In Situ Visualization of Performance-Related Data in Parallel CFD Applications

Rigel F. C. Alves(✉) and Andreas Knüpfer

Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, 01062 Dresden, Germany
rigel.alves@tu-dresden.de
https://tu-dresden.de/zih/

Abstract. This paper aims at investigating the feasibility of using ParaView as visualization software for the analysis and optimization of parallel CFD codes' performance. The currently available software tools for reading profiling data do not match the generated measurements to the simulation's original mesh and somehow aggregate them (rather than showing them on a time-step basis). A plugin for the open-source performance tool Score-P has been developed, which intercept 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 were any other flow-related variable. Results show that (i) the impact of mesh partition algorithms on code performance and (ii) the load imbalances (and their eventual relationship to mesh size/simulation physics) become easier to investigate.

Keywords: Parallel computing · Performance analysis · In situ processing

1 Introduction

Many tools for analyzing the performance of parallel applications exist; one example of them is *Score-P*[1]1 [11], whose development the University of Dresden participates in. It acts as a wrapper which encapsulates the original code, thus can be easily turned on or off by the user at compilation stage. This is illustrated in Fig. 1 below.

The original version of this chapter was revised: The two videos were added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-48340-1_64

[1] *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": https://www.vi-hps.org/projects/score-p/.

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-030-48340-1_31) contains supplementary material, which is available to authorized users.

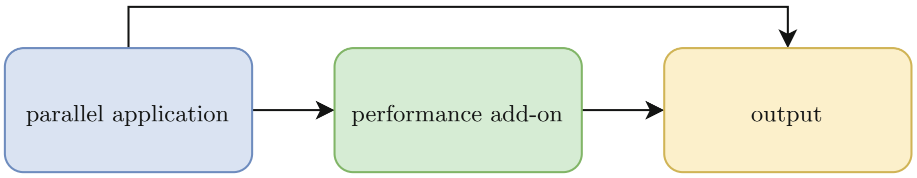


Fig. 1. Schematic of software components for performance analysis tools.

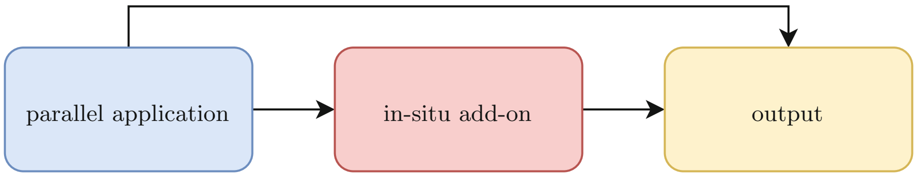


Fig. 2. Schematic of software components for in situ visualization.

As a separate category of add-ons, tools for enabling *in situ visualization* [5] of applications' output data (like *temperature* or *pressure* in a CFD simulation) already exist too; one example is *Catalyst*[2] [3]. It also works as an optional add-on to the original code and can be activated upon request, by means of preprocessor directives at compilation stage (Fig. 2).

This paper's goals are two-fold. First, unify the overlapping functionalities of both kinds of tools insofar as they augment a parallel application with additional functionality which is not strictly required for the application to work in the first place. Both collect or "steal" data from the parallel application and transfer it out via a side channel. Second, make use of the advanced visualization functionalities of dedicated visualization software tools for the purpose of performance analysis. With this we propose to map parallel performance properties to the simulation geometry as it is already done for flow-related properties. Figure 3 illustrates the idea.

The high-performance computing (HPC) performance tools usually output either *performance profiles* or *event traces*. In the case of Score-P, they are:

- performance profiles in the Cube4 format to be visualized at *Cube*[3] [14];
- parallel event traces in the OTF2 format to be visualized at *Vampir*[4] [10].

But neither of them, nor the other currently available performance tools (to be explained in Sect. 2), match their measurements to the original simulation's geometry; what makes the proposal novel. On the other hand, the proposal is

[2] 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": https://www.paraview.org/in-situ/.

[3] 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": http://www.scalasca.org/software/cube-4.x/download.html.

[4] An "easy-to-use framework that enables developers to quickly display and analyze arbitrary program behavior at any level of detail": https://vampir.eu/.

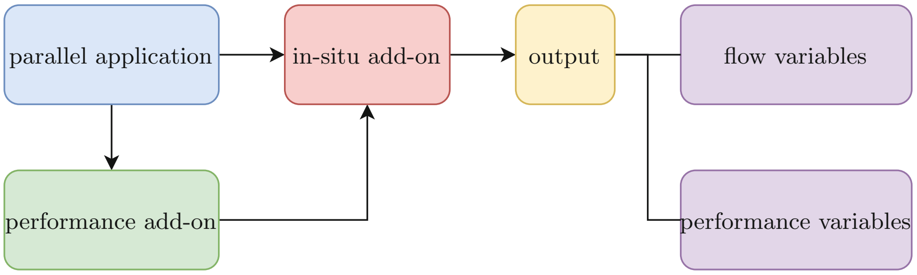


Fig. 3. Schematic of the software components for a combined add-on.

deemed also useful as, especially in CFD applications, the partitioning of the compute mesh for parallelization has direct influence on performance and load balancing. Hence for performance analysis and optimization a combined view into simulation properties and performance properties is helpful.

A design requirement is that the combined solution must be able to be integrated into a parallel code easily, yet without becoming a permanently required component. Instead, it needs to be easy to switch on and off on demand, as it is for each of its constitutive parts. As evaluation case, the Rolls-Royce in-house CFD code (*Hydra*) [12] will be used.

2 Related Work

Apart from Score-P → Cube and Score-P → Vampir (mentioned above), other workflows – with graphical support – used for performance analysis include:

– *HPCToolkit*[5] [1], whose outputs are visualized through *hpcviewer* (profiling) and *hpctraceviewer* (tracing);
– *Periscope*[6] [7], whose outputs are visualized through *Pathway* (an Eclipse-based graphical user-interface);
– *Tau*[7] [18], whose outputs are visualized through *ParaProf* [6] (profiling) and Vampir (tracing), among others;
– *Paraver*[8] [13] and *Dimemas*[9] [4], with integrated visualization capabilities;

[5] An open-source "integrated suite of tools for measurement and analysis of program performance on computers": http://hpctoolkit.org/.

[6] A free "suite of tools designed to assist the HPC application developer in the optimization of their application": https://periscope.in.tum.de/.

[7] A "portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, UPC, Java, Python": http://www.cs.uoregon.edu/research/tau/home.php.

[8] A "very powerful performance visualization and analysis tool based on traces that can be used to analyse any information that is expressed on its input trace format": https://tools.bsc.es/paraver.

[9] A "simulation tool for the parametric analysis of the behaviour of message-passing applications on a configurable parallel platform": https://tools.bsc.es/dimemas.

- *SLOG-2*, a drawable logging format visualized through *Jumpshot*[10] [8];
- *Scalasca*[11] [9] as an optional add-on to either Score-P or Tau;
- *READEX*[12] [17], with a bunch of visualization options.

None of them, however, currently match the generated data back to the simulation's geometry. Furthermore, displaying profiling results on a time-step basis is not straightforward. This paper would like to address those issues.

3 Prerequisites

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

3.1 Performance Measurement

When applied to a source file's compilation, Score-P automatically inserts probes between each code "region" (mostly function calls, but also constructors, destructors etc.), which will at run-time measure:

- the *number of times* that region was executed, and;
- the total *time* spent in those executions.

By each rank/thread within the simulation. Its application is done by simply prepending the word *scorep* into the compilation command, e.g.: `scorep mpicc foo.c`. The tool is also equipped with an API, which allows the user to extend its functionalities through plugins [15]. The combined solution proposed by this paper takes the form of such a plugin.

3.2 In Situ Processing

In order for Catalyst to interface with the 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, illustrated in Fig. 4.

Once implemented, the adapter allows the generation of post-mortem files (by means of the *VTK*[13] [16] library) and/or the live visualization of the simulation, both through *ParaView*[14] [2].

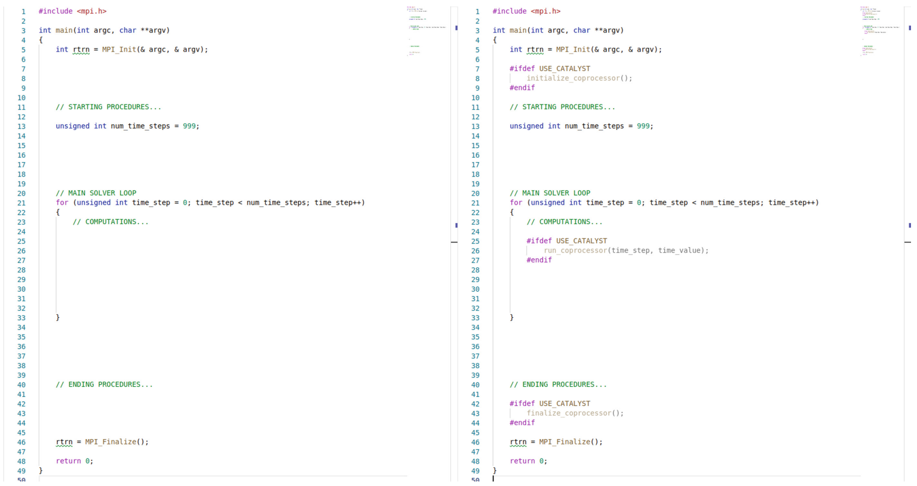
[10] A "Java-based visualization tool for doing postmortem performance analysis": https://www.mcs.anl.gov/research/projects/perfvis/.

[11] A "a software tool that supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior": http://www.scalasca.org/.

[12] A tool suite that "supports users to improve the energy-efficiency of their HPC applications": https://www.readex.eu/.

[13] An open-source "software for manipulating and displaying scientific data": https://www.vtk.org/.

[14] An open-source "multi-platform data analysis and visualization application": https://www.paraview.org/.



```
#include <mpi.h>

int main(int argc, char **argv)
{
    int rtrn = MPI_Init(& argc, & argv);




    // STARTING PROCEDURES...

    unsigned int num_time_steps = 999;






    // MAIN SOLVER LOOP
    for (unsigned int time_step = 0; time_step < num_time_steps; time_step++)
    {
        // COMPUTATIONS...









    }






    // ENDING PROCEDURES...





    rtrn = MPI_Finalize();

    return 0;
}
```

```
#include <mpi.h>

int main(int argc, char **argv)
{
    int rtrn = MPI_Init(& argc, & argv);

    #ifdef USE_CATALYST
        initialize_coprocessor();
    #endif

    // STARTING PROCEDURES...

    unsigned int num_time_steps = 999;






    // MAIN SOLVER LOOP
    for (unsigned int time_step = 0; time_step < num_time_steps; time_step++)
    {
        // COMPUTATIONS...

        #ifdef USE_CATALYST
            run_coprocessor(time_step, time_value);
        #endif






    }






    // ENDING PROCEDURES...

    #ifdef USE_CATALYST
        finalize_coprocessor();
    #endif

    rtrn = MPI_Finalize();

    return 0;
}
```

Fig. 4. Illustrative example of changes needed in a simulation code due to Catalyst.

4 Combining Both Tools

A Score-P plugin has been developed, which allows performance measurements for an arbitrary number of manually selected code regions to be pipelined to the simulation's Catalyst adapter. It must be activated at run-time through an environment variable (`export SCOREP_SUBSTRATE_PLUGINS=Catalyst`), but works independently of Score-P's profiling mode being actually on or off. Figure 5 illustrates the modifications needed in the source.

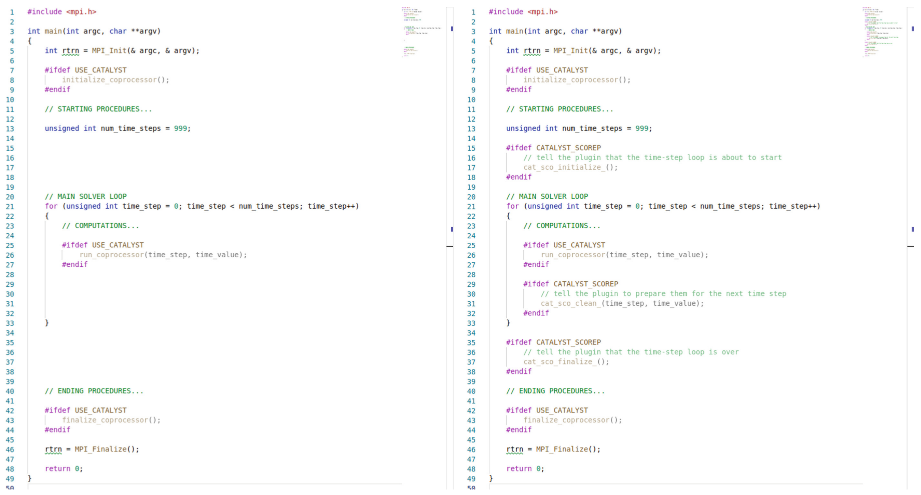
Apart from the three basic calls (*initialize*, "*run*" and *finalize*; like with the Catalyst adapter), a call must be placed immediately before each function to be pipelined; e.g.:

```
#ifdef CATALYST_SCOREP
    ! add this region to the list of plugin variables
    CALL cat_sco_pipeline_me()
#endif

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

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 or not be nested.

Finally, the user needs to add a small piece of code into the Catalyst adapter's source, in order for the plugin-generated variables to be pipelined (together with the traditional simulation variables), as shown in Fig. 6. It contains two vectors



```
#include <mpi.h>

int main(int argc, char **argv)
{
    int rtrn = MPI_Init(& argc, & argv);

    #ifdef USE_CATALYST
        initialize_coprocessor();
    #endif

    // STARTING PROCEDURES...

    unsigned int num_time_steps = 999;






    // MAIN SOLVER LOOP
    for (unsigned int time_step = 0; time_step < num_time_steps; time_step++)
    {
        // COMPUTATIONS...

        #ifdef USE_CATALYST
            run_coprocessor(time_step, time_value);
        #endif





    }






    // ENDING PROCEDURES...

    #ifdef USE_CATALYST
        finalize_coprocessor();
    #endif

    rtrn = MPI_Finalize();

    return 0;
}
```

```
#include <mpi.h>

int main(int argc, char **argv)
{
    int rtrn = MPI_Init(& argc, & argv);

    #ifdef USE_CATALYST
        initialize_coprocessor();
    #endif

    // STARTING PROCEDURES...

    unsigned int num_time_steps = 999;

    #ifdef CATALYST_SCOREP
        // tell the plugin that the time-step loop is about to start
        cat_sco_initialize_();
    #endif

    // MAIN SOLVER LOOP
    for (unsigned 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 prepare them for the next time step
            cat_sco_clean_(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

    rtrn = MPI_Finalize();

    return 0;
}
```

Fig. 5. Illustrative example of further changes needed in the code due to the plugin.

because for each selected region inside the simulation's code, the plugin will generate two variables (which correspond to the two basic measurements made by Score-P; see above).

5 Early Evaluation

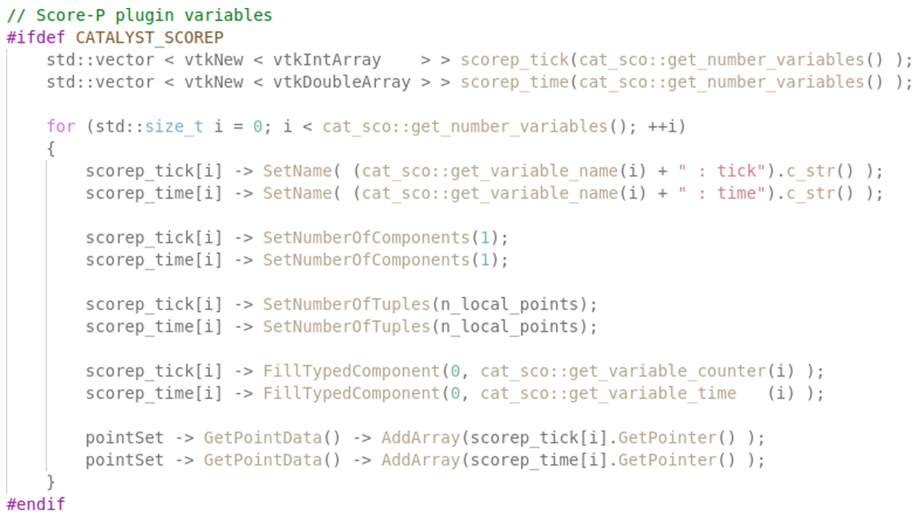
5.1 Settings

Hydra is Rolls-Royce's in-house CFD code [12]. Figure 9 shows the test case selected for this paper: it represents a generic Q3D idealized model for a turbine stage. Preliminary analyses with Score-P $\rightarrow$ Cube revealed two code functions to be especially time-consuming: *iflux_edge* and *vflux_edge* (both mesh-related); they were selected for pipelining.

All simulations were done using an entire node in Dresden University's HPC cluster (Taurus), with 12 ranks (i.e. pure MPI, no OpenMP), one per core, each with the entire core memory (3875 MB) available. One full engine's shaft rotation was simulated, comprised of 100 time-steps (i.e. one per 3,6°), each internally converged through 40 iteration steps. Catalyst was generating post-mortem output files every fifth time-step (i.e. every 18°), what led to 20 "stage pictures" by the end of the simulation. Finally, version 4.0 of Score-P was used in association with release 2018a of Intel® compilers.

5.2 Results

Hydra supports multiple mesh partition algorithms, selectable at run-time. We compared them with our newly proposed approach. Figure 7 shows the time spent



```
// Score-P plugin variables
#ifdef CATALYST_SCOREP
    std::vector < vtkNew < vtkIntArray    > > scorep_tick(cat_sco::get_number_variables() );
    std::vector < vtkNew < vtkDoubleArray > > scorep_time(cat_sco::get_number_variables() );

    for (std::size_t i = 0; i < cat_sco::get_number_variables(); ++i)
    {
        scorep_tick[i] -> SetName( (cat_sco::get_variable_name(i) + " : tick").c_str() );
        scorep_time[i] -> SetName( (cat_sco::get_variable_name(i) + " : time").c_str() );

        scorep_tick[i] -> SetNumberOfComponents(1);
        scorep_time[i] -> SetNumberOfComponents(1);

        scorep_tick[i] -> SetNumberOfTuples(n_local_points);
        scorep_time[i] -> SetNumberOfTuples(n_local_points);

        scorep_tick[i] -> FillTypedComponent(0, cat_sco::get_variable_counter(i) );
        scorep_time[i] -> FillTypedComponent(0, cat_sco::get_variable_time   (i) );

        pointSet -> GetPointData() -> AddArray(scorep_tick[i].GetPointer() );
        pointSet -> GetPointData() -> AddArray(scorep_time[i].GetPointer() );
    }
#endif
```

Fig. 6. Addition needed in the Catalyst adapter's code due to the plugin.

inside the two chosen functions in two different grid partitions: the upper images refer to geometric mesh partitioning and the lower ones were produced using ParMETIS;[15] the left-hand side pictures refer to function *iflux_edge*, whereas the right-hand side to *vflux_edge*. Here only one time-step is represented, but – as opposed to the traditional way of visualizing profiling results (which aggregate multiple time-steps into one single measurement) – in ParaView it is possible to see each time-step individually and even play them (as frames of a video). Finally, the minimum and maximum thresholds in each of the four pictures' scales are adjusted to comprise all time-steps.

The analysis of the results reveals that, when compared against the geometric mesh partition, using ParMETIS brings slight benefits to the selected functions' performance: the overall maximum execution time (per time-step) drops in both of them, the overall minimum in *vflux_edge*; and the max/min ratio of the execution time (per time-step) for both of them is also decreased.

Playing the saved time-steps in ParaView reveals a trend in all four layouts: the slowest/fastest rank to execute each function is always the same. This means there are still load imbalances when using ParMETIS; otherwise, the slowest/fastest rank should randomly change each time-step (due to stochastic phenomena at hardware-level during run-time). See the respective video.

Figure 8 compares the results when profiling is activated (below) or not (above). They let clear that doing simultaneous code profiling significantly slows each region's execution time, but the max/min ratio remains roughly the same:

[15] An open source "MPI-based parallel library that implements a variety of algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices": http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

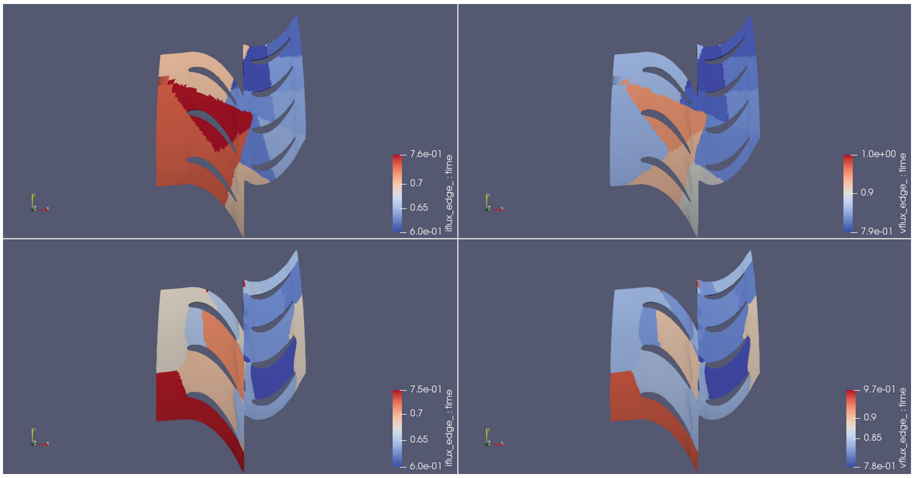


Fig. 7. Comparison between two code functions in two mesh partitions.

– from $0.57/0.46 \approx 1.24$ to $0.75/0.60 = 1.25$ in *iflux_edge*;
– from $0.85/0.69 \approx 1.23$ to $0.97/0.78 \approx 1.24$ in *vflux_edge*.

This means the overhead associated with each feature (Score-P's profiling and/or the plugin) is linear, hence the results are valid from a comparative point of view. Indeed, playing the respective video reveals the same trend (slowest/fastest rank) as in the previous comparison.

Finally, the generated performance variables are accessible also live (interactively) in ParaView. In Fig. 9, notice the "catalyst" icon on the *Pipeline Browser*, as well as the presence of the selected code regions' measurements among the *Data Arrays*.

5.3 Overhead

Table 1 analyses the impact of the proposed plugin on the code's performance. ParMETIS was used for mesh partitioning.

Memory. The "memory" row in Table 1 refers to the *peak* memory consumption per rank, reached somewhen during the simulation. From the numbers it is clear that the memory overhead introduced by Score-P is negligible (less than 10%); and that the memory overhead introduced by the plugin is also negligible. It may even require less memory than doing the traditional profiling (depending upon the number of code regions being pipelined) and, in our case, was below the statistical margin of oscillation (given *profiling + plugin* took less memory than *profiling only*). Indeed, in order to pipeline the two code functions shown above, it was not necessary to increase the default amount of memory (16 MB) that Score-P reserves for itself.

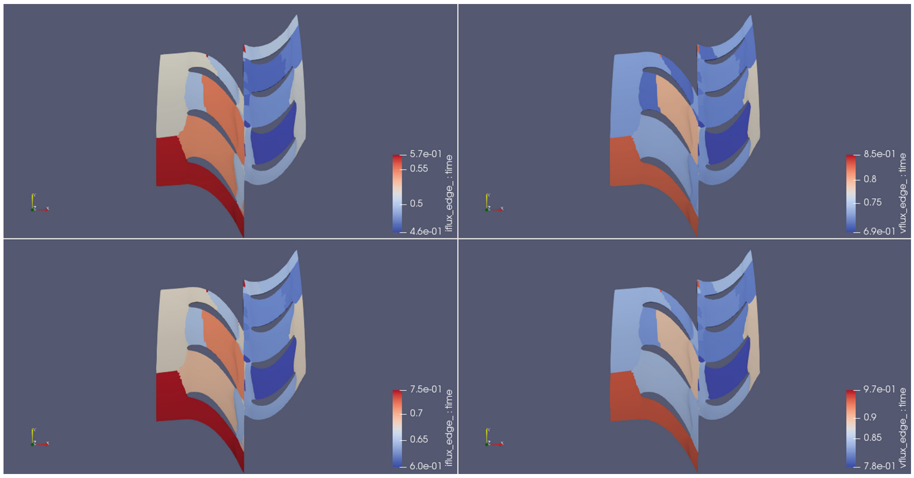
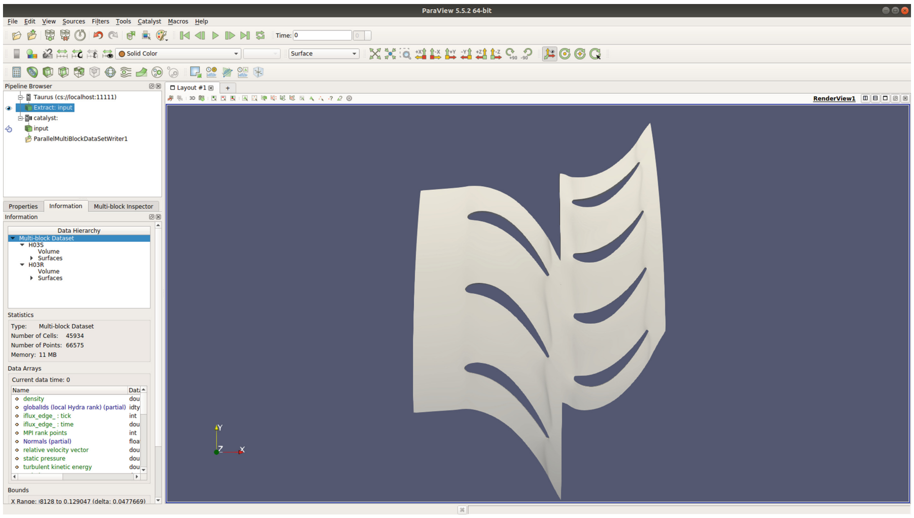


Fig. 8. Comparison between two code functions when profiling is activated (below) or not (above).

Table 1. Analysis of time and memory overhead of the plugin.

	Profiling + plugin	Profiling only	Plugin only	Without Score-P
Memory (kB)	290432 (6%)	294120 (7%)	284440 (4%)	273956 (-)
Run-time lightweight	4 m 18 s (8%)	4 m 10 s (5%)	4 m 06 s (3%)	3 m 58 s (-)
Run-time heavyweight	8 m 30 s (114%)	7 m 05 s (79%)	6 m 48 s (71%)	3 m 58 s (-)

Time. The run-time overhead is more critical and is shown in Table 1 with two cases. The light-weight instrumentation case shows the overhead of the presented approach with a sensible set of instrumented subroutines as it may have been achieved with carefully selecting the most interesting subroutines for the performance analysis process. This is the suggested way according to the Score-P documentation. In that case, the plugin produces a run-time overhead of 3%. This is less than Score-P in profiling mode with 5%. If both are used together, the overhead adds up. This is a sensible overhead and suitable for practical performance analysis. The second case with heavy-weight instrumentation reflects the worst-case scenario where some short subroutines are called very frequently (several billion times in this example). In that case, the overhead can dominate the entire run-time and the performance analysis insights are not reflecting the pristine parallel performance behavior. However, this scenario in Table 1 shows that our plugin behaves similar to Score-P in profiling mode; actually even slightly better with 71% overhead compared to 79%.



Pipeline Browser
- Taurus (cs://localhost:11111)
  - Extract: input
- catalyst:
  - input
  - ParallelMultiBlockDataSetWriter1

Information

Data Hierarchy
- Multi-block Dataset
  - H03S
    - Volume
    - Surfaces
  - H03R
    - Volume
    - Surfaces

Statistics

Type: Multi-block Dataset
Number of Cells: 45934
Number of Points: 66575
Memory: 11 MB

Data Arrays

Current data time: 0

Name	Data
density	dou
globalIds (local Hydra rank) (partial)	idty
iflux_edge_ : tick	int
iflux_edge_ : time	dou
MPI rank points	int
Normals (partial)	floa
relative velocity vector	dou
static pressure	dou
turbulent kinetic energy	dou

Bounds

X Range: 8128 to 0.129047 (delta: 0.0477669)

Fig. 9. Geometry used in the simulations.

6 Conclusions and Future Work

Visualization techniques are usually not the specialization field of researches working with code performance: it is more reasonable to take advantage of the currently available graphic programs (like ParaView) than attempting – from scratch – to equip the existing profiling tools with their own GUIs. In this threshold, the developed plugin adds to the currently available spectrum of performance optimization resources the capacity to:

– match performance-related measurements against the simulation's mesh, what makes the impact of grid partition algorithms on code performance easier to investigate;
– analyze performance-related measurements on a time-step basis, what makes the load imbalances (and their eventual relationship to mesh size/flow physics) easier to diagnose.

We plan to extend this work in multiple directions:

More Extensive Evaluation Cases. To run the plugin in bigger test cases, as the difficulty in matching each parallel region's id number with its respective grid part (hence the benefit of matching performance data back to the simulation's 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.

Improve and Further Integrate Tool's Runtime Components. To automatize the selection of code regions to be pipelined, what currently needs to be manually done by the user at compile time (as shown in Sect. 4).

Develop New Visualization Schemes for Performance Data. To take advantage of the multiple filters available in ParaView for the benefit of the performance optimization branch, e.g. by recreating in it the statistical analysis – display of *average* and *standard deviation* between the threads/ranks' measurements – already available in other tools.

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

References

1. Adhianto, L., et al.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs. Concurrency Comput. Pract. Exp. **22**(6), 685–701 (2010). https://doi.org/10.1002/cpe.1553
2. Ahrens, J., Geveci, B., Law, C.: ParaView: An End-User Tool for Large Data Visualization. The Visualization Handbook, vol. 717. Academic Press, Cambridge (2005)
3. Ayachit, U., 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, ISAV 2015. pp. 25–29. ACM, New York (2015). https://doi.org/10.1145/2828612.2828624
4. Badia, R.M., Labarta, J., Gimenez, J., Escale, F.: DIMEMAS: predicting MPI applications behavior in grid environments. In: Workshop on Grid Applications and Programming Tools (GGF8), vol. 86, pp. 52–62 (2003)
5. Bauer, A.C., et al.: In situ methods, infrastructures, and applications on high performance computing platforms. Comput. Graph. Forum **35**(3), 577–597 (2016). https://doi.org/10.1111/cgf.12930
6. Bell, R., Malony, A.D., Shende, S.: *ParaProf*: a portable, extensible, and scalable tool for parallel performance profile analysis. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 17–26. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45209-6_7
7. Benedict, S., Petkov, V., Gerndt, M.: PERISCOPE: an online-based distributed performance analysis tool. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009, pp. 1–16. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11261-4_1

8. Chan, A., Gropp, W., Lusk, E.: An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. Sci. Program. **16**, 155–165 (2008). https://doi.org/10.3233/SPR-2008-0252. https://www.hindawi.com/journals/sp/2008/749874/cta/
9. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurrency Comput. Pract. Exp. **22**(6), 702–719 (2010). https://doi.org/10.1002/cpe.1556
10. Knüpfer, A., et al.: The Vampir performance analysis tool-set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68564-7_9
11. Knüpfer, 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., Resch, M.M. (eds.) Tools for High Performance Computing 2011, pp. 79–91. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31476-6_7
12. Lapworth, L.: HYDRA-CFD: a framework for collaborative CFD development. In: International Conference on Scientific and Engineering Computation (IC-SEC), Singapore, June, vol. 30 (2004). https://www.researchgate.net/publication/316171819_HYDRA-CFD_A_Framework_for_Collaborative_CFD_Development
13. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: a tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: Transputer and OCCAM Developments, vol. 44, pp. 17–31. IOS Press (1995). https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.1277&rep=rep1&type=pdf
14. Saviankou, P., Knobloch, M., Visser, A., Mohr, B.: Cube v4: from performance report explorer to performance analysis tool. Procedia Comput. Sci. **51**, 1343–1352 (2015). https://doi.org/10.1016/j.procs.2015.05.320. International Conference On Computational Science, ICCS 2015
15. Schöne, R., Tschüter, R., Ilsche, T., Schuchart, J., Hackenberg, D., Nagel, W.E.: Extending the functionality of Score-P through plugins: interfaces and use cases. In: Niethammer, C., et al. (eds.) Tools for High Performance Computing 2016, pp. 59–82. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56702-0_4
16. Schroeder, W.J., Martin, K.M., Lorensen, W.E.: The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In: Proceedings of Seventh Annual IEEE Visualization 1996, San Francisco, CA, USA, pp. 93–100. IEEE, October 1996. https://doi.org/10.1109/VISUAL.1996.567752
17. Schuchart, J., et al.: The READEX formalism for automatic tuning for energy efficiency. Computing **99**(8), 727–745 (2017). https://doi.org/10.1007/s00607-016-0532-7
18. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006). https://doi.org/10.1177/1094342006064482