Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Elimination of I/O in Optimization Iterations using OpenFOAM Solvers

J. Coddé*[a], E. Tijskens*[b]

[a]*Diabatix (NV), Interleuvenlaan 62, 3001 Leuven, Belgium*
[b]*University of Antwerp, CalcUA, Middelheimlaan 1, 2000 Antwerp, Belgium*

Abstract

This project is concerned with eliminating the I/O bottleneck in the geometry optimisation of cooling components using OpenFOAM. The goal of one run of the algorithm is to create the optimal geometry of a cooling component. Each run consists of a sequence of hundreds of OpenFOAM simulations, interleaved with optimisation calls which update the geometry for the next OpenFOAM simulation. The communication between OpenFOAM and the optimiser is via files and is associated with a lot of I/O. The aim of the project is to develop a system that allows to carry out all communication in memory and eliminate the I/O all together. As the I/O increases super-linear with the amount of nodes and the size of the problem its elimination improves the scaling of the approach.

1. Introduction

The most important type of cases executed at Diabatix is a design run. Such a design run fully determines the most optimal design for a specific cooling problem. A design run is an iterative process, with each iteration being executed comprising a full CFD simulation run. Each CFD simulation run is actually an OpenFOAM solver being called, solving the CFD problem and returning both design performance and a result field. An essential part of running OpenFOAM in parallel are the decomposition and reconstruction operations though, necessary for the application of domain decomposition, but being separate binaries which themselves are not parallel. The decomposition is aimed at distributing all fields and meshes over all available cores (or processors, in OpenFOAM terminology). The reconstruction is for collecting all result data (written separately for each core/processor) to one file.

The workflow of a design run is visualised below in Figure 1, with the left figure the current workflow and the right figure the workflow envisioned by the PRACE project. Apart from the CFD solver run at the middle of a design iteration, an important part of the workflow is handling the design. The “handling” is mostly operations of fields, scaling in size with the number of design variables. The current handling relies on a significant number of I/O operations:

1. Writing the design field to a single file
2. Decomposition of this file across multiple processors (for parallel solver runs)
3. Reconstruction of the result field (which has the same size as the design field), by re-assembling the data spread across all processors
4. Reading the design field

More concretely, one scalar field for a case with 1 M cells in the mesh comprises 8.5 MB of data on disk. (Typical runs are currently between 10 and 20 M cells). It is written as a mixed ASCII-binary file, of which the field itself is written as a list of floats in binary. This is exactly the data that needs to be handled in current workflow in the

* Corresponding author email address: joris.codde@diabatix.com ; engelbert.tijskens@uantwerpen.be

"write design field", "decompose design field", "reconstruct result field" and "read result field" steps. The solver run reads and writes all fields (not only the design field). Both the decompose and reconstruct step read and write this field (though one of the two is stored distributed over different processor folders). Thus, 5 out of 6 steps use I/O operations, of which 4 are almost pure I/O operations. It is known that especially the decompose and reconstruct steps are inefficient, which can e.g. be addressed by using an MPI-I/O library outputting to HDF5 format[b].

The workflow envisioned by the PRACE project aims to perform all operations in memory, thus avoiding all I/O operations in the design loop. This is achieved by compiling the OpenFOAM source code into a dynamic library using pybind11 (https://pybind11-jagerman.readthedocs.io/en/latest/), rather than into an executable as usual. This dynamic library can be used as a Python binary extension module. The crucial feature of this dynamic library is that the lifetime of the memory it manages extends over the entire design run, rather than over a single OpenFOAM solver run. In other words, all fields remain in memory during the entire design run. After updating the design field, the solver run is restarted with the fluid fields still in memory from the previous step. They need no longer be written to disk to communicate them to the next step, and the design field can be manipulated directly in memory. As the computation of the updated design field is currently still serial, it must gather the fields first from, all the processes, and scatter the updated design field back to all the processes. This is likely to become a bottleneck in the future and will receive attention in due time. Ideally, the optimiser would operate in parallel and directly act on the distributed fields, eliminating the gather and scatter operation. This would yield an algorithm in which the scalability would only depend on the scalability of the OpenFOAM code.

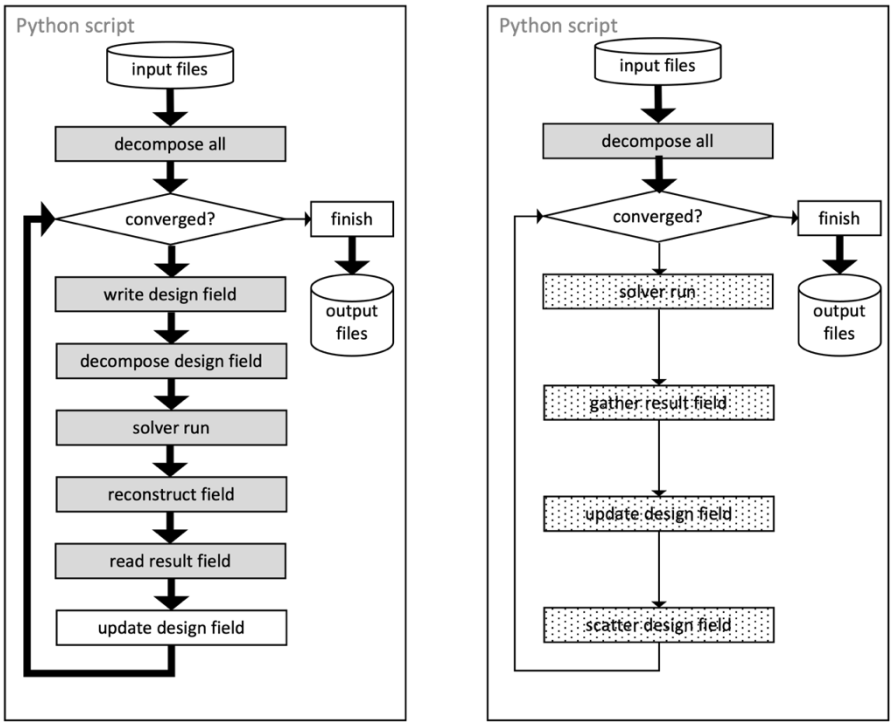

Figure 1: flow chart of a design run. Left figure: current workflow. The process consists of a Python script which calls OpenFOAM executables represented by grey boxes. These communicate their result to the next step through files written to/read from disk, as indicated by the bold arrows. Only 'update design field' is not an OpenFOAM executable, but it nevertheless communicates through files written to/read from disk. Thus, every grey box starts with reading the results from the files written in the previous step and ends with writing the files needed by the next step. Right figure: envisioned workflow. Here, the process is still a Python script, but the dotted boxes represent calls to a dynamic library built on top of the OpenFOAM source code. This dynamic library keeps all fields in memory during the entire process, where they can be accessed directly and modified in place. The 'decompose field' and 'reconstruct field' steps are replaced by gather and scatter which operate in memory, and all I/O during the entire process is completely eliminated. The only bold arrows that remain are at the beginning and the end of the process.

The performance of a design run, i.e. the time needed to reach convergence, is mostly dependent on:

- The size of the case being executed, i.e. the number of cells. This will be specified as a "number of cells per region x", as the CFD simulations executed at Diabatix typically involve multiple materials. Each block of material is termed a region (this is also the terminology used by OpenFOAM).

[b] https://www.konwihr.de/konwihr-projects/effective-openfoam-mpi-i-o-library-for-hdf5-archive-output/

- The number of nodes the case is executed on. Of course, more nodes will be used for larger cases.
- The number of design iterations, mostly dependent on the specifications of the optimisation problem.
- The number of time steps per design iterations, i.e. the number of solver steps within 1 design iteration (thus having a fixed design).

It is important to note that one of the regions serves a special role, that of "design region". The size of this region has an important influence on performance as well, as can be seen from the flow charts above. It is this region that determines the time for all steps within the loop, apart from the solver call.

2. Project work

At the start of the project the communication of information happens through rather expensive I/O operations because each design iteration contains an OpenFOAM solver run. The solver is an executable that writes the outcome of the solver to disk before it exits. The goal of the project is to transform the solver in a way that it does not exit, keeping the outcome in memory and not dumping it to disk. This enables all steps of the design loop to perform their work in memory, effectively removing I/O from the design loop in Figure 1. Since the design loop itself is implemented in Python, whereas OpenFOAM is written in C++, the C++ code of the OpenFOAM solver was exposed to Python. The approach is presented in the rest of the section. It needs to be applied to a single OpenFOAM solver (OpenFOAM contains multiple solvers), and was successfully implemented for the simpleFoam and chtMultiRegionFoam solvers.
The main() function (C++) of a typical OpenFOAM solver comprises three sections: Initialisation of the simulation, the simulation run itself, and the clean-up sections where the results are written to disk. This function is transformed to a C++ Solver class where the initialisation of the simulation is performed in the Solver class constructor, the simulation run is performed in a loop() member function, and the clean-up in the Solver class destructor. In addition, some functionality is added to access all the fields in the simulation. This Solver class is exposed to Python as a Python binary extension module, called pyFoam. The process of converting C++ code into Python binary extension modules is facilitated through Micc[c]. With pyFoam a Python script can setup a simulation, run the solver, access the fields computed by the solver run (in memory), modify the design fields (in memory) and run the solver again. These steps are repeated until the design run completes. The need for I/O communication is completely removed.
Turning OpenFOAM solvers into classes was complicated by the minimal documentation of OpenFOAM. Eventually, only limited changes to the OpenFOAM source code were necessary, which are related to the fact that the variables which are created in the main() function in the OpenFOAM solver are to be constructed in the pyFoam Solver class constructor.

3. Benchmarking

In order to assess the performance gain of the process as envisioned by the PRACE project, the methodology from previous section is applied to two standard OpenFOAM solvers: the simpleFoam and chtMultiRegionFoam solvers. Whereas the simpleFoam solvers is one of the simplest solvers in OpenFOAM, the chtMultiRegionFoam solvers is closer to Diabatix' own solver. The additional complexity for the application of the methodology to Diabatix' solver are discussed in the next section.
The benchmarking results are presented in the figures below. Both figures show the gain of using the process developed within the PRACE project, relative to the original process sequence. The gain is dependent on the size of the case being executed, the number of nodes the case is executed on, and the number of time steps needed to perform one design iteration. As the number of time steps is normally dependent on CFD simulation convergence, this number was fixed for the purpose of the tests, with the goal of a more consistent comparison. As such, it was an input to the OpenFOAM solver (i.e. it was forced to do a number of time steps).

[c] https://micc.readthedocs.io/en/latest/

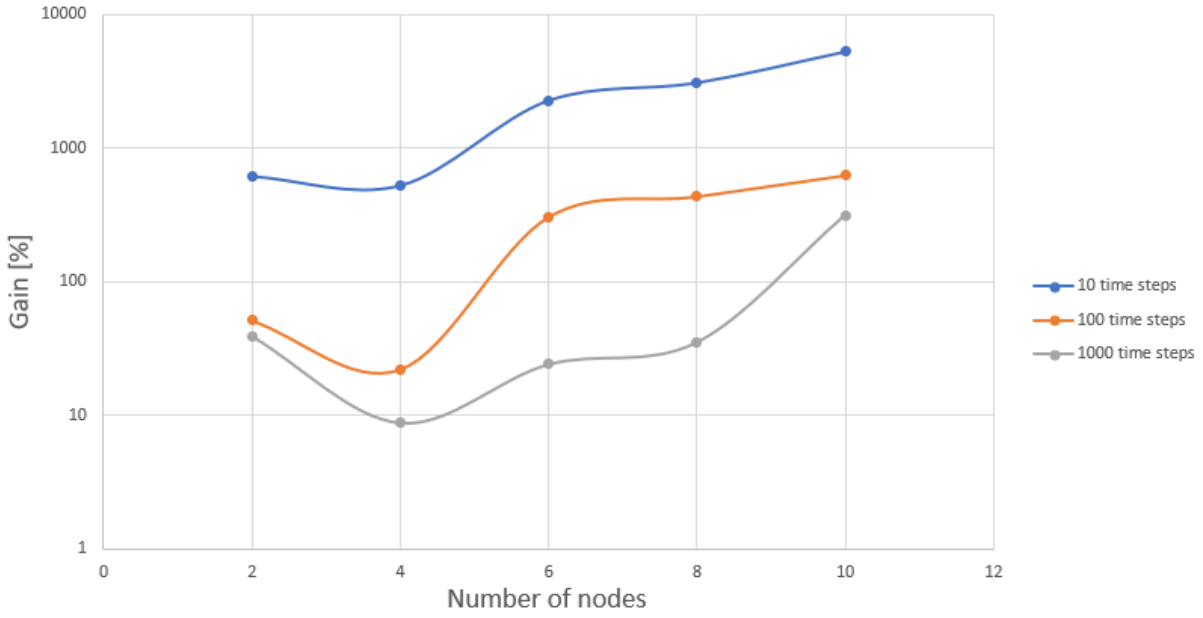

Figure 2: the performance gain in process time for the process sequence as per the PRACE project relative to the original process sequence for test case 1, executed with the simpleFoam solver. The test case is pitzDaily with 12 M cells executed on VSC Breniac's Broadwell nodes. Note the logarithmic scale.

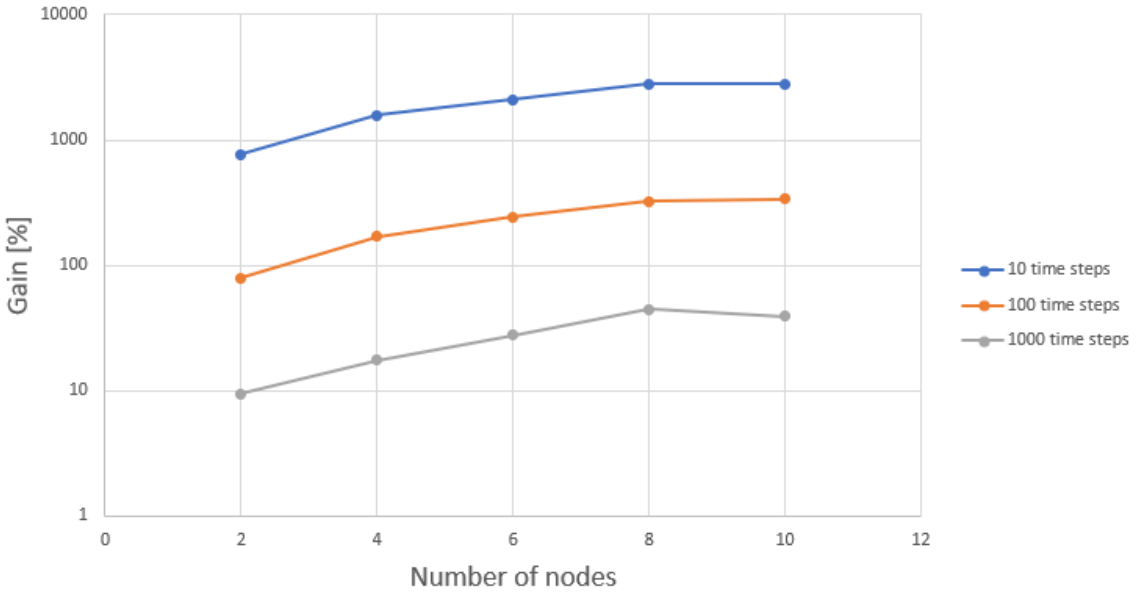

Figure 3: the performance gain in process time for the process sequence as per the PRACE project relative to the original process sequence for test case 2, executed with the chtMultiRegionFoam solver. The test case has 4 M cells in the fluid and 2 M cells in the solid, executed on VSC BrENIAC's Broadwell nodes. Note the logarithmic scale.

In general, the new sequence shows a significant gain for all situations. The gain is higher for lower number of time steps and for higher number of nodes, as expected. The results for the first benchmark are less consistent then those of the second benchmark. It is unclear why this is the case but can partially be explained due to the larger case size and the sensitivity of the InfiniBand network to concurrent jobs and the timing of the tests. The results can be further explained by the following observations, taken away from Table 1and Table 2.

- For the original sequence, the times related to I/O operations only (all except the solver) go up when using more nodes. For the PRACE sequence, these times are small w.r.t. the solver time (even with 10 time steps) and almost independent of the number of nodes.
- The solver-related times show a slight absolute difference in favour for the PRACE sequence. Relatively, this has the largest effect when using a small number of times steps. This is the advantage of avoiding the

part of the solver where files have to be re-read and loaded in memory. However, the difference in solver time is much smaller than the difference on the decomposition and reconstruction times

As a result of the times of operations in memory in the PRACE process being minimal vs I/O operations in the original process sequence, the relative time spent on I/O can be estimated as follows:

$$t_{I/O,relvative} \approx (t_{total,orig} - t_{total,prace})/t_{total,orig}$$

These numbers are presented as part of Table 2. The relative time increases with lower number of time steps and higher number of nodes.

The times related to I/O operations only (decomposition, reconstruction and reading/writing) are almost identical for a specific number of nodes when using a different number of time steps for design iteration. This is as expected. Hence, they were not presented here.

Process sequence	Nodes [#]	2	4	6	8	10
Original	Decomposition time [s]	163.6	184.5	208	230	246.22
	Solver time [s]	100	67	60	57	51.92
	Reconstruction time [s]	166	190	206.2	240	269.75
	Read/write time [s]	1.36	1.45	1.36	1.49	1.39
	Total [s]	430.96	442.95	475.56	528.49	569.28
PRACE	Decomposition time [s]	0.33	0.349	0.36	0.37	0.37
	Solver time [s]	49.5	26.06	21	17.77	19.18
	Reconstruction time [s]	0.11	0.16	0.15	0.16	0.149
	Read/write time [s]	0.000417	0.000395	0.00042	0.000376	0.000374
	Total [s]	49.94	26.57	21.51	18.30	19.70
Original	Percentage of time spent on I/O operations [%]	88	94	95	96	96

Table 1: raw data for both process sequences for 10 time steps per design iteration. The number of nodes are the inputs, all times are a result of executing the process sequence. Note that for clarity of the comparison, the same terms for both process sequences were used, while the terms used in Figure 1 are more exact.

Nodes [#]	Time step [#]	Solver time original [s]	Solver time PRACE [s]	Difference in solver time [s]	Original process: time spent on I/O operations [%]
2	10	100	50	51	88
2	100	491	449	42	44
2	1000	4447	4349	98	9
4	10	67	26.06	41	94
4	100	263	232	31	63
4	1000	2222	2196	26	15
6	10	60	21	39	95
6	100	212	183	29	71
6	1000	1692	1652	40	22
8	10	57	18	39	97
8	100	185	153	31	76
8	1000	1480	1362	118	31
10	10	52	19	33	97
10	100	196	161	35	77
10	1000	1449	1408	41	28

Table 2: raw data of test case 2, presented in Figure 3. Columns 3 to 5 contain the solver time for both the original process sequence as well as the PRACE process sequence, and the difference between them. The last column is the time spent on I/O operations for the original process sequence. The number of nodes and the number of time steps are the inputs; all times are a result of executing the process sequences.

Note that the situation of using 6-10 nodes for the 2 cases presented here is actually more realistic. On top of this, there is the trend of going to higher cell counts (like test case 1, see Figure 2), and thus higher node counts. While the first test case seems to indicate that there is no ceiling on the performance improvement that can be had by using the I/O-free algorithm when increasing the number of nodes, the second benchmark shows a stagnation in performance gain.
The range of the time steps chosen for the test is on the low side, with 1000 time steps per design iteration actually being more of a minimum and leading to gains of 10 – 60%. Nevertheless, lower time steps were included for:

- Showing the consistency of the decomposition, reconstruction and reading times over all cases.
- Near the end of design run, the number of iterations will typically decrease.

Also, the results are consistent enough to make extrapolations for higher, more realistic time steps – especially those of the chtMultiRegionFoam case. When applying these results to a realistic case being processed on Skylake nodes with 5k time steps per design iteration, the gain is around 5%. This is shown in Table 3. The effect of using Skylake nodes instead of Broadwell nodes is around 40%, as this is the performance difference measured for this type of problems, while both nodes have the same number of cores[d].

Number of Nodes	2	4	6	8	10
Gain	2.7%	3.4%	5.6%	6.1%	4.4%

Table 3: Gain for test case 2 (4 M cells in the fluid, 2 M cells in the solid) on VSC BrENIAC's Skylake nodes, with 5k time steps per design iteration.

[d] The full hardware details are available on: https://vlaams-supercomputing-centrum-vscdocumentation.readthedocs-hosted.com/en/latest/leuven/tier1_hardware/breniac_hardware.html

4. Assessment

This section assesses the gains by the PRACE process sequence in terms of scalability vs. cost of implementation, as the benchmarking was performed on example problems.

4.1. Decision criteria for implementation

There are two future trends at play which influence the decision to implement the PRACE process sequence into Diabatix' software:

- The number of cells will increase
- In turn, the number of nodes will increase as well, in order to limit solver time

To assess the effect of both on the two most important steps of the original process sequence, decomposition and reconstruction, the decomposition time was studied in detail. As explained in the first section, decomposition of all meshes and fields is necessary before executing an OpenFOAM solver in parallel. The decomposition operation is handled by a separate OpenFOAM binary, which runs sequentially. As such, the execution time is expected to increase with both the number of cells in the mesh and the number of nodes. Both increase the amount of I/O. The reconstruction time is similar to decomposition time, so we expect similar trends. This result is shown in Figure 4 below. As expected, the decomposition time goes up with both the number nodes and the number of cells. However, for 10 M cells there is a significant increase.

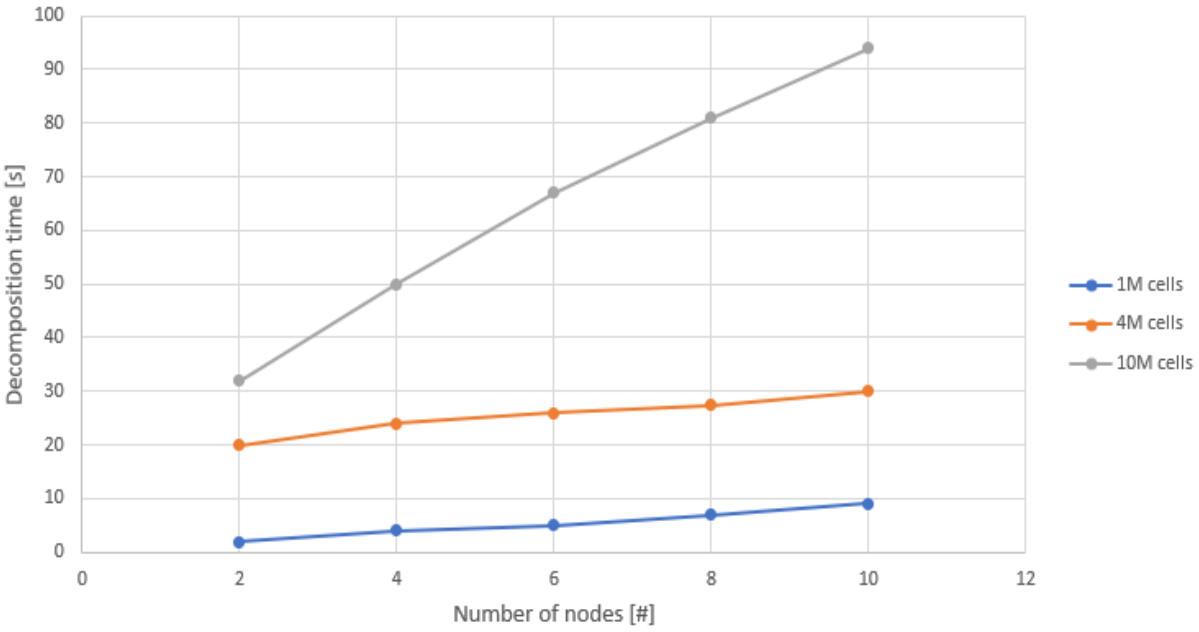

Figure 4: decomposition time of a scalar field in function of the number of cells and number of nodes.

Taken together with the benchmarking results, it makes the case that the PRACE process sequence is more scalable. It shifts the bottleneck for the speed of 1 design iteration almost fully to the OpenFOAM solver.
It has to be stressed that the gain is obviously dependent on the current state of Diabatix' software, which has been improved independently during the PRACE project.

4.2. Cost of implementation

The potential gains have to be weighed against the costs for effectively implementing the findings of the PRACE project. The cost here is a significant rise in complexity entailing a significant amount of changes to Diabatix' software. This could not be included in the release of the next version of Diabatix' software.

5. Conclusion

The process sequence tested in the PRACE project has a promising performance gain of 5% and upwards, depending on the exact case. Most importantly, the I/O bottleneck was completely eliminated, which is impactful at higher node and cell counts. This means scalability of the optimisation algorithm is solely dependent on OpenFOAM solver performance. This would allow Diabatix to go towards larger cases in the future, especially in situations where CFD solver time per design iteration is low. However, due to implementation complexity, it is deemed more interesting to investigate some alternative solutions first. The solution from the PRACE project can be elaborated more in parallel.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 823767. The PRACE machine used for this project was VSC's Tier-1 system BrENIAC.