CS402 Coursework 2

Guney Coban - 2110391

I. Introduction

The Karman code simulates two-dimensional fluid dynamics by processing a binary (.bin) file that represents a discretised cell array of fluid velocities and pressures. The simulation advances incrementally through computed time-steps, iteratively solving fluid dynamics equations until convergence is achieved in each step. The main loop involves computing the time step, calculating tentative velocities, generating the RHS of the pressure equation, solving the pressure field via the Poisson equation, updating velocities, and finally applying boundary conditions [1]. Due to these dependencies, parallelisation across time steps is not feasible. Therefore, the parallel strategy employed divides the computational domain spatially, assigning different sections of the 2-dimensional cell array to separate MPI processes.

II. MPI Strategy

The parallelisation strategy employed in this coursework utilises a one-dimensional (1D) domain decomposition approach along the horizontal (i) axis. Given a computational grid defined by $imax \times jmax$, the grid is partitioned horizontally among n nodes, each handling an individual sub-grid of size approximately $\frac{imax}{n} \times jmax$ [2].

As `imax` is approximately five times larger than `jmax`, a 2D decomposition would nearly double the amount of inter-process communication compared to a 1D decomposition. This is because 1D decomposition requires exchanging only full columns of ghost data between neighbouring processes, while a 2D decomposition would involve sending both rows and columns. This is why a 1D decomposition was chosen.

To ensure efficient load balancing, when $imax$ is not evenly divisible by the number of nodes n, any extra columns resulting from the division are distributed evenly among the first r (remainder) nodes, rather than assigning them all to a single node. For instance, if we have $imax = 11$ and $n = 4$, the grid decomposition will be as follows:

- Node 0: columns [0, 1, 2]
- Node 1: columns [3, 4, 5]
- Node 2: columns [6, 7, 8]
- Node 3: columns [9, 10]

This distribution strategy minimises workload imbalance, particularly beneficial when dealing with a large number of nodes.

Now, each MPI process independently executes the main simulation loop on its assigned subsection of the global grid and at the end, the local results from each process are gathered to reconstruct the complete solution across the entire computational domain.

III. MPI Implementation

A. Global Array Decomposition

The initial setup of the global arrays and their distribution is managed by the root process (process 0). These arrays which store velocity components (u, v), pressure (p), and cell flags are initially constructed in the same way as in the serial version of the code.

Once the full global domain is set up, process 0 partitions and distributes the relevant subdomains to the other MPI processes. This is done according to the 1D horizontal decomposition strategy previously described. For each non-root process, the root process calculates the starting index and number of rows to be sent, then constructs a local buffer of the subgrid including the required ghost columns. These are essential for the correctness of numerical computations at subdomain boundaries and work by storing a copy of neighbouring data from adjacent processes.

Each process receives a subgrid that includes not only its own rows but also an additional column at the left and right to account for these ghost cells. For the root and final node, one of these ghost columns is the global boundary. This setup ensures that each process begins with the correct local data and the necessary ghost data.

Figure 1 visually represents the concept of ghost cells and boundary communication for a decomposition example with three processes.

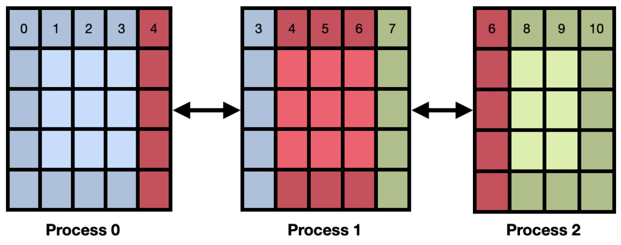


Fig. 1: Example 1D-Decomposition of a 10x5 grid with 3 processes.

B. Communication

Each function requiring computation on the arrays u,v,p and $flag$ had to be modified to accept rank and size as parameters and set up for accurate communication. Below are details regarding the modifications made to allow the different processes to communicate and ensure correctness in calculation for each function.

apply_boundary_conditions(): This standalone function in the boundary.c file is responsible for enforcing the physical boundary conditions of the simulation domain by modifying the velocity fields u and v based on the surrounding cells. In the parallel version, before any conditions are applied, the function performs a halo exchange using **MPI_Sendrecv** to share the current process' ghost rows of the velocity fields with neighbouring MPI processes 8. This ensures that each process has access to the most up-to-date boundary data from its adjacent subdomains before carrying out any calculations.

Once this data has been exchanged, the function applies the boundary conditions depending on the process's position in the domain. The leftmost process (rank 0) applies an inflow condition, while the rightmost process (rank n-1) allows fluid to exit freely. The function also enforces symmetry at the north and south boundaries of the grid and implements no-slip conditions around internal obstacle cells based on the flag matrix.

simulation.c: The rest of the utilised functions reside in simulation.c. To allow generalisation of the ghost column exchange, a helper function was defined which takes an array, rank and size and performs the required communication. For the rest of this section we are referring to this function (9) when discussing ghost column exchange.

compute_tentative_velocity(): This function calculates the tentative velocity fields f and g based on the current velocity values and fluid properties. While the core logic mirrors the serial version, the modified function uses the process rank and size, to correctly apply boundary conditions at the global domain edges. Specifically, only the first and last ranks apply updates to the leftmost and rightmost boundaries, respectively. Additionally, a ghost cell exchange is performed at the end of the function for both f and g, to ensure that neighbouring processes have consistent boundary data for the subsequent steps.

poisson(): This function solves the pressure Poisson equation using the Red/Black Successive Over-Relaxation (SOR) method to enforce the incompressibility condition of the flow [3]. Initial profiling revealed that this is the largest chunk of time taken in the program at around 95%, hence somewhere where we intend to make the most improvement.

In the parallel code, to maintain accuracy at subdomain boundaries, ghost cells are exchanged after each red and black sweep. Additionally, this function uses **MPI_Allreduce** to compute the global initial pressure norm and residual error, both of which are needed for convergence checking. Each process first calculates its local contribution, and **MPI_Allreduce** combines these into a global value to ensure all processes evaluate convergence against the same criteria. This guarantees consistent stopping conditions across the distributed system.

Finally, to preserve the red/black cell pattern across the global domain, the global row offset (*glob_index*) is used when determining cell colour, to ensure correctness.

update_velocity(): This function updates the velocity fields u and v using the corrected pressure values and the previously computed tentative velocities f and g. In the MPI version, the only modification is the use of the *rank* and *size* parameters to skip updating the final column of u to maintain the outflow boundary condition. No communication is required here, as all necessary values are locally available following the pressure solve and halo exchange.

set_timestep_interval(): This function determines the time step size Δt based on the Courant–Friedrichs–Lewy (CFL) condition, the Reynolds number, and the maximum velocities in the domain. In the parallel version, each MPI process first computes the local maximum values of u and v, and then uses **MPI_Allreduce** with the **MPI_MAX** operation to find the global maxima across all subdomains. This ensures that the computed Δt respects stability constraints globally, and that all processes advance in time using the same consistent step size.

C. Collation and Reconciliation

Once the simulation is complete, the distributed subgrids held by each MPI process are gathered and reassembled into the original global arrays by the root process (rank 0). Each non-root process sends its local portion of the domain (excluding the ghost columns) back to the root using **MPI_Send**. The root process calculates the correct insertion position based on the initial decomposition and receives each block using **MPI_Recv**, populating the global arrays row by row. This final step mirrors the distribution logic used at the beginning of the simulation and ensures that the global arrays reflect the complete solution across the domain. A final **MPI_Barrier** is used to synchronise all processes before writing the output to file.

IV. MPI Only Results

To evaluate the performance of the parallelised Karman code, a series of benchmarks were conducted across varying numbers of MPI processes. Two problem sizes were tested: a large grid of 1320×240 and a smaller grid of 660×120, allowing for comparison of scalability and efficiency across different computational loads. For each configuration, the total runtime and average time per iteration per function were recorded.

The simulations were run using 1, 2, 4, 8, 12, 16, and 20 MPI processes. Speedup was calculated relative to the single-node baseline for each problem size. All runs were executed using consistent initial conditions, with the root node distributing subgrids and gathering results at the end. The reported timings represent average values over all iterations once convergence was reached, ensuring consistency across problem sizes and process counts.

All results presented in this report have been verified for correctness using the diffbin utility, which compares the output .bin files from the parallel implementations against a reference file generated by the original serial code for each problem size.

Nodes	Large (1320×240)		Small (660×120)	
	Runtime (s)	Speedup	Runtime (s)	Speedup
1	174.78	1.00	19.94	1.00
2	102.60	1.70	10.84	1.84
4	63.25	2.76	6.08	3.28
8	46.53	3.76	7.77	2.57
12	40.90	4.27	8.99	2.22
16	39.43	4.43	10.70	1.86
20	40.79	4.28	12.35	1.61

TABLE I: Total Runtime and Speedup vs Number of Nodes

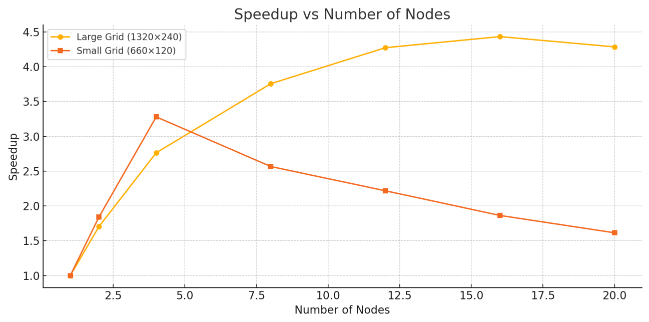


Fig. 2: Speedup vs Node Count

The runtime results for both the large (1320×240) and small (660×120) problem sizes demonstrate the expected benefits of parallelisation, with notable speedup as the number of MPI processes increases. For the large problem, runtime drops significantly from 174.78 seconds on a single node to 40.90 seconds with 12 nodes, achieving a speedup of approximately 4.27×. However, diminishing returns become apparent beyond 8–12 nodes, with runtime plateauing and even slightly increasing at 20 nodes. This is likely due to increased communication overhead outweighing computational gains, especially since ghost cell exchanges and global reductions become more frequent and costly.

In contrast, the small problem size achieves its best performance at 4 nodes (6.08 seconds), with a peak speedup of around 3.28×. Beyond this point, performance degrades slightly as communication overhead dominates the relatively modest computational load. The data confirms that parallel efficiency is strongly influenced by problem size. Larger problems scale more effectively because the cost of communication is amortised over more work, while smaller problems may become communication-bound more quickly.

If we profile by function as in Figure 3, we see that that RHS computation, velocity functions and boundary function benefit most from MPI parallelisation, showing strong and consistent scaling. These functions have minimal communication overhead, especially RHS which requires none, making them ideal candidates for domain decomposition.

Interestingly, the poisson function sees a relatively very early plateau in its speedup. This may be due to the requirement of global maximums which introduces synchronisation points and limits parallel efficiency. As the number of MPI processes increases, each process spends more time waiting for others to complete these collective operations, diminishing the overall benefit of additional parallelism despite the function's computational weight.

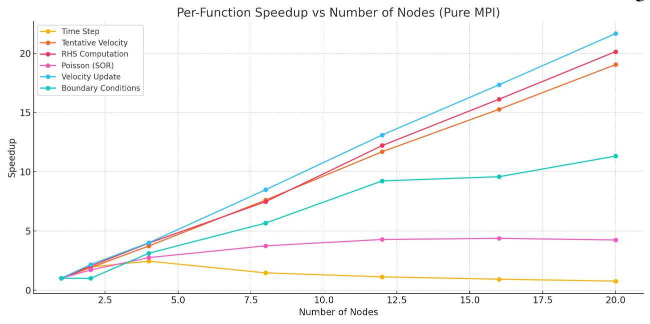


Fig. 3: Speedup by function (Large Grid Size)

Overall, the implementation validates the effectiveness of domain decomposition and MPI-based parallelism for the given problem, but also emphasises the importance of selecting an appropriate level of parallelisation relative to problem size to achieve optimal performance.

V. Hybrid Approach With OpenMP

To further improve performance, a hybrid parallelisation strategy was implemented by combining MPI for inter-node communication with OpenMP for shared-memory parallelism within each node [4]. This approach allows each process to utilise multiple threads on its allocated core group, improving CPU utilisation and reducing total runtime without increasing the number of MPI processes.

Integrating OpenMP into the existing MPI codebase was relatively straightforward, as many computational loops were already operating over independent grid cells with no cross-iteration dependencies, making them well-suited for parallel execution without requiring restructuring or logic changes. Parallel regions were introduced using **#pragma omp parallel for** [5] directives with static scheduling, which was suitable given the regular iteration patterns and consistent computational load across loop iterations.

Functions such as **poisson()** and **set_timestep_interval()** required more careful handling due to the presence of reductions. Both functions involve accumulating values across the domain and OpenMP reduction clauses were used to correctly manage shared variables within threads. These reductions complement the **MPI_Allreduce** calls used to consolidate values across processes, ensuring both intra- and inter-process consistency.

Initial testing also revealed a significant drop in performance when applying OpenMP to apply_boundary_conditions(), likely due to existing dependencies between iterations that limit the effectiveness of parallel execution. Hence the hybrid approach used at the end does not include this function.

VI. Hybrid Approach Results

Tests were conducted using 1, 2, 4 and 8 MPI processes and varying the number of OpenMP threads per process (1, 2, 3, and 4), on the large grid size (1320×240).

Nodes	Threads	Runtime (s)	Speedup
1	1	169.20	1.00
1	2	99.61	1.70
1	3	76.06	2.22
1	4	61.67	2.74
2	1	111.21	1.52
2	2	70.38	2.40
2	3	55.28	3.06
2	4	46.14	3.67
4	1	76.72	2.21
4	2	52.85	3.20
4	3	43.86	3.86
4	4	38.99	4.34
8	1	65.40	2.59
8	2	51.79	3.27
8	3	46.60	3.63
8	4	43.57	3.88

TABLE II: Hybrid MPI + OpenMP Runtime and Speedup Results

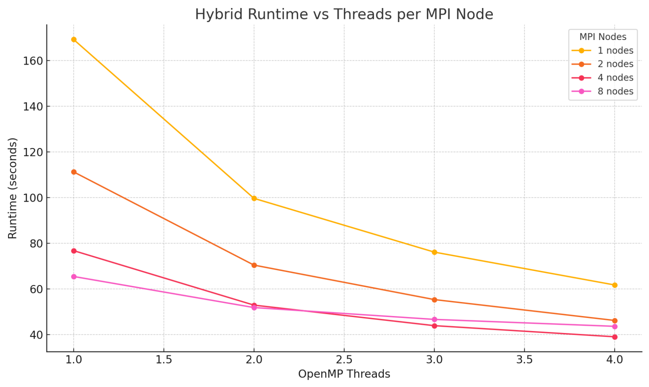


Fig. 4: Time Elapsed by Node and Thread Count (Large Grid)

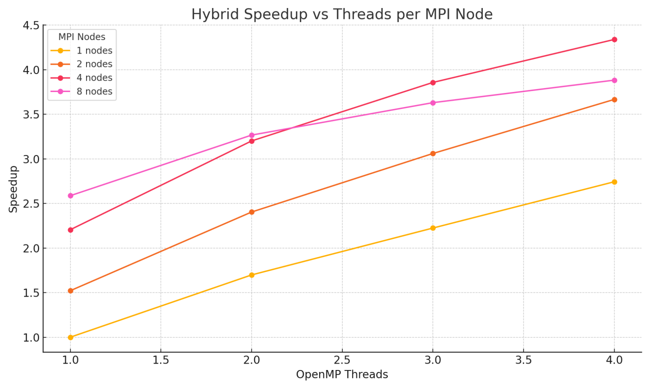


Fig. 5: Speedup by Node and Thread Count (Large Grid)

The hybrid implementation demonstrates clear improvements with the use of OpenMP, as runtime consistently decreases with an increasing number of threads for a fixed number of MPI processes. However, when comparing these results to the pure MPI implementation, it can be seen that the hybrid approach is often significantly slower at lower thread counts. This performance lag is likely attributed to the inherent overhead of managing threads, which becomes especially noticeable when the actual parallel workload per thread is small.

At lower node counts, specifically 1, 2, and 4, the impact of the hybrid approach is particularly impressive. By supplementing each MPI process with multiple threads, the program achieves runtimes that are not only faster than their MPI-only counterparts but also comparable to those achieved using more MPI nodes. For instance, the configuration with 2 MPI nodes and 4 threads matches the performance of the 4-node MPI setup, highlighting the hybrid model's ability to compensate for limited distributed resources.

However, when scaling to 8 MPI nodes, the hybrid benefits begin to taper off. Although adding threads still improves runtime, the rate of improvement diminishes, and the final configuration (8 MPI nodes with 4 threads) yields only a marginal speedup over the pure MPI equivalent. This plateau likely results from increased complexity in thread coordination, memory bandwidth contention, and communication overhead that grows with both thread and process count. Moreover, not all functions benefit equally from threading, meaning that beyond a certain point, the hybrid model is unable to fully utilize the available computational resources.

To understand these diminishing returns, we profile the performance of each function individually (Table IV).

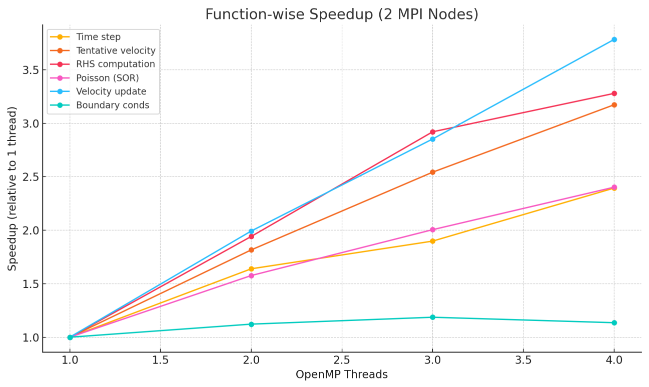


Fig. 6: Speedup by function (2 Nodes)

At 2 MPI nodes, threading leads to strong and consistent speedups across all modified functions with near linear scaling up to 4 threads (Fig. 6).

By contrast, at 8 nodes, function-level speedup is far less pronounced (Fig. 7). Poisson() in particular shows limited improvement, likely due to frequent MPI reductions and halo exchanges, which dominate runtime as computation shrinks. Considering the domination of the poisson function on the program's overall runtime, this explains the lack of speedup achieved from the OpenMP modifications at higher node counts.

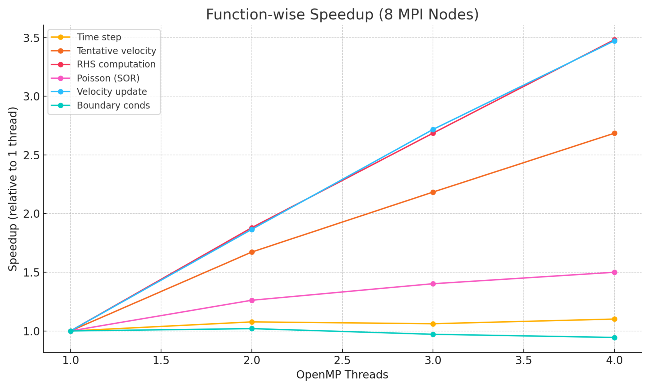


Fig. 7: Speedup by function (8 Nodes)

These trends highlight that OpenMP is most effective when each MPI process has sufficient local work to amortize threading costs, something increasingly lost at higher node counts.

VII. Conclusion

The MPI implementation of the karman code delivered significant performance improvements by effectively distributing the computational workload and memory usage across multiple nodes. Through the 1-Dimensional domain decomposition approach and efficient communication patterns, the parallel implementation achieved almost a 5x speedup for larger problem sizes, while maintaining correctness.

Building on this foundation, the hybrid MPI + OpenMP approach further enhanced performance by leveraging shared-memory parallelism within each node. By parallelising most of the computational loops at the thread level, the hybrid model reduced runtime even further compared to the pure MPI version - particularly at lower node counts.

Overall, the hybrid implementation demonstrates that combining distributed and shared-memory parallelism is a practical and impactful strategy for the given simulation code where there is an opportunity to decompose the problem domain into mostly independent subgrids.

References

[1] He L. Introduction to the Karman CFD Code and the SOR Pressure Solver; 2025. University of Warwick. Lecture slides, CS402 High Performance Computing. Available from: https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs402/coursework2-background.pdf.

[2] Smith B, Bjorstad P, Gropp W. Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press; 1996. Available from: https://www.cambridge.org/core/books/domain-decomposition/4717B2EAF355C01A92DDE037BE9C460E.

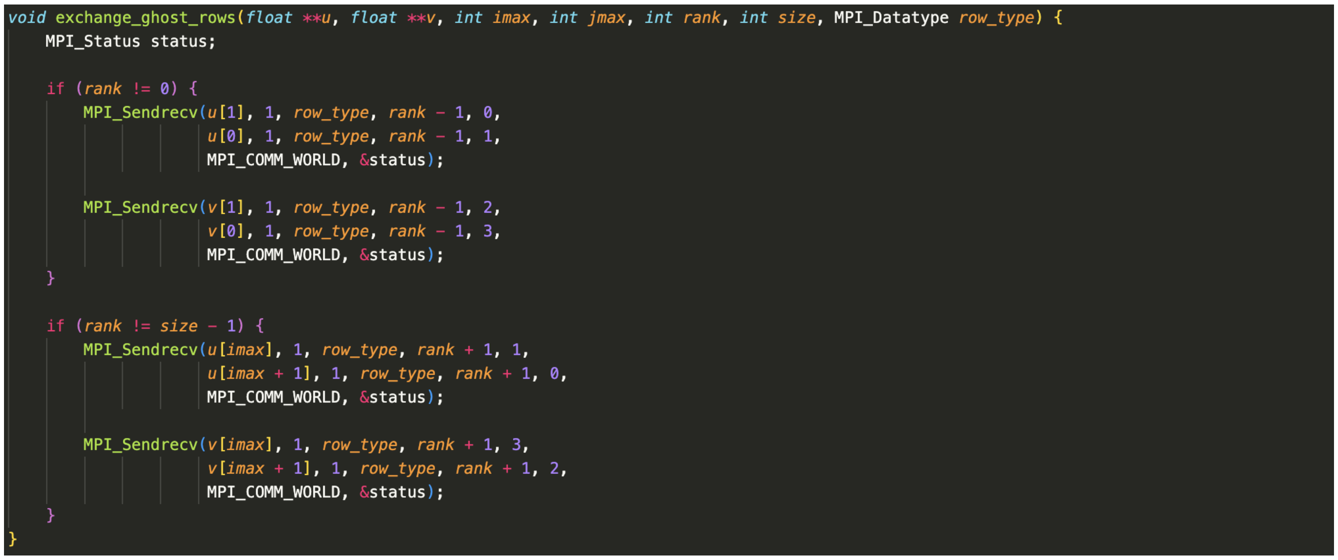
[3] Young DM. Iterative Solution of Large Linear Systems. Academic Press; 1971.

[4] Dagum L, Menon R. OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering. 1998;5(1):46-55.

[5] OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 5.0; 2018. Https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

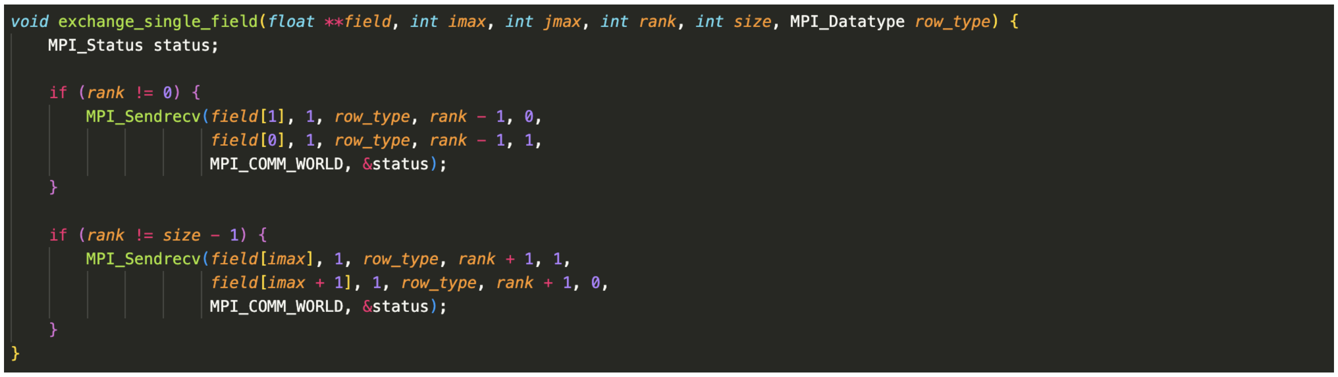
Appendix

The submission files **simulation.c** and **karman.c** use the hybrid approach. The code used to generate results for the MPI only approach are suffixed **.MPI**.



```c
void exchange_ghost_rows(float **u, float **v, int imax, int jmax, int rank, int size, MPI_Datatype row_type) {
    MPI_Status status;

    if (rank != 0) {
        MPI_Sendrecv(u[1], 1, row_type, rank - 1, 0,
                     u[0], 1, row_type, rank - 1, 1,
                     MPI_COMM_WORLD, &status);

        MPI_Sendrecv(v[1], 1, row_type, rank - 1, 2,
                     v[0], 1, row_type, rank - 1, 3,
                     MPI_COMM_WORLD, &status);
    }

    if (rank != size - 1) {
        MPI_Sendrecv(u[imax], 1, row_type, rank + 1, 1,
                     u[imax + 1], 1, row_type, rank + 1, 0,
                     MPI_COMM_WORLD, &status);

        MPI_Sendrecv(v[imax], 1, row_type, rank + 1, 3,
                     v[imax + 1], 1, row_type, rank + 1, 2,
                     MPI_COMM_WORLD, &status);
    }
}
```

Fig. 8: Helper function for ghost column exchange of u and v in boundary.c.



```c
void exchange_single_field(float **field, int imax, int jmax, int rank, int size, MPI_Datatype row_type) {
    MPI_Status status;

    if (rank != 0) {
        MPI_Sendrecv(field[1], 1, row_type, rank - 1, 0,
                     field[0], 1, row_type, rank - 1, 1,
                     MPI_COMM_WORLD, &status);
    }

    if (rank != size - 1) {
        MPI_Sendrecv(field[imax], 1, row_type, rank + 1, 1,
                     field[imax + 1], 1, row_type, rank + 1, 0,
                     MPI_COMM_WORLD, &status);
    }
}
```

Fig. 9: Generalised helper function in simulation.c for ghost column exchange of any array.

MPI	Time Step	Tentative Velocity	RHS	Poisson (SOR)	Velocity Update	Boundary Conds
1	0.000715	0.008535	0.000806	0.412677	0.001127	0.000249
2	0.000367	0.004509	0.000400	0.242774	0.000524	0.000252
4	0.000293	0.002293	0.000203	0.150564	0.000282	0.000080
8	0.000493	0.001123	0.000108	0.110424	0.000133	0.000044
12	0.000641	0.000730	0.000066	0.096580	0.000086	0.000027
16	0.000779	0.000559	0.000050	0.094477	0.000065	0.000026
20	0.000946	0.000448	0.000040	0.097451	0.000052	0.000022

TABLE III: Average Per-Iteration Function Timings (s) for Large Problem Size (1320×240) (MPI ONLY)

MPI	Threads	Time step	Tentative velocity	RHS computation	Poisson (SOR)	Velocity update	Boundary conds
1	1	0.000710	0.008515	0.000804	0.390451	0.001130	0.000281
1	2	0.000362	0.004489	0.000455	0.230362	0.000637	0.000287
1	3	0.000251	0.003162	0.000302	0.176166	0.000479	0.000297
1	4	0.000189	0.002395	0.000234	0.142988	0.000395	0.000295
2	1	0.000541	0.004622	0.000400	0.257594	0.000522	0.000477
2	2	0.000330	0.002545	0.000206	0.163413	0.000262	0.000425
2	3	0.000285	0.001818	0.000137	0.128472	0.000183	0.000402
2	4	0.000226	0.001457	0.000122	0.107243	0.000138	0.000420
4	1	0.000772	0.002455	0.000192	0.178334	0.000253	0.000231
4	2	0.000632	0.001341	0.000101	0.123083	0.000134	0.000245
4	3	0.000602	0.001010	0.000068	0.102156	0.000091	0.000252
4	4	0.000572	0.000832	0.000061	0.090825	0.000069	0.000263
8	1	0.001176	0.001286	0.000094	0.152467	0.000125	0.000204
8	2	0.001092	0.000769	0.000050	0.120850	0.000067	0.000200
8	3	0.001108	0.000589	0.000035	0.108712	0.000046	0.000210
8	4	0.001068	0.000479	0.000027	0.101661	0.000036	0.000216

TABLE IV: Average function timings (s) for Large Problem Size (1320×240) (MPI and OpenMP).