International Journal of Computational Fluid Dynamics

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/gcfd20

# A Generic Performance Analysis Technique Applied to Different CFD Methods for HPC

Marta Garcia-Gasulla, Fabio Banchelli, Kilian Peiro, Guillem Ramirez-Gargallo, Guillaume Houzeaux, Ismaïl Ben Hassan Saïdi, Christian Tenaud, Ivan Spisso & Filippo Mantovani

**To cite this article:** Marta Garcia-Gasulla, Fabio Banchelli, Kilian Peiro, Guillem Ramirez-Gargallo, Guillaume Houzeaux, Ismaïl Ben Hassan Saïdi, Christian Tenaud, Ivan Spisso & Filippo Mantovani (2020) A Generic Performance Analysis Technique Applied to Different CFD Methods for HPC, International Journal of Computational Fluid Dynamics, 34:7-8, 508-528, DOI: 10.1080/10618562.2020.1778168

**To link to this article:** https://doi.org/10.1080/10618562.2020.1778168



Published online: 16 Jul 2020.

INTERNATIONAL JOURNAL OF COMPUTATIONAL FLUID DYNAMICS
2020, VOL. 34, NOS. 7–8, 508–528
https://doi.org/10.1080/10618562.2020.1778168

Taylor & Francis
Taylor & Francis Group



# A Generic Performance Analysis Technique Applied to Different CFD Methods for HPC

Marta Garcia-Gasulla [a], Fabio Banchelli [a], Kilian Peiro [a], Guillem Ramirez-Gargallo[a], Guillaume Houzeaux [a], Ismaïl Ben Hassan Saïdi[b], Christian Tenaud [b], Ivan Spisso[c] and Filippo Mantovani [a]

[a]Barcelona Supercomputing Center, Barcelona, Spain; [b]Université Paris-Saclay, CNRS, LIMSI, Orasy, France; [c]CINECA, Bologna, Italy

**ABSTRACT**
For complex engineering and scientific applications, Computational Fluid Dynamics (CFD) simulations require a huge amount of computational power. As such, it is of paramount importance to carefully assess the performance of CFD codes and to study them in depth for enabling optimisation and portability. In this paper, we study three complex CFD codes, OpenFOAM, Alya and CHORUS representing two numerical methods, namely the finite volume and finite-element methods, on both structured and unstructured meshes. To all codes, we apply a generic performance analysis method based on a set of metrics helping the code developer in spotting the critical points that can potentially limit the scalability of a parallel application. We show the root cause of the performance bottlenecks studying the three applications on the MareNostrum4 supercomputer. We conclude providing hints for improving the performance and the scalability of each application.

**ARTICLE HISTORY**
Received 2 March 2020
Accepted 19 May 2020

**KEYWORDS**
Performance analysis; MPI; efficiency; parallelisation; CFD; finite element; finite volume

## 1. Introduction

Fluid-dynamics is more and more leveraging numerical techniques, that allow to compute solutions for the non-linear equations describing regimes and geometries relevant in engineering and industrial contexts. Several different numerical approaches have been theoretically developed and implemented on several massively parallel computers. In this paper, we consider three different complex Computational Fluid Dynamics (CFD) codes, for the following classes of numerical methods: finite element (Alya) and finite volume (OpenFOAM and CHORUS). Moreover, the three codes address different flow problems, incompressible flows (Alya and OpenFOAM) and compressible flows (CHORUS) that lead to different numerical time integrations, from completely explicit schemes to fully implicit integrations. Also, both unstructured meshes (Alya and OpenFOAM) and structured meshes (CHORUS) are represented.

Those numerical methods heavily rely on High Performance Computing (HPC) resources. Therefore, on the one hand, their efficient implementation is of paramount importance for extracting the computational power offered by new HPC systems. On the other hand, the portability of performance optimisation is also critical, since developers and maintainers of CFD codes cannot re-implement their method for each new HPC technology appearing on the market (Banchelli et al. 2020b; Mantovani et al. in press; Banchelli et al. 2020a).

For this reason, we focus the study of this paper on the performance analysis of the aforementioned CFD applications. We apply a set of general performance metrics defined in the POP performance model (Wagner et al. 2017); the POP metrics are a set of efficiency and scalability indicators that can be obtained for MPI applications. These indicators identify the main factors limiting the scalability of the code, based on these indicators, a more detailed analysis is conducted to understand which is the reason for the degradation of performance. From this analysis, we extract insights for improving the performance and the scalability of the selected codes. We are indeed interested in complex geometries and how they perform at mid-large-scale. The point of view that we would like to provide is the one of a 'regular domain scientist' who needs to run a real scientific case efficiently on an HPC cluster. The final goal of the paper is to highlight computational

**CONTACT** Marta Garcia-Gasulla marta.garcia@bsc.es

bottlenecks coming from the way the applications are programmed or, sometimes, the way HPC systems are configured. The added value of our approach is that none of the optimisations proposed should be system-specific so performance portability is preserved.

The main contributions of this paper are: (i) we apply the same performance analysis method to three CFD applications selected as representatives of the main CFD methods running on nowadays supercomputers; (ii) we study the outcome of performance analysis data gathered on a state-of-the-art HPC cluster and we provide insights for improving the performance and the scalability.

The remaining part of the document is structured as follows. In Section 2, we introduce each of the three computational methods studied in this paper. In Section 3, we describe the performance analysis method used to study the HPC applications. In Section 4, we discuss the performance analysis performed on each application. We conclude with Section 5 where we summarise performance and scalability insights.

## 2. Environment

We selected three CFD codes, employing different spatial numerical methods as well as time integration schemes. Table 1 summarises the different algorithms.

We carried out the performance study running on MareNostrum4, the Spanish national supercomputer hosted at the Barcelona Supercomputing Center. The rest of this section is dedicated to introduce the details of the hardware platform and the applications employed in our study.

### 2.1. Hardware Environment

The MareNostrum4 supercomputer is a Tier-0 supercomputer ranked 30th in the Top500 (November 2019) with a total of 3456 × 86 compute nodes. MareNostrum4 is based on Intel Xeon Platinum processors from the Skylake generation. The nodes of the system are interconnected using Intel Omni-Path high performance network and run SuSE Linux Enterprise Server as operating system. Each node is equipped with:

**CPU** – 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each running at 2.10 GHz for a total of 48 cores per node.

**Cache** – 32 KB of L1 data cache; 32 KB L1 instruction cache; 1024 kB L2 data cache; 33792 kB L3 data cache. L1 and L2 are local to each core while L3 is shared among all 24 cores of a socket.

**Memory** – 96 GB of main memory 1.880 GB/core, 12 × 8 GB 2667 Mhz DIMM (216 nodes high memory, 10368 cores with 7.928 GB/core).

**HPC Network** – 100 Gbit/s Intel Omni-Path HFI Silicon 100 Series PCI-E adapter.

**Service Network** – 10 Gbit Ethernet.

**Local Storage** – 200 GB local SSD available as temporary storage.

**Network Storage** – IBM GPFS filesystem.

Figure 1 shows a schematic view of a node of MareNostrum4.

### 2.2. OpenFOAM: Unstructured Mesh Finite Volume

#### 2.2.1. Context

OpenFOAM (for '**Open**-source **F**ield **O**peration **A**nd **M**anipulation ') is the free, open source CFD software developed primarily by OpenCFD Ltd since 2004 (Chen et al. 2014). Currently, there are three main variants of OpenFOAM software that are released as free and open-source software under the GNU General Public License Version 3: CFD Direct ,[1] the open source CFD toolbox[2] and foam-extend .[3] The development model follows a *cathedral* style (Morgan 2000).

OpenFOAM constitutes a C++ CFD toolbox for customised numerical solvers (over 60 of them) that can perform simulations of basic CFD, combustion, turbulence modelling, electromagnetics, heat transfer, multiphase flow, stress analysis and even financial mathematics. It has a large user base across most areas of engineering and science, from both commercial and academic organisations. The number of OpenFOAM users has been steady increasing. It is now estimated to

**Table 1.** CFD codes under study.

| Code | Regime | Numerical method | Mesh | Solution | Algorithm |
|---|---|---|---|---|---|
| OpenFoam | Incompressible | Finite volume | Unstructured | Implicit | Simple |
| Alya | Incompressible | Finite element | Unstructured | Semi-implicit | Fractional step |
| CHORUS | Compressible | Finite volume | Structured | Explicit | Operator splitting |

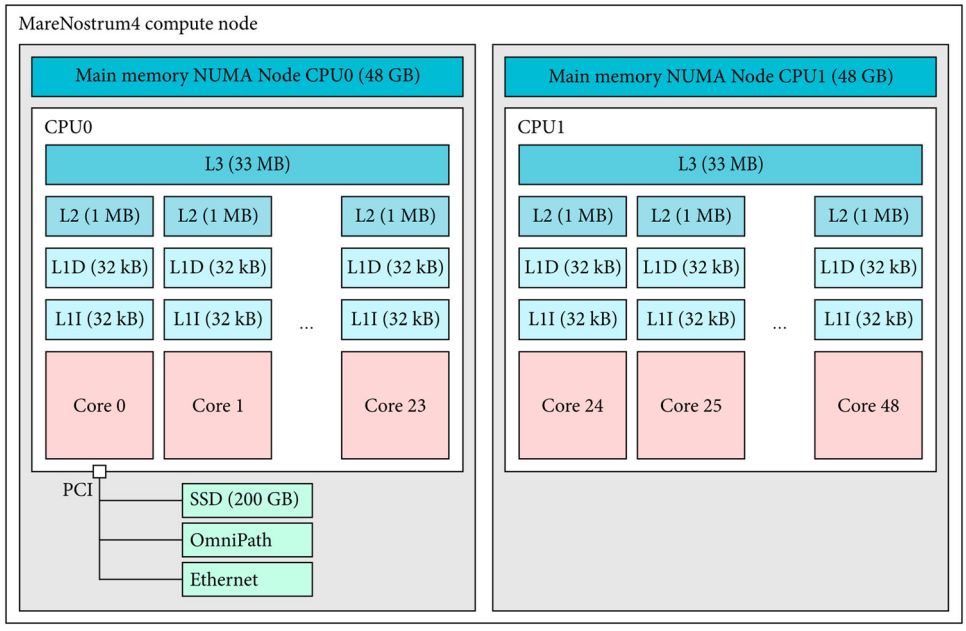



**Figure 1.** Schematic view of a node of MareNostrum4.

be in the order of thousands, with the majority of them being engineers in Europe. It is one of the most used open-source CFD code used in HPC environment.

### 2.2.2. Physical Problem

The problem considered in this work is the simulation of automotive aerodynamics using the DrivAer model, made publicly available by the Technical University of Munich .[4] The model consists of 61M cells. The solver used is the `simpleFoam` based on the SIMPLE (**S**emi-**I**mplicit **M**ethod for **P**ressure **L**inked Equations) algorithm implemented in OpenFOAM.

### 2.2.3. Numerical Method

The main features of the numerical method OpenFOAM relies upon are: segregated, iterative solution of linear algebra solvers, unstructured finite volume method (FVM), co-located variables, equation coupling (e.g. SIMPLE, PIMPLE, etc.). It uses C++ and object-oriented programming to develop a syntactical model of *equation mimicking* and scalar-vector-tensor operations .[5] Algorithm 1 shows a code snippet for the Navier–Stokes equations. Equation mimicking is quite obvious: `fvm:laplacian` means an implicit finite volume discretisation for the Laplacian operator, and similarly for `fvm::div` for the divergence operator. On the other hand, `fvc::grad` means an explicit finite volume discretisation for the gradient operator. The parentheses `(,)` means the product of the enclosed quantities, including tensor products. The discretisation is based on low-order (typically second-order) gradient schemes. It is possible to select different choices of iterative solvers for the solution of the linear solver algebra based on SpMV (Sparse Matrix-Vector) multiplication, ranging from Conjugate Gradient up to Multi-Grid Methods. The temporal schemes can be selected among first- and second-order implicit Crank–Nicolson and Euler schemes.

**Algorithm 1** OpenFOAM pseudo-code for Navier–Stokes equations

```
Solve (
    fvm::ddt(rho,U)
    +fvm::div(U,U)
    -fvm::laplacian(mu,U)
    ==
    -fvc::grad(p)
    +f
);
```

### 2.2.4. Parallelisation

The method of parallel computing used by OpenFOAM is based on domain decomposition, in which the geometry and associated fields are broken into

subdomains and allocated to separate processors for solution. The process of parallel computation involves: decomposition of mesh and fields; running the application in parallel; and, post-processing the decomposed case .[6] The method of parallel computing used by OpenFOAM is based on the standard MPI. A convenient interface, `Pstream`, that is a stream-based parallel communication library, is used to plug in any MPI library into OpenFOAM. It is a light wrapper around the selected MPI interface (Jasak 2012). Traditionally, FVM parallelisation uses the `halo layer` approach: data for cells next to a processor boundary is duplicated. Halo layer covers all processor boundaries and is explicitly updated through parallel communication cells. Instead, OpenFOAM operates in `zero halo layer approach`, which gives flexibility in the communication pattern. FVM operations 'look parallel' without data dependency (Jasak 2011).

#### 2.2.5. Actual Bottlenecks

From previous works (Culpo 2011; Bnà et al. 2019), it is known that the performance of the linear solvers limits the scalability of OpenFOAM to the order of a few thousand of cores. Meantime, the technological trends observed in the most powerful supercomputers (e.g. the Top500 list[7]) reveal that upcoming exascale systems will require the use of millions of cores. As of today, the well-known bottlenecks limiting the full enablement of OpenFOAM on massively parallel clusters are (i) the limit in the parallelism paradigm (Pstream Library); (ii) the I/O data storage system; (iii) the sub-optimal sparse matrices storage format (LDU) that does not enable any cache-blocking mechanism (SIMD, vectorisation).

The recently formed HPC technical committee[8] aims at working together with the community to overcome the aforementioned HPC bottlenecks. The priorities of the HPC technical committee are:

- To create an open and shared repository of HPC benchmarks, to provide the community with a homogeneous term of reference to compare different architectures, configurations and different software environments;
- To enable GP-GPUs and others accelerators;
- To enable Parallel I/O: Adios 2 I/O library as function object is now a git submodule in the OpenFOAM develop branch.

### 2.3. Alya: Unstructured Mesh Finite Element

#### 2.3.1. Context

Alya is a high performance computational mechanics code to solve complex coupled multi-physics/multi-scale/multi-domain problems, which are mostly coming from the engineering realm. Among the different physics solved by Alya we can mention: incompressible/compressible flows, non-linear solid mechanics, chemistry, particle transport, heat transfer, turbulence modelling, electrical propagation, etc. Alya (Vázquez et al. 2016) was specially designed for massively parallel supercomputers and is part of the Unified European Application Benchmark Suite (UEABS) a set of 13 codes highly scalable, relevant and publicly available.

#### 2.3.2. Physical Problem

The problem considered in this work is the simulation of an airplane, extensively described in Borrell et al. (2020). The mesh is hybrid and comprises 31.5M elements: 15.4M tetrahedra, 7K pyramids and 16.1M prisms. The algorithm solves incompressible flows, using a low-dissipative Galerkin method (Lehmkuhl et al. 2019), together with the VREMAN LES model for the subgrid scale stress tensor (Vreman 2004). A wall model is used at the airplane surface, implemented as a mixed Dirichlet and Neumann condition.

#### 2.3.3. Numerical Method

A Runge–Kutta scheme of third order is considered as the time discretisation scheme, involving three consecutive assemblies of the residual of the momentum equation, obtained through a loop over the elements and the boundaries of the mesh. Pressure is stabilised through a projection step, which consists of solving a Poisson equation for the pressure (Codina 2001), and then the velocity is corrected to satisfy mass conservation. Here, a Deflated Conjugate Gradient (Löhner et al. 2011) with linelet preconditioner (Soto, Löhner, and Camelli 2003) is used. The solution strategy is illustrated in Algorithm 2.

#### 2.3.4. Parallelisation

Alya implements different levels of parallelisation including MPI for distributed memory systems, OpenMP or OmpSs for shared memory (Garcia-Gasulla et al. 2019) and GPU support among others. As far as the parallelisation is concerned, in this work we rely exclusively on MPI as this is the version used in production runs. The MPI parallelisation

**Algorithm 2** Alya: solution strategy for solving Navier–Stokes equations.

```
for time steps do
    Compute time step
    for Runge–Kutta steps do
        Assemble residual of momentum equations
        Solve momentum explicitly
    end for
    Solve pressure equation with DCG and linelet preconditioner
    Correct velocity
end for
```

of Alya relies on Metis to partition the mesh in subdomains. It should be noted that contrary to classical implementations of the Finite Volume and Finite Difference methods, halo cells or halo nodes are not required to assemble the matrices or residuals in the Finite-Element method (Houzeaux et al. 2018). Avoiding duplicated work on these cells, provides a certain advantage in the assembly phase for the Finite-Element method, where the scalability only depends upon the control of the load balance. Although we have treated the load imbalance due to the different types of elements in previous papers (Garcia-Gasulla et al. 2019; Borrell et al. 2020), here, no specific strategy has been employed.

### 2.4. CHORUS: Structured Mesh Finite Volume

#### 2.4.1. Context

CHORUS is an in-house code developed at LIMSI-CNRS laboratory. It is a parallel (MPI) DNS/LES solver specially designed to solve unsteady high-Reynolds number compressible flows in the transonic and supersonic regimes. In these regimes, dealing with flows involving shock waves, one must use a numerical scheme which can both represent small scale structures with the minimum of numerical dissipation, and capture discontinuities with the robustness that is common to Godunov-type methods. To achieve this dual objective, high-order accurate shock capturing schemes is privileged. A shock capturing procedure has then been used to avoid spurious oscillations produced by high-order schemes in the vicinity of discontinuities such as shock waves (Pirozzoli 2011), for instance.

#### 2.4.2. Physical Problem

The problem considered in this work is the simulation of the well documented 3D Taylor–Green vortex at a Reynolds number of 1600 in a periodic cube (Wang et al. 2013; Brachet et al. 2006). A 3D periodic domain ($\Omega$) of $2\pi$ non-dimensional side length is considered. The initial flow field, solution of the Navier–Stokes equations, consists in an analytic solution of eight planar vortices (Brachet et al. 2006). A Cartesian uniform grid is used with the same spacing in each direction. The number of grid points used in this test case is 25M for the multi-node study and 3M for the intra node one.

#### 2.4.3. Numerical Method

The Navier–Stokes equations are solved by using a high-order finite volume approach on a Cartesian mesh. An operator splitting procedure is employed that splits the resolution into the Euler part and the viscous problem. The Euler part is discretised by means of a high-order one-step Monotonicity Preserving scheme, namely the OSMP7 scheme (Daru and Tenaud 2004), based on a Lax–Wendroff approach, which ensures a seventh-order of accuracy in both time and space in the regular regions. A Monocity-Preserving (MP) criterion is added to deal with discontinuity capturing by locally relaxing the classic TVD constraints for such scheme. Besides, the discretisation of the diffusive fluxes is obtained by means of a classical second-order centred scheme that has been coupled with a second-order Runge–Kutta time integration. The numerical approach is extensively described by Daru and Tenaud (2004).

#### 2.4.4. Parallelisation

The MPI parallelisation of CHORUS relies on the default cartesian mesh decompositions. At the subdomain interfaces, four ghost cells are required in the normal to the interface direction.

## 3. Methodology

The European Center of Excellence Performance Optimisation and Productivity[9] (POP) defined a set of metrics (Wagner et al. 2017) for modelling the performance of parallel applications. We apply such performance analysis method to the CFD applications introduced in Section 2. In the rest of the document, we refer to the efficiency metrics used for our performance analysis as *POP metrics* or *efficiency metrics*. The POP metrics express the efficiency of the MPI parallelisation and can be computed for any MPI application. They consist of indicators that serve as a guide for the following detailed analysis to spot the exact factors limiting the scalability.

For the performance analysis of each application, we apply the following steps:

**Tracing** – Run the application with a relevant input and core count and obtain a trace from this run.

**Selection of the focus of analysis (FoA)** – Based on the trace, select the region where to focus the performance analysis. To select the FoA first, we identify the iterative pattern within the trace and then select a couple of representative iterations disregarding the initialisation and finalisation phases.

**Single-node analysis** – Compute the efficiency metrics for different number of MPI processes in a single node. Based on these metrics we study the limiting factors and performance issues when scaling within a compute node. When the input set does not allow to run on a single node we will do the study using a constant number of MPI processes and increase the number of MPI processes spawned per node, this means that we will use more nodes as they will not be fully used.

**Multi-node analysis** – Compute the efficiency metrics when using more than one compute node.On MareNostrum4, we study from 1 to 16 nodes (corresponding to 48–768 MPI processes). Note that for CHORUS we extend our study from 8 to 32 compute nodes.As for the single-node analysis, we leverage the efficiency metrics to spot the issues that limit the scalability of the code on multiple nodes.

We focus on analysing the parallel efficiency of the MPI parallelisation, so no hybrid distributed+shared memory techniques are analysed here. Also, the study is based on traces obtained from real runs. Traces contain information about all the processes involved in an execution, and they can include, among others, information on MPI or I/O activity as well as hardware counters.

All the studies performed in this paper are using strong scalability but the methodology can be applied as well to weak scaling codes.

### 3.1. Metrics for Performance Analysis

For the definition of the POP metrics needed in the rest of the paper, we use the simplified model of execution depicted in Figure 2.

We call $P = \{p_1, \ldots, p_n\}$ the set of MPI processes. Also we assume that each process can only be in two states over the execution time: the state in which it is performing computation, called *useful* (blue) and the state in which it is not performing computation, e.g. communicating to other processes, called *not-useful* (red) . For each MPI process $p$, we can therefore define the set $U_p = \{u_1^p, u_2^p, \ldots, u_{|U|}^p\}$ of the time intervals where the application is performing useful computation (the set of the blue intervals). We define the sum of the durations of all useful time intervals in a process $p$ as:

$$D_{U_p} = \sum_{U_p} \blacksquare = \sum_{j=1}^{|U_p|} u_j^p \quad . \tag{1}$$

Similarly we can define $\overline{U}_p$ and $D_{\overline{U}_p}$ for the not-useful intervals.

The *Load Balance* measures the efficiency loss due to different load (useful computation) for each process globally for the whole FoA. Thus, it represents a

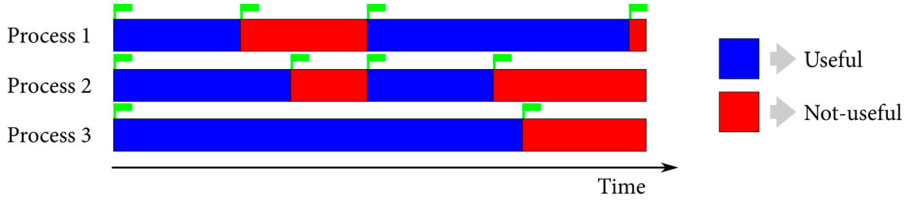



**Figure 2.** Example of parallel execution of three MPI processes.

meaningful metric to characterise the performance of a parallel code. We define $L_n$ the *Load Balance* among $n$ MPI processes as:

$$L_n = \frac{\text{avg}_{|P|}\left(\sum_{U_p} \blacksquare\right)}{\max_{|P|}\left(\sum_{U_p} \blacksquare\right)} = \frac{\sum_{i=1}^{n} D_{U_i}}{n \cdot \max_{i=1}^{n} D_{U_i}}$$

The *Transfer efficiency* measures inefficiencies due to transferring data with MPI communications. In order to compute the *Transfer efficiency*, we compare a trace of a real execution with the same trace processed by the Dimemas simulator (see Section 3.2 for more details) assuming all communication has been performed on an ideal network (i.e. with zero latency and infinite bandwidth). We define $T_n$ the *Transfer efficiency* among $n$ MPI processes as:

$$T_n = \frac{\max_{|P|} t'_p}{\max_{|P|} t_p}$$

where $t_p$ is the real runtime of the process $p$, while $t'_p$ is the runtime of the process $p$ when running on an ideal network.

The *Serialisation Efficiency* measures inefficiency due to idle time within communications (i.e. time where no data is transferred) and is expressed as:

$$S_n = \frac{\max_{|P|} D_{U_p}}{\max_{|P|} t'_p}$$

where $D_{U_p}$ is computed as in Equation (1) and $t'_p$ is the runtime of the process $p$ when running on an ideal network.

The *Communication Efficiency* is the product of the *Transfer efficiency* and the *Serialisation efficiency*: $C_n = T_n \cdot S_n$. Combining the *Load Balance* and the *Communication efficiency* we obtain the *Parallel efficiency* for a run with $n$ MPI processes: $P_n = L_n \cdot C_n$. Its value reveals the inefficiency in splitting computation over processes and then communicating data between processes. A good value of $P_n$ (i) ensures an even distribution of computational work across processes and (ii) minimises the time spent in communicating data among processes. Once defined the *Parallel efficiency*, the remaining possible source of inefficiencies can only come from the blue part of Figure 2, so from the useful computation performed within the parallel applications when changing the number of MPI processes. We call this *Computation Scalability* and we define it in case of strong scalability as:

$$U_n = \frac{\sum_{P_0} D_{U_p}}{\sum_{P} D_{U_p}}$$

where $\sum_{P_0} D_{U_p}$ is the sum of all useful time intervals of all processes when running with $P_0$ MPI processes and $\sum_{P} D_{U_p}$ is the sum of all useful time intervals of all processes when running with $P$ MPI processes, with $P_0 < P$.

We know that the execution time $t$ of a single program can be computed as:

$$t = \frac{i}{f\rho}$$

where $i$ is the number of instruction executed by the program, $f$ is the frequency of execution of the instructions of the program and $\rho$ is the IPC. This way we can divide the study of the *Computation Scalability* in three independent factors all related to the useful computation (blue part of Figure 2): the total number of instructions executed ($i$), the operational frequency of the program ($f$), and the IPC ($\rho$). The reader should remember that in a multi process machine, the frequency of execution of the program can be different from the frequency of the CPU.

We compute the scalability $\sigma$ of instructions $\sigma_i$ as the ratio of $i$ when running with $P_0$ MPI processes (taken as baseline) divided by the value of $i$ when running with $P_n$ MPI processes, varying $P_n$ in $\{4, 8, 16, 24, 48, 96, 192, 384, 768\}$:

$$\text{Instruction scalability} \qquad \sigma_i = \frac{i_{P_0}}{i_{P_n}},$$

Similarly, we compute the scalability $\sigma$ of the IPC $\sigma_\rho$ and frequency $\sigma_f$ as the ratio of $\rho$ and $f$ when running with $P_n$ MPI processes divided by the value of $\rho$ and $f$ when running with $P_0$ MPI processes (taken as baseline), varying $P_n$ in $\{4, 8, 16, 24, 48, 96, 192, 384, 768\}$:

$$\text{IPC scalability } \sigma_\rho = \frac{\rho_{P_n}}{\rho_{P_0}} \text{ with}$$

$$\rho_{P_0} = \frac{\sum_{P_0} i_0}{\sum_{P_0} c_0}, \rho_{P_n} = \frac{\sum_{P_n} i_n}{\sum_{P_n} c_n}$$

$$\text{Frequency scalability } \sigma_i = \frac{f_{P_0}}{f_{P_n}}$$

For an homogeneous representation, we report the scalability values $\sigma$ as '%' so to be compared with the *Parallel efficiencies* values.

Finally, we can combine the efficiency metrics introduced so far in the *Global Efficiency* defined as $G_n = P_n \cdot U_n$.

### 3.2. Performance Tools

For the analysis performed in this paper, we adopt a set of tools developed within the Barcelona Supercomputing Center.

**Extrae** is a tracing tool developed at BSC (Servat et al. 2013). It collects information such as PAPI counters, MPI and OpenMP calls during the execution of an application. The extracted data are stored in a trace that can be visualised with Paraver. Since the overhead introduced by the instrumentation can affect the overall performance we measure the overhead by comparing the time when running with and without instrumentation. For each run, we report the value of overhead as a percentage of the execution time without instrumentation.

**Paraver** takes traces generated with Extrae and provides a visual interface to analyse them (Pillet et al. 1995). The traces can be displayed as timelines of the execution but the tool also allows us to do more complex statistical analysis.

**Dimemas** accepts as input an Extrae trace gathered from an execution on a real HPC cluster. The output of Dimemas is a new trace reconstructing the time behaviour of the parallel application as if it was ran on a machine modelled by a set of performance parameters (also provided as input). For this paper, we employed Dimemas to simulate the behaviour of the CFD codes under study when running on a parallel system with an ideal network, i.e. zero latency and infinite bandwidth.

**Clustering** is a tool that, based in a Extrae trace, can identify regions of the trace with the same computational behaviour. This classification is done based on hardware counters defined by the user. Clustering is useful for detecting iterative patterns and regions of code that appear several times during the execution.

**Tracking** helps to visualise the evolution of the data clusters across multiple runs when using different amount of resources and exploiting different levels of parallelism.

Combining the metrics defined in Section 3.1 and the tools just introduced, we are able to summarise the *health* of a parallel applications running on a different number of MPI processes in a heat-map table. Rows indicate different POP metrics while columns indicate the number of MPI processes of a given run. Each cell is colour-coded in a scale from green, for values close to 100% efficiency; yellow, for values between 80% and 100% of the efficiency; down to red, for values below 80% efficiency.

## 4. Performance Analysis of CFD Methods

In this section, we apply the methods introduced in Section 3 to each of the CFD codes. For each application, first, we briefly introduce the structure and then we identify the FoA visualising it on an execution timeline. Since all codes are iterative, the FoA region is a set of three to five iterations.

Within the FoA region, we then collect and present the POP metrics in two scenarios:

*Single-node study*, i.e. when changing the number of processes assigned to each node up to filling a node with a number of MPI processes equivalent to the number of cores.

*Multi-node study*, i.e. when scaling the application up to 16 (768 cores) or 32 (1536 cores) full compute nodes of MareNostrum4.

We then study the spots where the POP efficiency metrics drop with the goal of predicting which fundamental factors will harm the performance of each application when scaling up the number of MPI processes. For each application, we identify critical points that can be related to hardware or software configurations as well as parallelisation practices.

### 4.1. OpenFOAM: Unstructured Mesh Finite Volume

Figure 3 shows a timeline visualised with Paraver of an execution of OpenFOAM on one node of MareNostrum4 (using 48 MPI processes). The $x$-axis represents the time while each value of the $y$-axis corresponds to one MPI process. The colour in this case represents the MPI call that is being executed in a given instant of time by a process. The white colour means that the process is performing useful computation.

In the top timeline, we can clearly identify the initialisation phase and the iterative pattern. The Focus of Analysis (FoA) are the three timesteps shown in

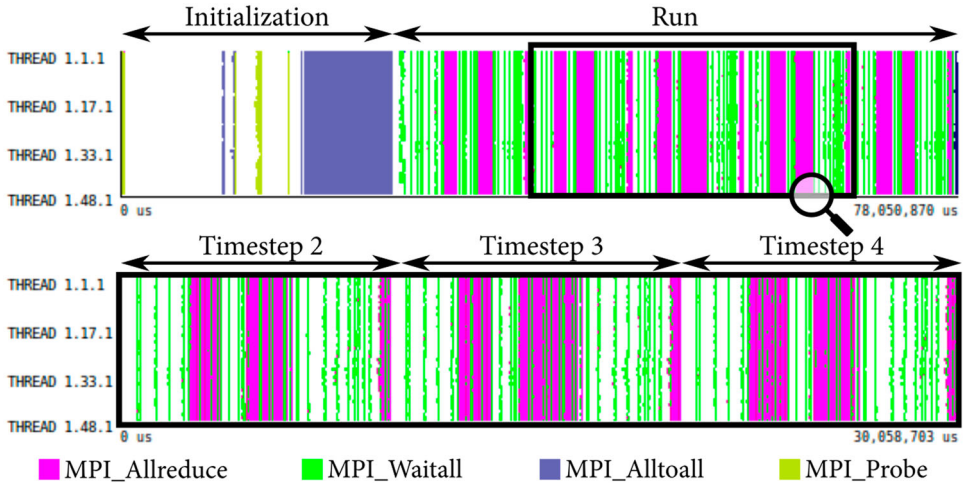



**Figure 3.** Timeline of OpenFOAM.

the bottom timeline. The analysis presented in the following sections is performed for this region.

#### *4.1.1. Single-node Analysis*

In Table 2, we can see the POP efficiency metrics obtained for OpenFOAM when using from 2 to 48 MPI processes within a computational node of MareNostrum4. We observe that the main factor limiting the scalability within a node is the IPC scalability, reaching values around 60% with 24 and 48 MPI processes.

Since we perform a strong scalability study within a node, we can reasonably expect that increasing the number of processes increases the pressure to the memory subsystem.

In Figure 4, we see a matrix of histograms obtained with Paraver from the OpenFOAM traces. Each one of these histograms depicts one horizontal line per MPI process. On the *x*-axis, the corresponding metric is represented (i.e. number of misses to L2 cache per 1000 Instructions, number of misses to L3 cache per 1000 Instructions and instructions per cycle). The colour represents the percentage of useful time that a given metric assumed in each process. The colour code is a gradient going from low values (painted as light green) to high values (painted as dark blue).

This matrix of histograms shows that the number of L2 misses is constant among the different processes of the same run and it is not affected by the increment of the number of MPI processes per node. We can deduce it from the blue lines in the first column of histograms (L2 MPKI): they are vertical within the same histogram, meaning that all processes stay a similar percentage of time with the same value of L2 MPKI. Also, all other histograms of the same column show a similar distribution of blue lines meaning

**Table 2.** Single-node efficiency metrics of OpenFOAM.

| | | | | | | |
|---|---|---|---|---|---|---|
| Global efficiency | 98.56 | 102.13 | 93.56 | 86.29 | 61.44 | 76.35 |
| ├ Parallel efficiency | 98.56 | 95.69 | 94.14 | 93.21 | 90.67 | 92.76 |
| │ ├ Load balance | 99.69 | 97.38 | 96.76 | 95.78 | 96.06 | 95.96 |
| │ └ Communication eff. | 98.87 | 98.26 | 97.29 | 97.32 | 94.39 | 96.67 |
| │ ├ Serialization eff. | 98.94 | 98.39 | 97.53 | 97.72 | 94.88 | 98.20 |
| │ └ Transfer eff. | 99.93 | 99.87 | 99.75 | 99.59 | 99.49 | 98.44 |
| └ Computation scalability | 100.00 | 106.73 | 99.38 | 92.57 | 67.77 | 82.30 |
| ├ IPC scalability | 100.00 | 102.80 | 97.26 | 88.57 | 60.48 | 60.57 |
| ├ Instruction scalability | 100.00 | 99.23 | 98.11 | 98.88 | 98.56 | 98.28 |
| └ Frequency scalability | 100.00 | 104.63 | 104.15 | 105.71 | 113.68 | 138.26 |
| Speedup | 1.00 | 2.07 | 3.80 | 5.25 | 7.48 | 18.59 |
| Average IPC | 1.21 | 1.24 | 1.17 | 1.07 | 0.73 | 0.73 |
| Average frequency [GHz] | 1.46 | 1.52 | 1.52 | 1.54 | 1.66 | 2.01 |
| Instrumentation overhead | 7.32% | 6.40% | 6.81% | 6.23% | 3.23% | 4.49% |

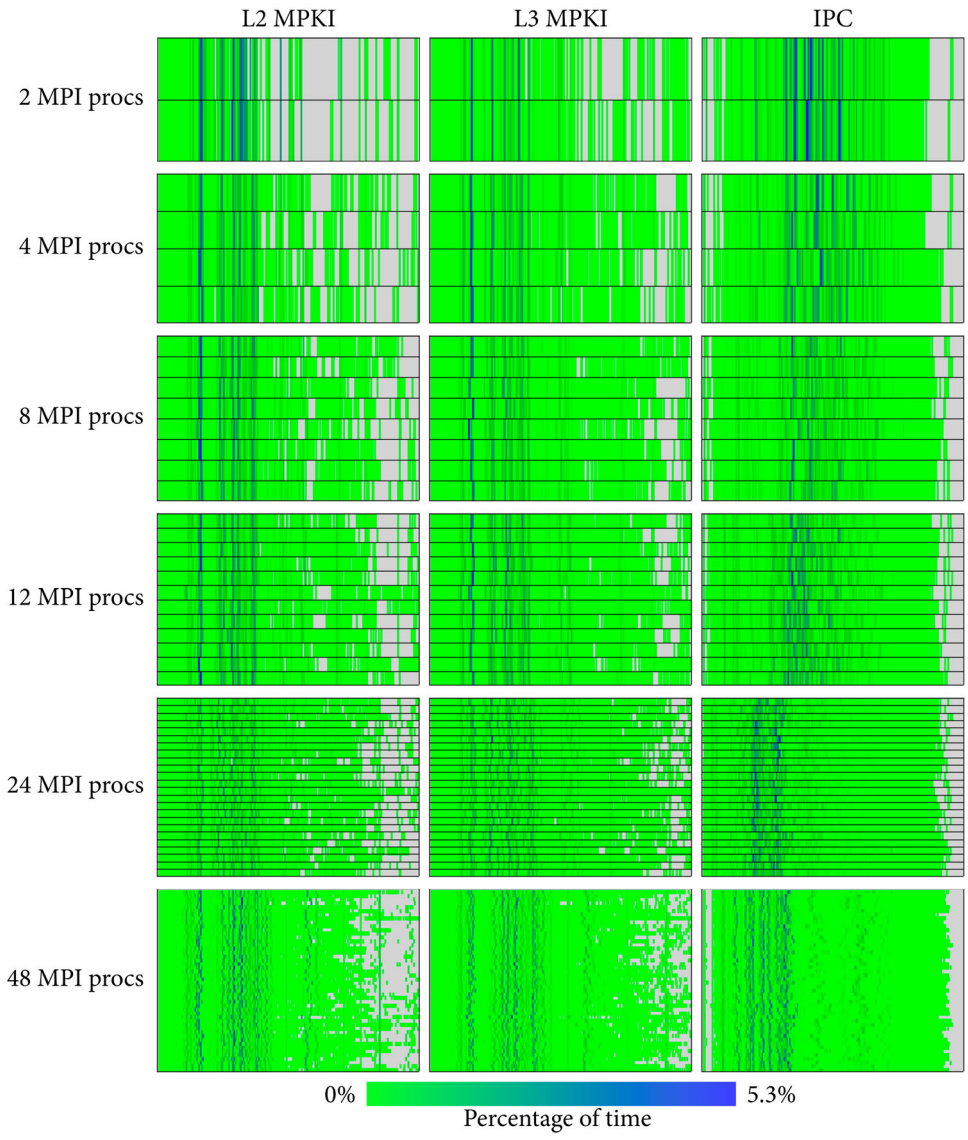



**Figure 4.** OpenFOAM: Histograms for L2 MPKI, L3 MPKI and IPC when increasing number of MPI processes per node.

that increasing the number of MPI processes does not affect the percentage of time spent in each value of L2 MPKI.

In a similar way, we can observe that the number of L3 misses does not present a relevant variability as the vertical blue lines follow the same pattern for all histograms in the second column (L3 MPKI).

Finally, as was highlighted by the efficiency metrics of Table 2, the IPC presents a sudden drop when going from 12 to 24 MPI processes per node. We can recognise this in Figure 4 because the vertical lines in the third column of histograms (IPC) move to the left (lower values of IPC) when going from 12 to 24 MPI processes per node.

With this we demonstrate that the drop in IPC is not due to a higher number of misses in L2 nor L3 caches. At this point the saturation of memory bandwidth seems to be the most feasible reason for the IPC drop. In order to verify this we studied then the density of misses over time, i.e. the total number of L3 misses per microseconds executed by all process.

In Figure 5, we show the number of misses per microsecond for the L3 cache ($x$-axis) when increasing the number of MPI processes ($y$-axis). Since the L3 is shared among all cores of a socket, this corresponds to the number of petitions to main memory that are issued per microsecond by each socket.

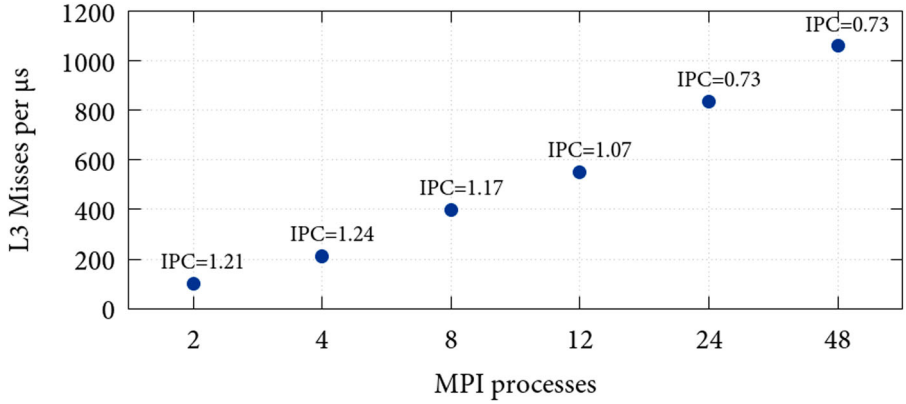



**Figure 5.** OpenFOAM: L3 Misses per microsecond.

We can see how the pressure to memory increases as the number of MPI processes increases.

We conclude that the main factor limiting the scalability of OpenFOAM within a computational node is the amount of L3 cache misses. This limitation could be mitigated with a data layout that ensures a better data locality for the different levels of caches.

### *4.1.2. Multi-node Analysis*

In Table 3, we can see the efficiencies obtained by OpenFOAM when using multiple nodes from 1 to 16 (48 to 768 MPI processes). We observe that the main factor limiting the scalability is the transfer efficiency followed closely by the serialisation.

An important observation for the multi-node efficiency is related to the instrumentation overhead, i.e. the interference that the instrumentation tool (Extrae) adds to the application while gathering figures to be stored in a trace. If we look at the instrumentation overhead we notice that for 384 MPI processes the runtime with the instrumentation is ∼17% longer than without instrumentation and with 768 MPI processes this overhead grows to 36%. This values of overhead tell that absolute measurements in these two cases should be discarded. Nevertheless, relative figures can be considered as reliable and most importantly the trend of the efficiency metrics maintains its validity. We still consider that the main limiting factor for the performance of OpenFOAM will be the transfer, because even if we ignore the columns of 384 and 768 the efficiency values with 48, 96 and 192 MPI processes suggest that when increasing the number of nodes the limiting factor will be the transfer efficiency.

As we explained in Section 3.1, the Transfer efficiency accounts for time spent inside MPI communication that it is not due to synchronisation between processes. We split the transfer time per class of MPI calls to identify what kind of calls are the ones limiting the scalability. The class of calls are *P2P sync* including `MPI_send`, `MPI_recv`, and `MPI_sendrecv`; *P2P async* including `MPI_isend`, `MPI_irecv`, `MPI_waitall`; *Allreduce* including

**Table 3.** Multi-node efficiency metrics of OpenFOAM.

| | | | | | |
|---|---|---|---|---|---|
| Global efficiency | 92.76 | 92.84 | 89.05 | 83.61 | 73.06 |
| ├ Parallel efficiency | 92.76 | 89.61 | 85.98 | 77.39 | 65.92 |
| │ ├ Load balance | 95.96 | 94.72 | 94.27 | 93.71 | 92.71 |
| │ └ Communication eff. | 96.67 | 94.60 | 91.20 | 82.59 | 71.11 |
| │ ├ Serialization eff. | 98.20 | 96.75 | 96.64 | 92.05 | 85.51 |
| │ └ Transfer eff. | 98.44 | 97.78 | 94.37 | 89.71 | 83.16 |
| └ Computation scalability | 100.00 | 103.61 | 103.58 | 108.04 | 110.83 |
| ├ IPC scalability | 100.00 | 103.11 | 104.32 | 109.71 | 118.84 |
| ├ Instruction scalability | 100.00 | 99.53 | 96.41 | 95.95 | 91.83 |
| └ Frequency scalability | 100.00 | 100.96 | 102.98 | 102.63 | 101.55 |
| Speedup | 1.00 | 2.00 | 3.84 | 7.21 | 12.60 |
| Average IPC | 0.73 | 0.75 | 0.76 | 0.80 | 0.87 |
| Average frequency [GHz] | 2.01 | 2.03 | 2.07 | 2.07 | 2.05 |
| Instrumentation overhead | 4.49% | 5.54% | 9.37% | 16.85% | 36.00% |

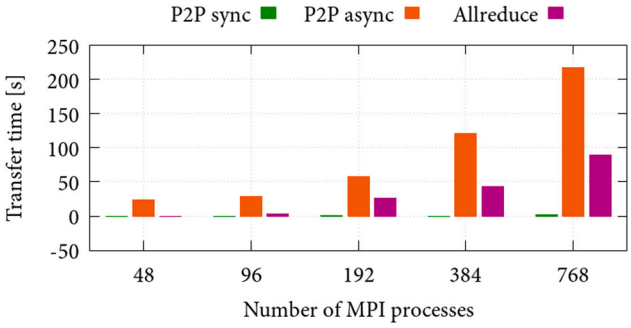



**Figure 6.** OpenFOAM: Transfer time per MPI primitive.

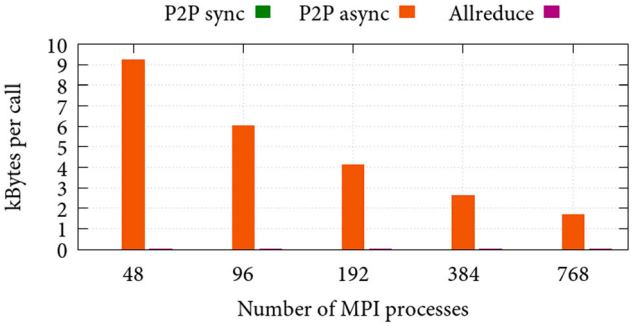



**Figure 7.** OpenFOAM: Bytes exchanged per MPI primitive.

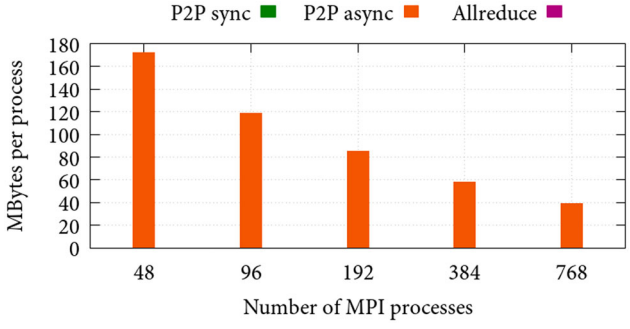



**Figure 8.** OpenFOAM: Bytes exchanged by each process per MPI primitive.

`MPI_Allreduce`. In Figure 6, we can see the total time spent in transfer when increasing the number of MPI processes per class of MPI call. We observe that the asynchronous point to point MPI calls are the ones that account for more transfer time.

Figures 7 and 8 show the number of bytes exchanged, in total and per process. It is clear that the total amount of data being exchanged decreases. Therefore, the problem does not seem to be related to network bandwidth.

In Figure 9, we can see the number of MPI calls of each type executed per process. We observe that the number of asynchronous point to point MPI calls performed per process increases with the number of MPI

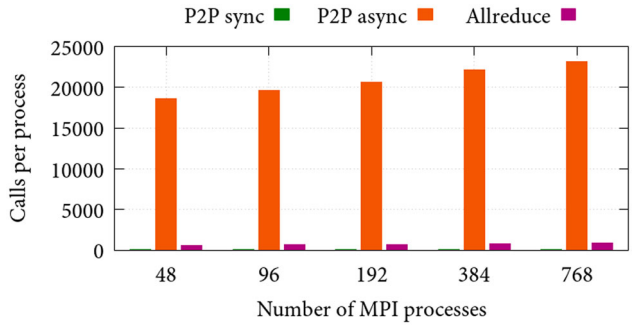



**Figure 9.** Number of calls by each process per MPI primitive.

processes. This observation together with the previous charts demonstrate that the transfer efficiency lost by OpenFOAM is due to the overhead introduced by calling the asynchronous point to point MPI calls a high number of times to exchange a low number of bytes. The suggestion for the developers to address this issue would be to try grouping several point to point MPI calls into a single one to reduce the number of calls. If the algorithm does not allow this, it means the focus must be put in the partition of the problem and the number of neighbours of each process. Reducing the number of neighbours would reduce in fact the number of point-to-point communications.

### 4.2. Alya: Unstructured Mesh Finite Element

In Figure 10, we report a timeline of Alya running in one node of MareNostrum4 using 48 MPI processes. We identify the initialisation and the finalisation phases as well as the iterative phase in between representing the most time consuming and compute intensive part of large Alya simulations. In the following sections, we study the FoA shown in the timeline at the bottom of Figure 10 including four time steps of the iterative phase of Alya.

#### 4.2.1. Single-node Analysis

In Table 4, we report the POP efficiency metrics when running Alya on a single node of MareNostrum4. The header of the columns expresses the number of processes per node. The reader should remember that we run Alya with a fixed number of processes (48), spawning an increasing number of processes per node in each test. So, the column with the header '2' reports the efficiencies of Alya running with 48 MPI processes, spawning 2 processes per node on 24 compute nodes of MareNostrum4.

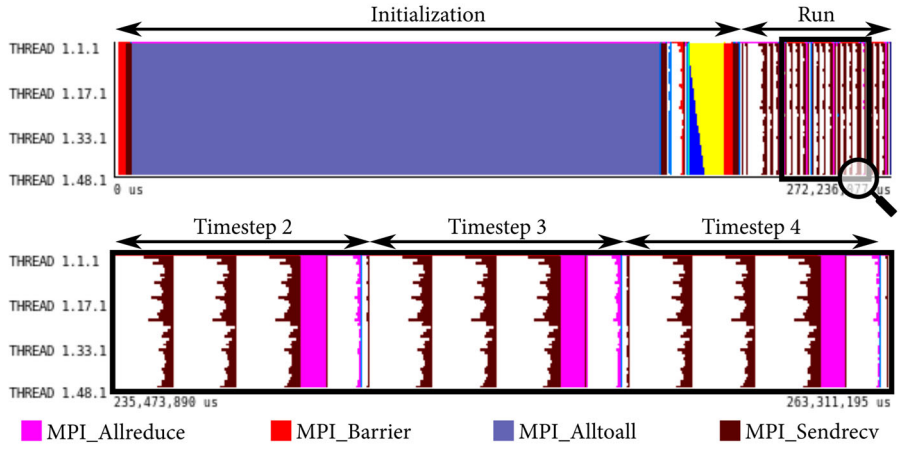
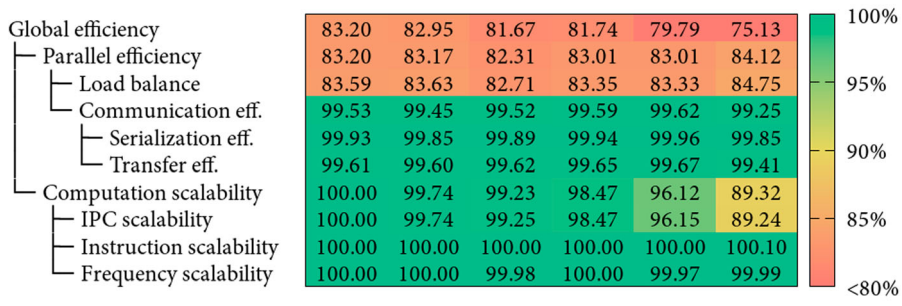



**Figure 10.** Timeline of Alya.

**Table 4.** Single-node efficiency metrics of Alya.

| | | | | | | |
|---|---|---|---|---|---|---|
| Global efficiency | 83.20 | 82.95 | 81.67 | 81.74 | 79.79 | 75.13 |
| ├ Parallel efficiency | 83.20 | 83.17 | 82.31 | 83.01 | 83.01 | 84.12 |
| │ ├ Load balance | 83.59 | 83.63 | 82.71 | 83.35 | 83.33 | 84.75 |
| │ └ Communication eff. | 99.53 | 99.45 | 99.52 | 99.59 | 99.62 | 99.25 |
| │ ├ Serialization eff. | 99.93 | 99.85 | 99.89 | 99.94 | 99.96 | 99.85 |
| │ └ Transfer eff. | 99.61 | 99.60 | 99.62 | 99.65 | 99.67 | 99.41 |
| └ Computation scalability | 100.00 | 99.74 | 99.23 | 98.47 | 96.12 | 89.32 |
| ├ IPC scalability | 100.00 | 99.74 | 99.25 | 98.47 | 96.15 | 89.24 |
| ├ Instruction scalability | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.10 |
| └ Frequency scalability | 100.00 | 100.00 | 99.98 | 100.00 | 99.97 | 99.99 |
| Speedup | 1.00 | 1.00 | 0.98 | 0.98 | 0.96 | 0.90 |
| Average IPC | 2.14 | 2.14 | 2.13 | 2.11 | 2.06 | 1.91 |
| Average frequency [GHz] | 2.09 | 2.09 | 2.09 | 2.09 | 2.09 | 2.09 |
| Instrumentation overhead | < 1% | < 1% | < 1% | < 1% | < 1% | < 1% |




We see that the load balance presents a low value that remains constant for the different runs. This behaviour is as expected because all runs refer to a partition of the problem into 48 MPI ranks. The load balance problem re-appear in the multi-node study and we analyse it in more details in the next Section. There we can see how it changes when increasing the number of MPI processes (partitions of the problem).

As often happens, Table 4 shows that after the load balance the main limiting factor when increasing the number of processes per node is the drop of the IPC.

To study this phenomenon, we use the Clustering tool introduced in Section 3.2. We clusterise each execution trace of Alya using IPC and number of instructions. We identify three main computational clusters corresponding to three well-known parts of the code: Residual Assembly (RAss), Timestep Computation (TsComp), Algebraic Solver (Solver).

In Figure 11, we highlight with different colours the clusters identified by the Clustering tool over a timeline. The blue region corresponds to the different Runge–Kutta iterations, introduced in Algorithm 2.

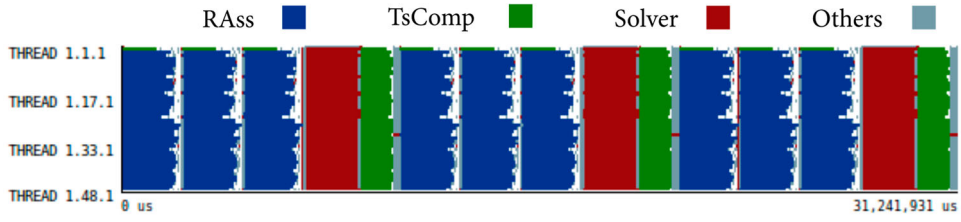



**Figure 11.** Alya: timeline of three iterations highlighting the clusters.

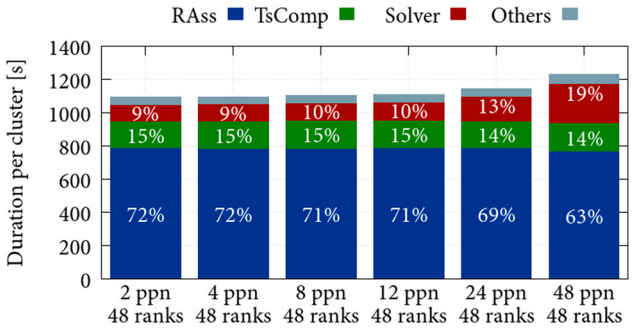



**Figure 12.** Alya: cluster duration with 48 MPI ranks with different processes per node distribution.

We characterise the clusters using their duration and we plot them in Figure 12. We notice that the Solver is the responsible for the performance degradation when increasing the number of processes per node.

The IPC drop could be generated by the saturation of resources (e.g. memory bandwidth). For this reason we study the density of L3 cache misses per microsecond per socket as we did for OpenFOAM in Figure 5. This time we perform the study on each cluster and

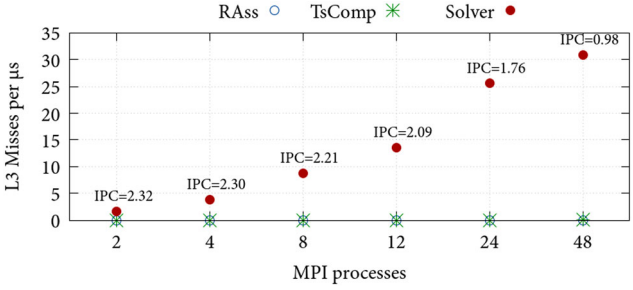



**Figure 13.** Alya: study of the density of L3 data cache misses per microsecond per socket.

we confirm that the Solver phase is the root cause of the IPC drop. In Figure 13, we report the number of L3 cache misses per microsecond per socket ($y$-axis) when changing the number of processes per node ($x$-axis). We notice that while the Residual Assembly phase and the Timestep Computation phase have a steady low L3 miss density, the Solver shows an increasing density of cache misses, corresponding to lower values of IPC when increasing the number of processes per node.

We conclude that IPC drop within the node is due to memory resource saturation when increasing the number of MPI processes per node. We suggest to find a better data layout that would allow a better cache data reuse.

### 4.2.2. *Multi-node Analysis*

In Table 5, we can find the efficiencies obtained by Alya when using from 48 to 768 MPI processes (1–16 nodes). We can see that the main factor limiting scalability is the load balance.

In Table 6, we can see the load balance in useful time as computed by the POP efficiency metrics, and the load balance in the number of instructions. We can conclude that the load balance of Alya comes from the partition of the problem, because some processes execute more instructions than others. The example considered for the present analysis is a full airplane simulation. The mesh is hybrid, composed of prisms in the boundary layer region, tetrahedra in the core flow and pyramids in the transition region. For this study, we have partitioned the mesh disregarding the type of elements, which cost for assembling the residuals is different. This relative cost is responsible for the load

**Table 5.** Multi-node efficiency metrics of Alya.

| | | | | | |
|---|---|---|---|---|---|
| Global efficiency | 83.95 | 80.06 | 81.42 | 76.51 | 65.34 |
| ├ Parallel efficiency | 83.95 | 82.38 | 81.18 | 74.75 | 62.69 |
| │ ├ Load balance | 84.51 | 83.63 | 83.09 | 80.19 | 70.95 |
| │ └ Communication eff. | 99.34 | 98.50 | 97.70 | 93.22 | 88.37 |
| │ ├ Serialization eff. | 99.71 | 99.18 | 99.14 | 96.67 | 93.95 |
| │ └ Transfer eff. | 99.64 | 99.32 | 98.55 | 96.43 | 94.05 |
| └ Computation scalability | 100.00 | 97.18 | 100.30 | 102.35 | 104.22 |
| ├ IPC scalability | 100.00 | 97.82 | 101.42 | 104.20 | 107.33 |
| ├ Instruction scalability | 100.00 | 99.51 | 99.20 | 98.73 | 97.91 |
| └ Frequency scalability | 100.00 | 99.85 | 99.69 | 99.49 | 99.18 |
| Speedup | 1.00 | 1.91 | 3.88 | 7.29 | 12.45 |
| Average IPC | 1.91 | 1.87 | 1.93 | 1.99 | 2.05 |
| Average frequency [GHz] | 2.09 | 2.09 | 2.08 | 2.08 | 2.07 |
| Instrumentation overhead | < 1% | 4.52% | < 1% | 2.27% | 2.09% |

**Table 6.** Alya: Load and Instruction Balance when increasing number of processes.

| Number of processes | 48 | 96 | 192 | 384 | 768 |
|---|---|---|---|---|---|
| Load Balance | 84.51% | 83.63% | 83.09% | 80.19% | 70.95% |
| Instruction Balance | 83.10% | 82.63% | 82.82% | 79.83% | 71.03% |

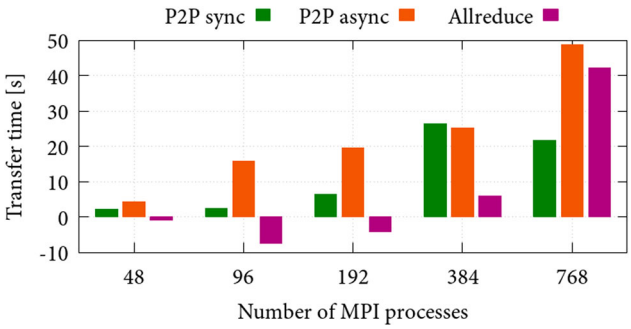



**Figure 14.** Alya: Transfer time per MPI primitive.

imbalance. This issue can be addressed using a better heuristic to partition the problem or use a dynamic load balancing mechanism as has been proven useful in other works (Garcia-Gasulla et al. 2019; Garcia, Labarta, and Corbalan 2014).

In the case of Alya, we study a second factor limiting the scalability, looking at Table 5 we can see that after the Load Balance, the main problems are serialisation and transfer. As serialisation problems usually come from load balancing in different phases with different patterns and we have already addressed the load balance problem, we study the transfer efficiency in Alya.

As we did in OpenFOAM, we analyse the transfer time spent by the different class of MPI calls. The class of calls are *P2P sync* including `MPI_send`, `MPI_recv`, and `MPI_sendrecv`; *P2P async* including `MPI_isend`, `MPI_irecv`, `MPI_waitall`; *Allreduce* including `MPI_Allreduce`. In Figure 14, we report the time spent communicating data for different types of MPI calls. We observe that the transfer time increases steadily for the asynchronous point-to-point MPI calls (P2P async). It also increases for the synchronous calls but stops increasing after 384 MPI processes, the amount of time spent in the AllReduce call also increases drastically for 768 MPI processes.

In Figures 15 and 16, we can see the number of bytes exchanged per MPI call and per MPI process, respectively. We can see that the number of bytes exchanged per synchronous and asynchronous MPI calls decreases drastically with the number of MPI processes. If we look at the number of bytes exchanged by process we observe that it also decreases for both kinds of calls as we increase the number of MPI processes.

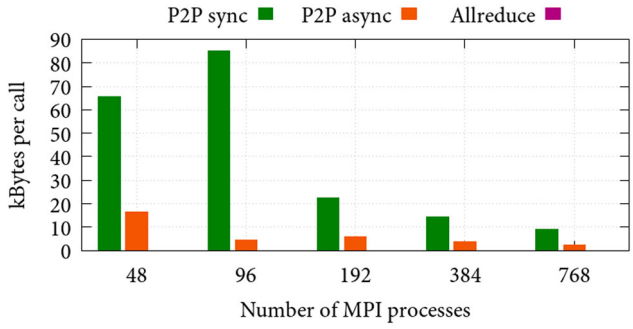



**Figure 15.** Bytes exchanged per MPI primitive.

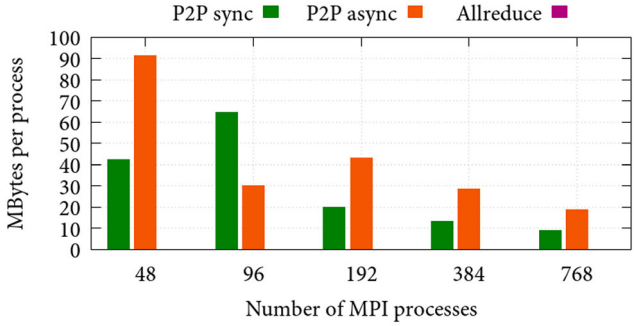



**Figure 16.** Byte exchanged by each process per MPI primitive.

In Figure 17, we show the number of calls performed of each type by each process. We observe that the number of asynchronous point to point is very high and increases with the number of MPI processes. Therefore, the transfer time spent in the asynchronous point to point MPI call is due to the high number of calls and the overhead associated to them. The suggestion to address this issue is to refactor, if possible, the communications to group them and use a single call to send several data. It can happen that algorithmically this is not possible, in that case the suggestion is to look

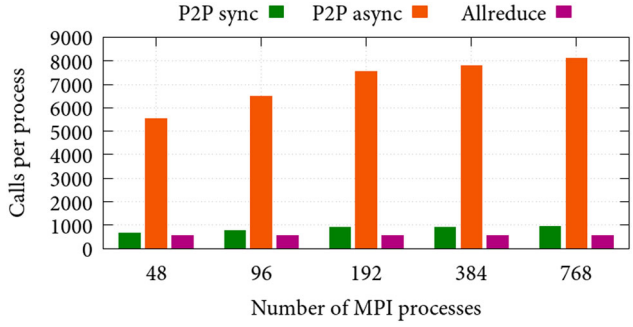



**Figure 17.** Number of calls by each process per MPI primitive.

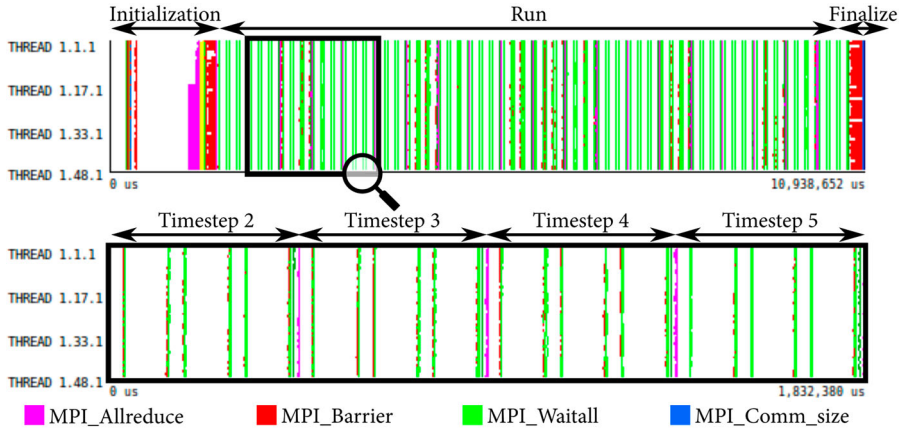



**Figure 18.** Timeline of Chorus.

at the partition of the problem because the high number of point to point communications is a symptom of having too many neighbours. In fact, a Hilbert Space Filling curve has been used to partition the mesh, and such a strategy does not enable to control the number of neighbours and the size of the interfaces.

### 4.3. CHORUS: Structured Mesh Finite Volume

In Figure 18, we can see a timeline of a run of CHORUS in one node of MareNostrum4 using 48 MPI processes. We can identify the initialisation, iterative and finalisation phases. In the following sections, we use the FoA shown in the timeline at the bottom of the image, including four time steps.

#### 4.3.1. Single-node Analysis

In Table 7, we can see the efficiencies obtained by CHORUS when increasing the number of MPI processes within a computational node. In the same way, as Alya the study of CHORUS can not be done in a single node because of the memory requirements of the input set. For this reason, we fix the number of MPI processes to 48 and increase the number of processes spawn per node. In Table 7, the label of columns does not refer to the total number of MPI processes but to the number of MPI processes per node. For example, the first column with a '2' label, means that we have executed 48 MPI processes using 24 nodes and spawning 2 MPI process per node.

The input set has been reduced respect to the one used in the multi-node study to be able to fit in less nodes.

We can observe in Table 7 that the main factor limiting the scalability within a node is the IPC scalability.

In Figure 19, we can find a matrix of histograms representing different metrics (L1 MPKI, L2 MPKI, L3 MPKI and IPC) at different number of MPI processes. The colour code represents the percentage of time in which the given metric assumed the value given on

**Table 7.** Single-node efficiency metrics of CHORUS.

| | | | | | | |
|---|---|---|---|---|---|---|
| Global efficiency | 92.19 | 89.07 | 84.62 | 80.40 | 65.11 | 37.12 |
| ├ Parallel efficiency | 92.19 | 91.93 | 90.73 | 90.50 | 91.36 | 94.22 |
| │ ├ Load balance | 99.23 | 97.60 | 98.66 | 98.78 | 98.14 | 98.61 |
| │ └ Communication eff. | 92.91 | 94.18 | 91.96 | 91.61 | 93.09 | 95.55 |
| │ ├ Serialization eff. | 99.29 | 99.20 | 98.17 | 97.76 | 98.38 | 99.06 |
| │ └ Transfer eff. | 93.58 | 94.95 | 93.67 | 93.71 | 94.62 | 96.45 |
| └ Computation scalability | 100.00 | 96.89 | 93.27 | 88.84 | 71.27 | 39.40 |
| ├ IPC scalability | 100.00 | 96.94 | 93.61 | 89.30 | 72.07 | 40.14 |
| ├ Instruction scalability | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| └ Frequency scalability | 100.00 | 99.95 | 99.63 | 99.49 | 98.89 | 98.15 |
| Speedup | 1.00 | 0.97 | 0.92 | 0.87 | 0.71 | 0.40 |
| Average IPC | 1.79 | 1.74 | 1.68 | 1.60 | 1.29 | 0.72 |
| Average frequency [GHz] | 2.09 | 2.09 | 2.08 | 2.08 | 2.06 | 2.05 |
| Instrumentation overhead | < 1% | < 1% | < 1% | < 1% | < 1% | < 1% |

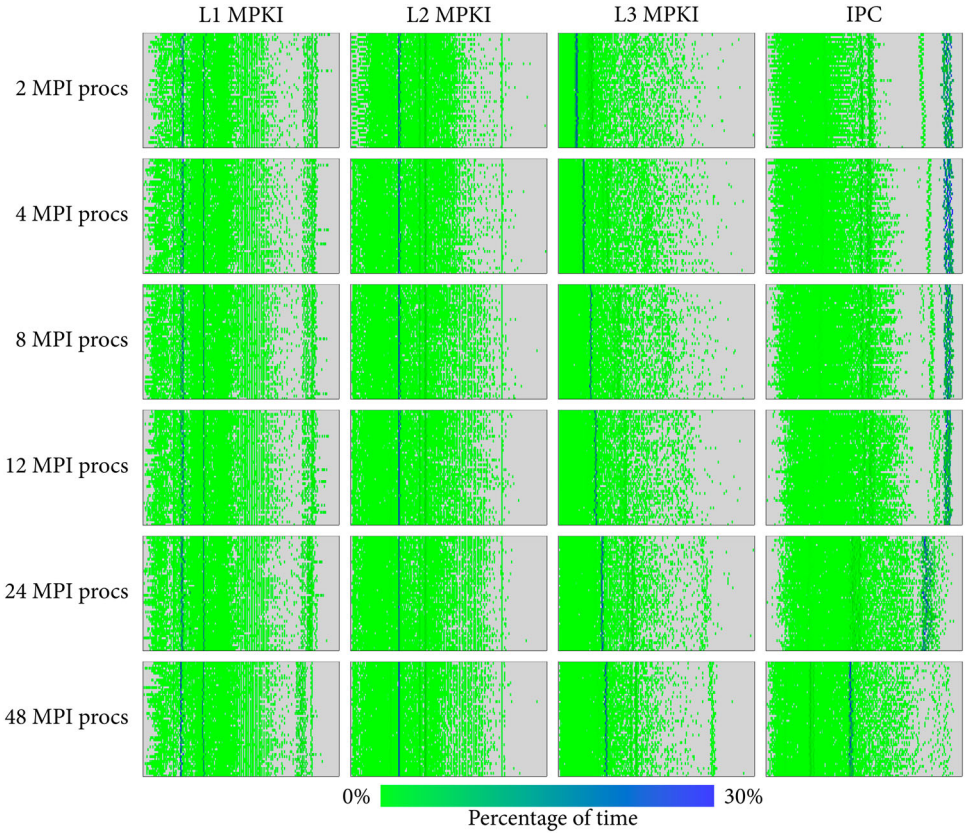



**Figure 19.** CHORUS: Histograms for L1 MPKI, L2 MPKI, L3 MPKI and IPC when increasing number of MPI processes per node.

the $x$-axis. Looking at the first column (L1 MPKI) we can observe that the number of L1 misses per thousand instructions remains constant when increasing the number of MPI processes. This can be explained by the vertical blue lines that are equal for the different number of MPI processes (rows). The second column (L2 MPKI) drives us to the same conclusion about the number of L2 misses.

When we look at the third column (L3 MPKI) we can see that the number of L3 misses increases as we increase the number of MPI process per node. The blue vertical line moves in fact to the right (corresponding to higher values of MPKI). This is the effect of fixing the number of MPI processes while increasing the number of nodes to study the scalability within a node. As L3 is a shared resource per socket, when we run 2 MPI processes per node we are using 24 nodes (48 sockets) but when we run 4 MPI processes per node we are using 12 nodes (24 sockets). Therefore, the total L3 capacity is reduced. The increase in L3 misses is due to the replacement of data done by other processes in the same node.

Although the number of L3 misses increases with the number of MPI process per node, the IPC (right most column of Figure 19) is not affected significantly: we observe in fact similar values of IPC from 2 to 12 MPI processes. Nevertheless, the IPC decreases from 12 to 24 MPI processes and from 24 to 48 MPI processes. As with OpenFOAM, this seems to be a problem of memory bandwidth and we studied this phenomenon measuring the L3 miss density.

In Figure 20, we report on the $y$-axis the number of L3 misses per socket while changing the number of processes per node. As for OpenFOAM in Figure 5, we can notice that the increasing density of L3 misses per socket corresponds to lower and lower values of IPC. This allows us to conclude that more processes on the

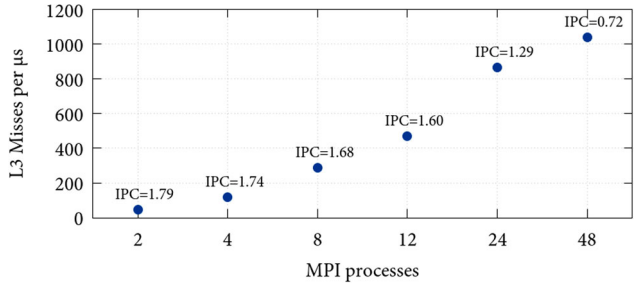



**Figure 20.** CHORUS: L3 Misses per microsecond.

same node competing for the same memory resource make the overall IPC efficiency decreasing.

### 4.3.2. Multi-node Analysis

In Table 8, we find the efficiencies obtained by CHORUS when running in 8–32 nodes (384–1536 MPI processes). We can see that the main factor limiting scalability is the instructions scalability. This metrics points out that there is code replicated or not parallelised. In order to identify the region of the code that is being affected by this metric, we are going to do a clustering of the trace based in number of instructions completed and IPC. The clustering process is done as explained for Alya.

In Figure 21, we can see a timeline of three timesteps of CHORUS with 384 MPI processes visualising the clusters detected by the clustering tool. We can see that the clustering tool has been able to detect the different phases of the application. The most relevant cluster in duration is Cluster 2, followed by Cluster 1. We observe that these two clusters alternate their position within the time step.

In Figure 22, we plot the total number of instructions executed in each cluster when increasing the number of MPI processes. We can see that all the clusters increase the number of instructions executed when increasing the number of MPI processes, it seems that Clusters 1 and 2 are the ones that increase more proportionally.

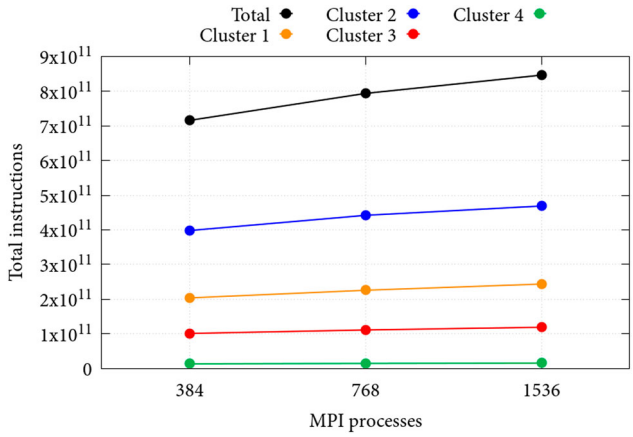



**Figure 22.** CHORUS: Instructions executed per cluster.

In Figure 23, we see the same numbers plotted as efficiencies, we can observe that the trend is similar for all the plots, being Cluster 4 the less affected and Cluster 1 the most affected one. We can conclude that

**Table 8.** Multi-node efficiency metrics of CHORUS.

| | | | |
|---|---|---|---|
| Global efficiency | 90.12 | 83.79 | 76.67 |
| ├ Parallel efficiency | 90.12 | 85.03 | 78.72 |
| │ ├ Load balance | 98.06 | 93.78 | 92.01 |
| │ └ Communication eff. | 91.90 | 90.68 | 85.55 |
| │ ├ Serialization eff. | 96.45 | 97.21 | 91.52 |
| │ └ Transfer eff. | 95.28 | 93.28 | 93.48 |
| └ Computation scalability | 100.00 | 98.54 | 97.39 |
| ├ IPC scalability | 100.00 | 109.06 | 118.71 |
| ├ Instruction scalability | 100.00 | 91.45 | 84.47 |
| └ Frequency scalability | 100.00 | 98.80 | 97.13 |
| Speedup | 1.00 | 1.86 | 3.40 |
| Average IPC | 0.74 | 0.80 | 0.87 |
| Average frequency [GHz] | 2.06 | 2.03 | 2.00 |
| Instrumentation overhead | < 1% | 7.07% | 2.73% |

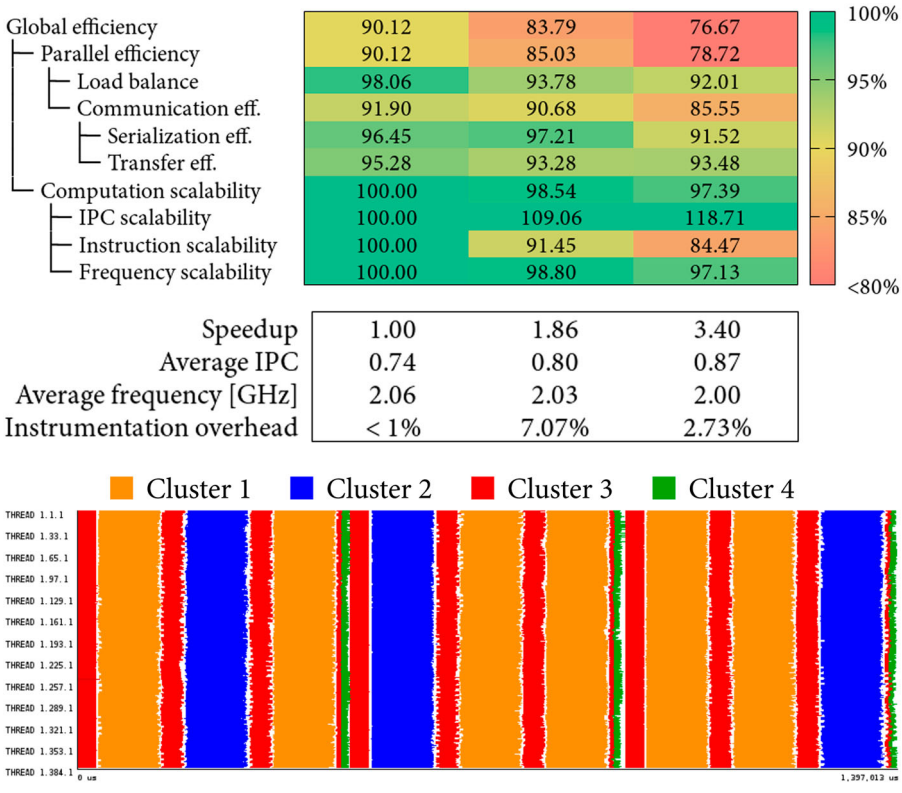



**Figure 21.** CHORUS: Timeline showing clusters of CHORUS.

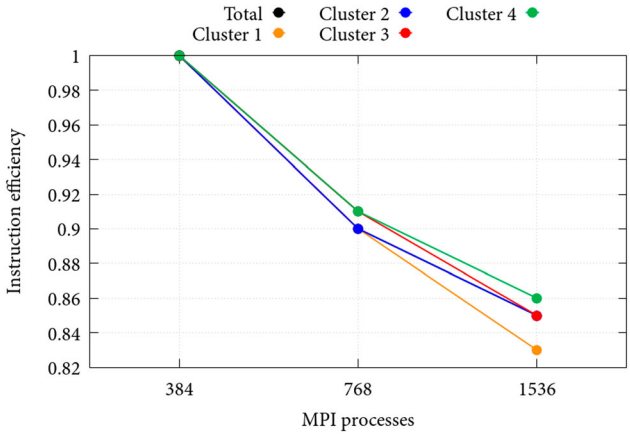



**Figure 23.** CHORUS: Instruction efficiency per cluster.

the whole code is affected by the instruction scalability in a similar way, we can not identify a single cluster responsible for the low value of instruction scalability. Therefore, the recommendation to address this issue is to review the parallelisation approach to detect replication of code in the different MPI processes.

## 5. Conclusions and Discussion

### 5.1. About the Method

In this paper, we utilised a general performance analysis method allowing us to highlight inefficiencies that can result in performance degradation when increasing the parallelism of HPC applications. We applied the method to three complex CFD codes representing three numerical methods: finite volumes, finite elements, and finite differences. Despite the diversity of the codes, we show that with a single method we are able to spot inefficiencies that can be common in different applications. An example is the IPC drop when scaling within a compute node of MareNostrum4: for all the applications that we analysed we verified that the problem is caused by the saturation of memory resources. Also, the paper shows that our method can identify limitations that are specific of the applications, thus producing insightful feedback for CFD code developers.

Beside the effectiveness of the method, we also reveal that it heavily depends on a set of tools that can have limitations. The Clustering tool, for example, while it has been extremely useful for the study of Alya, it was inapplicable to the FOA of OpenFOAM. In this case, we have to find alternative paths to gather insights.

Also, we measured the instrumentation overhead of our tracing tool and we discovered that sometimes it heavily interferes with the measurements. While this is in the nature of the method and the user needs to be aware and be able to read the measurements provided by the tools, we show that results can lead anyhow to insightful conclusions. An example of this is the study of the multi-node efficiencies of OpenFOAM.

### 5.2. About CFD Applications

The analysis of the scalability within a computational node of the three applications lead to the same conclusion: when using a high number of MPI processes per node (i.e. above 12 or 24) the IPC drops because of the amount of data petitions that are issued to memory. Our suggestion to application developers is to improve the pattern of data access and data layout to maximise the reuse of data in the different caches levels.

When analysing the multi-node scalability of OpenFOAM and Alya we concluded for both codes that their main limitation when scaling to a high number of cores will be the number of MPI asynchronous point to point calls. These calls seem innocuous to the programmer because they are asynchronous and allow to overlap communication and computation, but the reality is the overhead of calling MPI cannot be overlapped and is the bottleneck that prevents these applications to scale. To address this issue, we suggest to reduce the number of point to point calls, either by grouping them if the algorithm allows it or by reducing the number of neighbours of the partition.

Alya presents a load balance problem (due to the hybrid mesh used in this analysis) that we suggest to address with a better partition or a dynamic load balancing mechanism. In this case, the suggestion of preparing a better partition of the problem clashes with the previous one, where we suggest to improve the partition to reduce the number of neighbours. These two approaches are usually conflicting, because optimising the number of neighbours can lead to an imbalanced workload. For this reason we would emphasise the use of dynamic mechanisms to attack the load balance issue.

In the case of CHORUS, we detect that the main issue affecting the efficiency is the instruction scalability. This is a clear example of Amdahl's law being applied. Although the parallelisation is very good (90% of efficiency when doubling the number of cores) the

part of the code that is not parallelised ends up being the bottleneck.

## Notes

1. https://cfd.direct/
2. https://www.openfoam.com/
3. https://sourceforge.net/projects/foam-extend
4. http://www.aer.mw.tum.de/en/research-groups/automotive/drivaer/
5. https://openfoam.com/documentation/user-guide/userch1.php#x3-20001
6. https://www.openfoam.com/documentation/user-guide/running-applications-parallel.php
7. https://www.top500.org/lists
8. https://www.openfoam.com/governance/technical-committees.php
9. https://pop-coe.eu

## Acknowledgments

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (TIN2015-65316-P), by the Generalitat de Catalunya (2017-SGR-1414), and by the European POP CoE (GA n. 824080).

## Disclosure Statement

No potential conflict of interest was reported by the author(s).

## Funding

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (TIN2015-65316-P), by the Generalitat de Catalunya (2017-SGR-1414), and by the European POP CoE (GA n. 824080).

## ORCID

*Marta Garcia-Gasulla* http://orcid.org/0000-0003-3682-9905
*Fabio Banchelli* http://orcid.org/0000-0001-9809-0857
*Kilian Peiro* http://orcid.org/0000-0002-9791-2166
*Guillaume Houzeaux* http://orcid.org/0000-0002-2592-1426
*Christian Tenaud* http://orcid.org/0000-0002-2024-485X
*Filippo Mantovani* http://orcid.org/0000-0003-3559-4825

## References

Banchelli, Fabio, Marta Garcia-Gasulla, Guillaume Houzeaux, and Filippo Mantovani. 2020a. "Benchmarking of State-of-the-art HPC Clusters with a Production CFD Code." In *Proceedings of the Platform for Advanced Scientific Computing Conference*, 1–11.

Banchelli, Fabio, Kilian Peiro, Andrea Querol, Guillem Ramirez-Gargallo, Guillem Ramirez-Miranda, Joan Vinyals, Pablo Vizcaino, Marta Garcia-Gasulla, and Filippo Mantovani. 2020b. "Performance Study of HPC Applications on an Arm-based Cluster Using a Generic Efficiency Model." In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 167–174. IEEE.

Bnà, Simone, Ivan Spisso, Giacomo Rossi, and Mark Olesen. 2019. HPC Performance improvements for OpenFOAM linear solvers. Technical report, ESI OpenFOAM Conference, Berlin.

Borrell, Ricard, Damien Dosimont, Marta Garcia-Gasulla, Guillaume Houzeaux, Oriol Lehmkuhl, Vishal Mehta, Herbert Owen, Mariano Vázquez, and Guillermo Oyarzun. 2020. "Heterogeneous CPU/GPU Co-execution of CFD Simulations on the POWER9 Architecture: Application to Airplane Aerodynamics." *Future Generation Computer Systems* 107: 31–48.

Brachet, Marc E., Daniel I. Meiron, Steven A. Orszag, B. G. Nickel, Rudolf H. Morf, and Uriel Frisch. 2006. "Small-scale Structure of the Taylor-Green Vortex." *Journal of Fluid Mechanics* 130 (23): 411–452.

Chen, Goong, Qingang Xiong, Philip J Morris, Eric G Paterson, Alexey Sergeev, and Y Wang. 2014. "OpenFOAM for Computational Fluid Dynamics." *Notices of the AMS* 61 (4): 354–363.

Codina, Ramon. 2001. "Pressure Stability in Fractional Step Finite Element Methods for Incompressible Flows." *Journal of Computational Physics* 170: 112–140.

Culpo, Massimiliano. 2011. Current bottlenecks in the scalability of OpenFOAM on massively parallel clusters. *PRACE white paper to appear on http://www.praceri.eu*.

Daru, Virginie, and Christian Tenaud. 2004. "High Order One-step Monotonicity-Preserving Schemes for Unsteady Compressible Flow Calculations." *Journal of Computational Physics* 193 (2): 563–594.

Garcia, Marta, Jesus Labarta, and Julita Corbalan. 2014. "Hints to Improve Automatic Load Balancing with Lewi for Hybrid Applications." *Journal of Parallel and Distributed Computing* 74 (9): 2781–2794.

Garcia-Gasulla, Marta, Filippo Mantovani, Marc Josep-Fabrego, Beatriz Eguzkitza, and Guillaume Houzeaux. 2019. "Runtime Mechanisms to Survive New HPC Architectures: A Use Case in Human Respiratory Simulations." *The International Journal of High Performance Computing Applications* 34 (1): 42–56.

Garcia-Gasulla, Marta, Guillaume Houzeaux, Roger Ferrer, Artigues Antoni, Victor López, Jesus Labarta, and Mariano Vázquez. 2019. "MPI+X: Task-based Parallelization and Dynamic Load Balance of Finite Element Assembly." *Journal International Journal of Computational Fluid Dynamics*33 (3): 115–136.

Houzeaux, Guillaume, Ricard Borrell, Yvan Fournier, Marta Garcia-Gasulla, Jens Henrik Göbbert, Elie Hachem, Vishal Mehta, Youssef Mesri, Herbert Owen, and Mariano Vázquez.

2018. "High-performance computing: Dos and don'ts." In Adela Ionescu, editor, *Computational Fluid Dynamics – Basic Instruments and Applications in Science*, chapter 01. InTech: Rijeka.

Jasak, Hrvoje. 2011. HPC Deployment of OpenFOAM in an Industrial Setting Overview Objective – Review the state and prospects for massive parallelisation in CFD codes, with review of implementation in OpenFOAM Topics. Technical report, PRACE, Stockholm.

Jasak, Hrvoje. 2012. "Handling Parallelisation in Openfoam." In *Cyprus Advanced HPC Workshop*, 101, 3.

Löhner, Rainald, Fernando Mut, Juan Raul Cebral, Romain Aubry, and Guillaume Houzeaux. 2011. "Deflated Preconditioned Conjugate Gradient Solvers for the Pressure-poisson Equation: Extensions and Improvements." *International Journal for Numerical Methods in Engineering* 87: 2–14.

Lehmkuhl, Oriol, Guillaume Houzeaux, Herbert Owen, Georgios Chrysokentis, and Ivette Rodríguez. 2019. "A Low-dissipation Finite Element Scheme for Scale Resolving Simulations of Turbulent Flows." *Journal of Computational Physics* 390: 51–65.

Mantovani, Filippo, Marta Garcia-Gasulla, José Gracia, Esteban Stafford, Fabio Banchelli, Marc Josep-Fabrego, Joel Criado-Ledesma, and Mathias Nachtmann. 2020. "Performance and Energy Consumption of HPC Workloads on a Cluster Based on Arm ThunderX2 CPU." *Future Generation Computer Systems*.

Morgan, Eric Lease. 2000. "The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary." *Information Technology and Libraries* 19 (2): 105.

Pillet, Vincent, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver. 1995. "Paraver: A Tool to Visualize and Analyze Parallel Code." In *Proceedings of WoTUG-18: Transputer and Occam Developments*, 44, 1, 17–31.

Pirozzoli, Sergio. 2011. "Numerical Methods for High-speed Flows." *Annual Review of Fluid Mechanics* 43 (1): 163–194.

Servat, Harald, German Llort, Kevin Huck, Judit Gimenez, and Jesus Labarta. 2013. "Framework for a Productive Performance Optimization." *Parallel Computing* 39 (8): 336–353.

Soto, Orlando, Rainald Löhner, and Fernando Camelli. 2003. "A Linelet Preconditioner for Incompressible Flow Solvers." *Journals International Journal of Numerical Methods for Heat & Fluid Flow* 13 (1): 133–147.

Vreman, A. W.. 2004. "An Eddy-viscosity Subgrid-scale Model for Turbulent Shear Flow: Algebraic Theory and Applications." *Physics of Fluids* 16 (10): 3670–3681.

Vázquez, Mariano, Guillaume Houzeaux, Seid Koric, Antoni Artigues, Jazmin Aguado-Sierra, Ruth Arís, Daniel Mira et al. 2016. "Alya: Multiphysics Engineering Simulation Toward Exascale." *Journal of Computational Science* 14: 15–27.

Wagner, Michael, Stephan Mohr, Judit Giménez, and Jesús Labarta. 2017. "A Structured Approach to Performance Analysis." In *International Workshop on Parallel Tools for High Performance Computing*, 1–15. Cham: Springer.

Wang, Z. J., Krzysztof Fidkowski, Rémi Abgrall, Francesco Bassi, Doru Caraeni, Andrew Cary, Herman Deconinck, Ralf Hartmann, Koen Hillewaert, H. T. Huynh, Norbert Kroll, Georg May, Per-Olof Persson, Bram van Leer, and Miguel Visbal. 2013. "High-Order CFD Methods: Current Status and Perspective." *International Journal For Numerical Methods In Fluids* 72 (8): 811–845.