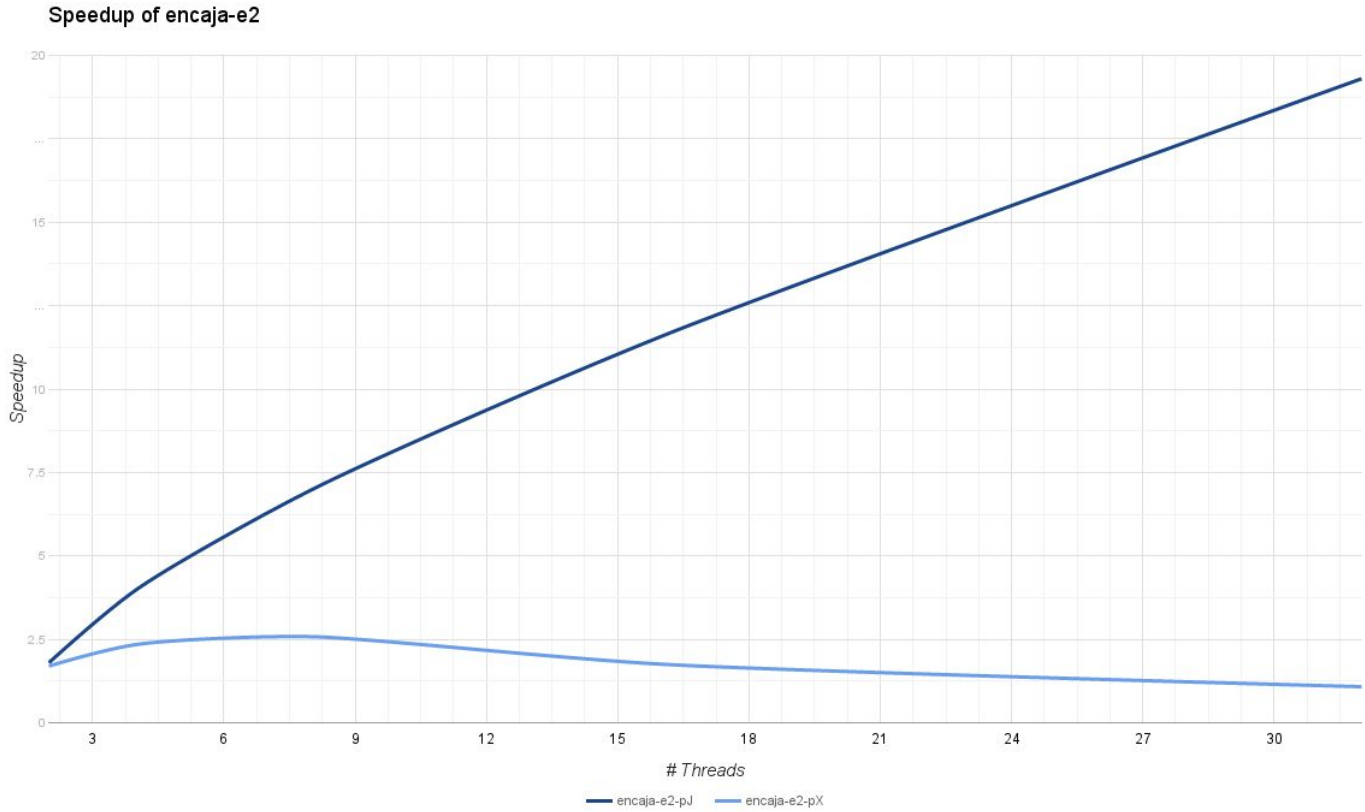
$$S(n,p) = \frac{t(n)}{t(n,p)}$$

The reference time $t(n) couldbe$:

- The best sequential algorithm at our knowledge
- The parallel algorithm using 1 processor

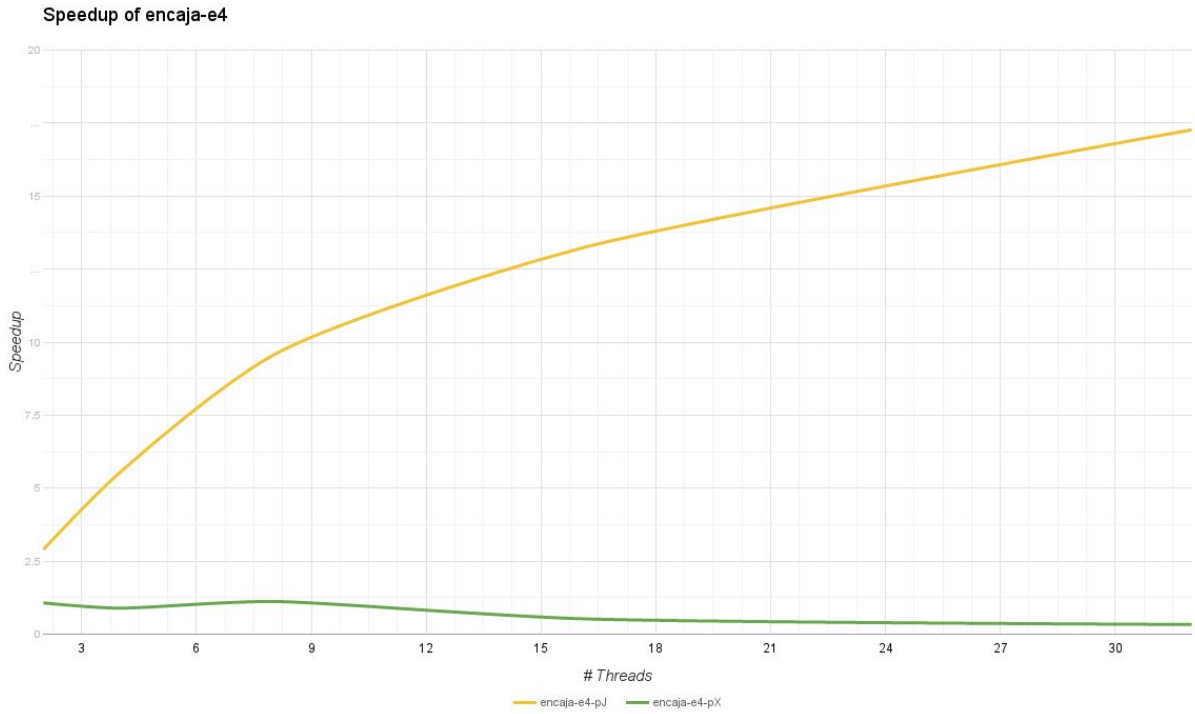
- *Speedup of encaja-e2 (comparing with encaja-e1)*

Program	2 threads	4 threads	8 threads	16 threads	32 threads
encaja-e2-pJ	1,80	3,99	6,98	11,58	19,30
encaja-e2-pX	1,70	2,345	2,58	1,755	1,08



- *Speedup of encaja-e4 (comparing with encaja-e3)*

Program	2 threads	4 threads	8 threads	16 threads	32 threads
encaja-e4-pJ	2,90	5,54	9,55	13,20	17,27
encaja-e4-pX	1,07	0,89	1,12	0,53	0,33



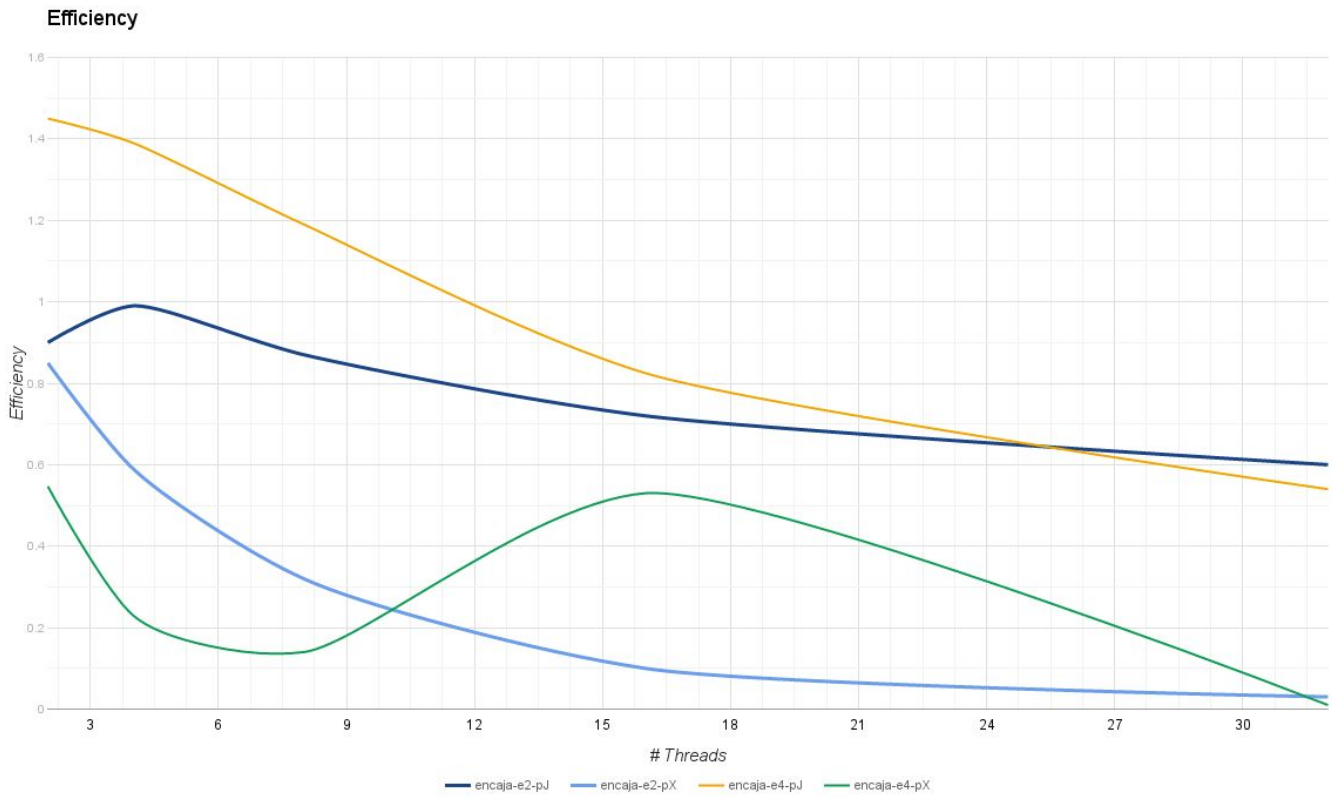
Efficiency

To compute the efficiency of the code, we have divided all the speedups computed in the point before by the number of threads of execution. All the results are written in the following table:

The Efficiency measures the degree of usage of the parallel units by an algorithm

$$E(n,p) = \frac{S(n,p)}{p}$$

Program	2 threads	4 threads	8 threads	16 threads	32 threads
encaja-e2-pJ	0,90	0,99	0,87	0,72	0,60
encaja-e2-pX	0,85	0,59	0,32	0,1	0,03
encaja-e4-pJ	1,45	1,39	1,19	0,825	0,54
encaja-e4-pX	0,547	0,23	0,14	0,53	0,01



Analysis of the results

Finally, we are going to interpret all the results that we have obtained during the practice and the execution of the different parallelized versions of the code.

First of all, *encaja-e2-pJ* improves completely the sequential version *encaja-e1*, but when we try to parallelize the loop X (the loop inside) we observe that there's overhead starting from 4 threads and the performance starts to decrease.
Later, *encaja-e3*, with the modification of the code, highly increase the performance of the sequential version and has the same behaviour as *encaja-e2* when we parallelize the code, there's a improve in the execution time with the outer loop but also has overhead with the inner loop when we reach 4 threads.

Moving to the speedup, we can see it works quite similar to the execution time. When we parallelized the loop j of encaja-e1, we obtain improvements in all the results whereas by parallelizing loop x it just improves until 4 threads, then due to overhead starts to lose performance. Exactly the same happens with encaja-e3, while in loop j parallelized all are improvements, overhead appears in the parallelization of loop x, causing a loss of performance starting from 4 threads.

Finally, we will talk about efficiency, as we can see, encaja-e4-pJ is most efficient implementation. Even so, it also starts to lose some efficiency when we increase the number of threads, owing to the overhead.