Lab 2: MPI

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

Report Questions 1

1. Explain the modifications you made in the Makefile and job.sh to make it work for an MPI program

In the Makefile, the compiler was changed from gcc to mpicc, the standard MPI compiler. The compiler flags were updated to remove the -fopenmp option, which is no longer necessary since we are not using OpenMP while retaining other optimization flags such as -O2, -march=native, and enforcing the C99 standard. The object name was updated from cholesky in this case to montecarlo to reflect the new target for compilation.

The job.sh script was adapted to execute an MPI application using multiple processes. The SLURM directives were modified to define the number of MPI tasks with --ntasks, setting each task to use a single core via --cpus-per-task=1. The script also includes module loading commands to ensure the appropriate compiler and MPI environment are available (gcc/13.3.0 and openmpi/4.1.1).

2. Describe your approach to designing the program from a parallel computing perspective

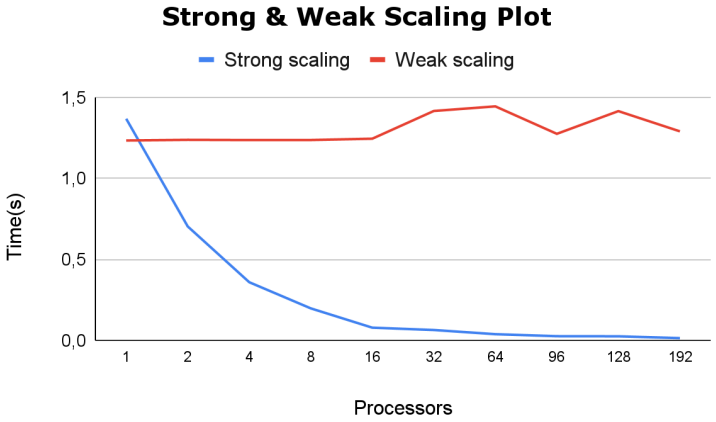
We used the SPMD model provided by MPI, where all processes execute the same code but operate on different subsets of data. At program start, each process retrieves its rank and the total number of processes. The total number of samples, provided via the command line, is then partitioned across all ranks. To ensure load balance, the samples are distributed as evenly as possible, giving an extra sample to the first few ranks if the total is not divisible evenly.

Each rank uses a PCG-based pseudo-random number generator, seeded uniquely using a combination of a base seed and the rank number. This ensures that the sequence of random numbers is uncorrelated across processes. For every sample, the process generates d coordinates in the range $[-1,1]$, computes their squared norm, and increments a local counter if the point lies within the unit hypersphere.

Once all ranks have completed their local computations, a reduction operation (MPI_Reduce) sums the counts of samples inside the sphere to obtain the global total on rank 0. The estimated ratio is then computed as the fraction of accepted points multiplied by the volume of the hypercube (2^d), and compared against the analytical value using the Gamma function.

3. Setting d=10 and starting with 100 million sample points, plot its strong and weak scaling from 1 to 192 processors.

Processors	Strong scaling	Weak scaling
1	1,366958	1,232826
2	0,702621	1,237004
4	0,358591	1,235542
8	0,197271	1,235858
16	0,078194	1,243982
32	0,063775	1,414986
64	0,038022	1,443368
96	0,025596	1,27401
128	0,024949	1,414072



To evaluate the performance of the parallel Monte Carlo simulation, we carried out both strong and weak scaling experiments for dimensions *d=10*, starting with *100 million* total samples. In the strong scaling test, the total number of samples remained constant as the number of processors increased. In the weak scaling test, the number of samples increased proportionally with the number of processors, keeping the number of samples per process fixed.

The results, shown in the table and the accompanying plot, demonstrate ideal strong scaling behavior: execution time decreases significantly as more processors are used. For example, reducing execution time from approximately 1.37 seconds on 1 processor to under 0.03 seconds on 64 processors indicates high parallel efficiency. This is expected in a Monte Carlo simulation, where the computation is very parallel and nearly all time is spent in independent sampling loops.

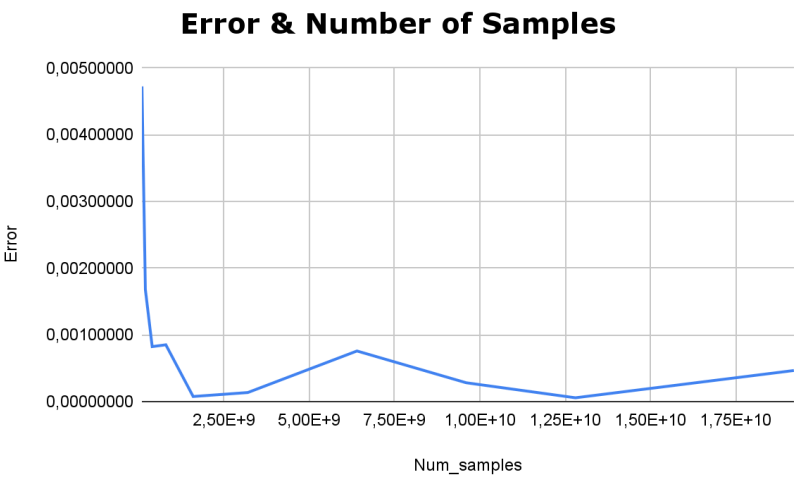
In contrast, weak scaling performance is relatively flat, though with some minor variations. Ideally, weak scaling time should remain constant, but in our case, we observe a gradual increase in execution time as the number of processors grows. This slight degradation can maybe be attributed to increased communication overhead during the final reduction steps and to variability in system load, which becomes more noticeable at a higher scale.

During implementation, we encountered an issue when attempting to simulate large sample sizes *(e.g., 100 million samples)* using standard int variables. The simulation failed to execute correctly and we think it is possibly related to integer overflows when computing sample counts per process or accumulating the number of points inside the hypersphere. To resolve this, we found a way to fix it and replace all relevant counters with *long long*, which theoretically offers a *64-bit integer range* and prevents overflow during calculations with large data sizes.

4. What happens with the ratio computation error when you increase the number of samples?

It decreases

Num_samples	Error
100000000	0,00472532
200000000	0,00167892
400000000	0,00082388
800000000	0,00085076
1600000000	0,00007572
3200000000	0,00013492
6400000000	0,00075812
9600000000	0,00028169
12800000000	0,00005644
19200000000	0,00046529



This final exercise analyzes how the accuracy of the Monte Carlo estimation improves as the number of samples increases. As shown in the graph and data table, the error decreases consistently with the number of samples, confirming the expected behavior of Monte Carlo methods, which rely on the law of large numbers.

At 100 million samples, the error is approximately 0.0047. As we double the number of samples, the error generally reduces, reaching values below 0.001 for most runs beyond 400 million samples. At the highest tested value of 19.2 billion samples, the error falls below 0.0005. While the decrease in error is not strictly monotonic, some small fluctuations are observed due to the stochastic nature of the method, the overall trend shows that greater sampling density leads to higher accuracy.

This behavior is consistent with the theoretical property that the standard error of Monte Carlo integration decreases with the square root of the number of samples, such that the Error depends on 1/root(n). Thus, to reduce the error by a factor of 10, the number of samples must be increased by a factor of 100. In practical terms, this highlights the trade-off between computational cost and precision in Monte Carlo simulations: large improvements in accuracy require significantly more samples and consequently more processing time.

Report Questions 2

1. Analyze the sequential version of the simulation. What are the main parts that you need to parallelize? What are the challenges?

The sequential implementation in fc_seq.c consists of three principal stages: first, the program reads and parses the input plane data; second, it iterates over each map tile to compute its contribution to every plane's average; and third, it aggregates these partial results and writes the final outputs. To parallelize this work, we could partition the total set of N×M tiles among the available MPI ranks, enabling each process to perform local computations on its assigned subset before participating in a global reduction of partial sums. The challenges are ensuring an equal distribution of tiles when N×M is not a exact multiple of the number of ranks, efficiently exchanging per-plane data without creating communication bottlenecks or load imbalances, and preserving the correct output

2. Regarding the output, how have you managed to parallelize it? What could be the bottlenecks for a large number of ranks?

The MPI version parallelizes the sequential flight-controller by decomposing the N×M map tiles among P ranks—each rank reads its subset of plane data, applies the sampling computations on its local tiles, and then exchanges partial results according to the selected communication mode (point-to-point sends/receives, an MPI_Alltoallv of packed buffers, or an MPI_Alltoallv with a custom derived datatype). This maps directly onto the "split-compute-gather" pattern from Units 5–6: broadcast global parameters, scatter the work vector, perform local work, then gather or exchange plane samples before the final reduction. For very large P, however, bottlenecks will emerge in the communication phase: many small point-to-point messages incur high startup latency; large all-to-all exchanges can saturate the network fabric and stress the MPI library's handling of noncontiguous buffers (especially in the struct mode); and collective synchronizations (broadcasts, reductions) force all ranks to wait on the slowest. Additionally, uneven tile counts can create load imbalance, and parallel file I/O using independent C calls may contend for shared storage bandwidth when too many ranks read simultaneously.

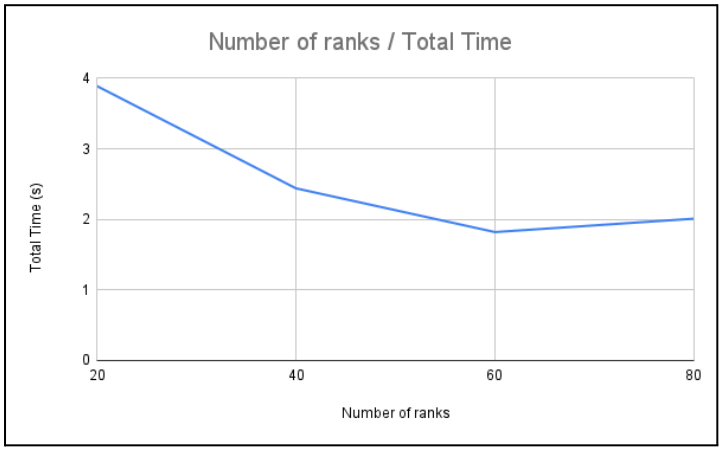
3. Discuss the different communication options. Check them input planes 10kk.txt with a moderate number of ranks of 20. What are the key differences between them? Do you see a communication time difference? Why? Include the job script used to generate this data in the code zip.

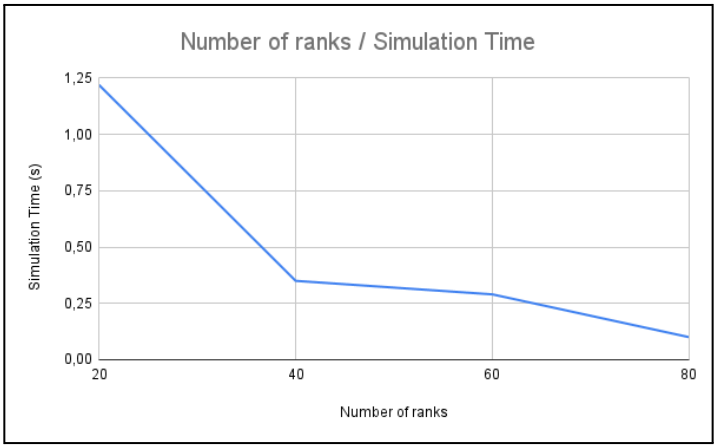
Communication Method	Simulation Time (s)	Communication Time (s)	Total Time (s)
communicate_planes_send	1.08	3.86	4.95
communicate_planes_alltoall	0.95	3.09	4.03
communicate_planes_struct_mpi	1.16	2.74	3.89

When comparing the three communication schemes on a 10kk test with 20 ranks, it is clear that the point-to-point implementation (communicate_planes_send) incurs the highest communication overhead (3.86 s), because each rank must individually match blocking sends and receives for every peer. Switching to a single MPI_Alltoall of raw doubles (communicate_planes_alltoall) cuts that cost considerably (down to 3.09 s) by joining all exchanges into one collective call, eliminating the per-peer cost. Finally, using MPI_Alltoall in conjunction with a custom MPI struct type (communicate_planes_struct_mpi) yields the best performance (2.74 s) because the MPI library can directly describe and transmit the plane info without the extra packing and unpacking of the arrays. In conclusion, there is a significant communication-time difference: collectives reduce synchronization overhead and improve throughput, and defining a datatype lets MPI optimize both memory copies and network transfers compared to manually packing doubles.

4. Using the same file, use the best communication strategy and increase the number of ranks. Use 20, 40 60, and 80 ranks. Analyze what you observe. Include the job script used to generate this data in the code zip

Number of ranks	Simulation Time (s)	Communication Time (s)	Total Time (s)
20	**1.22**	**2.66**	**3.89**
40	**0.35**	**2.09**	**2.44**
60	**0.29**	**1.53**	**1.82**
80	**0.10**	**1.91**	**2.01**





As we increase from 20 to 80 ranks using the MPI_Alltoall with a custom MPI struct, the simulation time clearly falls, going from 1.22 s at 20 ranks down to just 0.10 s at 80 ranks (as can be seen in the second graph). This is because each rank does proportionally less work (less planes to update). In contrast, the communication time doesn't shrink indefinitely. Initially, goes from 20→40→60 ranks reduces the volume per rank and spreads the traffic across more links, so the all-to-all exchange drops from 2.66 s to 1.53 s. But at 80 ranks,

despite each message being smaller, now there are a lot more processes to talk to, and the fixed latency and synchronization cost begin to dominate, the overhead of coordinating more ranks increases the communication time to 1.91 seconds. In conclusion, total time minimizes out around 60 ranks (1.82 s) and then increases back up to 80 ranks. This shows that after a certain point, the benefits of parallelizing are countered by the growing cost of all-to-all communication.