ENHANCING COMPUTATIONAL AERO-ACOUSTIC PROCESSES FOR GROUND VEHICLES RESOLVING OPEN SOURCE CFD

TORBJÖRN LARSSON[1], JOHAN HAMMAR[1], JING GONG[2], MICHAELA BARTH[2], LILIT AXNER[2]
[1]*Creo Dynamics AB, Sweden, [torbjorn.larsson, johan.hammar]@creodynamics.com*
[2]*PDC-HPC, KTH Royal Institute of Technology, Sweden, [gongjing, caela, lilit]@kth.se*

Keywords: *Scalability, High Performance Computing, CAA*

Introduction

The aerodynamic drag on a heavy-duty truck-trailer equipage is significant and, with new stringent legislations on fuel efficiency and CO_2 reductions in the pipeline, substantial development to improve the aerodynamic efficiency of such ground vehicles is required. Furthermore, the trend towards electrification and hybrid technologies for vehicles means that now, more than ever, there is a need for the aerodynamic drag to be decreased in order to increase the range of electric vehicles (EVs). With EVs expected to dominate the road car market in the near future, there is also renewed interest in topics relating to airborne noise.

Experimental aerodynamic testing of full-size trucks is challenging, particularly as there are only very few facilities in the world where it is possible to do such tests. Highly accurate predictions of the aerodynamic (and acoustic) properties of these large vehicles via physical testing are immensely expensive, and consequently often not justifiable in a repetitive manner, especially in the fast-paced development that is paving a route to sustainable transportation. Instead, further investments in dedicated virtual testing techniques are required with a firm commitment to large-scale high-performance computing (HPC). In this paper, we are aiming for high fidelity scale-resolving simulations to enable us to make accurate predictions, not only of the airflow distribution, but also of the aeroacoustic noise generation and propagation. For these reasons all the steps in the simulation processes need to be made efficient for parallel computation with scalability to many thousands of computational cores.

Herein, we set up and analyse an automated process that, using prepared CAD surfaces as input, generates a high resolution CFD mesh fulfilling predefined quality metrics. The process also performs a steady state RANS (Reynolds-averaged Navier-Stokes) simulation, and extracts results and post-processing data. All these steps are implemented in parallel on distributed compute nodes without the need for any intermediate input/output (I/O) or data transfers.

Open Source CFD Workflow

Parallel performance of all steps of the simulation process, as well as efficient I/O and data extraction, are becoming vital as the size of models and the complexity of simulations increases. Building a fully parallelized simulation process that removes serial bottlenecks, time consuming I/O operations, and labor intensive manual work is a quest that is being actively pursued in the automotive industry due to the significant benefits such a process will bring.

Here, in this project, the basic idea is to derive a (semi-) automated and fully scripted simulation process for a chosen application (truck-trailer) starting from prepared CAD data. The automation, along with the possibility to perform the mesh generation in parallel on distributed compute nodes where the same nodes are also used for subsequent CFD simulations, are central features of the work. These features mean that all the steps in the process can be executed in a homogeneous Linux-based computer environment thus avoiding serial bottlenecks, time-consuming I/O, and additional data transfers. The main steps of this process, illustrated in Figure 1, are based entirely on the *OpenFoam* toolbox [1].

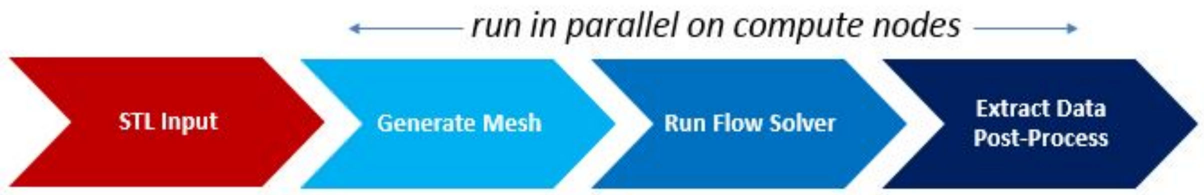

Figure 1: OpenFoam Workflow

The OpenFoam Utilities and Solvers

The pre-processing phase (that is, the Generate Mesh phase in Figure 1 is based on the *OpenFoam* meshing utility *snappyHexMesh,* which relies on a geometry description in STL file format. Before executing *snappyHexMesh* (in parallel) though, a couple of serial steps are performed to create and decompose an initial background hexahedral mesh and to extract geometrical feature edges. These operations only require a few seconds of computational time running on a single core, and consequently will not be described here. In this study, the three steps in the mesh generation process have been monitored in terms of memory usage and parallel run-time performance.

Creating high quality meshes on complex configurations with *snappyHexMesh* is far from trivial and finding suitable "meshing recipes" often requires a sound understanding of the various steps in the mesh generation algorithms, as well as rigorous knowledge of the model configuration and the predominant flow physics involved. Often a good deal of iterative testing and parameter tuning may be needed before a mesh of acceptable quality can be generated. In the *meshQualityControls* section of the *snappyHexMeshDict,* we define relatively strict criteria for the final mesh quality, with face skewness and face orthogonality being two very important quality metrics (see Figure 4 for the definition of these metrics).

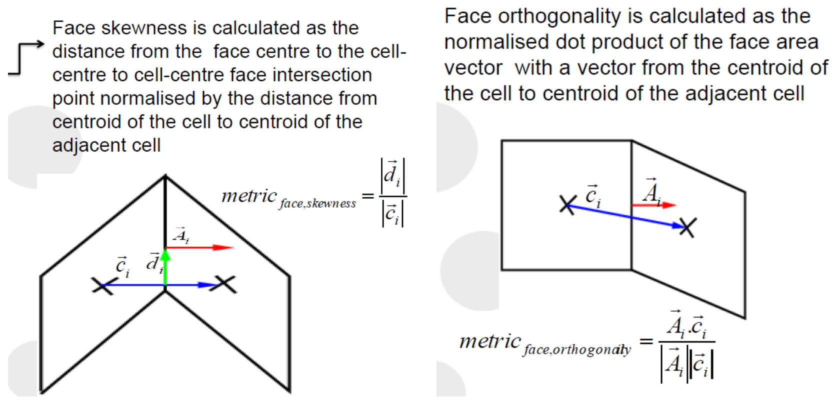
Face skewness is calculated as the distance from the face centre to the cell-centre to cell-centre face intersection point normalised by the distance from centroid of the cell to centroid of the adjacent cell
$metric_{face,skewness} = \frac{|\vec{d}_i|}{|\vec{c}_i|}$
$\vec{A}_i$
$\vec{c}_i$
$\vec{d}_i$
Face orthogonality is calculated as the normalised dot product of the face area vector with a vector from the centroid of the cell to centroid of the adjacent cell
$\vec{c}_i$
$\vec{A}_i$
$metric_{face,orthogonality} = \frac{\vec{A}_i.\vec{c}_i}{|\vec{A}_i||\vec{c}_i|}$

Figure 2: Skewness and orthogonality quality metrics

Since *snappyHexMesh* can run in parallel on distributed cluster nodes by using dynamic load balancing, it is well suited to become an integrated part of a fully scripted CFD process running in batch on a large parallel cluster.

Since the solutions for the coupled equations are not straightforward, the viscous and pressure sub-steps require solving a Poisson-equation subject to various boundary conditions. Within the project, the *simpleFoam* solver [2] is employed to address the large simulations. The Semi-implicit methods for Pressure-Linked Equations (SIMPLE) algorithm is implemented in the solver *simpleFoam* and couples the Navier-Stokes equations in an iterative procedure following [3].

Results and Discussions

The simulations were performed on the Cray XC40 system Beskow [4] at PDC, KTH Royal Institute of Technology, Sweden, and on MareNostrum [5] at BSC, Spain. Beskow is based on Intel Haswell processors and the Cray Aries interconnect technology. During the simulations it consists of 1676 compute nodes, each of which consists of 32 Intel Xeon E5-2698v3 cores. MareNostrum has 36 racks dedicated to calculations. These racks have a total of 48,448 Intel SandyBridge cores with a frequency of 2.6 GHz and 94.625 TB of total memory.

Before proceeding with the large-scale distributed parallel meshing, the parallel scalability of *snappyHexMesh* was first evaluated on a couple of thin compute nodes at PDC where each node has 24 cores and 512 GB RAM. Different releases of *snappyHexMesh* (2.3.x, 2.4, 3.0+, 1606+) were tested. Although there were some performance differences observed between the various releases, unfortunately all the releases were suffering from poor parallel scalability. For these tests, an intermediate sized mesh of 160 million cells was constructed to make the mesh generation possible on a single core. Figure 3 below shows the run-time performance and scalability obtained for this intermediate test case.

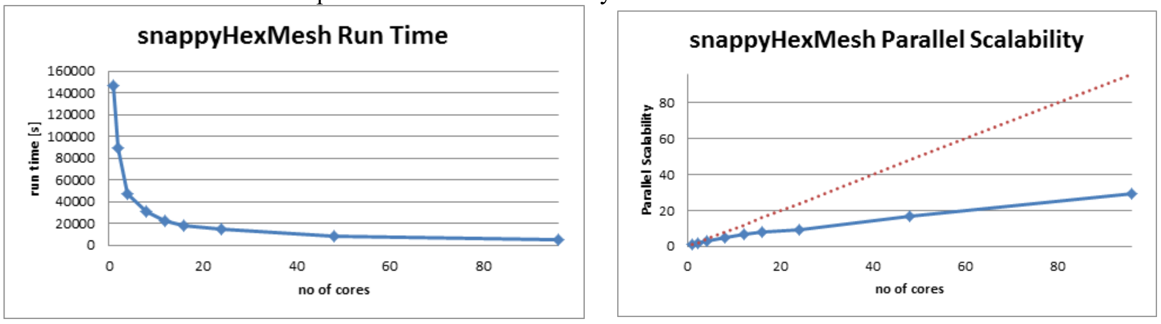

Figure 3: Run-time and scalability test of snappyHexMesh, intermediate case using OpenFoam v3.0+

Next, larger meshes of around 350 million cells were generated in parallel on a sequence of 512, 1024, 2048, 4096 and 8192 cores, respectively. The run-time performance and scalability of the entire meshing process, as well as the timings for each meshing step. *snappyHexMesh* also suffers from relatively high memory consumption. This may lead to performance implications if the computational nodes have limited RAM. Often separate (fat) nodes or another shared memory compute architecture must be brought in for the mesh generation for large and complex cases. This not only creates additional bottlenecks and performance deficits in the overall simulation process but it also increases the cost and complexity of the necessary compute infrastructure. Using distributed STL input can reduce the memory consumption of *snappyHexMesh* and may provide opportunities to generate larger meshes by using less overall memory. In the more recent versions of *snappyHexMesh* this functionality has been improved. Herein we present results obtained while using version 1606+. Using a distributed STL input we save close to 300 GB in total accumulated memory while meshing in parallel on 1024 cores. This represents about a 30% reduction in the overall memory consumption. However, this saving in RAM comes with a significant penalty in mesh generation time. Performance degradation in all steps in the mesh generation process is observed.

Assessments of the total run-times for the *simpleFoam* solver using the two mesh generation strategies, i.e. based on the standard and distributed STL input, are presented below. Figure 4 shows the flow solver scalability for a case containing 110 million cells on MareNostrum and Beskow, respectively. For each run we average the timings over 500 time steps.

An almost linear speed-up can be observed from 64 to1024 cores on MareNostrum for the standard STL input. A scalability of 74.4% is achieved using 2048 cores compared with 64 cores for the standard STL input. Due to memory limitations, the minimum number of cores that can be used for this case on MareNostrum is 64.
Here we use the definition of the parallel efficiency as

$$\eta(\%) = \frac{T_{min}}{T_{max}} \cdot \frac{N_{min}}{N_{max}} \cdot 100$$

where is T_{min} the execution time per step for the minimum number of cores N_{min}, while T_{max} is the execution time per step using the maximum number of cores N_{max}. The performance and scalability for the Distributed STL input is a bit worse than those for the Standard STL, see Figure 4. A partial reason for this deficit is that the more efficient partition tool *Scotch* [6] is employed for the Standard STL. Currently only simple partition methods, such as *hierarchical,* are implemented for the Distributed STL input. On Beskow we can obtain almost linear speed-up from 128 to 2048 cores for the Standard STL, see Figure 4. The efficiency is 82.9% with 4096 cores compared with 128 cores. Using the Distributed STL, the performance is worse, but this allows us to run on as little as just two nodes. Beyond 3072 cores the performance degrades significantly. Due to the limit of memory on Beskow (32G RAM per node), the minimum requirement for the Standard STL is four nodes (128 cores).

We have presented a complete CFD simulation process that runs in parallel on large Linux clusters. The entire workflow, from prepared CAD input to the output of results, is executed in batches without the need for any intermediate I/0, data transfer or human interventions in between. All the steps in this process rely exclusively on open source software. Although the scalability of *snappyHexMesh* is rather mediocre, we demonstrated that fairly large and complex CFD meshes could be successfully generated on distributed compute clusters in a limited period of time. Mesh generation of a detailed semi-trailer configuration comprising 350 million cells was completed in less than 25 minutes using 4096 compute cores. On the same number of cores, the *simpleFoam* solver was shown to perform well. Hence, for this size of problem, the presented process performs well.

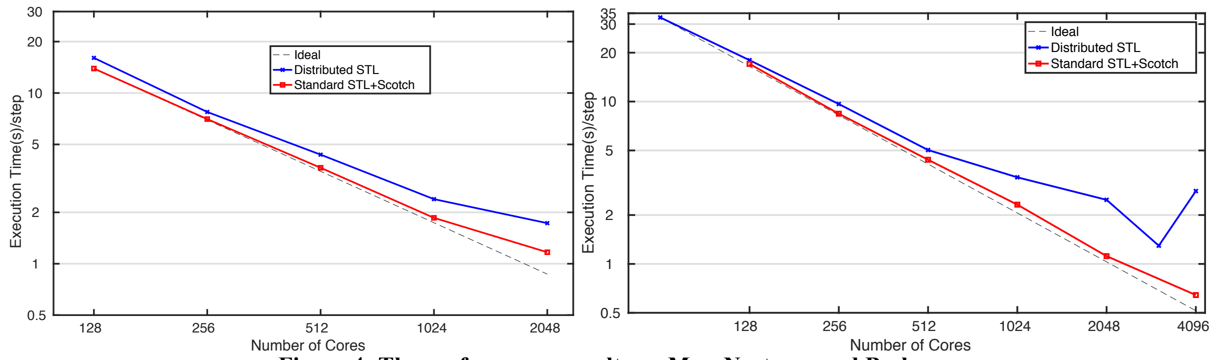

Figure 4: The performance results on MareNostrum and Beskow

However, while scaling beyond a few thousand computational cores and with model sizes above a few hundred million cells, a variety of problems and performance deficits arise. In particular, the mesh generation phase (*snappyHexMesh*) revealed weaknesses and several software issues have been reported.
Below are a few snapshots from a steady state Reynolds Averaged Navier Stokes (RANS) simulation.

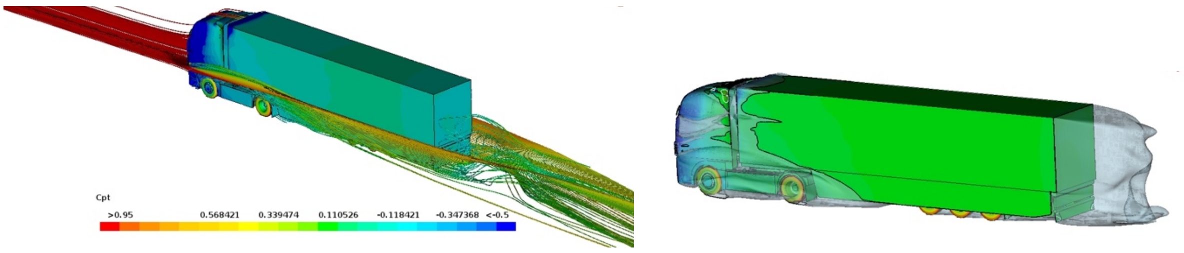

Figure 5: (left) Flow path lines rendered by total pressure (right) Iso-contours rendered by total pressure

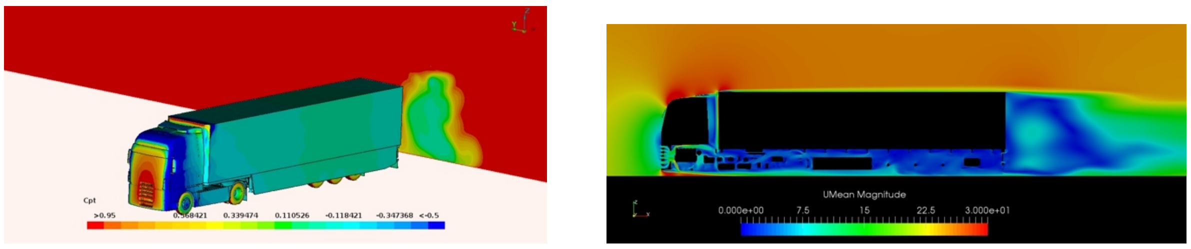

Figure 6: (left) Vehicle surfaces rendered by static pressure and plane rendered by total pressure (right) Vehicle centreline plane rendered by mean velocity magnitude

Conclusion

During the project a (semi) automated CFD simulation process tailored towards aerodynamics predictions for heavy duty semi-trailers was developed. A variety of benchmarks related to parallel meshing and flow solving using *OpenFoam* were performed. The memory usage and parallel scalability were monitored revealing performance deficits and weaknesses, particularly related to parallel mesh generation on many cores. For efficient simulations of large problems requiring many thousands of computational cores there are still critical bottlenecks and deficiencies in current *OpenFoam* distributions that need to be addressed.

References

[1] OpenFoam, http://www.openfoam.com/

[2] simpleFoam, https://openfoamwiki.net/index.php/OpenFOAM_guide/The_SIMPLE_algorithm_in_OpenFOAM

[3] J. H. Ferziger, M. Peric, Computational Methods for Fluid Dynamics, Springer, 3rd Edition, 2001

[3] Beskow, https://www.pdc.kth.se/hpc-services/computing-systems/beskow-1.737436

[4] MareNostrum, https://www.bsc.es/innovation-and-services/supercomputers-and-facilities/marenostrum

[5] Scotch, http://www.labri.fr/perso/pelegrin/scotch/

Acknowledgements

This work was financially supported by Creo Dynamics AB, the Swedish e-Science Center (SeRC), and the PRACE project funded in part by the EU's Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838.