Performance Analysis of Complex Engineering Frameworks

Michael Wagner, Jens Jägersküpper, Daniel Molka, Thomas Gerhold

Abstract Many engineering applications require complex frameworks to simulate the intricate and extensive sub-problems involved. However, performance analysis tools can struggle when the complexity of the application frameworks increases. In this paper, we share our efforts and experiences in analyzing the performance of CODA, a CFD solver for aircraft aerodynamics developed by DLR, ONERA, and Airbus, which is part of a larger framework for multi-disciplinary analysis in aircraft design. CODA is one of the key next-generation engineering applications represented in the European Centre of Excellence for Engineering Applications (EXCELLERAT). The solver features innovative algorithms and advanced software technology concepts dedicated to HPC. It is implemented in Python and C++ and uses multi-level parallelization via MPI or GASPI and OpenMP. We present, from an engineering perspective, the state of the art in performance analysis tools, discuss the demands and challenges, and present first results of the performance analysis of a CODA performance test case.

1 Introduction

Aviation is an essential part of our society and economy. In 2018 the total number of passengers rose to 4.3 billion and is estimated to grow at a rate of 4.3 % per year, i.e., doubling about every 15 years [1, 2]. Being the only rapid worldwide transportation network, air transport represents 35 % of all international trade and about 40 % of all international tourism. The global economic impact of aviation (direct, indirect, induced and catalytic, e.g. on tourism) is estimated at 7.5 % of the world's Gross

Michael Wagner, Daniel Molka and Thomas Gerhold
German Aerospace Center (DLR), Institute of Software Methods for Product Virtualization
e-mail: m.wagner@dlr.de

Jens Jägersküpper
German Aerospace Center (DLR), Institute of Aerodynamics and Flow Technology

Domestic Product (GDP) [3]. On the other side, air transport yields undeniable adverse effects on society and environment, most notably, noise pollution and the emission of greenhouse gases. Due to its increasing growth, aviation could play an increasingly important role in total CO_2 emisions in the future [4].

The European Commission defines in its vision for Europe's aviation several goals to, among others, mitigate the adverse impact of aviation on society and environment. These goals include a reduction of 75 % of CO_2 emissions, 90 % of NO_x emissions, and 65 % of perceived aircraft noise by 2050 (in comparison to a typical new aircraft in 2000) [5]. For the aerospace industry these goals impose heavy demands on future product performance, which require step changes in aircraft technology and mandate new design principles. Thus, future aircraft design may be driven by unconventional layouts such as the low noise aircraft model (LNA), a blended wing body aircraft, or the flying wing configuration. For these unconventional layouts flight characteristics will be dominated by non-linear effects.

In this case, high-fidelity numerical simulation of flight characteristics becomes inevitable for the design and assessment of future step-changing aircraft designs. Numerical simulation provides reliable insight into new aircraft technologies and allows aiming for best overall aircraft performance through integrated aerodynamics, structures and systems design and allows for consistent and harmonized aerodynamic and aeroelastic data across the flight envelope.

Another key aspect is the reduction of development time for new aviation technology. Today, the development, testing and production of new aircraft involve significant timing and financial risks. These risks and the resulting long aircraft operation spans slow down the introduction of progressive technology and dynamic improvements. For this reason, the German Aerospace Center (DLR) is putting the virtual product at the heart of its scientific work in its guiding concepts for aeronautics research. The virtual product, i.e., high-precision mathematical and numerical representation of a new aircraft and all its characteristics and components, allows faster development cycles; starting from product development up to approval, production and maintenance [6]. To achieve this, further improvements of simulation capabilities as well as computational efficiency and scalability on current and future high performance computing (HPC) systems is pivotal.

In this paper, we share our efforts and experiences in analyzing the performance of CODA, a CFD solver for aircraft aerodynamics developed by DLR, ONERA, and Airbus. While the CFD solver is only one part of FlowSimulator, the larger framework for multi-disciplinary aircraft design, it already reaches the limits of a detailed performance analysis with current performance analysis tools. CODA incorporates innovative algorithms and advanced software technology concepts dedicated to HPC. It is implemented in Python and C++ and uses multi-level parallelization via MPI or GASPI and OpenMP. Our principle contribution is an assessment, from an engineering perspective, of the state of the art in performance analysis tools, a discussion on demands and challenges, and a presentation of first results of the performance analysis of a CODA performance test case. This may provide guidance for researchers in their efforts to analyze and optimize other engineering applications as well as serve as feedback to performance analysis tool developers.

In the following section we provide background on the CFD solver CODA, its part in the larger framework and give a short overview on relevant performance analysis tools. In Sect. 3 we present first results of the performance analysis of CODA. After that, in Sect. 4 we discuss challenges and restrictions we experienced during our efforts to measure and analyze the CFD solver. Finally, we summarize the presented work and draw conclusions in Sect. 5.

2 Background

In this section we provide an introduction to CODA and its surrounding framework FlowSimulator. We give a brief overview of relevant performance tools and discuss the reasoning for the tools used in the performance study.

2.1 The CFD Solver CODA

At the German Aerospace Center (DLR), CFD codes have been developed for decades, several of which are in production, i.e., in regular industrial use. One of them is the DLR *TAU* code [7], which is used in the European aircraft industry, research organizations and academia since more than 15 years. It has more than hundred frequent users and was, for instance, used for the Airbus A380 and A350 wing design. TAU uses a classical MPI-only parallelization to simulate steady as well as unsteady external aerodynamic flows using a 2^{nd} order finite-volumes discretization.

In 2012 DLR decided on the development of a new flexible unstructured CFD solver called *Flucs* [8]. This gave the opportunity to design a modern, comprehensive HPC concept from scratch. However, HPC was only one of the design drivers; among others were: strong fully implicit schemes for improved algorithmic efficiency, higher-order spatial discretization (Discontinuous Galerkin method featuring hp-adaptation) in addition to finite volumes with maximum code share, improved integration into Python-based multi-disciplinary process chains, and modularity.

Though Flucs had been started as a DLR activity, it has become part of a larger development that is driven by Airbus, the French aerospace lab ONERA, and DLR. After Airbus expressed its interest for a new generation CFD solver that is co-developed by ONERA and DLR in 2015, in May 2017 all three parties reached an agreement based on the industrial needs and constraints and decided to pursue the joint effort. The common framework and architecture of the joint development of the CFD solver based on Flucs was called *CODA* to reflect the new collaboration and the involvement of all three partners.

Similar to TAU, CODA uses classical domain decomposition to make use of distributed-memory parallelism. However, CODA features overlapping halo-data communication with computation to hide network latency and, thus, improve scalability. In addition, the GASPI [9] implementation GPI-2 can be used for halo

communication as an alternative to MPI. This Partitioned Global Address Space (PGAS) library features highly efficient one-sided communication, minimizing network traffic as well as latency. Furthermore, CODA features additional sub-domain decomposition, i.e., each domain can again be partitioned into sub-domains, to make use of shared-memory parallelism resulting in a hybrid two-level parallelization. Each sub-domain is processed by a dedicated software thread that is mapped one-to-one to a hardware thread to maximize data locality.

2.2 The Multi-disciplinary Framework FlowSimulator

The CODA CFD solver is not operated as a stand-alone application but rather as a plugin to the multi-disciplinary analysis (MDA) framework *FlowSimulator* [10]. In particular, CODA uses FlowSimulator's core component, the FlowSimulator DataManager (FSDM) for I/O, where various I/O libraries are supported, for instance NetCDF, HDF5, and CGNS. FSDM is an open-source software hosted by DLR [11]. FSDM provides the FSMesh class, which is the preferred container for the exchange of data among FlowSimulator plugins. FSDM is MPI parallelized and an FSMesh instance is a distributed representation of the data, usually containing information on the geometry, the (computational) mesh, flow fields/solutions, as well as coupling strategies. Fully parallel workflows aim at the parallel scalability of MDA processes implemented via FlowSimulator. In addition to the distributed-memory parallelization using MPI (via the FSMesh class), FlowSimulator plugins may feature additional parallelization levels like CODA does.

2.3 Overview of Suitable Performance Analysis Tools

Since CODA is implemented in Python and C++ and uses multi-level parallelization via MPI or GASPI and OpenMP, the main decision criteria for the selection of appropriate performance analysis tools is the extent to which the tools support the standards and their combination. Please note that the following overview of tools we consider suitable is compiled to the best of our knowledge and reflects our personal experiences. Hence, we do not claim that the list is complete neither that the tools may not have recently been extended with the necessary functionality.

The easiest and probably most commonly used method to generate basic performance information for Python programs is via the Python modules *profile* or *cProfile* in combination with *pstats* [12]. Profile and cProfile are mostly interchangeable and provide statistics for accumulated duration and number of invocations for various parts of the program. Any python function can be profiled by calling `cProfile.run(<function>)` instead of `<function>` within a Python script that imports the profiling module. The generated statistics can be formatted into simple text reports via the pstats module. In a parallel execution, output is generated for each

process and is intermingled, which requires additional post-processing to provide meaningful results. Being specific to Python, the Python profile modules neither provide any information for the C++ parts of hybrid Python and C++ applications nor do they support OpenMP or GASPI.

The commercial tools Arm MAP and Intel Vtune claim to support mixed Python and C/Fortran applications. Arm MAP displays system information over time, e.g., CPU and memory utilization. Intel VTune's summarized information is based on periodic sampling for mixed Python and C/Fortran applications. However, we were unable to locate further details on the available Python support for these tools at the moment of writing this work and were unable to test the tools by ourselves since both are commercially distributed and generally unavailable on HPC systems.

Next to these, the well-established parallel performance tools Vampir [13] and Scalasca [14] based on the Score-P measurement environment [15] provide support for Python as well as the BSC tools [16] including the Extrae trace monitor [17] and the Paraver trace analyzer [18] (all except Vampir are open source). Score-P collects information for C/C++ and Fortran applications and supports among others MPI and OpenMP but not GASPI. While the main Score-P distribution does not support Python, there are separately developed Python bindings that allow the usage of Score-P with Python [19]. Since Score-P uses compiler instrumentation, for C++ applications it can create vast amounts of recorded data and an application slowdown of up to a factor of 100 rendering the measurement nearly useless [20, 21].

For our performance measurements we chose the BSC tools for two main reasons. First, Extrae combines the benefits of instrumentation and sampling by intercepting the parallel runtime to provide the exact communication behavior but uses sampling instead of function instrumentation to record the application behavior in the compute phases. Second, Extrae and Paraver have been successfully used before to analyze scientific HPC applications with mixed Python/C++ and hybrid MPI/OpenMP [22].

3 Performance Study of an Exemplary Test Case

CODA/FlowSimulator is one of the key next-generation engineering simulations represented in the European Centre of Excellence for Engineering Applications (EXCELLERAT) [23]. EXCELLERAT's vision is to close the gap between academic research and industrial practice, in particular, to move the European engineering market towards Exascale. It leverages scientific progress in HPC driven engineering by combining the expertise of the HPC centers involved and enables the development of next-generation engineering applications; one of them being CODA.

To utilize current HPC systems and emerging HPC technology, efficient algorithms, both, mathematically and computationally, as well as parallel algorithms with high scalability are of paramount importance. Performance analysis aids both targets by assisting developers not only in identifying performance issues in their applications but also in understanding their behavior on complex HPC systems.

Hence, one of the long-term goals in the development of workflows for virtual aircraft design, similar to other engineering disciplines, is to be capable of measuring and analyzing the performance of the entire application workflow. In the case of CODA, this means analyzing the performance of the entire multi-disciplinary aircraft analysis workflow (MDA), which itself may consist of a variety of different applications, programming languages and degrees as well as levels of parallelism. Towards this goal, the performance analysis of the CFD solver embedded in the workflow already reaches the limits of current performance analysis methods and, therefore, provides sufficient challenges whose solving may move ahead, both, application development and performance analysis tools.

3.1 The Test Case

For an initial analysis of CODA we chose a simple wing-body configuration with horizontal and vertical stabilizer. The test case uses the Reynolds-averaged Navier-Stokes equations (RANS) with a Spalart-Allmaras turbulence model (SA-neg). It uses finite volume spatial discretization with an implicit Euler time integration. The input of the test is a small unstructured tetrahedral mesh with 1.9 million cells. Please note that this rather small mesh (one to two orders of magnitude smaller than industrial cases) was chosen to allow a strong scalability analysis at relatively small core counts, i.e., neither the tetrahedral cells nor the small number of cells allow high CFD accuracy in the boundary layer. The test case simulates steady airflow at subsonic speed and computes typical characteristics like air velocity and direction, pressure and turbulence. Fig. 1 visualizes the test case output with the aircraft configuration and mesh on the left and the airflow around the wing and fuselage with the surface pressure on the aircraft on the right.

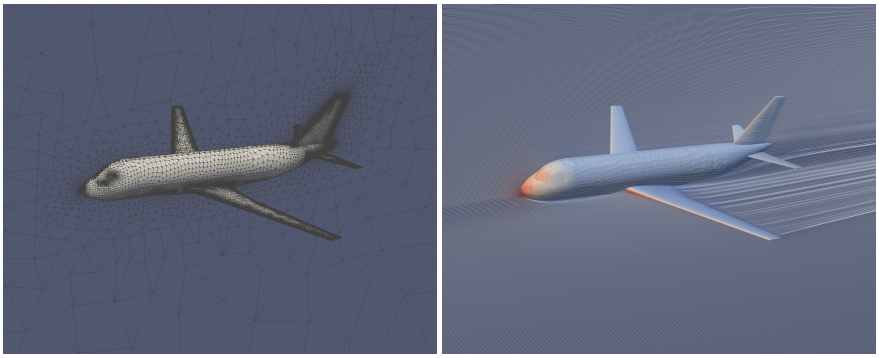

Fig. 1 Visualization of the test case simulation: aircraft configuration with mesh (left) and airflow around wing and fuselage with surface pressure on the aircraft (right).

With this test case, we evaluate two different methods for the partitioning of mesh data to the processes: the recursive coordinate bisection (RCB) method and the graph partitioning method Zoltan [24]. While Zoltan typically provides a better partitioning for the cost of longer partitioning time, it is not fully understood why. Some aspects, such as less communication partners per process and less overall communication have been previously identified for TAU. In this study, we analyze the impact of both partitioners to identify the causes for the different runtime behavior in CODA.

3.2 Measurement Collection

For the initial performance analysis the focus was on strong scalability. For both mesh partitioning methods, we recorded three measurement sets with 1, 2, 4, 8, and 16 nodes, i.e., 24 to 384 cores, on Taurus, a cluster with Intel Haswell CPUs, whereas each node consist of two sockets with 12 cores each, i.e., with two NUMA domains per node. We measured the code in MPI-only mode, i.e., without node-local sub-partition via OpenMP, in order to analyze the parallel behavior in the distributed memory partitioning. We used Extrae 3.7.1 and recorded the top-level Python calls, MPI communication and PAPI counters for the compute regions. The simulation was truncated to the first 10 time steps, which resulted in about 90 seconds of runtime and a manageable amount of about 1.3 GiB of trace data for the largest runs.

3.3 Test Case Analysis Results

For the performance analysis we followed the structured approach to performance analysis proposed by the European Center of Excellence for Performance Optimization and Productivity [25, 26]. The approach organizes the performance analysis into five main phases: first, the collection of a representative set of measurements, second, an overview of the application and the selection of the focus of analysis (FOA), third, the application of a performance model to identify potential issues and quantify their impact, fourth, a detailed analysis guided and prioritized by the performance model, and, fifth and last, the reporting of the applications performance, analysis results and recommendations for performance optimization.

3.3.1 RCB Partitioning Method

Fig. 2 shows an overview of the application behavior with 24 cores based on the measurements with RCB. The timeline displays represent the behavior of the application along time (horizontal) and processes (vertical) and provide a general understanding of the application behavior and simple identification of phases and patterns. The top timeline depicts the entire application execution with the metric Useful Duration,

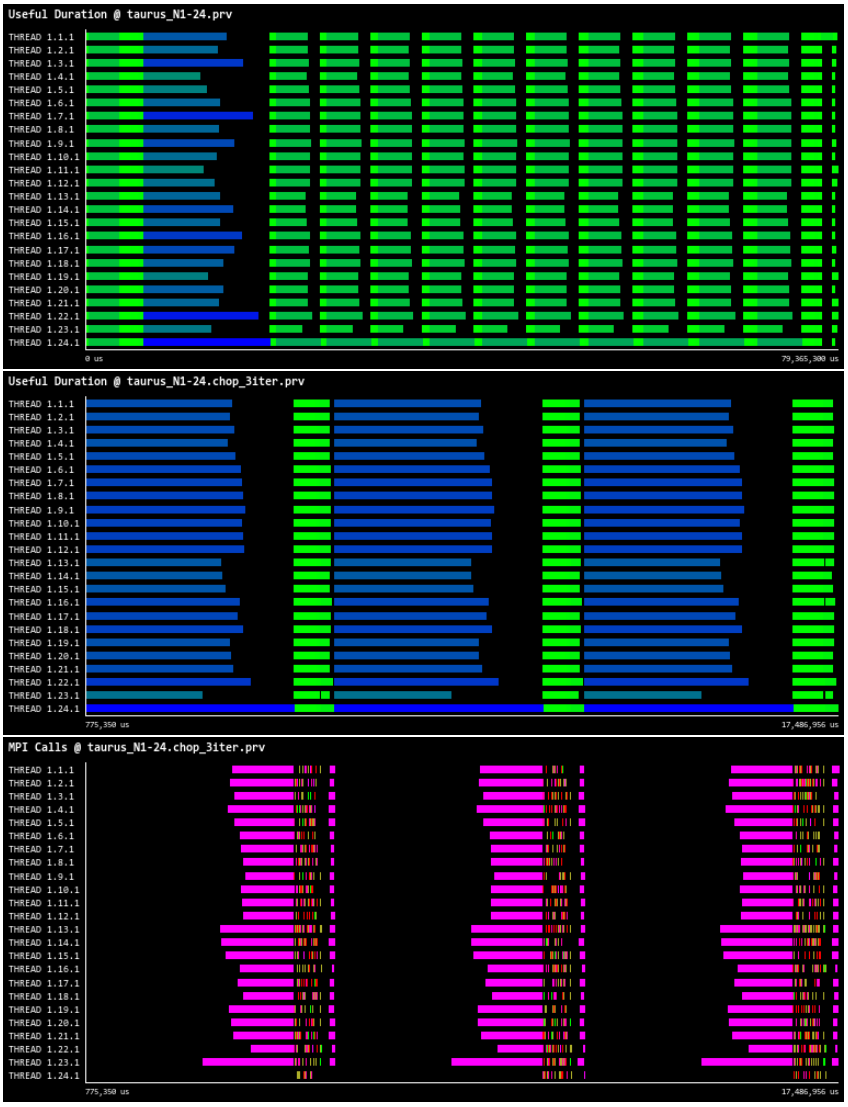

Fig. 2 Application behavior overview using RCB for 24 cores (1 node): application structure of the entire run (top), computation phases of the FOA (middle), and parallel behavior (bottom).

i.e., time spent for computation outside of the parallel runtime (MPI in this case); whereas the color gradient from green to blue represents the length of each compute phase from short to long, respectively; black marks time outside of useful computation, i.e., time in the parallel runtime. After an initialization phase (until about 25 % of the time), the application executes 10 iterations with similar behavior (after which iterations are stopped) and completes with a finalization phase (approx. the last 5 %).

Table 1 Efficiency and scalability factors for the executions using RCB with 24 to 96 processes.

	24	48	96
Global Efficiency	69.8 %	64.3 %	48.1 %
Parallel Efficiency	69.8 %	65.0 %	49.3 %
→ Load Balance	73.3 %	68.9 %	52.5 %
→ Comm Efficiency	95.3 %	94.4 %	94.2 %
Computation Scalability	100.0 %	98.9 %	97.5 %
→ IPC Scalability	100.0 %	100.0 %	99.9 %
→ Instructions Scalability	100.0 %	98.9 %	97.7 %
→ Frequency Scalability	100.0 %	100.0 %	99.9 %

Since the initialization and finalization phase are overrepresented and the iterations show no significant deviations along time, we selected the iterations four to six from the execution as focus of analysis (FOA). The second and third timeline of Fig. 2 depict the distribution of the compute phases and the parallel behavior with MPI, respectively. Each iteration consists of a large computation phase (blue, middle timeline) followed by many smaller computation bursts (green). The large computation is terminated by a global call to *MPI_Allreduce* (pink, bottom timeline). Within the smaller computation bursts there is a mix of non-blocking point-to-point communication (*MPI_Isend*, *MPI_Irecv*, *MPI_Waitany*) and global calls to *MPI_Allreduce*.

After determining the focus of analysis, we applied a performance model [27] to this application phase. The performance model combines fundamental performance factors that allow quantifying parallel efficiency and scalability with a single percentage value as well as providing an easy, high-level comparison of different executions. The performance model computes the *global efficiency*, i.e., the overall performance rating, based on the two main components: *parallel efficiency* and *computation scalability*. The parallel efficiency provides an overall assessment of the parallel behavior of the application and is expressed as the product of *load balance* and *communication efficiency*. The computation scalability describes the evolution of the total time spent in computation of multiple executions and, therefore, is only meaningful for comparing multiple executions, e.g. with increasing core counts. It can be further detailed in the scalability of IPC (instructions per cycle), instructions, and frequency. The performance model is described in more detail in [25, 27].

Table 1 shows an overview of the fundamental performance factors based on the performance model. While the performance model can be computed manually, Paraver's basic analysis package [16] computes all the performance factors automatically, which frees the user from manually collecting the data for the performance model and avoids potential errors in the process.

The observed global efficiency of the test case with RCB decreases from 69.8 % with 24 cores to 48.1 % with 96 cores. The decreasing global efficiency is mainly caused by a decreasing load balance, which is already rather low for the smallest measurement causing a rather low global efficiency to begin with. The other main

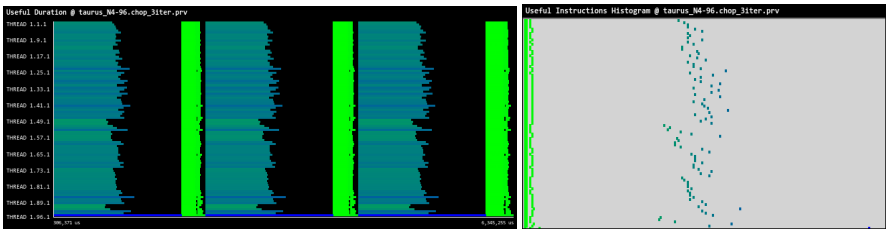

Fig. 3 Load balance with RCB using 96 cores: timeline with the compute phases (left) and a histogram showing the distribution of compute phases based on their number of instructions (right).

factors achieve very good values: the communication efficiency is very high, i.e., very little time is spent in MPI communication. Similarly, the computation scales very well and the computation efficiency is generally very high with an average of 2.2 instructions per cycle (IPC).

The further detailed analysis is focused and prioritized based on the performance model. In this case, the low load balance that decreases with scale is the main issue. Fig. 3 depicts the effects of the load imbalance within the focus of analysis; whereas the left side gives an overview of the different duration of the compute phases and the right side shows a histogram with the distribution of compute phases based on their number of instructions. The histogram represents for each process on the vertical axis the distribution of compute phases categorized by their number of instructions (horizontal axis). The color gradient reflects the duration of the compute phase and, thus, is identical with the gradient in the timeline on the left. The histogram allows for an easy identification of balance in the program: a perfectly balanced phase would form a straight line from top to bottom, while an imbalanced phase would produce a scattered pattern; the more scattered the higher the imbalance.

In this case, the imbalance in execution time strongly correlates with the imbalance in the number of instructions (52 % balance), while the IPC is well balanced (96 %, not shown here). This means, the origin of the imbalance in time is directly linked to an imbalanced distribution of the workload to the processes. In addition to the general imbalance in the large compute phase, the last processes carries extra load that is not linked to an extra task but to more of the same workload (small blue dot on the bottom right of the histogram in Fig. 3). Combining the load balance analysis with the performance model allows to quantify the potential performance gains if all load balance issues would be completely solved: 31 % runtime improvement when redistributing the extra workload on the last process and additional 10 % and 6 % runtime improvement for perfectly balancing the workload in the large compute phase and the smaller ones, respectively.

Based on the assessment that load balance is linked to the data partitioning, it can be concluded that the RCB leads to an unfavorable distribution of the mesh data. A further analysis revealed that, while the RCB partitioning method distributes the mesh nodes well, it leads to a poor partitioning of the mesh's volume elements (tetrahydra). However, since CODA's finite volume method uses a cell-centered metric, the partitioning of volume elements is much more important.

Table 2 Efficiency and scalability factors for the executions using Zoltan with 24 to 384 processes.

	24	48	96	192	384
Global Efficiency	97.7 %	96.0 %	93.8 %	91.5 %	80.8 %
Parallel Efficiency	97.7 %	96.1 %	94.5 %	92.8 %	85.1 %
→ Load Balance	98.0 %	96.4 %	95.6 %	94.5 %	91.7 %
→ Communication Efficiency	99.7 %	99.8 %	98.9 %	98.2 %	92.9 %
Computation Scalability	100.0 %	99.8 %	99.2 %	98.5 %	94.1 %
→ IPC Scalability	100.0 %	100.3 %	100.4 %	100.6 %	98.5 %
→ Instructions Scalability	100.0 %	98.3 %	97.7 %	96.9 %	94.5 %
→ Frequency Scalability	100.0 %	101.2 %	101.2 %	101.1 %	101.0 %

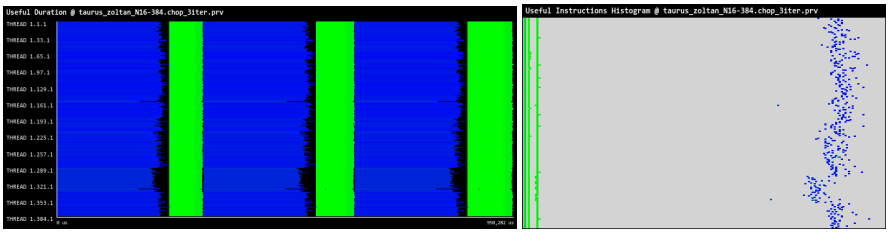

Fig. 4 Load balance with Zoltan: timeline with the compute phases (left) and a histogram showing the distribution of compute phases based on their number of instructions (right).

3.3.2 Zoltan Partitioning Method

For the measurements using the Zoltan graph partitioning method we proceeded as described above. Since the iterations show no significant deviations along time, as before, we again selected the iterations four to six from the execution as focus of analysis (FOA) and applied the performance model.

Table 2 shows an overview of the fundamental performance factors for the measurements with Zoltan. The observed global efficiency of the test case with Zoltan decreases from 97.7 % with 24 cores to 80.0 % with 384 cores with a noticeable drop from 192 to 384 cores. The global efficiency achieves a good value, in particular, since 384 cores is already beyond the target scale for such a small mesh. We identified three performance issues that diminish the overall performance.

First, the load balance achieves a much higher value compared to RCB but is still sub-optimal. The in-depth analysis reveals that the load balance is again mainly linked to the distribution of mesh elements (see Fig. 4), however, this time achieving acceptable values for unstructured mesh data.

Second, the communication efficiency decreases to 92.9 % with 384 cores, whereas the communication achieves almost perfect efficiency up to 192 cores. The detailed analysis for 384 cores found that the communication efficiency for the large compute phase (blue in Fig. 4) is almost ideal with 99.8 % but reaches only 75.4 % in the smaller compute phases (green in Fig. 4), where the communication

is dominated by small non-blocking point-to-point communication operations. For a small mesh at this scale, the communication overhead of the up to 630 small communication operations per process within only 70 ms starts becoming too significant in relation to the actual computation effort per process.

Third and last, the computation scalability decreases to 94.1 % with 384 cores, which is mainly linked to decreasing instructions scalability. Instructions scalability describes the evolution of the computational workload and is measured by the total number of instructions in computation over all processes in comparison to the reference execution. Thus, an instructions scalability of 94.1 % signifies a parallel workload replication of about 6.2 % versus the smallest run. The percentage of additional workload is again correlated to a relatively high scale for this small mesh, where control flow operations start becoming significant in relation to operations dedicated to computing the solution.

3.3.3 Analysis Summary

In comparison to the RCB partitioning method, the Zoltan graph partitioning allows for a significantly better distribution of volume elements and, thus, a much higher load balance. The test case with the graph partitioner achieves a much better scalability with a speedup of 3.84 out of 4 for 96 processes versus 2.76 with the RCB partitioner; and 13.1 out of 16 for 384 processes, which is already beyond the target scale for such a small mesh. Furthermore, the overall runtime at 96 processes was reduced by 46 % due to a much higher load balance of 95.6 % in comparison to 68.9 % before. In general, the test case achieves a very high parallel efficiency and computational scalability for such a small mesh size when using the Zoltan graph partitioning.

4 Challenges in the Performance Analysis of Engineering Codes

Although state-of-the-art performance analysis tools have been incredibly helpful by providing insight into the parallel application behavior that allowed us to assess the performance of the test case and understand why the two partitioning methods behave so differently, their use was not without certain pitfalls and limitations. This section discusses challenges and limitations we experienced during the analysis and arising requirements for performance analysis tools.

Hybrid MPI-OpenMP parallelization: The CODA CFD solver implements a two-level hybrid parallelization with MPI/GASPI and OpenMP. Such a hybrid parallelization still poses a challenge to most performance tools and excludes all tools that focus on only one parallelization level.

GASPI: Up to the time of writing, we were unable to find a performance analysis tool that officially supports the GASPI standard. Consequently, we are unable to perform any detailed analysis of the one-sided communication operations via GASPI.

Mixed programming language: Since CODA is implemented in Python and C++, performance tools require the capability to measure program sections implemented in both programming languages. For many tools this is not given. While Python related tools are mostly limited to Python and its native parallel constructs, many well-established HPC performance tools are restricted to common HPC programming languages like C/C++ or Fortran.

C++ templates: CODA makes use of many modern C++ features including a wide use of templates. As a result, many tools that rely on automatic compiler instrumentation can produce vast data volumes and significant application slow down when small, typically inlined, functions are instrumented and recorded, since the introduced measurement overhead drastically exceeds the call time of tiny functions such as get/set class methods. Potential solutions to this issue are either a combination of detailed instrumentation for the parallel runtimes and periodic sampling for the computation phases or intelligent compile time filters for the automatic function instrumentation.

Heterogeneous systems: For computationally intensive code sections such as the sparse linear systems solver the multi-layered parallelization is additionally extended to incorporate hardware accelerators like GPUs. This adds another requirement to performance measurement and representation to analyze CODA in all its complexity.

Thread-level performance: Next to parallel efficiency, thread-level performance, i.e., per-core computational efficiency, is an important aspect of numerical simulation. While many parallel performance analyzers provide basic capabilities in the form of hardware performance counters, e.g. cache misses, and derived metrics such as instructions per cycle (IPC), a quantitative assessment and an in-depth analysis of computational efficiency is currently not supported.

Different software versions: FlowSimulator relies on different external dependencies and sometimes on specific software packages or even specific software versions. This may cause interference with dependencies of performance analysis tools. To avoid this, applications and tools need to be installed with compatible software chains. While this is not an inherent limitation of performance tools, it adds another layer of complexity to software installations on HPC systems, in particular, if there is a variety of different applications with various software dependencies.

While the listed requirements originate from CODA, we expect similar requirements for other engineering applications. When going forward towards the performance analysis of not a single engineering application but rather entire engineering workflows, we anticipate even more complexity in the various requirements. Such workflows may include even more different programming languages, other parallelization schemes, different levels of parallelism in the different components of the workflow, more software dependencies, and generally more complexity that needs to be captured and represented by performance analysis tools.

5 Conclusion

This paper highlights our efforts in analyzing CODA, a CFD solver for aircraft aerodynamics, as part of a larger framework for the multi-disciplinary analysis in aircraft design. We demonstrate how state-of-the-art performance analysis tools provide profound insight into the parallel application behavior, which helped us to assess the performance of a CODA scalability test case and, in particular, understand how the chosen partitioning method impacts the parallel runtime behavior. In addition, we share our experience during the analysis and discuss challenges and limitations as well as arising requirements for performance analysis tools. We hope that this will be beneficial for researchers in their efforts to analyze and optimize other engineering applications as well as serve as feedback to performance analysis tool developers.

Acknowledgements This work has been supported by the EXCELLERAT project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 823691 and the German Federal Aviation Research Programme (LuFo) under grand agreement No. 20X1704A (cooperative project TOSCANA).

References

1. International Civil Aviation Organization: Annual Report of the Council 2018.
2. Airbus: Airbus Global Market Forecast 2019-2038.
3. Air Transport Action Group (ATAG): The economic and social benefits of air transport 2008.
4. Intergovernmental Panel on Climate Change (IPCC): Climate Change 2014: Synthesis Report. Contribution of Working Groups I, II and III to the Fifth Assessment Report of the International Panel on Climate Change.
5. Directorate-General for Mobility and Transport (European Commission), Directorate-General for Research and Innovation (European Commission): Flightpath 2050: Europe's vision for aviation: maintaining global leadership and serving society's needs (2012) DOI: https://doi.org/10.2777/15458
6. Guiding concepts for DLR aeronautics research. https://www.dlr.de/EN/research/aeronautics/guiding-concepts.html [Online; accessed 2019-10-08]
7. Dieter Schwamborn, Thomas Gerhold, Ralf Heinrich: The DLR TAU Code: Recent Applications in Research and Industry. In *Proc. of the European Conference on Computational Fluid Dynamics, ECCOMAS CFD* (2006)
8. Tobias Leicht, Daniel Vollmer, Jens Jägersküpper, Axel Schwöppe, Ralf Hartmann, Jens Fiedler and Tobias Schlauch: DLR-Project Digital-X – Next Generation CFD Solver 'Flucs'. Deutscher Luft- und Raumfahrtkongress 2016
9. Thomas Alrutz, Jan Backhaus, Thomas Brandes, Vanessa End, Thomas Gerhold, Alfred Geiger, Daniel Grünewald, Vincent Heuveline, Jens Jägersküpper, Andreas Knüpfer, Olaf Krzikalla, Edmund Kügeler, Carsten Lojewski, Guy Lonsdale, Ralph Müller-Pfefferkorn, Wolfgang E. Nagel, Lena Oden, Franz-Josef Pfreundt, Mirko Rahn, Michael Sattler, Mareike Schmidtobreick, Annika Schiller, Christian Simmendinger, Thomas Soddemann, Godehard Sutmann, Henning Weber and Jan-Philipp Weiss: GASPI – A Partitioned Global Address Space Programming Interface. In *Facing the Multicore-Challenge III*, LNCS 7686, pp. 135–136 (2013) DOI: https://doi.org/10.1007/978-3-642-35893-7_18

10. Michael Meinel and Gunnar Einarsson: The FlowSimulator Framework for Massively Parallel CFD Applications. In *PARA 2010*
11. FlowSimulator. https://gitlab.as.dlr.de [Online; accessed 2019-10-08]
12. The Python Profilers. https://docs.python.org/2/library/profile.html [Online; accessed 2019-09-12]
13. A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel: The Vampir Performance Analysis Tool Set. In *Tools for High Performance Computing*, pp. 139–155 (2008) DOI: https://doi.org/10.1007/978-3-540-68564-7_9
14. M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr: The Scalasca Performance Toolset Architecture. In *Concurrency and Computation: Practice and Experience*, 22(6):702–719 (2010) DOI: https://doi.org/10.1002/cpe.1556
15. A. Knüpfer, C. Rössel, D. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, pp. 79–91 (2012) DOI: https://doi.org/10.1007/978-3-642-31476-6_7
16. BSC Tools. http://tools.bsc.es [Online; accessed 2019-09-12]
17. Extrae instrumentation package. http://tools.bsc.es/extrae [Online; accessed 2019-09-12]
18. Paraver: a flexible performance analysis tool. http://tools.bsc.es/paraver [Online; accessed 2019-09-12]
19. Score-P Python bindings. https://github.com/score-p/scorep_binding_python [Online; accessed 2019-10-08]
20. Michael Wagner, Jens Doleschal and Andreas Knüpfer: Tracing long running applications: A case study using Gromacs. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*, pp. 129–136 (2015) DOI: https://doi.org/10.1109/HPCSim.2015.7237031
21. Michael Wagner, Jens Doleschal, Andreas Knüpfer and Wolfgang E. Nagel: Selective Runtime Monitoring: Non-intrusive Elimination of High-frequency Functions. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*, pp. 295–302 (2014) DOI: https://doi.org/10.1109/HPCSim.2014.6903698
22. Michael Wagner, Germán Llort, Estanislao Mercadal, Judit Giménez, and Jesús Labarta: Performance Analysis of Parallel Python Applications. In *Procedia Computer Science*, 108:2171–2179 (2017) DOI: https://doi.org/10.1016/j.procs.2017.05.203
23. The European Centre of Excellence for Engineering Applications (EXCELLERAT). http://www.excellerat.eu [Online; accessed 2019-09-12]
24. Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson and Courtenay Vaughan: Zoltan Data Management Services for Parallel Dynamic Applications. In *Computing in Science and Engineering* 4(2), pp. 90–97 (2002) DOI: https://doi.org/10.1109/5992.988653
25. Michael Wagner, Stephan Mohr, Judit Giménez, and Jesús Labarta: A Structured Approach to Performance Analysis. In *Tools for High Performance Computing 2017*, pp. 1–15 (2019) DOI: https://doi.org/10.1007/978-3-030-11987-4_1
26. The European Centre of Excellence for Performance Optimization and Productivity (POP). http://www.pop-coe.eu [Online; accessed 2019-09-12]
27. Claudia Rosas, Judit Giménez, and Jesús Labarta: Scalability Prediction for Fundamental Performance Factors. In *Supercomputing Frontiers and Innovations*, 1(2) (2014) DOI: https://doi.org/10.14529/jsfi140201