In Situ Visualization of Performance Data in Parallel CFD Applications

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Rigel Falcão do Couto Alves
geboren am 12. Juli 1987 in Salvador, Brazil

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Wolfgang E. Nagel

Dresden, 20 Dezember 2022

Kurzfassung

Diese Dissertation fasst die Arbeit des Autors zur Visualisierung von Leistungsdaten in parallelen *Computational Fluid Dynamics* (CFD) Simulationen zusammen.

Aktuelle Leistungsanalysewerkzeuge sind nicht in der Lage, ihre Daten auf komplexen Simulationsgeometrien (z.B. einem Flugzeugtriebwerk) anzuzeigen. Die Leistung von CFD-Simulationen ist jedoch durch die Struktur der zugrunde liegenden Berechnungen beeinflusst, die wiederum eng mit dem zugrunde liegenden Rechengitter verbunden sind. Daher ist es erforderlich, dass Leistungsdaten auf der gleichen Rechengeometrie visualisiert werden können, aus der sie stammen. Leistungswerkzeuge haben jedoch keine nativen Kenntnisse über das zugrunde liegende Berechnungsgitter der Simulation. Diese wissenschaftliche Lücke kann durch die Zusammenführung der Felder der HPC-*Leistungsanalyse* und der *in-situ-Visualisierung* von CFD-Simulationsdaten geschlossen werden, was durch die Integration bestehender, etablierter hochmoderner Werkzeuge aus jedem Bereich erfolgt.

Als Lösung wurde eine Erweiterung für das Open-Source-Performance-Tool Score-P entworfen und entwickelt, die eine beliebige Anzahl manuell ausgewählter *Codebereiche* (meist Funktionen) abfängt und deren jeweilige Messwerte – *Anzahl* der Ausführungen und kumulative verbrachte *Zeit* – an die Visualisierungssoftware ParaView – über seine in-situ-Bibliothek, *Catalyst* – überträgt, als ob es sich um irgendeine andere strömungsbezogene Variable handelte. Anschließend wurde das Tool um die Fähigkeit erweitert, auch Kommunikationsdaten (Nachrichten, die zwischen MPI-Ranks gesendet werden) über dem CFD-Gitter anzuzeigen. Die Tests und Evaluierungen werden mit zwei branchenüblichen CFD Codes durchgeführt: dem von Rolls-Royce, *Hydra*; und Onera, DLR und Airbus Code, *CODA*.

Andererseits wurde es auch festgestellt, dass die aktuellen Leistungsanalysewerkzeuge eine begrenzte Kapazität haben, ihre Daten über dreidimensionalen, zeitschrittartigen Darstellungen der Clustertopologie anzuzeigen. Parallel dazu, um den Ansatz des neuen Tools nicht auf Codes zu beschränken, die bereits über den in situ Adapter verfügen, wurde es erweitert, um die Leistungsdaten – auch in Codes ohne in-situ – auf einer dreidimensionalen, zeitschrittartigen Darstellung der von der Simulation verwendeten Hardwareressourcen anzuzeigen. Getestet wird es mit den NAS Parallel Benchmarks (NPB) *Multi-Grid* und *Block Tri-diagonal*, sowie nochmals mit Hydra und CODA. Anhand der Benchmarks wird demonstriert, wie die neuen Visualisierungen funktionieren, während echte Leistungsanalysen mit den branchenüblichen CFD-Codes durchgeführt werden.

Die vorgeschlagene Lösung ist in der Lage, konkrete Performance-Probleme aufzudecken, die mit den aktuellen Performance-Tools nicht gefunden worden wären, und die vorteilhafte Veränderungen im jeweiligen Quellcode im realen Leben motivierten. Schließlich wird der Overhead diskutiert und die Eignung für die Verwendung mit CFD-Codes nachgewiesen. Die Dissertation stellt eine wertvolle Ergänzung zum Stand der Technik der hochparallelen CFD-Performanceanalyse dar und dient als Grundlage für weiterführende Forschungsrichtungen.

Abstract

This thesis summarizes the work of the author on visualization of performance data in parallel Computational Fluid Dynamics (CFD) simulations.

Current performance analysis tools are unable to show their data on top of complex simulation geometries (e.g. an aircraft engine). But in CFD simulations, performance is expected to be affected by the computations being carried out, which in turn are tightly related to the underlying computational grid. Therefore it is imperative that performance data is visualized on top of the same computational geometry which they originate from. However, performance tools have no native knowledge of the underlying mesh of the simulation. This scientific gap can be filled by merging the branches of HPC *performance analysis* and *in situ visualization* of CFD simulations data, which shall be done by integrating existing, well established state-of-the-art tools from each field.

In this threshold, an extension for the open-source performance tool *Score-P* was designed and developed, 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 the visualization software *ParaView* – through its in situ library, *Catalyst* –, as if they were any other flow-related variable. Subsequently the tool was extended with the capacity to also show communication data (messages sent between MPI ranks) on top of the CFD mesh. Testing and evaluation are done with two industry-grade codes: Rolls-Royce's CFD code, *Hydra*, and Onera, DLR and Airbus' CFD code, *CODA*.

On the other hand, it has been also noticed that the 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. Parallel to that, in order for the approach not to be limited to codes which already have the in situ adapter, it was extended to take the performance data and display it – also in codes without in situ – on a three-dimensional, framed representation of the hardware resources being used by the simulation. Testing is done with the *Multi-Grid* and *Block Tri-diagonal* NAS Parallel Benchmarks (NPB), as well as with Hydra and CODA again. The benchmarks are used to explain how the new visualizations work, while real performance analyses are done with the industry-grade CFD codes.

The proposed solution is able to provide concrete performance insights, which would not have been reached with the current performance tools and which motivated beneficial changes in the respective source code in real life. Finally, its overhead is discussed and proven to be suitable for usage with CFD codes. The dissertation provides a valuable addition to the state of the art of highly parallel CFD performance analysis and serves as basis for further suggested research directions.

Contents

Nomenclature

API	Application Programming Interface
BT	Block Tri-diagonal (NPB pseudo-application benchmark)
CFD	Computational Fluid Dynamics
CLI	Command Line Interface
CPU	Central Processing Unit
CSM	Computational Solid Mechanics
GASPI	Global Address Space Programming Interface
GPU	Graphics Processing Unit
GUI	Graphic User Interface
HPC	High Performance Computing
hwloc	Hardware Locality (library)
I/O	Input / Output
MG	Multi-Grid (NPB kernel benchmark)
MPI	Message Passing Interface
netloc	Network Locality (library)
NPB	NAS Parallel Benchmarks
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
RAM	Random Access Memory
RANS	Reynolds-Averaged Navier-Stokes (equations)
RDMA	Remote Direct Memory Access

1 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 highly tunned for maximum productivity.

There are many tools for analyzing the performance of parallel applications; one of them is *Score-P*, whose development the *Centre for Information Services and HPC* (ZIH) of this University participates in. It instruments the simulation's code and monitors its execution, and can be easily turned on or off by the user at compile time.[1] However, all tools currently available to *visualize* the performance data (generated by software like Score-P) lack some visualization features, like three-dimensionality, time-step association (i.e. frame playing) and most importantly, matching to the simulation's original geometry (where everything happens in terms of computations and therefore where load imbalances lie).

As a separate category of parallel applications' add-ons, tools for enabling *in situ* visualization of simulations' output data – like 'temperature' or 'pressure' in a Computational Fluid Dynamics (CFD) simulation – already exist too; one example is *Catalyst*. 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. These tools have been developed by visualization specialists for decades and feature abundant visual resources.

The goal of this thesis is to propose a method that merges together the aforementioned approaches. Indeed, many dedicated tools have been created to tackle the countless hardware and software aspects involved in HPC and to make visible the performance bottlenecks they originate, but key correlations (like to the grid being used by the parallel simulation itself) remain untouched; whereas only if all aspects are observed together, maximum performance and scalability become reachable. Typical performance tools have been unable to reach the simulation's geometry and physics because these ones lie inside the realm of the application itself, therefore it is imperative to use a data collection method that reaches those depths as well. Combining the performance tools with in situ processes is thus the only sensible way to fill out this scientific gap.

This work summarizes the author's investigations on this regard. 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).[2] Second, by using the advanced functionalities of dedicated visualization software for the purpose of performance analysis. It is the fourth publication about the research, following a High-Performance Computing and Networking (HPCN) Workshop paper in Euro-Par 2019, a paper in *Supercomputing Frontiers and Innovations* (an international open access journal on the field of High Performance Computing) and a paper in *Peer*

[1] Or even run time, for Python codes.

[2] Both collect or "steal" data from the parallel application and transfer it out via a side channel.

Journal of Computer Science (an international open access journal on the field of Computer Science). The first publication introduced the approach as well as the tool that has been developed to implement it; the second publication expanded the tool to encompass also MPI communication; finally, the third article further expanded the tool to encompass also codes without an in situ processor.

This thesis is organized as follows: first the meaning of the most important concepts which will be mentioned throughout the text is explained (Chapter 2). Then the motivation of the work is further described, presenting the detected scientific bottleneck and the solution proposed to address it (Chapter 3). Next the work will go through the related literature and confirm that the necessity of merging the branches of performance analysis and visualization has already been acknowledged by the scientific community; and present some early attempts to accomplish that (Chapter 4). Then the proposed methodology is presented in detail: the tool that has been created, its prerequisites, integration with a simulation code, modes of operation, and input parameters. Next comes the introduction of the test-cases that will be used to test the software: four, in total – two well known, CFD-derived benchmarks and two industry-grade CFD codes (Chapter 6).

Then comes the most important part: the presentation of the tool's results. Here it will be demonstrated through many pictures the abundance of visualization options which the software has made possible, the insights that can come from them, the correlations that become immediately clear etc. (Chapter 7). This section presents the main contribution of the work: performance data can now be plotted on the original simulation's geometry (regardless of its type: structured grids, unstructured grids etc.) and / or on a three-dimensional representation of the computing architecture's topology, all that in a natively framed (time-step based) way, which can be then easily used to generate graphs, videos etc.

Finally, the overhead associated with using the tool (on each of its modes) will be presented (Section 6.3), before concluding the thesis and describing some ideas for future work. The software is being published under a GNU GPLv3 license; all its dependencies are also proudly open-source. In order to download or git clone the files, just click on the link below:

https://gitlab.hrz.tu-chemnitz.de/alves–tu-dresden.de/catalyst-score-p-plugin/

2 Preliminaries – understanding the concepts

The goal of this section is to present some concepts which will be used across the thesis. Its target audience are the readers who might not be familiarized with them. Such concepts include computational fluid dynamics, parallel applications, performance analysis etc.

2.1 Computational Fluid Dynamics

In order to understand what is Computational Fluid Dynamics, it is first necessary to go back some concepts. *Fluid Mechanics* is the branch of Mechanics which focus on *fluids* (i.e. liquids and gases), be them in motion (*Fluid Dynamics*) or at rest (*Fluid Statics*) [69]. The former is centred on the *Reynolds Transport Theorem*, which can be mathematically described by the equation below:

$$\frac{dB}{dt} = \frac{d}{dt}\left\{\int_{CV} \frac{dB}{dm}\, m'''\, dV\right\} + \int_{CS} \frac{dB}{dm}\, m'''\, (\boldsymbol{u_{rel}}\, \boldsymbol{n})\, dS$$

Where t is the *time*, CV an arbitrary *control volume*, CS its external *control surfaces*; m is the fluid *mass*, m''' its *density*, V its *volume*, S one of its *external surfaces*, n the vector normal to S, u_{rel} the relative velocity between the fluid and the control volume's boundaries; and B any of the three fundamental flow properties: *mass*, *linear momentum* or *energy*. Additionally, a thermodynamic state equation is needed to close the system (i.e. to make the number of unknowns equal to the number of equations involved in the problem). For most cases, this fourth equation will be the *Ideal Gas Law*:

$$p = m'''\, r\, T$$

Where p is the *pressure*, T the *temperature* and r a fluid specific *proportionality constant*[1]. The complete set of four equations is then theoretically solvable, but no one has yet managed to derive an analytical solution to it. The alternative is then to use numerical methods, which require the (physically continuous) domain to be "discretized" (fragmented) into separate pieces, creating something called *mesh* or *grid*, as illustrated in Figure 2.1.

The case shown in Figure 2.1 might still be doable by hand, although tediously. However, consider the example of Figure 2.2. The size of a typical engineering problem exceeds by far the capacity of the human being of manually computing the values of all those flow properties in each of the mesh's *points* or *cells*[2]. This brings the necessity of using computers to solve such problems, and this usage is what is called *Computational Fluid Dynamics*. But apart from the *feasibility* point of view, there is also an *economic* aspect to be taken into consideration, as illustrated by Anderson ([9], S. 3):

[1] Cases where the Ideal Gas Law fail (like in hypersonic flows) use a modified version of the equation presented. The main parameters involved remain, however, the same.

[2] Depending on how those equations are discretized (point-based or cell-based). Both ways are possible.

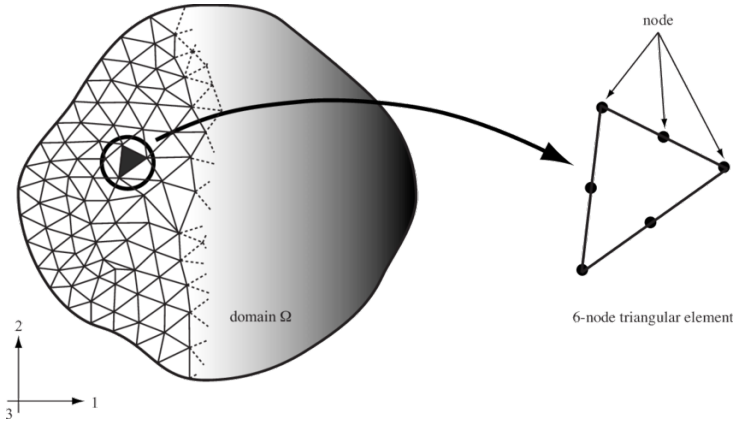


Figure 2.1: Example of the discretization of a continuous domain into separate pieces, for numerical solving (retrieved from [19]).

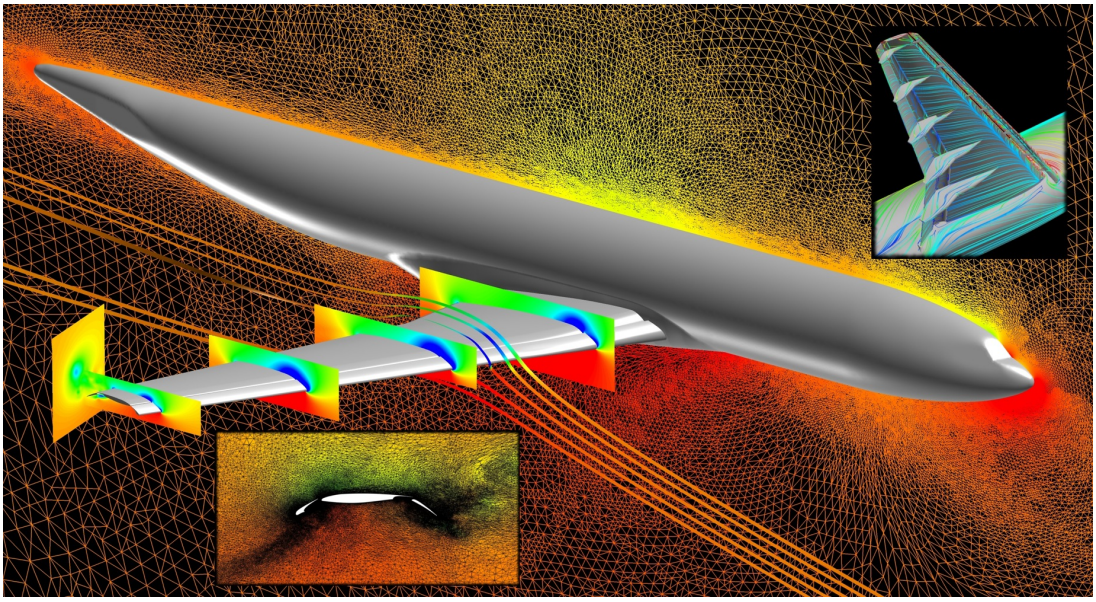


Figure 2.2: Mesh of the surroundings of an aircraft in a NASA problem (retrieved from [12]).

> "In the late 1970's, this approach (the use of supercomputers to solve aerodynamic problems) began to pay off. One early success was the experimental NASA aircraft called HiMAT (Highly Maneuverable Aircraft Technology), designed to test concepts of high maneuverability for the next generation of fighter planes. Wind tunnel tests of a preliminary design for HiMAT showed that it would have unacceptable drag at speeds near the speed of sound; if built that way the plane would be unable to provide any useful data. The cost of redesigning it in further wind tunnel tests would have been around $150,000 and would have unacceptably delayed the project. Instead, the wing was redesigned by a computer at a cost of $6,000."

Such real-life example shows how computer simulations may, if not *replace*, at least *decrease* the amount of experiments needed to understand fluid dynamics phenomena. However, there is still a cost to pay, namely the computational one: the size of many CFD problems exceeds the computing capacity of any computer currently available in the planet, no matter how powerful. What brings us to the next topic.

2.2 Parallel Applications

In order not to halt the progress of science to the ongoing computing capacity of a single *processing unit*,[3] paradigms have been developed in order for such units to be able to work simultaneously to one another (while running the same program); this is called *parallel computing*. There are two main ways of approaching it, illustrated in the figure below:

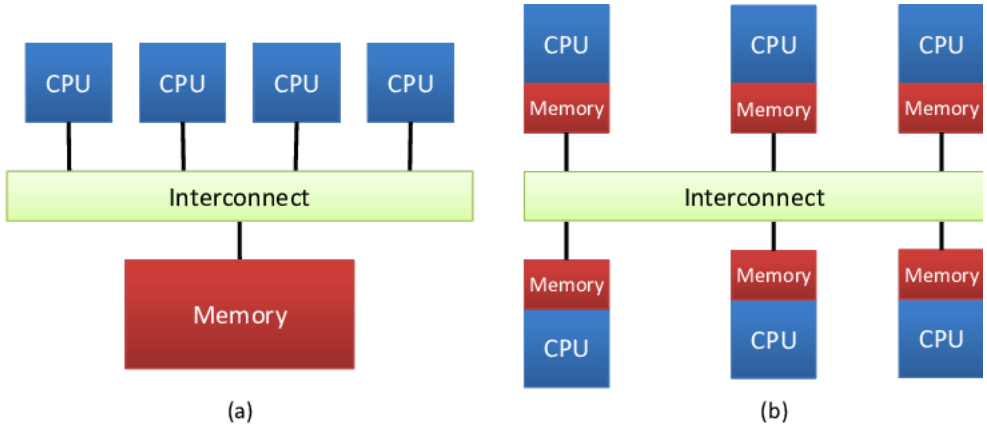


Figure 2.3: Two main approaches to parallel programming: *shared memory* (left) and *distributed memory* (right) (retrieved from [22]).

The approach shown on the left is called *shared memory*. It corresponds to the first and simplest way of parallelizing a program: in a multicore hardware architecture, many of its processing units will simultaneously execute the application. They all share the same memory space (the computer's memory), hence the amount of extra work for the programmer (in order to parallelize its software) is small.[4] However, it has a limit: the – total amount of memory of the – computer where the application is running. In other words, shared memory pushes the boundary of the solvable problems from the (maximum ongoing) size of the processing unit to that of the computer as a whole. The biggest engineering problems would still not be covered, if it was not for the following approach.

Distributed memory consists of interconnecting multiple computers (now called *compute nodes*) within a network, forming an aggregate commonly referred to as *supercomputer*. There is no clear limit anymore to the size of problems which can be solved, as there is no clear limit to the amount of compute nodes which can be connected to each other (in other words, the size of the *cluster* itself). However, the extended frontiers come with a cost for the programmer: schemes must be implemented in order for units (now called *ranks*) in one compute node to be able to communicate with their peers in another compute node. The most famous of such schemes is called *Message Passing Interface* (MPI) [30].

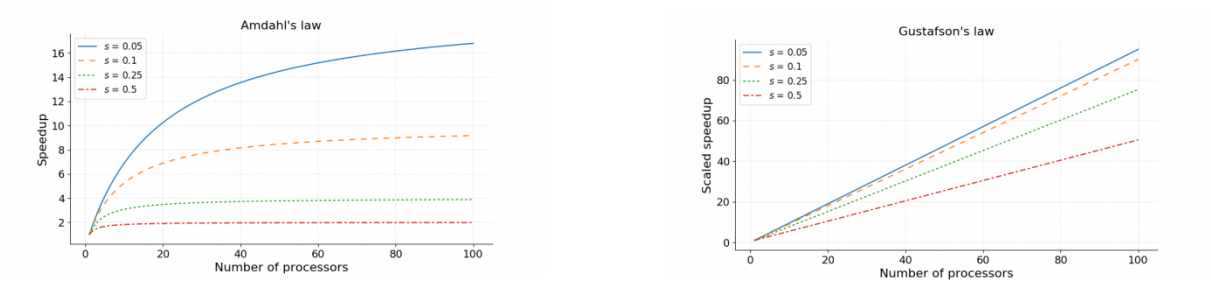
The two approaches described above are not mutually exclusive: it is possible to use distributed memory to run a code across different compute nodes, plus shared memory to further fragment the simulation across the processing units within each node. On the other hand, it is also possible to use distributed memory schemes when you are inside the node, thus simplifying the coding effort (i.e. select one approach which is valid everywhere). The test-cases selected for this thesis follow this path.

[3] The smallest part within a computer hardware responsible for executing computational instructions (basic arithmetic, logic evaluations etc.).

[4] The most famous implementation of this approach is called *Open Multi-Processing* (OpenMP) [24].

Finally, these two approaches refer to what is called *coarse grained parallelism*. There are also other ways to parallelize the execution of computational instructions, including *vectorization*: loop operations are performed simultaneously in many[5] of the arrays' elements if each iteration is independent of one another. This optimization of computer algorithms is usually performed automatically by the compiler when building the executable from the source code, providing the underlying hardware architecture is equipped with *single instruction, multiple data* (SIMD) capabilities.

2.2.1 Scaling



(a) $speedup = \frac{1}{s+\frac{p}{N}}$ (b) $speedup = s + p * N$

Figure 2.4: Comparative displays of Amdahl's and Gustafson's Laws behaviors (retrieved from [49]).

Intrinsically related to the idea of parallel computing is that one of *scaling*: how the code execution behaves as more computational power is added for it. There are two main approaches to look at scaling; the first of them is called *strong scaling*: running a fixed-size problem under more and more computational resources. This type of scaling is governed by the *Amdahl's Law* [8], which states:

$$speedup = \frac{1}{s + \frac{p}{N}}$$

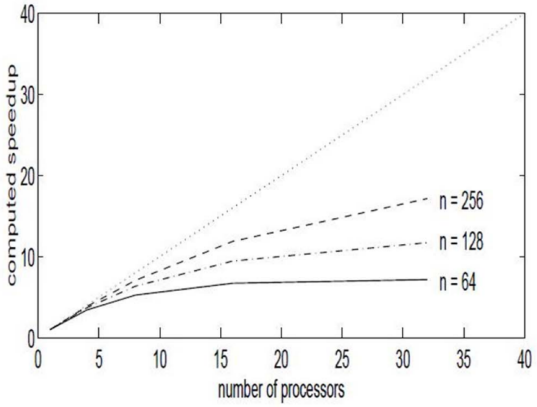
Where s corresponds to the proportion of the execution time spent on the part of the code which cannot be parallelized, p the proportion of the execution time spent on the part of the code which can be parallelized and N the number of parallel processes used. The consequence of such formulation is that the maximum speedup which can be achieved is limited by the serial fraction of the code. This is illustrated in Figure 2.4a.

The other type of scaling is called *weak scaling*: increasing the size of the problem (e.g. increasing the refinement of the computational grid) as you add more and more computational resources. This type of scaling is governed by the *Gustafson's Law* [32], which states:

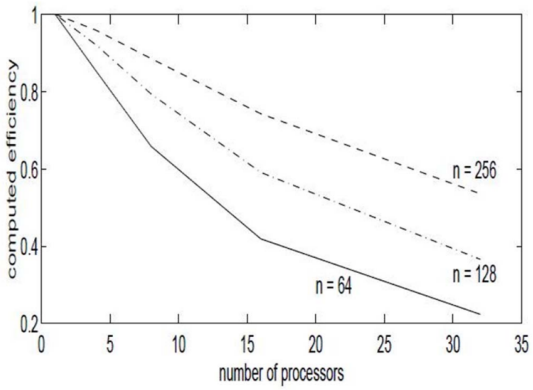
$$speedup = s + p * N$$

Where s, p and N have the same meaning as before. The consequence of such formulation is that there is no maximum speedup which can be achieved, but the slope of the curve will always be less than one. This is illustrated in Figure 2.4b.

[5]The exact amount of operations that can be performed simultaneously changes with respect to the underlying hardware architecture.



(a) $speedup = \frac{T_1}{T_N}$



(b) $parallel\,efficiency = \frac{S}{N}$

Figure 2.5: Comparative displays of the speedup and parallel efficiency behaviors of a typical algorithm, here the summation of n numbers (retrieved from [73]).

Finally, the *speedup* can also be expressed by the ratio between the time taken to solve the problem in serial over the time taken to solve the same problem in parallel (with N processes), what yields the formula below:

$$speedup = \frac{T_1}{T_N}$$

The ideal speedup is then the line $y = x$, i.e. using N processes makes the code run N times faster. Using the formula above, it is possible to define the analogous concept of *parallel efficiency*, as the ratio between the speedup (S) and the number of processes used:

$$parallel\,efficiency = \frac{S}{N}$$

The ideal parallel efficiency is then the line $y = 1$, i.e. it remains 100% regardless of the number of processes used. The typical behavior for both quantities is shown on Figure 2.5.

Being able to go beyond the computing capacity of one single machine is powerful, but at the same time takes the difficulty associated with writing, tunning and debugging computer applications to a whole new level. What brings us to the next topic.

2.3 Performance Analysis

Supercomputers are, from an economical point of view, heavy investments, both to build (the infrastructure required) and to operate (the energy consumption). This translates into a higher responsibility for its users: the programs they write must be able to use the available resources on the most efficient way possible – such that the concept of *supercomputer* is a synonym of that of *High Performance Computing* (HPC). It is imperative to ensure that the code execution will indeed get faster as more computational resources are added. In such a massive computing environment, analyzing – and attempting to improve – the performance of applications becomes a routine procedure, as illustrated in the diagram below:

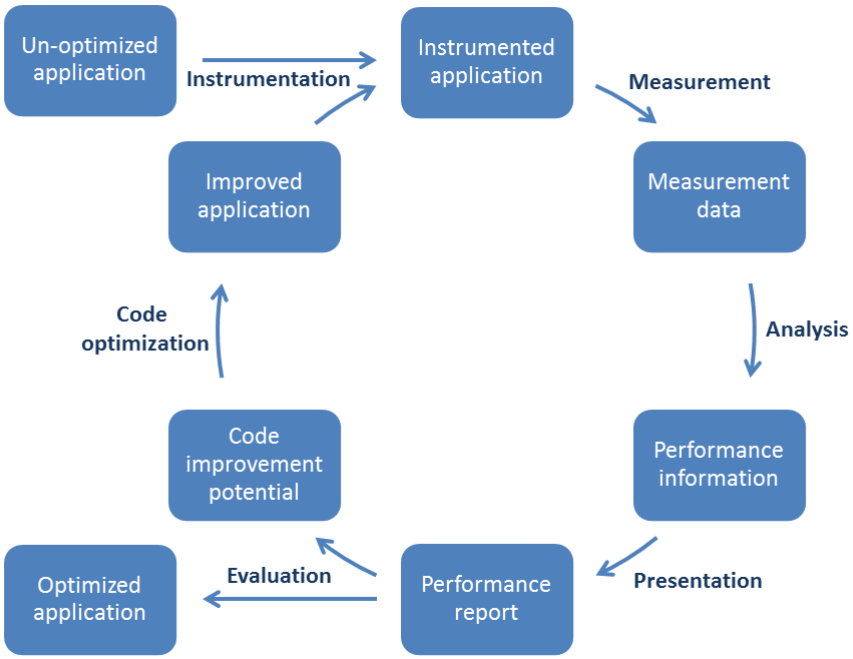


Figure 2.6: Work cycle of the performance optimization process of a computer application (retrieved from [2]).

In order to assist the programmer in such 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 (distributed memory), OpenMP, Pthreads, OmpSs (shared memory), CUDA, OpenCL, OpenACC (accelerators-based), UPC, SHMEM (partitioned global address space) 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) [62]. It all started, in any case, with simple start-stop timer measurements inserted manually inside the code, which then evolved in order to encompass more and more details about the performance of each of its sections, be it related to the functions executed, the communication made between processes etc.

When the goal is to monitor the performance of parallel codes, there are two main ways of doing it: *profiling* and *tracing*, illustrated in Figure 2.7. Historically, "*program profiling* counts the number of times each basic block in a program executes. *Instruction tracing* records the sequence of basic blocks

traversed in a program execution" (highlights are ours) [11]. *Basic blocks* were traditionally functions (and subroutines, to abide by Fortran's nomenclature). Performance tools have since then evolved in such a way that both methods became overlapped. Current literature states that, "independently of the acquisition method, data can be collected and recorded in an aggregated form or in its temporal order. The aggregated form is called a *profile*, whereas temporally ordered data is stored as a *trace*. The terms profiling and tracing refer to the process of generating a profile or trace and do not warrant a specific data acquisition method or presentation" (highlights are ours) [25].

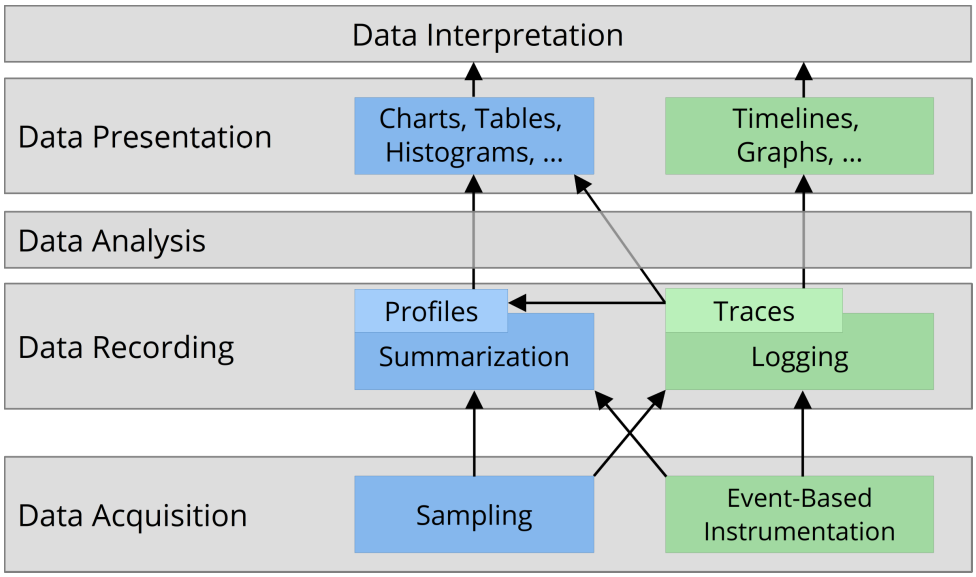


Figure 2.7: Layers which comprise the performance analysis process (retrieved from [25]).

Typical profiling tools nowadays will deliver to the user the amount of times each code function is executed and the total (i.e. accumulated) time spent in those executions. Calls to the same routine in distinct moments inside the code (e.g. synchronization barriers when running in parallel) cannot be distinguished from each other, unless they happen in different callpath depths. Given such simplifications, the time and memory footprints of profiling are considerably smaller than tracing, when the complete order of function executions is registered, with entrance and exit timestamps of every one of them. Current tracing tools are powerful enough to even portray the messages exchanged between ranks when running the code in parallel, with details about source and destination of each message; but such highly detailed analyses introduce heavy overheads (both from time and memory point-of-views) and therefore require fine tunning of the instrumentation process (usually by means of manually defined filters).

In general, profiling is advised when the user wants to get an idea about the overall behavior of the code, which functions are taking longer to execute, are there time imbalances between the parallel ranks etc. Tracing, on its turn, is recommended when the user wants to obtain a closer insight into a specific part of the code, the communication being made inside it etc. Regardless of the method chosen, it is important to ensure: a) the overhead introduced by the performance tool does not exceed a certain threshold, otherwise one cannot be sure anymore the measurements really reflect the code's natural behavior; and b) the performance tool does not break the code natural capacity to scale (i.e. to run with more and more processes) and to perform in parallel.

2.4 Hardware Topology

Central processing units, or simply *processing units* (PUs) are only one of many depths – namely, the lowest one – which comprise the topology of a computer's hardware. The others are illustrated in the figure below:

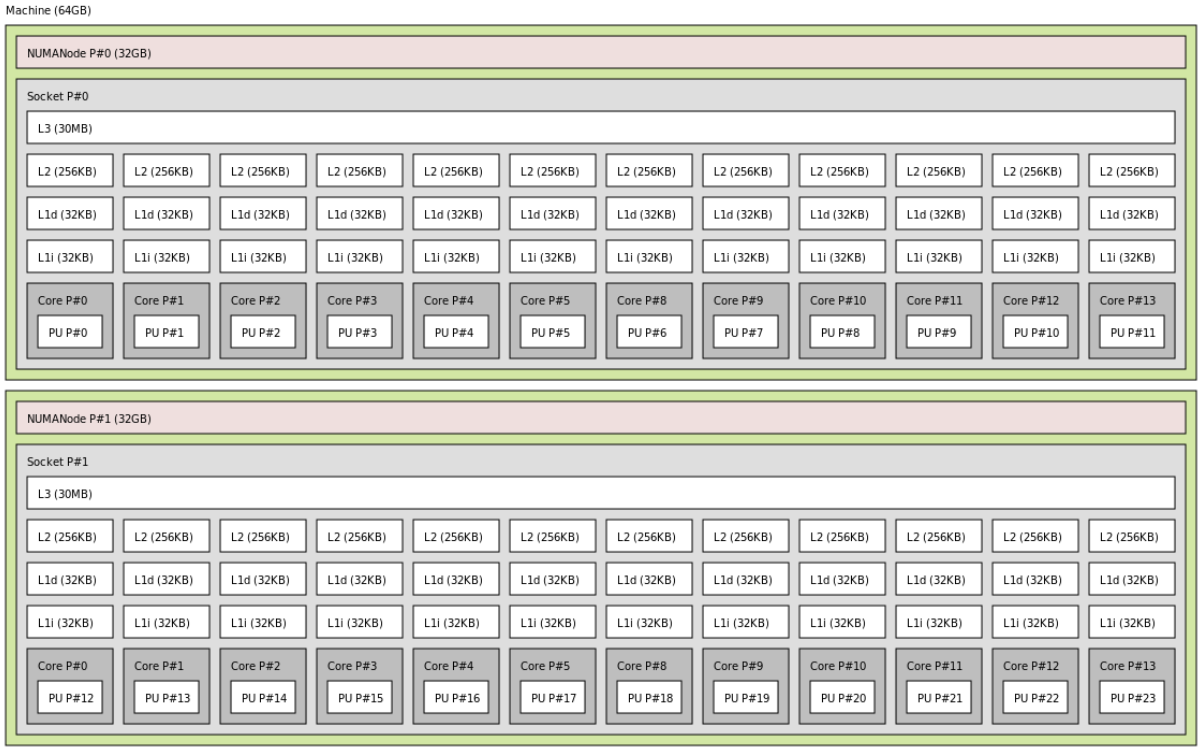


Figure 2.8: Example of the hardware topology of a supercomputer node (retrieved through the *lstopo* Linux utility).

The PUs are the layers closest to the user's application and responsible for executing its instructions. In the system shown above, there is one PU per *core*, which is in close contact with the L1 and L2 cache memories, which in turn are *Random Access Memories* (RAM) responsible for temporarily storing data used in the code (like the values of temperature, pressure etc. in each point of a CFD mesh). A group of cores share then a L3 cache memory, which in turn is connected to one of the motherboard's *sockets*. Finally, the coupling of a socket with an associated RAM memory which follows the *Non-Uniform Memory Access* (NUMA) approach[6] forms a NUMA node.[7]

When it comes to parallelize a simulation, shared memory schemes allow each of the PUs to simultaneously execute the application. Distributed memory schemes, on their turn, allow the PUs in this machine to communicate (to share a simulation job) with fellows in other machines. In these cases, the information flows from the origin PU through its L1, L2 and L3 caches, the socket, then from the machine through the *network* to the destination's machine, then all the way downwards towards the destination PU. Such layout has implications for the performance of the code being run: messages are faster when

[6] It consists of a design where the access time of memory located close to the accessing processor is shorter than that local to another processor.

[7] Extras like the graphic card or input / output (IO) ports are not shown on the picture, but would be part of a real hardware topology.

exchanged between PUs running under the same socket, followed by under different sockets of the same compute node, finally under different nodes. Hence a) it is strategic to increase the number of cores per socket and of sockets per node within the hardware architecture (as to increase its parallelism); and b) it is important to be able to visualize performance data correlated with the architecture information of the computational resources being used to run the simulation.

Finally, software must be able to properly explore the caching layout in order to take full advantage of the performance they can get from the hardware. That means making sure the most accessed data is stored in the L1 cache (the fastest, but also smallest of all), followed by the L2 cache etc.

2.5 Network Topology

As mentioned in the previous topic, when messages need to travel between different machines, they need to go through the *network*: switches which need to be traversed in order for inter-node communications to be performed, as illustrated in the diagram of Figure 2.9.

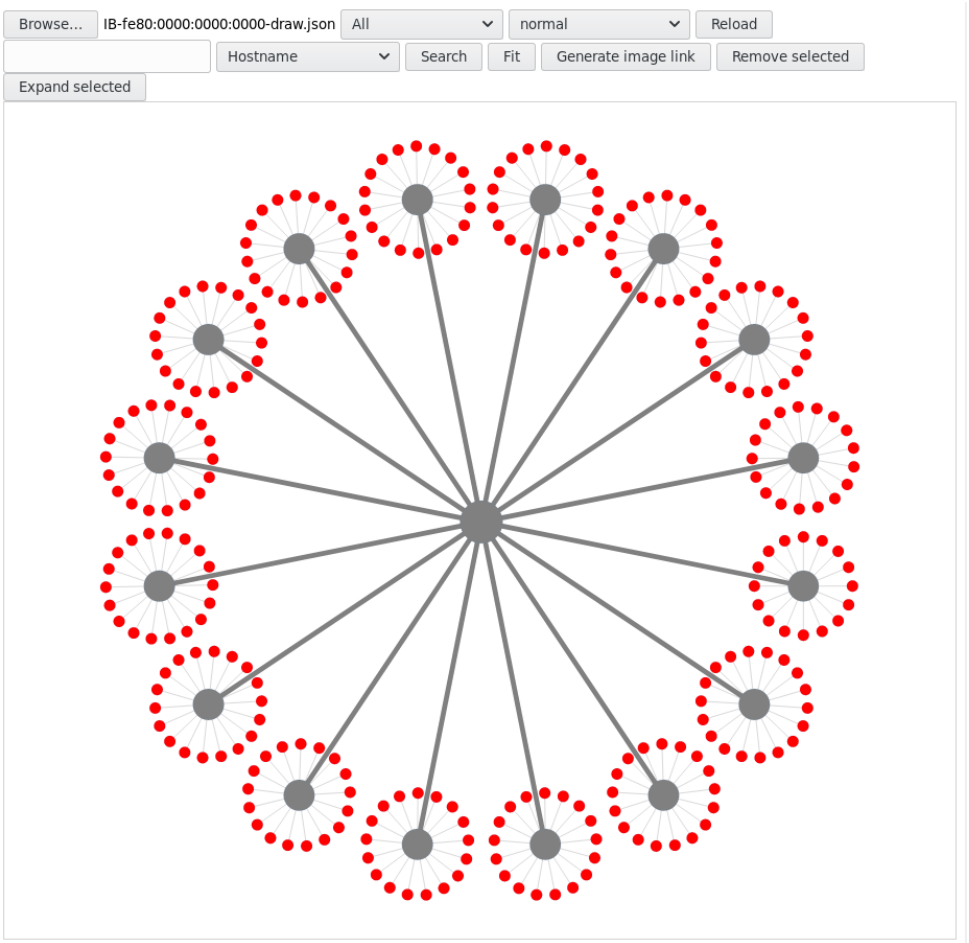


Figure 2.9: Symbolic representation of the network connecting compute nodes (the red circles in the figure) within a cluster architecture (retrieved from [1]).

Each machine (called *compute nodes* within a cluster context) is a red circle in the figure. They are organized in *islands*: 16 (in the picture's example) machines connected to each other through a *leaf*

(i.e. end-of-line) *switch*: MPI messages exchanged between nodes in this island will go through this switch before they are delivered. If the communicating nodes happen to be located in different islands, the messages between them will need to go up levels within the network hierarchy until they reach a common parent; in the example shown, there is only one: the master switch (the central grey circle in the figure), which in turn interconnects the 16 leaf-switches of the drawn cluster. As one might intuitively then guess, data is exchanged faster between nodes located within the same island when compared to those in different islands.[8]

Finally, an important optimization in the context of networks is the idea of *Remote Direct Memory Access* (RDMA): the capacity of transferring data directly between the wire and the application's memory buffers, without needing to pass by all the hierarchies of the hardware topology (described in the previous section) and, most importantly, without needing to involve the CPU responsible for managing the application. Figure 2.10 below compares the traditional way of communicating through the network (left) with the optimized approach (right-hand side). Famous vendor implementations of RDMA include RDMA over Converged Ethernet (RoCE) and InfiniBand.

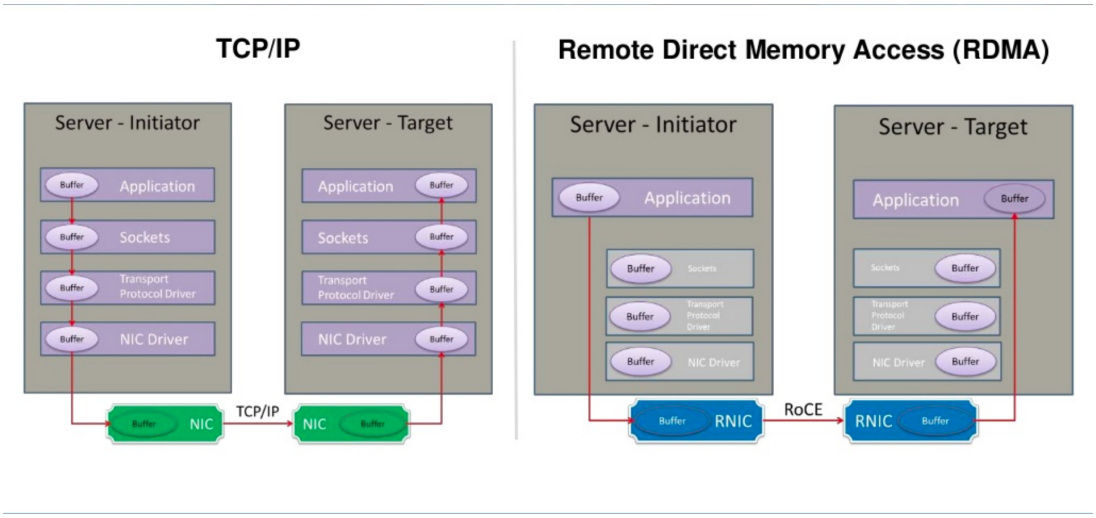


Figure 2.10: Comparison between the traditional way of communicating through the network (left) with the optimized approach using RDMA (right-hand side) (retrieved from [15]). Famous vendor implementations of RDMA include RDMA over Converged Ethernet (RoCE) and InfiniBand.

2.6 In Situ Processing

CFD simulations, especially when running in massively parallel environments (like supercomputers), may produce unmanageable amounts of data: information about each flow property (temperature, pressure, velocity etc.) in every one of the – billions (in the biggest cases) of – mesh points / cells. But apart from the *size* limit, there is also a *temporal* one: it is not feasible to wait (sometimes days or even weeks) until the simulation is finished in order to check the first results, sometimes just to realize a change was

[8] Parallel programs therefore aim to optimize rank placement within the network topology according to ranks' communication relationships. This means that performance analysis tools must be able to visualize both data (topology and communication) together.

needed in the test case inputs, and then run the simulation again... A paradigm change was necessary: the capacity to visualize the outputs on-the-go (as they are produced), as illustrated in Figure 2.11.

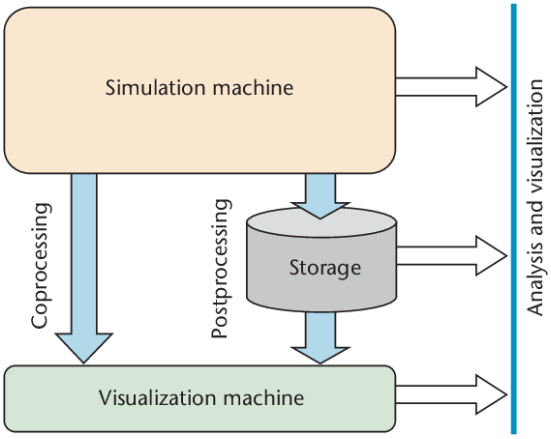


Figure 2.11: Rationale behind in situ: going directly from the simulation to the visualization, as shown on the left path, as opposed to the traditional way, shown on the right (retrieved from [50]).

In situ methods grew with time to also allow users to pause the simulation and modify its parameters, before resuming the code execution; or to stop saving data to disk altogether. On the other hand, they also strived for a full *in memory* operation: using the memory space of the simulation itself to plot the data, rather than making copies of it (just for visualization purposes). Finally, from the point of view of such visualization, in situ methods are commonly classified as per the scheme in Figure 2.12.

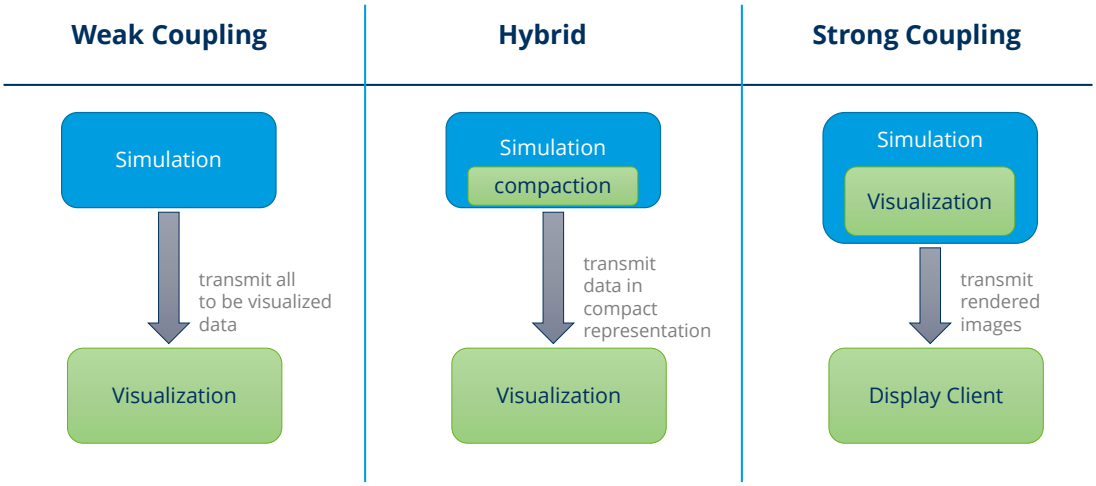


Figure 2.12: Classification of in situ methods according to how output images are produced (retrieved from the lecture notes of course *Data Visualization*, Prof. Raimund Dachselt and Prof. Stefan Gumhold, University of Dresden, Faculty of Computer Science, 2019).

Similarly to what happened in the performance optimization branch, many tools have been developed to cater for in situ needs. For a comprehensive analysis of them, with details about the pros and cons of each tool, the reader is referred to the paper of Bauer et al. [14]. In any case, the in situ is made to be an optional add-on to the main simulation code, i.e. activated at compile time upon request.

2.6.1 In Transit

There are two main ways of doing in situ analyses on a HPC simulation. Bennett et al. cleverly synthesizes them [17]:

> "Their difference lies in how and where the computation is performed. In-situ analysis typically shares the primary simulation compute resources. In contrast, when analyses are performed in-transit, some or all of the data is transferred to different processors, either on the same machine or on different computing resources all together.
>
> Both of these approaches have inherent advantages and disadvantages. In principle, in-transit analysis minimally impacts the scientific simulation. By using asynchronous data transfers to offload computations to secondary resources, the simulation can resume operation much more quickly than if it were to wait for a set of in-situ analyses to complete. However, in practice, transferring even a subset of the raw data over the network may become prohibitive, and furthermore, the memory and/or computing capabilities of the secondary resources can quickly be surpassed. In-situ analyses are not faced with the same resource limitations because the entirety of the simulation data is locally available. However, scientists will typically tolerate only a minimal impact on simulation performance, which places significant restrictions on the analysis. First, simulations are often memory bound and thus all analyses must operate within a very limited amount of scratch space. Second, the analysis is usually allotted only a short time window to execute. The latter restriction is particularly challenging as many data analysis algorithms are global in nature and few are capable of scaling satisfactorily."

As seen above, both approaches have pros and cons, but in situ is at least closer to the goal of a full *in memory* operation. On the other hand, in transit requires a perfect synchronization between the analysis time (in the visualization nodes) and the computations time (in the computation nodes) in order to be worthy: if the analysis finishes too early, the visualization nodes will become idle until the data from the next time step arrives. This is not good from a point of view of computational resources utilization. Alternatively, if the visualization finishes late, it will stop the solver from going on with the computations, as the solver will be waiting to send (for visualization) the data from the computed time step.

Finally, today's most famous visualization programs available for the scientific community are currently capable to handle both approaches [52]. The user toggles between them in the configuration settings.

2.7 Visualization

As a final section to this conceptual chapter, *visualization* as a scientific discipline will be briefly introduced. *Visualize* means to form a mental image of something. In science, this is done with the goal of investigating the observed object, in order to identify its characteristics and spot correlations otherwise invisible in the underlying raw data. The process of going from such raw data to the final images is described on Figure 2.13 below. Human interactions (or human-defined, when such interactions are executed by machines) are needed in order to process the data and map it onto geometrical models for display. In this so called *visualization pipeline*, it is possible to identify three important *steps*, namely:

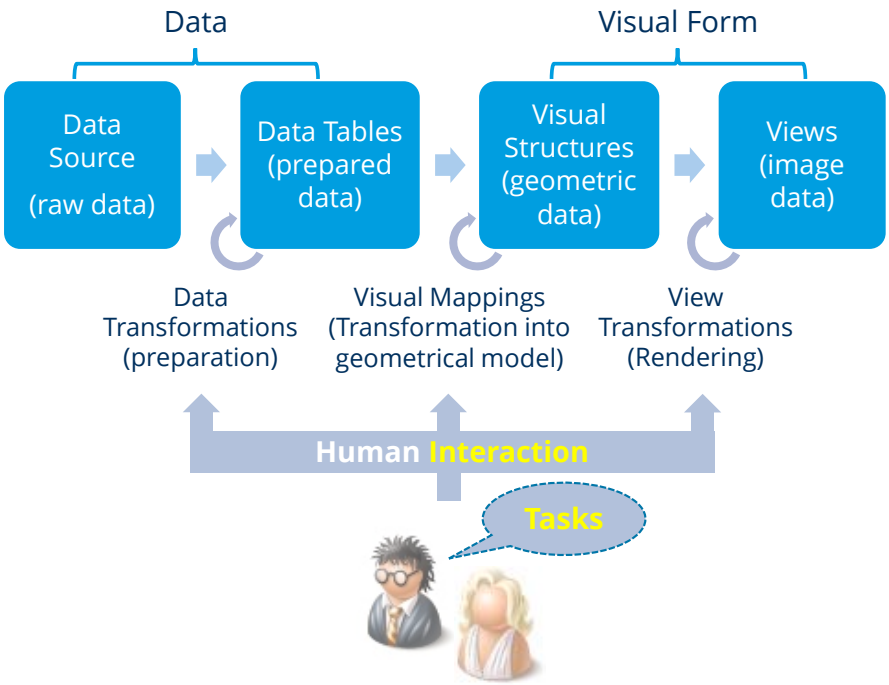


Figure 2.13: Process of going from raw data to final images for visualization (retrieved from the lecture notes of course *Data Visualization*, Prof. Raimund Dachselt and Prof. Stefan Gumhold, University of Dresden, Faculty of Computer Science, 2019).

- *data acquisition*: extract information from a given source. The sources of data and methods of extraction are extremely diverse.
- *analysis*: often the data cannot be transmitted to the observer the way it was collected, given its volume; a preliminary step of filtering and focusing is necessary.
- *visualization*: mapping of the (filtered) data onto visual models, often with geometrical properties, for display on screen.
- *knowledge*: to gain insights about the data is the final goal of the whole process.

The literature identifies three main *goals* of the visualization process, namely: a) *explore*: investigate the data, in search of its properties; b) *communicate*: describe the data, explain its features and allow for decision making on top of them; and c) *control*: be able to administer the underlying process that generated the data in the first place. Similarly, three *phases* of the visualization process can be described: a) *exploration*: the data is being investigated, hypotheses are being made; b) *validation*: visualization techniques are used in order to validate the hypotheses made in the previous phase; and c) *presentation*: results of the analyses are presented to stakeholders.

Shneiderman [61] proposed in his seminal paper the Visual Information Seeking Mantra, which reads: "overview first, zoom and filter, then details on demand". This should be the guideline for any visualization process, which is in turn comprised of *tasks*, classified by him as follows:

- *overview*: obtain an overall view of the entire observation space, recognize global patterns if possible.

- *zoom*: visually focus on specific parts of the observation space.
- *filter*: select a specific subset of the observation space based on a set of visual attributes.
- *details on demand*: display of more detailed information about a selected subset of the observation space.
- *relate*: analyse the data, try to establish relationships between objects.
- *history*: record the actions performed, in order to allow the user to undo them if needed.
- *extract*: extract data from the observation space, e.g. a picture or a video.

The user interacts with the observation space by many different ways, the most common of them being: *mouse click*, *mouse hover* (e.g. to get details on demand), *zooming* (i.e. changing the virtual distance between the screen and a specified focal point in the visualization), *rotating* (moving the screen across a virtual sphere at a fixed radius to a specified focal point in the visualization) and *panning* (moving the screen across a virtual plane at a fixed distance to a specified focal point in the visualization).

Finally, based upon its type, the data can be mapped onto the visualization by three different ways:

- *point data*: mapped to a single point in the visualization space, e.g. the value of a physical property at a specific place on a simulation space; interpolation between values is possible and shown on the area between the points involved.
- *cell data*: mapped to a unit surface or volume in the visualization, e.g. the value of an arbitrary parameter in a symbolic representation of something; interpolation between values is not possible.
- *field data*: applied to the overall visualization space, e.g. the acceleration of gravity or electric potential acting on a field.

All these conceptual aspects highlight the necessity of using dedicated, well-established software when it comes to visualization of scientific data. Layman extensions to performance analysis tools (in order to provide them with visualization capabilities) are not the optimal solution.

3 Motivation

3.1 Problem

All existing tools for performance analysis of parallel (CFD) codes – described in the *Tools Guide* (see section 2.3 above) – have limitations: firstly, they are not able to match their data back to the simulation's original geometry. But this is a fundamental correlation in parallel environments. Consider the following mesh:

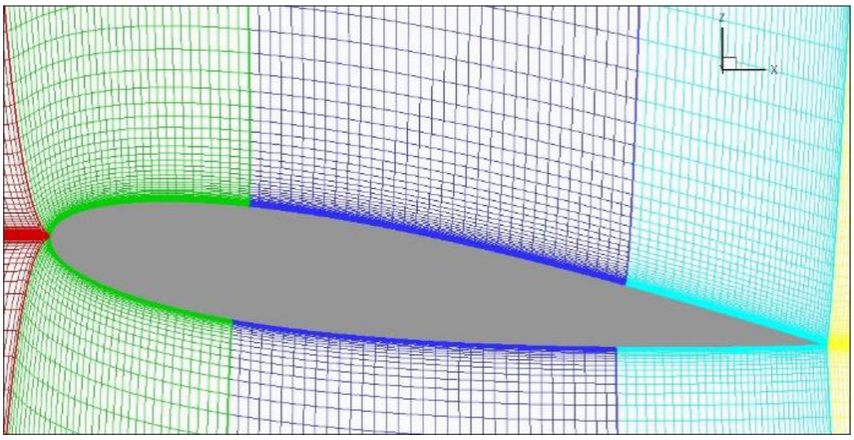


Figure 3.1: Example of a CFD mesh partitioned for parallel execution (retrieved from [34]).

It represents the grid around an airfoil and has been partitioned for parallel execution; each subdomain has a different color on the picture above. Different flow phenomena are expected above and below, upstream and downstream of the object; and such differences are expected to impact on the numerical behavior, hence on the local performance of the solver. With the current tools, the user has no other option than having to open two screens in front of it: one with the performance tool (with its data sorted by subdomain id), another with the grid (colored by the same subdomain id); and then visually matching the results from the former with the spatial location of the latter. It may work in a small test case like this (with around 5 subregions), but what about the one shown on Figure 3.2?

Attempting to visually correlate information from two *Graphic User Interfaces* (GUIs)[1] becomes less and less feasible as the size of the simulated problem scales. The generated performance data will eventually need to be plotted on the simulation's mesh itself, using as hook the subdomain id.

On the other hand, the current performance tools lack full three-dimensional features.[2] This limits their plotting capabilities; for example, when attempting to correlate their generated data to the underlying

[1] A program equipped with a graphic interaction panel (with buttons one can click on), as opposed to *Command Line Interfaces* (CLIs) programs.

[2] Three-dimensionality is not a virtue on itself; two-dimensional views have their benefits (e.g. the capacity to show data straightforwardly on a single screen), just like 3D views have their advantages as well. But few tools explore these last ones, inclusively because they are developed not by visualization specialists, but rather by performance experts.

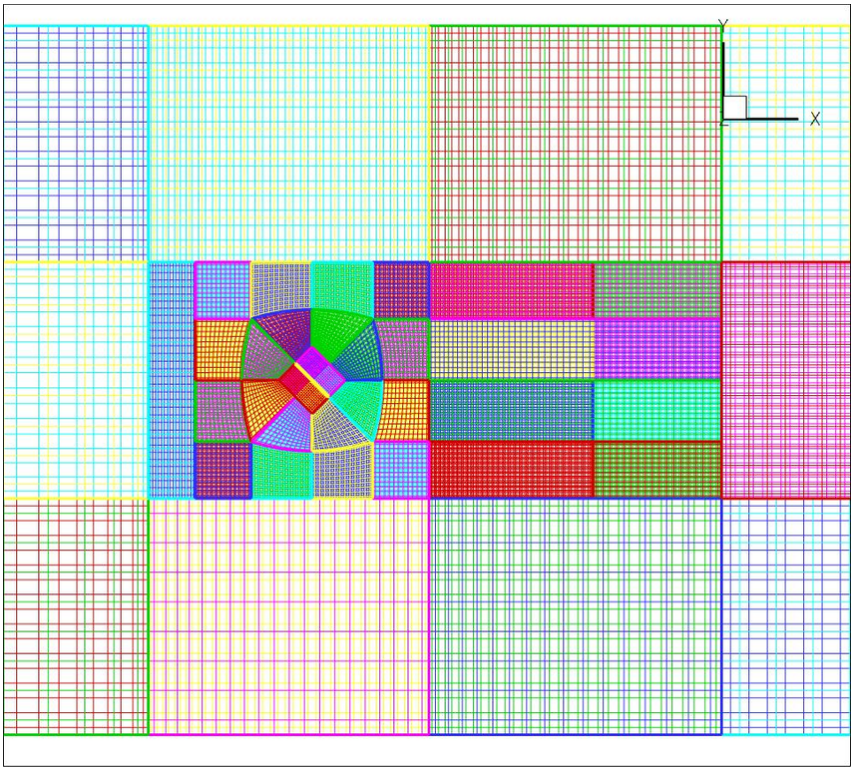


Figure 3.2: Example of a bigger CFD mesh partitioned for parallel execution (retrieved from [18]).

architecture of the computing hardware. Currently only one tool does that (ParaProf [16]), and the results look like Figure 3.3. Even though three-dimensional, they rank low when compared to today's top-of-the-art graphic visualizations.

The lack of three-dimensionality also impacts on the visualization of messages exchanged between ranks during the simulation, what currently needs to be done by means of e.g. the interface shown on Figure 3.4. In an output limited to the two-dimensional space, it is hard to encode e.g. the distinction between messages coming from processes located within the same compute node (where an arbitrary receiver runs) from those coming from processes located in another compute node. Finally, the capacity of alternating the views between the simulation time steps is also missing.

The performance tools are the results of meritorious work from countless performance analysts across decades. The problem being identified here is structural: visualization techniques are the specialization field of another branch of professionals, whose tools have been similarly maturing during the last decades. Attempting to recreate the visualization environment from scratch is like trying to go through all those decades again; a better approach would be to just use the current visualization tools for the purpose of performance analysis. What brings us to the next topic.

3.2 Proposed Solution

Performance analysis tools usually instrument a source code with parameters which will, at run time, monitor its execution. They are made to be easily turned on or off by the user at either compile or run time. Thus the results of the simulation will include its native outputs (e.g. the flow properties in each

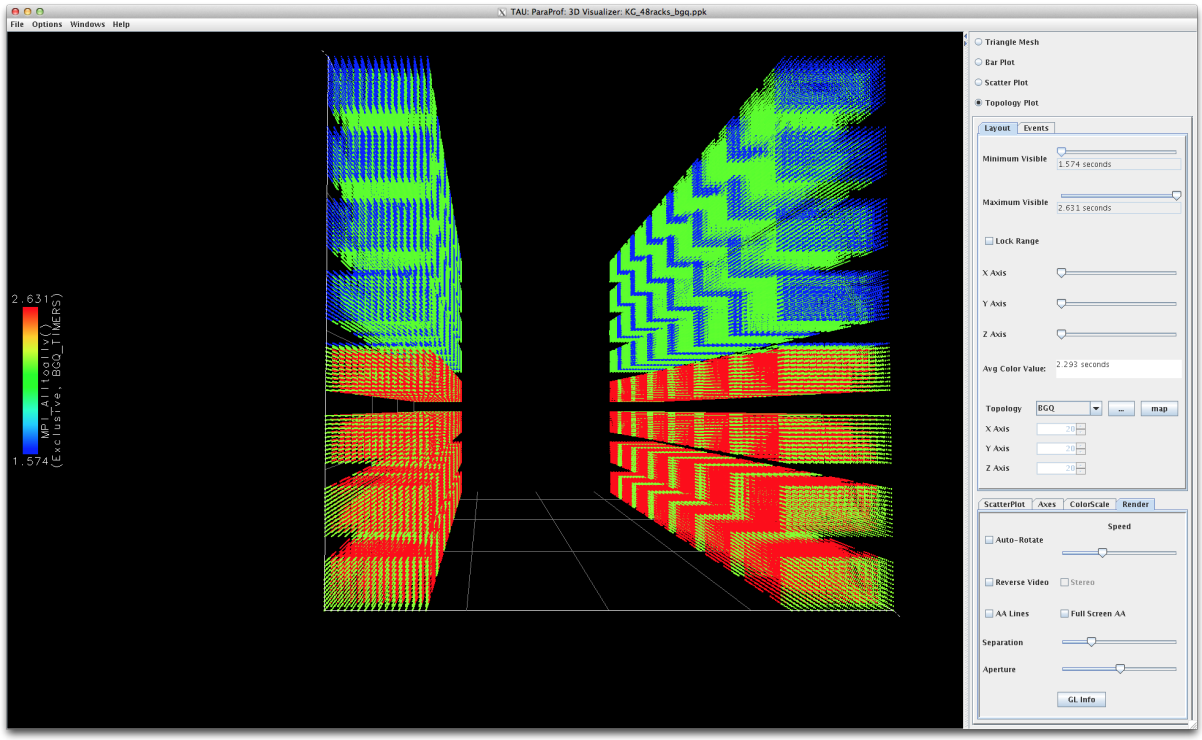


Figure 3.3: Visualization of performance data on top of the computing architecture topology (retrieved from the tool's – ParaProf – webpage). Even though three-dimensional, the visualization ranks low when compared to today's top-of-the-art graphic programs.

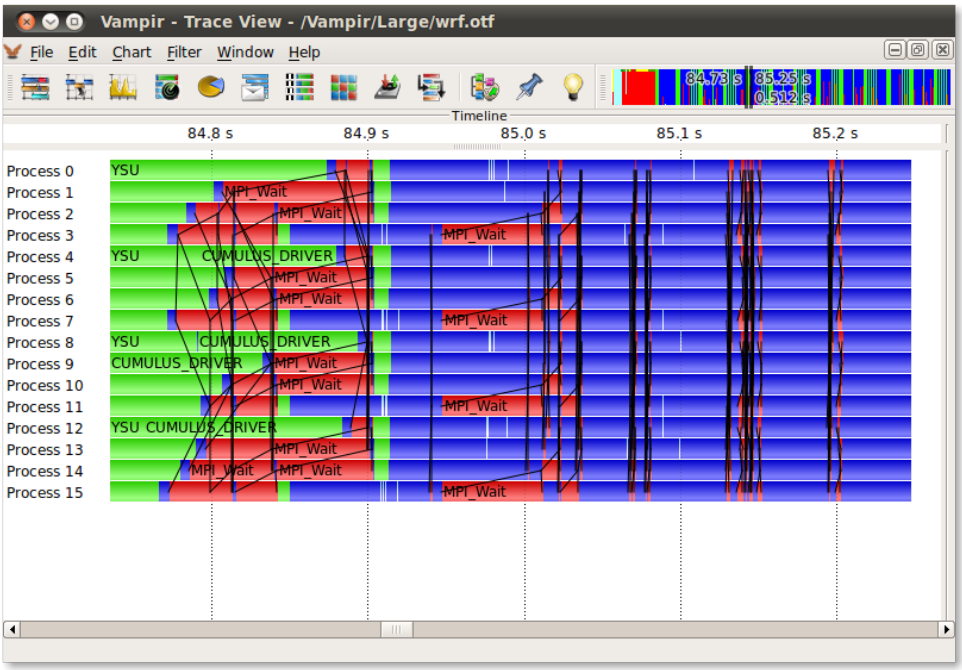


Figure 3.4: Example of the visualization of messages exchanged between ranks during the parallel execution of a code (retrieved from [3]). In an output limited to two dimensions, like this, it is hard to encode e.g. the distinction between messages coming from processes located within the same compute node (where an arbitrary receiver runs) from those coming from processes located in another compute node.

point / cell of the grid) plus the performance data, in separate files, with separate formats and to be read by separate visualization programs. This is illustrated in Figure 3.5 below.

In situ tools, on their turn, also function as optional add-ons to the original code and can be activated upon request, by means of preprocessor directives at compilation stage. The simulation results will then include the source code's native outputs 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 (like before). This is also illustrated in Figure 3.5.

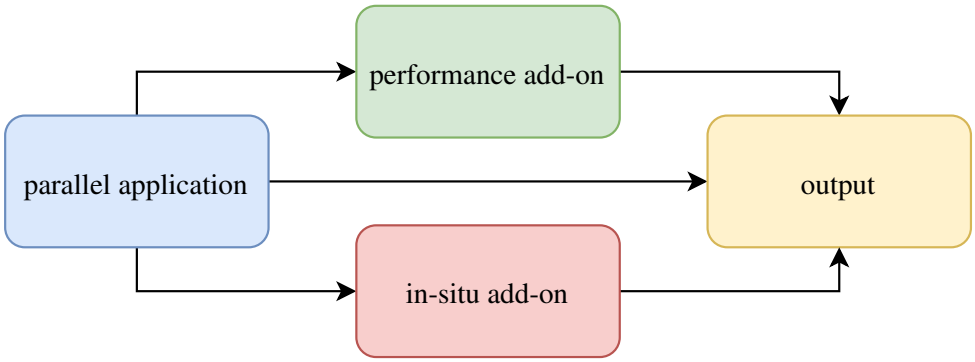


Figure 3.5: Schematic of software components for parallel applications.

So far, both add-ons have existed and been applied separately; but, in this thesis, the feasibility of merging them will be investigated. 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).[3] Second, by using the advanced functionalities of dedicated visualization software for the purpose of performance analysis. Figure 3.6 illustrates the idea.

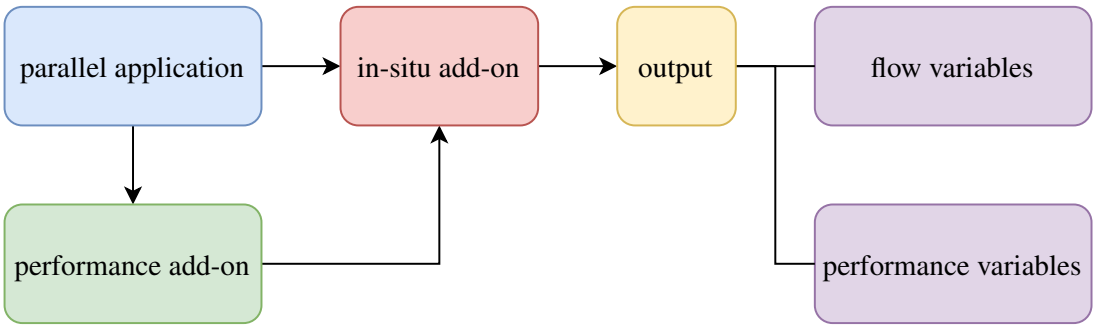


Figure 3.6: Schematic of the software components for a combined add-on.

A design requirement is that the combined solution must be easily applicable on the source code, yet without becoming a permanently required component. In other words, it needs to be "activatable" on demand, as it is the case for each of its constitutive parts (*performance measurement* and *in situ processing*). The aim will be to show performance data on top of the simulation's geometry itself, as well as on an abstract representation of the computing architecture. All that in a three-dimensional space and totally integrated with the time-stepping of the CFD solver.

[3] Both collect or "steal" data from the parallel application and transfer it out via a side channel.

4 Related Work

The necessity of bringing together the branches of *performance analysis* and *visualization* has already been identified by the scientific community [20, 40]. Bremer et. al ([20], S. 2) synthesizes this need with the following words (highlights are ours):

> "Performance analysis is a subfield of computer science that, for many years, has focused on the development of tools and techniques to quantify the performance of large-scale simulations on parallel machines. There are now a number of widely used tools and APIs to collect a wide range of performance data at the largest scales. [...] The success of these tools has created a new challenge: the resulting data is too large and too complex to be analyzed in a straightforward manner. *Existing tools use only rudimentary visualization and analysis techniques.* They rely on users to infer connections between measurements and observed behavior. The raw data is abstract and unintuitive, and it is often poorly understood as much of the hardware details are undocumented by vendors. Automatic analysis approaches must be developed to allow application developers to intuitively understand the multiple, interdependent effects their algorithmic choices have on the final performance. [...] *While some early attempts at including visualizations in performance tools have been proposed, these are rudimentary at best and have not found widespread adoption.*"

Let us now look closer into some of these attempts. Schulz et al. [59] was one of the first ones to map performance data to the simulation's geometry. In their work, an ablator drives a shock into an aluminum section containing a void. Figure 4.1 shows performance measurements (floating point operations per second in *d*, cache misses in *e*) together with flow variables (density in *b*, velocity in *c*). It is clear from the picture how the physics of the phenomenon impacts on the performance counters, a correlation that could not be perceived by plotting them e.g. in a line chart (as shown for Flops in *f*). However, the mapping was limited to a two-dimensional, rectilinear grid (far from the ones found in typical engineering problems).

Vierjahn et al. [67] managed to deal with three-dimensional grids. The result is shown on Figure 4.2 and displays the *severity* for each MPI rank (a concept defined inside the study) on top of the domain of a sheet-metal forming simulation. The color scale can be adjusted by the user and goes from black (0% severity) to light grey (100%) by default. Other variables are available for selection; they all appear on the parallel coordinates graph in the middle. The simulation domain (shown on top) can be manipulated by moving a virtual camera with five degrees of freedom using keyboard and mouse. But the work tried to develop 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. [35], on the other hand, did follow our approach: the performance tool *TAU* [60] was linked to the visualization software *VisIt* [23] to show performance data on top of the Earth's oceans

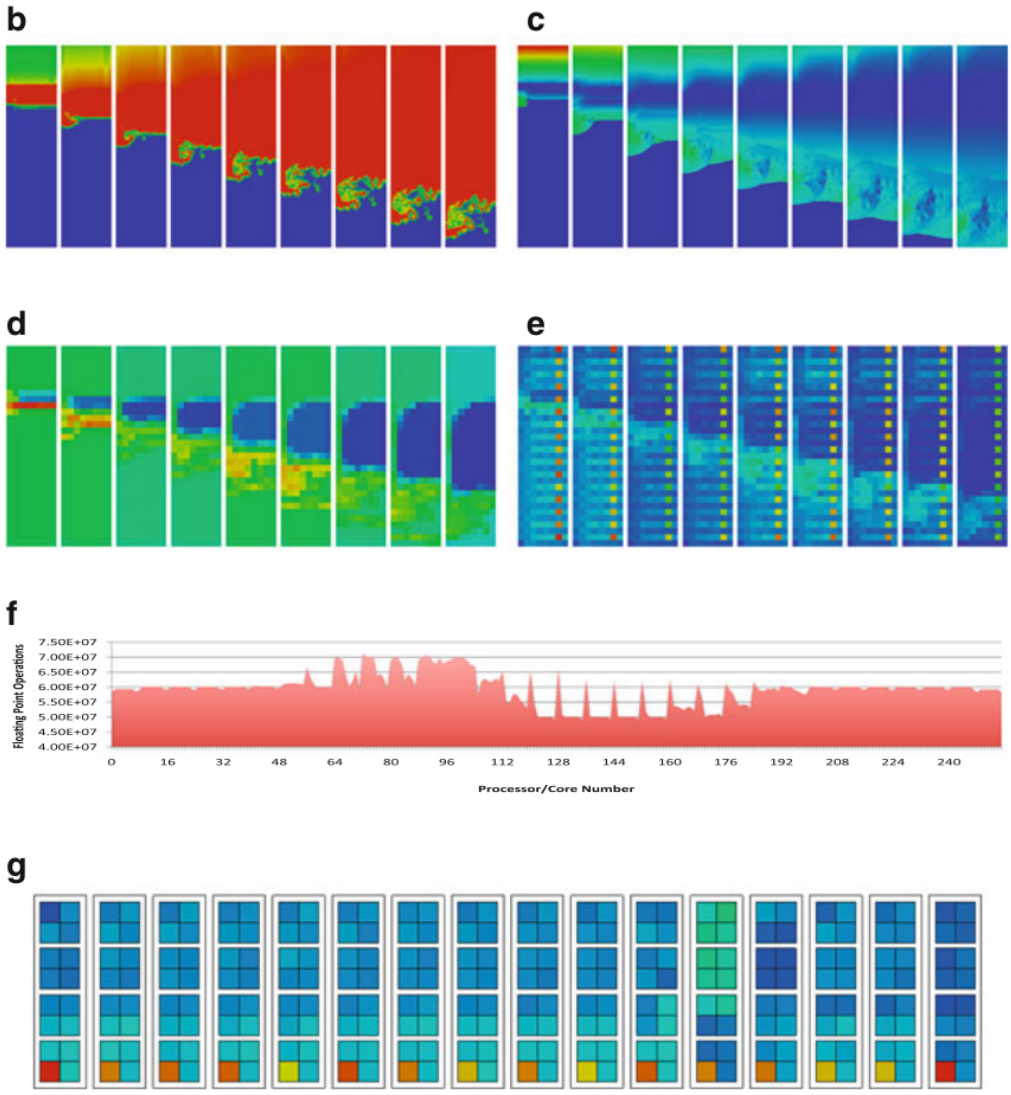
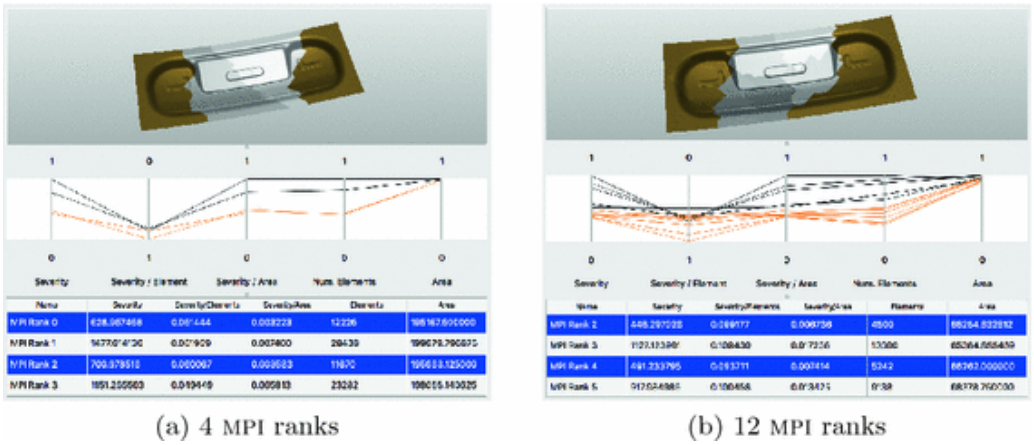


Figure 4.1: Visualization of flow variables (density in *b*, velocity in *c*) together with performance data (floating point operations per second in *d*, cache misses in *e*) in a CFD simulation of an ablator driving a shock into an aluminum section containing a void (retrieved from [59]). It is clear how the physics of the phenomenon impacts on the performance counters, a correlation that could not be perceived by plotting them e.g. in a line chart (as shown for Flops in *f*). *a* is just a diagram of the measurements collected for each type of mapping and was left out; *g* will be explained later.

in a climate simulation. The result is illustrated on the Figure 4.3. They confirm the importance of matching performance data to the simulated space: the original grid partitioning (shown on the left-hand side pictures) would create zones of very high computational time (third row, from top to bottom) in deep ocean blocks, hence zones of high waiting time (last row) in coastal blocks. However, the linking between the performance methods and the in situ ones required writing a new VisIt file format reader. To reproduce these results would require the effort of manually recreating such interfaces for each different (CFD) simulation code, what is undesirable.

Husain et al. [37] followed a similar path, also using VisIt as visualization tool, but MemAxes [28] as performance measurer – a software which does not seem to have a website, what impacts on its



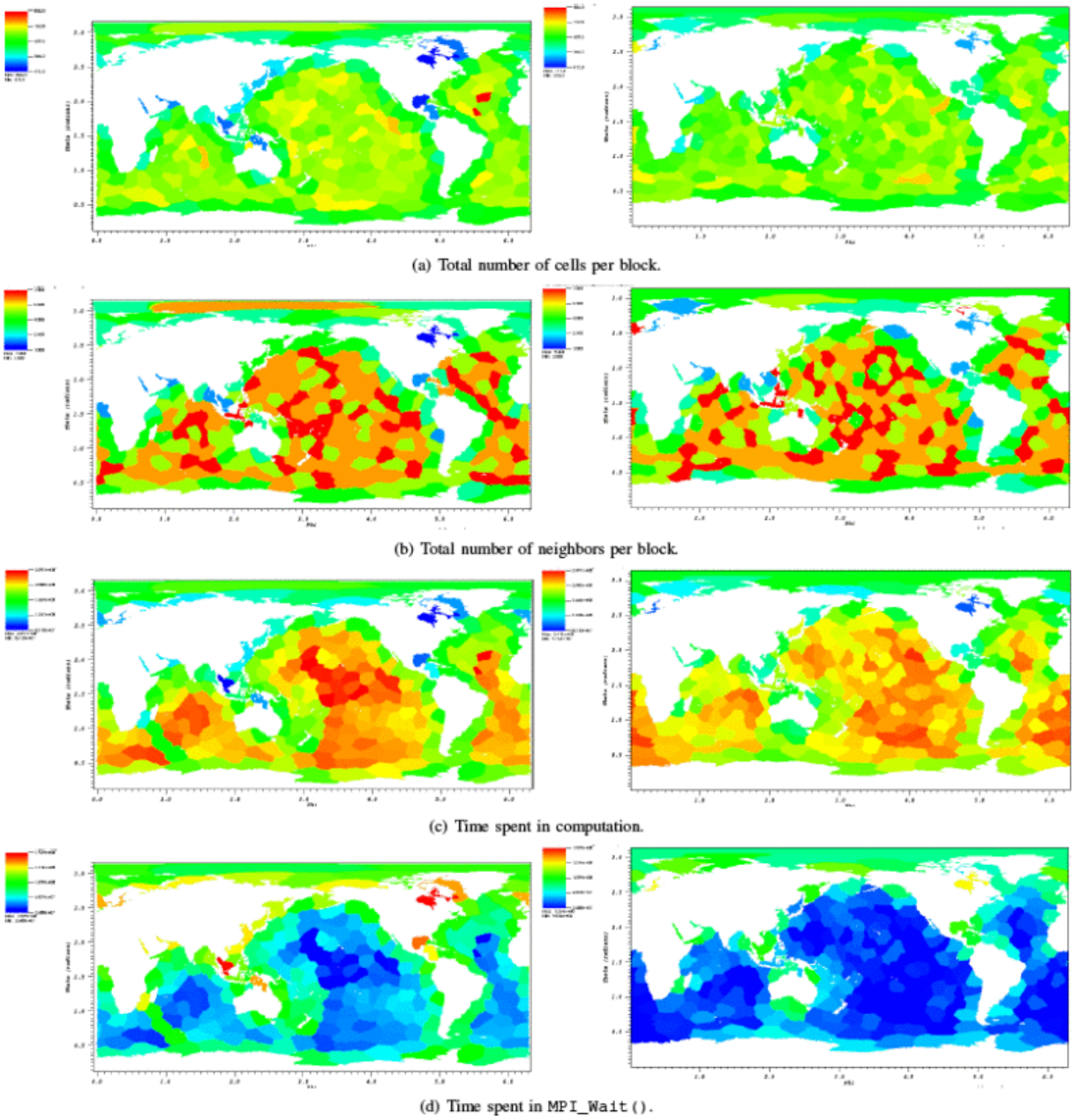
(a) 4 MPI ranks (b) 12 MPI ranks

Figure 4.2: Visualization of performance data – the *severity* for each MPI rank (a concept defined inside the study) – on top of the geometry of a sheet-metal forming simulation (retrieved from [67]). The color scale can be adjusted by the user and goes from black (0% severity) to light grey (100%) by default. Other variables are available for selection; they all appear on the parallel coordinates graph in the middle. The simulation domain (shown on top) can be manipulated by moving a virtual camera with five degrees of freedom using keyboard and mouse.

availability. Their results can be seen on Figure 4.4 below: over an automated mesh refinement (AMR) grid, it is represented, on the left-hand side, latency; and on the right-hand side, average latency per access. A logarithmic color scale is used in both cases, to allow the user to identify lower latency cells. The results are indeed valuable, as they permit the correlation between latency distribution and the physics of the simulation; however, the grid (of rectilinear type) is far simpler than those found on typical CFD problems. On the other hand, it was necessary to modify the source code of both tools involved (VisIt and MemAxes), what is again undesired.

Similar hurdles can be encountered in the work of Wood et al. [71]: the application needs to be first enveloped by a multipurpose framework (SOSflow) [70] and then linked with a non widely known in situ infrastructure (Ascent) [47] in order for performance data to be shown on simulation geometries which are comprised of rectilinear grids, as shown on Figure 4.5: the mesh is partitioned in 512 ranks, spread across 32 compute nodes. Three metrics are shown – from top to bottom: user CPU ticks, system CPU ticks and bytes read – across four stages – from left to right: cycles number 50, 250, 500 and 710 – of the simulation.

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 (non parallelepipedal) CFD meshes: the required user effort precluded it (their respective papers do not list what changes in source code were needed in order for their methods to work). The state-of-the-art will then be advanced 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, and which can output the results both on the simulation's geometry and on graphical representations of the computing architecture. An all-in-one tool, such that it will be tested against industry-grade CFD codes, simulating real-life engineering problems.



(a) Total number of cells per block.

(b) Total number of neighbors per block.

(c) Time spent in computation.

(d) Time spent in `MPI_Wait()`.

Figure 4.3: Visualization of performance data on the simulation's geometry (retrieved from [35]). The left-hand side pictures refer to the original grid partitioning, the right-hand side to a repartitioning using a special algorithm (for better load balance). Notice how the original partitioning would create zones of very high computational time (third row, from top to bottom) in deep ocean blocks, hence zones of high waiting time (last row) in coastal blocks. Such revelations demonstrate the importance of matching performance data to the simulation's geometry: there is a bias towards overloading deep ocean areas in the original partitioning scheme, what is probably related to the physics of the involved phenomena.

Moving on to a new topic, with regards to plotting performance data sorted by the simulation's time-step, Isaacs et al. [38] got close to it, by clustering event traces according to the self-developed idea of *logical time*, "inferred directly from happened-before relationships". The results of such approach can be seen on Figure 4.6: events are represented by boxes, colored by their wall-clock delay. The top part shows a logical timeline of the simulation, whereas the bottom part a clustered logical timeline. Each row on

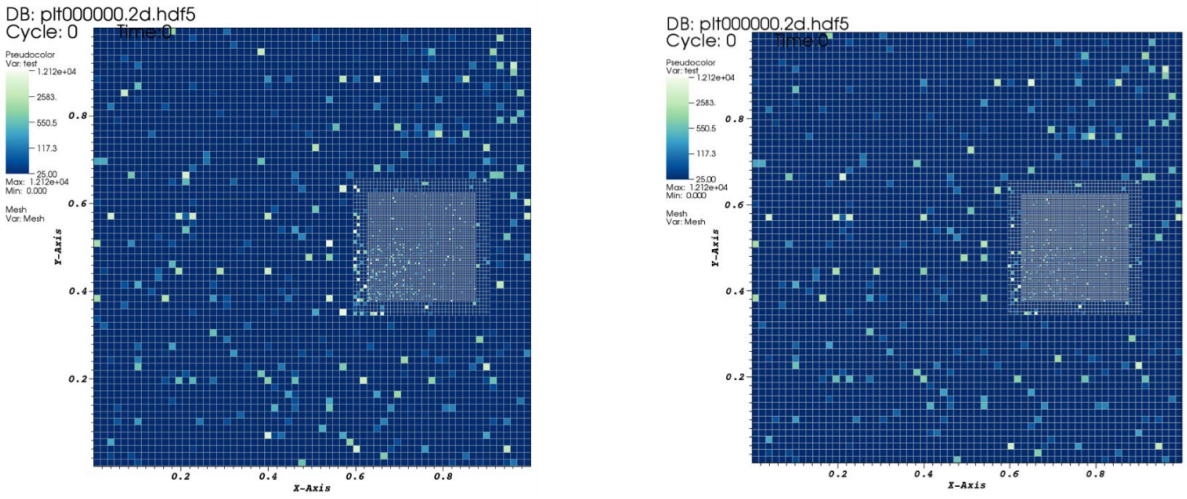


Figure 4.4: Visualization of performance data on the simulation's geometry (retrieved from [37]). Over an automated mesh refinement (AMR) grid, it is represented, on the left-hand side, latency; and on the right-hand side, average latency per access. A logarithmic color scale is used in both cases, to allow the user to identify lower latency cells. The results are indeed valuable, as they permit the correlation between latency distribution and the physics of the simulation; however, the grid (of rectilinear type) is far simpler than those found on typical CFD problems.

the top part refers to one parallel rank; messages sent between them are represented by the black lines. The idea of logical time indeed reveals communication patterns and facilitates the understanding of the program's structure. It represents 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. The state-of-the-art will be advanced in this thesis by showing the time-step itself.

Finally, when it comes to graphical representations of the computing hardware, there has also been attempts at the literature to improve the visualization produced by ParaProf (shown on Figure 3.3 above). Schulz et al. [59] did that in their work, plotting cache misses on top of the machine topology (visualization *g* on Figure 4.1). In the figure it becomes clear that the higher values experimented by some ranks are related to which CPU core was allocated to them, the same one in every node being more problematic than the others. Such correlation would not have been reached without the mapping to the machine topology; however, the output is limited to a two-dimensional space.

The lack of the advanced visualization features of dedicated graphic software, like three-dimensionality, was indeed noticed by the literature. Theisen et al. [63] addressed this bottleneck by combining multiple axes onto two-dimensional views, resulting on visualizations as seen on Figure 4.7: the multi-axes view has multiple dimensions hierarchically represented along a single horizontal or vertical axis, in a concept similar to those of memory representation of multidimensional arrays: always linear, regardless the number of dimensions of the stored array. The authors' idea was to display all planes along selected dimensions within a torus network. The result of their work is undeniably useful; but without true three-dimensionality, the multiplicity of two-dimensional planes overlapping each other can become cumbersome and preclude the understanding of the results.

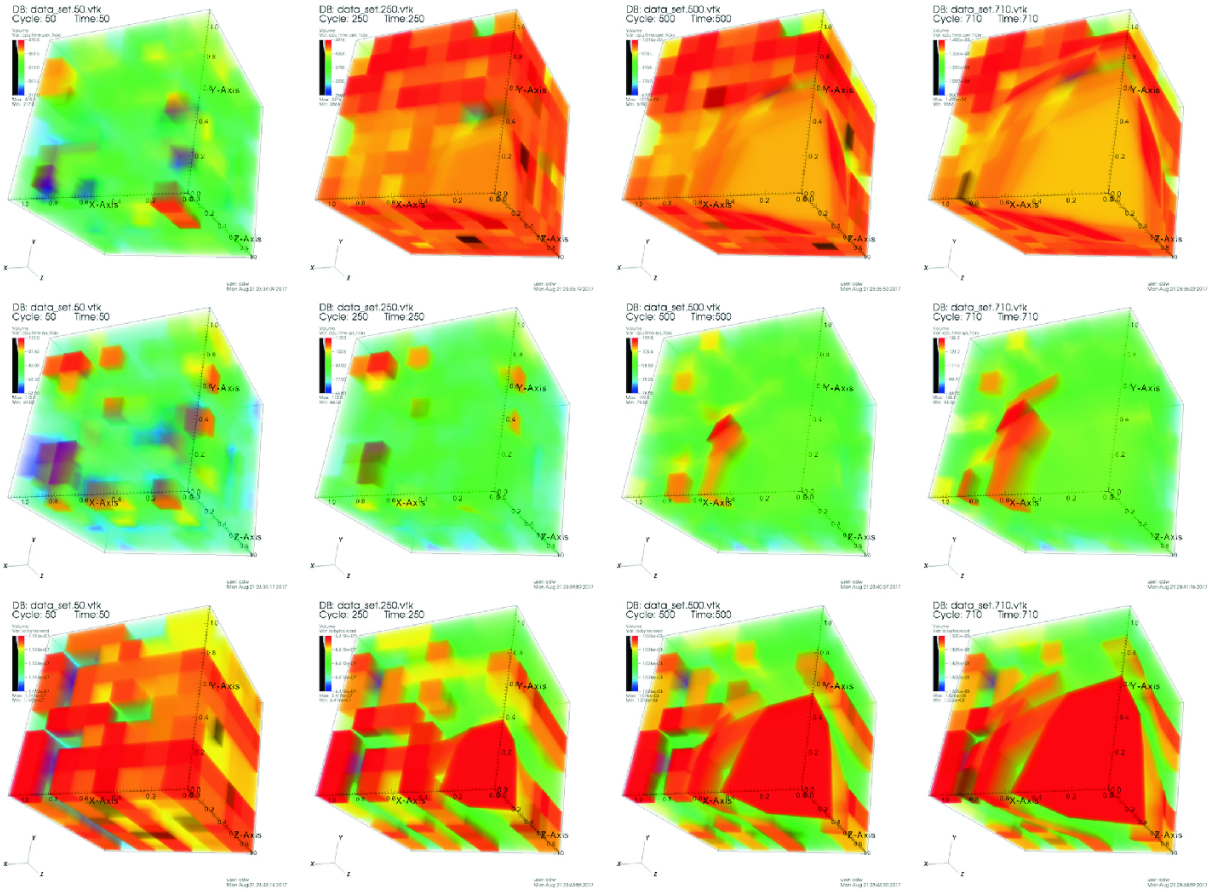


Figure 4.5: Visualization of performance data on the simulation's geometry (retrieved from [71]). The mesh is partitioned in 512 ranks, spread across 32 compute nodes. Three metrics are shown – from top to bottom: user CPU ticks, system CPU ticks and bytes read – across four stages – from left to right: cycles number 50, 250, 500 and 710 – of the simulation. Just like in the previous case, the results are valuable, but limited to a rectilinear grid.

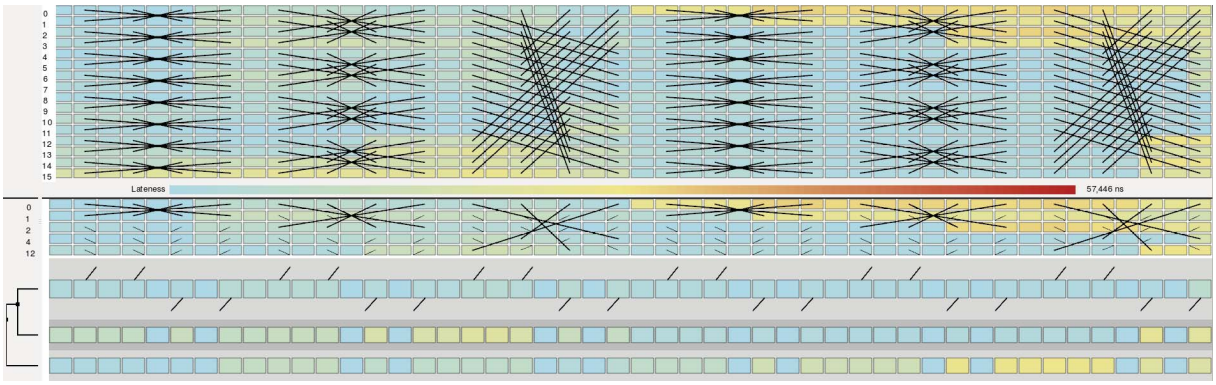


Figure 4.6: Plotting of event traces sorted by a *logical time* within the simulation (retrieved from [38]). Events are represented by boxes, colored by their wall-clock delay. The top part shows a logical timeline of the simulation, whereas the bottom part a clustered logical timeline. Each row on the top part refers to one parallel rank; messages sent between them are represented by the black lines. The idea of logical time indeed reveals communication patterns and facilitates the understanding of the program's structure. However, developers of CFD codes – many of them from a mechanical engineering background – would still have difficulties to understand the results, as they are not yet organized by the simulation time-step they are used to.

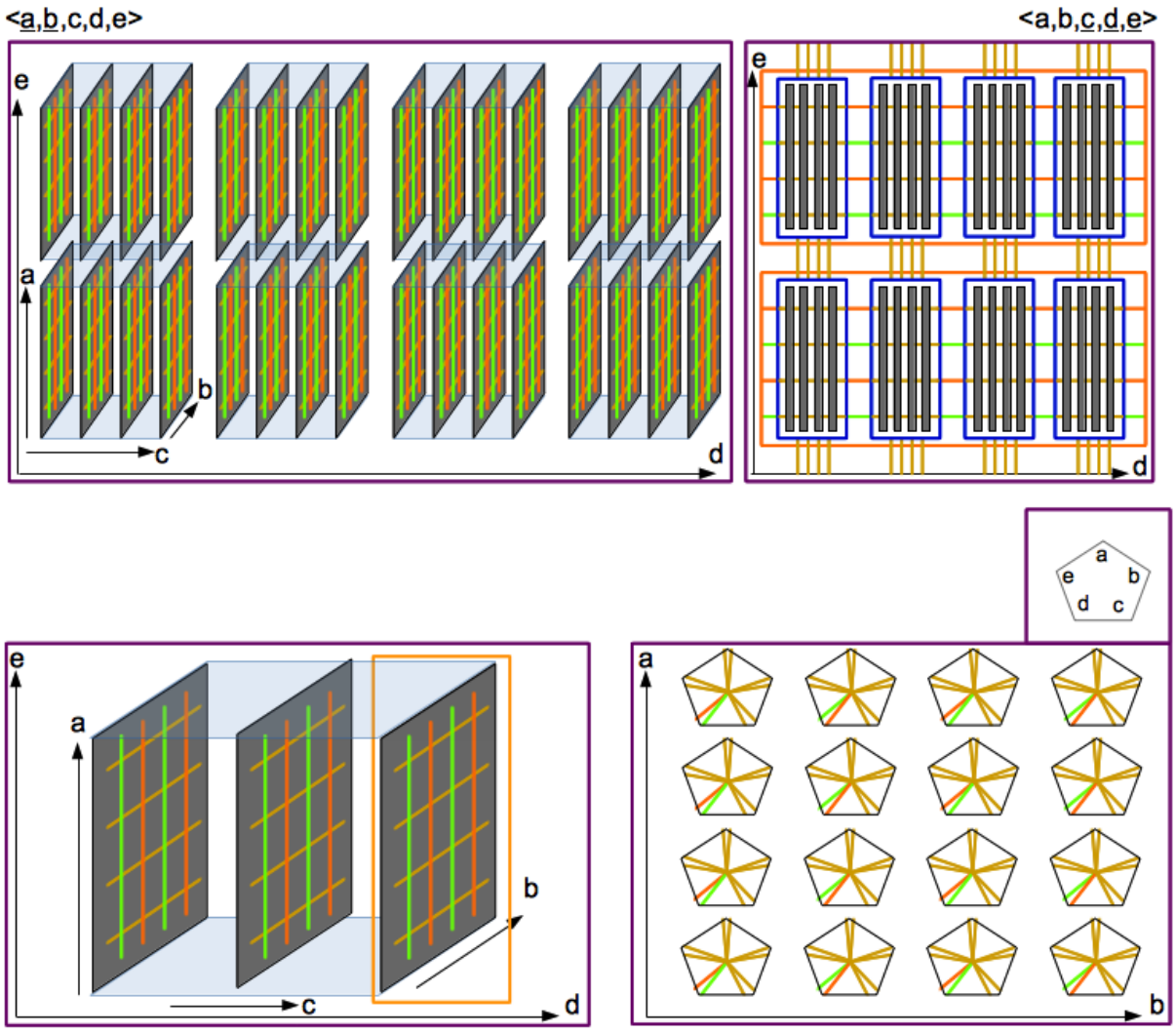


Figure 4.7: Three-dimensional representation of cluster by means of combining multiple axes onto two-dimensional views (retrieved from [63]). The multi-axes view has multiple dimensions hierarchically represented along a single horizontal or vertical axis, in a concept similar to those of memory representation of multidimensional arrays: always linear, regardless the number of dimensions of the stored array. The authors' idea was to display all planes along selected dimensions within a torus network. The result of their work is undeniably useful; but without true three-dimensionality, the multiplicity of two-dimensional planes overlapping each other can become cumbersome and preclude the understanding of the results.

Isaacs et al. [39] and Schnorr et al. [56], on the other hand, went for the path of writing a new three-dimensional visualization tool (shown on Figures 4.8 and 4.9 respectively). Figure 4.8 shows a 3D torus network represented simultaneously in two (left) and three-dimensions (right). *Compute nodes* are colored by their sub-communicators, *links* by the number of packages sent over them. It is possible to highlight some nodes, which then appear slightly bigger in the 2D visualization and more opaque in the 3D view. Figure 4.9, on its turn, shows a three-dimensional representation of resources being used by a parallel application. The columns can represent either processes or entire nodes; the lines represent communication between them. The columns are placed on the grid according to the topological distance between the resources within the machine architecture. This last tool is closer to our approach: only the

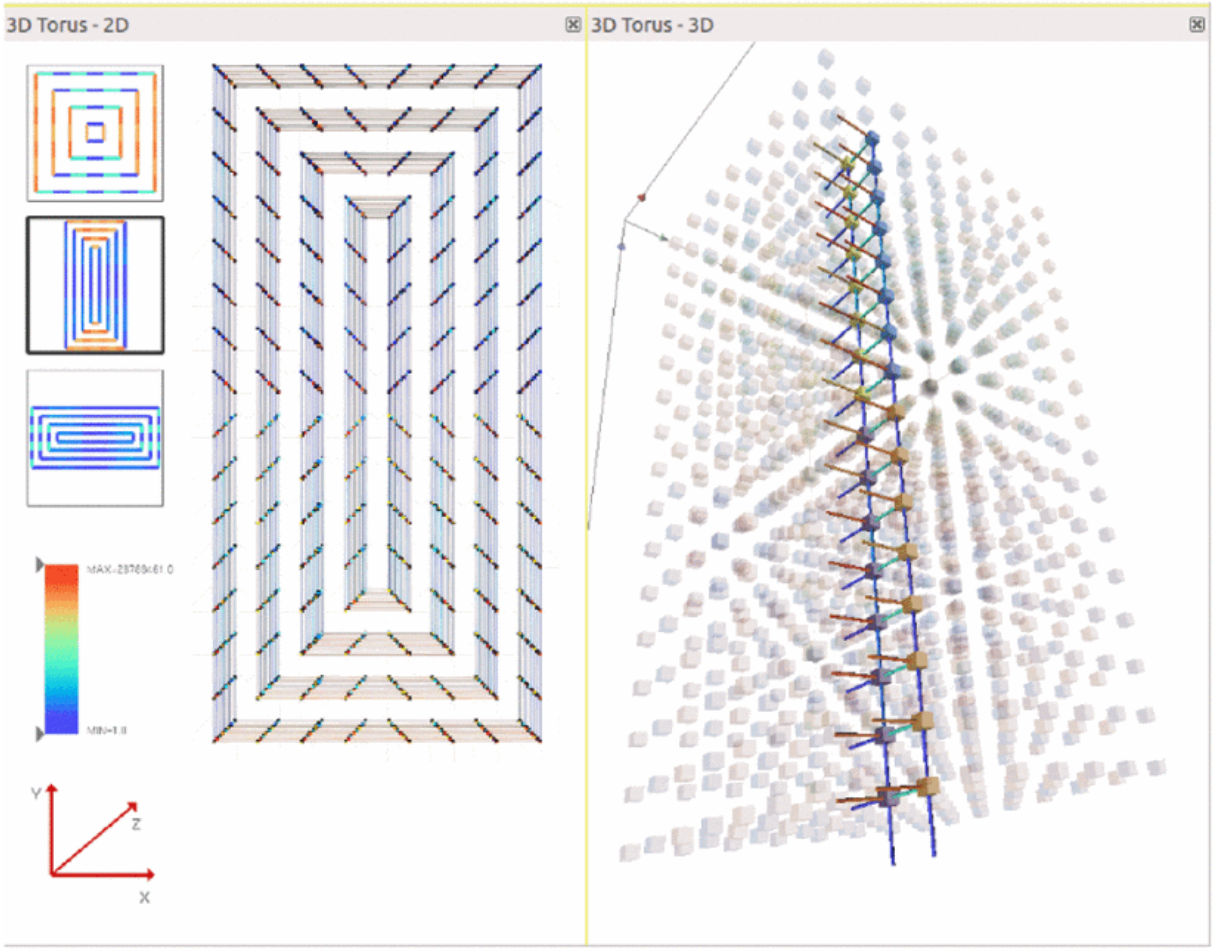
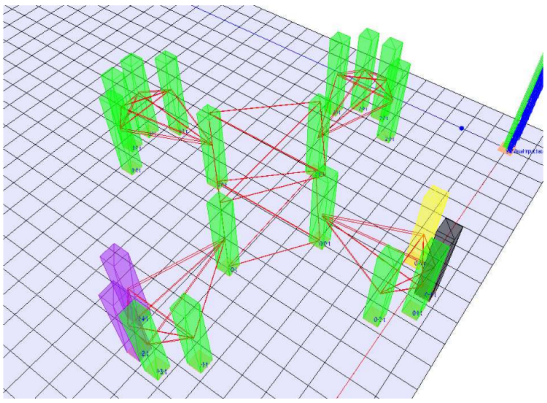
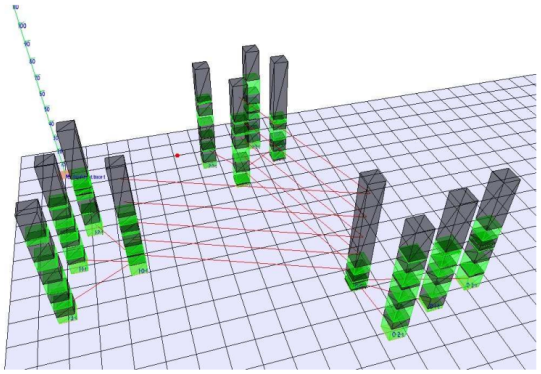


Figure 4.8: A 3D torus network represented simultaneously in two (left) and three-dimensions (right) (retrieved from [39]). *Compute nodes* are colored by their sub-communicators, *links* by the number of packages sent over them. It is possible to highlight some nodes, which then appear slightly bigger in the 2D visualization and more opaque in the 3D view. This novel tool is indeed promising, but its visualization options (quality of rendering etc.) are still behind today's top-of-the-art graphic programs.

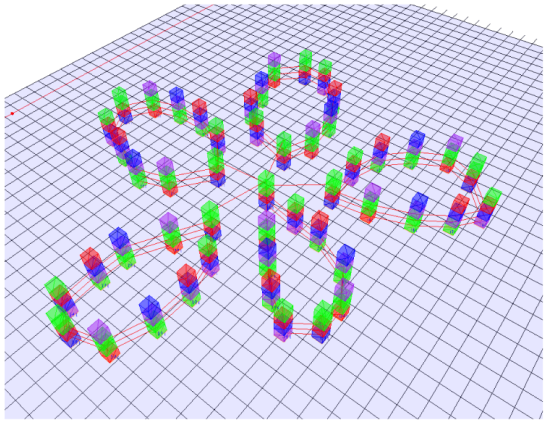
resources being used by the parallel application are drawn on the visualization. In all cases, however, the outputs are limited when compared to the current state-of-the-art of computer graphics manipulation, as one could expect from approaches which attempt to recreate those capabilities from scratch. In this thesis, the already established graphic programs will be used to portray the computing architecture running a simulation code; in other words, these programs – originally made to display simulation geometries – will be given a novel usage they had not been assigned to before. This path also has the benefit that developers of numerical (CFD) codes do not need to learn new software in order to visualize the performance of their application, but just stick with the programs they are already used to use (in order to visualize the flow solution).



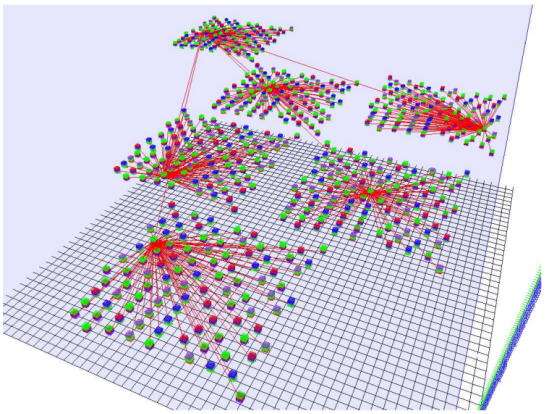
(a) Work stealing view of a Fibonacci parallel application with 20 processes in four sites of a Grid.



(b) Visualization of an application executing in 3 different sites, using 4 nodes per site.



(a) View of an application with 50 processes, executing in 5 sites with 10 processes each.



(b) View of an application with 600 processes, divided in 6 sites with 100 processes each.

Figure 4.9: Three-dimensional representation of resources being used by a parallel application (retrieved from [56]). The columns can represent either processes or entire nodes; the lines represent communication between them. The columns are placed on the grid according to the topological distance between the resources within the machine architecture.

4.1 Aspects to Consider

Performance analysis is about unveiling correlations: only when all aspects are considered together, maximum performance and scalability become reachable. Hence it is important to go through those aspects and describe which of the existing tools (if any) covers them.

Pure computation. The most basic aspect to consider in any performance analysis is the purely computational one: which code functions are taking longer to execute and has that something to do with how often they are executed. This information can be retrieved by many code instrumenters, among them the tool which will be used in this work, *Score-P*, in association with the GUI *Cube* (for visualization purposes; see Sec. 5 below).

Load imbalances. The second aspect to observe is also the first one that comes into consideration when the simulation is ported to run in parallel: which ranks (or threads, when using OpenMP) are taking longer to execute each specific region of the code (and has that to do with how often each process is executing them). This information can also be retrieved by Score-P in association with Cube and often is related to the partitioning of the grid among the parallel processes; but, as seen above, no tool is currently able to make the connection between simulation mesh and performance metrics.

Communication. Another aspect that arises from parallelizing the simulation has to do with the communication among ranks: which of them are sending/receiving more data and doing that more often. Communication is a downside of parallelization: it is needed when running the code in parallel, but it is not desired from the point-of-view of simulation outputs (the information one wants to extract from its code); hence it should be kept to a minimum. This information can be retrieved by Score-P in association with Cube and also *Vampir* (another visualization GUI; see Sec. 5 below).

Synchronization. The communication-equivalent to *load imbalances*, synchronization has to do with different ranks finishing their communication tasks at different speeds. This leads to the fastest processes having to wait for their slower peers to conclude their work, what represents wasted computational time and should therefore be avoided. This information can be retrieved by Score-P in association with Vampir. For more details on the topic, the reader is referred to the doctor thesis at [45].

Hardware counters. Comprised of metrics like floating point operations per second (Flop/s), cache misses in each of the hardware topological hierarchies etc., *hardware counters* constitute another aspect which helps to identify performance bottlenecks. They can be retrieved by Score-P and visualized with Cube. For more details on the topic, the reader is referred to the doctor thesis at [41].

Accelerators. *Graphics Processing Units* (GPUs) are specialized computer processors, made originally to tackle the demands of real-time high-resolution 3D graphics, but which can also be used to accelerate the tasks of general purpose processing (for which normal CPUs would be less efficient) [54]. They naturally impact on the performance of the code execution, hence need to be taken into consideration by the tools too. Score-P and Vampir are able to make such correlations visible. For more details on the topic, the reader is referred to the doctor thesis at [25].

I/O. The performance optimization of input/output operations has been object of extensive research in the last years. Albeit not being strictly mandatory, they remain a convenient way to put data into or to extract data out of a simulation. Their integration into performance tools is still a work in progress, but Score-P and Vampir are already able to display something on that regard. For more details on the topic, the reader is referred to the doctor thesis at [64].

5 Methodology

5.1 Prerequisites

The goals aimed by this research depend on the combination of two basic, scientifically established methods: *performance measurement* and *in situ processing*. On the following sections, the tools chosen in each of these categories shall be described, which will later be combined into the proposed solution.

5.1.1 Performance Analysis – introducing Score-P

Score-P [44] is an open-source "highly scalable and easy-to-use tool suite for profiling, event tracing, and online analysis of HPC applications" [tool's website]. When applied to a source file's compilation, Score-P automatically inserts probes between each code "region"[1], 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 within the simulation. Functions related to communication between ranks have extra details recorded, like the addressee of a specific message, the amount of bytes sent etc. The tool is applied by simply placing the word `scorep` before 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, whose detection 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 [57]. The combined solution proposed by this paper takes actually the form of such a plugin.

HPC analyses tools usually produce either *performance profiles* or *event traces*. In the case of Score-P, they are:

- performance profiles in the Cube4 format, to be visualized with *Cube*; and
- parallel event traces in the OTF2 format, to be visualized with *Vampir*.

On the following topics, each of them shall be presented in detail.

[1] 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.

5.1.1.1 Profiling mode – visualization in Cube

Cube [55] is 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]. It graphically outputs the performance data generated by Score-P in *profiling* mode. Its GUI is shown below:

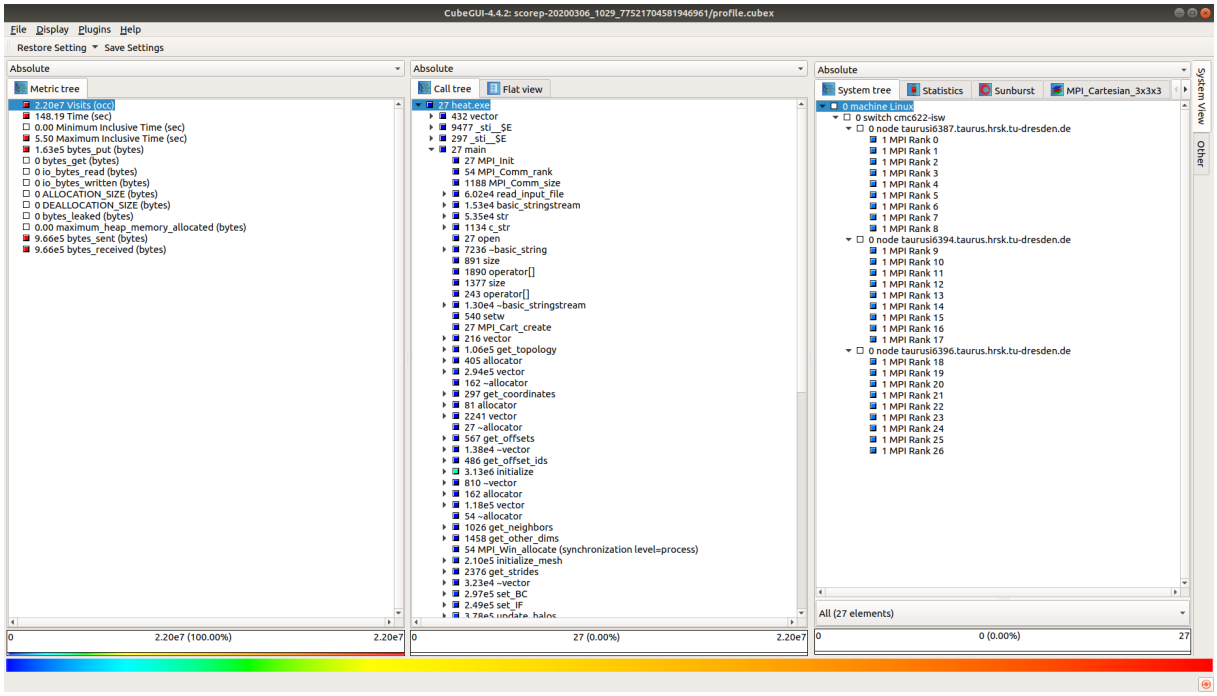


Figure 5.1: Graphic user interface of performance data visualization tool *Cube*.

On the left side column are the different metrics collected by the instrumenter (in this case, Score-P), the most important of them being the amount of executions and the total time spent inside each code region (as explained above). Notice that only the final numbers are available, i.e. after the simulation is finished – it is not possible to break those metrics down to a time step basis (to investigate e.g. if there are significant variations during the execution of the code). The mid column, on its turn, shows a tree view of the call path, by order of first invocation. It is possible to expand and collapse the children regions by clicking on the relevant parent. Finally, the right-hand side column breaks the measurements down to the *process* – in this case, since MPI is being used: *rank* – id of where they were collected. As in the previous column, here it is also possible to show / hide children data (ranks within the same compute node, compute nodes within the same network switch etc.).

Apart from this initial setting of views, there are many others available at the user discretion: flat view of the code regions (by alphabetical order), statistical displays of the measurements (box plot, violin plot), sunburst representation and a view of the cartesian topology of the ranks (when this MPI option was used), as shown on Figure 5.2 below. For the specific case of *rectilinear* CFD meshes, this view is equivalent to showing performance data on the simulation's geometry. But for all other cases – including the ones typical in CFD simulations, like airplanes' fuselage, airfoil and engine (where there is no cartesian topology within the MPI ranks), it is not possible to perform such visualization in Cube.

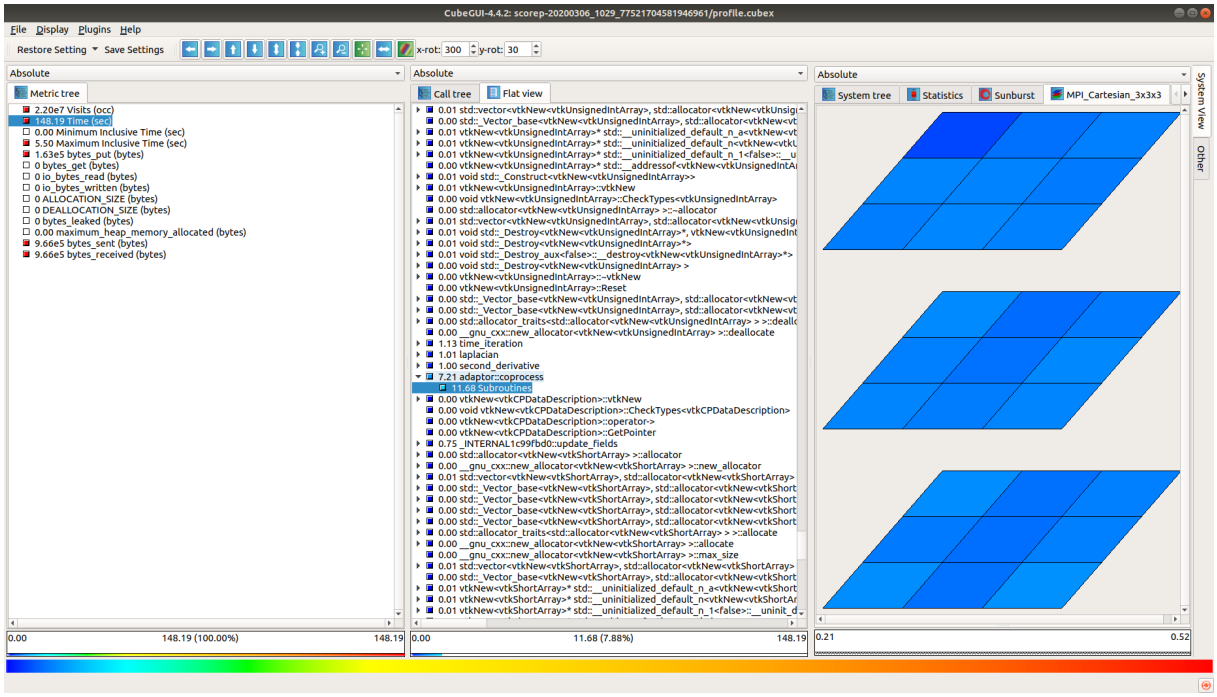


Figure 5.2: Other views of the performance data visualization tool *Cube*, including a representation of the cartesian topology of the MPI ranks (an option available inside MPI).

5.1.1.2 Tracing mode – visualization in Vampir

Vampir [43] is an "easy-to-use framework that enables developers to quickly display and analyze arbitrary program behavior at any level of detail" [tool's website]. It graphically outputs the performance data generated by Score-P in *tracing* mode. Its GUI is shown on Figure 5.3 below.

The left window displays a timeline of the code execution by each rank (in this example, 27 ranks were used). Code functions are colored by type: communication related ones (in this example, MPI) are red, computation ones (i.e. the core of the program itself) are green etc. The top right window, on its turn, shows a bar chart of the total time taken in each code function (the most time consuming ones shown above).

As opposed to Cube, here it is possible to manually identify the solver's time steps by zooming into the timeline view, as shown on Figure 5.4 below. However, that remains only on the visual side: the performance data itself is not marked with any time step tag, hence no further statistical analysis, comparisons etc. on that regard can be executed.

Finally, as shown on Figure 5.5, by further zooming into the timeline, it is possible to even see individual messages sent between ranks in a specific point during the simulation. However, given this is a two-dimensional view, it is hard to distinguish e.g. messages coming from ranks running under the same compute node from those coming from ranks running in other compute nodes.

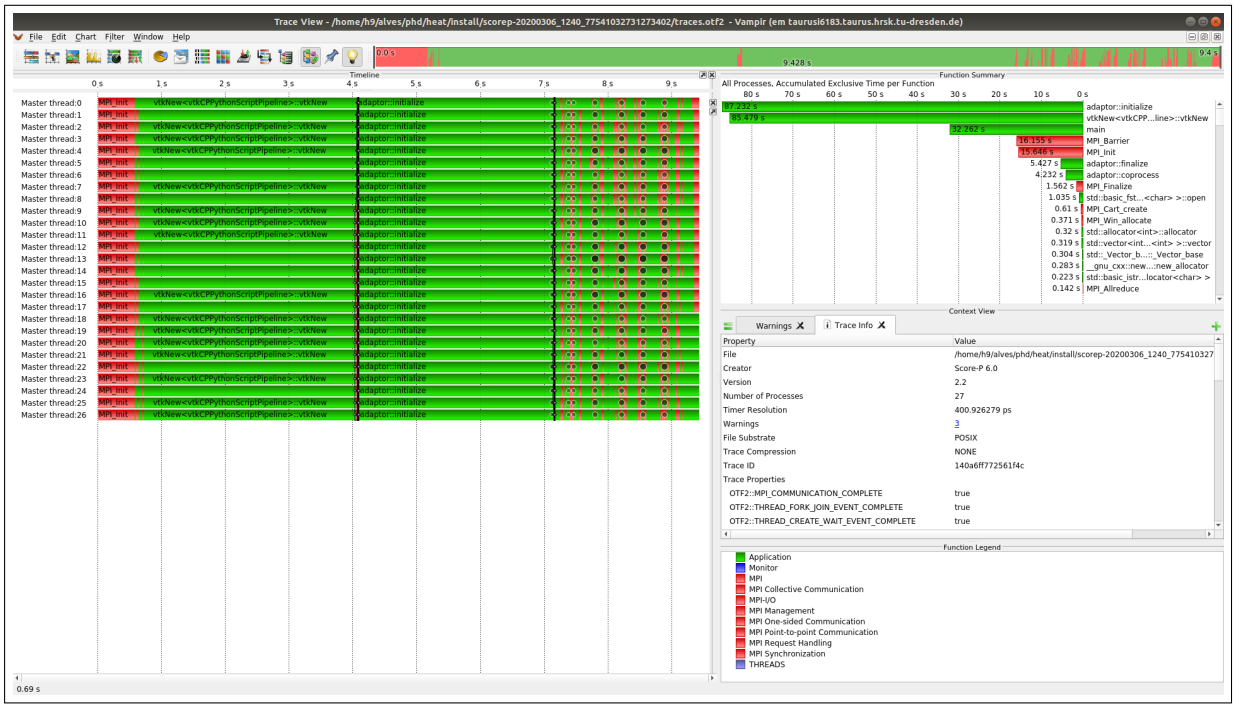


Figure 5.3: Graphic user interface of performance data visualization tool *Vampir*.

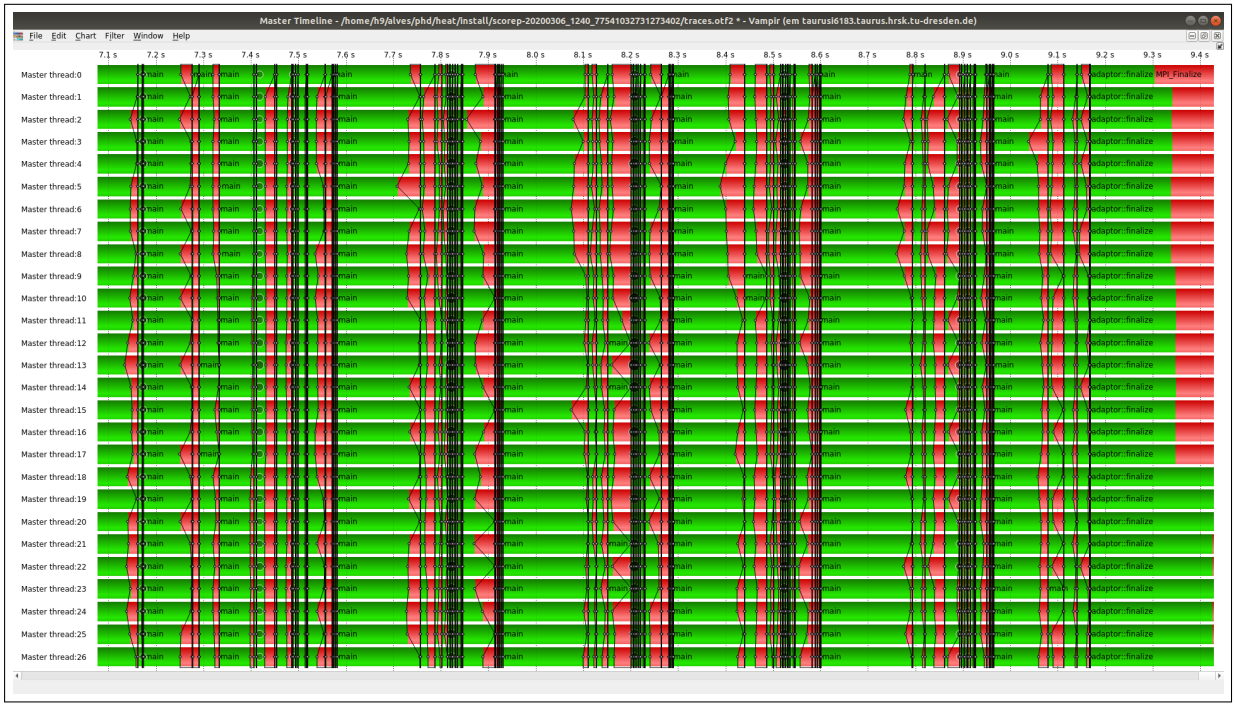


Figure 5.4: Zoomed view of the code execution timeline on performance data visualization tool *Vampir*.

Figure 5.5: Further zoomed view of the code execution timeline on performance data visualization tool *Vampir*, showing individual messages exchanged between ranks in a specific point during the simulation.

5.1.1.3 The substrate plugin API

It was said above that the proposed solution takes the form of a Score-P plugin; hence, here the correspondent API shall be briefly presented. Score-P offers two types of plugins: *metric plugins* "enable programmers to augment the event stream with metric data from supplementary data sources that are otherwise not accessible for Score-P" [57]; *substrate plugins*, on their turn, are more flexible and allow the programmer to define its own custom tasks for the events recorded by the instrumenter. This is the type of plugin adopted in this work.

Score-P instruments a simulation by adding triggers all around its source code: when a region is entered, when a region is exited, when this region is of a specific type etc. In the substrate plugin API, the programmer is then free to implement a routine that will be executed whenever one of those triggers is activated. This plugin uses six of them,[2] illustrated at Figure 5.6: the overall enter and exit region probes, used when measuring regions in the simulation code; plus four types of MPI calls, used when tracking communications during the execution of the program. These two features of the tool shall be presented in detail in Sec. 5.2 below.

5.1.2 In Situ Processing – introducing Catalyst

Catalyst [26, 10] is 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]. In order for it 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, illustrated in blue at Figure 5.7.

The functions *initialize* and *finalize* are straightforward and consist mostly of calling the correspondent peer in Catalyst; the *run* function, on the other hand, requires more programmer effort, as it will need to build the grid and the data structures to hold the flow properties (using the Catalyst methods available). The combination of these three functions forms the unit called Catalyst *adapter*, usually compiled as an independent library, which can then be optionally linked to the CFD program at build time. Such adapter

[2] Their full list can be found in file `SCOREP_SubstrateEvents.h`.

```
/* Register event functions */
static uint32_t get_event_functions(
                                    SCOREP_Substrates_Mode       mode,
                                    SCOREP_Substrates_Callback** returned
                                   )
{
  auto functions =        ( SCOREP_Substrates_Callback*)
                    calloc( SCOREP_SUBSTRATES_NUM_EVENTS,
                    sizeof( SCOREP_Substrates_Callback ) );

  if ( mode == SCOREP_SUBSTRATES_RECORDING_ENABLED )
  {
    functions[SCOREP_EVENT_ENTER_REGION] = (SCOREP_Substrates_Callback) enter;
    functions[SCOREP_EVENT_EXIT_REGION ] = (SCOREP_Substrates_Callback)  exit;
    functions[SCOREP_EVENT_MPI_SEND    ] = (SCOREP_Substrates_Callback)  send;
    functions[SCOREP_EVENT_MPI_ISEND   ] = (SCOREP_Substrates_Callback) isend;
    functions[SCOREP_EVENT_RMA_PUT     ] = (SCOREP_Substrates_Callback)   put;
    functions[SCOREP_EVENT_RMA_GET     ] = (SCOREP_Substrates_Callback)   get;
  }

  *returned = functions;
  return SCOREP_SUBSTRATES_NUM_EVENTS;
}
```

Figure 5.6: Plugin callbacks from the Score-P substrate plugin API. The two first calls are used when measuring regions in the simulation code; the four last calls are used when tracking communications during the execution of the program. These two features of the tool shall be presented in detail in Sec. 5.2 below.

works as an interface between the simulation code and Catalyst itself. It needs to be written in C++. It requires a Python input file, which sets parameters like the frequency of generation of output files (if any), whether live visualization is enabled etc. For instructions about how to write the adapter and the Python input file there are many examples within Catalyst's documentation.

Once implemented, the adapter allows the generation of post-mortem files and/or the live visualization of the simulation, through the program to be presented in the next topic.

5.1.2.1 ParaView

ParaView [4] is an open-source "software for manipulating and displaying scientific data" [tool's website]. It is built with *VTK* [58], an open-source "multi-platform data analysis and visualization application" [idem]. Both are made by the same company, *Kitware*. Its GUI is shown on Figure 5.8 below. Multiple visualization tabs can be opened simultaneously, sharing or not the view window, and displaying all types of information: from the simulation geometry to charts, graphs and even raw data tables. Tens of filters are available for the user to dig into the data; render views (i.e. display of spatially located information, like a CFD mesh) are fully zoomable, rotatable and pannable. A sample of the tool's capabilities can be found at `https://www.paraview.org/gallery/`.

```
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 5.7: Illustrative example of changes needed in a simulation code due to Catalyst (shown in blue). The code lines in violet will be explained later.

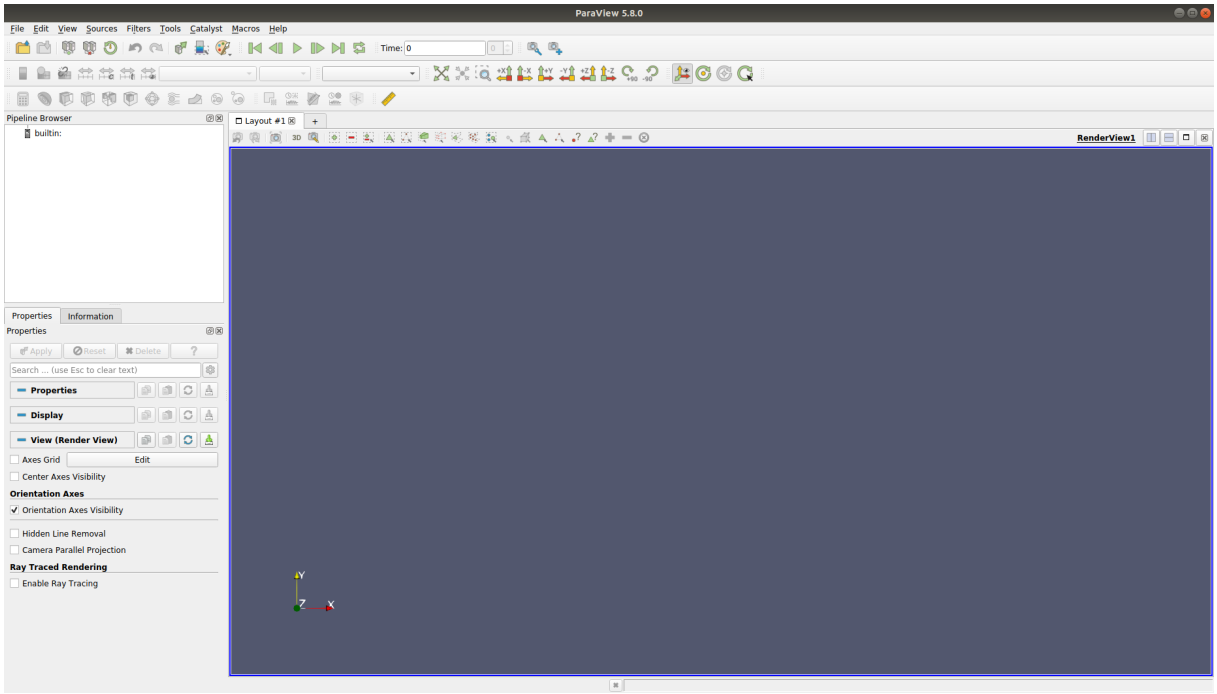


Figure 5.8: Graphic user interface of multipurpose visualization tool *ParaView*.

5.2 Combining the Tools

A Score-P plugin has been developed, which allows:

- performance measurements for an arbitrary number of manually selected code regions; and
- communication behavior (messages sent) between ranks;

to be mapped to:

- the simulation's original geometry, by means of its Catalyst adapter (*geometry mode*);
- a symbolic representation of the computing architecture's topology, by means of the plugin's own Catalyst adapter (*topology mode*).

It 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 Figure 5.7. As opposed to its peer in the Catalyst side, here the *run* call needs nothing more than the time-step number and value (in seconds, if desired). It is responsible, amongst other things, for running the Catalyst tasks pertaining to topology mode and preparing the plugin's data arrays for the next time-step. Finally, a call must be placed before each function to be *pipelined* (i.e. sent for visualization):

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

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

The above layout ensures that the desired function will be captured when executed at that specific source code location 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 when tracking communication between ranks, as the type of MPI call (i.e. whether it is MPI_Send, MPI_Isend, MPI_Put etc.) is automatically detected by Score-P at run-time.

When running the plugin in geometry mode, the user needs to add a small piece of code into its simulation's Catalyst adapter, in order for the plugin-generated variables to be pipelined (together with the traditional flow variables), as shown in Figure 5.9. For the part pertaining to the manually selected code regions, there are two vectors because for each region selected inside the simulation's code, the plugin will generate two variables, which correspond to the two basic measurements made by Score-P, as explained in Sec. 5.1.1 above. For the part pertaining to the communication, on its turn, there are also two vectors, as one will store the *amount of times* the correspondent MPI call was made (inside the time-step), whereas the other will store the total *amount of bytes* transported through those calls. The simulation used here (as an example) only calls MPI_Put inside the time-step, so only this type of call needs to be pipelined; but the plugin also supports MPI_Send, MPI_Isend and MPI_Get. Multiple of them can be pipelined together; for each there would be those two vectors, then. Finally, this intervention in the simulation's Catalyst adapter is not needed in topology mode, as in this mode the plugin is using its own Catalyst adapter (i.e. everything happens in the background).[3]

5.2.1 Understanding the Plugin

Now that what needs to be changed in the simulation's code due to the plugin has been presented, the focus will be placed on the plugin itself. Figure 5.10 is a scheme of its source code, comprised of only one translation unit: the main cxx file, which implements the six callbacks from the substrate plugin API (presented in Sec. 5.1.1.3 above); and which recursively includes all its headers. The procedures related to the manually selected regions (inside the main CFD code) and to the communications are defined in their respective files (respectively at the right and left side of the diagram). `interface.hpp` contains the definition of the three basic plugin calls (initialize, run and finalize), which are additionally declared in `simulation_interface.h` (for compatibility with simulation codes written in C or C++). During the initialization routine, the plugin's input file is read; therefore `interface.hpp` includes the header with the relevant function definitions (`initialization.hpp`). All include paths meet then at `bottom.hpp`, which is comprised solely of global variables and the headers needed by them (mostly of C and C++ standard libraries).

[3] 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

```
#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) + " : tick").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 5.9: Illustrative example of the addition needed in the simulation's Catalyst adapter due to the plugin. Here only one of the supported communication calls is requested (namely, MPI_Put), just for example purposes.

Some of the global variables in `bottom.hpp` are worth a special consideration. The first of them is the class with the definition of the parameters associated to the selected simulation code regions, seen below:

calls seen on Figure 5.9 are free from those restrictions.

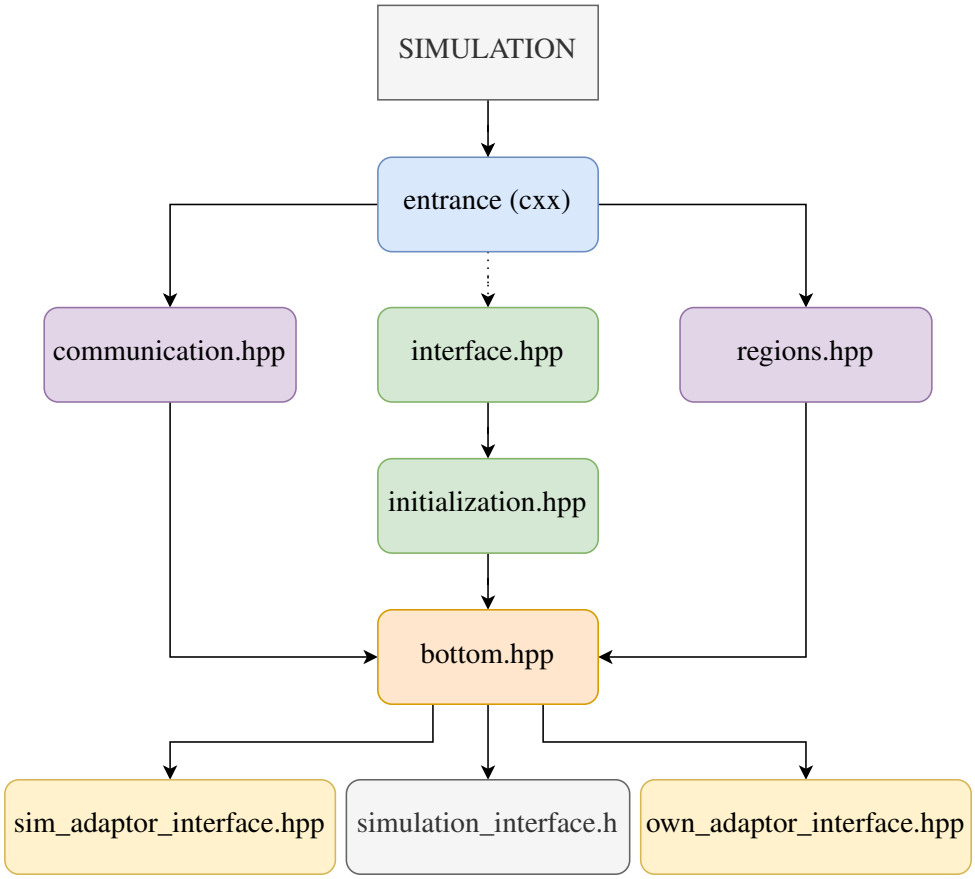


Figure 5.10: Schematic of the plugin's source code.

```
// To aggregate the properties of a code region being measured
class ScoreP_region
{
  public:
  std::string  region_name       ;
  uint32_t     region_id         ;
  uint64_t     timestamp_entrance;
  uint64_t     timestamp_exit    ;
  uint64_t     timestamp_delta   ; // goes indirectly to ParaView
  int          flag_exit         ;
  unsigned int counter           ; // goes to ParaView
};

// Stores the code regions being measured
static std::vector < ScoreP_region * > measuring_regions;
```

For each region the user selects in its simulation code, the plugin will store its name and identifier (an integer assigned to each code region by Score-P at run-time). Every time this region is entered, Score-P collects the respective timestamp information and increases the region's execution counter by

1. When the region is then exited, the exit timestamp is collected, whose difference to the entrance peer corresponds to the time spent inside the region (measured in clock cycles). This measurement will be divided on-the-go by the clock frequency when the simulation's Catalyst adapter calls the plugin, yielding a result in seconds. Finally, `flag_exit` ensures the measurements do not include executions of the same region (in the remaining of the time-step) other than the currently selected ones by the user. Each region selected inside the simulation code will be then one element in the `measuring_regions` vector (as seen above).

Another global variable of unique importance is the definition of the parameters associated with the communication tracking. It bears a similar layout to its measurements counterpart, as seen below:

```
// To aggregate the properties of my communication with other ranks
class ScoreP_communication
{
  public:
  int           id           ; // goes to ParaView
  unsigned int  counter_send ; // goes to ParaView
  unsigned long   bytes_send ; // goes to ParaView
  unsigned int  counter_isend; // goes to ParaView
  unsigned long   bytes_isend; // goes to ParaView
  unsigned int  counter_put  ; // goes to ParaView
  unsigned long   bytes_put  ; // goes to ParaView
  unsigned int  counter_get  ; // goes to ParaView
  unsigned long   bytes_get  ; // goes to ParaView
};

// Stores the messages sent by me to other ranks
static std::vector < ScoreP_communication * > addressees;
```

Each *addressee* (i.e. someone the present rank sends messages to) will become an element on the vector of addressees. `id` is the destination's rank number. For each of the supported MPI calls (MPI_Send, MPI_Isend, MPI_Put and MPI_Get), the communication class stores the amount of times such call is made inside the time-step and the total amount of bytes sent through those calls; hence the total of 8 variables (2 for each type of communication).

Finally, when running the plugin in geometry mode, the Catalyst adapter of the simulation code will need to call the plugin; therefore the latter exposes the necessary functions through a specialized header (`sim_adaptor_interface.hpp`), which essentially contains getters for those variables defined inside the two main plugin classes (explained above). Similarly, when running the plugin in topology mode, the Catalyst adapter of the plugin itself will need to call its main layer (which works now as the simulation); therefore the latter exposes the necessary functions through a specialized header (`own_adaptor_interface.hpp`), which is mostly the same as its simulation's counterpart.

5.2.1.1 The Plugin's input file

The plugin has been written such that as much of its options as possible are chosen at run-time, through its input file, which can be seen below:

```
print_log = 1
clock_resolution = 2400000000
measuring_frequency = 2
measurements = 1
communication = 1
renew_comm = 1
topology_mode = 1
python_script = plugin.py
```

`print_log` accepts either 1 or 0 and is responsible for turning output logs (useful for debugging) on or off; each rank will print to a separate file. `clock_resolution` is the unsigned long precision *clock frequency* (in Hz) to divide the number of clock cycles, when computing the amount of time (in seconds) spent inside the selected code regions; it should be set to the nominal frequency of the machine being used (e.g. 2.4 GHz in the example above). `measuring_frequency` determines how often, in terms of time-steps, the plugin should run; this is useful for heavy (industry-grade) CFD codes, where otherwise (i.e. collecting measurements every time-step) the plugin's overhead would become too big for meaningful analyses.

The following two options accept either 1 or 0 and are responsible for turning on the main features present inside the tool: *measure regions* (which need to be manually selected inside the code) and *track communications*. They are independent from each other: the plugin will warn the user if both of them are turned off, or if no region was selected inside the code (when `measurements` is on), or if none of the four supported MPI calls was detected (when `communication` is on). In all cases, as well as if the input file itself is not found, the simulation will resume as normal (the plugin doing nothing in the background).

If the boundaries between the subdomains (i.e. the neighboring information) of the simulation's mesh do not change with the time-steps, `renew_comm` can be set to 0, reducing therefore the overhead of tracking the communications (the vector of addressees will not be cleaned every time-step, just its variables reset to zero). Finally, `topology_mode` turns on the creation of a symbolic representation of the hardware topology, to be done by the plugin's own Catalyst adapter; which as such requires its python input file (as explained in section 5.1.2 above), whose name is provided at `python_script`. It comes with the plugin's source code.

5.2.2 Understanding the Plugin's Catalyst adapter

Figure 5.11 is a scheme of the plugin's Catalyst adapter's source code, comprised again of only one translation unit: the main cxx file, which recursively includes all its headers. This main file, called *entrance* in the diagram, is the first point of contact with the upper layer: the plugin's main one, which

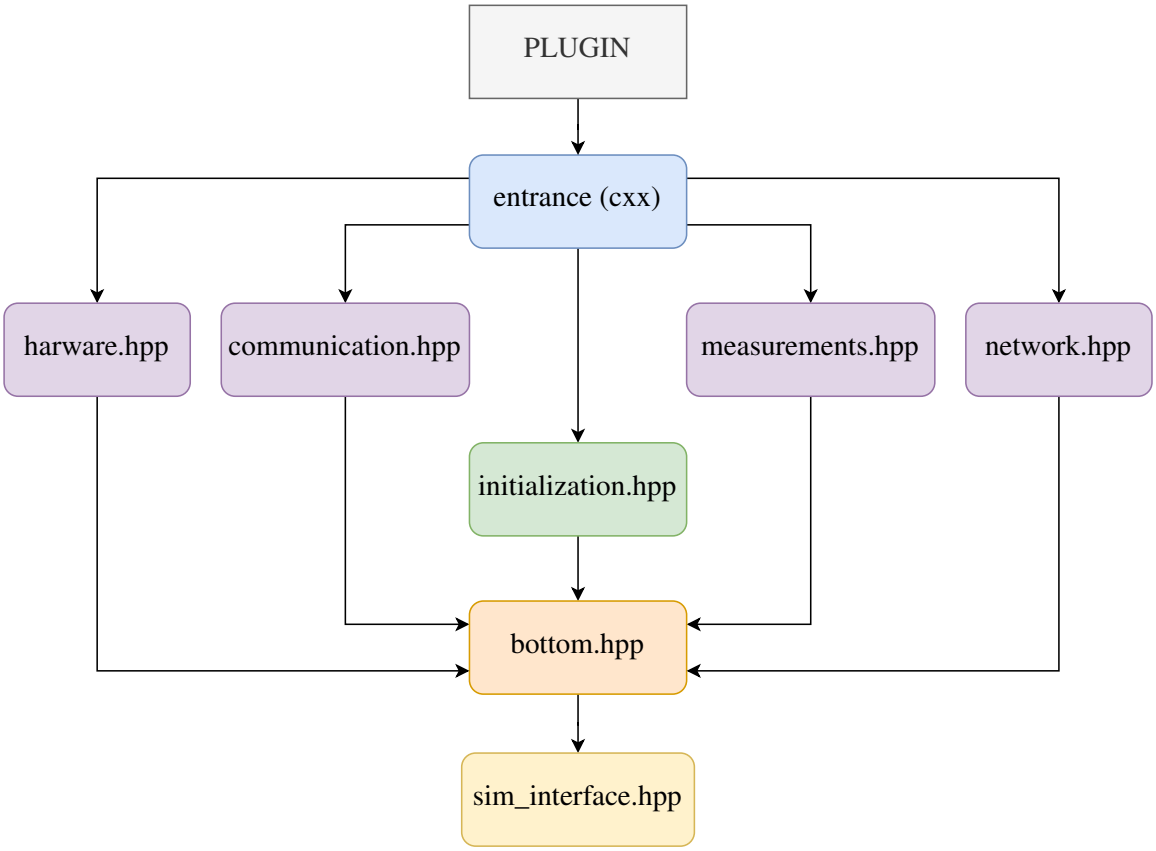


Figure 5.11: Schematic of the plugin adapter's source code.

here works as the "simulation" (being instrumented with Catalyst). It contains the definitions of those three basic calls: initialize (to be executed only once, at the beginning of the simulation), run (to be executed at the end of each time-step) and finalize (to be executed only once, at the end of the simulation).

`hardware.hpp` is the heart of topology mode: it contains the function that will create at run-time a symbolic representation of the computing architecture's topology, what is done by means of the *Hardware Locality* library (hwloc)[4] [21]: an open-source "software package [that] provides a portable abstraction (across OS, versions, architectures, ...) of the hierarchical topology of modern architectures, including NUMA memory nodes, sockets, shared caches, cores and simultaneous multithreading" [tool's website]. Such representation – which will become clearer later, when the outputs of the tool are presented – is the only feature activated by default when the user turns on topology mode in the plugin's input file (as shown in the previous topic). All other features are optional and independent from each other.

`measurements.hpp` is responsible for building a space for the manually selected code regions (if any) and updating it every time-step. `network.hpp` will add to the visualization *network topology* information. Then comes the `communication.hpp`, which will symbolically draw the messages sent between the ranks and update their information (amount of invocations and amount of bytes sent, per communication type), every time-step. The entrance (cxx) file needs to read the plugin adapter's input file (to be presented in the next topic); the related functions are defined in `initialization.hpp` (just

[4]This library is installed by default in Linux distributions.

like in the main plugin's code). Both entrance file and all its headers will then meet at `bottom.hpp`, which is again comprised only of global variables and the headers needed by them. Finally, since the three basic calls (initialize, run and finalize) will need to be called by an external translation unit (in this case, the plugin's main one), they are also declared in the interface header `sim_interface.hpp`.

5.2.2.1 The Plugin adapter's input file

Like the plugin itself, its adapter has been written such that as much of its options as possible are chosen at run-time, through its input file, which can be seen below:

```
print_log = 1
inter_node_offset = 5
network_topology = 1
topology_file = topology.conf
measurements = 1
communication = 1
comm_offset = 5
renew_comm = 1
periodic_BC = 0
```

Similarly to the main plugin's code, `print_log` turns this translation unit's output logs on or off; they are printed in separate files as those of the plugin's main layer. `inter_node_offset` controls the standard distance to be applied between compute nodes in ParaView; it will become clearer when the outputs of the tool are shown, later in this thesis.

`network_topology` turns the drawing of the network switches on or off. This feature requires a configuration input file, which contains the information about the cluster's node connectivity. `topology_file` provides the name of such file, which must be located at the working directory at run-time. How to generate such file will be explained in Sec. 6.1 below.

`measurements` turns on the display of the data about the selected code regions in topology mode. `communication`, on its turn, activates the drawing of the messages exchanged between ranks. Here it is also possible to adjust one of such drawing's parameters (through the variable `comm_offset`), which will become clearer when the results of the tool are shown. `renew_comm` has here the same meaning as in the main plugin's input file; actually if any of the two options is on, the vector of addressees will be cleaned every time-step. Finally, `periodic_BC` can be activated when the borders of the simulation domain contain periodic boundary conditions: the final drawing in ParaView will be clearer. More about that, and how it manifests itself in the resulting visualizations, in Chapter 7 below.

5.2.3 A Word About Installation

Before going on to the next chapter of this thesis, how to install the tool will be briefly discussed. When intended to be used solely on geometry mode, its installation is straightforward and can be accomplished by a couple of commands. Assuming Score-P is installed under `/home/user/scorep/install`:

```
export PATH="$PATH:/home/user/scorep/install/bin"

g++ -c -fPIC entrance.cxx  \
    -o core.o `scorep-config --cppflags`
g++ -shared -Wl,-soname,libscorep_substrate_Catalyst.so  \
    -o libscorep_substrate_Catalyst.so core.o
```

In order, however, to use the plugin to its full potential, it is necessary to build its Catalyst adapter. The installation then requires using *CMake* [13], "an open-source, cross-platform family of tools designed to build, test and package software" [tool's website]. It is developed by Kitware, the same company behind Catalyst itself. Assuming ParaView is installed under `/home/user/paraview/install` and the plugin files have been downloaded under `/home/user/plugin`:

```
export PATH="$PATH:/home/user/scorep/install/bin"
SCOREP_PATH="`scorep-config --cppflags`"
PATH_ARRAY=($SCOREP_PATH)

CMAKE_FLAGS=""
CMAKE_FLAGS="${CMAKE_FLAGS} -D Dir_0=${PATH_ARRAY[0]}"
CMAKE_FLAGS="${CMAKE_FLAGS} -D Dir_1=${PATH_ARRAY[1]}"
CMAKE_FLAGS="${CMAKE_FLAGS} -D ParaView_DIR=/home/user/paraview/install/lib64/cmake/paraview-5.7"
CMAKE_FLAGS="${CMAKE_FLAGS} -D CMAKE_C_COMPILER=gcc"
CMAKE_FLAGS="${CMAKE_FLAGS} -D CMAKE_CXX_COMPILER=g++"
CMAKE_FLAGS="${CMAKE_FLAGS} /home/user/plugin/src"

cmake ${CMAKE_FLAGS}

make
```

The plugin's Catalyst adapter is built as a static library, which is then incorporated into the plugin itself, a shared library – ready to be linked with the simulation's executable. It is important to note, however, that editions of the open-source compiler toolchain from 2020a onwards will load a version of OpenMPI which does not work with Score-P (version 6.0) yet. Thus, if edition 2019b or lower is not available in the system, it is recommended to use the Intel® compilers. In this case, it will be necessary to provide to CMake the location of the hwloc library (either version 1.0 or 2.0) installation.

6 Evaluation

This chapter describes the computational resources and test cases selected to use with the plugin.

6.1 HPC infrastructure

Most simulations were done in *Haswell* nodes of TU Dresden's HPC cluster, *Taurus*.[1] The Haswell CPU family [33] is an Intel® processor microarchitecture originally anounced in 2013 and comprised of a 22 nm process technology. The topology of the variation used in the cluster's nodes is shown on Figure 2.8 in Section 2.4 above: a total of 64 GB of RAM memory is distributed across two sockets (Intel® Xeon® CPU E5-2680 v3 @ 2.5 GHz), each with 30 MB L3 cache shared amongst 12 CPU cores, each in turn provided with 256 kB L2 cache and 32 kB L1 cache memories. There is a total of 1328 nodes of that type in the cluster, interconnected through *InfiniBand* [53]: a Mellanox® computer networking communications standard commonly used in supercomputers. It provides RDMA capabilities and uses a switched fabric topology: the nodes are organized into network islands (83 in total), each comprised of 16 nodes, all under the same leaf network switch. Such leaf switches are in turn interconnected through the master switch.

Taurus uses as job scheduler *Slurm* [72]: "an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters" [tool's website]; which in turn comes with a set of tools[2] which will go through the cluster network and generate automatically its connectivity information, saving it onto a file. This file has been used as the network topology configuration file (described in Section 5.2.2.1 above).

6.2 Test-cases

Four test-cases will be used to demonstrate the functionalities of the plugin: two well-known benchmarks and two industry-grade CFD codes. Both benchmarks and the Rolls-Royce's CFD code were built with release 2018a of Intel® compilers, in association with version 2.0 of the hwloc library; CODA, on its turn, was built with release 2019a of the GNU open-source compiler toolchain (which loads version 1.0 of hwloc).

6.2.1 Benchmarks

The *NAS Parallel Benchmarks* (NPB) [27] "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" [tool's website]. Here one

[1] For details about the cluster, see `https://doc.zih.tu-dresden.de/jobs_and_resources/hardware_overview/`.

[2] Notably the `slurmibtopology.sh` script.

of each was used: the *Multi-Grid* (MG) and the *Block Tri-diagonal* (BT) respectively (version 3.4). Both were run in a modified Class *D* layout in four nodes, each with 16 ranks (i.e. pure MPI, no OpenMP), one per core and with exclusive node utilization. Their grids are simple and consist of a parallelepiped with the same number of points in each cartesian direction (1024 for MG, 408 of BT), hence here only topology mode shall be tested. 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,[3] MG was run for 3000 iterations (each comprised of 9 multigrid levels), whereas BT for 1000. The plugin would generate 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. Finally, both benchmarks were tested with versions 6.0 of Score-P and 5.7.0 of ParaView.

6.2.2 Industrial CFD codes

6.2.2.1 Rolls-Royce's Hydra

Hydra is Rolls-Royce's in-house CFD code [46], based on a preconditioned time marching of the Reynolds-Averaged Navier-Stokes (RANS) equations. They are discretized in space using a second-order, edge-based finite volume scheme with a multistage, explicit Runge-Kutta scheme as a steady time marching method. Multigrid (like the MG benchmark) and local time-stepping acceleration techniques are used to improve steady-state convergence [42].

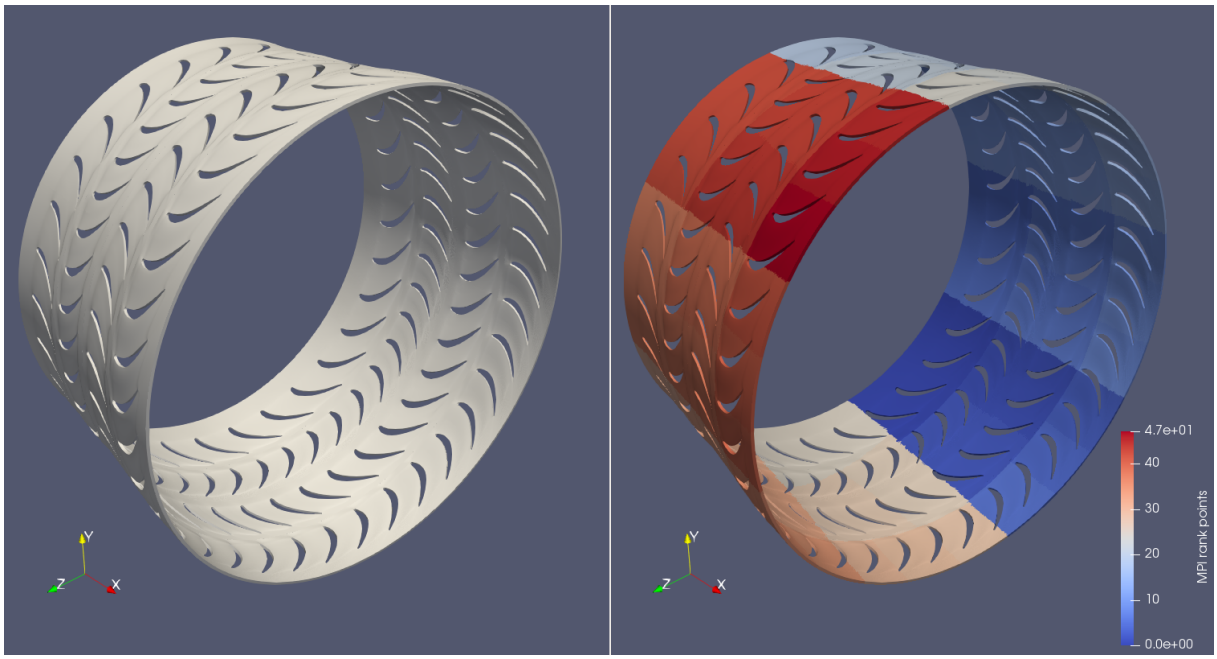


Figure 6.1: Geometry used in the Rolls-Royce's CFD code simulations (left) and its partitioning among ranks (right). The scale in the lower-right corner reveals the number of MPI ranks used in the simulation: 48 (*0* counts as *1*).

[3] And therefore the statistical oscillation of their run time not to invalidate the analyses.

Figure 6.1 shows the test case selected for this thesis: 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.

Here the simulations were done using two entire nodes in Taurus, each with 24 ranks (again pure MPI), again one per core and with exclusive node utilization. Figure 6.1 shows the domain's partitioning among the processes, done in a geometric fashion – which ensures a similar number of grid points between each sub-domain[4] – and not subject to any stochastic variance. 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. Catalyst was generating post-mortem files every 20th time-step (i.e. every 36°), what led to 10 stage pictures by the end of the simulation. Finally, versions 4.0 of Score-P and 5.5.2 of ParaView were used.

6.2.2.2 CFD for Onera, DLR and Airbus (CODA)

CODA[5] is a new CFD solver [36], made from scratch with HPC in mind and used to study the external aerodynamics around both aircraft and spacecraft. Figure 6.2 gives an overview about all the software tools which were taken into account when designing the new solver. They range from mesh manipulation, discretization parameters, flow modelling etc. up to HPC communication strategies [68].

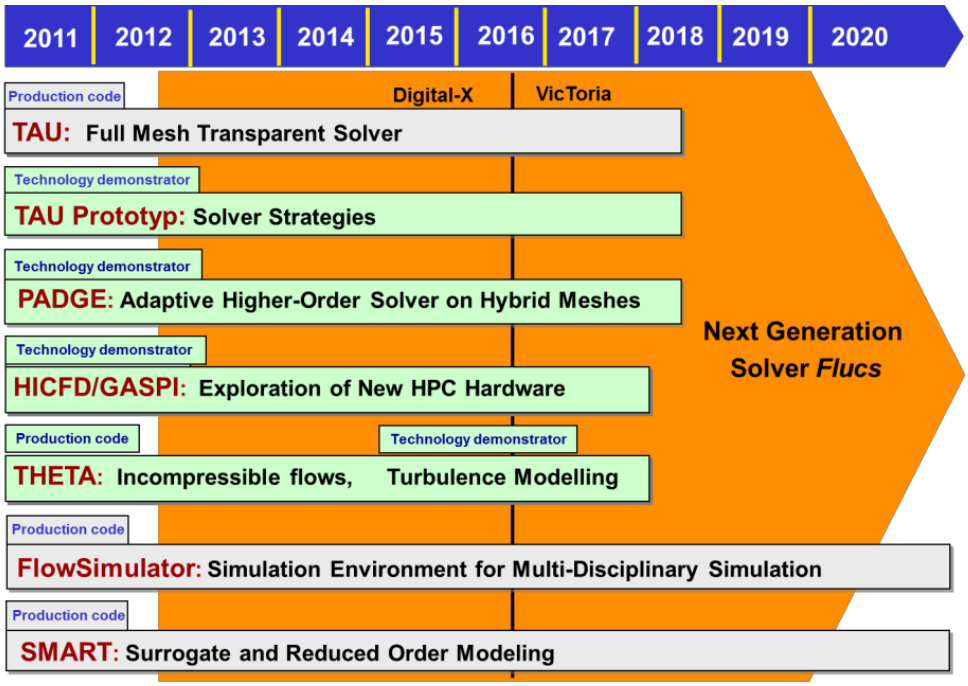


Figure 6.2: Overview about all the software tools which were taken into account when designing the new CFD solver, CODA (at the time of this diagram, still called *Flucs*; retrieved from [68]). Such tools range from mesh manipulation, discretization parameters, flow modelling etc. up to HPC communication strategies.

[4] It does not look so in the picture because the grid gets finer in the x direction.
[5] Stands for 'CFD for Onera, DLR and Airbus'.

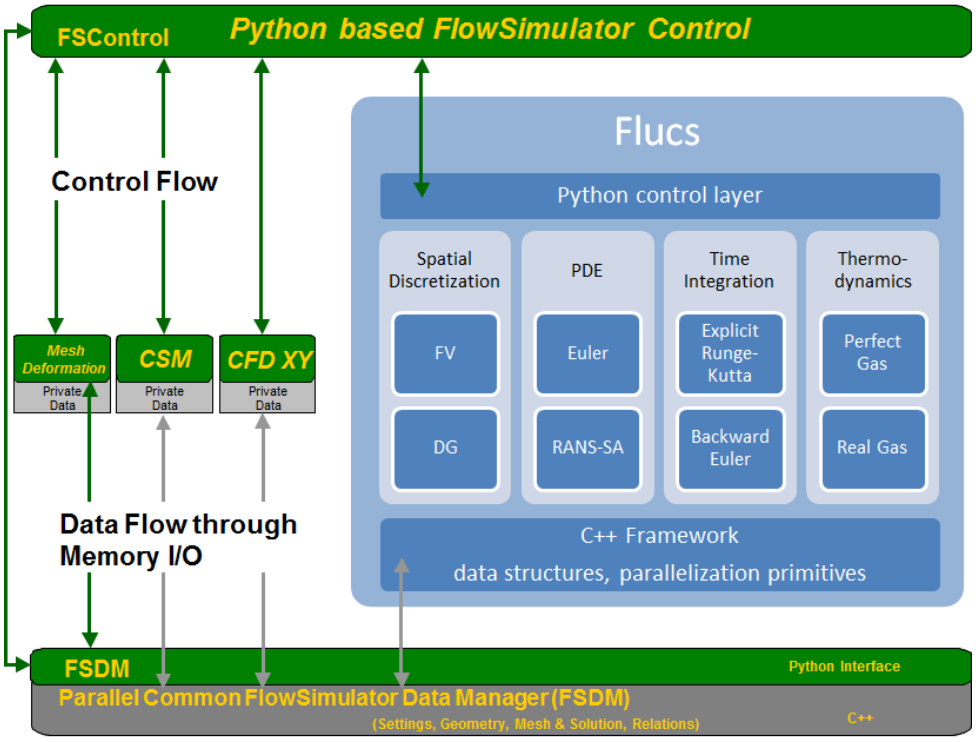


Figure 6.3: Chart of the components of DLR's multipurpose, interdisciplinary simulation workflow, Flow Simulator (retrieved from [48]). CODA (called Flucs at the time) appears as one of the components, namely the one responsible for the flow solution; others include CSM, mesh generation and manipulation, I/O filters etc.

The new CFD solver was made to operate within a preexisting simulation workflow, called *Flow Simulator*, which includes also modules for CSM (Computational Solid Mechanics),[6] mesh generation and manipulation, I/O filters to different formats etc. [51]. Such workflow is illustrated in Figure 6.3, which also shows the options available for each of the components of the CFD simulation; in our case: finite volume for spatial discretization, RANS equations (including one additional equation for turbulence) for the flow description, explicit Runge-Kutta for time discretization and Perfect Gas for the thermodynamics modelling (equation of state as system closure).

The test case selected for this code comes from the *5th AIAA CFD Drag Prediction Workshop* (DPW-V) [65], which uses an aircraft model (shown on Figure 6.4) for which experimental tests' data is available (important for validation purposes); it is called 'NASA Common Research Model' (CRM). Its correspondent computational model is provided in different versions, depending upon how many geometric features of the airplane are to be simulated [66]; here the Wing-Body configuration shown on Figure 6.5 will be used, which was also the one used at the Workshop for the CFD analyses.

The simulation domain is then shown on the left-hand side of Figure 6.6 (view from far away from the airplane's wings) and Figure 6.7 (view from the mid plane crossing the airplane), as well as the top part of Figure 6.8 (same view, but from close). The correspondent CFD mesh is made available in four different sizes, with varied resolution; here the finest grid shall be used, comprised of 17.441.905 points (34.504.704 prisms). A scalability analysis was conducted on such grid – as per a sample setup of 10

[6]For example, to account for stress analysis at the airplane's structure during operation.

Figure 6.4: Experimental setup of the aircraft model used with CODA (retrieved from [65]). Called NASA Common Research Model (CRM), it represents a common civil airplane under standard flight conditions.

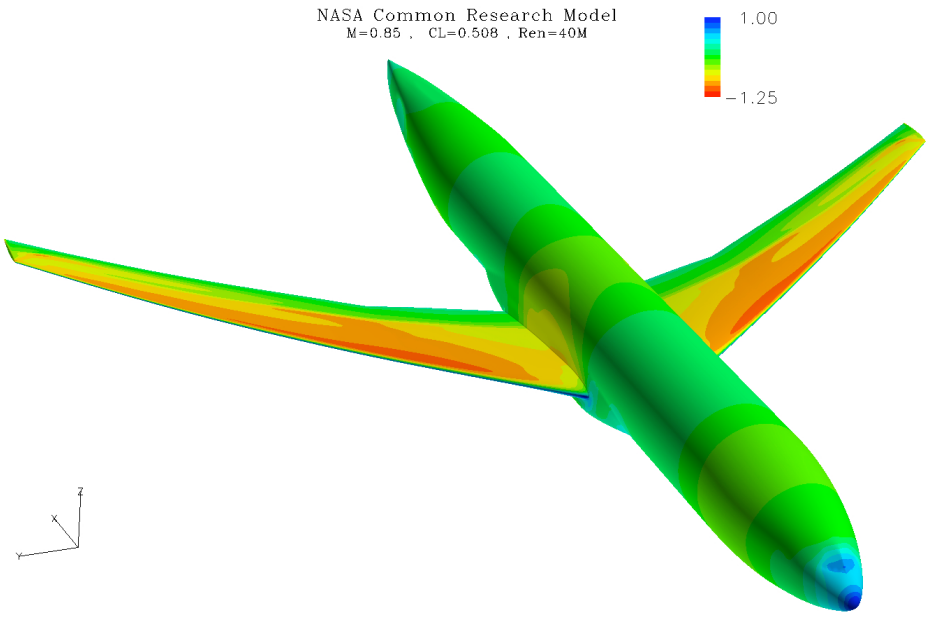


Figure 6.5: Wing-Body version of the computational model of NASA's CRM (retrieved from [66]).

time-steps, each comprised of 10 iteration steps – within partition Haswell in TUD's cluster (where each compute node is comprised of 24 CPU cores), in order to figure out what is the ideal amount of MPI ranks which should be used in the simulations. The results of such analysis is shown on Figure 6.9 and Figure 6.10 below and reveal that it is not worthy going beyond 32 nodes (768 cores), as there was no perceivable benefit on the overall simulation's run time, i.e. considering initialization, mesh reading, partitioning etc.; whilst with increased peak memory consumption per core, reached somewhen

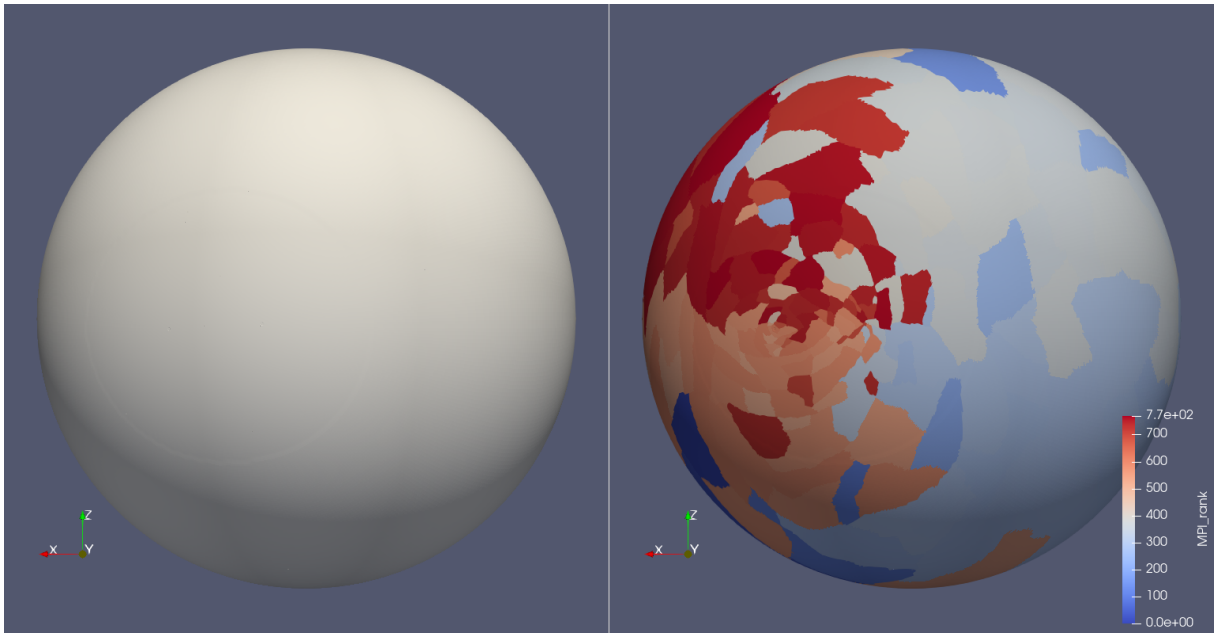


Figure 6.6: Computational domain for the CFD simulations in NASA's CRM (as seen from far away from the airplane's wings), shown on the left; and its distribution across MPI ranks (768 in total) for parallel execution, on the right-hand side.

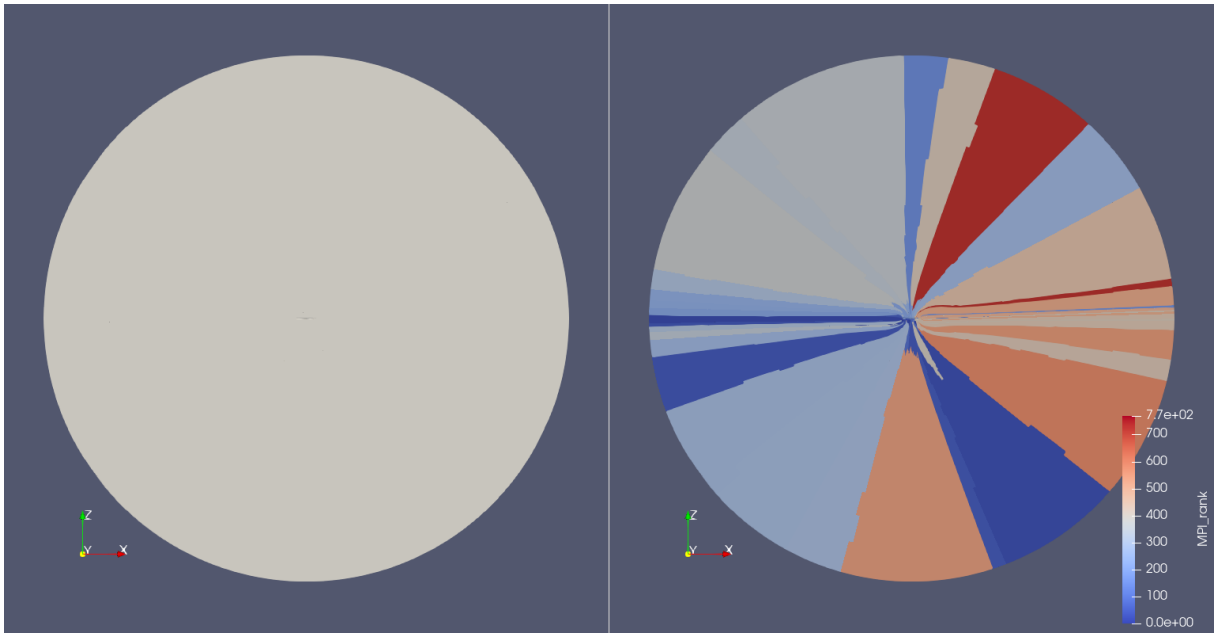


Figure 6.7: Computational domain for the CFD simulations in NASA's CRM (as seen from the airplane's sagittal plane), shown on the left; and its distribution across MPI ranks (768 in total) for parallel execution, on the right-hand side.

during the simulation.[7] The results in Figure 6.10 were computed from the iterations time only, the parallel efficiency at 32 nodes (768 cores) is then 88%; whereas those of Figure 6.9 refer to the overall simulation.

[7]Remember, however, that this conclusion applies to when running only a total of 100 iterations on the grid. The more iterations one runs, the lower the effect of initialization, mesh reading, mesh partitioning etc. becomes on the overall simulation time.

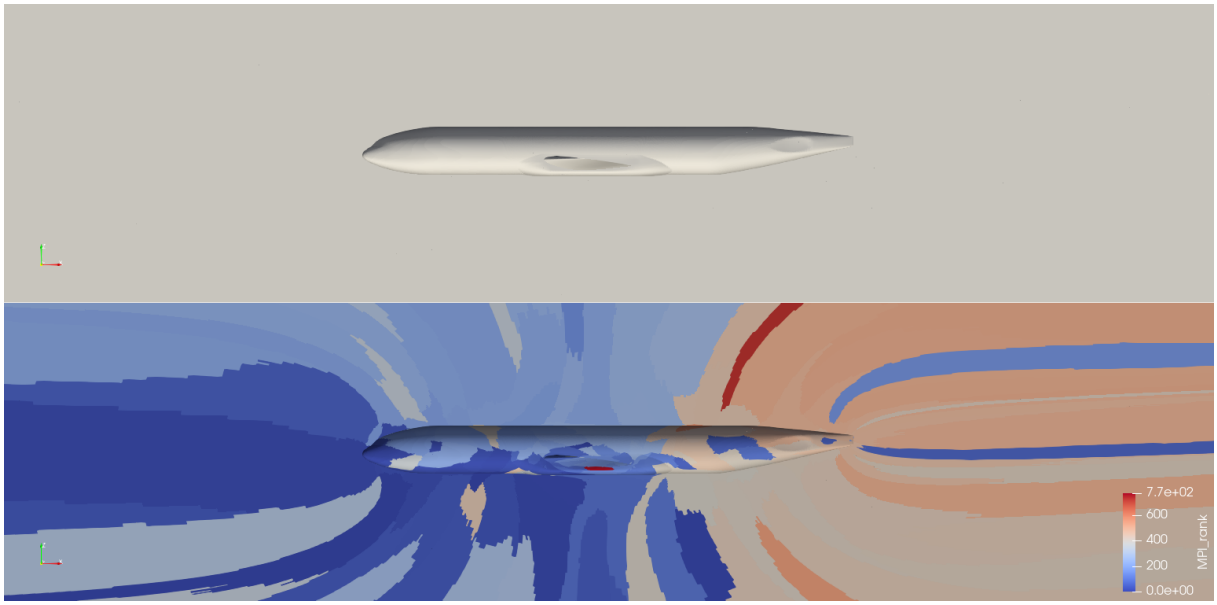


Figure 6.8: Computational domain for the CFD simulations in NASA's CRM (as seen close from the airplane's sagittal plane), shown on the top; and its distribution across MPI ranks (768 in total) for parallel execution, on the bottom.

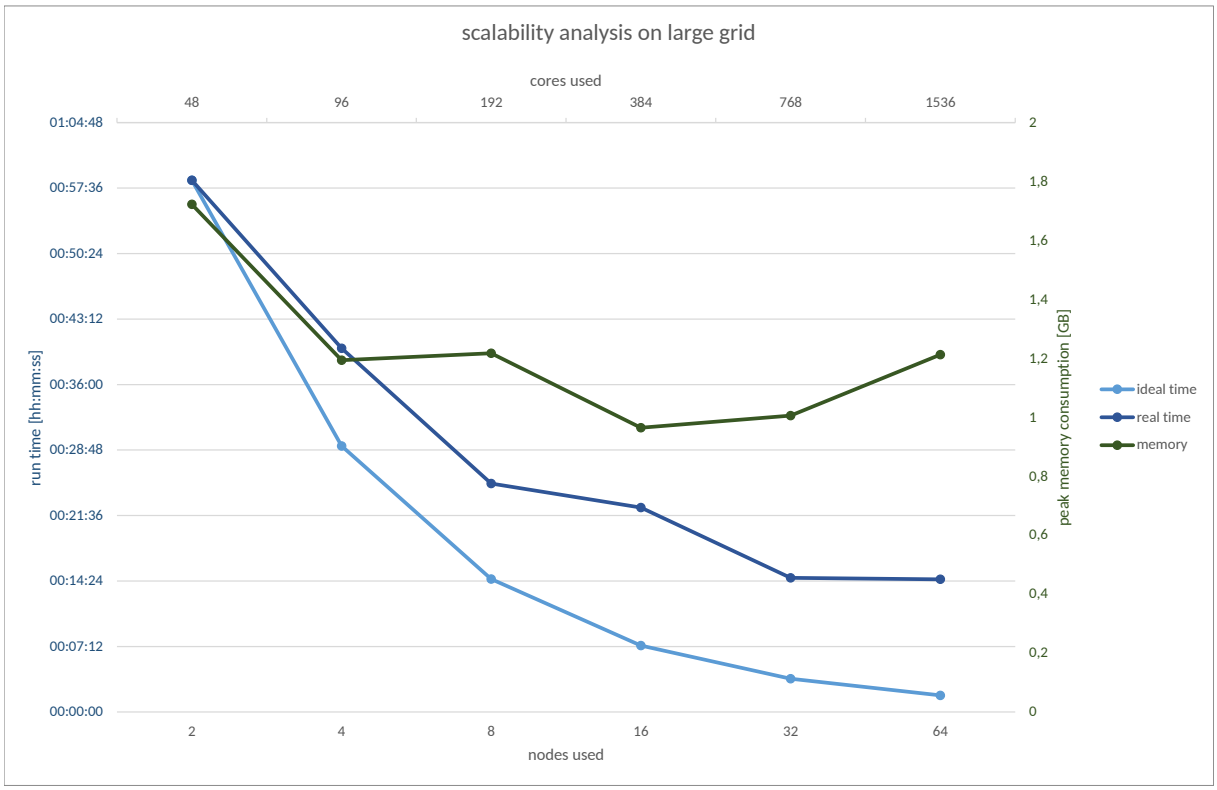
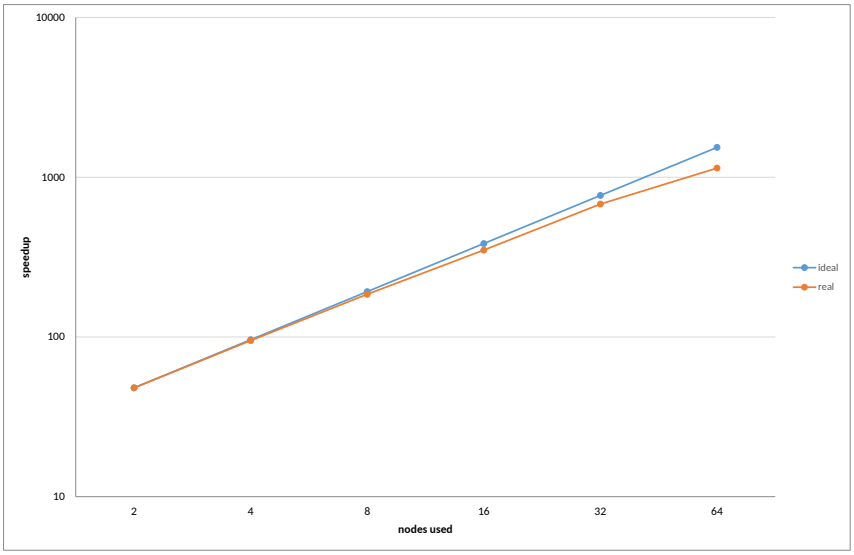
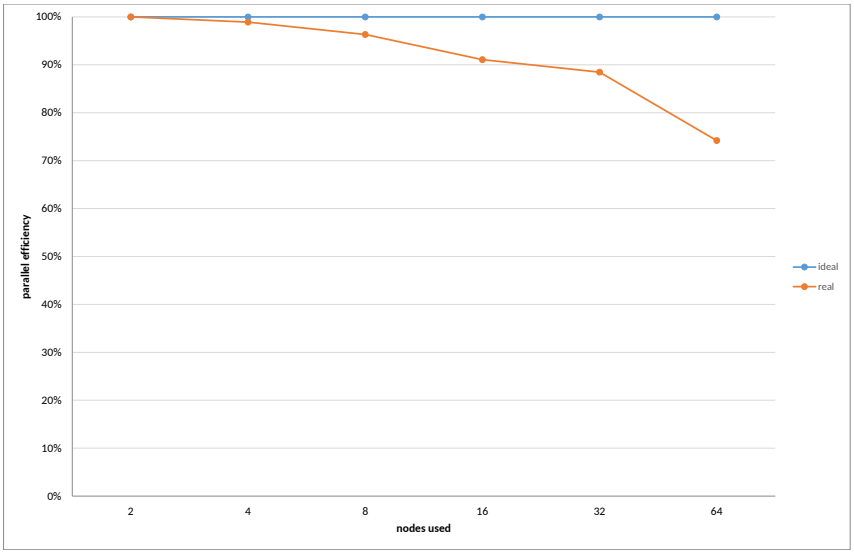


Figure 6.9: Scalability analysis conducted on the finest grid of NASA's CRM within partition Haswell in TUD's cluster. It revealed that it is not worthy going beyond 32 nodes (768 cores), as there was no perceivable benefit on the overall simulation's run time, whilst with increased peak memory consumption per core.

Once the number of parallel processes to be used is chosen, their distribution across the simulation domain is then shown on the right-hand side of Figures 6.6 and 6.7, as well as the bottom of Figure 6.8;



(a) Speedup results on CODA's test case



(b) Parallel efficiency results on CODA's test case

Figure 6.10: Comparative displays of the speedup and parallel efficiency on CODA's test case, computed from the iterations time only of the simulations.

here again only MPI (no OpenMP) was used. Preliminary analyses on CODA revealed the function that computes fluxes across faces to be the heaviest one, in terms of computations, without communication and related to the simulation geometry; it was selected for pipelining. No other code region, exempt from communications, that had a significative impact on the code's performance could be found. Ten time-steps were simulated, each internally converged through 100 iterations.[8] Catalyst was generating post-mortem files every 2nd time-step, which led to 5 stage pictures by the end of the simulation.[9] Finally, versions 6.0 of Score-P and 5.7.0 of ParaView were used.

[8] I.e. more iterations steps than before, when the scalability analysis was conducted, as now the task is the detailed investigation of the code with the plugin.

[9] Remember that the grid here is approximately 17.5 times bigger than in Rolls-Royce's test-case.

6.3 Overhead

In this Section, the overhead associated with using the plugin shall be analysed. Two main parameters will be measured: the *time* taken to run the simulation and the *memory* consumed by it.

6.3.1 Settings

In the following tables, the *baseline* results refer to the pure simulation code, running as per the settings presented above; 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, running with both profiling and tracing modes deactivated, as neither of them is needed for the plugin to work.[10] Finally, ++ *plugin* refers to when the plugin is also used: running only one *feature* (measurements or communication) and *mode* (geometry or topology) at a time (hence the total of four tables)[11] and on the iterations when there would be generation of output files[12]. The percentages shown in these last 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` option. When the plugin is used to show communication between ranks, this option would be enough, as no instrumentation (manual or automatic) is needed when only MPI calls are being tracked[13]. 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 analysis (as opposed to when tracking communications, when only MPI-related calls are intercepted). In this case, it was then necessary to add the Score-P compile flag `-user` 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 Figure B.1: `if MODULO...` additionally ensures measurements are collected solely 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 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)[14]. Furthermore, when measuring code functions, interception of MPI-related calls was turned off at run-time[15].

In the overhead tests for topology mode, the industrial CFD codes were used without their Catalyst adapters. The plugin can perfectly run in this mode when the simulation has the in situ adapter; but in order to isolate its overhead, it will be measured in codes without Catalyst.

Finally, the *memory* results shown refer to the *peak instantaneous* memory consumption per rank, reached *somewhen* during the simulation. It neither means that *all* ranks needed that much memory (simultaneously or not), nor that the memory consumption was like that during the *entire* simulation.

[10] If present, there would be at the end of the simulation, apart from the plugin's output files, those 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 of plugin, profiling and/or tracing sum up.

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

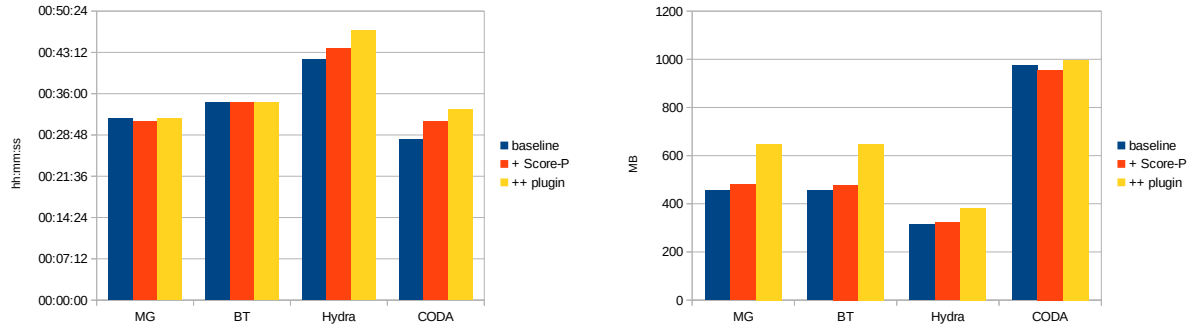
[12] Given the simulation was not being visualized live in ParaView, there was no need to let the plugin work in time-steps when no data would be saved to disk.

[13] Remember that detection of MPI-related functions is done independently at run-time (see Sec. 5.1.1 above).

[14] 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.

[15] By means of the `SCOREP_MPI_ENABLE_GROUPS` environment variable (see Sec. 5.1.1 above).

6.3.2 Results

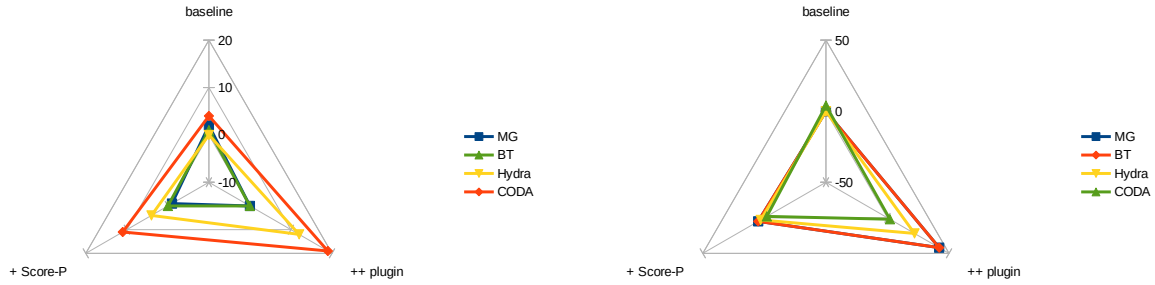


(a) Time (b) Peak memory consumption

Figure 6.11: Comparative displays of the simulation time and peak memory consumption when measuring code functions in topology mode.

Table 6.1: Plugin's overhead when measuring code functions in topology mode.

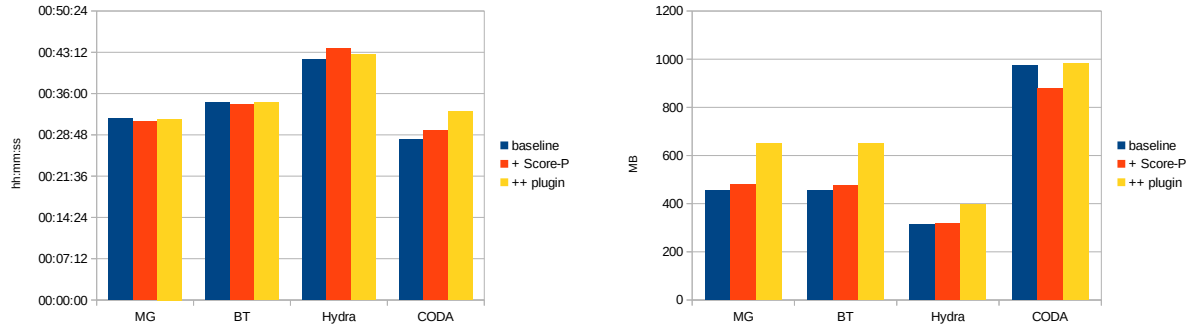
	running time			**memory (MB)**		
	baseline	**+ Score-P**	**++ plugin**	**baseline**	**+ Score-P**	**++ plugin**
MG	31m37s ± 2%	31m09s (-1%)	31m42s (0%)	455 ± 0%	479 (5%)	648 (42%)
BT	34m28s ± 1%	34m26s (0%)	34m28s (0%)	455 ± 0%	478 (5%)	648 (42%)
Hydra	42m00s ± 0%	43m52s (4%)	47m04s (12%)	314 ± 0%	323 (3%)	382 (22%)
CODA	27m59s ± 4%	31m09s (11%)	33m14s (19%)	974 ± 4%	956 (-2%)	997 (2%)



(a) Time (b) Peak memory consumption

Figure 6.12: Comparative displays of the simulation time and peak memory consumption's percentage overhead when measuring code functions in topology mode.

Table 6.1 and Figures 6.11 and 6.12 show the results when measuring the selected code regions in topology mode. Table 6.2 and Figures 6.13 and 6.14 refer to when tracking communications in topology mode. Table 6.3 and Figures 6.15 and 6.16 are then for measuring the selected code regions in geometry mode. Finally, Table 6.4 and Figures 6.17 and 6.18 refer to when tracking communications in geometry mode. The lines in the tables are ordered approximately by the size of the respective test-case. For CODA (the biggest one), the Catalyst adapter replaces completely the simulation's native outputs (i.e. those that would come from the simulation itself, outside the in situ channel); this means that, for this test-case, almost all I/O associated with running the code will come from Catalyst (if present).

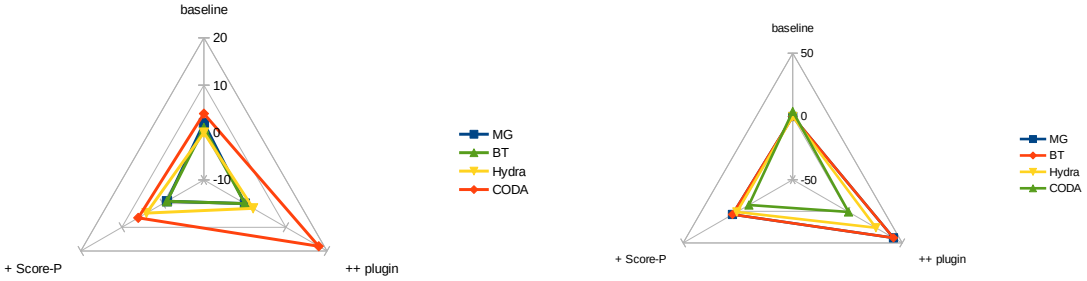


(a) Time (b) Peak memory consumption

Figure 6.13: Comparative displays of the simulation time and peak memory consumption when showing communication in topology mode.

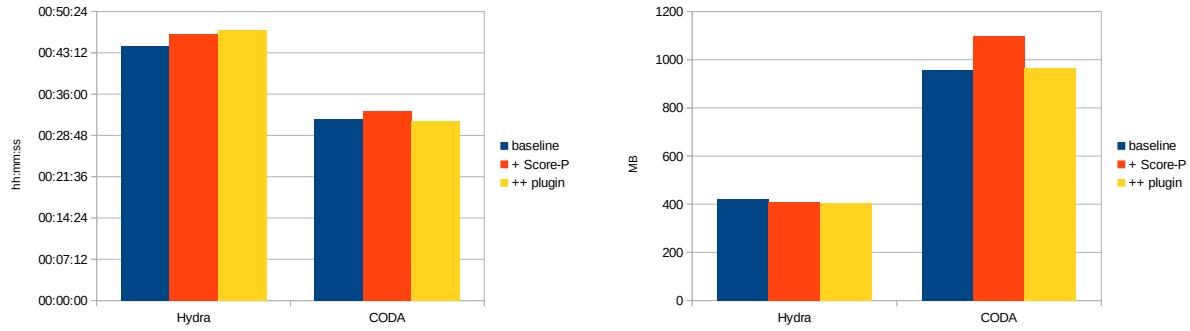
Table 6.2: Plugin's overhead when showing communication in topology mode.

	running time			**memory (MB)**		
	baseline	**+ Score-P**	**++ plugin**	**baseline**	**+ Score-P**	**++ plugin**
MG	31m37s ± 2%	31m09s (-1%)	31m34s (0%)	455 ± 0%	479 (5%)	648 (42%)
BT	34m28s ± 1%	34m08s (-1%)	34m24s (0%)	455 ± 0%	477 (5%)	648 (42%)
Hydra	42m00s ± 0%	43m50s (4%)	42m53s (2%)	314 ± 0%	316 (1%)	397 (26%)
CODA	27m59s ± 4%	29m33s (6%)	32m56s (18%)	974 ± 4%	880 (-10%)	983 (1%)



(a) Time (b) Peak memory consumption

Figure 6.14: Comparative displays of the simulation time and peak memory consumption's percentage overhead when showing communication in topology mode.

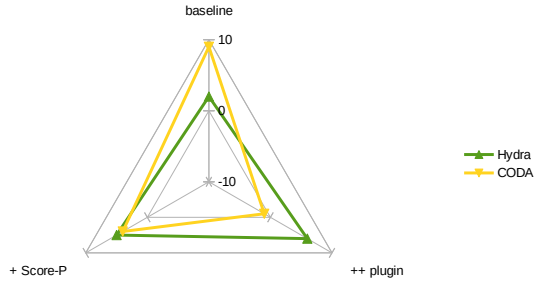


(a) Time (b) Peak memory consumption

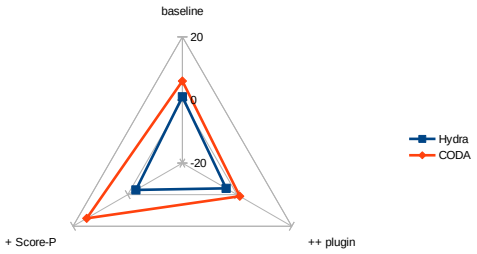
Figure 6.15: Comparative displays of the simulation time and peak memory consumption when measuring code functions in geometry mode.

Table 6.3: Plugin's overhead when measuring code functions in geometry mode.

	running time			memory (MB)		
	baseline	**+ Score-P**	**++ plugin**	**baseline**	**+ Score-P**	**++ plugin**
Hydra	44m20s ± 2%	46m30s (5%)	47m12s (6%)	423 ± 1%	410 (-3%)	405 (-4%)
CODA	31m43s ± 9%	33m01s (4%)	31m20s (-1%)	958 ± 6%	1100 (15%)	966 (1%)

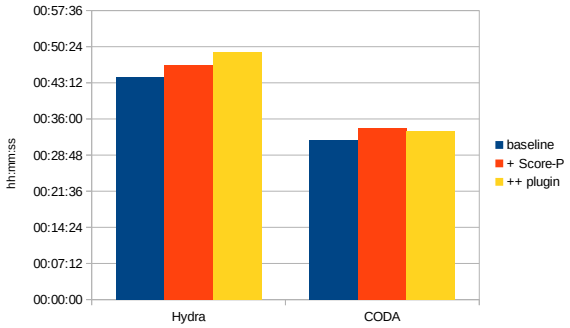


(a) Time

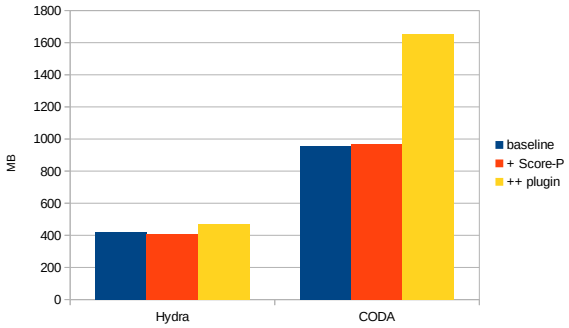


(b) Peak memory consumption

Figure 6.16: Comparative displays of the simulation time and peak memory consumption's percentage overhead when measuring code functions in geometry mode.



(a) Time

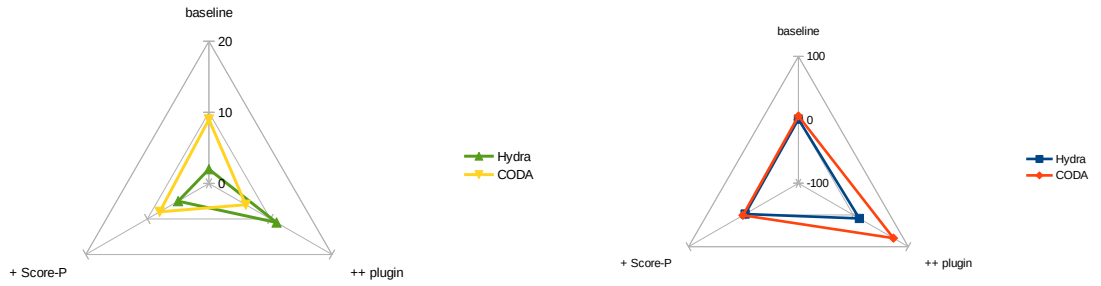


(b) Peak memory consumption

Figure 6.17: Comparative displays of the simulation time and peak memory consumption when showing communication in geometry mode.

Table 6.4: Plugin's overhead when showing communication in geometry mode.

	running time			memory (MB)		
	baseline	**+ Score-P**	**++ plugin**	**baseline**	**+ Score-P**	**++ plugin**
Hydra	44m20s ± 2%	46m38s (5%)	49m21s (11%)	423 ± 1%	410 (-3%)	468 (11%)
CODA	31m43s ± 9%	34m14s (8%)	33m37s (6%)	958 ± 6%	966 (1%)	1655 (73%)



(a) Time

(b) Peak memory consumption

Figure 6.18: Comparative displays of the simulation time and peak memory consumption's percentage overhead when showing communication in geometry mode.

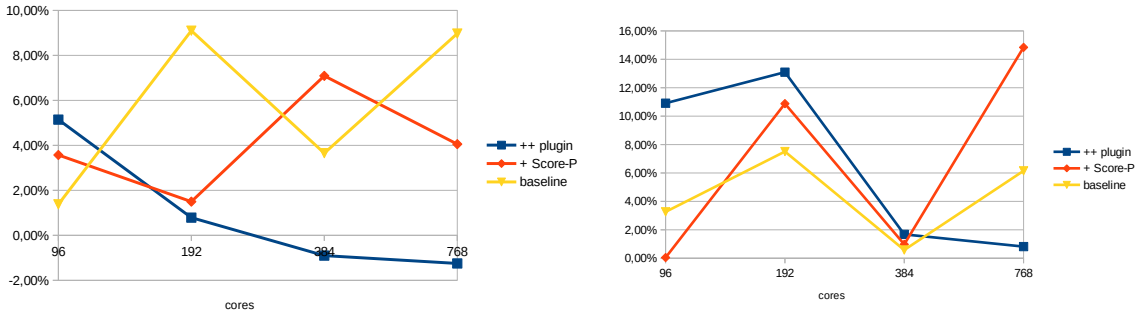
From the tables it is initially noticeable that the memory measurements for CODA in Table 6.2 and Table 6.3 seem to be outliers, as it does not make sense to have a smaller overhead when adding more computations and memory usage onto the simulation. Topology mode was expected to scale worse as geometry mode when it comes to run time, as the latter just adds a couple of extra data arrays into a Catalyst adapter which was already present in the code, whereas the former needs to create a brand new in situ adapter from scratch. However, the overhead is not in the instrumentation side itself (as seen from the Score-P time results), but rather on the I/O associated with Catalyst saving all the results to disk. And such I/O brings instability to the measurements: compare the oscillation of the baseline time results for CODA when naturally with Catalyst (geometry mode) and not (topology mode) – the fluctuation of the results is higher in geometry mode, as data values for millions of mesh points are being saved to disk.

The relative increase in memory consumption in topology mode tends to diminish as you increase the size of the test-case, which is desired from the point of view of scalability. On the other hand, the high result for the plugin when showing communication in geometry mode in CODA is due to the fact that this test-case has been run with the highest amount of ranks (768) and each of them introduces 3 new data arrays (location within the grid, amount of messages received, amount of bytes received; $768 * 3 = 2304$ extra arrays in total) into the simulation's mesh (17.5 million values for each array) in Catalyst, which increases the memory needed by it. Even though the values are the same for each point located within one specific subdomain, they are already passed to Catalyst through an optimized, loop-free way (i.e. once for all points within my rank's specific grid). On the other hand, having two grids in Catalyst (one 'real', for the flow variables, and a coarser one, for the performance data) would break the correlation with the simulation's running properties and might even have detrimental effects on its performance, as the second mesh would require its own memory, e.g. for the definition of the coordinates of each of its points.

Finally, it is possible to improve the tool's performance even more, by means of a simple intervention in its source code. When measuring regions, the communication-related callbacks in the Score-P's plugin API may be commented out; similarly, when tracking communications, the regions-related callbacks may be commented. This eliminates the overhead associated with one if statement (evaluated to false) in the plugin side whenever the correspondent trigger is activated, e.g. when the simulation code enters/exits a MPI call while tracking communications. Figure 5.6 contains the code where this can be done.

6.3.3 Scalability Analysis

In order to further investigate the overhead brought by the plugin, a comparative study has been conducted on the largest test-case (CODA), with different amount of resources: 4, 8, 16 and 32 nodes (96, 192, 384 and 768 cores respectively). The baseline results of the stages outside the operating point are the average and standard deviation of three measurements. In the following graphs, the numbers shown are the relative (percentage) overhead of Score-P (+ plugin) or the relative standard deviation of the measurements (for the baseline values).



(a) Time

(b) Peak memory consumption

Figure 6.19: Comparative displays of the simulation time and peak memory consumption's percentage overhead when measuring code functions in geometry mode in CODA test-case.

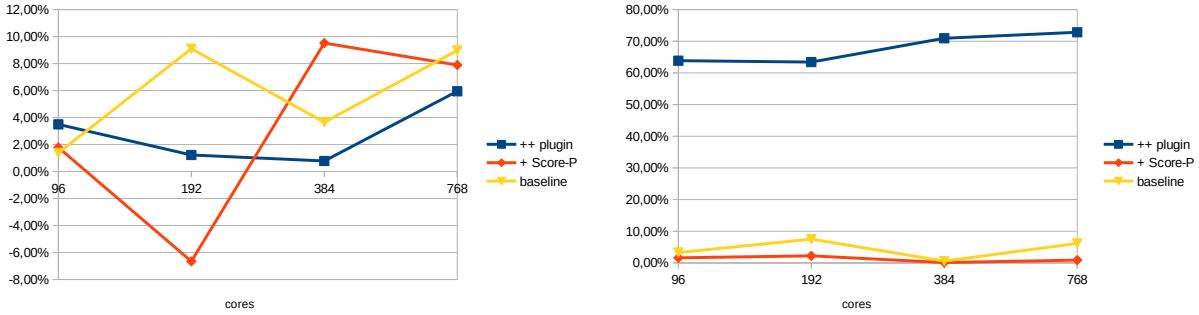
Table 6.5: Plugin's overhead when measuring code functions in geometry mode in CODA test-case.

		running time			memory (MB)		
nodes	**cores**	**baseline**	**+ Score-P**	**++ plugin**	**baseline**	**+ Score-P**	**++ plugin**
4	96	1h48m05s ± 1%	4%	5%	1197 ± 3%	0%	11%
8	192	1h07m52s ± 9%	1%	1%	983 ± 8%	11%	13%
16	384	0h42m33s ± 4%	7%	-1%	944 ± 1%	1%	2%
32	768	0h31m43s ± 9%	4%	-1%	958 ± 6%	15%	1%

Table 6.5 and Figure 6.19 show the results when measuring the selected code region in geometry mode. The high standard deviation found for the baseline values shows how strongly oscillating the numbers for this test-case are; maybe, with more measurements, the results for 4 and 16 nodes (96 and 384 cores) would oscillate as much as the other ones. Considering such high deviations, there is no clear trend to draw from the diagram, apart from the fact that the new approach shows no noticeable overhead.

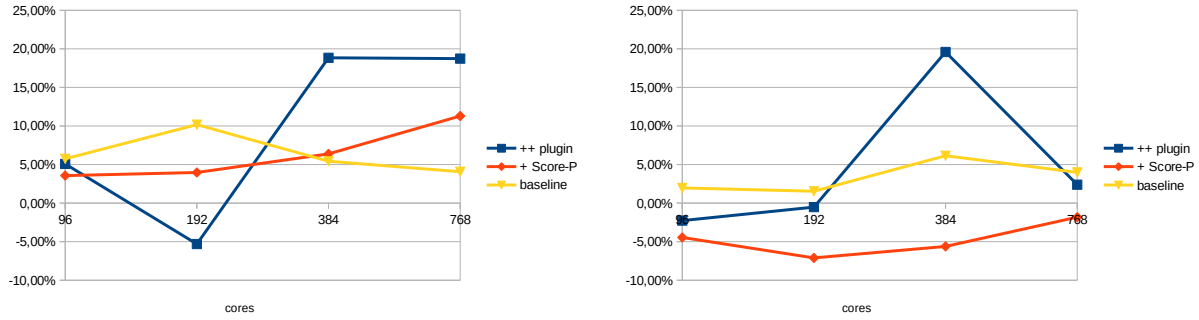
Table 6.6: Plugin's overhead when showing communication in geometry mode in CODA test-case.

		running time			memory (MB)		
nodes	**cores**	**baseline**	**+ Score-P**	**++ plugin**	**baseline**	**+ Score-P**	**++ plugin**
4	96	1h48m05s ± 1%	2%	3%	1197 ± 3%	2%	64%
8	192	1h07m52s ± 9%	-7%	1%	983 ± 8%	2%	63%
16	384	0h42m33s ± 4%	10%	1%	944 ± 1%	0%	71%
32	768	0h31m43s ± 9%	8%	6%	958 ± 6%	1%	73%



(a) Time (b) Peak memory consumption

Figure 6.20: Comparative displays of the simulation time and peak memory consumption's percentage overhead when showing communication in geometry mode in CODA test-case.



(a) Time (b) Peak memory consumption

Figure 6.21: Comparative displays of the simulation time and peak memory consumption's percentage overhead when measuring code functions in topology mode in CODA test-case.

Table 6.7: Plugin's overhead when measuring code functions in topology mode in CODA test-case.

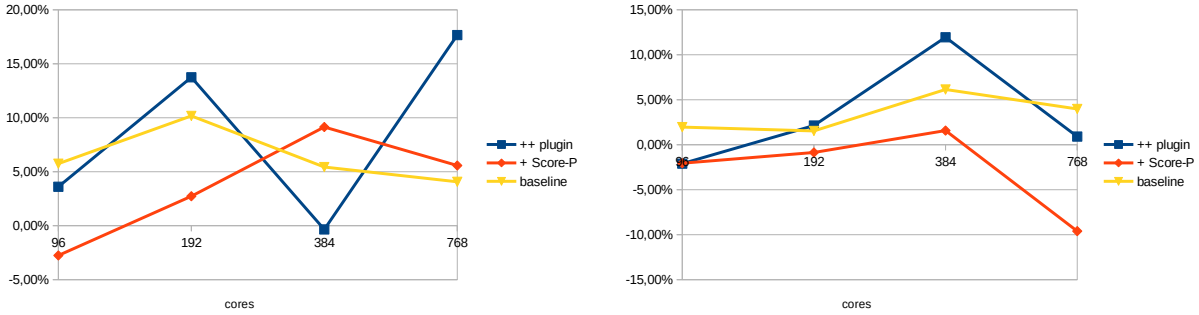
		running time			**memory (MB)**		
nodes	**cores**	**baseline**	**+ Score-P**	**++ plugin**	**baseline**	**+ Score-P**	**++ plugin**
4	96	1h46m50s ± 6%	4%	5%	1187 ± 2%	-4%	-2%
8	192	1h05m57s ± 10%	4%	-5%	972 ± 2%	-7%	-1%
16	384	0h41m36s ± 5%	6%	19%	854 ± 6%	-6%	20%
32	768	0h27m59s ± 4%	11%	19%	974 ± 4%	-2%	2%

Table 6.6 and Figure 6.20 display the results when showing communication in geometry mode. Here one trend can be observed: the high memory overhead of the plugin, which happens for the reasons explained in the previous section.

Table 6.7 and Figure 6.21 show the results when measuring the selected code region in topology mode. No clear trend can be drawn from the results (given their oscillation), apart from the fact that the overhead of the new approach is lower than the fluctuation of the original behavior.

Table 6.8 and Figure 6.22 show the results when showing communication in topology mode. Same conclusion as before: the overhead of the new approach lies within acceptable values.

As a final note, the high oscillations on the plugin curves may be partially due to the fact that they concentrate most of the I/O present in this test-case.



(a) Time

(b) Peak memory consumption

Figure 6.22: Comparative displays of the simulation time and peak memory consumption's percentage overhead when showing communication in topology mode in CODA test-case.

Table 6.8: Plugin's overhead when showing communication in topology mode in CODA test-case.

		running time			**memory (MB)**		
nodes	**cores**	**baseline**	**+ Score-P**	**++ plugin**	**baseline**	**+ Score-P**	**++ plugin**
4	96	1h46m50s ± 6%	-3%	4%	1187 ± 2%	-2%	-2%
8	192	1h05m57s ± 10%	3%	14%	972 ± 2%	-1%	2%
16	384	0h41m36s ± 5%	9%	0%	854 ± 6%	2%	12%
32	768	0h27m59s ± 4%	6%	18%	974 ± 4%	-10%	1%

7 Results

This chapter presents how the newly designed tool is applied to the selected test scenarios and which insights become possible, which would not have with conventional tools.

7.1 Industrial CFD codes

The industrial CFD codes have been introduced in Sec. 6.2.2. They will be used now to present the results of *Geometry mode* of the tool.

7.1.1 Rolls-Royce's Hydra

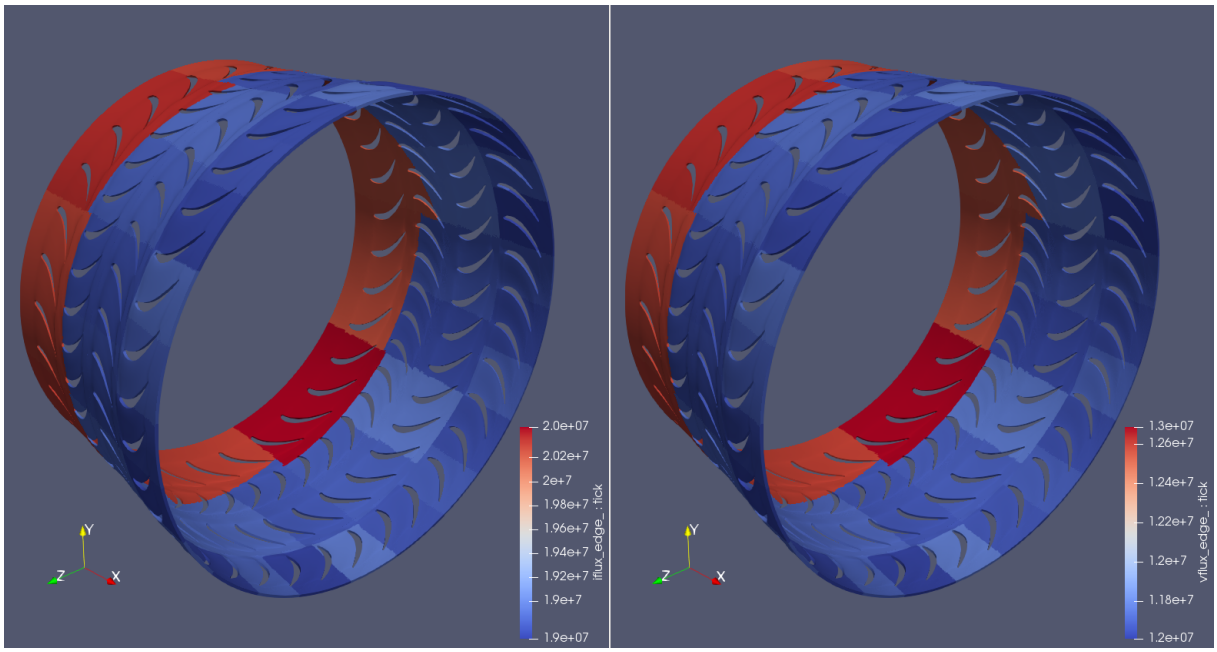


Figure 7.1: Amount of executions, in an arbitrary time-step, of two selected code functions (*iflux_edge* on the left, *vflux_edge* on the right) in Rolls-Royce CFD code's test-case. Notice the overload towards one end of the geometry (the inlet, in the negative x direction). Such correlation becomes clear due to matching the performance data to the simulation's geometry; otherwise it would have been most likely missed.

Figure 7.1 shows the amount of executions, in an arbitrary time-step, of the two selected functions inside Hydra's code; it is constant in every time-step and not subject to any stochastic variance. From the picture it is visible that ensuring a similar number of mesh *points* between the sub-domains does not necessarily mean an equally similar number of *edges*,[1] as both functions are applied at 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's inlet (the air

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

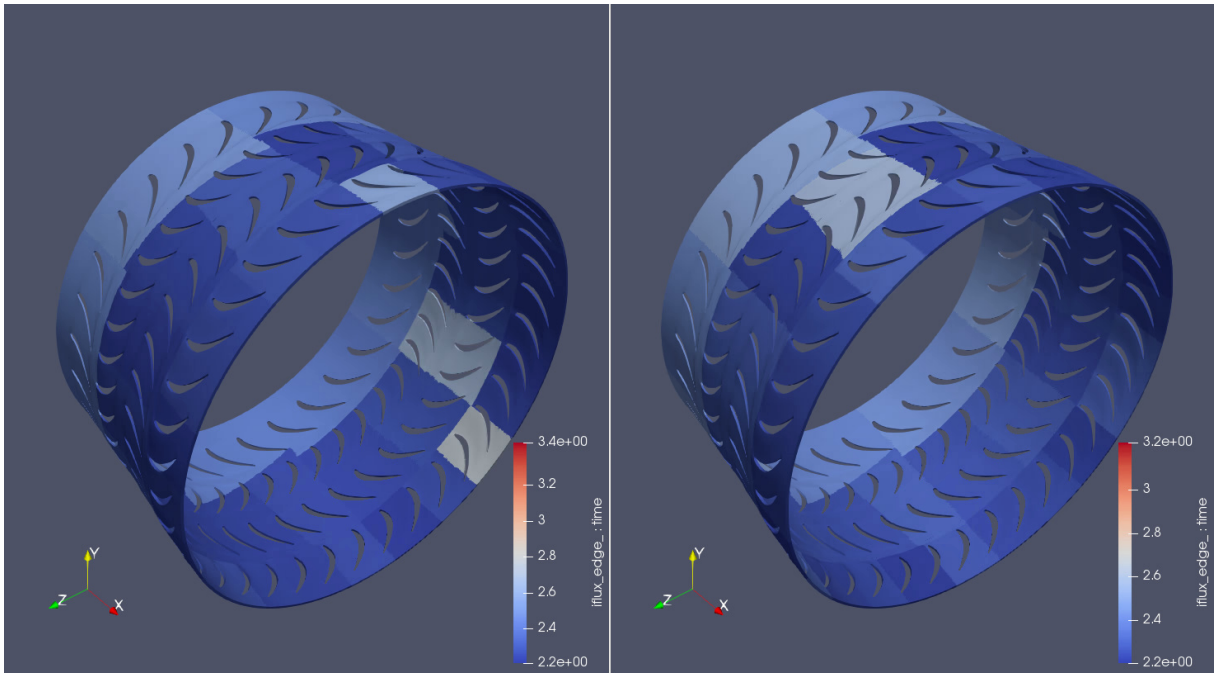


Figure 7.2: Total time spent in function *iflux_edge*, in an arbitrary time-step, on two consecutive runs of Rolls-Royce CFD code's test-case. Notice the overload towards one end of the geometry (the inlet, in the negative x direction).

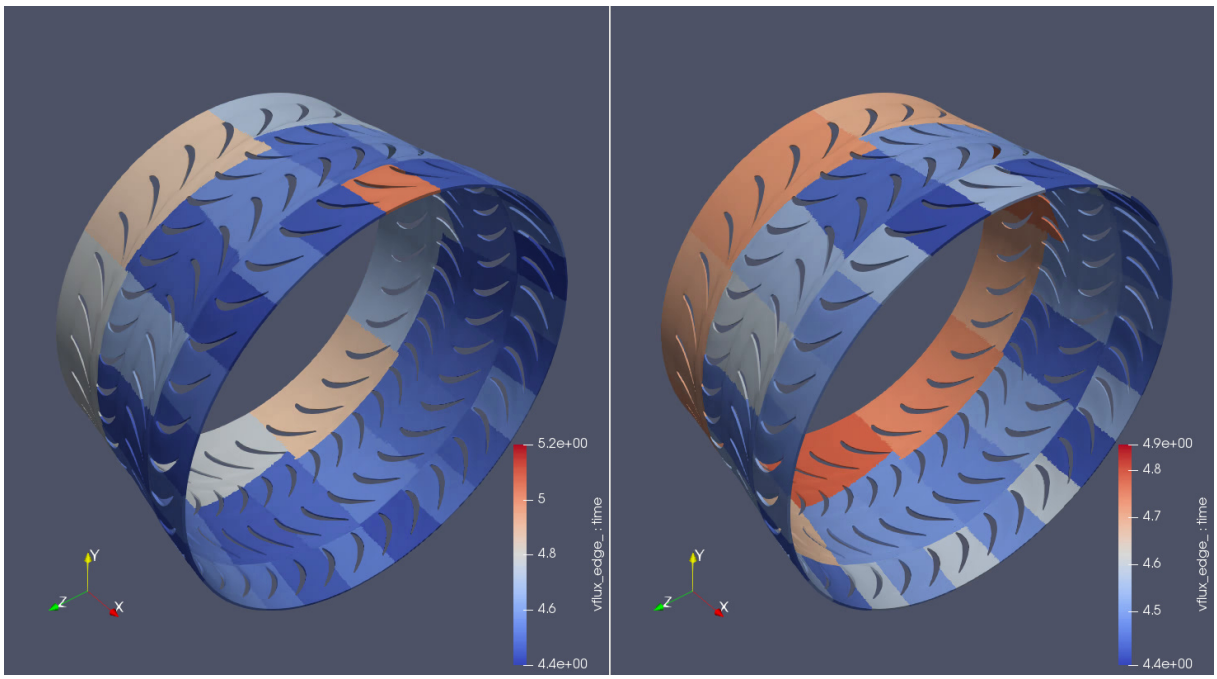


Figure 7.3: Total time spent in function *vflux_edge*, in an arbitrary time-step, on two consecutive runs of Rolls-Royce CFD code's test-case. Notice the overload towards one end of the geometry (the inlet, in the negative x direction).

flows in the positive x direction), plus the creation of a chess board-like pattern (interleaved areas of higher and lower number of executions). Such correlations confirm the benefit of matching performance data back to the simulation's geometry; otherwise they would have been most likely missed.

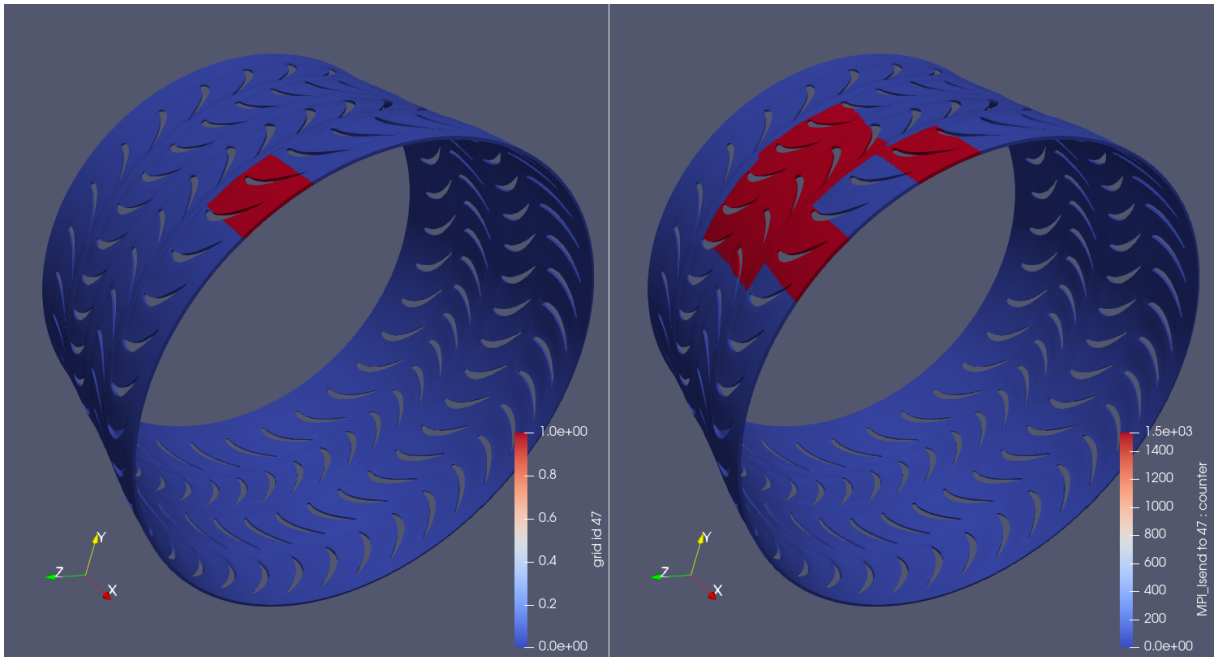


Figure 7.4: Location of an arbitrary subdomain (left) and those communicating with it (right), colored by amount of messages sent (in this case, MPI_Isend calls) in an arbitrary time-step of Rolls-Royce CFD code's test-case.

The aforementioned distortions are indeed detrimental to the code's performance. Figure 7.2 and 7.3 show the *total* – i.e. comprising all executions – time spent (in seconds) in the selected functions 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*, with a maximum difference (i.e. between slowest and fastest ranks) of up to 0.5 second (roughly 10%) within the measured sample (10 time-steps out of 200).

The benefits of the novel visualizations can also be seen on the tracing part of the tool. Figure 7.4 shows the location of an arbitrary subdomain (left) and those communicating with it (right), colored by 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 talk to the selected rank (the last one, number 47). However, this is not the case for other subdomains within the same simulation: Figure 7.5 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's 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 Figure 7.6, which corresponds to Figure 7.5, but now coloring the sender subdomains by total amount of *bytes* sent at the time-step shown. This issue was reported to Rolls-Royce: the outcomes will be presented below, when Hydra is once again analysed, but this time in topology mode.

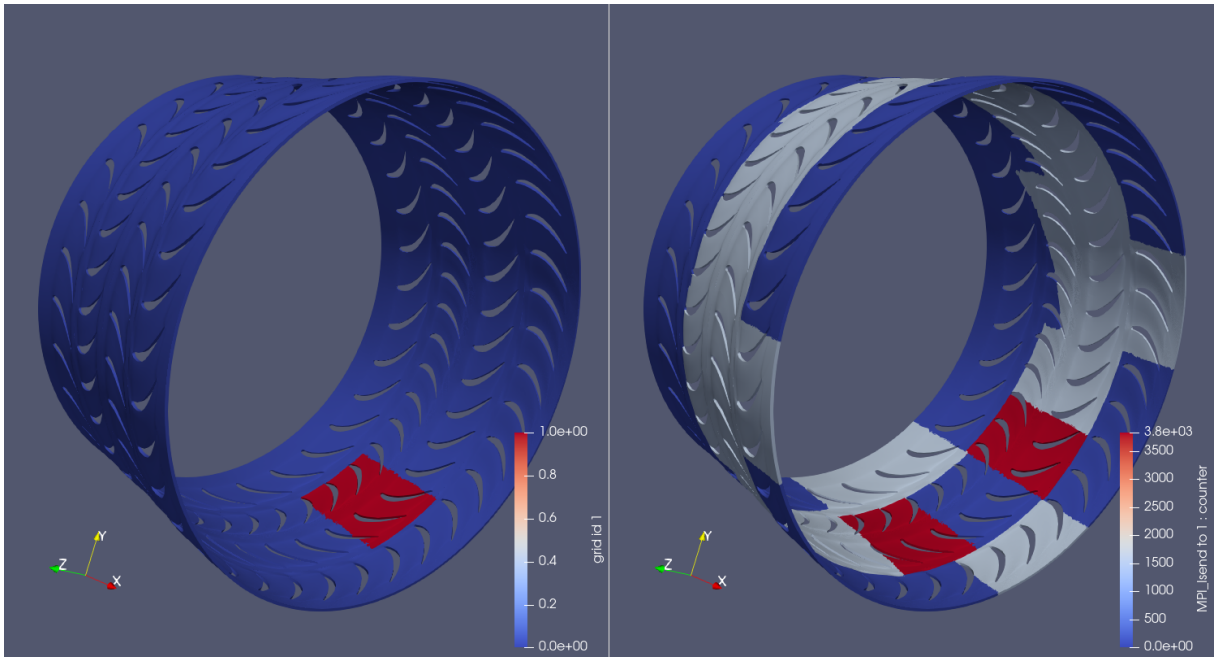


Figure 7.5: Location of another arbitrary subdomain (left) and those communicating with it (right), colored by amount of messages sent (in this case, MPI_Isend calls) in an arbitrary time-step of Rolls-Royce CFD code's test-case.

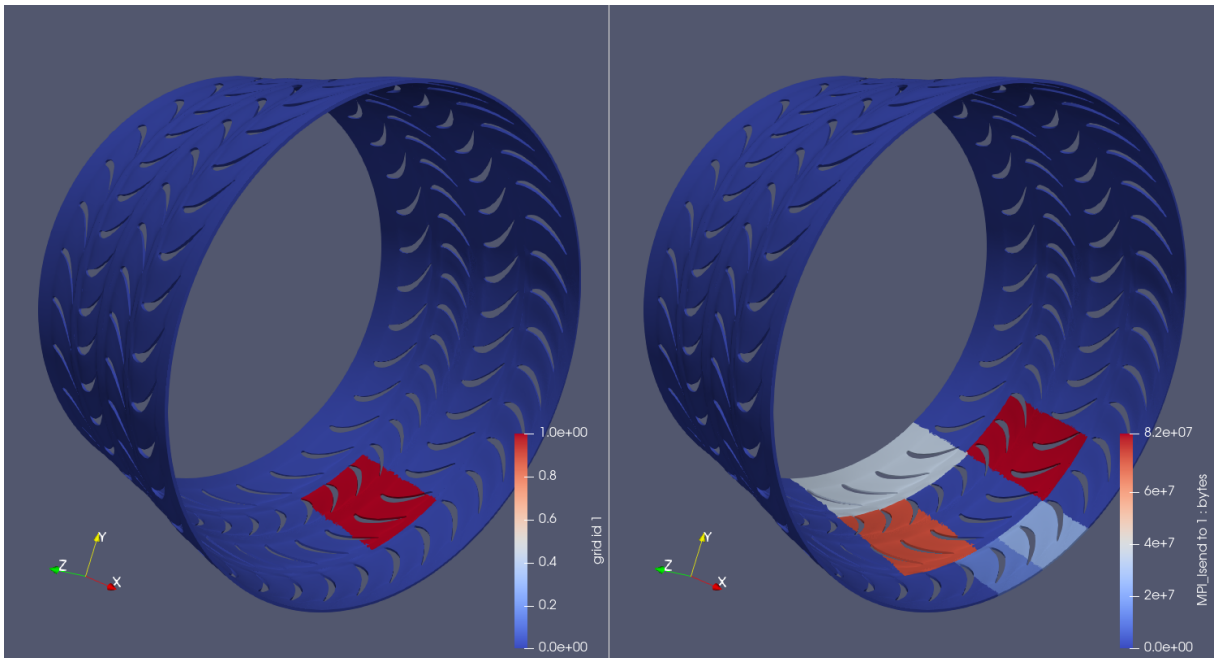


Figure 7.6: Location of another arbitrary subdomain (left) and those communicating with it (right), colored by amount of bytes sent (in this case, through MPI_Isend calls) in an arbitrary time-step of Rolls-Royce CFD code's test-case.

7.1.2 Onera, DLR & Airbus's CODA

480 cores (20 nodes) were needed in order to visualize the results of NASA's grid in ParaView (given their size). Figures 7.7, 7.8, 7.9 and 7.10 show the amount of executions of the face flux function per subdomain, shown on the left/top; and the correspondent time taken on them, shown on the right/bottom;

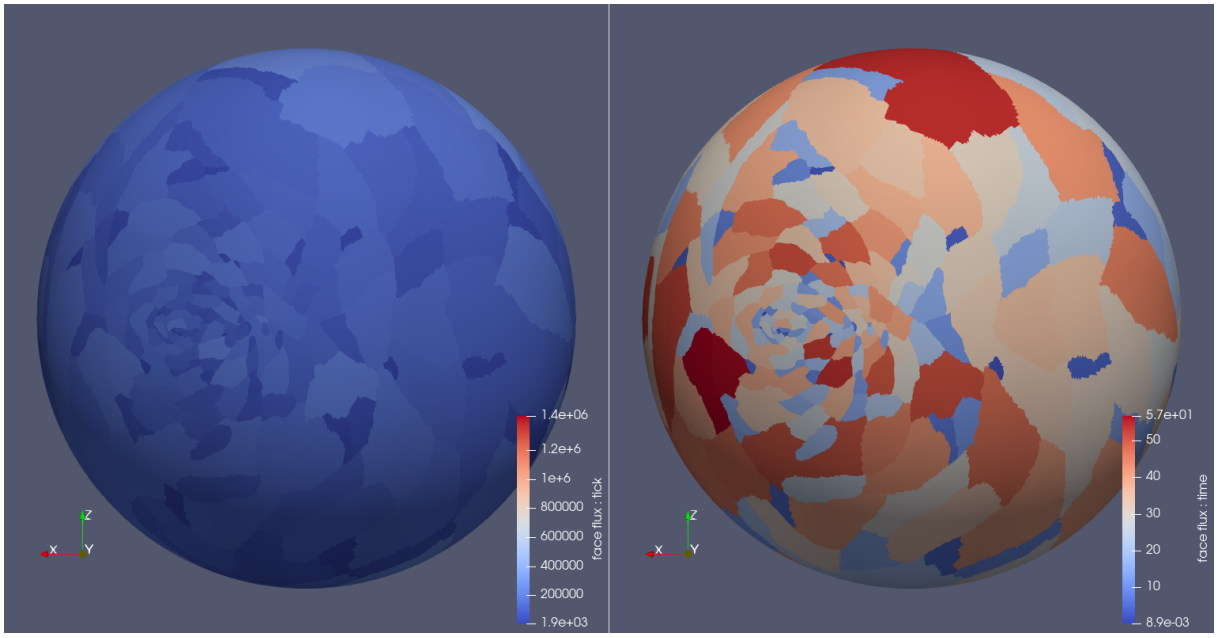


Figure 7.7: Amount of executions of the face flux function per subdomain, shown on the left; and the correspondent time taken on them, shown on the right-hand side; in an arbitrary time step of CODA's test-case (as seen from far away from the airplane's wings).

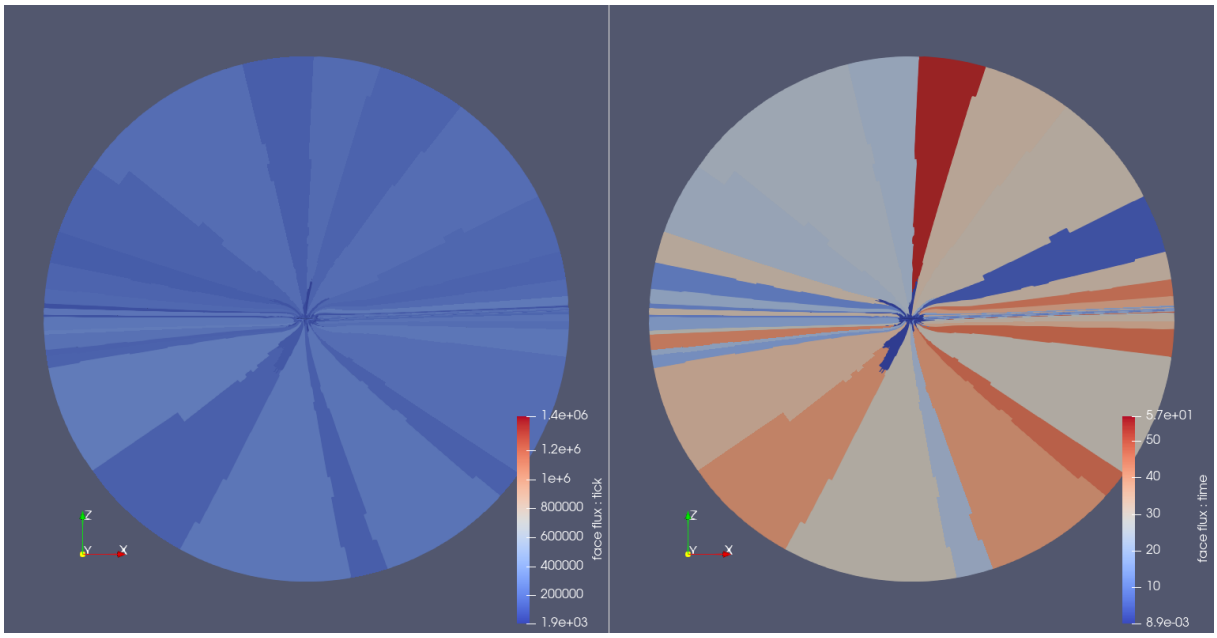


Figure 7.8: Amount of executions of the face flux function per subdomain, shown on the left; and the correspondent time taken on them, shown on the right-hand side; in an arbitrary time step of CODA's test-case (as seen from the airplane's sagittal plane).

in an arbitrary time step of CODA's test-case. It is not possible to identify the regions of higher function execution in the first three pictures, so a clipped view through the plane's transverse plane is provided in the fourth figure (showing the area immediately below the aircraft). From the pictures one can see that the regions of higher execution time seem not to be correlated with those of higher execution frequency, but rather to their location relative to the airplane, with those far from it taking longer to compute the

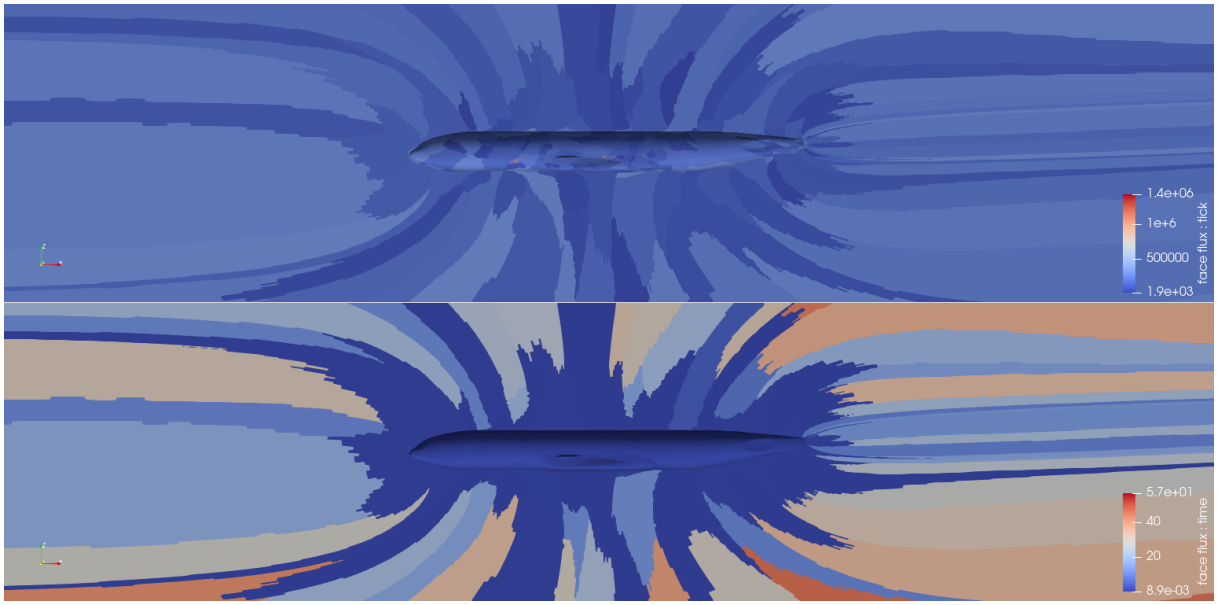


Figure 7.9: Amount of executions of the face flux function per subdomain, shown on top; and the correspondent time taken on them, shown on the bottom; in an arbitrary time step of CODA's test-case (as seen close from the airplane's sagittal plane).

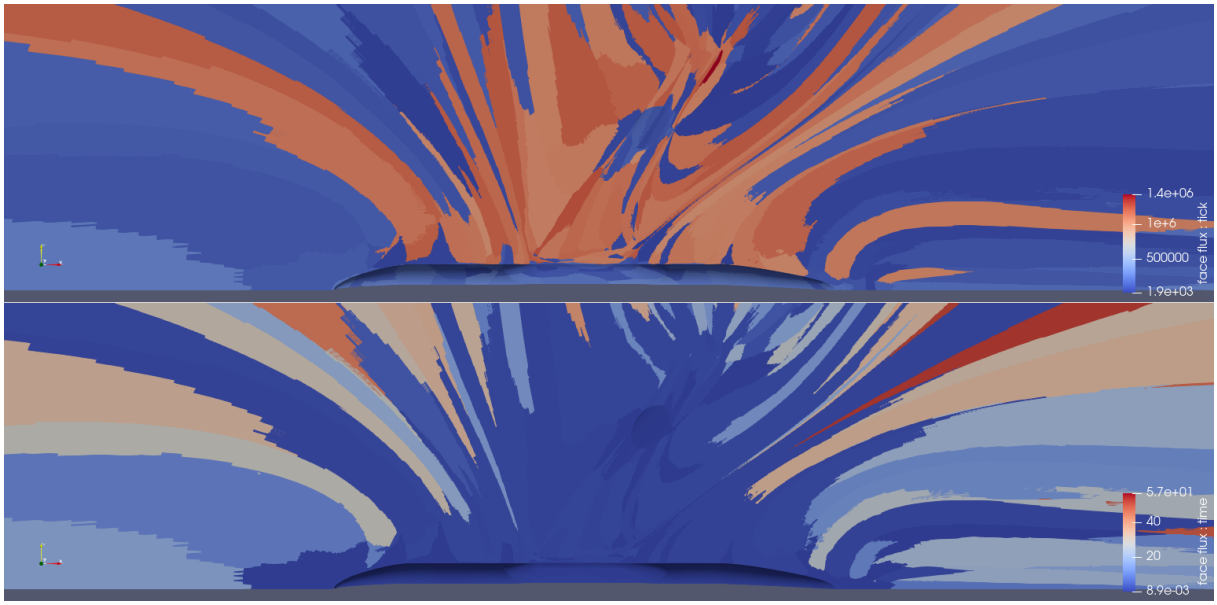


Figure 7.10: Amount of executions of the face flux function per subdomain, shown on top; and the correspondent time taken on them, shown on the bottom; in an arbitrary time step of CODA's test-case (as seen from a clipped view through the airplane's transverse plane).

face fluxes. This observation raised the suspicion that it might be related to the treatment being done to the outer boundary conditions, namely one that requires a heavy usage of the `pow()` function,[2] i.e. an inefficiency not related to CODA itself. A change was made with that regard in the simulation code (the adoption of a new boundary condition treatment), and the result can be seen in Figures from 7.11 to 7.14. Now the regions of higher frequency and higher time are synchronized, as expected.

[2]In an attempt to circumvent this inefficiency, some vendors – like AMD – provide their own version of the `pow()` function. Given here Intel hardware is being used, the version of `pow()` being used is the default Linux one.

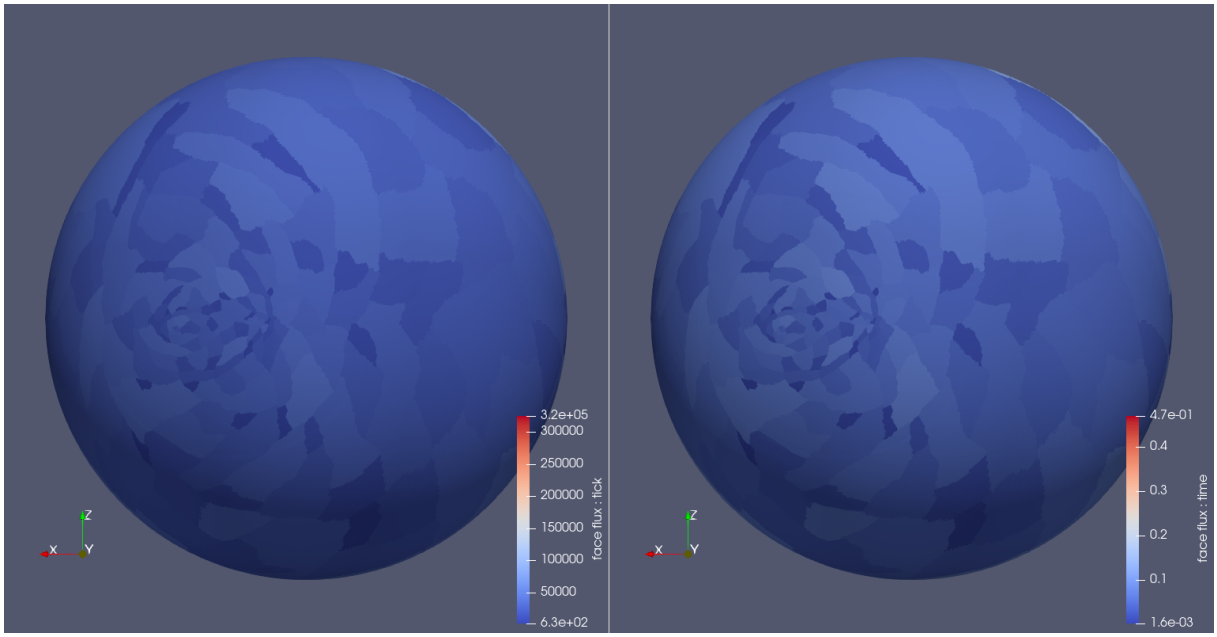


Figure 7.11: New amount of executions of the face flux function per subdomain, shown on the left; and the correspondent time taken on them, shown on the right-hand side; in an arbitrary time step of CODA's test-case (as seen from far away from the airplane's wings).

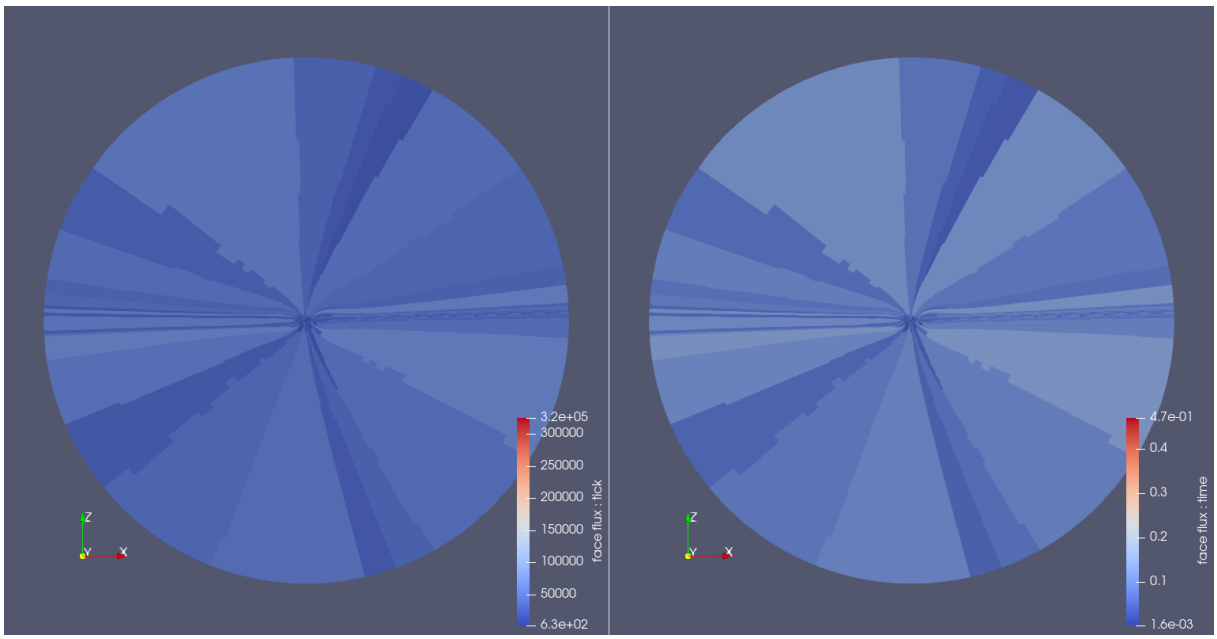


Figure 7.12: New amount of executions of the face flux function per subdomain, shown on the left; and the correspondent time taken on them, shown on the right-hand side; in an arbitrary time step of CODA's test-case (as seen from the airplane's sagittal plane).

With regards to communication, no messaging between non neighbors has been detected within the simulation. The code behavior in this aspect is clean, what could be easily confirmed by the new approach being proposed in this thesis, but would be harder to state with the conventional tools.

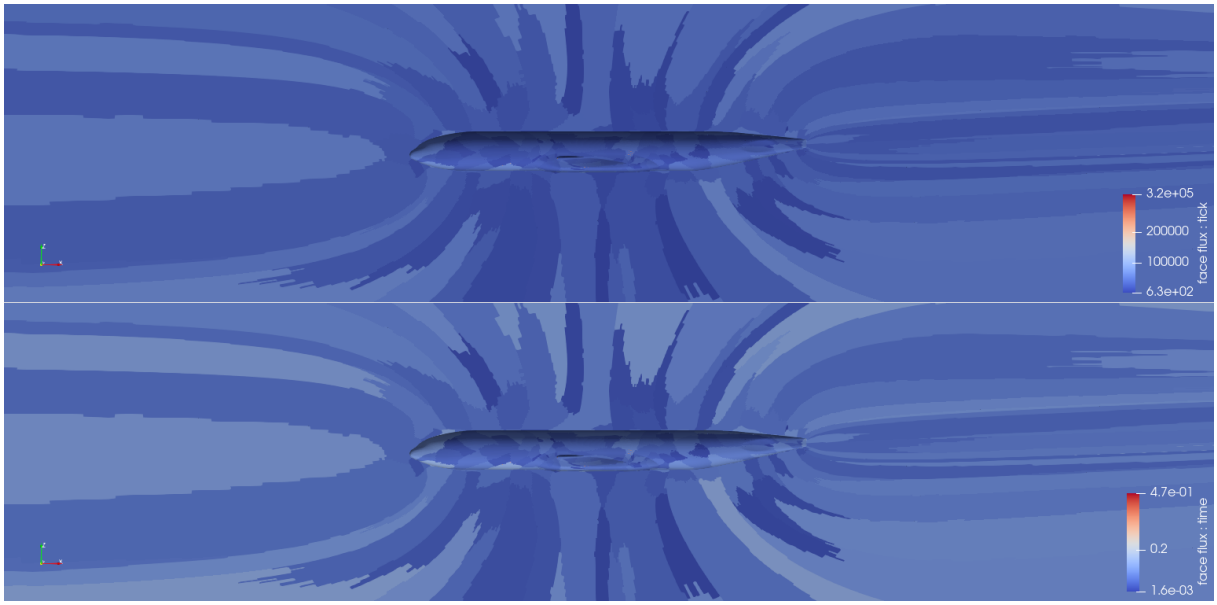


Figure 7.13: New amount of executions of the face flux function per subdomain, shown on top; and the correspondent time taken on them, shown on the bottom; in an arbitrary time step of CODA's test-case (as seen close from the airplane's sagittal plane).

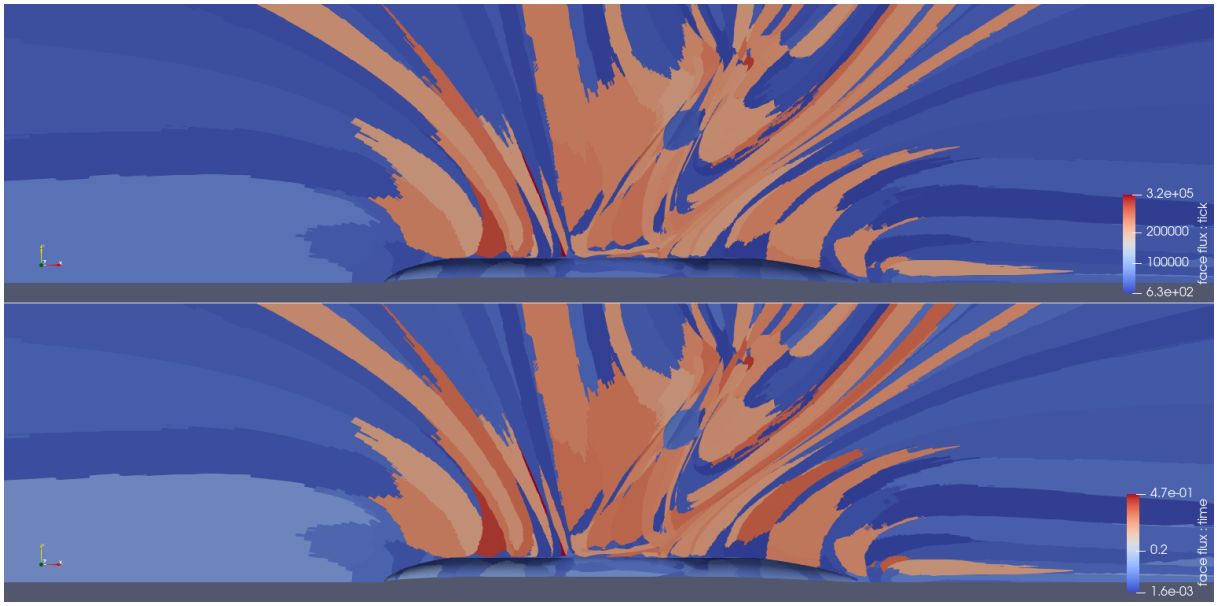


Figure 7.14: New amount of executions of the face flux function per subdomain, shown on top; and the correspondent time taken on them, shown on the bottom; in an arbitrary time step of CODA's test-case (as seen from a clipped view through the airplane's transverse plane).

7.2 Benchmarks – presenting *Topology Mode*

Given the benchmarks have no simulation grid, they will be used to present the plugin's topology mode. In the following section, the thesis will then go back to the industrial CFD codes and show the outputs of topology mode on them. In order to illustrate the capacity of the tool to adapt to the underlying hardware architecture, visualizations generated from using it in distinct topologies shall be presented. The following ones come from *Sandy Bridge* nodes, which have two sockets, each with 8 cores. Figure 7.15

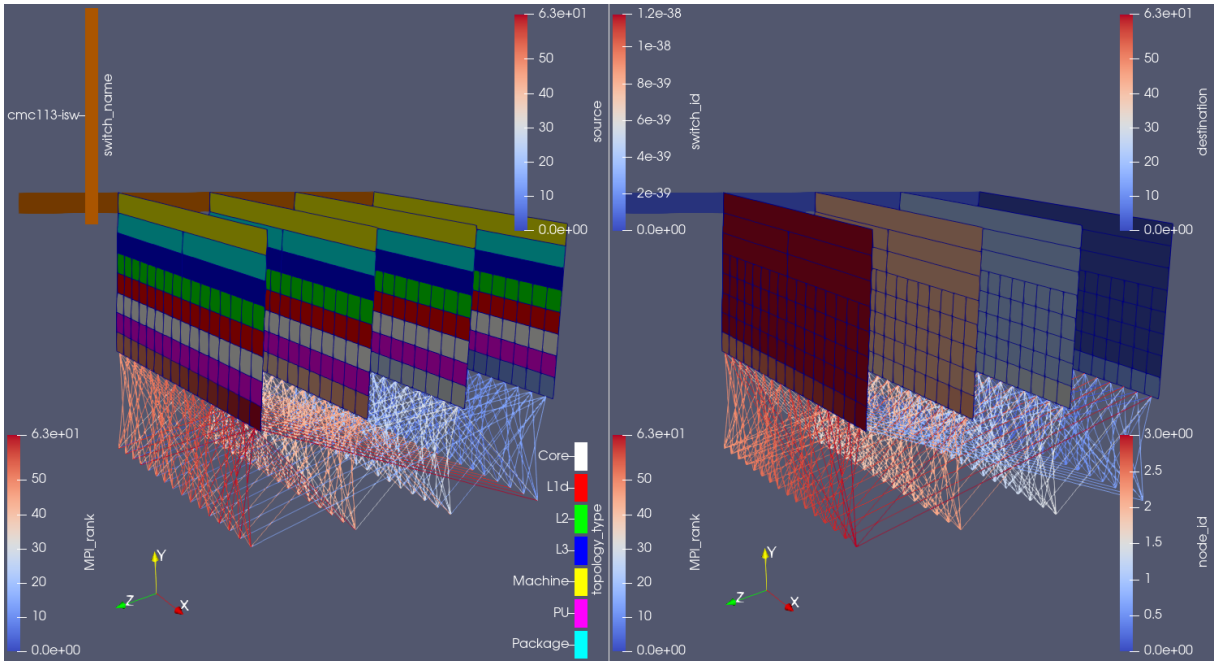


Figure 7.15: Plugin outputs in topology mode for an arbitrary time-step in the MG benchmark, visualized from the same camera angle, showing the *topology type* (left) and the *node id* (right). Network switches are colored by *name* (left) and by *id* (right). Messages sent between ranks are colored by *source* (left) and by *destination* (right).

shows the visualizations produced by the tool in topology mode 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, 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 7.15) and separated by a fixed length (adjustable at run-time through the variable `inter_node_offset` in the plugin adapter'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 and in view that drawing the entire cluster would not help the user to understand its code's behaviour[3]. This means that, between any pair of planes in Figure 7.15 there might be other compute nodes (by order of id number) in the cluster logical infrastructure; but, if that is the case, none of its cores are participating in the current simulation. The inter-node distance in ParaView does not take into consideration such

[3]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, has pros and cons that need to be considered.

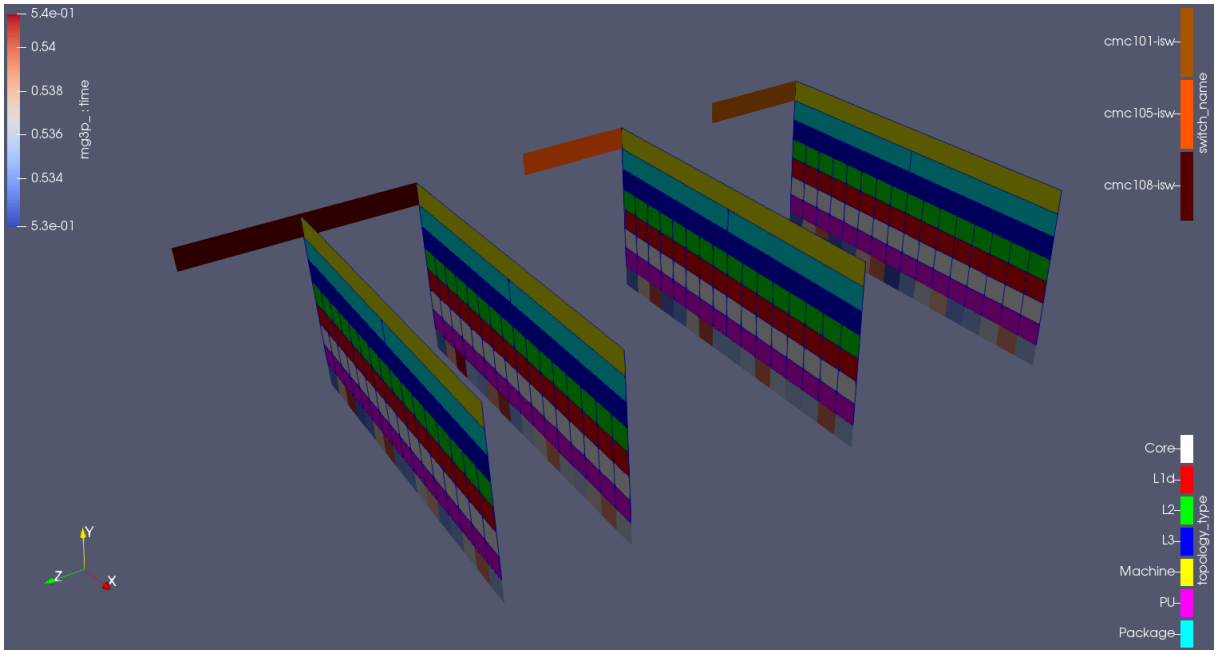


Figure 7.16: Plugin outputs for the MG benchmark. The leaf switch information is encoded both on the color (light brown, orange and dark brown) and on the position of the node planes (notice the extra gap when they do not belong to the same switch).

non used nodes.[4] It 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* (explained in Section 2.5), as shown on Figure 7.16.

Notice here how the inter-node (the inter-plane) distance in ParaView is bigger when there is a change in (leaf) network switch. This is indeed intuitive, as messages exchanged between nodes under different switches will need to *travel longer* (traverse more "hops") in order to be delivered when compared to those exchanged between nodes under the same switch. This means increased latency, hence it is wise to place together ranks which are close to one another in the simulation's geometry: the combination of the plugin's modes (geometry and topology, as it will be shown later) helps to achieve this goal.

Slurm actually 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 perspective, as to minimize their communications' latency. But just to illustrate the plugin's potential, Figure 7.17 shows the results when forcing the scheduler to arrange the processes over to many compute nodes. 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

[4] Indeed, all nodes under the same leaf switch are equally close to each other; i.e. the order of id number is logical, but not necessarily physical.

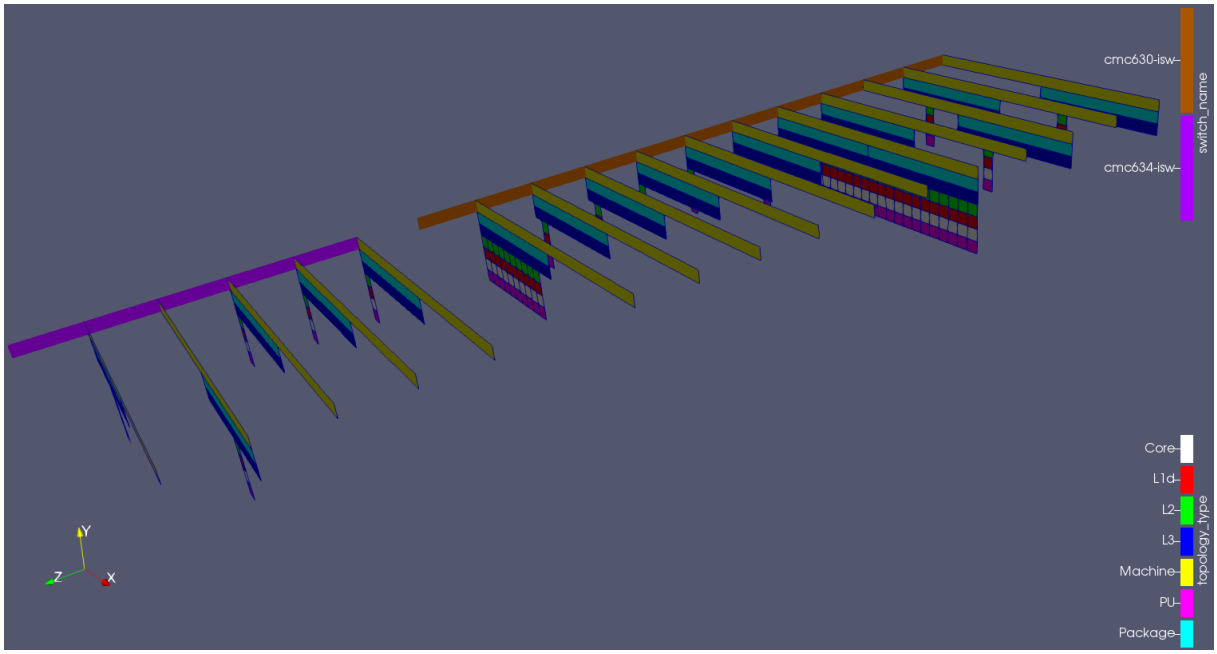


Figure 7.17: "Messed up" distribution of ranks across compute nodes, for illustration purposes.

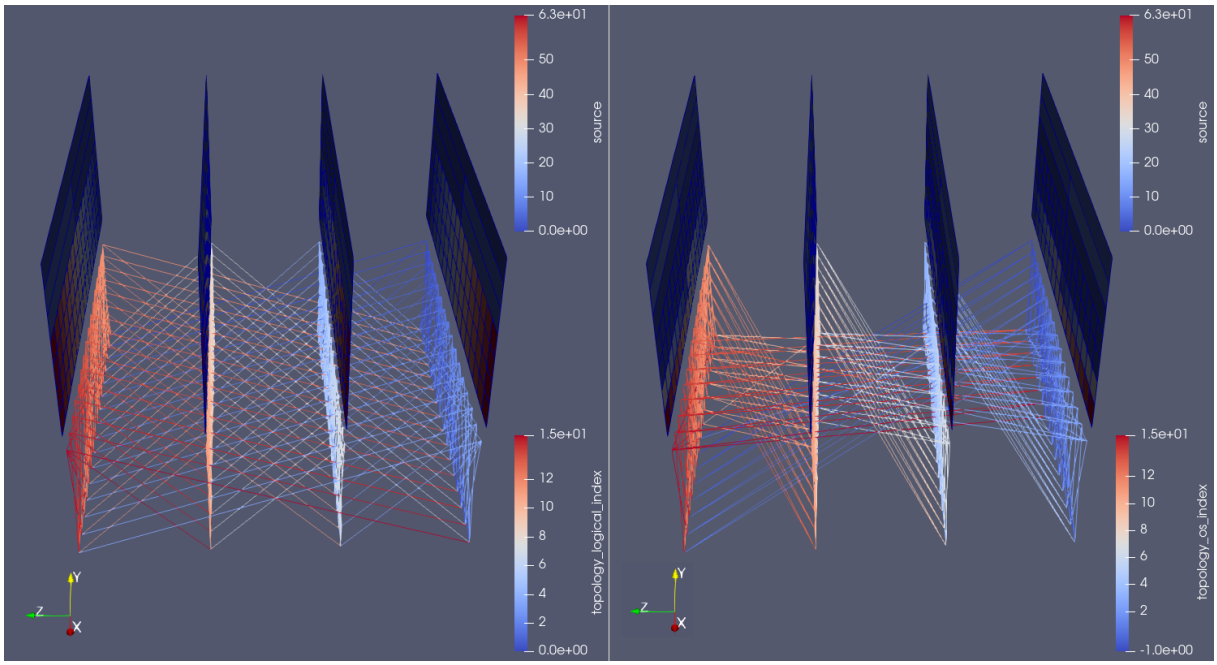


Figure 7.18: Side-by-side comparison of the communication pattern between the MG (left) and BT (right) benchmarks, at an arbitrary time-step, colored by source rank of messages. Notice how the manipulation of the camera angle (an inherent feature of visualization software like ParaView) allows the user 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.

start from the bottom of the sending rank and go downwards towards the receiving one. This way, it is possible to distinguish, and simultaneously visualize, messages sent from A to B and from B to A and inter-node from intra-node communication. In Figure 7.18, notice how all ranks on both benchmarks talk either to receivers within the same node or the nodes immediately before / after. The big lines connect-

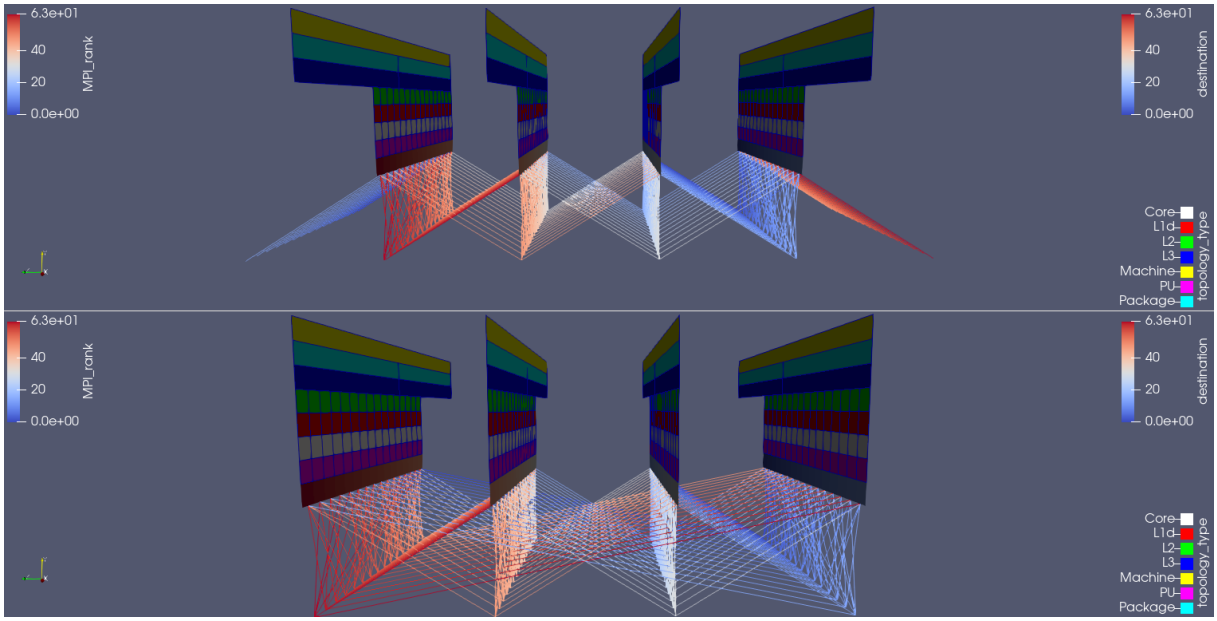


Figure 7.19: Side-by-side comparison of the communication pattern in the MG benchmark when using the periodic boundary condition feature (top) and not (bottom). The communication lines are colored by destination rank of the messages. Notice how the periodic nature of this test-case's boundary conditions become clearer in the top picture: all ranks talk either to those located on the same node, or to those located on the node immediately before / after.

ing the first and last nodes suggest some sort of periodic boundary condition inside the grid. Finally, the "grid" (the lines) can also be colored by total amount of bytes sent through each channel in that time-step.

In case of periodic or cyclic communication patterns, the previously shown visualization of Figure 7.18 causes clutter: lines between cores in the first and last nodes will need to cross the entire visualization space, making it harder to understand. Therefore, and improved variant was designed for this frequent case; it can be activated by means of the variable `periodic_BC` in the plugin adapter's input file. Its outputs are visible in Figure 7.19. It shows the topology when using *Haswell* nodes, which have two sockets, each with 12 cores. 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: all ranks talk either to those located on the same node, or to those located on the node immediately before / after.

7.2.1 Comparison with previous approaches

As mentioned before, displaying three-dimensional representations of clusters is not new; for instance, see Figures 3.3 or 4.8. However, the novel approach being proposed in this thesis brings features which the previous ones would be unable to provide, e.g.:

- detailed view up to topology component level (i.e. in which core of which socket of which node a specific MPI rank is running);
- native association with the simulation's time-step (and not with a logical time, as in Figure 4.6); this is an aspect by which our approach is application-developer friendly: the time-step is the code execution delimiter the developers of CFD codes are naturally used to deal with;

- 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 (see Figure 5.5);
- individually applicable color scale to each element of the visualization, allowing, for example, to color the communication lines by amount of bytes sent;

And finally, 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. This also means that there is no need to learn how to use a new visualization tool for the sake of analysing the performance data: just stick with the software already used to analyse the simulation's native results themselves (with ParaView).

7.3 Topology mode on the industrial CFD codes

7.3.1 Hydra

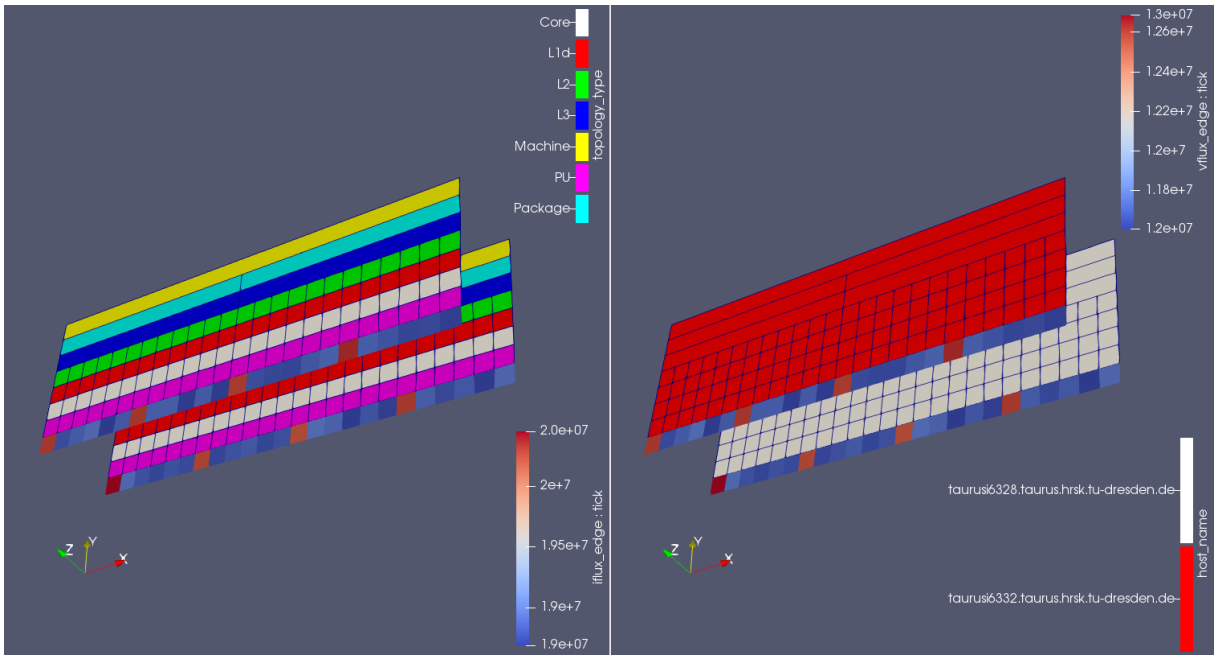


Figure 7.20: Topology mode of the plugin being used in Rolls-Royce's CFD code, showing the total amount of executions, in an arbitrary time-step, of the selected subroutines (*iflux_edge* on the left, *vflux_edge* on the right); from the same camera angle in both sides of the picture, but depicting the *hardware topology type* on the left, whereas the *host name* (i.e. name of the compute node running the simulation) on the right.

Topology mode can also be used in Hydra CFD code. For instance, Figure 7.20 shows the plugin outputs in ParaView when measuring the selected code subroutines: notice how every sixth rank (starting from the first one) gets overloaded in both functions (the red squares in the bottom rows of the planes).

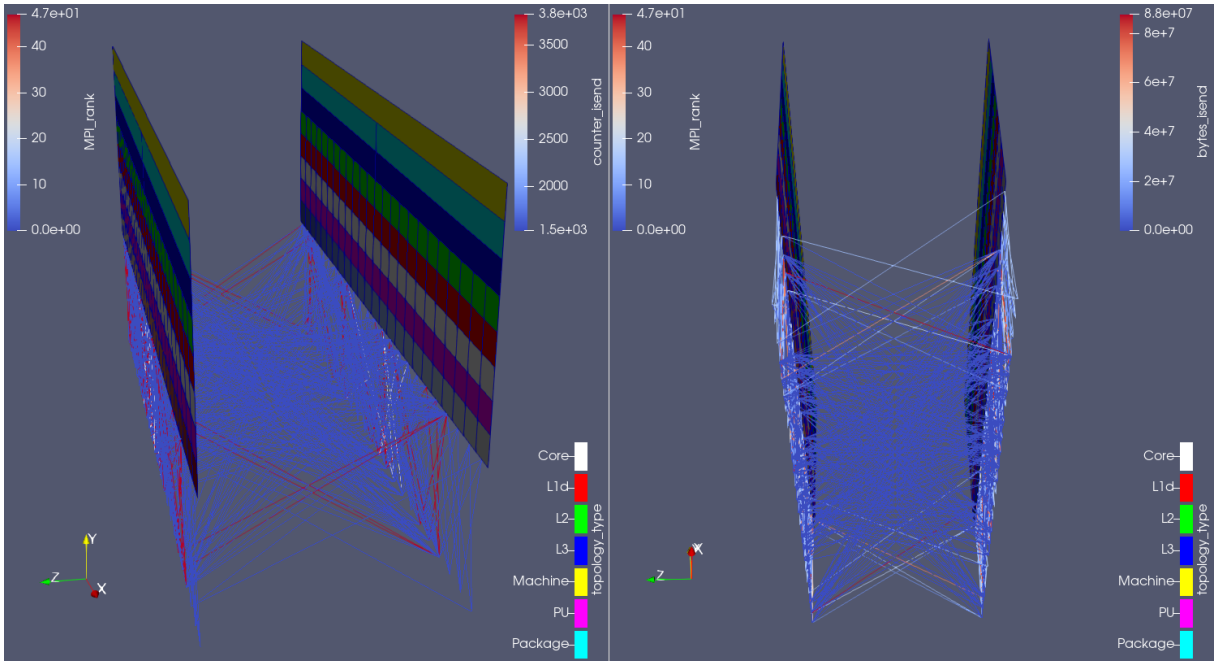


Figure 7.21: Topology mode of the plugin being used in Rolls-Royce's CFD code, showing the overall state of communications in an arbitrary time-step, from two different camera angles, depicting the total number of MPI_Isend calls on the left, whereas the total amount of bytes sent through those calls on the right.

This pattern in the load imbalance might suggest some relation to data locality, but from the results of geometry mode it is known that this is actually being caused by the mesh partitioning (overloading the subdomains located in the geometry's inlet). The every sixth rank overload frequency is then most likely a coincidence, coming from the size of the test-case combined with the number of MPI ranks being used.

Topology mode can also be used to show the big picture of communications within Hydra. On Figure 7.21, notice on the right side how many of the communication channels (the lines) did not properly transfer any data in the depicted time-step (their color is blue, which from the scale is mapped to zero) and should therefore be removed. Inclusively because the least used channels were used at least 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 involved).

Such results were submitted to Rolls-Royce, whose developers fixed the issue. The new communication behavior can be seen in Figure 7.22. 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 would be difficult if using the currently available tools.

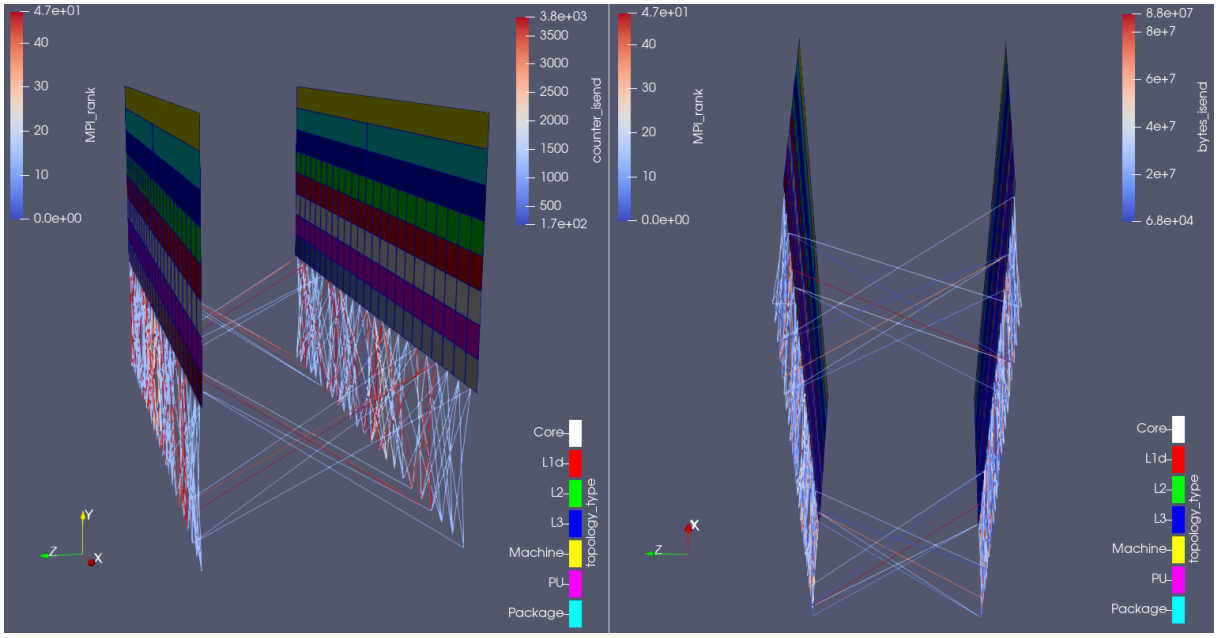
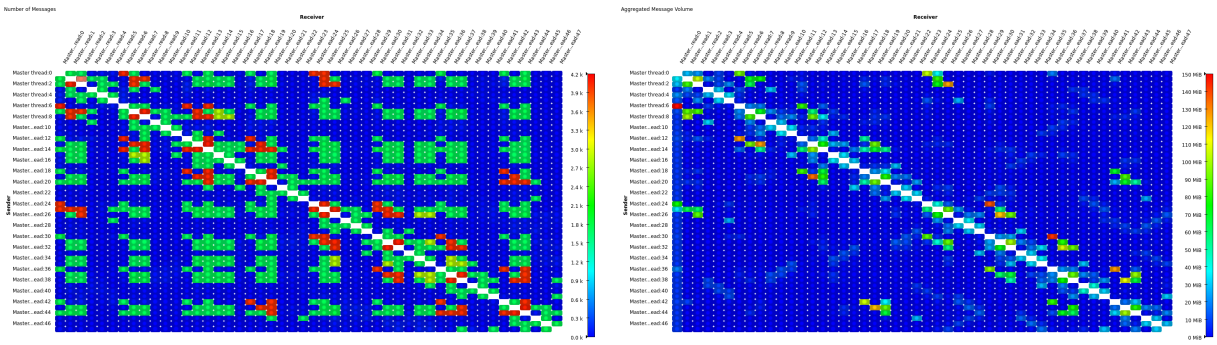


Figure 7.22: Visualization of the new communication pattern in Hydra from two different camera angles, at an arbitrary time-step, colored by number of MPI_Isend calls (left) and total amount of bytes sent on those calls (right) on that time-step.



(a) Number of messages sent

(b) Amount of data sent in those messages

Figure 7.23: Comparative displays of the number of messages sent and amount of data sent in those messages after the execution of 1 time step in Hydra test case, shown on Vampir 2D matrices.

To reproduce such analysis via the existing 2D communication matrix display in Vampir has actually been attempted. This was done with the original, i.e. unoptimized, version of the code. A single time step was traced due to overhead considerations (higher overhead for full event tracing when compared to using Score-P only for its substrate plugin API) and to explicitly isolate a single time step so that it direclty corresponds to Figures 7.21 and 7.22.[5] The results can be seen in Figure 7.23: dark blue entries correspond either to no messages or to messages with zero bytes sent between the sender and receiver ranks. With this, it is indeed possible to identify areas with high number of messages (the green spots on the left picture), but no (or few) bytes sent (no corresponding patterns on the right picture).

However, with the existing Vampir visualization and similar visualization schemes it is impossible to see the hardware topology next to the communication behavior. In the new visualization scheme in Fig-

[5]This could also be manually achieved by selecting a correct time interval in Vampir, but there is no straightforward way in this tool to isolate the events pertaining to a single simulation time step.

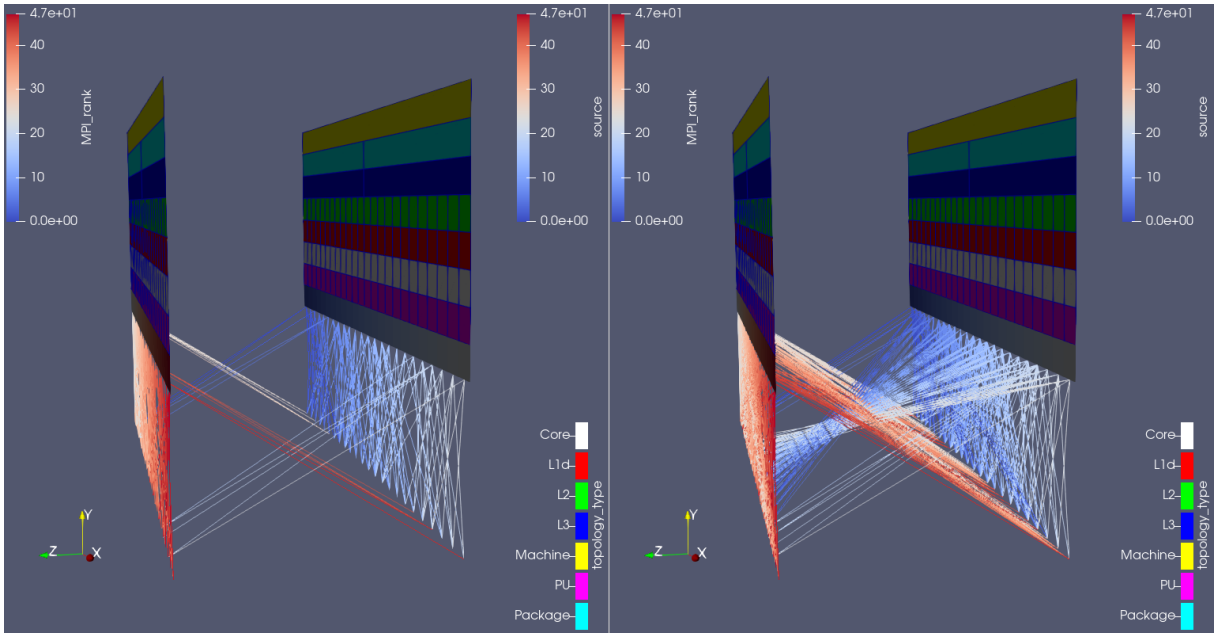


Figure 7.24: Topology mode of the plugin being used in Rolls-Royce's CFD code, comparing (from the same camera angle) the overall state of communications in two different time-steps: when (right) and when not (left) Hydra is saving its native outputs to disk. The analysis reveals a burst in point-to-point communication when that is the case, what is undesired (it would have been better to use collective MPI calls).

ure 7.22 one can clearly tell that there are two compute nodes with a number of CPU cores each. It is most obvious whether a message is transmitted between neighbor CPU cores or between cores in different compute nodes. It is also easy to tell apart the good communication pattern in Figure 7.22, where many messages are exchanged within a node, but few between the nodes; from the worse communication pattern in Figure 7.21, where many messages are exchanged between nodes. In contrast, the existing communication matrix displays show only the effects of the hardware topology on the performance behavior. Sometimes, there is a clear situation where message speeds between neighboring ranks are high, but much lower between far-apart ranks. This results in a block-diagonal picture similar to Figure 7.23b if it would show message speed. This is obviously caused by inter-node vs. intra-node communication speeds and easy enough to explain for an expert. But what about subtle situations where this is hardly visible or where it should be visible but isn't? Then the representation of the pure hardware topology is missing. Therefore, the topology view as presented here provides a clear advantage to the analyst.

Finally, one of ParaView's native – and most useful – features is the capacity to play the time-steps associated with the data. Doing that to Rolls-Royce's code reveals another important issue (which was hidden until then). On Figure 7.24, the left-hand side view shows the overall state of communications in an ordinary time-step, after the developers intervened in the code (following the revelations as shown on Figure 7.21); the right-hand side view shows the same thing, but in the last time-step of the simulation. A burst in point-to-point communication can be seen. Further tests were then conducted and led to the conclusion that this burst happens not necessarily in the last time-step, but actually whenever Hydra is saving its native outputs to disk. However, such burst is undesired: it would have been better to use collective MPI calls. Yet one more performance issue made visible by the plugin.

7.3.2 CODA

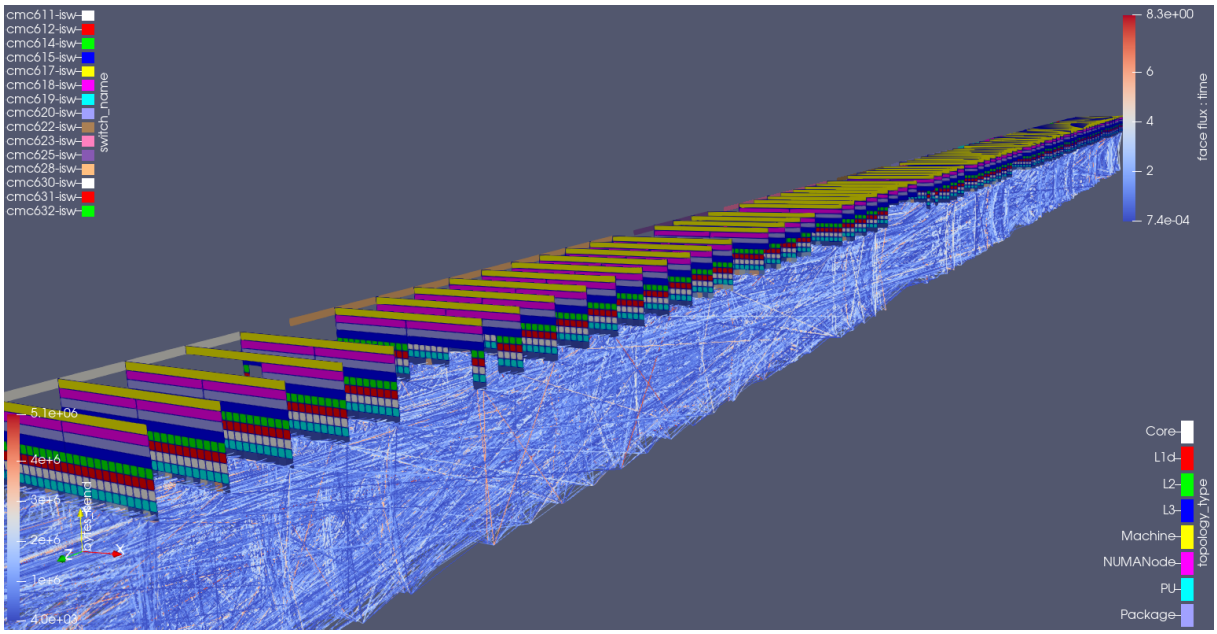


Figure 7.25: Topology mode's results for running CODA's test-case with actually more cores (1536) than advised from the test-case's scalability analysis (768). The face flux function execution time seems to be uncorrelated to the position of the respective core within the machine topology, both from an intra and from an inter-node points of view (see the bottom row squares at each plane and compare with the scale on the top-right corner).

Running the CODA test-case with 1536 cores (equivalent to 64 nodes) actually goes beyond its scalability recommendation; but in order to illustrate the capacity of the plugin to scale and adapt to increasingly bigger amounts of resources, the results for this mode when running the code under these conditions shall be presented. The face flux function execution time seems to be uncorrelated to the position of the respective core within the machine topology, both from an intra and from an inter-node points of view (see the bottom row squares at each plane on Figure 7.25 and compare with the scale on the top-right corner). This represents the expected behaviour from a purely local (i.e. without communication) function, as long as all pieces of hardware being used are operating properly (what is fortunately being confirmed here).

Finally, if the amount of communication lines in Figure 7.25 seems overwhelming, it is possible to use ParaView dedicated visualization features to filter them, as shown on Figure 7.26: the same lines are now portrayed solely by their endpoints, and with the assistance of the spreadsheet view, it is possible to select and show only an arbitrary set of the lines (here, all those originating from the first rank).

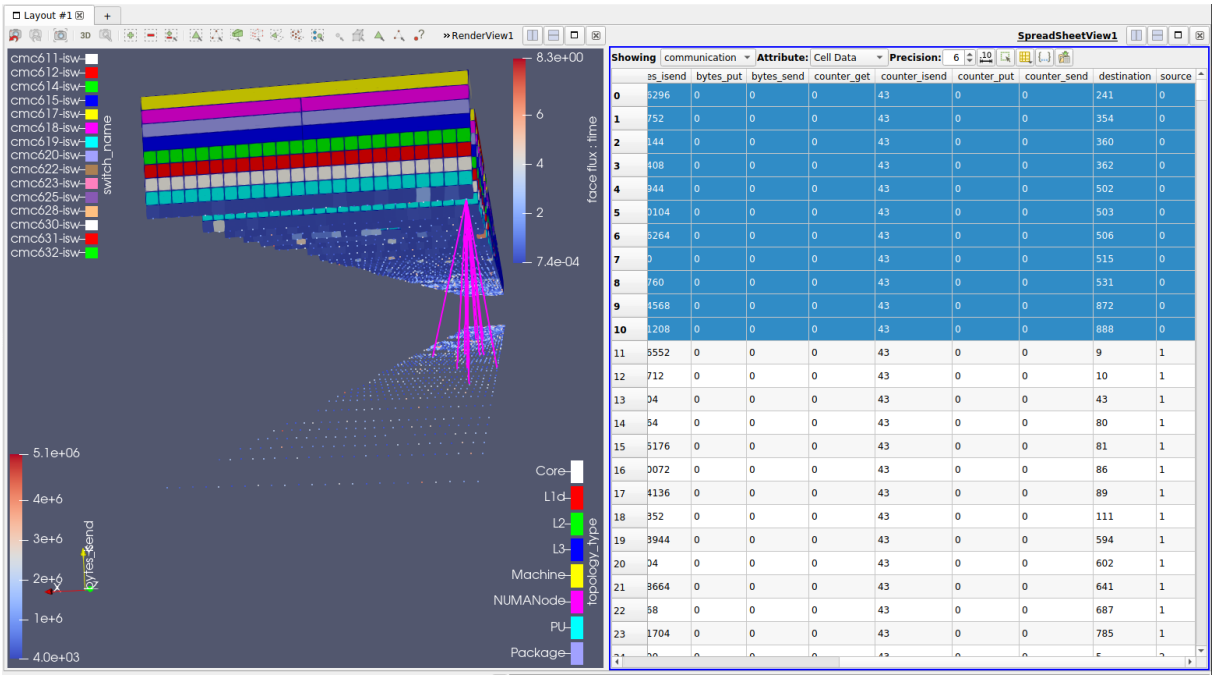


Figure 7.26: Topology mode's results for running CODA's test-case with actually more cores (1536) than advised from the test-case's scalability analysis (768). It is possible to use ParaView dedicated visualization features to filter the communication lines: they are now portrayed solely by their endpoints, and with the assistance of the spreadsheet view, it is possible to select and show only an arbitrary set of the lines (here, all those originating from the first rank).

7.4 A glance into new possibilities

The views shown for topology mode in the previous figures represent just the first of new possibilities when it comes to visualization of performance data in parallel (CFD) algorithms. Now that ParaView has been involved into the process, a new spectrum has been openned for investigation and trial. The topology of the computational resources being used by the simulation can be represented in any form; the planes shown above can be placed side-by-side, in a circle (see Figure 7.27), other geometrical configurations can be used etc. In particular, the visualization concept of *details on demand* can be applied to gradually reveal the information as the user manipulates the views on the screen, or to highlight the parts of the data which the user is currently zooming into (and hide the others) etc. This is especially useful for the visualization of large pieces of information as the one shown on Figure 7.25 above.

One of such alternative views shall be discussed with more detail: the rotated views. Consider again Figure 7.27: it shows the results for topology mode in an arbitrary time step on the MG benchmark. This time the planes are displayed rotated about a center of curvature. This can be easily achieved in ParaView[6] , by applying the calculator filter to produce a coordinate transformation on the points of the grid. In the case shown, the transformation formula is:

$$(coordsX + A) * (cos(2\pi/B * node\_id) * iHat + sin(2\pi/B * node\_id) * kHat) + coordsY * jHat$$

[6]I.e. after the simulation is finished, or the time step information has been received (in a live visualization). No need to modify the plugin's source code.

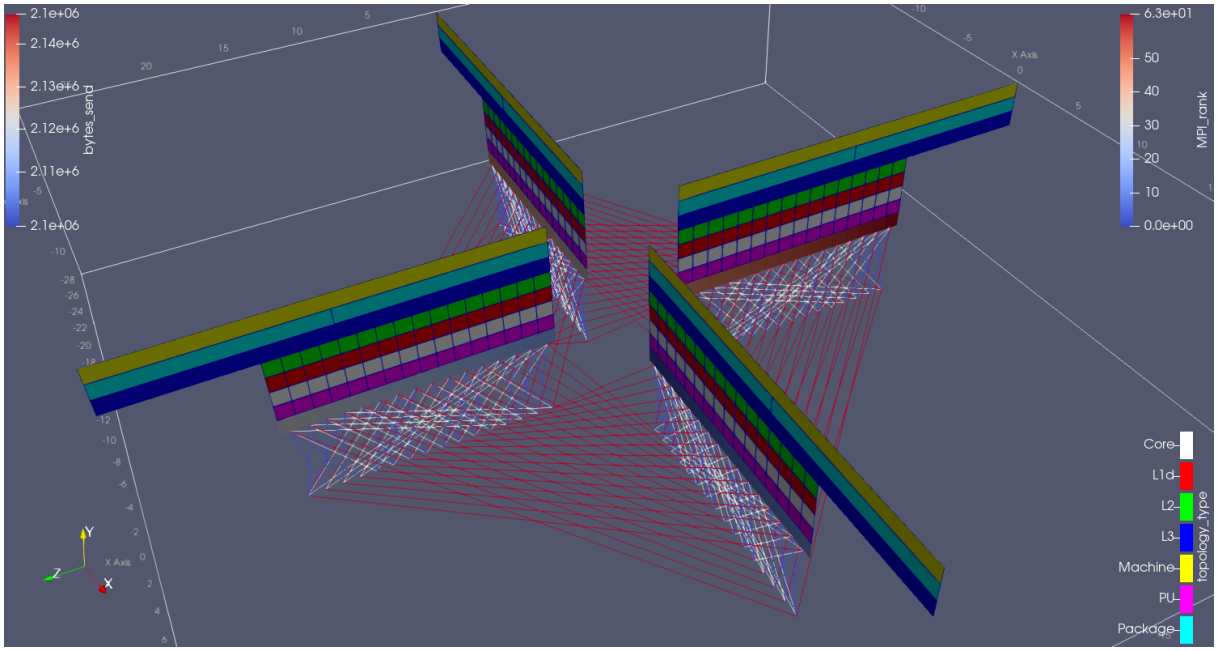


Figure 7.27: Alternative way of displaying topology mode's information in ParaView, for illustration purposes. It shows the results in an arbitrary time step on the MG benchmark. Here the planes are displayed rotated about a center of curvature.

Where A is an arbitrary inner radius to be applied to the rotated planes, such that they do not all start from the very same radial coordinate[7]; and B is the number of planes in the visualization, which is also the number of compute nodes used in the simulation. $coordsX$, Y and Z; i, j and $kHat$ are variables naturally provided by ParaView. Finally, $node\_id$ is a data array provided by the plugin. Notice how the periodic nature of the boundary conditions becomes visible, in the communication pattern, even better than shown before: each core talks either to peers in the same compute node, or in compute nodes immediately before/after. Notice also how the heaviest channels (the ones transmitting more data) are precisely the ones crossing the boundaries between nodes (see the color of the lines and compare with the scale on the top-left corner): this is the opposite behavior to the ideal one (leave the heaviest communication inside the node, in order to decrease their latency).

The equation shown above works for any number of planes in the visualization. For instance, see Figure 7.28: it shows the results for topology mode in an arbitrary time step on the BT benchmark. Notice how the diagonal nature of this test case becomes even more visible in the rotated view: communication is heavy (the respective lines are red) when going to the following cores (both from an intra and an inter node perspectives), whereas light (the lines are blue) when going to the previous cores.

The new possibilities do not end here. By applying another coordinate transformation before the one shown above, namely the equation:

$$(coordsY * (cos(-\pi/2) + 1) + 7) * iHat + coordsX * sin(-\pi/2) * jHat + coordsZ * kHat$$

It is possible to further rotate the objects and produce visualizations as the ones shown on Figure 7.29.

[7]It is recommended to set it to the same value of constant B (see ahead). This way, the bigger the amount of planes in the visualization, the bigger the inner radius of the circumference, as the angular displacement between the planes gets smaller.

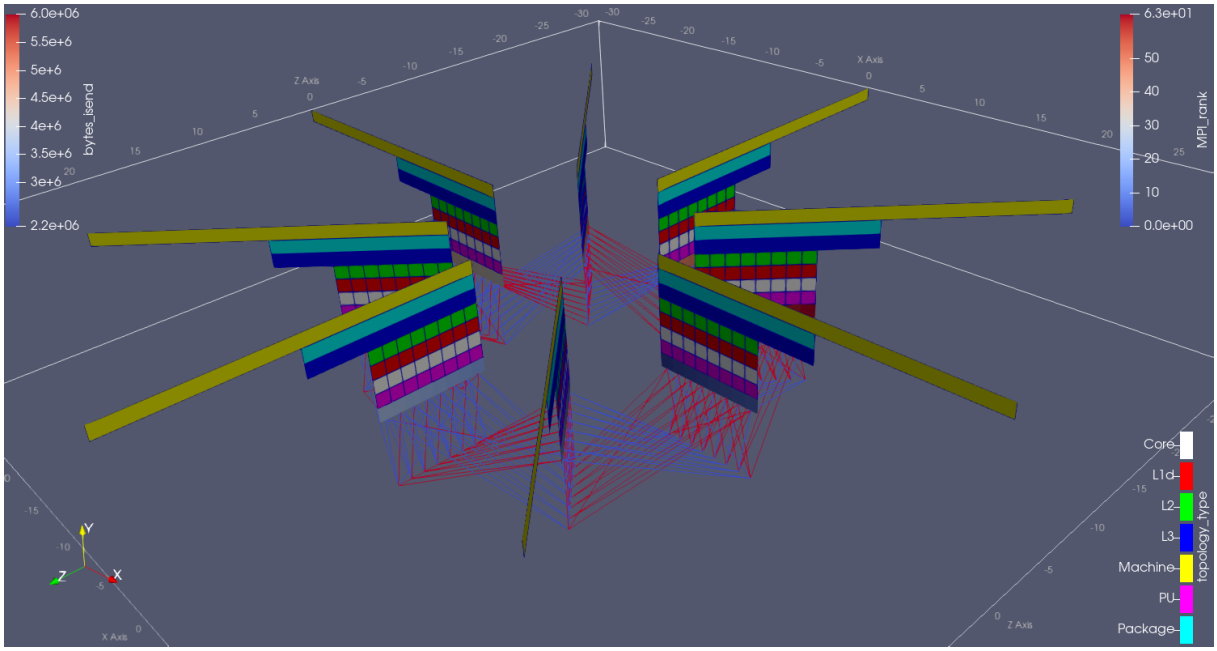
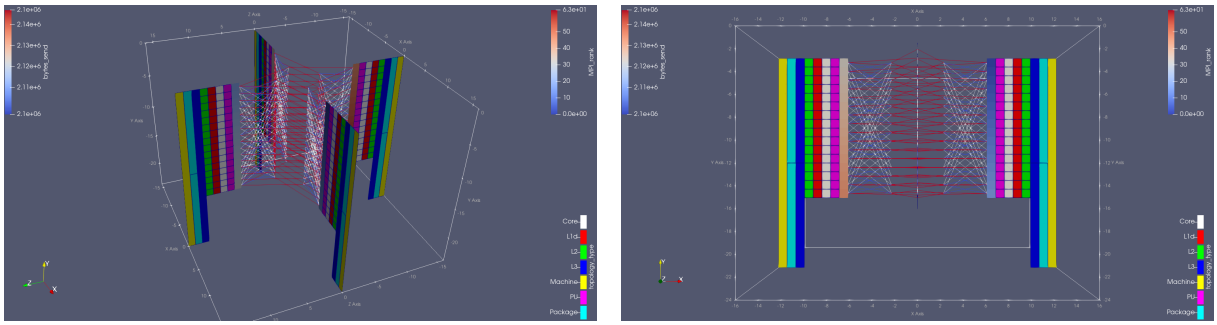


Figure 7.28: Alternative way of displaying topology mode's information in ParaView, for illustration purposes. It shows the results in an arbitrary time step on the BT benchmark. This time the planes are displayed rotated about a center of curvature.

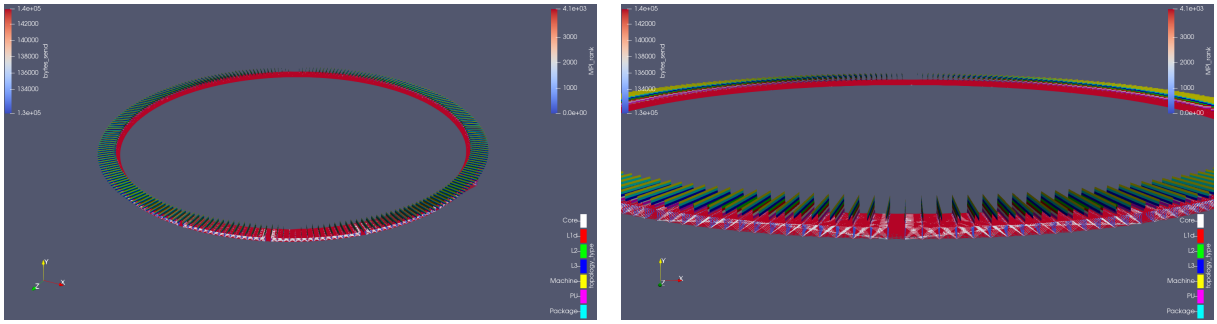


(a) Perspective view (b) Front view

Figure 7.29: Comparative displays of multiply rotated planes in an arbitrary time step of the MG benchmark. Notice how the symmetric nature of the communications becomes visible (especially in the front view): same intra node pattern inside each node.

Notice how the symmetric nature of the communications becomes visible (especially in the front view): same intra node pattern inside each node.

Finally, the capacity of ParaView to rotate the planes should scale as much as its capacity to produce them. For instance, consider Figure 7.30: it shows the results for topology mode in an arbitrary time step on the MG benchmark, here running on 256 compute nodes, each with 16 cores (i.e. 4096 in total). Notice how there seems to be a periodicity in the communications every 16 planes: the white lines in the pictures (see Figure 7.30b) reveal that, at each group of consecutive 16 nodes, each of them talks to the previous and the following ones (including the very first and last ones of the group, which talk to one another). Notice also how no lines cross the interior of the circumference created by the planes: this is a strong indication of a rectilinear grid (with periodic boundaries) in the simulation domain.



(a) Perspective view (b) Zoomed view

Figure 7.30: Comparative displays of rotated planes in an arbitrary time step of the MG benchmark, running on 256 compute nodes, each with 16 cores (i.e. 4096 in total).

7.5 I/O Tracking (Experimental Feature)

The plugin presented in this thesis can be easily extended to account for new performance aspects as Score-P – and its correspondent Substrate Plugin API – is also expanded. This Section will show an example of that. When using a development branch of Score-P, it is possible to apply its compile time flag `--io=runtime:posix`[8] and track the I/O made during the execution of the code. This information will be then available for visualization in Cube or Vampir after the simulation is finished. But the plugin has been extended accordingly and can now also be used to show the I/O made during the time step loop in ParaView. For instance, Figure 7.31 shows the amount of writing events in an arbitrary time step (in which data is written to disk) in Hydra's test case, shown on the simulation geometry (geometry mode) on the left, whereas on the hardware topology representation (topology mode) on the right.

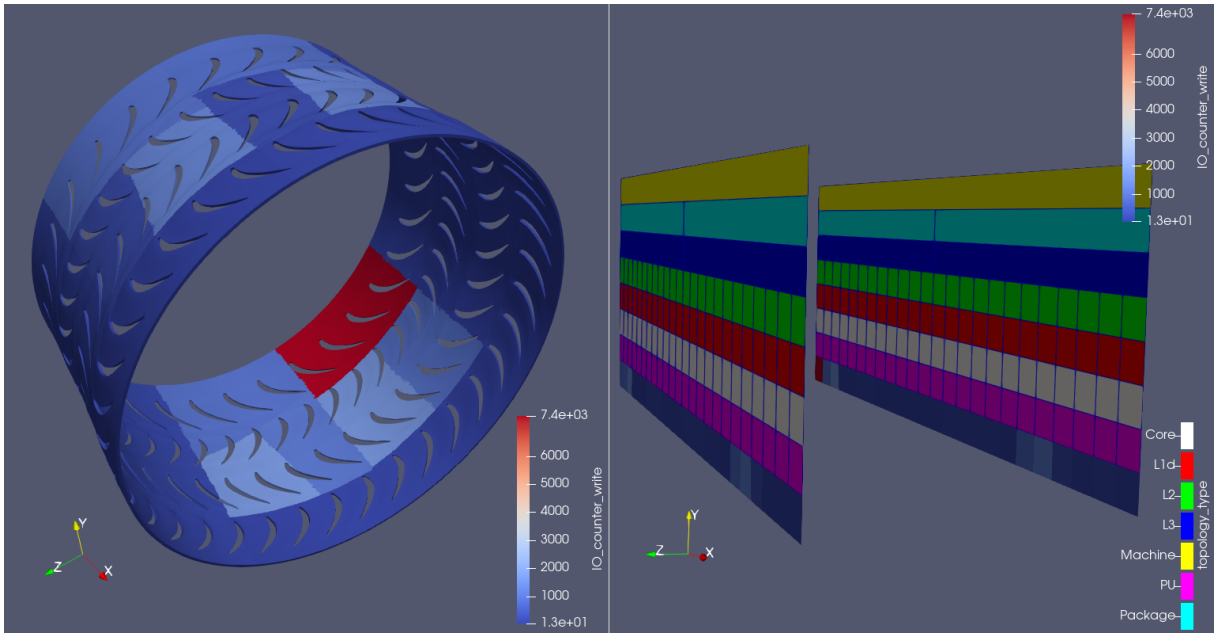


Figure 7.31: Amount of writing events in an arbitrary time step (in which data is written to disk) in Hydra's test case, shown on the simulation geometry (geometry mode) on the left, whereas on the hardware topology representation (topology mode) on the right.

[8] It may be necessary to add the flag `--thread=pthread` as well.

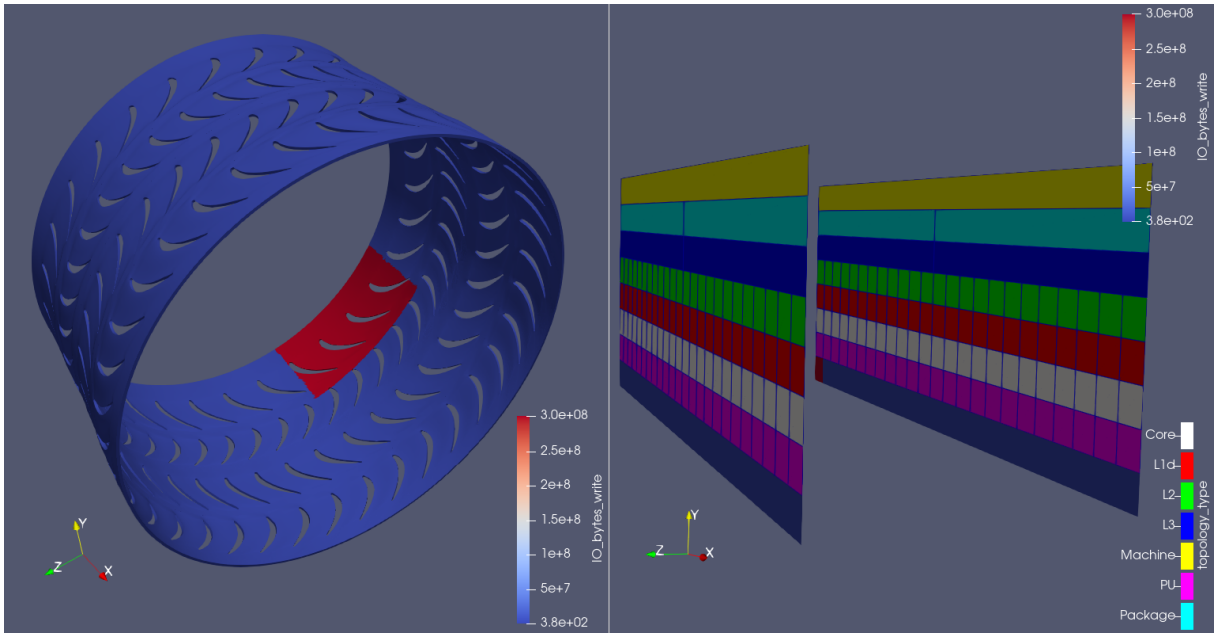


Figure 7.32: Amount of bytes written in an arbitrary time step (in which data is written to disk) in Hydra's test case, shown on the simulation geometry (geometry mode) on the left, whereas on the hardware topology representation (topology mode) on the right.

The new performance aspect is plotted on the bottom line of each plane in topology mode (the same place where the regions measurements are shown). Notice how rank 0[9] concentrates most of the data writing, which is undesirable: the other processes will remain waiting in the meantime, which in turn represents waisted computation time. A better distribution of the I/O load is advised.

The I/O load imbalance becomes even clearer when the size of the data is plotted, which the plugin can also do. Figure 7.32 displays the amount of bytes written in an arbitrary time step (in which data is written to disk) in Hydra's test case, shown again on the simulation geometry (geometry mode) on the left, whereas on the hardware topology representation (topology mode) on the right. Notice how rank 0 has an enormous writing load – in the order of tenths of gigabytes, whereas the other processes do not even reach 1 kB.

This led Rolls-Royce's engineers to introduce some changes into their code, with the aim of improving its I/O performance. A comparison of the before (left-side) and after (right-side) such changes is shown on Figures 7.33 and 7.34. Notice how the scales on the top-right corners oscillate less on the right pictures when compared to their peers on the left pictures. The plugin is therefore able to even quantify the performance improvement between different I/O approaches.

Finally, this new capability of the plugin has also been tested on the I/O version of the BT benchmark (subtype 'full'). The result is shown on Figure 7.35: it displays the amount of writing events (shown on the left) and of bytes written (shown on the right) in an arbitrary time step (in which data is written to disk) when running the code in 4 nodes, each with 16 cores (64 total) – the configuration used previously. Notice how the first rank (the left-most square in the bottom row) on each node has to handle a bigger

[9] It is possible to immediately identify that it is rank 0, and not other one, by the position of the red square in the bottom line of the planes in the topology mode visualization.

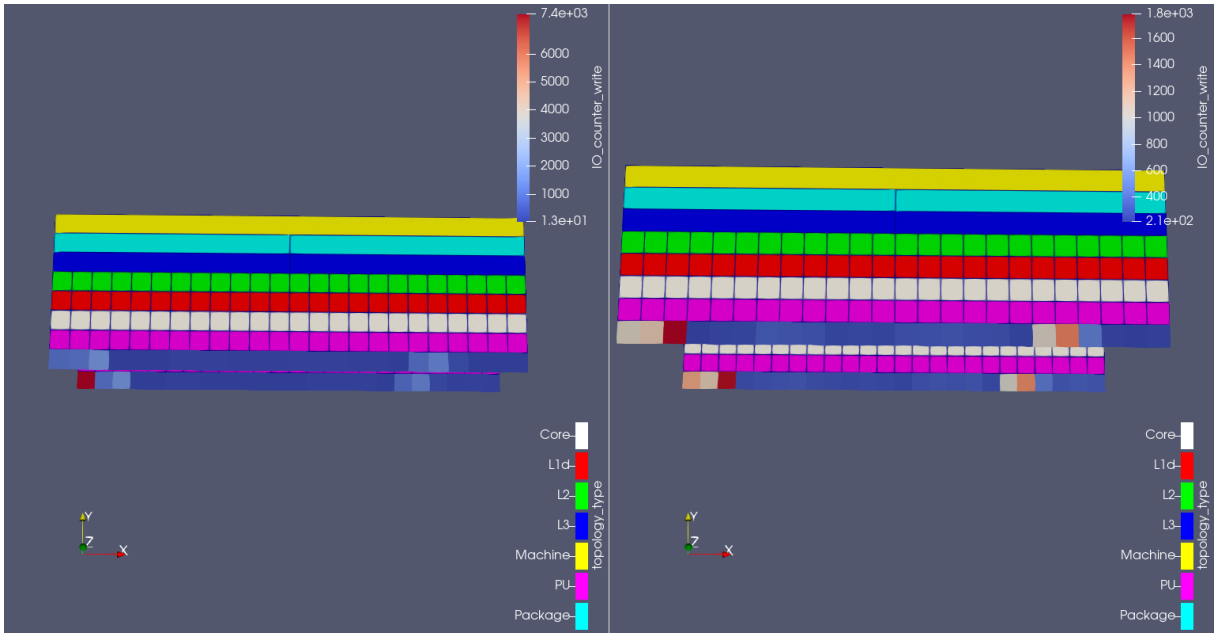


Figure 7.33: Comparison of the amount of writing events in an arbitrary time step (in which data is written to disk) in Hydra's test case, showing before code optimizations on the left, whereas after them on the right.

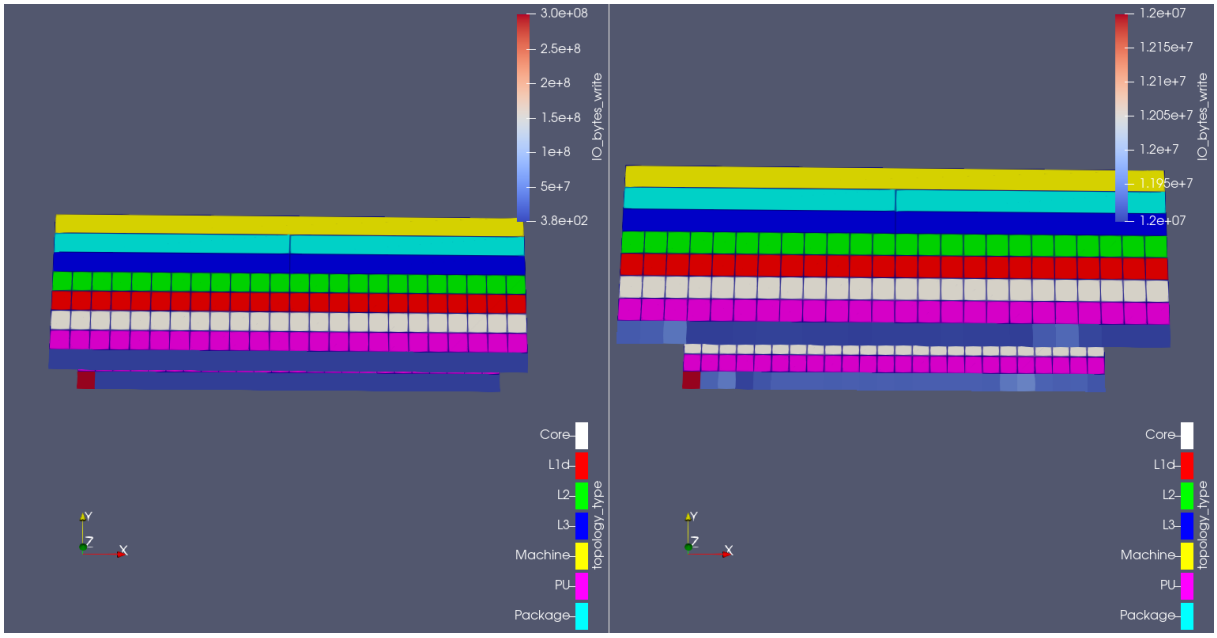


Figure 7.34: Comparison of the amount of bytes written in an arbitrary time step (in which data is written to disk) in Hydra's test case, showing before code optimizations on the left, whereas after them on the right.

load when compared to the others. This might have been a coincidence: every 16^{th} process, starting from the first one, gets overloaded. In order to solve such doubt, another simulation was conducted, this time with 8 nodes, each with 8 cores (same total amount): the result is shown on Figure 7.36 and confirms the observation.

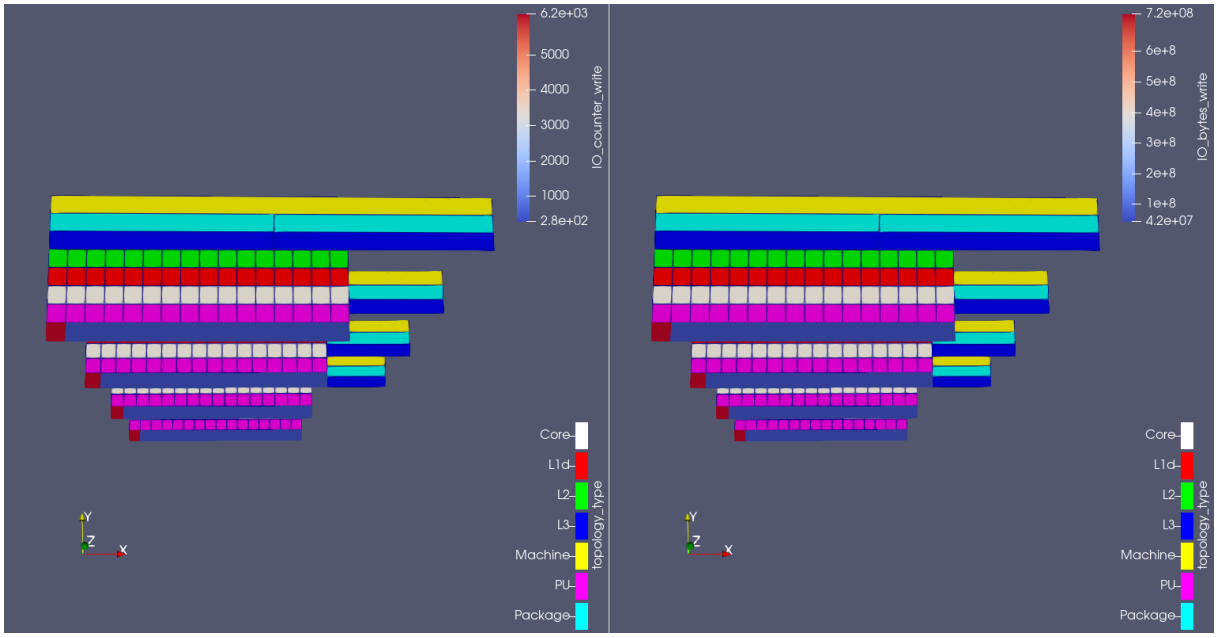


Figure 7.35: Amount of writing events (shown on the left) and of bytes written (shown on the right) in an arbitrary time step (in which data is written to disk) in the I/O version of the BT benchmark, when running it in 4 nodes, each with 16 cores (64 total).

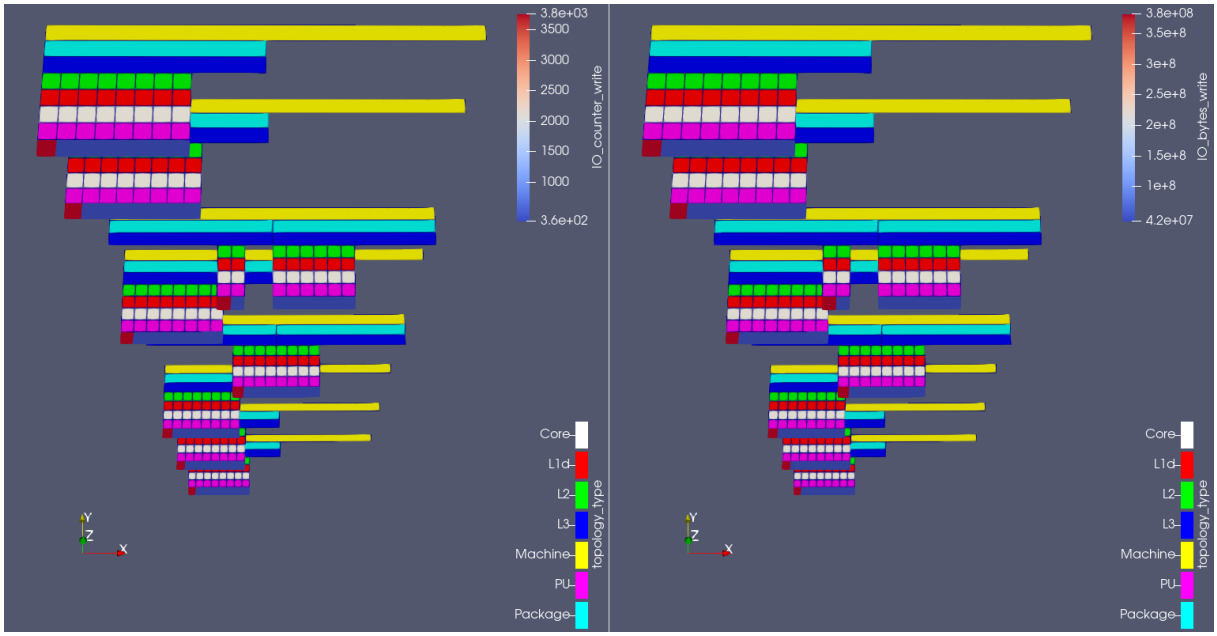


Figure 7.36: Amount of writing events (shown on the left) and of bytes written (shown on the right) in an arbitrary time step (in which data is written to disk) in the I/O version of the BT benchmark, when running it in 8 nodes, each with 8 cores (64 total).

All the information shown above – amount of events and amount of data – can also be generated for the reading side of I/O; but given most of the I/O made during the time step loop is of writing nature, the figures and analyses in this Section have been limited to this aspect. This new feature of the plugin should be fully available when version 7.0 of Score-P is released.

8 Conclusions

8.1 Summary

This thesis presents a concept that allows mapping code performance data back to the simulation's geometry for any type of grid (using the same grid of the simulation itself) and evaluates its implementation regarding the novel visualizations made possible therein with 4 evaluation cases (two benchmarks and two industry-grade CFD codes). This is made possible 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 selected code regions – first feature of the tool, introduced in the author's paper [5] at Euro-Par 2019;
- communication data (messages exchanged between MPI ranks) – second feature of the tool, introduced in the author's paper [6] at Journal *Supercomputing Frontiers and Innovations* 2020.

Furthermore, the tool is also capable of displaying the performance data on top of a symbolic representation of the computational resources being used by the simulation (second mode of the tool, introduced in the author's paper [7] at Journal *Peer Journal of Computer Science* 2021). Three-dimensional representations of clusters are not new; however, the novel approach presented by this thesis brings features which the previous ones would be unable to provide, e.g.:

- detailed view up to topology component level (i.e. in which core of which socket of which node a specific MPI rank is running);
- native association with the simulation's time-step (and not with a logical time, as in Figure 4.6); this is an aspect by which our approach is application-developer friendly: the time-step is the code execution delimiter the developers of CFD codes are naturally used to deal with;
- 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 (see Figure 5.5);
- individually applicable color scale to each element of the visualization, allowing, for example, to color the communication lines by amount of bytes sent;

And finally, all that based exclusively on open-source dependencies (*Score-P*, ParaView, hwloc, Slurm) and under the graphic quality of today's state-of-the-art visualization program, ParaView: render views are fully manipulatable and tens of filters are available to further dig into the data (like the *calculator* filter, used to rotate the visualizations in Sec. 7.4 above).

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 performance tools with their own GUIs. On the other hand, by working in close contact with Rolls-Royce's and DLR's engineers, it has been noticed how important it is for them to obtain the information they need (in our case, the performance of their code) in a straightforward manner. In this sense, using pre-existing visualization tools (like ParaView, which they already use to analyse the flow solution) represents a major benefit to them, as they don't need to learn new software (like Cube or Vampir) for the task, but rather stick with programs they are already used to.[1]

In this threshold, the developed plugin makes load imbalances (be them from a *grid partitioning* or a *computing architecture* origin) and communication inefficiencies easier to identify. The tool's source code has been released open-source and is freely available[2], as the raw data of the benchmark results presented in this thesis[3]. It is ready to be downloaded and installed by anyone willing to try it. 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's input file), thus the tool's overhead can be controlled to a certain extent.

To the best of the author's knowledge, matching performance data to the same grid of the simulation itself for any type of mesh is novel.

8.2 Future Work

There are multiple ways to continue the work compiled by this thesis:

More extensive evaluation cases.

To run the plugin in bigger test cases, as the difficulty in matching each parallel region's rank 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 domains 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, e.g.:

[1] In the end, the goal is to facilitate the work between three different roles: a) the developer of parallel (CFD) simulation codes, b) the performance analyst and c) the code user running simulations. They can better communicate information to each other with the appropriate visualizations and by using (as much as possible) the same software tools.

[2] https://gitlab.hrz.tu-chemnitz.de/alves–tu-dresden.de/catalyst-score-p-plugin.

[3] https://dx.doi.org/10.25532/OPARA-119. We are unable to provide the raw data related to Rolls-Royce's and DLR's codes due to copyright issues.

- chemical reactions in the combustion chamber;
- shock waves in the inlet / outlet, when at the supersonic flow regime;
- air dissociation in the free-stream / inlet, when at the hypersonic flow regime etc.

Develop new visualization schemes for performance data.

To take advantage of the multiple 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. Albeit not being a scientific goal, such functionalities might produce valuable best practices for productive work with the developed software.

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*[4] [29] – 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). Again not a properly scientific target, but important for convenience reasons.

Add (GPU) acceleration.

To make the tool capable of working with (GPU) hardware acceleration and explore the new performance data, and therefore visualizations, that can come out of that.

Extend list of supported communication calls.

To make the tool capable of detecting calls of other communication protocols, like *GPI-2*[5] [31]. 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 etc. They are other aspects to be considered when analysing the performance of a parallel application, as discussed in Sec. 4.1. This expansion 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.

[4] An open-source "software package [that] provides network topology discovery tools, and an abstract representation of those networks topologies for a range of network types and configurations". [tool's website]

[5] The open-source implementation of the *GASPI* (Global Address Space Programming Interface) standard – an open-source alternative to MPI which approaches *PGAS* (Partitioned Global Address Space) programming models.

Extending the lists of supported communication calls and detectable performance phenomena are trully scientific goals which should be attained as Score-P itself evolves. Especially RDMA and one-sided synchronization, important aspects for better performance and scalability, should require dedicated visualizations for proper understanding. These will be made possible once such features are fully supported in Score-P and the plugin is updated accordingly.

A The Plugin's Test-Case

In order to assist the user of the plugin in integrating it to its simulation's code, the repository at https://gitlab.hrz.tu-chemnitz.de/alves–tu-dresden.de/catalyst-score-p-plugin/ is equipped with a sample test-case. It solves the heat diffusion problem within a three-dimensional, parallelepipedal space, as shown below:

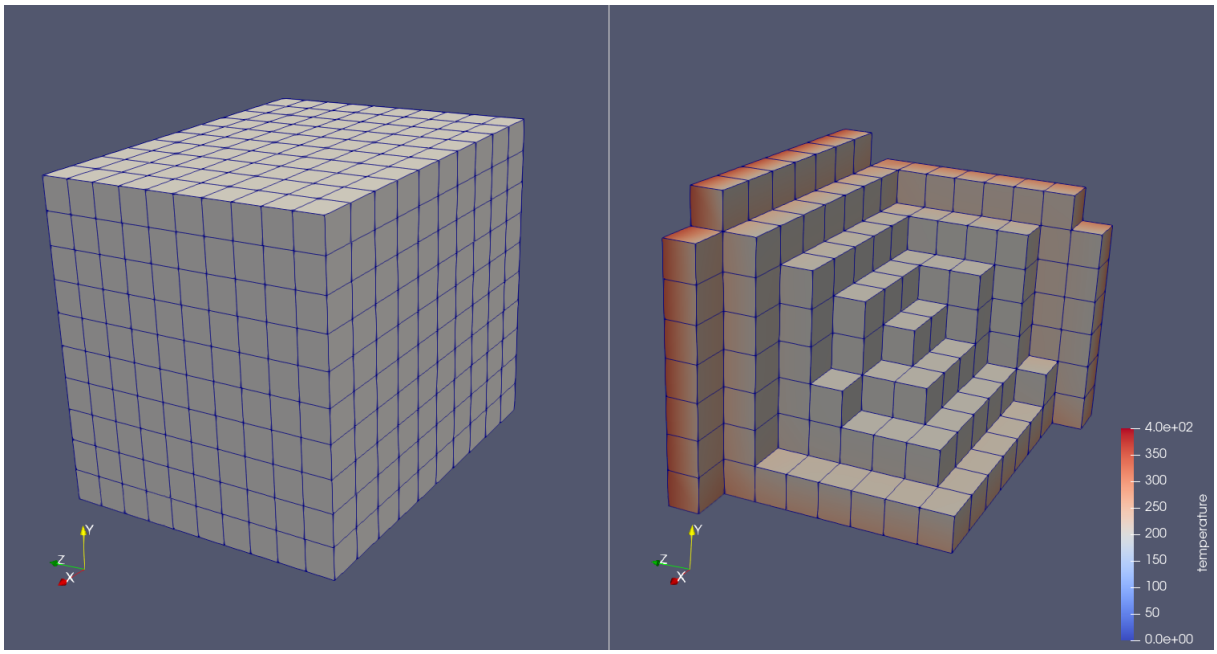


Figure A.1: Grid of the plugin's test-case shown on the left. It solves the heat diffusion problem within a three-dimensional, parallelepipedal space. A threshold of temperature values between an arbitrary range after an arbitrary number of time-steps is shown on the right. The goal of this sample test-case is to serve as reference for the plugin's user of how to integrate the tool with its own simulation.

The Heat Equation is discretized in *time* through a first-order forward difference, whereas in *space* through a second-order, central, second difference. Each wall of the parallelepiped is assigned a fixed temperature (which is nothing more than a *fixed value boundary condition*, as known in CFD simulations), which can be conveniently adjusted at run time through the test-case's input file, shown below:

```
print_log = 1
multiple_minds = 3
base_minds_dim = 1 1 1
multiple_points = 9
base_points_dim = 1 1 1
max_num_time_steps = 10
dt = 0.5
```

```
check_convergence = 0
precision_T = 1.0
thermal_diffusivity = 0.000864
internal_T = 200.0
d_dim = 0.12 0.1 0.09
boundary_conditions = 300.0 100.0 400.0 50.0 150.0 350.0
```

The `boundary_conditions` expect temperature values for respectively the *left* ($-x$), *right* ($+x$), *bottom* ($-y$), *top* ($+y$), *back* ($-z$) and *front* ($+z$) walls. `internal_T` controls, on its turn, the initial temperature to be applied to the simulation's domain (known as *internal field* in CFD). The `thermal_diffusivity`, here assumed constant, is a material property of the medium and defines how fast heat is transfered through it. `dt` controls the temporal length of a time-step and `max_num_time_steps` defines how many time-steps will be simulated, unless `check_convergence` is set to `1` and the simulation manages to converge the results up to the precision specified at `precision_T` before that number of time-steps is reached.

`d_dim` specifies the distance between each grid point in each of the three cartesian directions (x, y and z respectively). `multiple_points`, on its turn, defines the amount of grid points in each of those same directions. Such amount will be divided by the number of MPI ranks in each of those directions, set by the parameter `multiple_minds`; the remainder of such division needs to be zero (i.e. `multiple_points` needs to be divisable by `multiple_minds`). In the example shown above, each process will receive three points in each cartesian direction to work with; and the grid will be composed of three subdomains in each cartesian direction (i.e. $3^3 = 27$ ranks in total). Finally, `print_log` turns the generation of output logs on or off.

With those input parameters set, the test-case is then able to produce all outputs of the plugin. Using the above configuration as an example, Figure A.2 shows the partitioning of the grid on the left; and the time taken by each rank to compute the main solver loop's calculation, plotted on top of the simulation's geometry, on the right. Figure A.3, on its turn, shows the location of an arbitrary subdomain (left) and those communicating with it (right), colored by amount of messages sent (in this case, MPI_Put calls) in an arbitrary time-step during the simulation. These two pictures correspond to the *geometry mode* of the plugin. The last picture, Figure A.4, displays the *topology mode*'s results of the test-case, showing the location of each core used by the simulation (within the node's architecture), the correspondent MPI rank running there, the name of the network switch connecting the nodes and the communication channels (colored by the source rank id of the messages).

With the full set of outputs from the plugin, the user has a comprehensive example on how to integrate a simulation's code with our tool. Indeed, the plugin's test-case is even useful to teach how to create a Catalyst adapter for a simulation (using a rectilinear grid).

In order to build the test-case in Taurus, it is enough to execute the script at `test-case/taurus.sh`; it will also recursively execute the script at `plugin/taurus.sh`, which builds the plugin, and run the test-case (with the configuration presented above). Both scripts are configured for the Taurus scheme of modules; they may be used as reference to install the tool in other systems. After the simulation job is

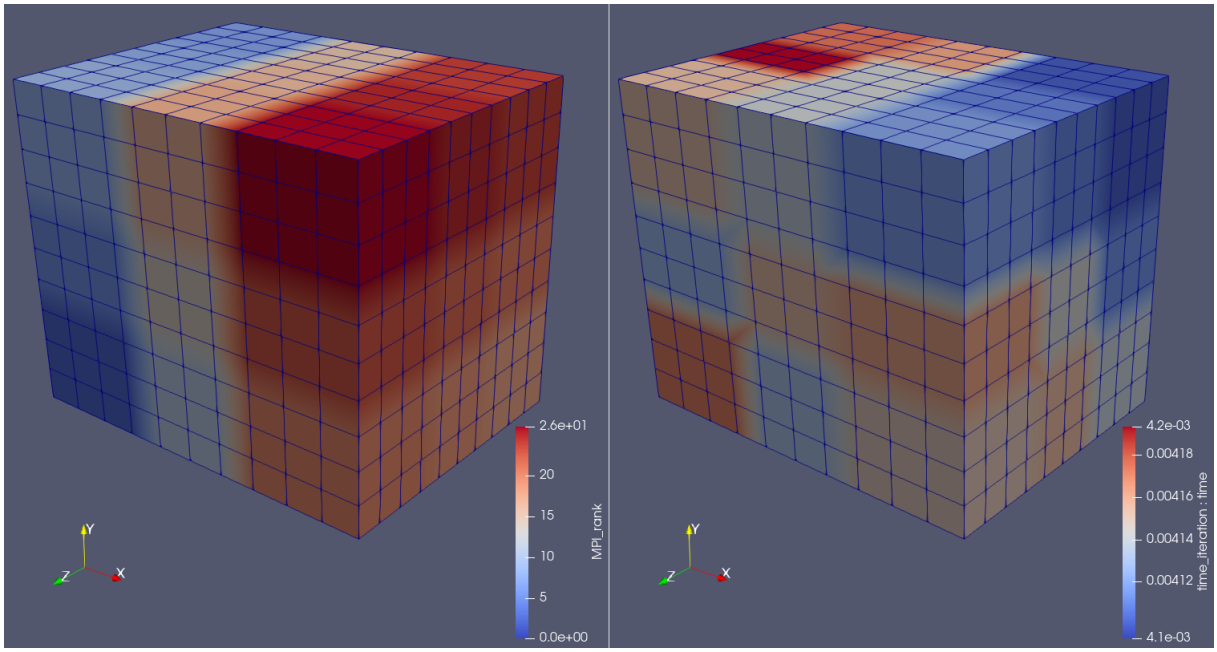


Figure A.2: Example of partitioning of the plugin's test-case's grid shown on the left; the simulation domain is decomposed in three subdomains in each cartesian direction (i.e. $3^3 = 27$ ranks in total). On the right-hand side, on its turn, the time taken by each rank to compute the main solver loop's calculation is shown (on top of the simulation's geometry itself).

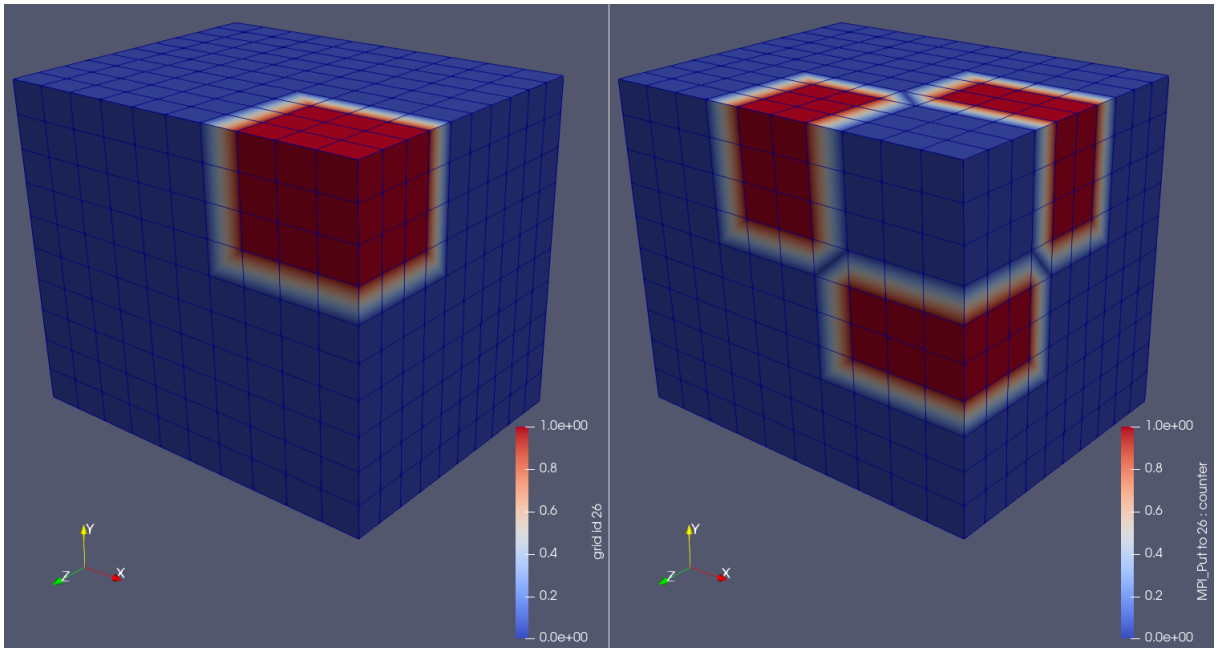


Figure A.3: Location of an arbitrary subdomain (left) and those communicating with it (right), colored by amount of messages sent (in this case, MPI_Put calls) in an arbitrary time-step of the plugin's test-case.

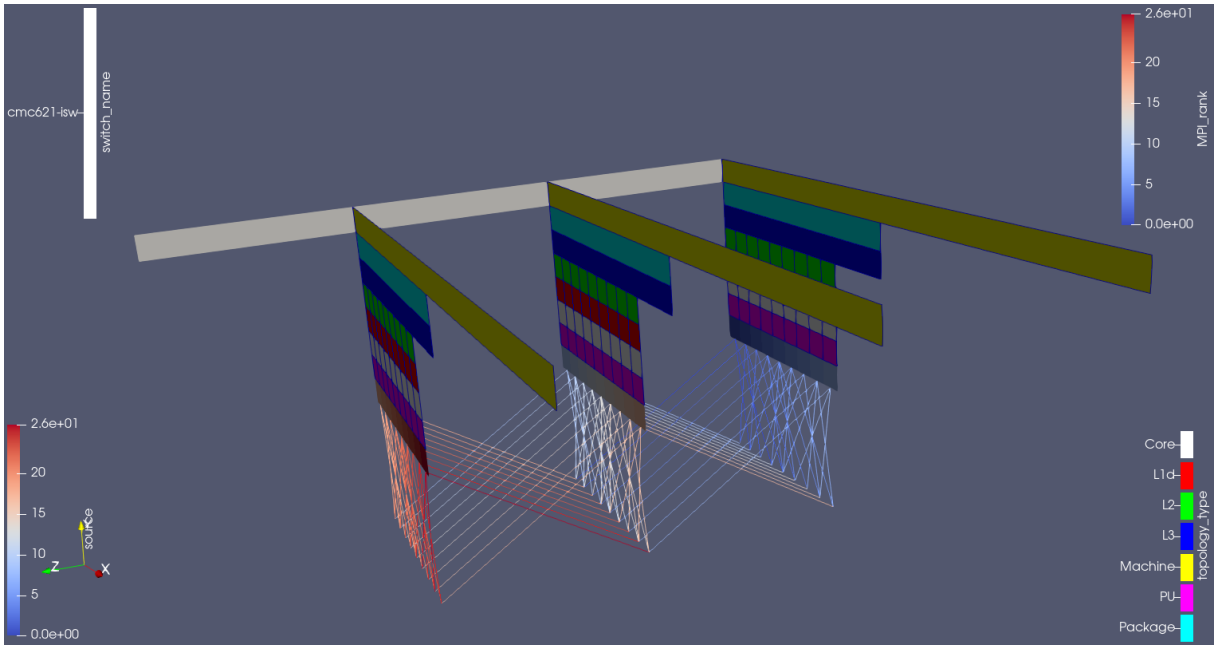


Figure A.4: *Topology mode*'s results of the plugin's test-case, showing the location of each core used by the simulation (within the node's architecture), the correspondent MPI rank running there, the name of the network switch connecting the nodes and the communication channels (colored by the source rank id of the messages).

finished, all outputs can be found at `test-case/install` and can be visualized with version 5.7.0 or later of ParaView. Its precompiled binary distribution is enough to open the test-case's outputs; but in order to build the plugin (with the capacity to run in topology mode), it is necessary to build ParaView from source. For that purpose, a personal installation in Taurus was used during this thesis; the script to generate it (with all the installation options already set) is provided at `paraview.sh`. The same applies to the Score-P installation (another dependency of the plugin): it can be straightforwardly reproduced by means of the script `scorep.sh`.

B Manual code instrumentation with Score-P

The code snippet below illustrates how to perform a manual code instrumentation using Score-P:

```
#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 B.1: 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.

Bibliography

[1] *Netloc 2.1.0 Documentation*.

[2] *Score-P 6.0 Documentation*.

[3] *Vampir 9.11 Documentation*.

[4] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. *The visualization handbook*, 717, 2005.

[5] Rigel F. C. Alves and Andreas Knüpfer. In Situ Visualization of Performance-Related Data in Parallel CFD Applications. In Ulrich Schwardmann, Christian Boehme, Dora B. Heras, Valeria Cardellini, Emmanuel Jeannot, Antonio Salis, Claudio Schifanella, Ravi Reddy Manumachu, Dieter Schwamborn, Laura Ricci, Oh Sangyoon, Thomas Gruber, Laura Antonelli, and Stephen L. Scott, editors, *Euro-Par 2019: Parallel Processing Workshops*, pages 400–412, Cham, 2020. Springer International Publishing.

[6] Rigel F. C. Alves and Andreas Knüpfer. Enhancing the in Situ Visualization of Performance Data in Parallel CFD Applications. *Supercomputing Frontiers and Innovations*, 7(4), 2020.

[7] Rigel F. C. Alves and Andreas Knüpfer. Further enhancing the in situ visualization of performance data in parallel CFD applications. *Peer Journal of Computer Science*, 7(e753), 2021.

[8] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[9] John David Anderson. *Computational Fluid Dynamics: The Basics with Applications*. McGraw-Hill, 1995.

[10] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. 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, 2015. ACM.

[11] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994.

[12] Jim Banke. Seeking reality in the future of aeronautical simulation. https://www.nasa.gov/aero/aeronautical-simulation.html, 2017.

[13] Radovan Bast and Roberto Di Remigio. *CMake Cookbook: Building, testing, and packaging modular software with modern CMake*. Packt Publishing Ltd, 2018.

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

[15] Motti Beck. Virtualization acceleration. VMworld 2014 in San Francisco, CA, USA. https://www.slideshare.net/mellanox/motti-virtualization-accelration-over-ro-ce, 2014.

[16] Robert Bell, Allen D. Malony, and Sameer Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, pages 17–26, Berlin, Heidelberg, 2003. Springer.

[17] Janine C. Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, Philippe Pebay, David Thompson, Hongfeng Yu, Fan Zhang, and Jacqueline Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–9, 2012.

[18] Maryam Besharati-Givi. *Numerical Investigation of Flat Plate Flow with a Jet Actuator*. PhD thesis, Technical University of Berlin, Mechanical Engineering Department, Aug 2009.

[19] A Bobet, A Fakhimi, Scott Johnson, Joseph Morris, Fulvio Tonon, and M. Yeung. Numerical models in discontinuous media: Review of advances for rock mechanics applications. *Journal of Geotechnical and Geoenvironmental Engineering*, 135, Nov 2009.

[20] Peer-Timo Bremer, Bernd Mohr, Valerio Pascucci, Martin Schulz, Todd Gamblin, and Holger Brunst. Connecting Performance Analysis and Visualization. *Dagstuhl Manifestos*, 5(1):1–24, 2015.

[21] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186, Pisa, Italy, Feb 2010. IEE.

[22] Remi Busseuil, Gabriel Marchesan Almeida, Luciano Ost, Sameer Varyani, Gilles Sassatelli, and Michel Robert. Adaptation strategies in multiprocessors system on chip. In José L. Ayala, David Atienza Alonso, and Ricardo Reis, editors, *VLSI-SoC: Forward-Looking Trends in IC and Systems Design*, pages 233–257, Berlin, Heidelberg, 2012. Springer.

[23] Hank Childs. VisIt: An end-user tool for visualizing and analyzing very large data. 2012.

[24] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[25] Robert Dietrich. *Scalable Applications on Heterogeneous System Architectures: A Systematic Performance Analysis Framework*. PhD thesis, Technische Universität Dresden, 2019.

[26] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth Jansen. The ParaView coprocessing library: A scalable, general purpose in situ visualization library. *1st IEEE Symposium on Large-Scale Data Analysis and Visualization 2011, LDAV 2011 - Proceedings*, Oct 2011.

[27] Michael Frumkin, Haoqiang Jin, and Jerry Yan. Implementation of NAS parallel benchmarks in high performance fortran. *NAS Technical Report NAS-98-009*, 1998.

[28] Alfredo Giménez, Todd Gamblin, Ilir Jusufi, Abhinav Bhatele, Martin Schulz, Peer-Timo Bremer, and Bernd Hamann. MemAxes: Visualization and analytics for characterizing complex memory performance behaviors. *IEEE Transactions on Visualization and Computer Graphics*, 24(7):2180–2193, 2018.

[29] B. Goglin, J. Hursey, and J. M. Squyres. 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, Sep. 2014. IEEE.

[30] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[31] Daniel Grünewald and Christian Simmendinger. The GASPI API specification and its implementation GPI 2.0. In N. Johnson M. Weiland, A. Jackson, editor, *Proceedings of the 7th International Conference on PGAS Programming Models*, pages 243–248, Scotland, UK, 2013. The University of Edinburgh.

[32] John L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, May 1988.

[33] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, Steve Gunther, Tom Piazza, and Ted Burton. Haswell: The fourth-generation Intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[34] Chao-Tsung Hsiao, Jingsen Ma, and Georges Chahine. Multiscale tow-phase flow modeling of sheet and cloud cavitation. *International Journal of Multiphase Flow*, 90, Jan 2017.

[35] K. A. Huck, K. Potter, D. W. Jacobsen, H. Childs, and A. D. Malony. Linking performance data into scientific visualization tools. In *2014 First Workshop on Visual Performance Analysis*, pages 50–57. IEEE, Nov 2014.

[36] Immo Huismann, Stefan Fechter, and Tobias Leicht. HyperCODA – extension of flow solver CODA to hypersonic flows. In Andreas Dillmann, Gerd Heller, and Claus Wagner, editors, *STAB-Symposium 2020*, volume XIII of *New Results in Numerical and Experimental Fluid Mechanics*. Springer International Publishing, 2020.

[37] Benafsh Husain, Alfredo Giménez, Joshua A. Levine, Todd Gamblin, and Peer-Timo Bremer. Relating memory performance data to application domain data using an integration API. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*, VPA '15, New York, NY, USA, 2015. Association for Computing Machinery.

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

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

[40] Katherine E. Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. State of the Art of Performance Visualization. In R. Borgo, R. Maciejewski, and I. Viola, editors, *EuroVis - STARs*. The Eurographics Association, 2014.

[41] Guido Juckeland. *Trace-based Performance Analysis for Hardware Accelerators*. PhD thesis, Technische Universität Dresden, 2012.

[42] B. Khanal, L. He, J. Northall, and P. Adami. Analysis of Radial Migration of Hot-Streak in Swirling Flow Through High-Pressure Turbine Stage. *Journal of Turbomachinery*, 135(4), 06 2013.

[43] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir performance analysis tool-set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[44] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A joint performance measurement runtime infrastructure for Periscope, Scalasca, TAU, and Vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[45] Olaf Krzikalla. *Neue Ansätze zur Speicherzugriffsanalyse paralleler Anwendungen mit gemeinsam genutztem Adressraum*. PhD thesis, Technische Universität Dresden, 2018.

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

[47] Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ISAV'17, page 42–46, New York, NY, USA, 2017. Association for Computing Machinery.

[48] T. Leicht, D. Vollmer, J. Jägersküpper, A. Schwöppe, R. Hartmann, J. Fiedler, and T. Schlauch. DLR-Project Digital-X next generation CFD solver 'Flucs'. In DLR, editor, *Deutscher Luft- und Raumfahrtkongress*, 2016.

[49] Xin Li. Scalability: strong and weak scaling. https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/, 2018.

[50] Kwan-Liu Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Applications*, 29:14–19, 2009.

[51] Michael Meinel and Gunnar O. Einarsson. The FlowSimulator framework for massively parallel CFD applications. In *PARA2010*, June 2010.

[52] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Jourdain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, Michael E. Papka, and Scott Klasky. Examples of in transit visualization. In *Proceedings of the 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities*, PDAC '11, page 1–6, New York, NY, USA, 2011. Association for Computing Machinery.

[53] Gregory F. Pfister. An introduction to the InfiniBand architecture. *High performance mass storage and parallel I/O*, 42(617-632):10, 2001.

[54] Matt Pharr and Randima Fernando. *GPU gems 2: programming techniques for high-performance graphics and general-purpose computation*, volume 1. Addison-Wesley Reading, 2005.

[55] Pavel Saviankou, Michael Knobloch, Anke Visser, and Bernd Mohr. Cube v4: From performance report explorer to performance analysis tool. *Procedia Computer Science*, 51:1343 – 1352, 2015. International Conference on Computational Science, ICCS 2015.

[56] Lucas Mello Schnorr, Guillaume Huard, and Philippe O.A. Navaux. Triva: Interactive 3D visualization for performance analysis of parallel applications. *Future Generation Computer Systems*, 26(3):348 – 358, 2010.

[57] Robert Schöne, Ronny Tschüter, Thomas Ilsche, Joseph Schuchart, Daniel Hackenberg, and Wolfgang E. Nagel. Extending the functionality of Score-P through plugins: Interfaces and use cases. In Christoph Niethammer, José Gracia, Tobias Hilbrich, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2016*, Cham, 2017. Springer International Publishing.

[58] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *Proceedings of Seventh Annual IEEE Visualization '96*, pages 93–100, San Francisco, CA, USA, Oct 1996. IEEE.

[59] Martin Schulz, Abhinav Bhatele, David Böhme, Peer-Timo Bremer, Todd Gamblin, Alfredo Gimenez, and Kate Isaacs. A flexible data model to support multi-domain performance analysis. In Christoph Niethammer, José Gracia, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2014*, pages 211–229, Cham, 2015. Springer International Publishing.

[60] Sameer S. Shende and Allen D. Malony. The Tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[61] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The craft of information visualization*, pages 364–371. Elsevier, 2003.

[62] Virtual Institute High Productivity Supercomputing. Tools guide. Technical report, 2020.

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

[64] Ronny Tschüter. *Holistic Performance Analysis of Multi-layer I/O in Parallel Scientifc Applications*. PhD thesis, Technische Universität Dresden, 2021.

[65] John Vassberg, Mori Mani, Ben Rider, Ed Tinoco, Kelly Laflin, David Levy, Rich Wahls, Joe Morrison, Dimitri Mavriplis, Olaf Brodersen, Simone Crippa, and Mitsuhiro Murayama. 5th AIAA CFD Drag Prediction Workshop. 2012.

[66] John C. Vassberg, Mark A. DeHaan, S. Melissa Rivers, and Richard A. Wahls. Development of a common research model for applied CFD validation studies. Technical report, AIAA, 2008.

[67] Tom Vierjahn, Torsten W. Kuhlen, Matthias S. Müller, and Bernd Hentschel. Visualizing performance data with respect to the simulated geometry. In Edoardo Di Napoli, Marc-André Hermanns, Hristo Iliev, Andreas Lintermann, and Alexander Peyser, editors, *High-Performance Scientific Computing*, pages 212–222, Cham, 2017. Springer International Publishing.

[68] Michael Wagner, Jens Jägersküpper, Daniel Molka, and Thomas Gerhold. Performance analysis of complex engineering frameworks. *Tools for High Performance Computing*, 2019.

[69] Frank M. White. *Fluid Mechanics*. McGraw-Hill, 7th edition, 2009.

[70] Chad Wood. SOSflow: A scalable observation system for introspection and in situ analytics. In *ICPP*, Aug 2018.

[71] Chad Wood, Matthew Larsen, Alfredo Gimenez, Kevin Huck, Cyrus Harrison, Todd Gamblin, and Allen Malony. Projecting performance data over simulation geometry using SOSflow and ALPINE. In Abhinav Bhatele, David Boehme, Joshua A. Levine, Allen D. Malony, and Martin Schulz, editors, *Programming and Performance Visualization Tools*, pages 201–218, Cham, 2019. Springer International Publishing.

[72] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer.

[73] Jun Zhang. Parallel Computing: Performance and Scalability. https://www.cs.uky.edu/ jzhang/CS621/chapter7.pdf, 2019.

List of Figures

List of Tables

Acknowledgments

The following people and institutions, which the author would like to warmly thank, have been fundamental in making this work possible:

Rolls-Royce Germany, especially Marcus Meyer, Axel Gerstenberger, Jan Suhrmann & Paolo Adami, for providing one of the industrial CFD codes and its test case for this thesis, what has been done in the context of BMWi research project *Prestige* (FKZ 20T1716A).

The German Aerospace Center (DLR) (also part of Prestige), especially Thomas Gerhold (from the Institute of Software Methods for Product Virtualization), for providing the other industrial CFD code for this thesis; Immo Huismann (from the same institute), for providing its correspondent test case and thoroughly supporting me throughout its implementation; and Tobias Leicht (also from the same institute), for his inputs and final approval of the work from DLR's point of view.

Kitware (also part of Prestige), especially Mathieu Westphal & Nicolas Vuaille, for the support on implementing the Catalyst adapter.

The Centre for Information Services and High Performance Computing (ZIH) of the University of Dresden (TUD), for housing me during my PhD and providing all physical resources needed for my work.

The Score-P support and development team, especially Bert Wesarg (from ZIH), for the assistance with the tool overall; and Andreas Gocht (from the same institute), for the help with the Score-P Python bindings.

The Taurus' admins, especially Maik Schmidt (from ZIH), for the help with the supercomputer.

Holger Brunst (from ZIH), for his comments on preliminary versions of this work.

Prof. Stefan Gumhold (from TUD's Chair of Computer Graphics and Visualisation), for accepting to be the *Fachreferent* of my PhD and taking part on its examination board.

Prof. Wolfgang Nagel (Director of ZIH), for accepting to be the *Betreuer* of my PhD and taking part on its examination board.

And Andreas Knüpfer (CTO of ZIH), for his invaluable supervision at all times.