DOI: 10.14529/jsfi200402

Enhancing the in Situ Visualization of Performance Data in Parallel CFD Applications

Rigel F. C. Alves[1], ***Andreas Knüpfer***[1]

This paper continues the work initiated by the authors on the feasibility of using *ParaView* as visualization software for the analysis of parallel CFD codes' performance. Current performance tools are unable to show their data on top of complex simulation geometries (e.g. an aircraft engine). In our previous paper, a plugin for the open-source performance tool *Score-P* has been introduced, which intercepts an arbitrary number of manually selected code *regions* (mostly functions) and send their respective measurements – *amount* of executions and cumulative *time* spent – to ParaView (through its in situ library, *Catalyst*), as if they are any other flow-related variable. This paper adds to such plugin the capacity to also show communication data (messages sent between MPI ranks) on top of the CFD mesh. Testing is done again with Rolls-Royce's in-house CFD code, *Hydra*. The plugin's original feature (regions' measurements) is here revisited, in a bigger test-case, which is also used to illustrate the new feature (communication data). The benefits and overhead of the tool are discussed.

Keywords: parallel computing, performance analysis, in situ processing, computational fluid dynamics.

Introduction

Computers have become mandatory resources in solving engineering problems. For the size of today's typical ones (like designing aircraft), one needs to *parallelize* the simulation (e.g. of the air flowing through the airplane's engine) and run it in High Performance Computing (HPC) hardware. Those are expensive infrastructures, both from *time* and *energy* consumption point-of-views. Therefore the application needs to have its parallel performance high-tunned for maximum productivity.

There are many tools for analyzing the performance of parallel applications; one of them is *Score-P*[2] [9], the development of which the *Centre for Information Services and HPC* (ZIH) of the Technische Universität Dresden participates in. It instruments the simulation code and monitors its execution, and can be easily turned on or off by the user at compile time. When applied to a source code, the simulation will produce in the end, apart from its native outputs, also the performance data. This is illustrated in the upper part of Fig. 1 below.

However, all tools currently available to *visualize* the performance data (generated by software like Score-P) lack important features, like three-dimensionality, time-step association (i.e. frame playing) and most importantly, matching to the simulation original geometry (where everything happens in terms of computations and therefore where load imbalances lie).

As a separate category of add-ons, tools for enabling *in situ* visualization of applications' output data – like temperature or pressure in a Computational Fluid Dynamics (CFD) simulation – already exist too; one example is *Catalyst*[3] [3]. They also work as an optional layer to the original code and can be activated upon request, by means of preprocessor directives

[1]Technische Universität Dresden, Center for Information Services and High Performance Computing (ZIH), Dresden, Germany

[2]*Scalable Performance Measurement Infrastructure for Parallel Codes* – an open-source "highly scalable and easy-to-use tool suite for profiling, event tracing, and online analysis of HPC applications" [tool's website].

[3]An open-source "in situ use case library, with an adaptable application programming interface (API), that orchestrates the delicate alliance between simulation and analysis and/or visualization tasks" [tool's website].

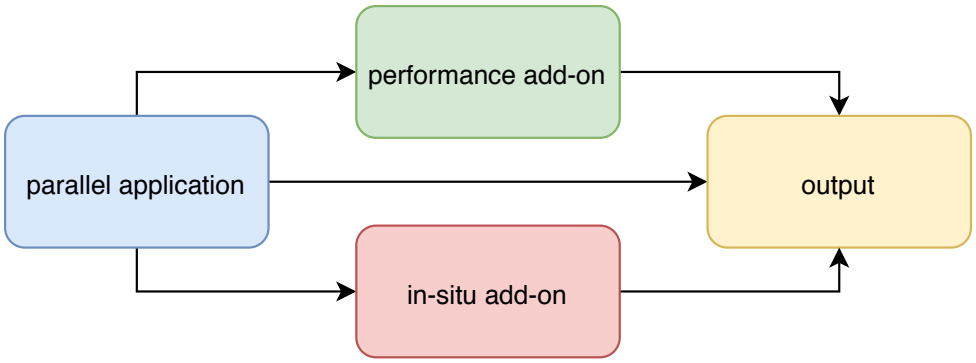

Figure 1. Schematic of software components for parallel applications

at compilation stage. The simulation will then produce its native outputs, if any,[4] plus the *coprocessor*'s (a piece of code responsible for allowing the original application to interact with the in situ methods) ones, in separate files. This is also illustrated in Fig. 1. These tools have been developed by visualization specialists for decades long by now and feature abundant visual resources.

Then the question comes: why not use such in situ tools (made to extract data from the simulation by separate side channels, just like the performance instrumenters) for the benefit of the performance analysis of parallel applications (filling by that the lack of visual resources of the performance tools)?

This work continues our investigations on the feasibility of merging the aforementioned approaches. First, by unifying the overlapping functionalities of both kinds of tools, insofar as they augment a parallel application with additional features (which are not strictly required for the application to work in the first place). Second, by using the advanced functionalities of dedicated visualization software for the purpose of performance analysis. Figure 2 illustrates the idea.

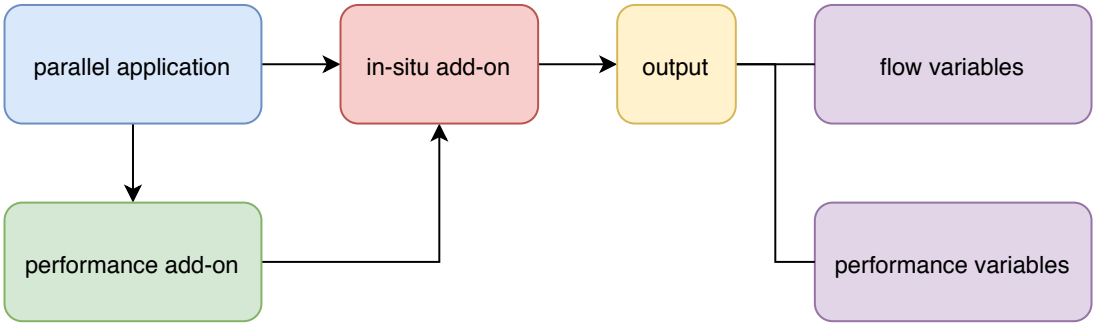

Figure 2. Schematic of the software components for a combined add-on

In our previous paper [2], we mapped performance measurements of code regions – *amount* of executions and cumulative *time* spent – to the simulation's geometry, just like it is done for flow-related properties. In this paper, we shall add to our tool the capacity of showing communication data (messages sent between MPI ranks) on top of the CFD mesh. We will then see how communication inefficiencies become immediately visible.

HPC analysis tools usually produce either *performance profiles* or *event traces*. In the case

[4] I.e. if the in situ channel does not replace completely the code's original outputs, as it uses to happen.

of Score-P, they are: performance profiles in the Cube4 format, to be visualized with *Cube*[5]; and parallel event traces in the OTF2 format, to be visualized with *Vampir*[6]. But neither of them, nor the other currently available tools, nor the related attempts in the literature (to be summarized in Section 1 below), display their measurements onto complex geometries (like those found in industry-grade CFD problems), what makes our proposal novel.

A design requirement is that the combined solution must be easily applicable on the source code, yet without becoming a permanently required component: it needs to be "activatable" on demand, as it is the case for each of its constitutive parts (*performance measurement* and *in situ processing*). As evaluation case, the Rolls-Royce's in-house CFD code (*Hydra*) will be used.

This paper is organized as follows: in Section 1 we discuss the efforts made so far at the literature to map performance data to the simulation's geometry and the limitations of their results. In Section 2 we present the methodology of our approach, which is then evaluated in the test-case in Section 3. Finally, Section 4 discusses the overhead associated with using our tool. We then conclude the article with a summary and point directions for future work.

1. Related Work

In order to assist the developer of parallel codes in its optimization tasks, many software tools have been developed. For a comprehensive list of them, including information about their:

- *scope*, whether single or multiple nodes (i.e. shared or distributed memory);
- *focus*, be it performance, debugging, correctness or workflow (productivity);
- *programming models*, ranging through MPI, OpenMP, Pthreads, OmpSs, CUDA, OpenCL, OpenACC, UPC, SHMEM and their combinations;
- *languages*: C, C++, Fortran or Python;
- *processor architectures*: x86, Power, ARM, GPU;
- license types, platforms supported, contact details, output examples etc.

the reader is referred to the *Tools Guide* of the *Virtual Institute – High Productivity Supercomputing* (VI-HPS). However, none of them currently match the generated data back to the simulation original geometry.

The necessity of bringing together the branches of *performance analysis* and *visualization* has already been identified by the scientific community [4, 7] and is being pursued at research level. Vierjahn et al. [12] mapped performance data to the simulation geometry, but developing the visualization environment from scratch (the test-case was indeed simpler than a CFD problem). Going this way, it would take decades for the tool to reach the same capabilities of today's top graphic programs. Huck et al. [5], on the other hand, did follow our approach: performance tool *TAU* was linked to visualization software *VisIt* to show performance data on top of the Earth's oceans in a climate simulation; however the linking required writing a new VisIt file format reader. To reproduce those results would require the effort of manually recreating such interface for each different (CFD) simulation code, which was undesired. Husain et al. [6] followed a similar path, also using VisIt as visualization tool, but MemAxes as performance measurer – a software which does not seem to have a website, what impacts on its availability. Their results required modifying the source code of both tools involved, which was again undesired. Finally,

[5]A free, but copyrighted "generic tool for displaying a multi-dimensional performance space consisting of the dimensions (i) performance metric, (ii) call path, and (iii) system resource" [tool's website].

[6]An "easy-to-use framework that enables developers to quickly display and analyze arbitrary program behavior at any level of detail" [tool's website].

similar hurdles can be encountered in the work of Wood et al. [13]: the application needs to be first enveloped by a multipurpose framework (SOSflow) and then linked with a non widely known in situ infrastructure (Ascent) in order for performance data to be shown on simulation geometries which are comprised of parallelepipedic rectilinear grids.

Not without reason, all attempts (to match performance data to the simulation's geometry) described above were unable to test their features on more complex (CFD) meshes: the required user effort precluded it... We will then advance the state-of-the-art by aiming for a solution which uses an already established way of extracting data from a simulation, which can be directly (i.e. without wrapping layers) applied to any numerical code, running any type of mesh.

2. Methodology

2.1. Prerequisites

The goal aimed by this research depends on the combination of two basic, scientifically established methods: *performance measurement* and *in situ processing*.

2.1.1. Performance measurement

When applied to a source file compilation, Score-P automatically inserts probes between each code "region"[7], which will at run-time measure a) the *number of times* that region has been executed and b) the total *time* spent in those executions, by each process (MPI rank) within the simulation. Its application is done by simply prepending the word `scorep` into the compilation command, e.g.: `scorep [Score-P's options] mpicc foo.c`. It is possible to exclude regions from the instrumentation (e.g. to keep the related overhead low), by adding the flag `--nocompiler` to the command above. In this case, Score-P sees only user-defined regions (if any) and MPI-related functions, the detection of which can be easily (de)activated at run-time, by means of an environment variable: `export SCOREP_MPI_ENABLE_GROUPS=[comma-separated list]`. Leaving it blank turns off instrumentation of MPI routines. Its default value is set to catch all of them.

Finally, the tool is also equipped with an API, which allows the user to extend its functionalities through plugins [11]. The combined solution proposed by this paper takes indeed the form of such a plugin.

2.1.2. In situ processing

In order for Catalyst to interface with a simulation code, an *adapter* needs to be built, which is responsible for exposing the native data structures (mesh and flow properties) to the *coprocessor* component. Its interaction with the simulation code happens through three function calls (*initialize*, *run* and *finalize*), illustrated in blue at Fig. 3. Once implemented, the adapter provides the generation of post-mortem files (by means of the *VTK*[8] library) and/or the live visualization of the simulation, both through *ParaView*[9] [1].

[7]Every "function" is naturally a "region", but the latter is a broader concept and includes any user-defined aggregation of code lines, which is then assigned a name. It could be used e.g. to aggregate all instructions pertaining to the main solver (time-step) loop.
[8]An open-source "software for manipulating and displaying scientific data" [tool's website].
[9]An open-source "multi-platform data analysis and visualization application" [tool's website].

```
int main(int argc, char **argv)
{
    MPI_Init(& argc, & argv);
    #ifdef USE_CATALYST
        initialize_coprocessor_();
    #endif

    // STARTING PROCEDURES...

    #ifdef CATALYST_SCOREP
        // tell the plugin that the time-step loop is about to start
        cat_sco_initialize_();
    #endif
    // MAIN SOLVER LOOP
    for (int time_step = 0; time_step < num_time_steps; time_step++)
    {
        // COMPUTATIONS...

        #ifdef USE_CATALYST
            run_coprocessor_(time_step, time_value, ...);
        #endif
        #ifdef CATALYST_SCOREP
            // tell the plugin to process the current time step
            cat_sco_run_(time_step, time_value);
        #endif
    }
    #ifdef CATALYST_SCOREP
        // tell the plugin that the time-step loop is over
        cat_sco_finalize_();
    #endif

    // ENDING PROCEDURES...

    #ifdef USE_CATALYST
        finalize_coprocessor_();
    #endif
    MPI_Finalize();
    return 0;
}
```

Figure 3. Illustrative example of changes needed in a simulation code due to Catalyst (blue) and then due to the plugin (violet)

2.2. Combining both Tools

In our previous paper [2], we have introduced a Score-P plugin, which allows performance measurements for an arbitrary number of manually selected code regions to be mapped to the simulation original geometry. In this paper, we are extending our software to *pipeline* (i.e. send for visualization) also communication data (messages exchanged between ranks) on top of the CFD mesh. The plugin must be activated at run-time through an environment variable (`export SCOREP_SUBSTRATE_PLUGINS=Catalyst`), but works independently of Score-P's *profiling* or *tracing* modes being actually on or off. Like Catalyst, it requires three function calls (*initialize*, *run* and *finalize*) to be inserted in the source code, illustrated in violet at Fig. 3. Additionally, a call must be placed before each function to be pipelined:

```
#ifdef CATALYST_SCOREP
    ! send the following region's measurements to ParaView
    CALL cat_sco_pipeline_next_()
#endif

    CALL desired_function(argument_1, argument_2...)
```

Figure 4. Illustrative example of the call to tell the plugin to show the upcoming function's measurements in ParaView

The above layout ensures that the desired function will be captured when executed at that specific moment and not in others (if the same routine is called multiple times – with different inputs – throughout the code, as it is usual for CFD simulations). The selected functions may be nested. This is not needed for the new feature of our tool (show communication on the simulation geometry), as the instrumentation of MPI routines is done independently at run-time (see Section 2.1.1 above).

Finally, the user needs to add a small piece of code into its simulation Catalyst adapter, in order for the plugin-generated variables to be pipelined, as shown on Fig. 5.[10] The first part is related to the selected regions inside the simulation code (the original feature of our tool); it contains two vectors since for each selected region the plugin will generate two variables (which correspond to the two basic measurements made by Score-P, as explained in Section 2.1.1 above). The second part refers to the tracking of communication between MPI ranks (the new feature of our tool); it also contains two vectors for each of the supported calls (MPI_Send, Isend, Get and Put), one to store the *amount of times* that specific call was made (inside the time-step), another to store the total *amount of bytes* transported through those calls.

3. Evaluation

3.1. Settings

Hydra is Rolls-Royce's in-house CFD code [10], based on a preconditioned time marching of the Reynolds-averaged Navier-Stokes (RANS) equations. They are discretized in space us-

[10]In order for the plugin to work with simulation codes written in C or Fortran, its three main calls have name mangling and no namespaces. However, given VTK requires C++ features, a Catalyst adapter needs to be written in C++, hence the plugin calls shown on Fig. 5 are free from such restrictions.

```
#ifdef CATALYST_SCOREP
  // Related to the selected regions
  std::vector<vtkNew<vtkUnsignedIntArray> >freq(cat_sco::meas::get_size());
  std::vector<vtkNew<vtkDoubleArray     > >time(cat_sco::meas::get_size());

  for (std::size_t i = 0; i < cat_sco::meas::get_size(); ++i)
  {
    freq[i] -> SetName( (cat_sco::meas::get_name(i) + " : freq").c_str() );
    time[i] -> SetName( (cat_sco::meas::get_name(i) + " : time").c_str() );
    freq[i] -> SetNumberOfComponents(1);
    time[i] -> SetNumberOfComponents(1);
    freq[i] -> SetNumberOfTuples(vtk_grid->GetNumberOfPoints() );
    time[i] -> SetNumberOfTuples(vtk_grid->GetNumberOfPoints() );
    freq[i] -> FillTypedComponent(0, cat_sco::meas::get_counter(i));
    time[i] -> FillTypedComponent(0, cat_sco::meas::get_time   (i));

    vtk_grid -> GetPointData() -> AddArray(freq[i].GetPointer() );
    vtk_grid -> GetPointData() -> AddArray(time[i].GetPointer() );
  }

  // Related to communication
  std::vector<vtkNew<vtkUnsignedIntArray > >counter(cat_sco::comm::get_size());
  std::vector<vtkNew<vtkUnsignedLongArray> >bytes  (cat_sco::comm::get_size());
  std::stringstream name;

  for (std::size_t i = 0; i < cat_sco::comm::get_size(); ++i)
  {
    name << "MPI_Put to " << i;
    counter[i] -> SetName( (name.str() + " : counter").c_str() );
    bytes  [i] -> SetName( (name.str() + " : bytes"  ).c_str() );
    counter[i] -> SetNumberOfComponents(1);
    bytes  [i] -> SetNumberOfComponents(1);
    counter[i] -> SetNumberOfTuples(vtk_grid->GetNumberOfPoints() );
    bytes  [i] -> SetNumberOfTuples(vtk_grid->GetNumberOfPoints() );
    counter[i] -> FillTypedComponent(0, cat_sco::comm::get_counter_put(i));
    bytes  [i] -> FillTypedComponent(0, cat_sco::comm::get_bytes_put  (i));

    vtk_grid -> GetPointData() -> AddArray(counter[i].GetPointer() );
    vtk_grid -> GetPointData() -> AddArray(bytes  [i].GetPointer() );
    name.str(""); name.clear();
  }
#endif
```

Figure 5. Illustrative example of the addition needed in the simulation Catalyst adapter due to the plugin

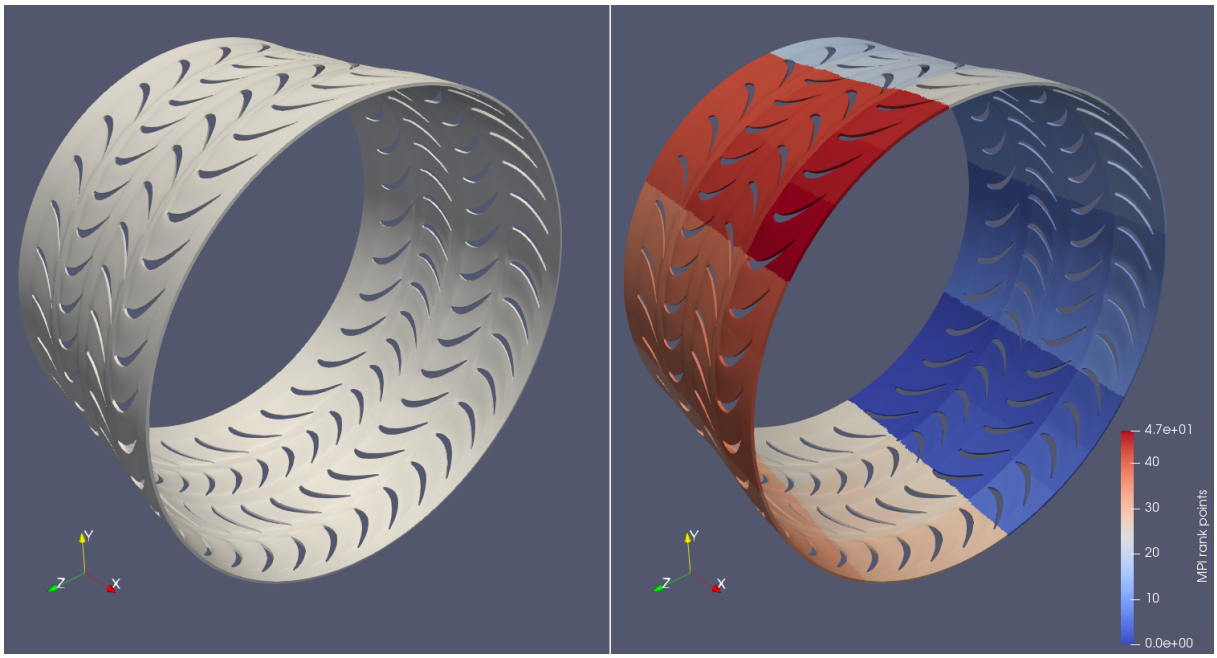

Figure 6. Geometry used in the simulations (left) and its partitioning among processes for parallel execution (right)

ing a second-order, edge-based finite volume scheme with a multistage, explicit Runge-Kutta scheme as a steady time marching approach. Multigrid and local time-stepping acceleration techniques are used to improve steady-state convergence [8]. Figure 6 shows the test-case selected for this paper: it represents a simplified (single cell thickness), 360° experimental grid of two turbine stages in an aircraft engine, discretized through roughly 1 million points. Unsteady RANS calculations have been made with second-order, time-accurate dual time-stepping. Turbulence modelling was based on standard 2-equation closures. Preliminary analyses with Score-P and Cube revealed two code functions to be especially time-consuming: *iflux_edge* and *vflux_edge* (both mesh-related); they were selected for pipelining.

The simulations have been done using two entire *Haswell* nodes of Dresden University's HPC cluster (Taurus), each with 24 ranks (i.e. pure MPI, no OpenMP), one per core and with the entire core memory (2583 MB) available. Processes are pinned by default. Figure 6 shows the domain partitioning among the ranks, done in a geometric fashion – which ensures a similar number of grid points between each sub-domain[11] – and not subject to any stochastic variance.

One full engine shaft rotation was simulated, comprised of 200 time-steps (i.e. one per 1.8°), each internally converged through 40 iteration steps. Catalyst was generating post-mortem files every 20$^{\text{th}}$ time-step (i.e. every 36°), what led to 10 stage pictures by the end of the simulation.

Finally, everything was built / tested with release 2018a of Intel® compilers in association with versions 4.0 of Score-P and 5.5.2 of ParaView.

3.2. Results

3.2.1. Original feature – manually selected code regions

Figure 7 shows the amount of executions, per time-step, of the two selected regions; it is constant in every time-step and not subject to any stochastic variance. From the picture it

[11] It does not look so in the picture because the grid gets finer in the x direction.

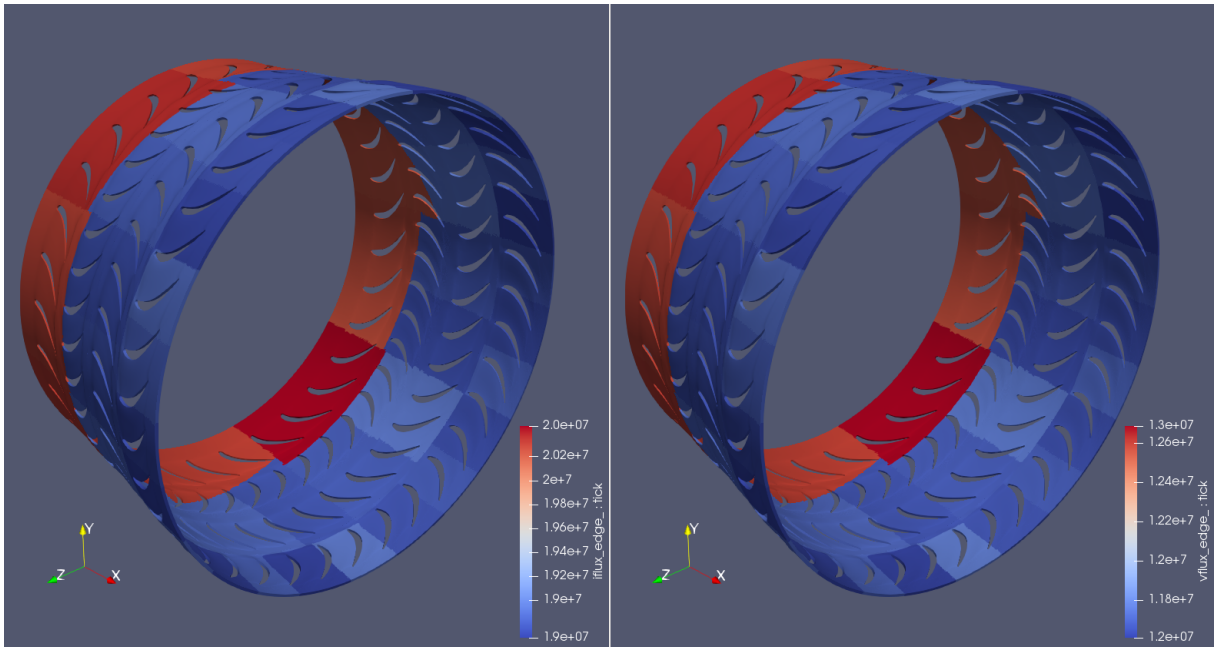

Figure 7. Amount of executions, per time-step, of two selected code functions (*iflux_edge* on the left, *vflux_edge* on the right) in the test-case

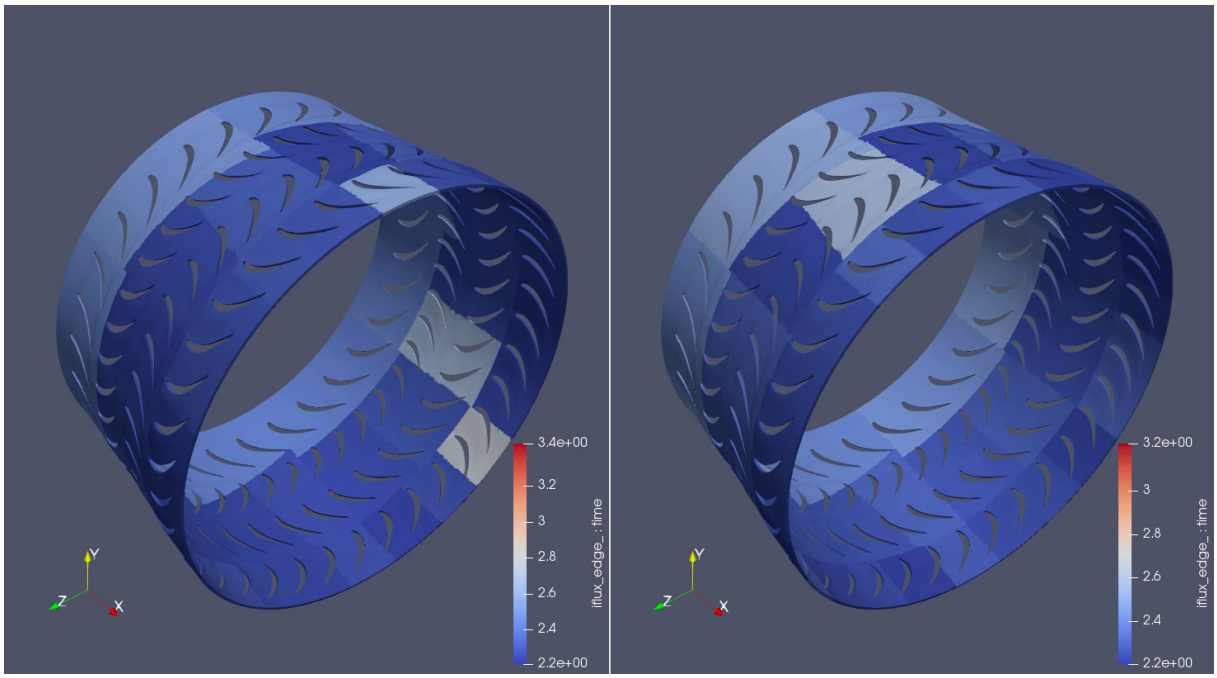

Figure 8. Total time spent in function *iflux_edge*, in an arbitrary time-step, on two consecutive runs of the test-case

is visible that ensuring a similar number of mesh *points* between the sub-domains does not necessarily mean an equally similar number of *edges*,[12] as both functions are applied at the edge level and their amount of executions differ up to 1 million times (every time-step) between maximum and minimum among the ranks. There is a clear bias towards overloading the sub-domains closer to the turbine inlet (the air flows in the positive x direction), plus the creation of a chess-like pattern (interleaved areas of higher and lower number of executions).

[12]Hydra works with *unstructured* meshes: grid cells do not need to be of uniform type across the entire domain.

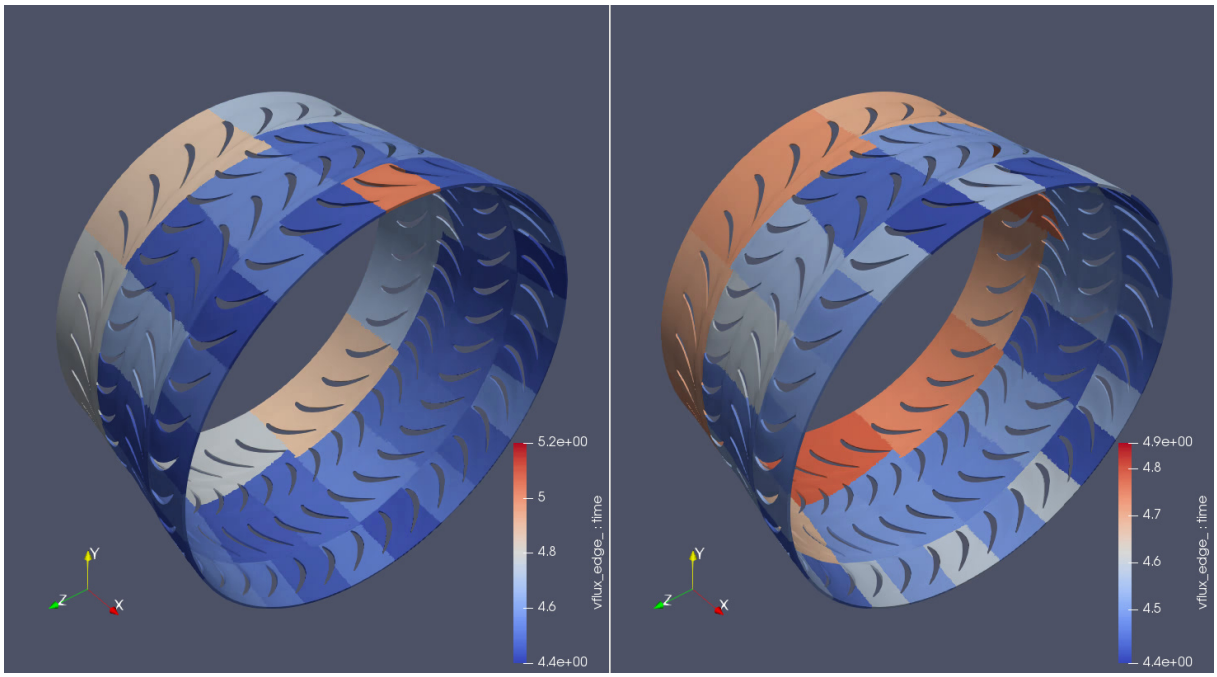

Figure 9. Total time spent in function *vflux_edge*, in an arbitrary time-step, on two consecutive runs of the test-case

The aforementioned distortions indeed reflect on the code performance. Figures 8 and 9 show the *total* – i.e. comprising all executions – time spent (in seconds) in the selected regions in an arbitrary time-step: they change every time-step and are subject to stochastic variance (hence two pictures per function, each referring to the same time-step at consecutive runs of the test-case). Both the bias in the inlet / outlet direction and the chess-like pattern are visible; furthermore, it becomes clear that the load imbalance is stronger in *vflux_edge*.

3.2.2. New feature – tracking of communications

Mapping communications data to the simulation geometry is a new feature of our plugin and is being presented here for the first time. Figure 10 shows the location of an arbitrary subdomain (left) and those communicating with it (right), colored by the amount of messages sent (in this case, MPI_Isend calls) in an arbitrary time-step of the test-case. It represents the expected behavior: only the neighbors communicate with the selected rank (the last one, number 47). However, this is not the case for other subdomains within the same simulation: Figure 11 shows the same information as the previous one, but now for rank number 1. Notice how many non-neighbors communicate with it in the selected time-step, and thousand of times indeed. This means an unneeded burden on the simulation run-time and should be avoided. It actually could be avoided, as such non-neighbors are not properly sending any data through those thousands of MPI calls, as revealed by Fig. 12, which corresponds to Fig. 11, but now coloring the sender subdomains by total amount of *bytes* sent at the time-step shown.

Such conclusions are only made possible thanks to the mapping of communication data back to the simulation geometry; it would indeed be difficult to reach the same insights from the traditional way of showing communication traces (lines crossing each other on a two-dimensional, horizontal bar-chart-like visualization). It has proved the benefits of our tool.

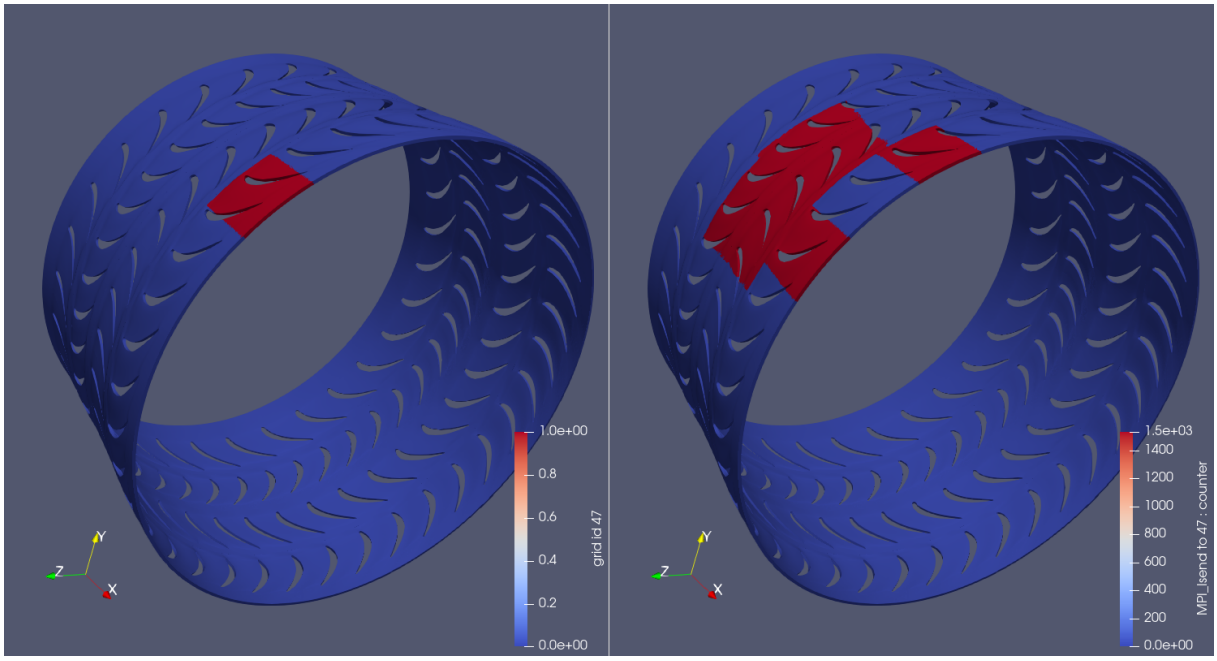

Figure 10. Location of an arbitrary subdomain (left) and those communicating with it (right), colored by the amount of messages sent (in this case, MPI_Isend calls) in an arbitrary time-step of the test-case

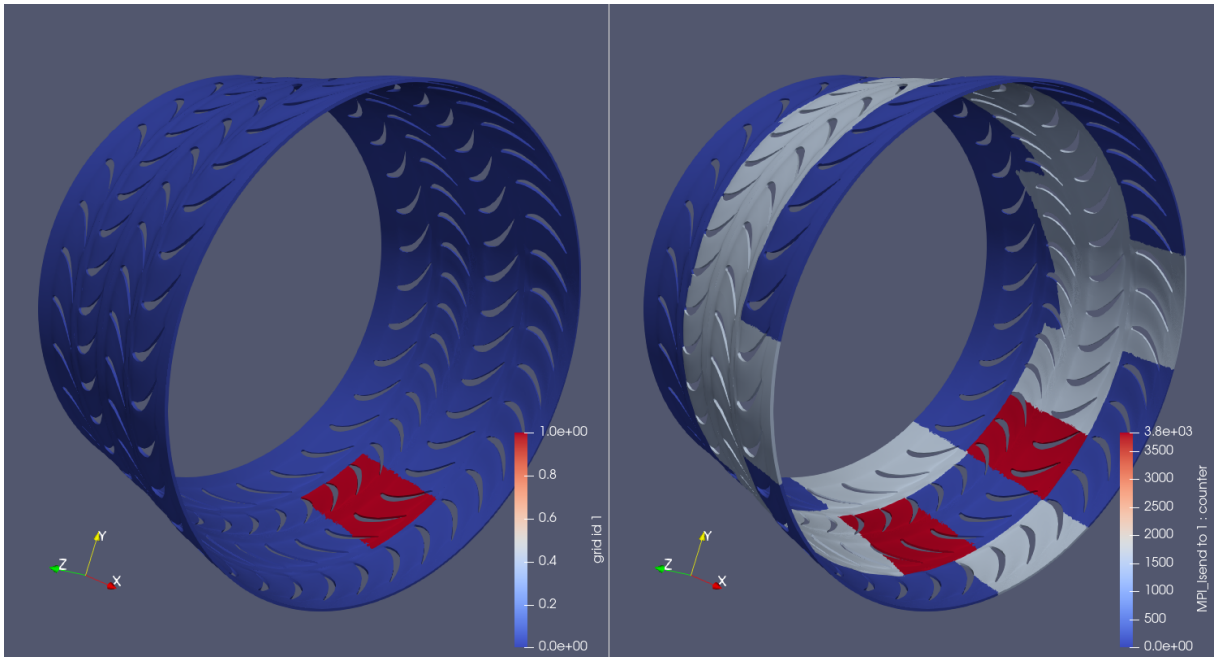

Figure 11. Location of another arbitrary subdomain (left) and those communicating with it (right), colored by the amount of messages sent (in this case, MPI_Isend calls) in an arbitrary time-step of the test-case

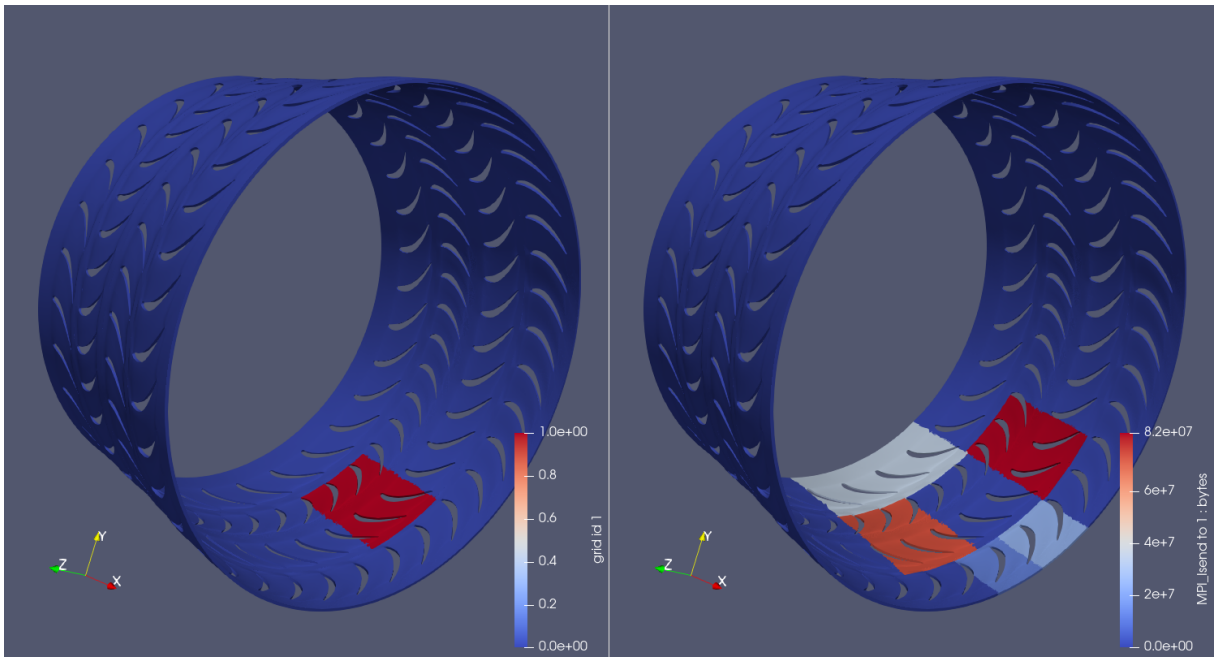

Figure 12. Location of another arbitrary subdomain (left) and those communicating with it (right), colored by the amount of bytes sent (in this case, through MPI_Isend calls) in an arbitrary time-step of the test-case

4. Overhead

Provided we are talking about performance analysis, it is important to analyse the impact of our tool itself on the performance of the instrumented code execution.

4.1. Settings

In the following table, the *baseline* results refer to the pure simulation code, running as per the settings presented in Section 3; the numbers given are the average of 5 runs ± 1 relative standard deviation. The *+ Score-P* results refer to when Score-P is added onto it, running with both profiling and tracing modes deactivated (as neither of them is needed for the plugin to work)[13]. Finally, *++ plugin* refers to when the plugin is also used: running only one *feature* (regions or communication) at a time[14] and on the iterations when there would be generation of output files[15].

Score-P has been always applied with the `--nocompiler` option. When the plugin is used to show communication between ranks, this option will be enough, as no instrumentation (manual or automatic) is needed when only MPI calls are being tracked. On the other hand, when the goal is to measure code regions, the instrumentation overhead is considerably higher, as every single function inside the simulation code is a potential candidate for the analysis (as opposed to when tracking communications, when only MPI-related calls are intercepted). In this case,

[13]If present, there would be at the end of the simulation, apart from the simulation output files, those generated by Score-P for visualization in Cube (profiling mode) or Vampir (tracing mode). Their generation can co-exist with the plugin execution, but it is not recommended: the overheads sum up.

[14]The plugin can perfectly run in all its features simultaneously. However, this is not recommended: the overheads sum up.

[15]Given the simulation was not being visualized live in ParaView, there was no need to let the plugin work in time-steps when no data would be saved to disk.

```
#ifdef SCOREP_USER
#include "scorep/SCOREP_User.inc"
#endif
        ! {...}
        subroutine IFLUX_EDGE(...)
          implicit none
#ifdef SCOREP_USER
          SCOREP_USER_REGION_DEFINE( iflux_region )
#endif
          ! {variable declarations}
#ifdef SCOREP_USER
          if(MODULO(time_step, 20) == 0 .OR. time_step == 1) then
            SCOREP_USER_REGION_BEGIN(iflux_region, "iflux_edge",
      &     SCOREP_USER_REGION_TYPE_COMMON)
          endif
#endif
          ! {function body}
#ifdef SCOREP_USER
          if(MODULO(time_step, 20) == 0 .OR. time_step == 1) then
            SCOREP_USER_REGION_END( iflux_region )
          endif
#endif
          return
        end
```

Figure 13. Example of a manual (user-defined) code instrumentation with Score-P; the optional `if` clauses ensure measurements are collected only at the desired time-steps

it is necessary to add the `--user` Score-P compile flag and manually instrument the simulation code (i.e. only the desired regions were visible to Score-P). This is achieved by means of an intervention as illustrated in Fig. 13: `if MODULO...` additionally ensures measurements are collected solely when there will be generation of output files and at time-step 1 – the reason for it is that Catalyst runs even when there is no post-mortem files being saved to disk (as the user may be visualizing the simulation live) and the first time-step is of special importance, as all data arrays must be defined then (i.e. the (dis)appearance of variables in later time-steps is not allowed)[16]. Finally, when measuring code functions, interception of MPI-related calls has been turned off at run-time[17].

4.2. Results

Table 1 shows the impact of the proposed plugin on the test-case performance. The memory section refers to the *peak* memory consumption per rank, reached *somewhen* during the

[16]Hence, in the end there were two narrowing factors for Score-P: the *spacial* (i.e. accompany only the desired functions) and the *temporal* (accompany only at the desired time-steps) ones.
[17]By means of the `SCOREP_MPI_ENABLE_GROUPS` environment variable (see Sec. 2.1.1 above).

Table 1. Overhead results of the plugin on the test-case's performance

	running time			memory (MB)		
	++ plugin	**+ Score-P**	**baseline**	**++ plugin**	**+ Score-P**	**baseline**
regions	47m12s (6%)	46m30s (5%)	44m20s ± 2%	405 (–4%)	410 (–3%)	423 ± 1%
comm.	49m21s (11%)	46m38s (5%)	44m20s ± 2%	468 (11%)	410 (–3%)	423 ± 1%

simulation; it neither means that *all* ranks need that much memory (simultaneously or not), nor that the memory consumption is like that during the *entire* simulation. Score-P's individual footprint is so small that it lies within the statistical margin of oscillation of the value itself; the same applies to the plugin footprint when measuring the two code regions. Tracking communications, on the other hand, adds many more data arrays to ParaView (two per rank per type of communication call), hence the associated overhead is higher.

The run-time overhead, in its turn, is more critical. But fortunately it lies within acceptable thresholds. The regions feature has again been less burdensome, but that has to do with how many regions are being intercepted within the source code (here, only two of them).

Conclusions

In this paper, we have extended our software that allows mapping parallel performance data back to the simulation geometry, by means of (combining) the code instrumenter *Score-P* and the in situ library *Catalyst*, resulting on three-dimensional, time-stepped (framed) visualizations in the graphical program *ParaView*. The tool, which takes the form of a Score-P plugin, is capable of matching to the domain mesh (e.g. an aircraft engine):

- measurements for an arbitrary number of manually selected code regions – original feature of the tool, introduced in our previous paper [2] and here revisited (in a bigger test-case);
- communication data (messages exchanged between MPI ranks) – new feature of the tool, presented here for the first time.

All that is based exclusively on open-source dependencies. The tool source code is available at https://gitlab.hrz.tu-chemnitz.de/alves–tu-dresden.de/catalyst-score-p-plugin.

The advantage of using ParaView as visualization software comes to all the resources already available in – and experience accumulated by – it after decades of continuous development. Visualization techniques are usually not the specialization field of researchers working with code performance: it is more reasonable to take advantage of the currently available graphic programs than attempting – from scratch – to equip the existing profiling tools with their own GUIs. In this threshold, the developed plugin makes load imbalances and communication inefficiencies easier to identify. It works independently of Score-P's *profiling* or *tracing* modes and with either *automatic* or *manual* code instrumentation. Finally, like Catalyst itself, its output frequency (when doing post-mortem analyses) is adjustable at run-time (through the plugin input file).

Future Work

We plan to continue this work in distinct directions:

More extensive evaluation cases.

To run the plugin in bigger test-cases, as the difficulty in matching each parallel region id number (the MPI rank) with its respective grid part (hence the benefit of matching performance data back to the simulation mesh) increases with scaling. Concomitantly, to run the plugin in test-cases which comprise regions with distinct flow physics, when the computational load becomes less dependent on the number of points / cells per domain and more dependent on the flow features themselves (given their non-uniform occurrence): chemical reactions in the combustion chamber, shock waves in the inlet / outlet (at the supersonic flow regime), air dissociation in the free-stream / inlet (at the hypersonic flow regime) etc.

Develop new visualization schemes for performance data.

To take advantage of the many filters available in ParaView for the benefit of the code optimization branch, e.g. by recreating in it the statistical analysis – display of *average* and *standard deviation* between the threads/ranks measurements – typically available in performance tools.

Acknowledgments

The authors would like to thank: Rolls-Royce Germany (especially Marcus Meyer, Axel Gerstenberger, Jan Suhrmann & Paolo Adami), for providing both the industrial CFD code and its test-case for this paper, what has been done in the context of BMWi research project *Prestige* (FKZ 20T1716A); Kitware (also part of Prestige; especially Mathieu Westphal & Nicolas Vuaille), for the support on implementing the Catalyst adapter; the Score-P support and development team (especially Bert Wesarg), for the assistance with the plugin API; and the Taurus admins (especially Maik Schmidt), for the help with the supercomputer.

References

1. Ahrens, J., Geveci, B., Law, C.: Paraview: An end-user tool for large data visualization. The visualization handbook 717 (2005)

2. Alves, R.F.C., Knüpfer, A.: In situ visualization of performance-related data in parallel CFD applications. In: Schwardmann, U., Boehme, C., B. Heras, D., et al. (eds.) Euro-Par 2019: Parallel Processing Workshops. pp. 400–412. Springer International Publishing, Cham (2020), DOI: 10.1007/978-3-030-48340-1_31

3. Ayachit, U., Bauer, A., Geveci, B., et al.: Paraview Catalyst: Enabling in situ data analysis and visualization. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, ISAV2015, Austin, TX, USA. pp. 25–29. ACM (2015), DOI: 10.1145/2828612.2828624

4. Bremer, P.T., Mohr, B., Pascucci, V., et al.: Connecting Performance Analysis and Visualization. Dagstuhl Manifestos 5(1), 1–24 (2015), DOI: 10.4230/DagMan.5.1.1

5. Huck, K.A., Potter, K., Jacobsen, D.W., et al.: Linking performance data into scientific visualization tools. In: 2014 First Workshop on Visual Performance Analysis. pp. 50–57. IEEE (2014), DOI: 10.1109/VPA.2014.9

6. Husain, B., Giménez, A., Levine, J.A., et al.: Relating memory performance data to application domain data using an integration API. In: Proceedings of the 2nd Workshop on Visual Performance Analysis, VPA '15, Austin, Texas. ACM (2015), DOI: 10.1145/2835238.2835243

7. Isaacs, K.E., Giménez, A., Jusufi, I., et al.: State of the Art of Performance Visualization. In: Borgo, R., Maciejewski, R., Viola, I. (eds.) EuroVis - STARs. The Eurographics Association (2014), DOI: 10.2312/eurovisstar.20141177

8. Khanal, B., He, L., Northall, J., et al.: Analysis of Radial Migration of Hot-Streak in Swirling Flow Through High-Pressure Turbine Stage. Journal of Turbomachinery 135(4) (2013), DOI: 10.1115/1.4007505

9. Knüpfer, A., Rössel, C., Mey, D.a., et al.: Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Brunst, H., Müller, M.S., Nagel, W.E., et al. (eds.) Tools for High Performance Computing 2011. pp. 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), DOI: 10.1007/978-3-642-31476-6_7

10. Lapworth, L.: Hydra-CFD: a framework for collaborative CFD development. In: International Conference on Scientific and Engineering Computation, IC-SEC, June, Singapore. vol. 30 (2004), https://www.researchgate.net/publication/316171819_HYDRA-CFD_A_Framework_for_Collaborative_CFD_Development

11. Schöne, R., Tschüter, R., Ilsche, T., et al.: Extending the functionality of Score-P through plugins: Interfaces and use cases. In: Niethammer, C., Gracia, J., Hilbrich, T., et al. (eds.) Tools for High Performance Computing 2016. pp. 59–82. Springer International Publishing, Cham (2017), DOI: 10.1007/978-3-319-56702-0_4

12. Vierjahn, T., Kuhlen, T.W., Müller, M.S., et al.: Visualizing performance data with respect to the simulated geometry. In: Di Napoli, E., Hermanns, M.A., Iliev, H., et al. (eds.) High-Performance Scientific Computing. pp. 212–222. Springer International Publishing, Cham (2017), DOI: 10.1007/978-3-319-53862-4_18

13. Wood, C., Larsen, M., Gimenez, A., et al.: Projecting performance data over simulation geometry using SOSflow and ALPINE. In: Bhatele, A., Boehme, D., Levine, J.A., et al. (eds.) Programming and Performance Visualization Tools. pp. 201–218. Springer International Publishing, Cham (2019), DOI: 10.1007/978-3-030-17872-7_12