Further enhancing the in situ visualization of performance data in parallel CFD applications

Rigel F. C. Alves Corresp., 1 , **Andreas Knüpfer** 1

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

Corresponding Author: Rigel F. C. Alves
Email address: rigel.alves@tu-dresden.de

This paper continues the work initiated by the authors on the feasibility of using *ParaView* as visualization software for the analysis of parallel Computational Fluid Dynamics (CFD) codes' performance. Current performance tools have limited capacity of displaying their data on top of three-dimensional, framed (i.e. time-stepped) representations of the cluster's topology. In our first paper, a plugin for the open-source performance tool *Score-P* was 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 were any other flow-related variable. Our second paper added to such plugin the capacity to (also) map communication data (messages exchanged between MPI ranks) to the simulation's geometry. So far the tool was limited to codes which already have the in situ adapter; but in this paper, we will take the performance data and display it -- also in codes without in situ -- on a three-dimensional representation of the hardware resources being used by the simulation. Testing is done with the *Multi-Grid* and *Block Tri-diagonal* NPBs, as well as Rolls-Royce's CFD code, *Hydra*. The benefits and overhead of the plugin's new functionalities are discussed.

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

Rigel F. C. Alves[1] **and Andreas Knüpfer**[2]

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

Corresponding author:
Rigel F. C. Alves[1]

Email address: rigel.alves@tu-dresden.de

ABSTRACT

This paper continues the work initiated by the authors on the feasibility of using *ParaView* as visualization software for the analysis of parallel Computational Fluid Dynamics (CFD) codes' performance. Current performance tools have limited capacity of displaying their data on top of three-dimensional, framed (i.e. time-stepped) representations of the cluster's topology. In our first paper, a plugin for the open-source performance tool *Score-P* was introduced, which intercepts an arbitrary number of manually selected code *regions* (mostly functions) and sends 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. Our second paper added to such plugin the capacity to (also) map communication data (messages exchanged between MPI ranks) to the simulation's geometry. So far the tool was limited to codes which already have the in situ adapter; but in this paper, we will take the performance data and display it – also in codes without in situ – on a three-dimensional representation of the hardware resources being used by the simulation. Testing is done with the *Multi-Grid* and *Block Tri-diagonal* NPBs, as well as Rolls-Royce's CFD code, *Hydra*. The benefits and overhead of the plugin's new functionalities are discussed.

INTRODUCTION

Computers have become crucial in solving engineering problems. However, standard computers do not have enough power to run more complex simulations (such as those involved in modern engineering problems, like designing an aircraft) on their own. They require parallelized simulation (for instance of the air flowing through the airplane's engine) to be run in High Performance Computing (HPC) hardware. Such infrastructures are expensive, as well as time and energy consuming. It is thus imperative that the application has its parallel performance tuned for maximum productivity.

There are several tools for analyzing the performance of parallel applications. An example is *Score-P*[1] (Knüpfer et al., 2012), which is developed in partnership with the *Centre for Information Services and HPC* (ZIH) of the Technische Universität Dresden. It allows the user to instrument the simulation's code and monitor its execution, and can easily be turned on or off at compile time. When applied to a source code, the simulation will not only produce its native outputs at the end, but also the performance data. Figure 1 below illustrates the idea.

However, the tools currently available to *visualize* the performance data (generated by software like Score-P) lag in important features, like three-dimensionality, time-step association (i.e. frame playing), color encoding, manipulability of the generated views etc.

[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" [tool's website].

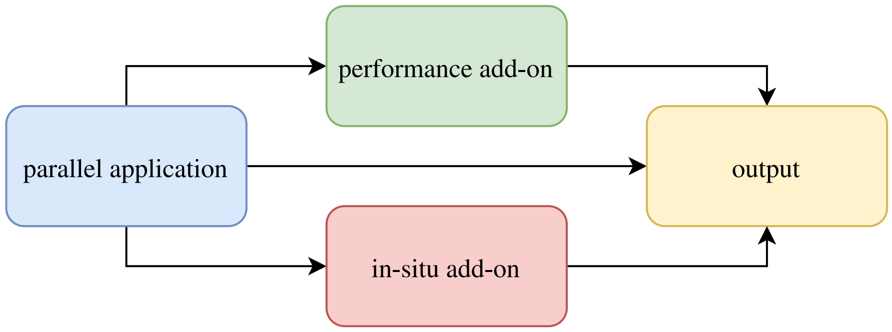

Figure 1. Schematic of software components for parallel applications

As a different 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*[2] (Ayachit et al., 2015). They also work as an optional layer to the original code and can be activated upon request, by means of preprocessor directives at compilation stage. The simulation will then produce its native outputs, if any, plus the *coprocessor*'s (a piece of code responsible for permitting the original application to interact with the in situ methods) ones, in separate files. This is illustrated in the bottom part of Figure 1. These tools have been developed by visualization specialists for a long time and feature sophisticated visual resources.

In this sense, why not apply such in situ tools (which enable data extraction from the simulation by separate side channels, in the same way as performance instrumenters) to the performance analysis of parallel applications, thus filling the blank left by the lack of visual resources of the performance tools?

This work is the third in a series of our investigations on the feasibility of merging the aforementioned approaches. First, by unifying the coinciding characteristics of both types of tools, insofar as they augment a parallel application with additional features (which are not required for the application to work). Second, by using the advanced functionalities of specialized visualization software for the goal of performance analysis. Figure 2 illustrates the idea.

In our first paper (Alves and Knüpfer, 2019), 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 our second paper (Alves and Knüpfer, 2020), we added to such mapping communication data (messages exchanged between MPI ranks). Henceforth this feature shall be called *geometry mode*.

Following feedback we have received since, we thought about how our approach could be used to assist with the performance optimization of codes without an in situ adapter. What happens if you move such adapter inside our tool? This corresponds to flipping the positions of the *performance* and the *in situ* add-ons on Figure 2; i.e. so far we were doing performance analysis inside in situ, now we will do in situ inside performance. In this paper, we present the result of such investigation: a new feature in our tool, called *topology mode* – the capacity of matching the performance data to a three-dimensional representation of the cluster's architecture.

There are two approaches to HPC performance analysis. One uses *performance profiles* which contain congregated data about the parallel execution behavior. Score-P produces them in the Cube4 format, to be visualized with *Cube*[3]. The other uses *event traces* collecting individual run-time events with precise timings and properties. Score-P produces them in the OTF2 format, to be visualized with *Vampir*[4]. The outputs of our tool are somehow a mixture of both: aggregated data, but by time step.

The presented solution is not intended for permanent integration into the source code of the target application. Instead it should be applied on demand only with little extra effort. This is solved in accordance with the typical approaches of *parallel performance analysis tools* on the one hand and *in situ processing toolkits* on the other hand. As evaluation cases, the *Multi-Grid* and *Block Tri-diagonal NAS Parallel Benchmarks* (NPB) (Frumkin et al., 1998) will be used, together with Rolls-Royce's in-house CFD code, *Hydra* (Lapworth, 2004).

[2]https://www.paraview.org/in-situ/
[3]http://www.scalasca.org/software/cube-4.x/download.html
[4]https://vampir.eu/

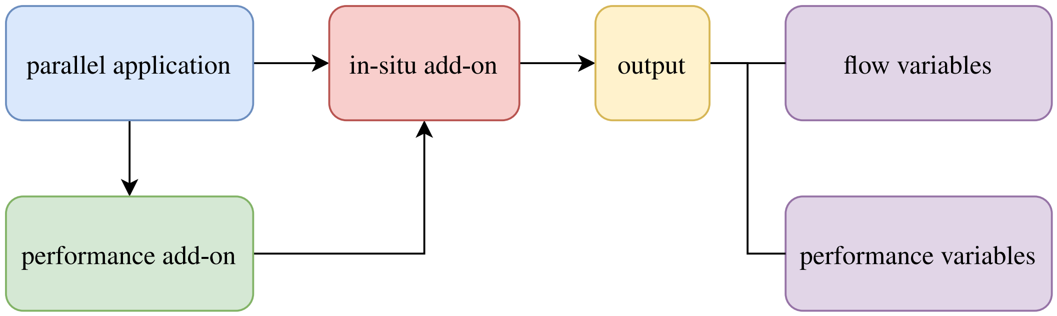

Figure 2. Schematic of the software components for a combined add-on

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 computing architecture's topology and the limitations of their results. In section 2 we present the methodology of our approach, which is then evaluated in the test-cases in section 3. Finally, section 4 discusses the overhead associated with using our tool. We then conclude the article with a summary.

1 RELATED WORK

In order to support the developer of parallel codes in his optimization tasks, many software tools have been developed. For an extensive 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*, including 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*[5] of the *Virtual Institute – High Productivity Supercomputing* (VI-HPS). Only one of them matches the performance data to the cluster's topology: *ParaProf* (Bell et al., 2003), whose results can be seen in the tool's website[6]. The outputs are indeed three-dimensional, but their graphical quality is low, as one could expect from a tool which tries to recreate the visualization environment from scratch. The same hurdle can be found on the works of Isaacs et al. (2012) and Schnorr et al. (2010), which also attempt to create a whole new three-dimensional viewing tool (just for the sake of performance analysis). Finally, Theisen et al. (2014) combined multiple axes onto two-dimensional views: the generated visualizations are undeniably rich, but without true three-dimensionality, the multiplicity of two-dimensional planes overlapping each other can quickly become cumbersome and preclude the understanding of the results.

On the other hand, when it comes to display messages exchanged between MPI ranks during the simulation, *Vampir* is the current state-of-the-art tool on the field, but it is still unable to generate three-dimensional views. This impacts e.g. on the capacity to distinguish between messages coming from ranks running within the same compute node from those coming from ranks running in other compute nodes. Also, Vampir is not able to apply a color scale to the communication lines. Finally, it has no knowledge of the simulation's time-step, whereas this is the code execution delimiter the developers of CFD codes are naturally used to deal with. Isaacs et al. (2014) got close to it, by clustering event traces according to the self-developed idea of *logical time*, "inferred directly from happened-before relationships". This represents indeed an improvement when compared with not using any sorting, but it is not yet the time-step loop as known by the programmer of a CFD code. Alternatively, it is possible to

[5]`https://www.vi-hps.org/cms/upload/material/general/ToolsGuide.pdf`
[6]`https://www.cs.uoregon.edu/research/tau/images/KG_48r_topo_alltoallv.png`

isolate the events pertaining to the time step by manually instrumenting the application code and inserting a region called e.g. "Iteration" (see section 2.1.1 below). Solórzano et al. (2021) and Miletto et al. (2021) have applied such method. We would like then to simplify this process and make it part of the tool's functioning itself.[7]

Finally, with regards to in situ methods, for a comprehensive study of the ones currently available, the reader is referred to the work of Bauer et al. (2016).

2 METHODOLOGY

This section presents what is necessary to implement our work.

2.1 Prerequisites

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

2.1.1 Performance Measurement

When applied to a source file's compilation, Score-P automatically inserts probes between each code "region"[8], which will at run-time measure a) the *number of times* that region was executed and b) the total *time* spent in those executions, by each process (MPI rank) within the simulation. It is applied by simply prepending the word `scorep` into the compilation command, e.g.: `scorep [Score-P's options] mpicc foo.c`. It is possible to suppress regions from the instrumentation (e.g. to keep the associated overhead low), by adding the flag `--nocompiler` to the command above. In this scenario, Score-P sees only user-defined regions (if any) and MPI-related functions, whose detection can be easily (de)activated at run-time, by means of an environment variable: `export SCOREP_MPI_ENABLE_GROUPS=[comma-separated list]`. Its default value is set to catch all of them. If left blank, instrumentation of MPI routines will be turned off.

Finally, the tool is also equipped with an API, which permits the user to increase its capabilities through plugins (Schöne et al., 2017). The combined solution proposed by this paper takes actually 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 created, which is responsible for exposing the native data structures (grid 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 Figure 3. Once implemented, the adapter allows the generation of post-mortem files (by means of the *VTK*[9] library) and/or the live visualization of the simulation, both through *ParaView*[10].

2.2 Combining both Tools

In our previous works (Alves and Knüpfer, 2019; Alves and Knüpfer, 2020), a Score-P plugin has been developed, which allows performance measurements for an arbitrary number of manually selected code regions and communication data (i.e. messages exchanged between MPI ranks) to be mapped to the simulation's original geometry, by means of its Catalyst adapter (a feature now called *geometry mode*). In this paper, we are extending our software to map those measurements to a three-dimensional representation of the cluster's topology, by means of the plugin's own Catalyst adapter (a new feature named *topology mode*). The plugin must be turned on 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 needs three function calls (*initialize*, *run* and *finalize*) to be introduced in the source code, illustrated in violet at Figure 3. However, if the tool is intended to be used exclusively in *topology mode*, the blue calls shown at Figure 3 are not needed, given in this mode the plugin depends only on its own Catalyst adapter (i.e. the simulation code does not need to have any reference to VTK whatsoever).

[7]The correspondent drawback is that the tool will not be suitable for detecting variations inside the course of one time step. For such analyses, the user is referred to the currently available tools, like Vampir.

[8]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 given a name. It could be used e.g. to gather all instructions pertaining to the main solver (time-step) loop.

[9]`https://www.vtk.org/`

[10]`https://www.paraview.org/`

```
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)

```
#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

Finally, a call must be inserted before each function to be pipelined, as illustrated in Figure 4 below. This layout ensures that the desired region will be captured when executed at that specific moment and not in others (if the same routine is called multiple times – with distinct inputs – throughout the code, as it is common for CFD simulations). The selected functions may even be nested. This is not needed when tracking communications between ranks, as the instrumentation of MPI regions is made independently at run-time (see section 2.1.1 above).

3 EVALUATION

This section presents how our work is going to be evaluated.

3.1 Settings

Three test-cases will be used to demonstrate the new functionality of the plugin: two well-known benchmarks and an industry-grade CFD Code. All simulations were done in Dresden University's HPC cluster (Taurus), whose nodes are interconnected through Infiniband. Everything was built / tested with release 2018a of Intel® compilers in association with versions 6.0 of Score-P and 5.7.0 of ParaView.

3.1.1 Benchmarks

The *NAS Parallel Benchmarks* (NPB) (Frumkin et al., 1998) "are a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications". Here one of each is used: the *Multi-Grid* (MG) and the *Block Tri-diagonal* (BT) respectively (version 3.4). Both were run in a Class *D* layout by four entire *Sandy Bridge* nodes, each with 16 ranks (i.e. pure MPI, no OpenMP), one per core and with the full core memory (1875 MB) available. Their grids consist of a parallelepiped with the same number of points in each cartesian direction. Finally, both are sort of "steady-state" cases (i.e. the *time*-step is equivalent to an *iteration*-step).

In order for the simulations to last at least 30 minutes,[11] MG was run for 3000 iterations (each comprised of 9 multigrid levels), whereas BT for 1000. The plugin generated VTK output files every 100 iterations for MG (i.e. 30 "stage pictures" by the end of the simulation, 50 MB of data in total), every 50 iterations for BT (20 frames in the end, same amount of data), measuring the solver loop's central routine (*mg3P* and *adi* respectively) in each case.

3.1.2 Industrial CFD Code

Hydra is Rolls-Royce's in-house CFD code (Lapworth, 2004), based on a preconditioned time marching of the Reynolds-averaged Navier-Stokes (RANS) equations. They are discretized in space using an edge-based, second-order finite volume scheme with an explicit, multistage Runge-Kutta scheme as a steady time marching approach. Steady-state convergence is improved by multigrid and local time-stepping acceleration techniques (Khanal et al., 2013). Figure 5 shows the test case selected for this paper: it represents a simplified (single cell thickness), 360° testing mesh of two turbine stages in an aircraft engine, discretized through approximately 1 million points. Unsteady RANS calculations have been made with time-accurate, second-order dual time-stepping. Turbulence modelling was based on standard 2-equation

[11] Less than that would make the relative (percentage) statistical oscillation of the run time too big for valid comparisons (see section 4 below).

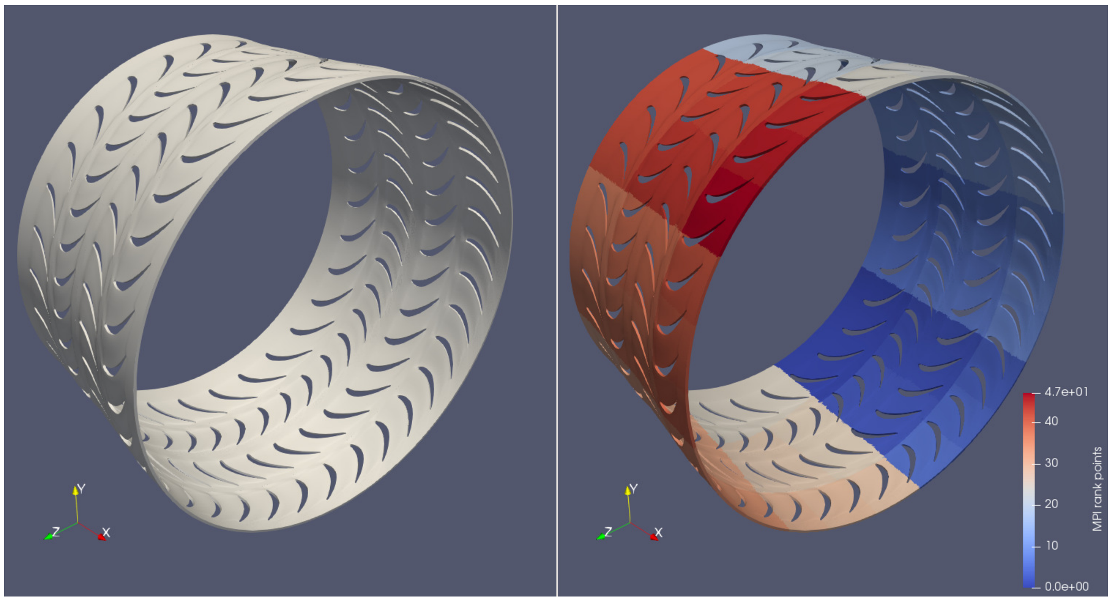

Figure 5. Geometry used in the industrial CFD code simulations (left) and its partitioning among processes for parallel execution (right)

closures. Preliminary analyses with Score-P and Cube revealed two code functions to be especially time-consuming: *iflux_edge* and *vflux_edge*; they were selected for pipelining.

Here the simulations were done using two entire *Haswell* nodes, each with 24 ranks (again pure MPI), one per core and with the entire core memory (2583 MB) available. Figure 5 shows the domain's partitioning among the processes. The shape of the grid, together with the rotating nature of two of its four blade rings (the rotors), anticipates that the communication patterns here are expected to be extremely more complex than in the benchmarks.

One full engine's shaft rotation was simulated, comprised of 200 time-steps (i.e. one per 1,8°), each internally converged through 40 iteration steps. The plugin was generating post-mortem files every 20th time-step (i.e. every 36°), what led to 10 stage pictures (12 MB of data) by the end of the simulation.

3.2 Results

The second part of this section presents the results of applying our work on the selected test-cases. The benchmarks will be used more to illustrate how the tool works, whereas a true performance optimization task will be executed with the industrial CFD code.

3.2.1 Benchmarks

Figure 6 shows the plugin outputs for an arbitrary time-step in the MG benchmark. The *hardware* information (i.e. in which core, socket etc. each rank is running) is plotted on constant z planes; the *network* information (i.e. switches that need to be traversed in order for inter-node communications to be performed), on its turn, is shown on the $x = 0$ plane.

Score-P's measurements, as well as the rank id number, are shown just below the *processing unit (PU)* where that rank is running, ordered from left to right (in the x direction) within one node, then from back to front (in the z direction) between nodes. Finally, the MPI communication made in the displayed time-step is represented through the lines connecting different rank ids' cells.

Here, notice how each compute node allocated to the job becomes a plane in ParaView. They are ordered by their id numbers (see the right-hand side of Figure 6) and separated by a fixed length (adjustable at run-time through the plugin's input file). Apart from the node id, it is also possible to color the planes by the *topology type*, i.e. if the cell refers to a socket, a L3 cache, a processing unit etc., as done on the left side of the figure.

Only the resources being used by the job are shown in ParaView, as to minimize the plugin's overhead

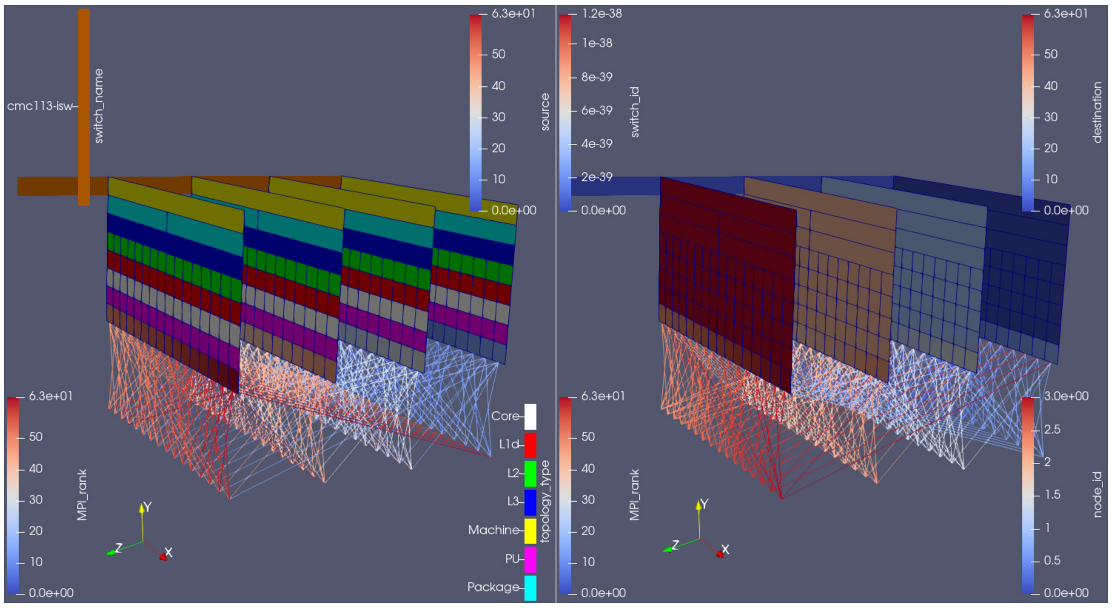

Figure 6. Plugin outputs for an arbitrary time-step at the MG benchmark, visualized from the same camera angle, but with different parameters on each side

and in view that drawing the entire cluster would not help the user to understand the code's behaviour [12]. This means that, between any pair of planes in Figure 6 there might be other compute nodes (by order of id number) in the cluster infrastructure; but, if that is the case, none of its cores are participating in the current simulation. The inter-node distance in ParaView will be bigger, however, if the user activated the drawing of network topology information and the compute nodes involved in the simulation happen to be located in different network *islands*, as shown on Figure 7. This is indeed intuitive, as messages exchanged between nodes under different switches will need to *travel longer* in order to be delivered (when compared to those exchanged between nodes under the same switch).

Taurus uses *Slurm*[13], which carefully allocates the MPI ranks by order of compute node id (i.e. the node with lower id will receive the first processes, whereas the node with higher id will receive the last processes). It also attempts to place those ranks as close as possible to one another (both from an *intra* and *inter* node perspectives), as to minimize their communications' latency. But just to illustrate the plugin's potential, Figure 8 shows the results when forcing the scheduler to use at least a certain amount of nodes for the job. Notice how only the sockets (the cyan rectangles in the figure) where there are allocated cores are drawn in the visualization; the same applies to the L3 cache (the blue rectangles). Also, notice how the switches are positioned in a way that looks like a linkage between the machines (the yellow rectangles in the figure) they connect. This is intentional (it makes the visualization intuitive).

With regards to messages sent between ranks, in order to facilitate the understanding of the communication behavior, the source / destination data is also encoded in the position of the lines themselves: they start from the bottom of the sending rank and go downwards toward the receiving one. This way, it is possible to distinguish – and simultaneously visualize – messages sent from A to B and from B to A. In Figure 9, notice how the manipulation of the camera angle (an inherent feature of visualization software like ParaView) allows the user to immediately get useful insights about its code behaviour (e.g. the even nature of the communication channels in MG versus the cross-diagonal shape in BT).

In Figure 9, notice also how all ranks on both benchmarks talk either to receivers within the same node or the nodes immediately before / after. The big lines connecting the first and last nodes suggest some sort of periodic boundary condition inside the grid. This can be misleading: lines between cores in the first and last nodes will need to cross the entire visualization space, making it harder to understand. For

[12]Companies like Rolls-Royce usually purchase computational resources: they are not willing to buy the compute time of e.g. 16 nodes when they only need 4 for a specific simulation. In this sense, performance degradation due to nearby jobs (sharing the same network switch) is seen as "part of life".

[13]`https://slurm.schedmd.com/`

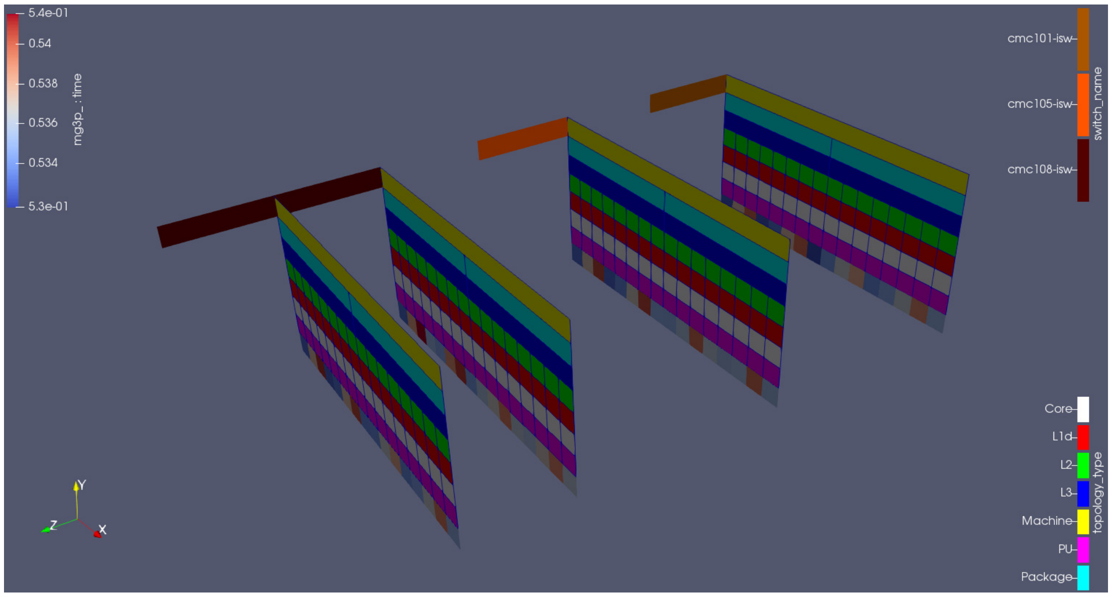

Figure 7. Plugin outputs for the MG benchmark. The leaf switch information is encoded both on the color (light brown, orange and dark brown) and on the position of the node planes (notice the extra gap when they do not belong to the same switch)

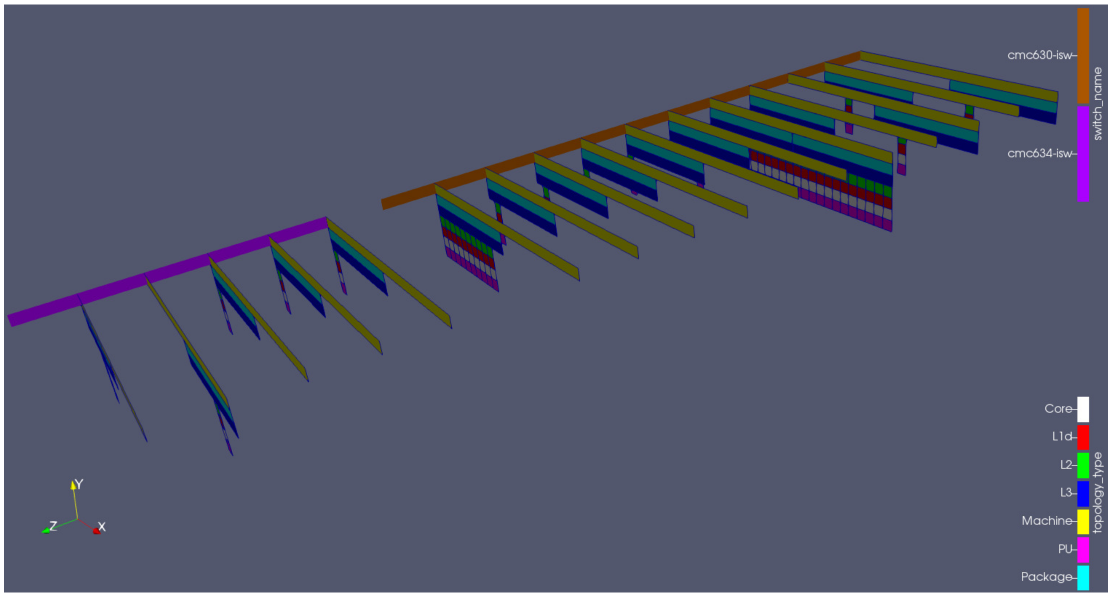

Figure 8. Poorly distributed ranks across compute nodes, for illustration purposes

that reason the plugin's runtime input file has an option to activate a periodic boundary condition tweak, whose outputs are visible in Figure 10. It shows the topology when using *Haswell* nodes, whose sockets have 4 more cores than *Sandy Bridge*. The communication lines are colored by destination rank of the messages; they refer to the MG benchmark. Notice how the periodic nature of this test-case's boundary conditions become clearer in the top picture: the big lines mentioned above are gone.

Finally, Slurm comes with a set of tools which will go through the cluster network (Infiniband, in our case) and automatically generate its connectivity information, saving it into a file. This file has been used as the network topology configuration file and is read by the plugin at run time. If it is not found, the drawing of the planes in ParaView will not take the switches into account.

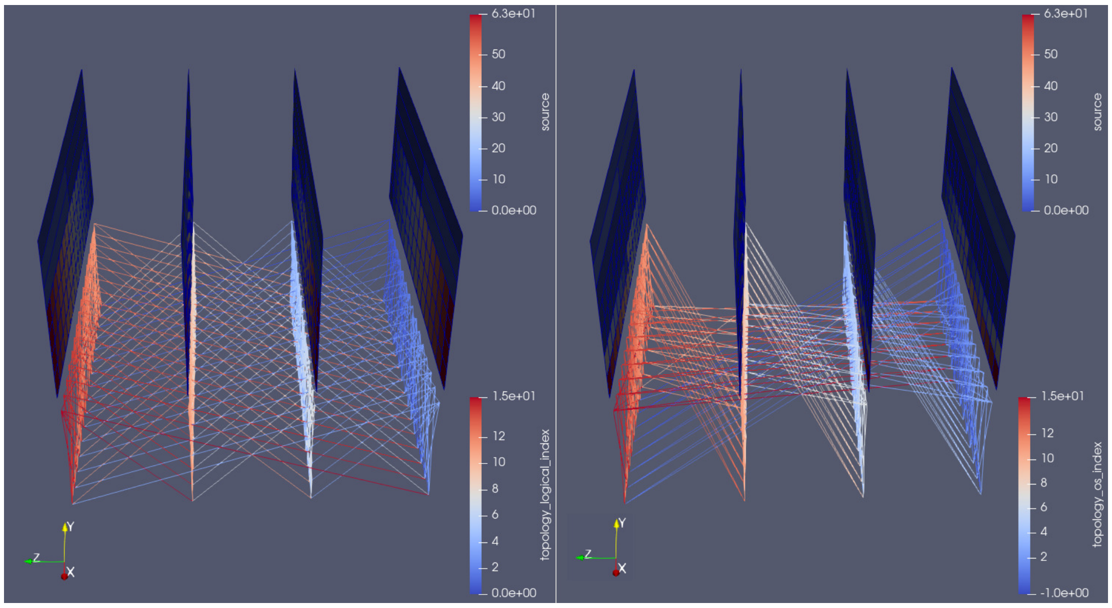

Figure 9. Side-by-side comparison of the communication pattern between the MG (left) and BT (right) benchmarks, at an arbitrary time-step, colored by source rank of messages

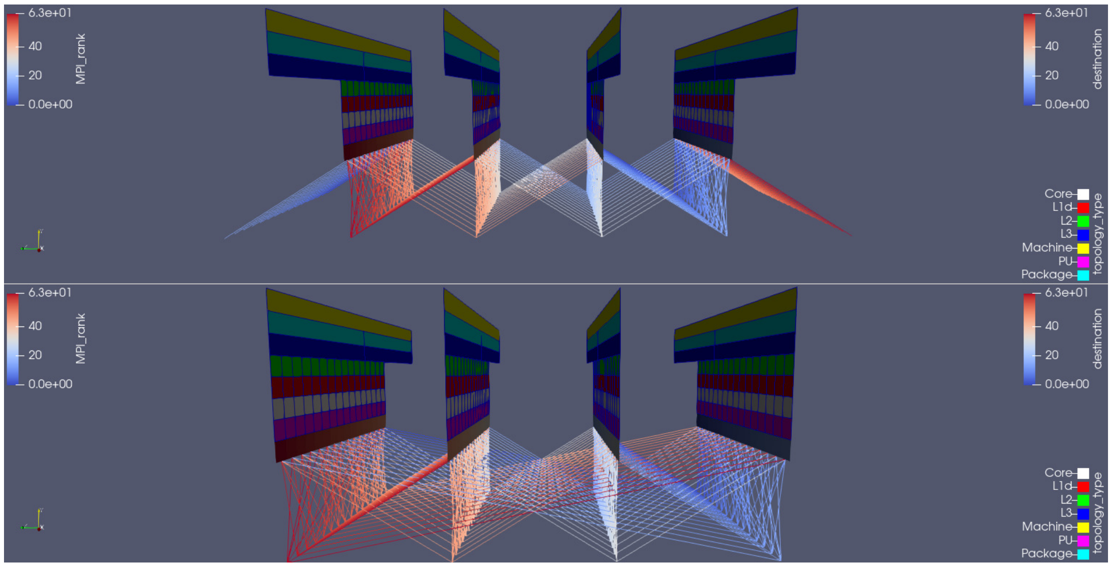

Figure 10. Side-by-side comparison of the communication pattern in the MG benchmark when using the periodic boundary condition feature (top) or not (bottom); communication lines are colored by destination rank of messages

3.2.2 Industrial CFD Code

Figure 11 shows the plugin outputs for an arbitrary time-step in the Hydra test-case, from two different camera angles; the communication lines are colored by number of MPI_Isend calls (left) and total amount of bytes sent on those calls (right) on that time-step. Notice on the right-hand side how many of the communication channels (the lines) did not properly transfer any data in that time-step (their color is blue, which from the scale is mapped to zero) and should therefore be removed. Also because the least used channels were used 1500 times within that time-step, as seen from the lower limit of the scale on the left part of the figure (counter_isend, which refers to the total amount of times MPI_Isend was called in the time-step shown). In other words, the plugin was able to estimate *how many* communication calls (per sender/receiver pair) could be spared per time-step in Rolls-Royce's code, and *where* (i.e. which ranks are

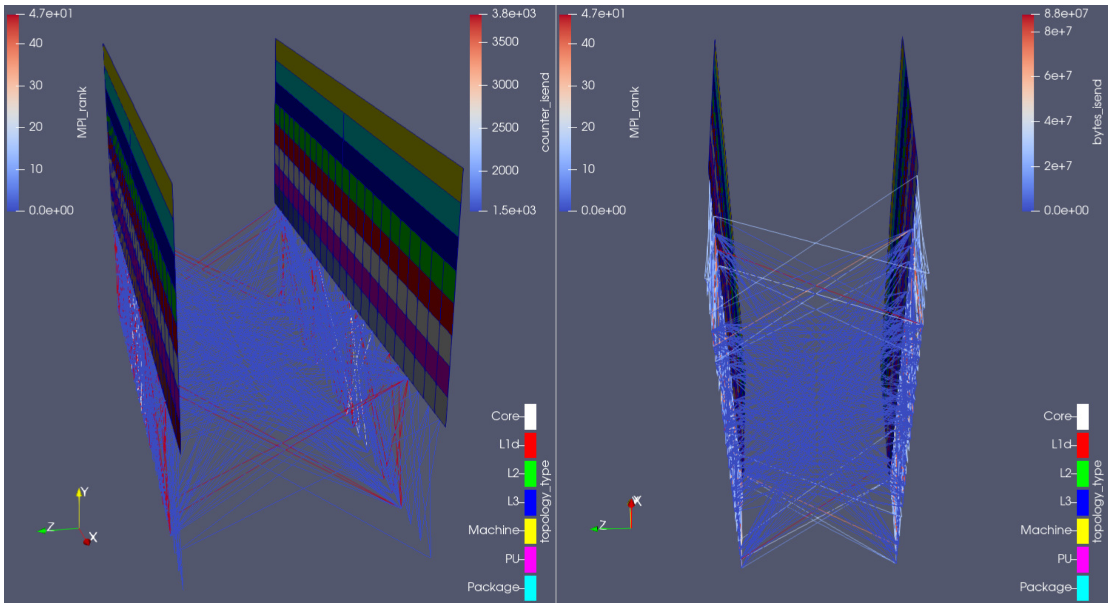

Figure 11. Visualization of the communication pattern in Hydra from two different camera angles, at an arbitrary time-step, colored by number of MPI_Isend calls (left) and total amount of bytes sent on those calls (right) on that time-step

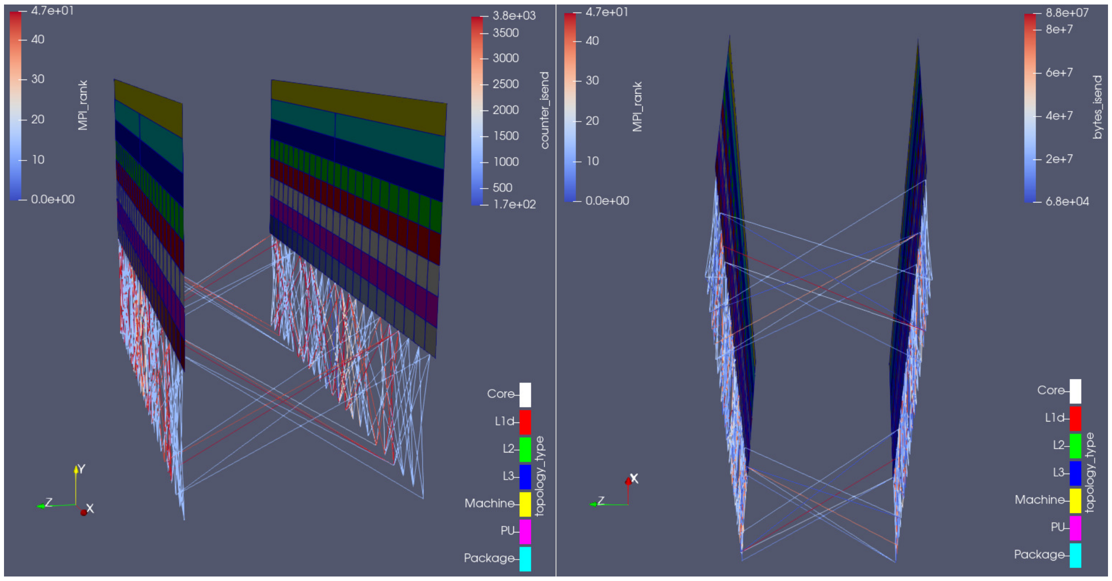

Figure 12. Visualization of the new communication pattern in Hydra from two different camera angles, at an arbitrary time-step, colored by number of MPI_Isend calls (left) and total amount of bytes sent on those calls (right) on that time-step

involved).

We submitted such results to Rolls-Royce, whose developers then changed their code and sent it back to us. The new communication behavior can be seen in Figure 12. Notice how the minimum number of messages sent between any pair of processes dropped from 1500 to 170 (see the lower limit of the scale at the upper-right corner of the left picture); analogously, how the minimum amount of data sent raised from 0 to 68 kB (see the lower limit of the scale at the upper-right corner of the right picture). I.e. now there are no more empty messages being sent, and this is visible in the visualization of the communication lines. The plugin has been successfully used in a real life performance optimization problem, whose detection

```
#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

would be difficult if using the currently available tools[14].

4 OVERHEAD

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

4.1 Settings

In the following tables, the *baseline* results refer to the pure simulation code, running as per the settings presented in Sec. 3; the numbers given are the average of 5 runs $\pm$ 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)[15]. Finally, ++ *plugin* refers to when the plugin is also used: running only in topology mode and in only one *feature* (regions or communication) at a time[16] and on the iterations when there would be generation of output files[17]. The percentages shown in these two columns are not the variation of the measurement itself, but its deviation from the average baseline result.

Score-P was always applied with the `--nocompiler` flag. This option is enough when the plugin is used to show communication between ranks, as no instrumentation (manual or automatic) is needed

[14]Vampir, for instance, is not able to show an aggregated view of the communication pattern inside the time step, as it has no knowledge about it (when it starts and when it finishes). The data scales shown on Figures 11 and 12 are not available then, what makes it difficult to spot channels (pairs of sender/receiver) through which no proper data is sent (messages with 0 size).

[15]If activated, there would be at the end of the simulation, apart from the simulation's 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 usage, but it is not recommended: the overheads sum up.

[16]The plugin can perfectly run in all its modes and features at the same time (geometry mode requires the simulation to have a Catalyst adapter; see our previous papers). However, this is not recommended: the overheads sum up.

[17]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.

when solely MPI calls are being tracked. On the other hand, the instrumentation overhead is considerably higher when the target is to measure code regions, as every single function inside the simulation code is a potential candidate for analysis (as opposed to when tracking communications, when only MPI-related calls are intercepted). In this case, it was 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). An intervention as illustrated in Figure 13 achieves this: `if MODULO...` additionally guarantees measurements are collected only when there would 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 unique importance, as all data arrays must be defined then (i.e. the (dis)appearance of variables in later time-steps is not allowed)[18]. Finally, when measuring code regions, interception of MPI-related routines was turned off at run-time[19].

4.2 Results

Tables 1 and 2 show the impact of the proposed plugin on the test-cases performance. The memory section refers to the *peak* memory consumption per parallel process, reached *somewhen* during the simulation; it neither means that *all* ranks needed that amount of memory (at the same time or not), nor that the memory consumption was like that during the *entire* simulation. Score-P itself introduced no perceptible overhead; on its turn, the plugin did, an that is because it is equipped with a Catalyst adapter (whose footprint lies mostly on memory consumption (Ayachit et al., 2015)). Catalyst needs this memory to store the artificial geometry's (the topological representation of the hardware resources being used) coordinates and cells definition, plus all the data arrays associated with them (amount of times a function was executed, amount of messages sent between two ranks etc.), for each time-step during the simulation. Hence the added memory footprint is higher.

The run time overhead, on its turn, is only critical when measuring the two code regions selected in Hydra: they are called millions of times per time-step, hence their instrumentation is heavy. Otherwise the plugin's or Score-P's footprints lie within the statistical oscillation of the baseline results.

Table 1. Plugin's overhead when measuring code functions on topology mode.

	running time			**memory (MB)**		
	++ plugin	**+ Score-P**	**baseline**	**++ plugin**	**+ Score-P**	**baseline**
MG	31m42s (0%)	31m09s (-1%)	31m37s $\pm$ 2%	648 (42%)	479 (5%)	455 $\pm$ 0%
BT	34m28s (0%)	34m26s (0%)	34m28s $\pm$ 1%	648 (42%)	478 (5%)	455 $\pm$ 0%
Hydra	47m04s (12%)	43m52s (4%)	42m00s $\pm$ 0%	382 (22%)	323 (3%)	314 $\pm$ 0%

Table 2. Plugin's overhead when showing communication on topology mode.

	running time			**memory (MB)**		
	++ plugin	**+ Score-P**	**baseline**	**++ plugin**	**+ Score-P**	**baseline**
MG	31m34s (0%)	31m09s (-1%)	31m37s $\pm$ 2%	648 (42%)	479 (5%)	455 $\pm$ 0%
BT	34m24s (0%)	34m08s (-1%)	34m28s $\pm$ 1%	648 (42%)	477 (5%)	455 $\pm$ 0%
Hydra	42m53s (2%)	43m50s (4%)	42m00s $\pm$ 0%	397 (26%)	316 (1%)	314 $\pm$ 0%

CONCLUSIONS

In this paper, we have extended our software to allow mapping performance data to a three-dimensional representation of the cluster's architecture, by means of (combining) the code instrumenter *Score-P* and the graphics manipulation program *ParaView*. The tool, which takes the form of a Score-P plugin, introduces the following novel capabilities to the spectrum of code analysis resources:

- detailed view up to topology component level (i.e. in which core of which socket of which node a specific MPI rank is running);

[18]Hence, there were two narrowing factors for Score-P in the end: the *spacial* (i.e. accompany only the desired functions) and the *temporal* (accompany only at the desired time-steps) ones.

[19]By means of the `SCOREP_MPI_ENABLE_GROUPS` environment variable (see Sec. 2.1.1 above).

- limit visualization to resources being used by the simulation;
- native association with the simulation's time-step;
- individual components of the visualization (like the network switches) are optional to produce and to display (i.e. see only what you want to see);
- easily distinguish between messages coming from ranks within the same compute node from those coming from ranks running in other compute nodes, something not possible in a tool like Vampir;
- individually applicable color scale to each element of the visualization, allowing, for example, to color the communication lines by amount of bytes sent, receiver id, sender id etc (something also not possible in Vampir);

All that under the graphic quality of today's top-of-the-art visualization program, ParaView: render views are fully manipulatable and tens of filters are available to further dig into the data. ParaView is the best option as visualization software because of all the resources already available in – and experience accumulated by – it after decades of continuous development. Visualization techniques do not use to be the specialization field of programmers working with code performance: it is more reasonable to take advantage of the currently available graphic programs than attempting to equip the performance tools with their own GUIs (from scratch).

Our tool is based exclusively on open-source dependencies; its source code is freely available[20], as the raw data of the benchmark results presented in this paper[21]. It works with either *automatic* or *manual* code instrumentation and independently of Score-P's *profiling* or *tracing* modes. Lastly, its output frequency (when doing post-mortem analyses) is adjustable at run-time (through the plugin input file), like in Catalyst itself.

FUTURE WORK

We plan to continue this work in multiple directions:

Scale the tool: To keep testing our tool in bigger and bigger test cases, in order to investigate its scalability limits (if any).

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.

Remove the necessity of the topology configuration file: When running the plugin in topology mode, get the network details directly from system libraries (as done with the hardware details). Both Slurm and the hwloc team – through its sister project, *netloc*[22] (Goglin et al., 2014) – are straining in that direction, but it is currently not yet possible (partially because the retrieval of the switches configuration requires root access and therefore needs to be executed by the cluster's admins).

Extend list of supported communication calls: To make the tool capable of detecting calls of other communication protocols, like *GPI-2*[23] (Grünewald and Simmendinger, 2013). This will require a respective extension of Score-P's substrate plugin API.

Extend list of detectable performance phenomena: To extend the list of performance-relevant phenomena which can be detected by the plugin, for example: cache misses, memory accesses, I/O flows etc. This will also require a respective extension of Score-P's substrate plugin API.

Use plugin for teaching: Finally, explore the possibility of using the tool for teaching of *parallel computing*, especially in topics like data locality, job allocation, computer architecture, sharing of computational resources etc.

[20]https://gitlab.hrz.tu-chemnitz.de/alves--tu-dresden.de/catalyst-score-p-plugin
[21]https://dx.doi.org/10.25532/OPARA-119. We are unable to provide the raw data related to Rolls-Royce's code due to copyright issues.
[22]https://www.open-mpi.org/projects/netloc/
[23]The open-source implementation of the *GASPI* standard, see https://www.gaspi.de/.

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

Alves, R. F. C. and Knüpfer, A. (2019). In Situ Visualization of Performance-Related Data in Parallel CFD Applications. In Schwardmann, U., Boehme, C., B. Heras, D., Cardellini, V., Jeannot, E., Salis, A., Schifanella, C., Manumachu, R. R., Schwamborn, D., Ricci, L., Sangyoon, O., Gruber, T., Antonelli, L., and Scott, S. L., editors, *Euro-Par 2019: Parallel Processing Workshops*, pages 400–412, Cham. Springer International Publishing.

Alves, R. F. C. and Knüpfer, A. (2020). Enhancing the in Situ Visualization of Performance Data in Parallel CFD Applications. *Supercomputing Frontiers and Innovations*, 7(4).

Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., and Mauldin, J. (2015). 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, pages 25–29, New York, NY, USA. ACM.

Bauer, A. C., Abbasi, H., Ahrens, J., Childs, H., Geveci, B., Klasky, S., Moreland, K., O'Leary, P., Vishwanath, V., Whitlock, B., and Bethel, E. W. (2016). In situ methods, infrastructures, and applications on high performance computing platforms. *Computer Graphics Forum*, 35(3):577–597.

Bell, R., Malony, A. D., and Shende, S. (2003). Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In Kosch, H., Böszörményi, L., and Hellwagner, H., editors, *Euro-Par 2003 Parallel Processing*, pages 17–26, Berlin, Heidelberg. Springer.

Frumkin, M., Jin, H., and Yan, J. (1998). Implementation of NAS parallel benchmarks in high performance fortran. *NAS Technical Report NAS-98-009*.

Goglin, B., Hursey, J., and Squyres, J. M. (2014). Netloc: Towards a comprehensive view of the HPC system topology. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 216–225, Minneapolis, MN, USA. IEEE.

Grünewald, D. and Simmendinger, C. (2013). The GASPI API specification and its implementation GPI 2.0. In M. Weiland, A. Jackson, N. J., editor, *Proceedings of the 7th International Conference on PGAS Programming Models*, pages 243–248, Scotland, UK. The University of Edinburgh.

Isaacs, K. E., Bremer, P., Jusufi, I., Gamblin, T., Bhatele, A., Schulz, M., and Hamann, B. (2014). Combing the communication hairball: Visualizing parallel execution traces using logical time. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2349–2358.

Isaacs, K. E., Landge, A. G., Gamblin, T., Bremer, P., Pascucci, V., and Hamann, B. (2012). Abstract: Exploring performance data with boxfish. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1380–1381.

Khanal, B., He, L., Northall, J., and Adami, P. (2013). Analysis of Radial Migration of Hot-Streak in Swirling Flow Through High-Pressure Turbine Stage. *Journal of Turbomachinery*, 135(4).

Knüpfer, A., Rössel, C., Mey, D. a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., and Wolf, F. (2012). Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In Brunst, H., Müller, M. S., Nagel, W. E., and Resch, M. M., editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lapworth, L. (2004). Hydra-CFD: a framework for collaborative CFD development. In *International Conference on Scientific and Engineering Computation (IC-SEC), Singapore, June*, volume 30.

Miletto, M. C., Schepke, C., and Schnorr, L. M. (2021). Optimization of a Radiofrequency Ablation FEM Application Using Parallel Sparse Solvers. In *The 2020 International Conference on High Performance Computing & Simulation (HPCS 2020)*.

Schnorr, L. M., Huard, G., and Navaux, P. O. (2010). Triva: Interactive 3D visualization for performance analysis of parallel applications. *Future Generation Computer Systems*, 26(3):348 – 358.

Schöne, R., Tschüter, R., Ilsche, T., Schuchart, J., Hackenberg, D., and Nagel, W. E. (2017). Extending the functionality of Score-P through plugins: Interfaces and use cases. In Niethammer, C., Gracia, J., Hilbrich, T., Knüpfer, A., Resch, M. M., and Nagel, W. E., editors, *Tools for High Performance Computing 2016*, pages 59–82, Cham. Springer International Publishing.

Solórzano, A. L. V., Navaux, P. O. A., and Schnorr, L. M. (2021). Temporal Load Imbalance on Ondes3D Seismic Simulator for Different Multicore Architectures. In *The 2020 International Conference on High Performance Computing & Simulation (HPCS 2020)*.

Theisen, L., Shah, A., and Wolf, F. (2014). Down to Earth – How to Visualize Traffic on High-dimensional Torus Networks. In *2014 First Workshop on Visual Performance Analysis*, pages 17–23.