
This is the **published version** of the bachelor thesis:

Sharma, Arun. *Dynamic Process Reconfiguration in OpenFOAM : A Malleable HPC Approach*. Treball de Final de Grau (Universitat Autònoma de Barcelona), 2026 (Enginyeria Informàtica)

This version is available at <https://ddd.uab.cat/record/334406>

under the terms of the  license.

Dynamic Process Reconfiguration in OpenFOAM: A Malleable HPC Approach

Arun Sharma

30 de juny de 2026

Resum– La mal·leabilitat és la capacitat d'un codi de canviar el nombre de nuclis que utilitza durant l'execució, en coordinació amb el planificador. El seu principal benefici és permetre el control de mètriques de cost de la simulació, com ara el temps d'execució, l'eficiència paral·lela i el consum de recursos. Presentem una implementació mal·leable d'OpenFOAM que integra la biblioteca Dynamic Management of Resources (DMR), amb les decisions de reconfiguració guiades per una mesura en temps d'execució de l'eficiència de comunicació, és a dir, la fracció del temps total dedicada a càlcul útil. El flux de treball s'ha validat al superordinador MareNostrum 5 amb el cas d'aerodinàmica externa drivAer (aproximadament 92 milions de cel·les). Partint tant d'una assignació sobre- com d'una infra-dimensionada, el flux convergeix autònomament a la mateixa regió eficient, i, al llarg de nou reconfiguracions, el coeficient de resistència es preserva dins de la variabilitat pròpia de la redescomposició (aproximadament l'1,5%).

Paraules clau– Mal·leabilitat, Computació d'altres prestacions (HPC), Dinàmica de fluids computacional (CFD), OpenFOAM, MPI, Dynamic Management of Resources (DMR), eficiència de comunicació, MareNostrum 5.

Abstract– Malleability is the ability of a code to change the number of cores it uses at runtime, in coordination with the scheduler. Its main benefit is to enable control over simulation cost metrics such as execution time, parallel efficiency, and resource consumption. We present a malleable build of OpenFOAM that integrates the Dynamic Management of Resources (DMR) library, with reconfiguration decisions driven by a runtime measurement of communication efficiency, the fraction of total runtime spent on useful computation. The workflow has been validated on the MareNostrum 5 supercomputer with the drivAer external aerodynamics case (approximately 92 million cells). Starting from either an over- or an under-provisioned allocation, the workflow converges autonomously to the same efficient node range, and across nine reconfigurations the drag coefficient is preserved to within the decomposition-noise floor of the fixed-allocation runs (about 1.5%).

Keywords– Malleability, HPC, CFD, OpenFOAM, MPI, Dynamic Management of Resources (DMR), Communication Efficiency, MareNostrum 5.



1 INTRODUCTION

HIGH-Performance Computing (HPC) solves problems too large for one computer by combining many. Each computer is a *node*, and the proces-

ses of a parallel program running across nodes communicate by exchanging messages through the Message Passing Interface (MPI) [1]. Programs are submitted as jobs to a batch scheduler, most commonly Slurm [2], which places each job on a set of nodes and queues the rest when the machine is full. Computational Fluid Dynamics (CFD), the numerical simulation of fluid motion, is among the largest consumers of supercomputer time, underpinning the design of aircraft, vehicles, turbines and reactors with meshes of tens of millions of cells and hours of wall-clock time per study. OpenFOAM [3], the open-source toolbox maintained by the OpenFOAM Foundation, is widely regarded as

- E-mail de contacte: arun.sharma@autonoma.cat
- Menció: Enginyeria de Computació
- Treball tutoritzat per: Francisco Javier Sanchez Pujadas (Departament de Ciències de la Computació.) Sergio Iserte Agut (Barcelona Supercomputing Center)
- Curs 2025/26

the most used open-source CFD solver, deployed by both academic groups and industrial users.

OpenFOAM simulations are launched on a fixed number of processes and keep that allocation until the job ends. This is convenient for the batch system that schedules jobs on a supercomputer, but it is suboptimal for the application: how efficiently an OpenFOAM run uses its processes is not constant. It depends on the mesh, on the physics being solved, and on the ratio of cells per process. A simulation that is well balanced at launch can become communication-bound after a mesh refinement, and the converse can also occur. Without a mechanism to reshape the allocation at runtime, neither the user nor the scheduler can react to these changes.

Malleability is the ability of a code to adapt the resources it uses, primarily the number of CPU cores, at runtime, in coordination with the batch system. Its main benefit is to enable control over simulation cost metrics such as wall-clock time, parallel efficiency, and resource consumption (measured in node-hours). The Dynamic Management of Resources (DMR) library [4], developed at the Barcelona Supercomputing Center (BSC), exposes this capability to applications through a generic checkpoint and restart cycle, together with a policy interface that decides when to reconfigure. Runtime efficiency measurements are provided to DMR by the TALP module of the Dynamic Load Balancing (DLB) framework [5], also developed at BSC.

This work integrates DMR with OpenFOAM and uses the communication-efficiency measurements provided by TALP to produce a malleable CFD workflow that keeps the simulation in an efficient region of resource usage while preserving the physical results of the corresponding fixed-allocation run. To the best of our knowledge, this is the first integration of DMR with OpenFOAM. Along the way we characterise DMR's efficiency-driven policy as a proportional controller and study how its gain governs the speed and stability of convergence. The resulting workflow has been validated on the MareNostrum 5 supercomputer with the *drivAer* case, and its generality is confirmed on a second, moving-mesh case in the appendix.

The remainder of the document is organised as follows. Section 2 reviews the state of the art, including OpenFOAM, the DMR runtime, and prior work on malleable CFD. Section 3 states the objectives. Section 4 describes the integration methodology. Section 5 details the experimental setup. Section 6 presents and discusses the results. Section 7 concludes, and Section 8 outlines the lines of continuation. A final section documents the use of generative AI, and an appendix collects supporting material, including a second, moving-mesh test case and additional implementation and reproducibility details.

2 STATE OF THE ART

2.1 OpenFOAM and its Parallel Architecture

OpenFOAM (Open Field Operation And Manipulation) is an open-source toolbox for simulating fluid flows [3]. It solves the governing equations of fluid motion numerically on a discretized geometry called a *mesh*. From a description of the problem (geometry, boundary conditions, fluid properties), OpenFOAM advances the solution through a *time loop*: each pass discretises the equations into a large sparse linear

system and solves it to update the fields (velocity, pressure, temperature). The loop repeats, either marching through physical time or, for a steady-state case such as the one here, until the fields stop changing. At a user-chosen cadence (the *writeInterval*) the current fields are written to disk, both to save results and to let a long run resume from its last snapshot after a crash or a wall-clock-time limit. These periodic snapshots are OpenFOAM's *natural writes*, and the malleable workflow builds on this existing save-and-resume machinery rather than adding its own.

The toolbox is maintained by the OpenFOAM Foundation under an open-source licence and is used both by academic researchers and by industrial engineering teams. Its solvers cover a broad range of problems, from low-speed flows around vehicles to turbulent combustion and multiphase flows with free surfaces. By the OpenFOAM Foundation's own characterization, the codebase is of the order of 1.5 million lines of code [6], organized into modular solver applications, mesh utilities and physical models.

Most realistic OpenFOAM simulations are too large for a single computer. The standard practice is to split the domain into as many subdomains as there are processes, one per process. A process owns the cells of its subdomain and, on every pass of the time loop, exchanges a thin layer of boundary values, the *halo*, with the processes that own the neighbouring subdomains, so that the discretised equations stay consistent across the cuts. These halo exchanges are point-to-point between face-sharing neighbours, not a global synchronisation. The only collective communication per iteration is a small number of global reductions inside the linear solver, where convergence is tested on a residual norm gathered from all processes. This decomposition is performed once, before the run, by a dedicated utility, and the subdomains are then fixed for the whole simulation. The number of processes is therefore decided when the job is submitted to the batch system and does not change while it runs.

How efficiently this scheme runs depends on how many cells each process owns. When a process owns many cells, it spends most of each iteration computing, and the halo it exchanges with its neighbours is small by comparison. Splitting the domain across more processes leaves each one with fewer cells, so the computation per process shrinks while the halo exchange and the global reductions do not shrink as fast, and they take up a growing share of every iteration until the run becomes communication-bound. There is therefore an efficient middle ground rather than a simple rule that more nodes are always faster, and finding it at runtime is what motivates the communication-efficiency signal introduced next.

This fixed-process assumption is the structural constraint that makes a malleable build of OpenFOAM a non-trivial integration problem: the partition is computed for a specific process count, the internal communication structure is built once for that count, and changing the count requires coordinating both pieces with the running solver state.

2.2 Malleability and the DMR Runtime

Parallel jobs differ in how much control they grant the resource manager over their size [7]. A *rigid* job fixes its process count for its entire lifetime. A *modable* job lets

the scheduler choose that count once, at launch, but not afterwards. A *malleable* job goes further and changes it *during* execution, in coordination with the resource manager. A malleable application can therefore release processes when fewer are enough, or request additional ones when more would help, without restarting from scratch and without the user intervening.

Several research projects have made MPI applications malleable. Some extend MPI itself with primitives that let a running job grow or shrink (ULFM [8], originally designed for fault tolerance, and Elastic-MPI [9], designed specifically for dynamic resizing). Other programming models, such as Charm++ [10], support process migration as part of their normal execution model. These approaches share one trait: the application has to be written, or substantially rewritten, around the runtime’s primitives.

The Dynamic Management of Resources (DMR) library [4] follows a different design. Instead of requiring new MPI primitives in the application, it offers a generic *checkpoint and restart* cycle through a small API. The application calls three hooks, wrapped in the `DMR_AUTO` macro: `dmr_init` to initialise the library, `dmr_check` to consult it at a synchronisation point such as the end of each time-loop iteration, and `dmr_finalize` to shut it down. At each check a policy decides whether to reconfigure and by how much, and since each process usually occupies a core, a reconfiguration changes the allocation a whole node at a time. When a reconfiguration is triggered, DMR shuts the running job down cleanly and restarts it on the new allocation. Carrying the application’s state across the change in process count is the responsibility of the application, through the user-supplied checkpoint and restart hooks, which build on whatever persistence it already has. The two policies used in this work, `TEST_POLICY` (which we call *Round*) and `CE_POLICY`, are described in Section 4.3, and the full API is documented in the DMR manual [11].

This design fits applications, like OpenFOAM, that were not written for malleability and that already have a robust mechanism to save and load their state. The integration only needs to bridge two existing pieces: OpenFOAM’s saved-state utilities, and DMR’s reconfiguration cycle.

To decide when to reconfigure, DMR needs a metric it can read while the simulation runs. In this work that metric is the *communication efficiency* (CE), as defined by the POP performance methodology [12] and measured by TALP, a profiling module of the Dynamic Load Balancing (DLB) framework [5]. CE is the maximum, across all processes, of the fraction of the total runtime that a process spends on useful computation. It is a single number between zero and one: a value close to one means the processes spend almost all of their time computing, while a value close to zero means they spend it waiting on one another through MPI.

CE is the right criterion for sizing the allocation. A run split too finely is communication-bound, so its CE is low, and returning processes to the scheduler raises it by giving each one more cells. A run with ample work per process is compute-bound, so its CE is high and it can absorb more processes before communication dominates. Steering CE toward a target therefore steers the process count toward the efficient middle ground. We prefer CE to measures such as speed-up or wall-time per iteration because those are rela-

ve, meaningful only against a reference run at another size that is not available while the job is resizing itself, whereas CE is meaningful on its own and read live. This matters as machines grow toward exascale: a simulation spread across ten thousand cores at a low CE is wasting most of them, and CE exposes that waste in time to act on it. CE is also normalised, so a single target, 0.80 here, means the same thing on any case and machine, and one policy works across OpenFOAM’s solvers without retuning.

That last property, the low cost of reading CE, comes from how TALP measures it. Through the standard MPI profiling interface, TALP intercepts the MPI calls the solver already makes and accumulates, per process, the time spent inside MPI against the time spent computing outside it. CE is then the share of useful computation in the total. Because TALP only times calls the solver already issues, the measurement adds no communication of its own, and the profiling it does is lightweight, with low and well-bounded overhead [13]. The communication-efficiency metric needs only this MPI profiling and not the optional hardware-counter sampling that accounts for most of TALP’s cost, so on our runs the profiling overhead stays under five percent of execution time and reading CE does not meaningfully perturb the simulation. DMR reads the resulting CE while the job runs and uses it to decide whether, and by how much, to change the process count.

2.3 Prior Work on Malleable CFD

Malleability has been demonstrated on a number of parallel applications, ranging from compact benchmarks (NAS Parallel Benchmarks, stencil codes, conjugate-gradient kernels) to small finite-element prototypes [14]. These targets are valuable because their simplicity makes the integration tractable and the analysis clean, but their size and structural simplicity also limit what their results say about production codes.

The closest prior work for the present integration is the line of malleability experiments led by the DMR group itself on ALYA [15], the in-house parallel multiphysics solver developed at BSC. ALYA covers a similar problem class to OpenFOAM, fluid mechanics with extensions to coupled physics, and its malleable build is the existing reference point for malleability applied to a real CFD code. The ALYA work has demonstrated that the *checkpoint and restart* pattern is sound on a production-scale solver, and has informed the design of DMR itself.

The present work extends that line in two directions. First, OpenFOAM is open-source and community-maintained, while ALYA is internal to BSC. Bringing malleability to OpenFOAM places the same capability in the hands of the broader CFD community, both academic and industrial. Second, OpenFOAM exposes a larger and more heterogeneous ecosystem of solvers and physical models than the ALYA configurations previously reported, so the integration extends malleability to a broader range of solver types in principle, and the present campaign exercises it on representative production cases.

3 OBJECTIVES

The general objective of this work is to integrate the DMR runtime into OpenFOAM and to validate the resulting malleable workflow under realistic HPC conditions. It is pursued through the following specific objectives.

- **OE1.** Integrate the DMR runtime into OpenFOAM so that checkpoint, redistribution and restart cycles are triggered transparently within the solver's time loop, without any modification to the solver's numerical core.
- **OE2.** Implement a state-transfer mechanism that preserves field data, boundary conditions and solver state across reconfigurations, so that simulations resume correctly on the new process count.
- **OE3.** Implement the integration for both OpenFOAM single-region (`foamRun`) and multi-region (`foamMultiRun`) launchers, so that the malleable workflow is available across OpenFOAM's solver families.
- **OE4.** Evaluate the malleable workflow on MareNostrum 5, quantifying reconfiguration overhead, process-count dynamics and communication-efficiency trajectories under realistic simulation conditions.
- **OE5.** Verify that the physical results produced by the malleable workflow remain consistent with those of equivalent fixed-allocation executions, within the variability introduced by parallel redecomposition.
- **OE6.** Assess the integration's suitability for long-term maintenance and upstream adoption within the OpenFOAM Foundation ecosystem, identifying the architectural changes required for community contribution.

4 METHODOLOGY

This section describes the design and implementation of the malleable OpenFOAM workflow. The integration combines OpenFOAM's existing checkpoint and domain-decomposition mechanisms with the DMR runtime. Section 4.1 describes how DMR is embedded into OpenFOAM's launchers, and Sections 4.2 and 4.3 explain the redistribution workflow and the reconfiguration policies, respectively.

4.1 Integration into OpenFOAM

OpenFOAM ships two generic solver *launchers*, and the integration hooks both. They are not physical solvers: each loads the solver named in the case dictionary and runs its time loop, so hooking the two launchers makes every one of OpenFOAM's physical solvers malleable rather than any single one. `foamRun` runs single-region cases, such as the flow around a vehicle, and `foamMultiRun` runs multi-region cases where several regions coexist and exchange information at their boundaries, such as conjugate heat transfer between a fluid and a solid.

The integration is opt-in along two axes. At compile time, the build defines the guard `FOAM_USE_DMR` automatically when DMR is reachable in the environment. When it

is not, no DMR symbol is linked and the resulting binary is byte-for-byte vanilla OpenFOAM. At run time, the binary accepts a command-line flag `-dmr` that, when absent, makes a DMR-built launcher behave as the vanilla launcher. When present, it activates the malleability pipeline for the run. The flag is preferred over an environment variable because it scopes malleability to a single command and is explicit in the launch line, rather than being inherited by every process the job spawns.

The three DMR calls are inserted at three positions in the launcher source (Figure 1). Because DMR needs an MPI session to manage, the launcher first initialises MPI itself, and only then calls `dmr_init`, before any OpenFOAM object is constructed. OpenFOAM initialises MPI later only when DMR has not, that is when DMR is absent or the `-dmr` flag is not given. The `-dmr` flag is parsed by hand here, because OpenFOAM's own argument-parsing objects do not exist yet. `dmr_check` is placed inside the time loop, right after the per-step call to `runTime.write()`. That call writes the fields to disk only on the steps that match the user's `writeInterval` and does nothing on the others, so placing `dmr_check` after it lets a reconfiguration reuse a natural checkpoint when one has just been written and forces its own otherwise. `dmr_finalize` is the last call before `return`.

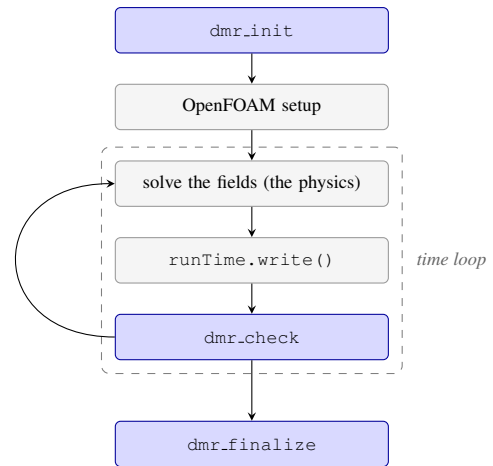


Fig. 1: Placement of the three DMR calls inside `foamRun`. Each pass of the time loop solves the discretised equations to update the fields (the physics), calls `runTime.write()` (which writes the fields to disk only at the user's `writeInterval`), and then consults `dmr_check`, which decides whether to reconfigure before the next step. The same three calls are inserted at the same three positions in `foamMultiRun`, with the multi-region preprocessor switch `DMR_ALL_REGIONS` defined before the includes.

The corresponding code lives in three short header files (`dmrInit.H`, `dmrCheck.H`, `dmrFinalize.H`) under `src/OpenFOAM/include/`, which both launchers `#include` at the three positions described above. The only difference between the two is a single preprocessor switch (`#define DMR_ALL_REGIONS`) defined before the includes, which informs the shared code that the multi-region variants of the redistribution utilities must be used. The redistribution utilities themselves are implemented in a new OpenFOAM module, `libfoamDmr`, at

src/parallel/dmr/. The next two subsections describe what these utilities do.

4.2 OpenFOAM Reconfiguration

A *checkpoint* is a complete on-disk snapshot of the simulation state at a given instant (mesh, fields, current time, control parameters), sufficient to resume the simulation from that instant on a different process count. A *restart* loads the snapshot on a new allocation and continues from that instant.

OpenFOAM already stores its state in a parallel layout: each process writes its share to its own subdirectory (*processor0/*, ..., *processor(N-1)/*). The integration reuses this layout and two existing utilities. *reconstructPar* merges the per-process subdirectories into a single serial case, and *decomposePar* partitions a serial case into a chosen number of subdomains.

A reconfiguration proceeds in three coordinated steps (Figure 2), each separated by an *MPIBarrier*, a point every rank must reach before any of them moves on. The barriers matter because the heavy work between the steps, above all the serial *reconstructPar*, is done by rank zero alone. The other ranks must wait rather than advance into the next iteration or read state that rank zero has not finished writing.

On the running side, all ranks flush their state to disk in parallel, and the first barrier waits for every write to land. Rank zero then runs *reconstructPar -newTimes* as a subprocess, which merges every time directory that does not yet have a serial counterpart: the checkpoints the user requested through *writeInterval*, plus any extra ones DMR forced at reconfiguration instants. A second barrier holds the old ranks until reconstruction is complete, after which DMR releases the allocation.

On the restart side, DMR launches the solver on the new allocation. Each rank queries the new size via *MPI_Comm_size*, and rank zero updates the decomposition dictionary (*system/decomposeParDict*) to the new size, sets the simulation control dictionary to start from the latest time, and runs *decomposePar* as a subprocess. Some cases carry extra, pre-computed meshes that are swapped in during the run. The usual example is an internal combustion engine, where the geometry changes as the pistons and valves move. These meshes live under *constant/meshes/*, and the *-allRegions* flag does not visit them, so each one needs its own *decomposePar* call on restart. The reconstruction side needs no such loop, because these sub-meshes are fixed inputs the simulation never modifies. The running mesh itself does move, and OpenFOAM saves its full shape only now and then rather than every step, so on restart *decomposePar* is given the chain of saved times from the last full mesh up to the restart time, which lets it carry the moving mesh forward to the exact position it had when the run stopped [16]. A final barrier synchronises the new ranks before OpenFOAM's parallel initialisation reads the freshly partitioned case.

4.3 Reconfiguration Policies

The DMR integration uses two of DMR's policies. The *Round* policy (*TEST_POLICY* in DMR's vocabulary) re-

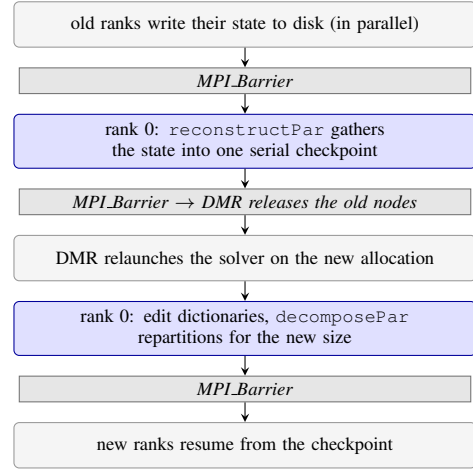


Fig. 2: The checkpoint and restart mechanism of one reconfiguration. The shaded steps (rank zero) run serially. The barriers are necessary, not merely tidy: the first stops rank zero from starting *reconstructPar* before every rank has finished writing its processor subdirectory, and the later ones stop ranks from advancing or reading state that rank zero has not finished writing.

configures deterministically each time the inhibitor window elapses, cycling through node counts between user-configured bounds $N_{\min}$ and $N_{\max}$ by doubling at each reconfiguration until $N_{\max}$ is reached, then resetting to $N_{\min}$. With $N_{\min}=1$ and $N_{\max}=8$, a sufficiently long run sees the cycle 1, 2, 4, 8, 1, 2, 4, 8, Round is used for correctness stress-testing: it forces the integration through many reconfigurations per run, exposing rare corner cases that would otherwise need many runs to surface.

CE_POLICY acts as a proportional controller on the communication efficiency that TALP accumulates over the most recent inhibitor window. It is governed by three parameters: a target CE value, a sensitivity s (the controller gain) that scales how strongly the policy reacts to deviations from the target, and an inhibitor that sets the minimum number of steps between two consecutive reconfigurations, which is also the window over which CE is accumulated. At each invocation the policy computes a change in the node count, ΔN , from the distance between the measured efficiency CE and its target CE_{target} :

$$\Delta N = \text{round}[(CE - CE_{\text{target}}) \cdot s], \quad (1)$$

where s is the sensitivity, the controller gain. The resulting ΔN is added to the current node count and clamped to the configured bounds $N_{\min}$ and $N_{\max}$. The sign of ΔN expands the allocation when CE is above the target (additional processes can absorb the slack) and shrinks it when CE is below (fewer processes retain more useful computation per process). Its magnitude grows with the distance from the target, so the controller takes large steps when far off and small steps when close. Because ΔN is rounded to an integer, the policy makes no change while the CE error stays within $\pm 0.5/s$ of the target, an emergent dead band that needs no separate threshold parameter. Accumulating CE over the whole inhibitor window, rather than sampling it at a single step, smooths the signal the controller acts on and keeps reconfigurations from being driven by transient fluc-

tuations. The accumulator is cleared every time the window elapses and the policy is evaluated, not only when an evaluation leads to a reconfiguration.

The experiments in this study use both policies on the same case: Round drives a correctness stress test that forces nine consecutive reconfigurations within a single run, while `CE_POLICY` drives the convergence and gain-tuning experiments. The concrete parameter values are listed in Appendix A.1.

5 EXPERIMENTAL SETUP

This section describes the elements that every experiment holds fixed: the computing platform (Section 5.1) and the test case (Section 5.2). The DMR parameter set is listed in Appendix A.1. Each experiment reported in Section 6 keeps these fixed and varies a single control choice: the starting node count, the reconfiguration policy, or the controller gain.

5.1 Computing Environment

The experiments reported in this work are executed on MareNostrum 5 [17], the EuroHPC pre-exascale supercomputer hosted at the Barcelona Supercomputing Center. Earlier development and integration of the malleable workflow was carried out on a Docker-based cluster maintained by the DMR group, a containerised simulation of an HPC environment that reproduces the full runtime stack (Slurm, MPI, DMR, OpenFOAM) at a small scale and allows rapid, low-cost iteration without queuing on the MN5 production scheduler. Jobs on both platforms are managed through the Slurm scheduler.

The runs use the General Purpose Partition (GPP) of MareNostrum 5, whose nodes carry 112 physical cores and 256 GB of memory each. So that a reconfiguration obtains its nodes immediately, rather than waiting in the production queue, the DMR campaigns run under *Slurm4DMR*, a second Slurm instance nested inside the job’s allocation that hands a reconfiguration its nodes from a private pool instead of the production queue. The full node and interconnect specification, the *Slurm4DMR* mechanism, and the DMR parameter set are given in Appendix A.1.

5.2 Test Case

The main test case is the *drivAer Fastback* [18], a generic passenger-car body developed at the Technical University of Munich for the systematic study of automotive external aerodynamics. The Fastback variant is a sedan-style profile with a smooth, downward-sloping rear. It is one of the standard public benchmarks used by both academic and industrial CFD groups, and Figure 7 shows the flow around it. A second, moving-mesh case is described in Appendix A.7.

The geometry is meshed with OpenFOAM’s *snappyHexMesh* utility to a final size of 91,858,226 cells, with progressive cell-size refinement towards the body surface and prismatic boundary layers added on the body. The simulation solves the steady-state, incompressible Reynolds-averaged Navier–Stokes (RANS) equations using the SIMPLE pressure–velocity coupling algorithm. The air domain encloses the body, with inlet, outlet and

far-field boundary conditions representative of an automotive wind-tunnel campaign. Each experiment runs for 2,000 solver iterations, long enough for the drag coefficient to reach a steady value in our runs, and the drag and lift coefficients are reported through the *forceCoeffs* function object.

6 RESULTS AND DISCUSSION

This section reports the experimental campaign. Each experiment keeps the setup of Section 5 fixed and varies a single control choice. The first subsection states how the malleable runs are judged. The subsections that follow present and discuss each experiment, and the last collects the limitations of the approach.

6.1 Evaluation Method

The malleable runs are judged against vanilla, non-malleable runs of the same case, with identical settings but a fixed node count for the full 2,000 iterations. Two criteria are used. First, the converged drag coefficient C_d of a malleable run should match that of a vanilla run at any fixed node count, up to the variability that two vanilla runs at different node counts show between themselves. This variability is the case’s intrinsic decomposition-noise floor, because a different node count repartitions the domain and reorders the parallel, non-associative floating-point sums [19, 20]. Second, the end-of-run velocity, pressure and turbulence fields should differ from a matched vanilla run by no more than the same floor. Since every run is reconstructed to the same serial mesh, the fields are directly comparable across decompositions, and we compare them on a representative mid-plane.

6.2 Convergence to an Efficient Allocation

The first experiment asks whether the CE policy reaches an efficient allocation on its own, and whether the allocation it reaches depends on where it starts. Two runs of the same case are launched from opposite extremes: a *grow* run from a single, under-provisioned node, and a *shrink* run from twenty-five, over-provisioned nodes. Figure 3 shows the node count each run holds, the measured CE, and the wall-clock time per iteration.

The grow run begins communication-light: a single node spends almost all of its time computing, so CE is near 0.98, well above the 0.80 target. By Equation 1 the policy takes a large first step ($\Delta N = +3$, to four nodes) and then a smaller one ($+2$, to six), at which point CE has fallen into the stay-put band and the run holds six nodes for the remainder. The shrink run begins communication-bound: twenty-five nodes leave each process with too few cells, CE is near 0.39, and the policy contracts in correspondingly large steps ($25 \rightarrow 19 \rightarrow 16 \rightarrow 13 \rightarrow 10 \rightarrow 9 \rightarrow 8$), settling at eight nodes as CE rises into the band.

Both runs therefore converge, from an eight-fold difference in starting size, into the same narrow efficient region. They do not stop at the same node count: grow settles at six, shrink at eight. This difference is *hysteresis*, the expected signature of a controller with a dead band, where the stopping point depends on the direction of approach. The

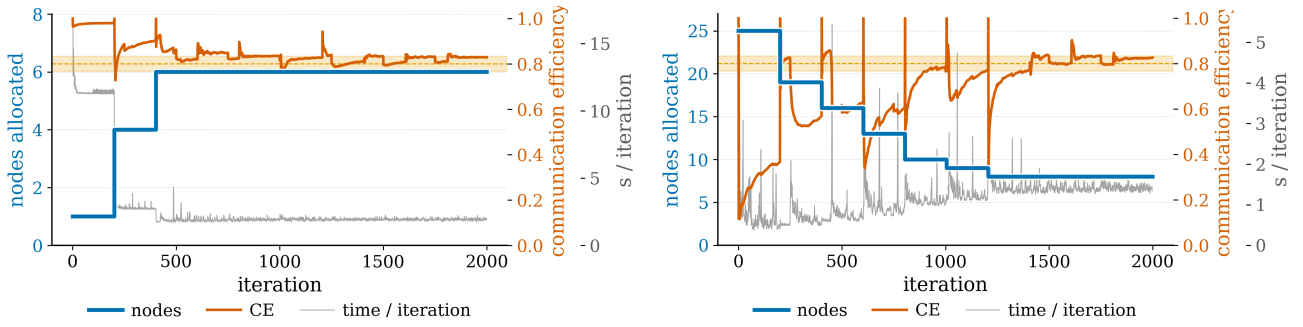


Fig. 3: Convergence of the CE policy from opposite starts. Left: the *grow* run from one node. Right: the *shrink* run from twenty-five. Blue steps are the node count the job holds, red is the accumulated communication efficiency over each inhibitor window with its orange stay-put band, and the faint grey line is the per-iteration compute time. Both runs converge into the efficient region of $[6, 8]$ nodes: the grow run settles at six and the shrink run at eight, an instance of hysteresis.

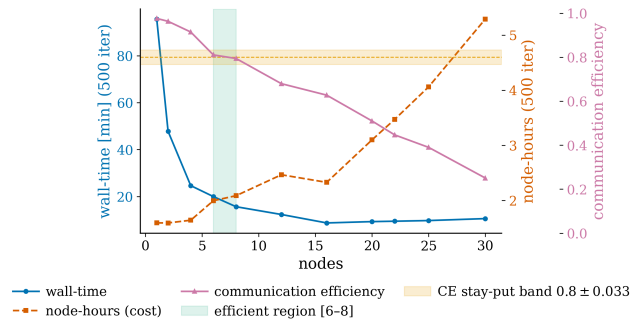


Fig. 4: Resource-efficiency landscape of the case across fixed node counts (500-iteration sweep): wall time (left axis), node-hours (first right axis) and communication efficiency (second right axis). CE crosses the target band between six and eight nodes, the green region where both controlled runs settle. Wall time is minimised near sixteen nodes and cost near two, and the efficient region sits between the two optima.

efficient operating point is thus a region, $[6, 8]$ nodes, rather than a single value.

Reaching that region autonomously has a direct payoff for the under-provisioned start: the grow run completes the 2,000 iterations in 2 h 40 min, against 6 h 15 min for the static single-node run, a $2.34\times$ speed-up obtained without the user knowing the efficient size in advance. This compares the adaptive run against a poor (single-node) static choice. Section 6.5 gives the full cost accounting.

Figure 4 explains *why* the policy settles at $[6, 8]$. A static sweep across fixed node counts shows CE crossing the target band between six and eight nodes (0.81 at six, 0.79 at eight), exactly where both controlled runs land. The two cost metrics pull in opposite directions: wall time keeps falling until about sixteen nodes, while node-hours are cheapest at one or two and rise with size. The region the policy finds sits in the efficient middle of this trade-off.

6.3 Physics Preservation

The second experiment tests whether reconfiguration corrupts the physics. The dedicated correctness run uses the *Round* policy, which forces nine reconfigurations within a single 2,000-iteration run (the deterministic cycle

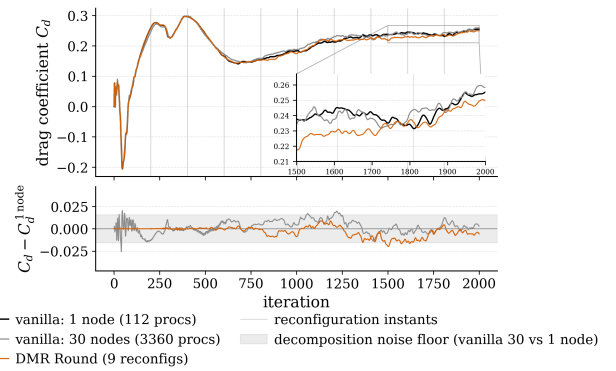


Fig. 5: Physics preservation against the decomposition-noise floor. Top: drag coefficient over the run for the static single-node baseline (black), a static thirty-node run (grey) and the nine-reconfiguration Round run (red), with light verticals marking the reconfiguration instants. Bottom: each run's drag minus the single-node baseline, against the shaded decomposition-noise floor (the spread of two static runs at different node counts). The malleable difference (red) is no larger than the static-to-static floor (grey).

1, 2, 4, 8, ...), a far harsher checkpoint and restart stress than any efficiency-driven run would impose. Its drag coefficient is compared against static, non-malleable runs of the same case at a fixed node count, both at one node and at thirty.

The comparison must be made against the right yardstick, the decomposition-noise floor defined in Section 6.1. Running the static case at one node (112 processes) and at thirty nodes (3,360 processes) measures that floor directly: the two agree to within 1.5% ($C_d = 0.245$ and 0.249). This is the level of disagreement that correct, non-malleable OpenFOAM shows with itself, and therefore the bar a malleable run must meet.

Against that bar, the malleable run passes. The nine-reconfiguration Round run converges to $C_d = 0.242$, a deviation of 1.4% from the single-node baseline, smaller than the 1.5% floor set by the two static runs. Its converged drag sits just below the static band, but the magnitude of its deviation stays within the decomposition-noise floor, and Figure 5 shows the drag histories lying on top of each other across all nine reconfigurations. The grow run agrees even

more closely, to within 0.4%. The end-of-run velocity field tells the same story: on a mid-plane slice through the domain, the malleable and static results are visually indistinguishable, with the differences confined to the turbulent wake and within the same few-percent floor. Reconfiguration therefore adds no error beyond the one the decomposition choice already introduces.

6.4 Controller Tuning

The CE policy is a proportional controller (Equation 1), so its gain s should govern the speed and stability of convergence. Repeating the grow scenario at $s = 5, 15$ and 30 reproduces the textbook behaviour of such a controller (Table 1). The default $s = 15$ is well matched to this case, reaching six nodes in just two reconfigurations. The low gain $s = 5$ is over-damped, crawling up one node at a time over five reconfigurations, while the high gain $s = 30$ is under-damped, overshooting and then jittering between six and seven nodes. Mistuning the gain in either direction therefore costs several times more reconfigurations than the default. The policy is thus not a black box but a tunable controller whose single parameter trades convergence speed against stability in a predictable way. The node trajectories are shown in Appendix A.4.

Table 1: EFFECT OF THE CONTROLLER GAIN ON THE GROW SCENARIO.

Gain s	Reconfigurations	Overhead	Settles at
5	5	134 min	6 nodes
15	2	36 min	6 nodes
30	4	120 min	6–7 (jitter)

6.5 Reconfiguration Cost and Amortisation

Every reconfiguration has a cost, and that cost is the main practical limitation of the approach. It is dominated by the two serial OpenFOAM utilities the integration drives at rank zero: `reconstructPar`, which gathers the distributed state to a single process, and `decomposePar`, which repartitions it for the new size. As Figure 6 shows, both grow roughly linearly with the process count, but `decomposePar` dominates: it rises to about 66 minutes at 2,128 processes, some seven times steeper than `reconstructPar`. Writing the field data in binary rather than as text cuts this file I/O by about six times, but it stays serial and still grows with the process count.

Table 2 splits each run’s wall time into useful compute and serial reconfiguration overhead. The overhead is paid up front and is not amortised within a single short run, so the picture is honest in both directions. From a single under-provisioned node the grow run spends only 23% of its time reconfiguring, because its two reconfigurations touch at most 672 processes and cost 36 minutes together. It finishes in 2 h 40 min against 6 h 15 min for the static single-node run, a $2.34\times$ speed-up obtained without knowing the efficient size in advance. From twenty-five over-provisioned nodes the shrink run is the opposite: its six reconfigurations each reconstruct between 1,000 and 2,800 processes, the first alone taking 86 minutes, so 82% of its 6 h 35 min is reconfiguration and it ends up slower than simply staying

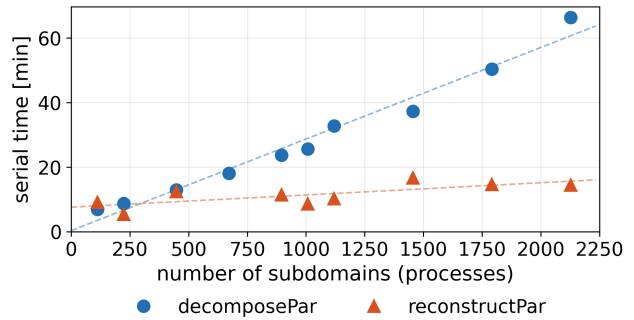


Fig. 6: Serial cost of the two reconfiguration stages versus the number of subdomains each handles. `decomposePar` (blue) creates the new subdomains and dominates, about 28 minutes per thousand processes. `reconstructPar` (orange) gathers the old subdomains to one and stays almost flat, about 4 minutes per thousand.

Table 2: OVERHEAD BREAKDOWN ON THE DRIVAER CASE (2,000 ITERATIONS). FOR EACH RUN THE WALL TIME IS SPLIT INTO USEFUL COMPUTE AND THE SERIAL RECONFIGURATION OVERHEAD, WITH THE NUMBER OF RECONFIGURATIONS N_r AND THE NODE-HOURS HELD. THE OVERHEAD IS THE GATHER-AND-SCATTER AT RANK ZERO, WHICH GROWS WITH THE PROCESS COUNT, SO IT IS MINOR FOR THE GROW RUN AND DOMINANT FOR THE SHRINK RUN. THE ROUND RUN CYCLES THROUGH ALL SIZES, SO A SINGLE NODE-HOURS FIGURE IS NOT MEANINGFUL FOR IT.

Run	N_r	Compute	Reconf.	Total	Node-h
static, 1 node		6h15m		6h15m	6.3
grow, 1 $\rightarrow$ 6	2	2h03m	0h36m	2h40m	11.7
static, 30 nodes		0h24m		0h24m	12.0
shrink, 25 $\rightarrow$ 8	6	1h12m	5h23m	6h35m	91.7
round (stress)	9	3h27m	3h02m	6h30m	

at the large allocation. The two regimes are separated by a break-even run length of about 460 iterations for the grow case (Appendix A.5), well below the 2,000 a production run performs.

The same split governs the node-hours, the resource cost the introduction set out to control. The grow run uses 11.7 node-hours against 6.3 for the static single node, a modest premium for a run more than twice as fast that needs no prior tuning. The shrink run uses 91.7 against 12.0 for the static thirty-node run, because it holds a large allocation through its long serial reconstructs. Malleability is therefore a resource-efficiency mechanism whose value is gated by the reconfiguration cost. Growing into an efficient allocation already pays off, but returning over-provisioned nodes to the scheduler will pay off only once the serial gather-and-scatter is replaced by parallel reconfiguration.

6.6 Limitations

The principal limitation is the cost of reconfiguration, and it is structural. The repartition utilities run only on rank zero, invoked as external subprocesses rather than library calls. This is not a stylistic choice: reusing them in-process corrupts the MPI world for the run that follows, a known OpenFOAM limitation its maintainers have confirmed [21].

The reconstruct-and-decompose step is therefore serial and process-spawning, and it is the dominant overhead of Section 6.5. Memory is a second structural limit. During a reconfiguration the solver ranks stay in memory while rank zero also loads the full field set to reconstruct it, so the node briefly holds both at once. This sets a ceiling on the largest mesh that can be reconfigured from the smallest allocation on the 256 GB MN5 nodes. Two further constraints follow from reusing OpenFOAM’s utilities: only the *scotch* method is supported (Appendix A.2), so a case set up with another decomposer is switched to *scotch* with a notice, and the smallest allocation is at least two processes per node, since OpenFOAM rejects a single-process *parallel* run.

A second limitation is the scope of the evaluation. The main campaign studies one production case in depth, with a second, dynamic-mesh case in Appendix A.7, rather than a broad multi-case survey. The multi-region launcher (`foamMultiRun`) is implemented and exercised on a multi-region engine case (Appendix A.8), but only the single-region launcher is run at production scale here. Adding more cases is not only a matter of compute. OpenFOAM ships its tutorials as small, representative cases, and scaling one to tens of millions of cells is a CFD task in itself: the cell count cannot be raised in isolation, because the initial and boundary conditions and the solver settings must be adjusted to keep the case physically valid at the new resolution. As this is outside our expertise, a broader multi-case study would best be carried out together with the CFD users who own those cases. The reported experiments are also single runs, so the run-to-run variability of the CE measurement and of the reconfiguration overhead (visible as scatter in the in-reconfiguration points of Figure 6) is not quantified, and repeating the convergence experiment to attach error bars is a straightforward extension.

7 CONCLUSIONS

This work has made OpenFOAM malleable. To our knowledge, it is the first time the most widely used open-source CFD solver has been freed from its fixed-process-count assumption without disturbing its numerical core. Because the integration hooks OpenFOAM’s two generic launchers rather than any single solver, every physical solver inherits the capability. The integration with DMR is opt-in and non-intrusive: when DMR is absent the binary is byte-for-byte vanilla OpenFOAM, and the malleability pipeline reuses the solver’s own checkpoint and decomposition machinery. It has been validated at production scale on MareNostrum 5. The efficiency-driven policy reaches an efficient allocation on its own from either extreme: starting from one node and from twenty-five, the workflow converges into the same narrow region that a static efficiency sweep independently identifies as the balance between wall time and resource cost, stopping at different points within it as the controller’s dead-band hysteresis predicts. Reconfiguration preserves the physics: across a nine-reconfiguration stress run the drag coefficient and the flow fields match the static reference to within the case’s intrinsic decomposition-noise floor, and the same convergence and preservation hold on a second, moving-mesh case (Appendix A.7).

We have also characterised the CE policy as a proportional controller whose gain trades convergence speed against

stability in a predictable way, and we have quantified the cost of malleability honestly: it is the serial repartition at rank zero, it grows with the process count, and it makes malleability a resource-efficiency tool rather than a single-run accelerator. Taken together, these results show that malleability is feasible, correct and genuinely useful on a real, production-scale CFD solver, and they pinpoint the serial repartition as the one change that would most extend its reach.

8 FUTURE WORK

This integration is a foundation rather than an endpoint. With a correct, validated malleable build of OpenFOAM in place, and with the practical understanding of its internals gained in the process, two lines of continuation stand out.

The immediate line of continuation is to remove the single-threaded reconfiguration bottleneck. The serial `reconstructPar` and `decomposePar` pair gathers the whole state onto rank zero and scatters it back, which is both the dominant overhead of Section 6.5 and the cause of the memory ceiling. OpenFOAM’s `redistributePar` repartitions a case directly between process counts without this gather-and-scatter, so replacing the pair with it would remove both problems. The substitution is not direct, however. `redistributePar` is a separate utility, like the other two, and it must run on the larger of the old and new node counts: at the checkpoint step when shrinking, and at the restart step when growing. The current DMR hooks do not expose which direction a reconfiguration is taking, so the integration first needs that information to place the call correctly. An early implementation along these lines is under way. A second line is to broaden the reach of the integration. OpenFOAM is released as two parallel distributions: the OpenFOAM Foundation build (`openfoam.org`) used in this work, and the one maintained by ESI/OpenCFD (`openfoam.com`). The integration will be ported to the ESI/OpenCFD distribution and submitted upstream to both. Finally, once the implementation is complete, a joint publication with the Barcelona Supercomputing Center would give the work more credibility and bring it to a wider audience.

USE OF GENERATIVE AI

Generative AI tools were used as a learning and editing aid under the author’s direction. Starting from no prior background in CFD, OpenFOAM or the DMR runtime, the author used them to build an understanding of those topics and the surrounding HPC tooling, to refine the writing of this article, and to assist with code review. The engineering itself, the integration, the experiments and their analysis, is the author’s own work, and every AI-assisted suggestion was reviewed and verified before use.

ACKNOWLEDGEMENTS

The author thanks Francisco Javier Sánchez Pujadas (UAB) for his academic supervision, and Sergio Iserte Agut (BSC) for his guidance on the DMR runtime and the malleable workflow and for generously sharing his deep experience

in high-performance computing. This work used the MareNostrum 5 supercomputer and the DMR library, both provided by the Barcelona Supercomputing Center. The author is grateful to the DMR team, part of the Accelerators and Communications in HPC group at BSC, for their help throughout the implementation and their prompt responses, and to the members of that group who attended the project presentation at BSC for their interest in the work. Thanks also go to the DLB and TALP team at BSC for their guidance on the communication-efficiency profiling and its overhead, to the BSC CFD users for providing the OpenFOAM ESI test cases that motivated the ESI port proposed as future work, and to the OpenFOAM Foundation maintainers for their responsiveness during the integration and their orientation on the dynamic-mesh behaviour.

REFERENCES

- [1] MPI Forum. *MPI: A Message-Passing Interface Standard, Version 4.0*. 2021. <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>
- [2] Yoo, A. B., Jette, M. A., Grondona, M. *SLURM: Simple Linux Utility for Resource Management*. Job Scheduling Strategies for Parallel Processing, LNCS 2862, pp. 44–60, Springer, 2003.
- [3] Weller, H. G., Tabor, G., Jasak, H., Fureby, C. *A Tensorial Approach to Computational Continuum Mechanics Using Object-Oriented Techniques*. Computers in Physics, vol. 12, no. 6, pp. 620–631, 1998.
- [4] Iserte, S., Mayo, R., Quintana-Ortí, E. S., Peña, A. J. *DMR API: Improving Cluster Productivity by Turning Applications into Malleable*. Parallel Computing, vol. 78, pp. 54–66, 2018.
- [5] Garcia-Gasulla, M., Houzeaux, G., Ferrer, R., Artigues, A., López, V., Labarta, J. *Runtime Mechanisms to Survive New HPC Architectures: A Use Case in Human Respiratory Simulations*. International Journal of High Performance Computing Applications, vol. 33, no. 1, pp. 106–120, 2019.
- [6] OpenFOAM Foundation. *How to Contribute*. <https://openfoam.org/dev/how-to-contribute/> (accessed June 2026).
- [7] Feitelson, D. G., Rudolph, L., Schwiegelshohn, U., Sevcik, K. C., Wong, P. *Theory and Practice in Parallel Job Scheduling*. Job Scheduling Strategies for Parallel Processing, LNCS 1291, Springer, 1997.
- [8] Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G., Dongarra, J. J. *An Evaluation of User-Level Failure Mitigation Support in MPI*. Computing, vol. 95, no. 12, pp. 1171–1184, 2013.
- [9] Comprés Ureña, I. A., Mo-Hellenbrand, A., Gerndt, M., Bungartz, H.-J. *Infrastructure and API Extensions for Elastic Execution of MPI Applications*. Proceedings of EuroMPI 2016, pp. 82–97, ACM.
- [10] Kale, L. V., Krishnan, S. *CHARM++: A Portable Concurrent Object Oriented System Based on C++*. Proceedings of OOPSLA 1993, pp. 91–108, ACM.
- [11] Barcelona Supercomputing Center. *DMR v2: Dynamic Management of Resources*. <https://gitlab.bsc.es/accelcom/releases/dmr> (restricted access).
- [12] Performance Optimisation and Productivity (POP) Centre of Excellence. *POP Standard Metrics for Parallel Performance Analysis*. <https://pop-coe.eu/node/69> (accessed June 2026).
- [13] Lopez, V., Ramirez Miranda, G., Garcia-Gasulla, M. *TALP: A Lightweight Tool to Unveil Parallel Efficiency of Large-scale Executions*. Proceedings of the International Workshop on Performance Engineering, Modelling, Analysis, and Visualization Strategy (PERMAVOST '21), ACM, 2021.
- [14] Barcelona Supercomputing Center. *DMR: Scientific Applications and References*. <https://accelcom-bsc.github.io/DMR/references> (accessed June 2026).
- [15] Iserte, S., Houzeaux, G., Sandås, P., Peña, A. J., Garcia-Gasulla, M. *Malleable Computational Fluid Dynamics Simulations*. Proceedings of PARCFD 2025, Barcelona.
- [16] OpenFOAM Foundation. *Issue Tracker, Report 4368*. <https://bugs.openfoam.org/view.php?id=4368> (accessed June 2026).
- [17] Barcelona Supercomputing Center. *MareNostrum 5*. <https://www.bsc.es/marenostrum/marenostrum-5> (accessed June 2026).
- [18] Heft, A. I., Indinger, T., Adams, N. A. *Introduction of a New Realistic Generic Car Model for Aerodynamic Investigations*. SAE Technical Paper 2012-01-0168, 2012.
- [19] IEEE. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019, 2019.
- [20] Collange, C., Defour, D., Graillat, S., Iakymchuk, R. *Numerical Reproducibility for the Parallel Reduction on Multi- and Many-Core Architectures*. Parallel Computing, vol. 49, pp. 83–97, Elsevier, 2015.
- [21] OpenFOAM Foundation. *Issue Tracker, Report 4359*. <https://bugs.openfoam.org/view.php?id=4359> (accessed June 2026).
- [22] INRIA. *scotch: Static Mapping, Graph Partitioning, and Sparse Matrix Ordering*. <https://gitlab.inria.fr/scotch/scotch> (accessed June 2026).

APPENDIX

A.1 Experimental Configuration and Reproducibility

MareNostrum 5 hardware. MareNostrum 5 entered production in 2024. Its General Purpose Partition (GPP), used

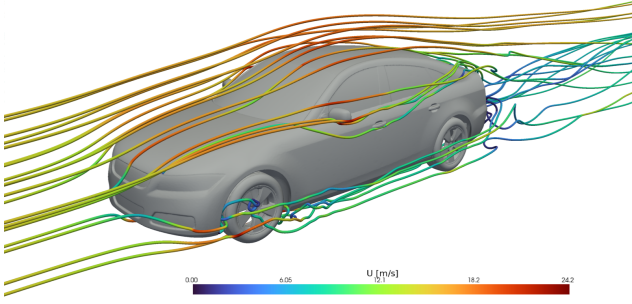


Fig. 7: The drivAer Fastback, the main test case: streamlines of the flow around the car body.

in this work, comprises 6,408 compute nodes based on Intel Sapphire Rapids processors. Each node has two Intel Xeon Platinum 8480+ sockets at 2.0 GHz, providing 112 physical cores, and 256 GB of DDR5 memory. Each node connects to the interconnect through one NDR200 InfiniBand link shared between two nodes via a SharedIO module, giving an effective 100 Gb/s per node. Storage is provided by GPFS (IBM Spectrum Scale), mounted from every node, with 960 GB of local NVMe per node.

Resource acquisition with Slurm4DMR. On MareNostrom 5, expansion requests issued by a running job to the production Slurm scheduler are placed in the regular queue and may wait minutes or hours before the additional nodes become available. To run the DMR campaigns where a reconfiguration obtains its nodes immediately, we run them under Slurm4DMR, a nested Slurm instance that the DMR group provides. Slurm4DMR reserves a pool of nodes from the production scheduler at submission time. Those nodes are then only accessible to jobs inside the nested instance, and DMR allocates and releases them from this private pool during the run. The shrink experiment, which may hold up to 25 nodes, therefore submits a Slurm4DMR pool of 26 nodes: one for the nested controller, plus the maximum the run can hold.

DMR parameters. DMR is configured at runtime through the environment variables listed in Table 3. `DMR_PROCS_PER_NODE` fixes the processes spawned per node, set to 112 on MN5 (one per physical core). `DMR_DEFAULT_POLICY_MIN` and `DMR_DEFAULT_POLICY_MAX` bound the node count the policy can request, and apply to both Round and CE_POLICY. The CE policy uses `DMR_TALP_TARGET_CE` as the target communication efficiency and `DMR_TALP_SENSITIVITY` as the gain that scales the controller’s response. `DMR_DEFAULT_INHIBITOR` sets the minimum number of steps between two consecutive reconfigurations and applies to both policies.

A.2 Mesh Partitioning with Scotch

OpenFOAM offers several domain-decomposition methods. The two simplest, *simple* and *hierarchical*, slice the domain into rectangular blocks along the coordinate axes

Table 3: DMR RUNTIME PARAMETERS FOR THE DRIVAER CAMPAIGN ON MARENOSTRUM 5. `DMR_DEFAULT_POLICY_MAX` IS 10 FOR THE GROW EXPERIMENT, 25 FOR SHRINK, AND 8 FOR ROUND.

Parameter	Value
<code>DMR_PROCS_PER_NODE</code>	112
<code>DMR_DEFAULT_POLICY_MIN</code>	1
<code>DMR_DEFAULT_POLICY_MAX</code>	10–25
<code>DMR_DEFAULT_INHIBITOR</code>	200
<code>DMR_TALP_TARGET_CE</code>	0.80
<code>DMR_TALP_SENSITIVITY</code>	15 (also 5, 30)

and need the user to factor the process count into one integer per axis. The integration uses the third, *scotch* [22], for two reasons. First, it represents the mesh as a graph, one node per cell and one edge per shared face, and partitions it so as to cut as few edges as possible. Each cut edge is a halo exchange every step, so fewer cuts mean less communication and a higher efficiency. Second, it accepts any process count, while the rectangular methods cannot: a prime such as 113 can only be written as $1 \times 1 \times 113$, a one-dimensional slicing with a poor surface-to-volume ratio. A malleable run may resize to any value within its bounds, so *scotch* is the natural choice.

A.3 Reconfiguration Cleanup and Logging

DMR’s own off-cadence checkpoints would otherwise accumulate on disk, so the integration cleans them up. Each leaves a sentinel file, and at the next forced checkpoint every earlier sentinel directory is removed. Dynamic-mesh cases are the one exception: time directories holding a complete topology snapshot (the *mesh-write anchors*) are kept, and the subsequent `decomposePar` receives a `-time` chain of those anchors so OpenFOAM can evolve the mesh topology forward to the restart instant.

Subprocesses are limited to three utilities, all invoked from rank zero of the appropriate group: `reconstructPar` and `decomposePar` for the redistribution itself, and `foamDictionary` for the dictionary edits during the restart. In-process reuse of these utilities is not a supported workflow in OpenFOAM. A `log.dmr/` directory in the case records the lifecycle of each reconfiguration and the output of each subprocess.

A.4 Controller Gain Study

Figure 8 shows the node trajectory of the grow scenario at the three controller gains summarised in Section 6.4 and Table 1.

A.5 Break-Even Run Length

Section 6.5 reports a break-even run length, below which the adaptive grow run does not recover the one-time cost of converging. We derive it here. Let L^* be the run length, in iterations, at which the adaptive run and the static single-node run finish in the same wall-clock time. Let t_1 and t_6 be the measured per-iteration times at one and at six nodes. Let c be the number of iterations the grow run spends converging before it reaches six nodes (here $c \approx 400$, spent

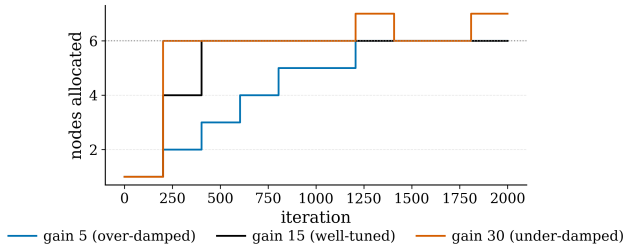


Fig. 8: Node trajectory of the grow scenario at three controller gains. Gain 5 (over-damped) crawls up one node at a time. Gain 15 (well-tuned) reaches the target in two steps. Gain 30 (under-damped) overshoots and oscillates between six and seven nodes.

at one and then four nodes) at a mean per-iteration time $\bar{t}_c$, and let O be its total reconfiguration overhead. The static single-node run takes L^*t_1 . The adaptive run takes $c\bar{t}_c + O$ to converge and then runs its remaining $L^* - c$ iterations at t_6 . Equating the two and solving for L^* gives

$$L^* = \frac{c(\bar{t}_c - t_6) + O}{t_1 - t_6} \approx 460 \text{ iterations.} \quad (2)$$

A run shorter than L^* would finish sooner by staying static, while a longer run profits from the efficient allocation the policy found. For this case the threshold is about 460 iterations, well below the 2,000 a production run performs.

A.6 Run Timeline and Reconfiguration Cost

Table 4 gives the full timeline of the grow and shrink runs of Section 6.5, alternating the compute spent at each node count, with the iterations run there, and the reconfiguration that follows. Each reconfiguration runs `reconstructPar` on the old process group and `decomposePar` on the new one, and a final `reconstructPar` gathers the result to a single process when the run ends. The decompose dominates each reconfiguration, and both steps grow with the process count, which is why the grow run’s reconfigurations cost a few minutes each while the shrink run’s, which reconstruct up to 2,800 processes, cost up to 86 minutes. The compute rows give the solver’s own time, 1 h 42 min for grow and 46 min for shrink. In Table 2 the reconfiguration figure is the sum of the in-run reconfigurations here, and the compute figure also absorbs the final reconstruction.

A.7 A Dynamic-Mesh Case: the Rushton Stirred Tank

The main results use a static mesh. To check the workflow on the harder case of a mesh that moves while the simulation runs, we ran a second case: a Rushton stirred tank, which is a tank of fluid stirred by a rotating impeller. The impeller and the region around it rotate while the tank wall stays still, and the two meshes meet at a sliding, non-conformal interface. This is a tougher test of checkpoint and restart than the static car body, because the connection across the sliding interface changes as the impeller turns, and that changing connection, not only the field values, has to be saved in every checkpoint and rebuilt on every restart. The mesh

Table 4: TIMELINE OF THE GROW AND SHRINK RUNS: THE COMPUTE AT EACH NODE COUNT (WITH ITERATIONS) AND THE COST OF EACH RECONFIGURATION (`RECONSTRUCTPAR` ON THE OLD NODES, `DECOMPOSEPAR` ON THE NEW), PLUS THE FINAL RECONSTRUCTION. ALL TIMES ARE IN MINUTES, EXCEPT THE LAST COLUMN, WHICH IS IN NODE-HOURS. THE ROWS SUM TO THE TOTALS OF TABLE 2.

phase	iters	recon.	decomp.	wall	node-h
<i>grow</i> 1 → 6					
1 node	201			38.9	0.6
reconfigure 1 → 4		1.9	11.8	13.7	0.8
4 nodes	201			9.9	0.7
reconfigure 4 → 6		4.2	18.5	22.7	2.1
6 nodes	1598			52.9	5.3
final reconstruct		21.6		21.6	2.2
<i>shrink</i> 25 → 8					
25 nodes	201			5.7	2.4
reconfigure 25 → 19		19.2	66.8	86.0	29.2
19 nodes	201			2.9	0.9
reconfigure 19 → 16		14.5	50.8	65.3	18.1
16 nodes	201			3.1	0.8
reconfigure 16 → 13		14.7	37.7	52.4	12.1
13 nodes	201			4.4	1.0
reconfigure 13 → 10		16.7	33.2	49.9	9.2
10 nodes	201			5.1	0.9
reconfigure 10 → 9		10.2	27.1	37.3	5.8
9 nodes	201			4.7	0.7
reconfigure 9 → 8		8.7	24.2	32.8	4.5
8 nodes	794			19.9	2.7
final reconstruct		25.7		25.7	3.4

has about 17 million cells, the flow is laminar, and the quantity we track is the power the impeller delivers to the fluid, which plays the same role here that drag plays for the car.

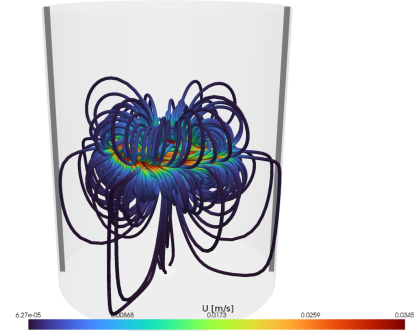


Fig. 9: The Rushton stirred tank: the impeller (red) at the centre and streamlines of the recirculating flow it drives, coloured by speed.

This case is deliberately small, and that makes it a good test of how the target is chosen. A single node already runs it efficiently: with 17 million cells over the 112 cores of one node, each core holds about 152,000 cells, which keeps it busy computing and gives a communication efficiency near 0.85. Moving to two nodes halves the work per core and the efficiency falls to about 0.55. The efficient size is therefore one node, the smallest the workflow allows.

Because of this, the target value matters, as Table 5 and Figure 10 show. With a target of 0.80 the policy is caught between the two sizes: one node sits above the target, so the policy grows to two, but two nodes sit below it, so the policy shrinks back to one, and the run flickers between one

and two nodes for the rest of the run. Each of these needless reconfigurations costs a serial reconstruct, so the flickering run takes 3 h 21 min against 1 h 36 min for a fixed four-node run. With a target of 0.85 the single-node efficiency of about 0.85 falls inside the policy’s stay-put band, so one node is a stable choice. The shrink run, started on fifteen nodes at a CE near zero, drives the efficiency up to the target in two steps, from fifteen to two to one, and then holds it there, returning fourteen of its fifteen nodes to the scheduler in the process. The grow run starts already at the target on one node and maintains it for the whole run. The policy therefore both reaches the target communication efficiency and keeps it, on a moving mesh, from either starting point. The target must sit just above the single-node efficiency because this case is too small for two nodes to ever be efficient: only then is the efficient size a stable fixed point. A target of 0.85 achieves this, a target of 0.80 does not.

Reconfiguring the moving mesh does not damage the physics. The impeller power of the one-node adaptive run matches a fixed four-node run to four significant figures, a difference below 0.1%, even though the two runs split the mesh differently. The flow is laminar and the run is short, so this is a consistency check rather than the tight statistical match reported for drivAer, but it confirms that saving and restoring a sliding, non-conformal mesh keeps the solution intact.

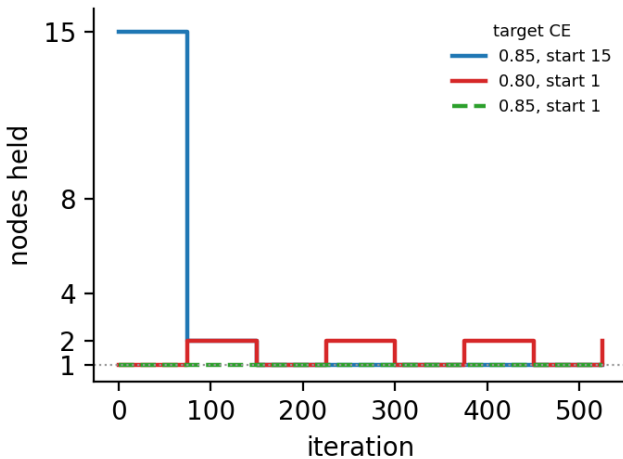


Fig. 10: Node count over the Rushton run under the CE policy. At target 0.80 (red) the policy flickers between one and two nodes, because neither size matches the target. At target 0.85 the single efficient node sits inside the stay-put band, so the run started on one node holds there (green, dashed) and the run started on fifteen nodes contracts to one (blue). The dotted line marks the efficient single node.

A.8 A Multi-Region Case: a Two-Valve Engine

The production campaign uses the single-region launcher `foamRun`. To show that the workflow also drives the multi-region launcher `foamMultiRun`, and that it copes with a mesh that changes shape during the run, we ran a two-dimensional model of an engine cylinder. The case couples six regions: the gas in the cylinder, the cylinder head, the liner, the piston and the two valves, each a separate moving mesh. As the engine turns, the piston travels and the valves

Table 5: THE RUSHTON STIRRED TANK (ABOUT 17 MILLION CELLS) UNDER THE CE POLICY. ONE NODE IS THE EFFICIENT SIZE, SO A TARGET OF 0.85 MAKES IT A STABLE CHOICE, WHILE A TARGET OF 0.80 MAKES THE RUN FLICKER BETWEEN ONE AND TWO NODES. WALL TIME IS FOR THE FULL RUN.

Run	Node trajectory	Reconfigs.	Wall time
vanilla, 4 nodes	fixed	–	1 h 36 m
CE 0.80, start 1	1 $\leftrightarrow$ 2 (flicker)	6+	3 h 21 m
CE 0.85, start 1	1 (held)	0	2 h 09 m
CE 0.85, start 15	15 $\rightarrow$ 2 $\rightarrow$ 1	2	4 h 34 m

open and close, so cells are created and destroyed and the mesh itself, not only the field data, changes from one step to the next. Every checkpoint and restart therefore rebuilds the mesh as well as reloading the data, which is one of the most demanding scenarios in CFD. The simulation marches at a fixed time step of 0.05 under a Courant limit of 2.

We ran the case on the DMR development cluster, not on MareNostrum 5, because we could not scale it to production size. Growing a moving-mesh engine to the tens of millions of cells that would justify a supercomputer is itself an engine-CFD task, and it needs expertise we do not have, so the case stayed small. That is enough for what we want from it: a yes-or-no answer to whether the multi-region moving-mesh reconfiguration works at all, run on the containerised cluster that reproduces the full HPC stack (Slurm, MPI, DMR, OpenFOAM) at small scale and on which the integration is developed and checked. For the same reason we did not size the case for the efficiency-driven CE policy, and drove it instead with the deterministic Round policy, bounded between one and four nodes (two processes per node) and reconfiguring every seven steps, whose job is simply to force as many reconfigurations as possible and check that each one succeeds. The case completed 62 reconfigurations, cycling through one, two and four nodes, with no failures, confirming that the multi-region checkpoint and restart path is sound even when the mesh is changing.

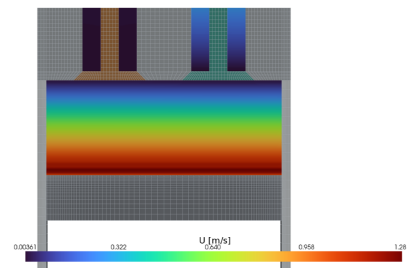


Fig. 11: The two-valve engine, a cross-section through the cylinder: the head, the exhaust valve (bronze) and the intake valve (teal), the piston, and the gas speed in the chamber. The mesh moves and changes shape as the valves and piston move.