Lab 1: OpenMP

Biel Solà Gràcia u214971
Mauro Paniagua Suetta u232628
Martí Pascual Barluenga u232620

Report Questions 1

1. Expose your parallelization strategy to divide the work in the Cholesky algorithm and in the matrix multiplication. Justify the selection of the scheduler and chunk size, and compare different schedulers with different chunk sizes and show the results.

Parallelization Strategy

For both the Cholesky factorization and the matrix multiplication, we divided the work along the matrix's rows. In the **factorization phase**, each iteration i builds one row of the upper-triangular matrix U, keeping the outer loop sequential (to preserve data dependencies) and parallelizing the inner loop over columns $j>i$. We ensure that all threads can simultaneously compute the non-diagonal elements of a given row once its diagonal element is known. In the **multiplication phase**, computing each row of the product B only depends on the corresponding row of L and the entire matrix U, so we parallelize the outer loop over i, assigning different sets of rows to each thread. This row decomposition guarantees data separation; no two threads ever write to the same element, and each thread's working set is contiguous in memory.

Scheduler & Chunk-Size Selection

Because the Cholesky factorization's inner loops perform roughly the same amount of work for each column index j, we used the **static scheduler by default**. Static scheduling gives the lowest runtime overhead, it computes once which iterations each thread will handle and then goes through them. However, in the matrix-multiplication step there could be load imbalances, some rows of L (near the bottom of the matrix) involve fewer non-zero multiplications. Here, switching to **dynamic scheduling** with default chunk size of 1, making faster threads take extra work when they finished early, while keeping the overhead of task assignment low.

Comparison of Schedulers

Phase	static (def.)	static (64)	dynamic (1)	dynamic (32)
Cholesky factor	2.812663s	35.323250s	3.986092s	3.404408s
B = L·U multiply	10.314750s	6.329132s	5.624496s	6.175314s

```
OpenMP Cholesky
Initialization: 0.471088
Cholesky: 3.404408
L=U': 0.012987
B=LU: 5.624496
Matrices are equal
A==B?: 0
```

```
OpenMP Cholesky
Initialization: 0.372653
Cholesky: 35.323250
L=U': 0.063362
B=LU: 6.329132
Matrices are equal
A==B?: 0
```

```
OpenMP Cholesky
Initialization: 0.465454
Cholesky:
L=U': 0.016568
B=LU: 10.314750
Matrices are equal
A==B?: 0
```

```
OpenMP Cholesky
Initialization: 0.468179
Cholesky: 2.812663
L=U': 0.014648
B=LU: 6.175314
Matrices are equal
A==B?: 0
```

We deleted the rest of .out .err files and just leave the finally used ones.

From these results:

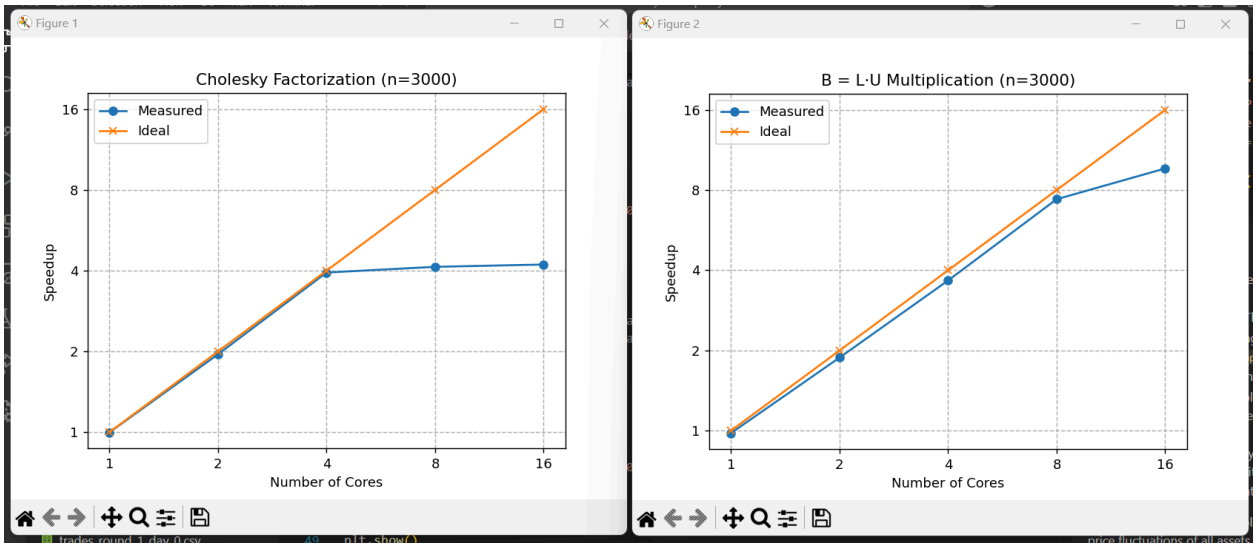
Cholesky: The default static scheduler (with automatic equal-sized chunks total_iterations / threads) was both simplest and fastest, and dynamic variations introduced extra overhead without improving balance.
Multiplication: dynamic(1) with default chunk of 1 has done the best throughput by smoothing out the small per-row variation in work, has more overhead but better performance.

2. Make two plots: one for the speedup of the Cholesky factorization and another for the matrix multiplication for n " 3000. Use 1, 2, 4, 8, and 16 cores for a strong scaling test. Plot the ideal speedup in the figures and use a logarithmic scale to print the results. Discuss the results.

Speedup measures the ratio between the sequential time and parallel time. We used **Matplotlib**, we had trouble finding how to make the axis in log-scale in excel. To change the number of cores, we modified the file job.sh, specially `#SBATCH --cpus-per-task= X` where X are the number of cores (cpus = cores with SLURM).
Here are the values we collected from changing the cores per task:

Cores	1	2	4	8	16
Parallel Cholesky	**12,191216**	**6,238744**	**3,092166**	**2,944783**	**2,886943**
Parallel Multiplication	**24,492041**	**12,75446**	**6,54957**	**3,244548**	**2,4872**
Sequential cholesky	**12,174418**	12,867348	12,88921	11,256139	11,139684
Sequential Multiplication	**23,988808**	25,369177	25,269575	22,242528	21,009141

We deleted the rest of .out .err files and just leave the finally used ones.



Ideally, the speedup (sequential time / parallel time) grows linearly, since when we double the number of cores, then the speedup should double, **but in practice it does not happen**. We observe that at ceratin number of nodes the curve starts having less slope, and this is due too several reasons, one that we detected is that there is some serial work that is not or cannot be parallelized, such the computation of the Uii diagonal in Cholesky factorization, causing that as you add more threads, that fixed serial cost becomes a larger fraction of total time (Amdahl's Law).

Report Questions 2

1. Explain how have you solved each of the parallelizations.

To parallelize the histogram construction, we implemented four versions:
Sequential: A baseline implementation where a single thread assigns values to histogram buckets in a for-loop.
Critical: I parallelized the loop using `#pragma omp parallel for`, and protected the shared hist array updates with a `#pragma omp critical` section. This ensures mutual exclusion, allowing only one thread to increment a bucket at a time.
Locks: I initialized an array of `omp_lock_t`, one lock per bucket. Threads use `omp_set_lock` and `omp_unset_lock` to control access to specific buckets, its simpler but has a lot of overhead, too much synchronization.
Reduction: Each thread maintained a private copy of the histogram in a local array (local_hist). After all increments were done, each thread merged its results into the shared hist array using a `#pragma omp critical` section. This cancels completely race condition as the other parallelization and minimizes contention during updates, which is when multiple threads are trying to access the same memory location at the same time.

2. Explain the time differences between different parallel methods if there are any.

Execution time varied significantly depending on the synchronization method:

```
4 threads
Sequential
 histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.000990 seconds
par with critical
 histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.068189 seconds
par with locks
 histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.055095 seconds
par with reduction
 histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.000348 seconds
```

Critical was the slowest among the parallel versions (0.068189s) because all updates, regardless of which bucket, were serialized, since all threads are waiting to access the critical section. This creates a bottleneck, especially when many threads try to update the histogram concurrently.
Locks performed better than critical (0.055095s) by allowing concurrent updates to different buckets. However, the overhead of setting and releasing locks still have affected performance, especially if threads struggle for accessing the same bucket.
Reduction was the fastest among all parallel versions (0.000348s). It avoids contention during the main computation by letting each thread update its local copy. The only synchronized part is the final merge, which is relatively fast. This approach scales better with the number of threads.

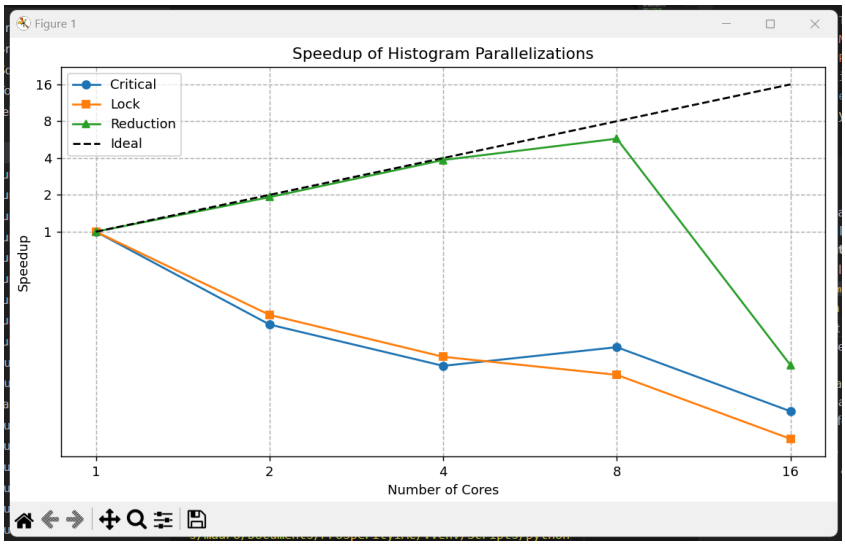
3. Make a speedup plot for the different parallelization methods for 1, 2, 4, 8, and 16 cores. Discuss the results.

Speedup measures the ratio between sequential time and parallel time. We used **Matplotlib** again. To change the number of cores, we modified the file job.sh `#SBATCH --cpus-per-task= X` where X are the number of cores (cpus = cores with SLURM).

Here are the values we collected from changing the cores per task:

Cores	1	2	4	8	16
Critical	0.005459s	0.031333s	0.068189s	0.047973s	0.160282s
Lock	0.005230s	0.025039s	0.055095s	0.077388s	0.258106s
Reduction	0.001342s	0.000700s	0.000348s	0.000233s	0.016536s

We deleted the rest of .out .err files and just leave the finally used ones to not be so messy.



Firstly, we want to state that the values from the table are from several executions, as exercise 1. We wanted solid results and avoid outliers (results with very low probability that sometimes can appear for some reason), the final values reflect our executions correctly after different tests.

The ideal line (black dashed) represents perfect linear scaling, representing that ideally while increasing number of cores, then we increase the speedup.

Reduction (green line) delivered the best overall speedup. It scales efficiently up to 8 threads, as each thread works independently on its local copy of the histogram. Only at the end is a small critical section used to merge results. This minimizes both contention and synchronization overhead. However, the speedup drops at 16 threads, likely due to increased overhead in the merging step and limited benefit from additional threads due to memory bandwidth constraints.

Critical and **Lock** (blue and orange lines) implementations both show poor scalability. Their speedup decreases with more threads, indicating that the synchronization overhead outweighs any benefits from parallelism. In both cases, contention for shared buckets forces threads to wait, reducing overall efficiency. Critical is particularly limiting since it serializes all bucket updates, even when different buckets are being accessed.

Report Questions 3

1. Explain the different implementations of the argmax function (sequential and recursive), and how you parallelized each of them.

In our program, the argmax function is implemented in four different ways: two sequential and two parallel. Each one follows a different approach to find the maximum value in a vector and the index where it appears.

The first implementation, argmax_seq, is the most basic one. It just loops through the vector from beginning to end, comparing each value with the current maximum. If it finds a bigger value, it updates the maximum and its position. This version runs on a single thread and doesn't use any kind of parallelism.

The second version, argmax_par, parallelizes the loop using OpenMP. Instead of all threads working on the same maximum (which would cause problems), each thread keeps track of its maximum and index. At the end of the loop, they all enter a critical section to update the global result. This works, but the critical section can limit performance because only one thread can access it at a time.

The third one, argmax_recursive, uses a recursive approach to divide the problem in half until it reaches a base case. Each half is solved recursively, and then the two results are compared. This version is still sequential, but it sets up the structure for the last version, which is parallel.

The fourth version, argmax_recursive_tasks, is the parallel recursive one. It uses OpenMP tasks to solve each half of the vector in parallel. Every time the function splits the vector, it creates two tasks. Once both sides are done, it compares their results. The performance of this version depends a lot on the value of K, which tells the program when to stop dividing. If K is too small, it creates too many tasks and slows down. If it's too big, it behaves more like the sequential version.

2. Run the code with 2, 4 and 8 threads for a vector of size N " 4096 ˚ 4096 and plot the strong speedup for both parallel implementations. For the recursive and tasks implementations plot the a strong speedup curve for the following values of K, K " 16, 512, 2048, 4096 and K " 8192. Include all curves in the same plot for better comparison. Comment on the obtained results, and explain the behaviors of the different parallel implementations. Hint: depending on the node load, the results might vary. Launch the job several times and keep the cases with the largest speedups. If well implemented, the tasks version should be at least two times faster using K " 8192 and 8 threads.

To complete this exercise, we first executed the code to change the job.sh files for all the requested data, that is, K = 16, 512, 2048, 4096, 8192 for 8 threads and K = 2048 with some threads 2, 4, and 8.

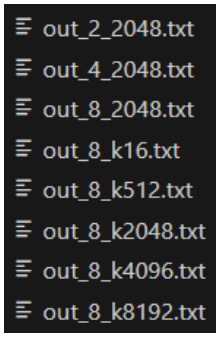
Here you can observe all the changes made to the job.sh file for the example K = 16 and 8 threads, the changes are commented,

lab1_T1G12 > 3_argmax_ex2 > $ job.sh

```
#!/bin/bash

#SBATCH --job-name=ex3
#SBATCH -p std
#SBATCH --output=out_8_k16.txt # the name of the output file so we can know what cout is
#SBATCH --error=out_8_k16.txt
#SBATCH --cpus-per-task=8  # change the number of threads to 8
#SBATCH --ntasks=1
#SBATCH --nodes=1
#SBATCH --time=00:05:00

make >> make.out || exit 1

./argmax 8 16  # Run argmax.c with 8 threads and K = 16
```

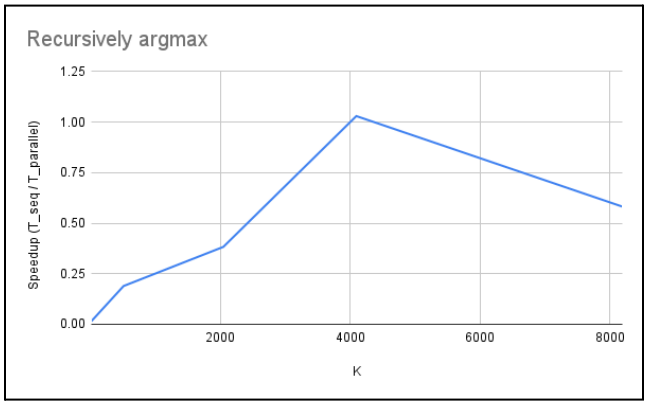
Then, we submitted all the requests to the Pirineus cluster and obtained the results after waiting a bit. *We decided to print both the error and output messages into the same .txt file, since it was easier and helped us avoid handling too many separate files, its the same leaving .out and .err but less messy.*

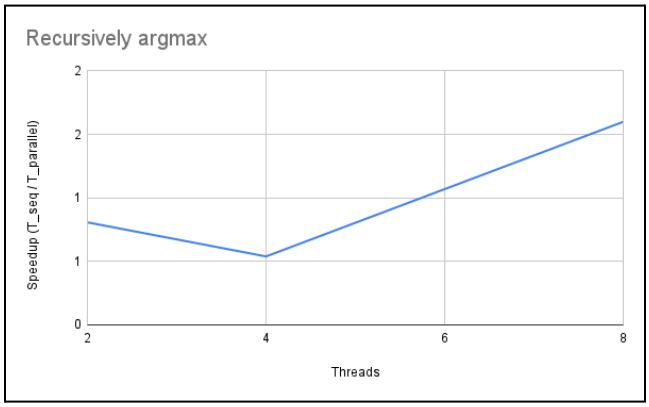
In the pictures below, one can see the list of all files received and the transcribed results in a Google spreadsheet for easier data management and to be able to plot the results:

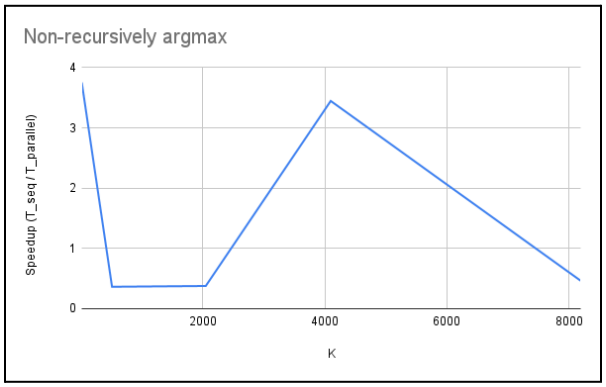


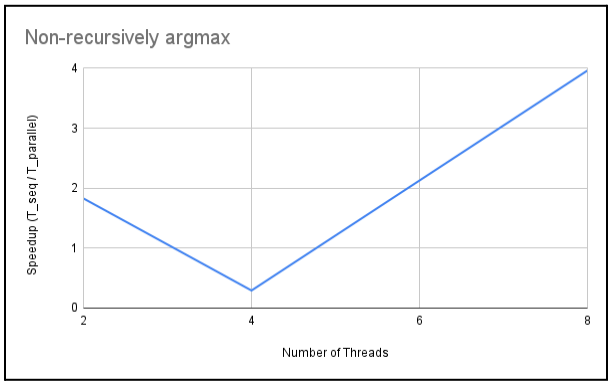
Threads	K	seq_time	par_time	rec_time	task_time		speedup Regula	Speedup sep
2	2048	2.30820E-02	1.26310E-02	1.05490E-02	1.30650E-02		1.82741E+00	8.07424E-01
4	2048	1.38950E-02	4.74560E-02	3.32980E-02	6.17830E-02		2.92798E-01	5.38951E-01
8	2048	1.36810E-02	3.45300E-03	9.27300E-03	5.79800E-03		3.96206E+00	1.59934E+00
Threads	K	seq_time	par_time	rec_time	task_time			
8	16	0.013718	0.003655	0.012519	0.829526		3.75321E+00	0.01509175119
8	512	0.016261	0.044720	0.022309	0.117930		3.63618E-01	0.1891715424
8	2048	0.018164	0.048234	0.021313	0.055680		3.76581E-01	0.3827765805
8	4096	0.024418	0.007085	0.016569	0.016088		3.44644E+00	1.029898061
8	8192	0.017529	0.037767	0.020405	0.035056		4.64135E-01	0.5820686901

Now we are ready to plot the results:









Now we are ready to analyze the results bycomparing the speedup of the two main implementations: the non-recursive and the recursive versions.

In the non-recursive version, we compared the sequential argmax_seq function with the parallel for-loop version argmax_par, which uses OpenMP and a critical section to avoid race conditions when updating the maximum value. When we plotted the speedup as the number of threads increased, we saw that the performance generally improved, especially going from 2 to 8 threads. However, there was a noticeable drop at 4 threads, which is probably due to the critical section slowing things down when several threads try to update the maximum at the same time. Even with that, the implementation still scaled decently with more threads. We also tested different values of K, even though the non-recursive version doesn't directly depend on it. Interestingly, we saw a peak around K = 4096, but in general, the variation with K wasn't very significant, since this version doesn't use it in the parallel part.

In the recursive version, we compared the sequential divide-and-conquer function argmax_recursive with the parallel task-based version argmax_recursive_tasks, which uses OpenMP tasks to split the work across threads. Here, the speedup with more threads was more consistent. Going from 2 to 8 threads gave us a clear improvement, though not quite linear. When we looked at how the speedup changed with different values of K (keeping 8 threads fixed), the results were much more dependent on that parameter. With small K values like 16 or 512, the program created too many tiny tasks, which led to a lot of overhead. On the other hand, when K was too large like 8192, there weren't enough tasks to keep all the threads busy. The best performance was around K = 2048 or 4096, where the amount of work per task was more balanced.

To sum up, both parallel implementations improved with more threads, but the recursive one was more sensitive to the value of K. The non-recursive version had a performance bottleneck because of the critical section, while the recursive version could reach better speedup if K was chosen well. These results show that tuning parameters like the number of threads and task size is essential to get the most out of parallel code.

3. What is the arithmetic intensity of the argmax algorithm? Which resource (memory bandwidth or peak computing capacity) do you expect to be the bottleneck for throughput?

To understand the arithmetic intensity of the argmax algorithm, we can think about how much computation it does compared to how much data it needs to move from memory. In this case, the algorithm just goes through a vector of size N, comparing each element to the current maximum. That's one floating-point comparison per element, and maybe an update if the condition is true. So, the number of operations per element is really small.

At the same time, every element of the vector needs to be read from memory, which means a lot of data is being moved compared to the small amount of computation. This gives us very low arithmetic intensity. Because of that, the limiting factor for performance isn't how fast the processor can compute, but how fast it can access memory. In other words, the bottleneck is the memory bandwidth, not the CPU's peak performance. This is pretty common in algorithms that just scan through large arrays and don't do much work on each element.

Remark: We decided to print both the error and output messages into the same .txt file, since it was easier and helped us avoid handling too many separate files.