Runtime Mechanisms to Survive New HPC Architectures: A Use-Case in Human Respiratory Simulations

Marta Garcia-Gasulla*
Barcelona Supercomputing Center
marta.garcia@bsc.es

Filippo Mantovani
Barcelona Supercomputing Center
filippo.mantovani@bsc.es

Marc Josep-Fabrego
Barcelona Supercomputing Center
marc.josep@bsc.es

Beatriz Eguzkitza
Barcelona Supercomputing Center
beatriz.eguzkitza@bsc.es

Guillaume Houzeaux
Barcelona Supercomputing Center
guillaume.houzeaux@bsc.es

April 15, 2019

Abstract

Computational Fluid and Particle Dynamics (CFPD) simulations are of paramount importance for studying and improving drug effectiveness. Computational requirements of CFPD codes demand high-performance computing (HPC) resources. For these reasons we introduce and evaluate in this paper system software techniques for improving performance and tolerate load imbalance on a state-of-the-art production CFPD code. We demonstrate benefits of these techniques on Intel-, IBM-, and Arm-based HPC technologies ranked in the Top500 supercomputers, showing the importance of using mechanisms applied at runtime to improve the performance independently of the underlying architecture. We run a real CFPD simulation of particle tracking on the human respiratory system, showing performance improvements of up to 2×, across different architectures, while applying runtime techniques and keeping constant the computational resources.

I. Introduction and Related Work

Aerosolized delivery of drugs to the lungs is used to treat some respiratory diseases, such as asthma, chronic obstructive pulmonary disease, cystic fibrosis, and pulmonary infections. However, it is well known that a significant fraction of the inhaled medicine is lost in the extrathoracic airways.

During the last decade an exponential growth in the application of Computational Fluid-Particle Dynamics (CFPD) methods in this area has been observed, [34, 12]. CFPD simulations can be used to help scientists to reproduce and reduce the lost aerosol fraction and improve the overall performance of the drug [18, 13].

Validated CFPD methods offer a powerful tool to predict the airflow and localized deposition of drug particles in the respiratory airways, to improve our understanding of the flow and aerosol dynamics as well as optimize inhaler therapies. Moreover, a model of lung inflammation produced by pollutant particle inhalation is critical to predicting therapeutic responses related to the chronic obstructive pulmonary disease. Deposition maps generated via CFPD simulations and their integration into clinical practice is an essential point to develop such a model. The understanding of this kind of dynamics can give hope for improving the living conditions of affected patients and reducing the costs associated with hospitalizations.

Accurate and efficient numerical simulations tracking the particles entering the respiratory system pose a challenge due to the complexities associated with the airway geometry, the flow dynamics, and aerosol physics. Due to the complexity and computational requirements of the models simulating such phenomena, the use of large-scale computational resources paired with highly optimized simulation codes is of paramount importance.

A successful study of aerosol dynamic strongly depends on two challenges: on the one hand, we have the physics of the problem, that needs to be translated into more and more precise models increasing the accuracy of the simulations; on the other hand, we have the computational part of the problem, that requires the development of more efficient codes able to exploit massively parallel supercomputers reducing the simulation time.

Looking closer at the computational challenge, we can note that, not only the number of computational resources available to run CFPD simulations is growing but also the diversity of hardware where simulations are performed is increasing (e.g., special purpose architectures and emerging

*Corresponding author

technologies). As proven by previous studies on other fluid dynamics codes [4, 6], it is essential to be able to efficiently exploit state-of-the-art architectures maintaining at the same time correctness and portability of the code.

Given these observations, we consider in this paper Alya [35], a simulation code for high-performance computational mechanics developed at the Barcelona Supercomputing Center (BSC). Alya is part of the Unified European Applications Benchmark Suite (UEABS), a selection of 12 codes scalable, portable, and relevant for the scientific community. Alya is also currently adopted by industrial players, such as Iberdrola and Vortex for wind farms simulations and Medtronic for medical device and bio-mechanics experiments.

To deep dive into the computational challenge, we test Alya on three state-of-the-art HPC technologies ranked in the Top500 list of November 2018: *i)* MareNostrum4, based on the latest Intel Skylake CPU; *ii)* IBM Power9, powered by the same IBM architecture driving Summit, the fastest supercomputer according to the Top500 list of November 2018 [33]; *iii)* Dibona, a production-ready cluster based on Marvell ThunderX2 CPU, the same Arm-based architecture sitting inside the Astra supercomputer, the first Arm-based supercomputer entering the Top500 list. A performance evaluation of Dibona can be found in [2].

The efficient execution and portability of codes like Alya on modern HPC architecture such as MareNostrum4, Power9, and Dibona are of fundamental importance. There have been previous evaluations of relevant scientific applications on emerging HPC platforms targeting both performance improvement [1, 30, 19, 28, 7] and energy reduction [8, 24], however, we want to stress the fact that we study in our paper a production code evaluating a real use case on state-of-the-art HPC technologies ranked in the Top500 list and targeting code performance and portability.

As already mentioned, the complexity and the size of production CFPD codes do not allow machine dependent fine-tuning of the applications for each case study and each platform. For this reason we propose here two software techniques: *multidependences*, for improving performance avoiding atomic operations while running on a shared memory system [36] and *DLB*, a dynamic load balancing library able to solve load imbalance within a parallel system by redistributing the computational resources in order to minimize the inefficiency [15, 9].

The problem of homogeneously distribute the workload among computational nodes can be tackled using static or dynamic strategies and it is well studied in the literature. The static workload balance is handled in our case by the graph and mesh partitioning library Metis [22]. Some examples of dynamic strategies to handle load balance by redistributing the workload at runtime are PAMPA, a utility for redistributing mesh based arrays [23] and Adaptive MPI (AMPI), an object-oriented parallel programming language that migrates objects to dynamically achieve load balance [3]. However, we chose DLB for this study for two reasons: we avoid the costly reallocation of data in favour of the reallocation of idle resources. Also, DLB is a library that plugs directly into OpenMP and MPI, both well known and community supported industrial standards, while the previous approaches require specific programming environments.

The idea is to demonstrate how tools such as the multidependences and DLB, deployed at the level of system software, require minimal or even no changes in the source code, boosting the performance without harming portability nor the semantic of the source code. On the longer term, we believe tools like the multidependences and DLB will allow programmers to survive to the waves of architectural novelties without drowning into fine-tuning optimizations of the code.

Within the code of Alya, we already tested runtime mechanisms to mitigate load imbalance penalties on an Intel-based HPC cluster [16]. This work is an extension of our previous work [17]. While the underlying runtime techniques are substantially the same in the two papers, we include in this extension a more extensive evaluation, including state-of-the-art HPC CPU architectures. We replaced the performance measured on the Arm-based SoC, Cavium ThunderX, with ThunderX2, the latest CPU by Marvell[1]. We also complement our experimental data with measurements on the latest IBM Power9 CPU.

The main contributions of this paper are: *i)* we provide the evaluation of a production use case of real biological HPC simulation on three HPC clusters, based on different architectures, Intel x86, IBM Power9, and Marvell Arm-based ThunderX2. *ii)* we introduce programming models techniques applied to a production simulation that show benefits on current and emerging HPC architectures.

The remaining of this document is organized as follows: in Section II we introduce the architectural challenge including details of the three HPC clusters on which we performed our experiments. Section III is dedicated to the scientific challenge: we introduce here the Alya simulation infrastructure as well as its computational profiling. Section IV introduces the computational challenge of our work, i.e., the key ideas of the runtime techniques that we evaluate. In Section V we briefly introduce the software configurations used for our tests and the results of our evaluation. In Section VI we discuss the lessons learned and the conclusions of our work.

II. The Architectural Challenge

The tests presented in this work have been executed on three state-of-the-art production clusters based on different

[1]Cavium has been recently acquired by Marvell, so that we will refer to its latest CPU model in the remaining of the text as Marvell ThunderX2

HPC architectures.

MareNostrum4 is a supercomputer based on Intel Xeon Platinum processors, Lenovo SD530 Compute Racks, a Linux Operating System and an Intel Omni-Path interconnection. Its general purpose partition has a peak performance of 11.15 Petaflops, 384.75 TB of main memory spread over 3456 nodes. Each node houses 2× Intel Xeon Platinum 8160 with 24 cores at 2.1 GHz, 216 nodes feature 12 × 32 GB DDR4-2667 DIMMS (8 GB/core), while 3240 nodes are equipped with 12 × 8 GB DDR4-2667 DIMMS (2 GB/core). MareNostrum4 is the PRACE Tier-0 supercomputer hosted at the Barcelona Supercomputing Center (BSC) in Spain and ranked 25 in the Top500 list of November 2018 [33].

Power9 is also hosted at Barcelona Supercomputing Center. This cluster is based on IBM Power9 8335-GTG processors with 20 cores each CPU operating at 3 GHz. Each compute node contains two CPUs, plus four GPUs NVIDIA V100 with 16 GB HBM2. Each compute node is equipped with 512 GB of main memory distributed in 16 DIMMS of 32 GB operating at 2666 MHz. Nodes are interconnected with an Infiniband Mellanox EDR network, and the operating system is Red Hat Enterprise Linux Server 7.4. The Power9 cluster has been included in our study because its computational elements are architecturally identical to the ones of the Summit supercomputer, ranked first in the Top500 of November 2018 [33]. It must be clarified that we do not consider in our evaluation the accelerator part composed by the GP-GPUs.

Dibona is an Arm-based HPC cluster, designed and deployed by ATOS/Bull within the Mont-Blanc 3 project and announced as a product during 2018 [25]. Each compute node is powered by two Marvell's ThunderX2 CPUs with 32 cores each operating at 2.0 GHz. The main memory on each node is 256 GB of DDR4 running at 2667 MHz. Nodes are interconnected with Infiniband Mellanox EDR network. The Dibona cluster has been considered for our study because it features the same CPU technology that composes the Astra supercomputer, the first Arm-based system ranked 204 in the Top500 list of November 2018 [31].

Platform	Compiler	MPI version
MareNostrum4	GCC 8.1.0	OpenMPI 3.0.0
Power9	GCC 8.1.0	OpenMPI 3.0.0
Dibona	GCC 7.2.1	OpenMPI 2.0.2.14

Table 1: *Software environment employed on different platforms*

In all platforms we have used GNU as Fortran compiler and OpenMPI as MPI library, the exact version used in each platform can be found in table 1. We use the same version of the DLB library (2.0.2) and the OmpSs programming model (17.12.1) composed by the *Mercurium* 2.1.0 source to source compiler and the *Nanox* 15a compiler.

III. The Scientific Challenge

In this section, we introduce the biological use-case on which we validate our runtime techniques. The effect of aerosol therapies depends on the dose deposited beyond the oropharyngeal region as well as its distribution in the lungs. The efficiency of the treatment is affected by the amount of drug lost at the airway as well as its deposition on regions that are not affected by the pathology. At the same time, factors as the size of the aerosol particles, breathing conditions, the geometry of the patient, among others, are decisive in the resulting deposition maps of the lung. All these parameters must be considered in clinical practice to personalize therapies involving treatments with aerosol.

i. Particle tracking in the respiratory system

In this work, we simulate the transport of particles injected in an unsteady flow in the large human airways during a rapid inhalation. The use of massive computational resources allows capturing all the spatial and temporal scales of the flow necessaries to reduce the lost aerosol fraction and improve the effectiveness of treatments.

The CFPD simulation is performed on a complex subject-specific geometry extended from the face to the 7th branch generation of the bronchopulmonary tree and a hemisphere of the subject's face exterior [5]. In particular, the mesh is hybrid and composed of 17.7 million elements, including: *prisms*, to resolve the boundary layer accurately; *tetrahedra*, in the core flow; *pyramids* to enable the transition from prism quadrilateral faces to tetrahedra. Figure 1 shows some details of the mesh and in particular the prisms in the boundary layer.

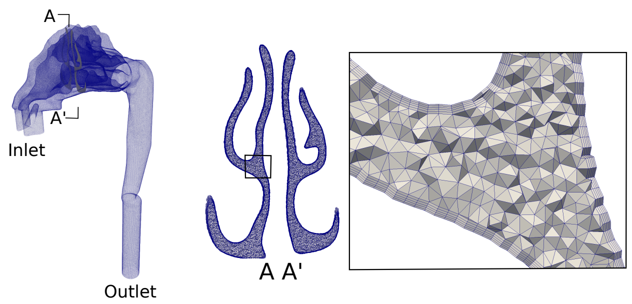

Figure 1: *Detail of the mesh representing the human respiratory system.*

As introduced in Section I, the application used in this document is the high-performance computational mechanics code, Alya [35]. Alya is parallelized using MPI and OpenMP, but, production runs are usually performed using a MPI-only parallelization approach. The mesh partition in subdomains to distribte the work among the different MPI processes is done using Metis [22].

CFPD implies the solution of the incompressible fluid flow obtained through Navier-Stokes equations as well as the Lagrangian particle tracking.

The Navier-Stokes equations express Newton's second law for a continuous fluid medium, whose unknowns are the velocity $\boldsymbol{u}_f$ and the pressure p_f of the fluid. Two physical properties are involved, namely μ_f be the viscosity, and ρ_f the density. At the continuous level, the problem is stated as follows: find the velocity $\boldsymbol{u}_f$ and pressure p_f in a domain Ω such that they satisfy in a given time interval

$$\begin{aligned} \rho_f \frac{\partial \boldsymbol{u}_f}{\partial t} + \rho_f(\boldsymbol{u}_f \cdot \nabla)\boldsymbol{u}_f - \mu_f \Delta \boldsymbol{u}_f + \nabla p_f &= \mathbf{0}, \quad (1) \\ \nabla \cdot \boldsymbol{u}_f &= 0, \quad (2) \end{aligned}$$

together with initial and boundary conditions. The numerical model of the Navier-Stokes solver implemented in Alya code and used in this study is a stabilized finite-element method, based on the Variational MultiScale (VMS) method [21]. As far as time integration is concerned, a second order backward differentiation formula (BDF) scheme is considered.

Particles are transported solving Newton's second law, and by applying a series of forces. Let m_p be the particle mass, d_p its diameter, ρ_p its density, $\boldsymbol{a}_p$ its acceleration. According to Newton's second law, if $\boldsymbol{F}_p$ is the total force acting on the particle, then

$$\boldsymbol{F}_p = m_p \boldsymbol{a}_p. \quad (3)$$

The different forces considered in this work are the drag, gravity and buoyancy forces, given by

$$\begin{aligned} \boldsymbol{F}_g &= m_p \boldsymbol{g}, \quad (4) \\ \boldsymbol{F}_b &= -m_p \boldsymbol{g} \rho_f / \rho_p, \quad (5) \\ \boldsymbol{F}_D &= (\pi/8)\mu_f d_p C_d \mathrm{Re}_p (\boldsymbol{u}_f - \boldsymbol{u}_p), \quad (6) \end{aligned}$$

respectively, where Re and C_D are the particle Reynolds number and drag coefficient given by Ganser's formula [14], respectively:

$$\begin{aligned} \mathrm{Re}_p &= \rho_f d_p |\boldsymbol{u}_f - \boldsymbol{u}_p| / \mu_f, \quad (7) \\ C_D &= \frac{24}{\mathrm{Re}_p}[1 + 0.1118(\mathrm{Re}_p)^{0.65657}] + \frac{0.4305}{1 + \frac{3305}{Re_p}}. \quad (8) \end{aligned}$$

By considering only gravity, buoyancy and drag, the previous model assumes that particles have a sufficiently low inertia to neglect history effects, meaning that particle velocity tends to the fluid velocity almost instantaneously. Lift force has also been neglected, assuming particles are sufficiently small not to be sensitive to local shear. Finally, Ganser's formula is a correlation to fit experimental data of non-spherical particles, and valid up to particle Reynolds number sufficiently high to be considered valid in our case. For the sake of simplicity, the sphericity of unity has been assumed in Equation 7, thus making the formula valid only for spherical particles.

The time integration of particle motion is based on Newmark's method. From time step n to time step $n+1$, the following sequence is solved to update the particle dynamics, acceleration $\boldsymbol{a}_p$, velocity $\boldsymbol{u}_p$ and position $\boldsymbol{x}_p$:

$$\boldsymbol{a}_p^{n+1} = \boldsymbol{F}_p^{n+1}/m_p, \quad (9)$$

$$\boldsymbol{u}_p^{n+1} = \boldsymbol{u}_p^n + [(1-\gamma)\boldsymbol{a}_p^n + \gamma \boldsymbol{a}_p^{n+1}]\delta t_p, \quad (10)$$

$$\boldsymbol{x}_p^{n+1} = \boldsymbol{x}_p^n + \boldsymbol{u}_p^n \delta t_p + \frac{\delta t_p^2}{2}[(1-2\beta)\boldsymbol{a}_p^n + 2\beta \boldsymbol{a}_p^{n+1}], \quad (11)$$

where δt_p is the particle time step, and β and γ are constants to control the stability and accuracy of the scheme. We note that because of drag, the acceleration is position and velocity dependent. To increase robustness, a Newton-Raphson is also considered to solve the non-linearity in velocity. Once the Newton-Raphson scheme is converged, the position is then updated with Equation 11.

The fluid time step used in this paper is 10^{-4} seconds, providing sufficient accuracy to the fluid solution. From one fluid time step to the next one, an adaptive time step based on an error estimate is used to control the accuracy of particle trajectories. In practice, the particle adaptive time step leads to δt_p in the range of $[10^{-6}, 10^{-4}]$, depending on the local conditions of the airflow.

IV. The Computational Challenge

In this section, we analyze the different parts of a time step of Alya to identify the challenges and quantify the performance of the different parts that compose the respiratory simulation. We also introduce the Multidependences and a Dynamic Load Balancing library, the runtime techniques that we apply to Alya and evaluate on the platforms introduced in Section II.

i. Profile and performance analysis

To identify the different parts of a simulation and how they can be improved, we use a trace of the Alya simulation gathered on four nodes of the MarenNostrum4 supercomputer (introduced in Section II).

We use Extrae [32] to obtain a performance trace and then Paraver [29] to visualize it. In this simulation, $4 \cdot 10^5$ particles where injected in the respiratory system during the first time step.

Figure 2 shows the timeline of one simulation step: on the y-axis are represented the different OpenMP threads, grouped by process, while on the x-axis we plot the execution time. The different colors identify the different parallel regions. White color corresponds to MPI communication; brown is matrix assembly of the Navier-Stokes equations; yellow and green are algebraic solvers to compute the momentum and continuity of the fluid; pink represents the

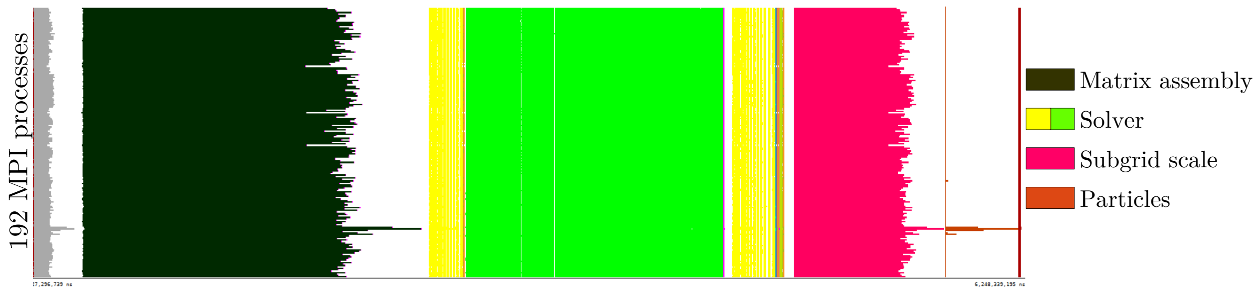

Figure 2: *Trace of respiratory simulation with 192 MPI processes gathered on 4 nodes of MareNostrum4*

computation of the velocity subgrid scale vector (SGS). Finally, once the velocity of the fluid has been computed, the transport of the particles is computed: this is shown in orange on the right most of the trace.

From the trace, we can observe that the active work performed by each process (i.e., each colored part in the trace) within the same phase is not homogeneous: we call this phenomenon *load imbalance*. Load imbalance is one of the main sources of inefficiency in this execution and it is present in different phases and with a different pattern in each phase. We define L_n the load balance among n MPI processes within each phase as:

$$L_n = \frac{\sum_{i=1}^{n} t_i}{n \cdot \max_{i=1}^{n} t_i} \qquad (12)$$

where t_i is the elapsed time by process i during that phase.

Using this metric, $L_n = 1$ corresponds to a perfectly balanced execution on n MPI processes, while $L_n = 0.5$ is an execution that it is losing 50% of the computational resources due to load imbalance.

Phase	L_{96}	**% Time**	**I**
Matrix assembly	0.66	40.84%	0.09
Solver1	0.90	16.13%	0.03
Solver2	0.89	4.20%	0.12
SGS	0.61	21.43%	0.07
Particles	0.02	3.37%	0.05

Table 2: *Load balance, percentage of the total execution time and arithmetic intensity for different phases of the respiratory simulation executed with 96 MPI processes.*

In Table 2, the first column shows the load balance measured in each phase of the execution; the second column shows the percentage of execution time spent by each phase, within a time step.

We can observe low values of load balance in the matrix assembly and the subgrid-scale (SGS) phases, both $L_{96} \sim 0.6$. But the lowest value of load balance appears in the computation of particles: $L_{96} = 0.02$ means that globally 98% of the time of that phase is wasted. For a complete analysis of load unbalance in Alya see [16].

The rightmost column of Table 2 shows the *arithmetic intensity I*, a metric common in the HPC community, that characterizes the application with no (or very little) dependence on the hardware platform where it will run. The arithmetic intensity I is defined as $I = \frac{W}{D}$, where W is the computational workload, so the number of floating point operations executed by the application while D is the application data set, so the bytes that the application exchanges with the main memory. We measured W as the number of double precision operations reported by the CPU counters. Also, we measured $D = (L + S) \cdot 8$, where L is the number of load instructions and S is the number of store instructions on double precision data values (8 bytes) measured reading hardware counters of the CPU. We show in Table 2 the values of I, arithmetic intensities, of the different phases of Alya, ranging from 12 bytes transferred per hundred floating point operations in the first type of solver to 3 bytes transferred per hundred floating point operations in the second solver of Alya.

It is important to note that the percentage of time spent in the particle phase is directly proportional to the number of particles injected into the system. In the simulation we show in Figure 2 we are injecting $O(10^5)$ particles, but in production simulations, we can inject up to $O(10^7)$ particles or inject particles several times during the simulation (e.g., when simulating the inhalation of pollutants when breathing). This, of course, affects the load balance as well: increasing the number of particles in the system, translates in fact into higher and higher inefficiency.

Moreover, the high load imbalance of the particles computation is inherent to the problem because the particles are always introduced in the system through the nasal orifice. Therefore, at the injection they are located in one or few MPI subdomains, and as the simulation advances the particles get distributed among the different MPI subdomains, changing the load balance between MPI processes.

To avoid the inefficient use of resources during the computation of the particles phase, Alya offers the possibility of running a coupled execution. A coupled execution runs two instances of Alya within the same MPI communicator, one of them solving the fluid and the other one the transport of particles.

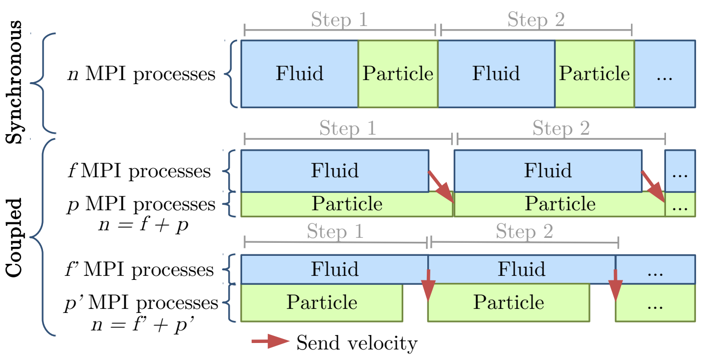

Figure 3: *Execution modes for CFPD simulations with Alya*

In Figure 3 we can see the different options to run the same simulation. In the top, we can see the synchronous execution, where all the processes first solve the velocity of the fluid and then the transport of particles. In the bottom the coupled execution is represented; in this case, some MPI processes solve the velocity of the fluid and send them to the MPI processes that are solving the transport of particles.

When using the coupled simulation, the user can decide how many processes are dedicated to solving the fluid part of the problem (f MPI processes) and how many processes solve the particles part of the problem (p MPI processes). The load of each MPI process depends on the number of particles injected and the load distribution between processes solving the fluid and the particle problems (i.e., the distribution of f and p). This is depicted at the bottom of Figure 3, where $f > f'$ and $p < p'$. The conclusion is that depending on the decision taken by the user the performance of the simulation can vary. The optimum distribution of MPI processes depends on the parameters of the simulation, the architecture and the number of resources used.

To alleviate this decision we propose to use a dynamic load balancing mechanism to adapt the resources given to the different codes. This mechanism is explained in detail in Section iii.

ii. Multidependences

As we have seen in the previous section, one of the main computational tasks in a CFDP code is the algebraic system assembly, depicted in Figure 2 as *Matrix assembly*. This phase consists of a loop over the elements of the mesh (see Figure 1): for each element, a set of local operations is performed to assemble the local matrices. More details can be found in [20] and [16].

From the parallelism point of view this phase has two important characteristics:

1. On the one hand, the algebraic system assembly over the elements is performed locally to each MPI process. Thus, no MPI communication is involved during this operation. This characteristic makes the assembly phase well suited to apply shared memory parallelism within each element.

2. On the other hand, the algebraic system assembly over the elements for generating the local matrices consists of a reduction operation over a mesh that is local to each element but presents irregular connectivity. This translates into indirect and irregular accesses to the data structure storing the local mesh that can involve two local elements that share a node. When this happens, it can mean that two separate OpenMP threads, processing two independent local elements that share a node, could update the same position of the matrix, resulting in a race condition. To avoid the race condition between two threads updating the same position of the matrix we evaluate in this section different parallelization alternatives.

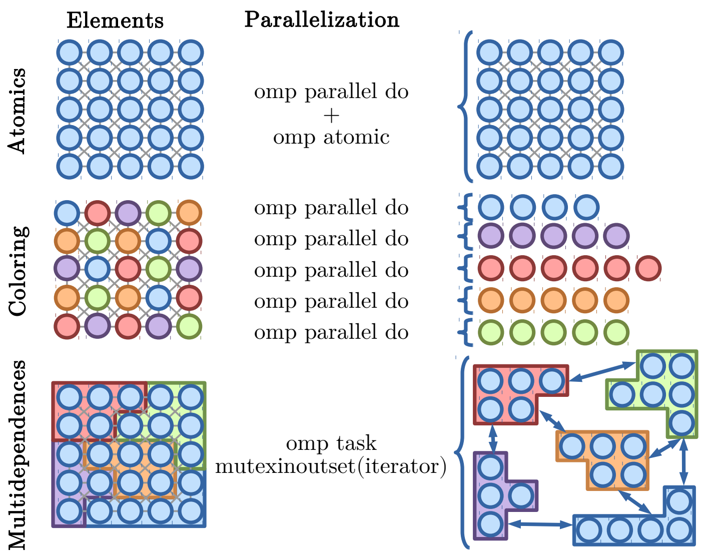

Figure 4: *Parallelization approaches for the matrix assembly*

In Figure 4 we can see the three approaches to parallelize the matrix assembly that we have considered. The straightforward approach would be to protect the update of the local matrix with an `omp atomic` pragma. This approach has a negative impact on the performance because when computing each element an atomic operation must be performed whether or not there is a conflict in the update of the matrix. The pseudo-code of this approach can be seen in Algorithm 1.

Algorithm 1 Parallelization of matrix assembly with *atomic*

```
!$OMP PARALLEL DO &
!$OMP PRIVATE (...) &
!$OMP SHARED  (...)
for elements $e$ do
    Compute element matrix and RHS: $\mathbf{A}^e$, $\mathbf{b}^e$
    !$OMP ATOMIC
    Assemble matrix $\mathbf{A}^e$ into $\mathbf{A}$
    !$OMP ATOMIC
    Assemble RHS $\mathbf{b}^e$ into $\mathbf{b}$
end for
```

To avoid the use of an atomic operation, a coloring technique can be used as explained e.g., in [11]. The coloring technique, as can be seen in Figure 4, consists in assigning a color to each element. Elements that share a node can not have the same color. Once each element is assigned a color, the elements of the same color can be computed in parallel (e.g., in a parallel loop without an atomic operation to avoid the race condition). The main drawback of the coloring approach is that it hurts spatial locality since contiguous elements are not computed by the same thread. Algorithm 2 shows the code corresponding to the parallelization with coloring.

Algorithm 2 Parallelization of matrix assembly with *coloring* elements approach

```
Partition local mesh in $nsubd$ subdomains using a coloring strategy
for $isubd = 1, nsubd$ do
    !$OMP PARALLEL DO &
    !$OMP PRIVATE (...) &
    !$OMP SHARED  (...)
    for elements $e$ in $isubd$ do
        Compute elem. matrix and RHS: $\mathbf{A}^e$, $\mathbf{b}^e$
        Assemble matrix $\mathbf{A}^e$ into $\mathbf{A}$
        Assemble RHS $\mathbf{b}^e$ into $\mathbf{b}$
    end for
    !$OMP END PARALLEL DO
end for
```

The third approach that we consider is *multidependences*. For this approach we divide the domain assigned to each MPI process into subdomains. To this end, we use Metis [22] since is already used to partition the mesh among MPI processes. We map each subdomain into an OpenMP task. Knowing that two subdomains are adjacent if they share at least one node, we can use the information about the adjacency of subdomains provided by Metis to know which OpenMP tasks cannot be executed at the same time. Therefore those subdomains that are adjacent, i.e., that share at least one node, will be processed sequentially, those that are not adjacent, i.e., that do not share nodes, can be processed in parallel.

Algorithm 3 Parallelization of matrix assembly using *multidependences* betweent element chunks

```
Partition local mesh in $nsubd$ subdomains
Store connectivity graph of subdomains in $subd$
for $isubd = 1, nsubd$ do
    $nneig = SIZE(subd(isubd)\%lneig)$
    !$OMP TASK &
    !$omp task depend( iterator(i=1:nneig), &
          mutexinoutset: subd(subd%lneig(i)))&
    !$OMP PRIORITY    (nneig) &
    !$OMP PRIVATE     (...) &
    !$OMP SHARED      (subd, ...)
    for Elements $e$ in $isubd$ do
        Compute elem. matrix and RHS: $\mathbf{A}^e$, $\mathbf{b}^e$
        Assemble matrix $\mathbf{A}^e$ into $\mathbf{A}$
        Assemble RHS $\mathbf{b}^e$ into $\mathbf{b}$
    end for
    !$OMP END TASK
end for
!$OMP TASKWAIT
```

In the multidependences version, we used two new features that have recently been introduced in the OpenMP standard in the 5.0 release [27]. On the one hand, the `iterators` to define a list of dependences, this feature allow defining a dynamic number of dependences between tasks that are computed at execution time. On the other hand, a new kind of relationship between tasks: `mutexinoutset`. This relationship implies that two tasks cannot be executed at the same time, but the execution order between them is not relevant. This relationship can express "incompatibility" between tasks. In Algorithm 3 we can see the implementation in pseudo-code of the multidependences version.

These two new features of the standard were first implemented in the OmpSs programming model [10, 26]. The OmpSs programming model is a forerunner of OpenMP where extensions to the OpenMP model are implemented and tested, and some of them are finally added to the standard [36]. We take advantage of these early implementations to test these two new features in a real code.

An important added value of the multidependences version is that its implementation does not require significant changes in the code and leaves the parallelization quite clean and straightforward. For large production code, such as Alya, this is highly desirable.

With the parallelization introduced with multidependences, we do not need `omp atomic` and the elements that are consecutive in memory are executed by the same thread, so a certain degree of spatial locality is preserved.

iii. Dynamic Load Balancing (DLB)

Dynamic Load Balancing (DLB) is a library that aims to improve the load balance of hybrid applications [9]. In an application leveraging multi-level parallelism, e.g., MPI+OpenMP, DLB uses the second level of parallelism (usually OpenMP) to improve the load balance at the MPI level and achieve so better overall performance. The load balancing library acts at runtime, reacting to the load imbalance whenever it is appearing. It has been proved beneficial in several HPC codes [15].

The DLB library is transparent to the application; therefore it does not require to modify the source code. It uses standard mechanisms provided by the programming models, such as PMPI interception from MPI and OpenMP call `omp_set_num_threads()`.

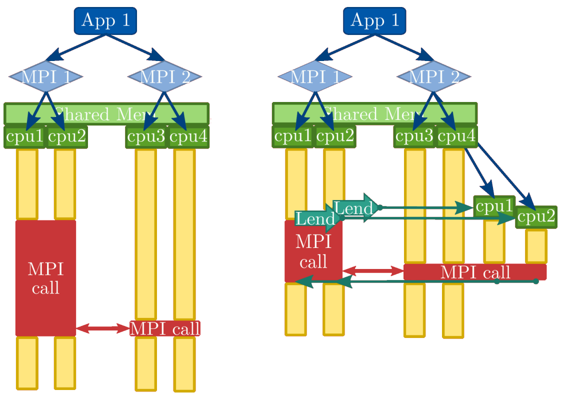

Figure 5: *Left: Unbalanced hybrid application. Right: Hybrid application balanced with DLB*

In Figure 5 we can see the behavior of DLB when load balancing a hybrid application. On the left side, we can see an unbalanced MPI+OpenMP application with two MPI processes and two OpenMP threads per process. On the right side, we can see the same execution when load balanced with DLB. We can observe that when the first MPI process reaches a blocking MPI call, it lends its resources to the second MPI process. At this point, the second MPI process can use four OpenMP threads and finish its computation faster. When completing the MPI blocking call, each MPI process recovers its original resources.

The DLB approach can be used with any of the parallelizations (i.e., Multidependences or Coloring), but as DLB relies on the shared memory parallelism to load balance, its performance is dependent on the performance of the shared memory parallelization. For this reason, all the runs with DLB, both for the current research and production runs, are performed using the Multidependences version for the assembly phase and atomics for the subgrid scale.

V. Performance Evaluation

i. Experimental setup

In this evaluation, we simulate the transport of particles in the human airways during a rapid inhalation. The details of the simulation are described in Section i and have been obtained running the latest version of Alya (r8941) averaging 10 time steps. Production simulations can run for up to 10^5 time steps.

We evaluated the runtime methods on three different platforms that are explained in Section II. All the experiments have been executed in two nodes of each platform.

ii. Multidependences

In this section, we evaluate the performance of multidependences compared with the implementation using a coloring algorithm or `atomic`. We will evaluate the performance in two phases of the simulation, the matrix assembly and the subgrid-scale (SGS).

The benefit of using multidependences in the matrix assembly is to avoid the use of `atomic` and preserve the spatial locality. In the case of the SGS, no update of a shared structure is involved; therefore, there is no need for using `atomic` pragmas. Nevertheless, we will evaluate the performance in this phase to see the overhead added by the multidependences.

We have executed three different versions of each simulation, using `atomic` pragmas labeled as *Atomics*, a coloring algorithm labeled as *Coloring* or the multidependences implementation labeled as *Multidep*. For each version we have executed different combinations of MPI processes and threads, with 1, 2, 4, 8 or 16 threads per MPI process. In the charts the combination is shown as: *Total number of MPI processes* $\times$ *Number of OpenMP threads per MPI process*.

In this section, we show the speedup obtained by each hybrid execution with respect to the MPI-only version using the same number of nodes in each cluster (i.e., running with 96 MPI processes in MareNostrum4, 80 in Power9 and 128 in Dibona). Within the same cluster, we compute the speedup S as: $S_c = t_M / t_c$, where t_c is the time spent for simulating a given problem with the configuration c of MPI processes and OpenMP threads and t_M is the time spent for simulating the same problem using an MPI-only implementation.

Platform	**#MPI**	**Ass.** [s]	**SGS** [s]
MareNostrum4	96	6.75	4.16
Dibona	128	7.48	4.72
Power9	80	9.30	6.45

Table 3: *Time spent in each phase of Alya when running the MPI-only version in two nodes of the evaluation platforms*

Table 3 contains the average duration in seconds of the two phases of Alya that we are considering when running with the MPI-only implementation. These numbers have been obtained running in two nodes of each platform filling the nodes with MPI processes. The values shown in this table are the ones used to compute the speed up in

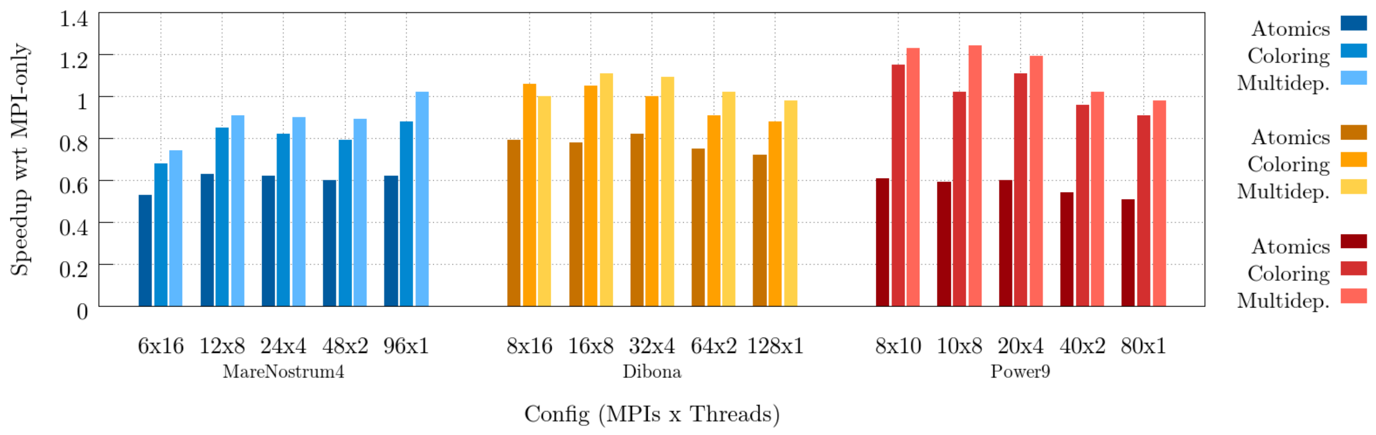

Figure 6: *Speedup of the assembly phase of Alya with respect to the MPI-only version of the code (higher is better).*

the following plots. We show them here for completeness although our goal is not to compare the architectures in terms of absolute performance.

In Figure 6 we can see the speedup obtained by each version with different combinations of MPI processes and threads in the matrix assembly phase.

We can observe that adding the second level of parallelism with the most naive approach (using the `atomic` pragma) have worse performance than the MPI-only version (the speedup is in fact below one in all the cases). Although this observation is correct for all the clusters, the negative impact of the `atomic` in the performance is higher in MareNostrum4 (based on Intel technology) and Power9 (based on IBM technology) than in Dibona (based on Arm technology). The different impact of the `atomic` in the performance can be due to the architectural differences between the three clusters.

Although it is out of the scope of this paper to evaluate how each architecture handles the atomic operations, we observe that the architectural differences among the platforms are reflected in the Instruction Per Cycle (IPC) achieved by the different versions in this phase. We chose IPC as a high-level metric because it is a common metric in HPC and it allows us to spot inefficiencies. A low IPC value will in fact highlight problems such as stalls on the different stages of the pipeline, sub-optimal memory access patterns or architectural limitations hit by the applications.

When running the MPI-only version, the average IPC in the Dibona cluster is ~ 1.56, while with the atomic version the IPC is ~ 1.12. On the other hand, in MareNostrum4 the IPC of the MPI-only version is ~ 1.79 and the IPC in the matrix assembly when using atomics is ~ 1.23 (corresponding to a reduction of 30%). In Power9 the IPC in the matrix assembly when using the MPI-only version is ~ 1.38 and it is reduced by 50% when using the Atomics parallelization obtaining ~ 0.70.

The coloring version achieves a better performance than the atomics version on the three architectures, in the case of MareNostrum4, it is still far from achieving the performance of the MPI-only code. In the Arm-based cluster, the performance of the coloring version depends on the configuration of OpenMP threads and MPI processes (varying between a speed up of ~ 0.9 and ~ 1.1). Finally, in Power9, the performance of the hybrid version using the coloring parallelization is better than the MPI-only version for some of the configurations.

We can observe that the performance of the coloring version with respect the MPI-only version is different for each architecture. The coloring version has worse data locality than the other approaches; the difference in performance that obtains in each cluster can be explained by the behavior of the different memory and cache configurations that each architecture leverages.

Nevertheless, the best version in all cases is the multidependences version; this version has a good data locality and does not need atomic operations. It is confirmed by the IPC values obtained: in all the clusters IPC is, in fact, 94% to 96% of the one achieved by the MPI-only version.

Code version	**MareNostrum4**		**Dibona**		**Power9**	
	Ass.	**SGS**	**Ass.**	**SGS**	**Ass.**	**SGS**
MPI-only	1.79	1.82	1.58	1.58	1.38	1.28
Atomics	1.23	1.84	1.09	1.57	0.70	1.25
Coloring	1.63	1.74	1.19	1.17	1.27	1.21
Multidep.	1.88	1.94	1.55	1.57	1.34	1.22

Table 4: *Instructions per Clock Cycle (IPC) for each of the experimental platforms in two relevant phases of the code, Assembly (Ass.) and SubGrid Scale (SGS)*

Table 4 contains the average IPC obtained with the different versions in each cluster for each phase. These values have been obtained instrumenting the executions with Extrae and analyzing the traces with Paraver. For studying the IPC, we filled the nodes with MPI processes and used one OpenMP thread per process: this setting was selected to avoid interference of other factors like different data partitioning.

In Figure 7 is presented the execution time of the subgrid-scale computation for the different versions and clusters. As we explained before, the subgrid-scale does not perform global operations, so it does not need to protect a race condition, and therefore should not require `atomic` operations. Nonetheless, we want to show the performance of the coloring and multidependences versions to evaluate the overhead introduced by these techniques.

We can see that in the three clusters and all configurations the multidependences implementation obtains about the same performance as the atomics version, while the coloring version is the one with the lowest performance. This is due to the loss of data locality when using the coloring algorithm. In all clusters, MareNostrum4, Dibona and Power9, we can observe an overhead below 10% associated with the use of coloring and multidependences.

We can observe that all the hybrid MPI+OpenMP versions outperform the MPI-only execution in Dibona and Power9 with all the configurations of MPI processes and threads. In MareNostrum4 only the hybrid configurations 96×1 achieves a performance close to the MPI-only version, while the other ones perform worse than the MPI-only.

With the evaluation presented in this section, we can conclude that the multidependences version is the best option in all the platforms when performing a reduction over a large array. We have also seen that the overhead introduced by multidependences is negligible if used in a code without a race condition (i.e., subgrid-scale).

iii. DLB

To evaluate the impact of using DLB on the performance of CFPD codes, we run two kinds of simulations: in one of them $4 \cdot 10^5$ particles are injected in the respiratory system, and in the other one $7 \cdot 10^6$ particles are injected.

With these two simulations we represent two different scenarios: one where the main computational load is in the fluid code, injecting only $4 \cdot 10^5$ particles; and another one where the main computational load is in the particles code, injecting $7 \cdot 10^6$ particles.

All the experiments executed in this section were obtained using the multidependences version of the code to solve the matrix assembly and the atomic version for the subgrid-scale because in the previous section were the versions that obtained the best performance in each phase. Also, all tests have been performed using one OpenMP thread for each MPI process.

As explained in Figure 3, this CFPD simulation can be executed in a synchronous or coupled mode. When running the coupled mode, the number of processes assigned to the computation of the fluid f and the number of processes assigned to the computation of the particles p must be decided by the user. We present experiments using both modes and varying f and p when running coupled simulations.

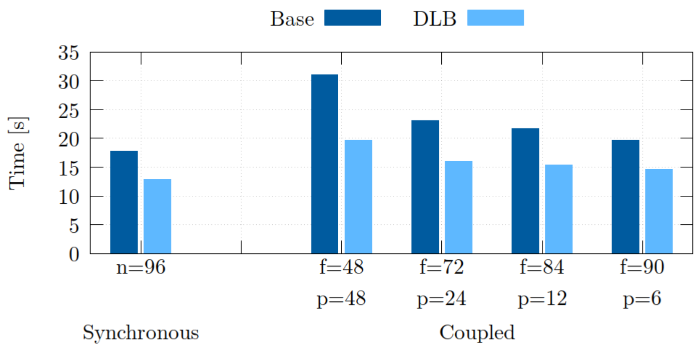

Figure 8: *Simulation of* $4 \cdot 10^5$ *particles in MareNostrum4*

In Figure 8 we can see the execution time when simulating the transport of $4 \cdot 10^5$ particles in MareNostrum4. In the x axis the different modes and combinations of MPI processes are represented in the form $f + p$. We can observe that depending on the mode and combination of MPI processes the execution time can change up to $2\times$ compared to the original code. The use of DLB improve all the executions of the original code. The improvement obtained running with DLB depends on the mode (synchronous or coupled) and on the combination of MPI processes assigned to each part of the problem (fluid or particle).

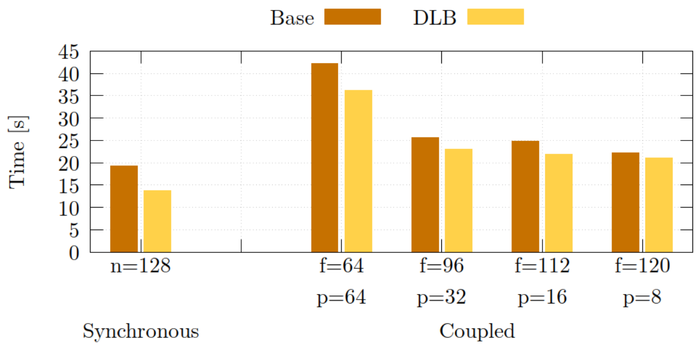

Figure 9: *Simulation of* $4 \cdot 10^5$ *particles in Dibona*

In Figure 9 is shown the execution time for simulating $4 \cdot 10^5$ particles in Dibona. In the Arm-based cluster, the trend in performance of this simulation is similar to the Intel-based one. If the user takes a wrong decision (e.g., running the coupled execution with 64 MPI processes for the fluid and 64 MPI processes for the particles), the simulation can be $2\times$ slower than running the best configuration (e.g., synchronous execution). Also in Dibona, the use of DLB improves the performance of all the configurations, and minimize the effect of choosing a bad combination of MPI processes.

Figure 10 shows the execution time when simulating $4 \cdot 10^5$ particles using the Power9 cluster. In this cluster,

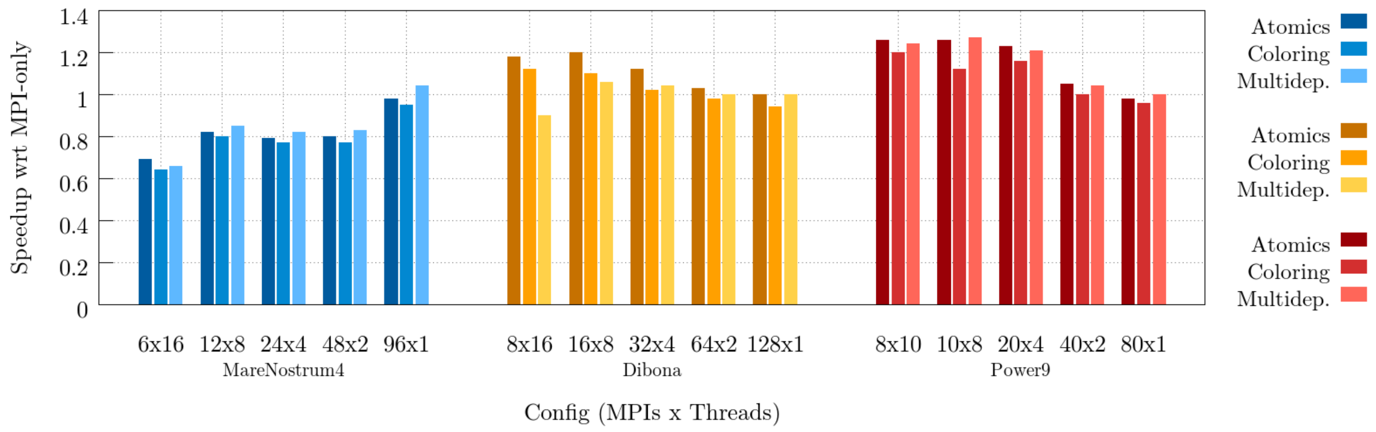

Figure 7: *Speedup of the subgrid-scale (SGS) phase of Alya with respect to the MPI-only version of the code (higher is better)*

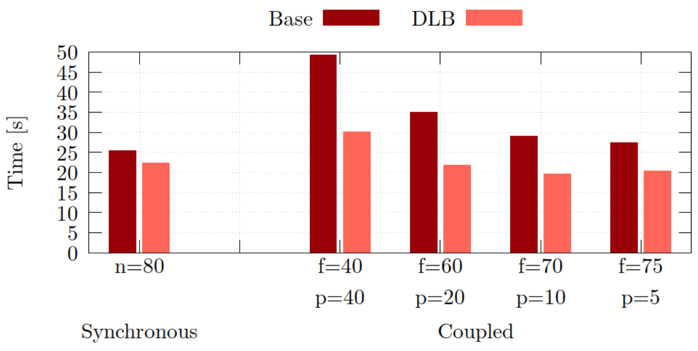

Figure 10: *Simulation of* $4 \cdot 10^5$ *particles in Power9*

the conclusions are similar to the ones obtained with the Intel and Arm ones. The performance can vary drastically depending on the configuration of the coupled execution chosen, and the use of the DLB library can alleviate this effect improving all the executions.

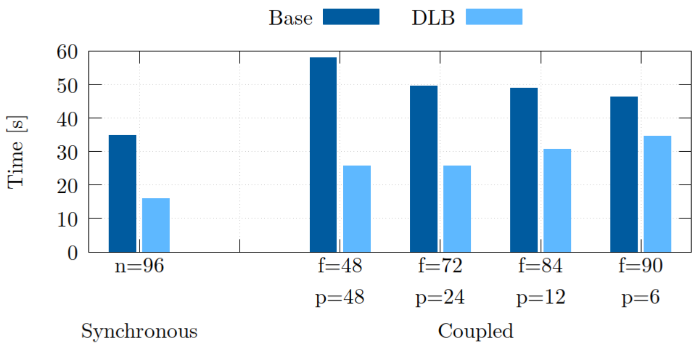

Figure 11: *Simulation of* $7 \cdot 10^6$ *particles in MareNostrum4*

The execution time when simulating the transport of $7 \cdot 10^6$ of particles in MareNostrum4 can be seen in Figure 11. We can see that respect the simulation of $4 \cdot 10^5$ particles the computational load has increased significantly. The impact of using DLB in this simulation is even higher than in the previous one, obtaining an improvement between 1.7× and 2.2× respect to the original execution.

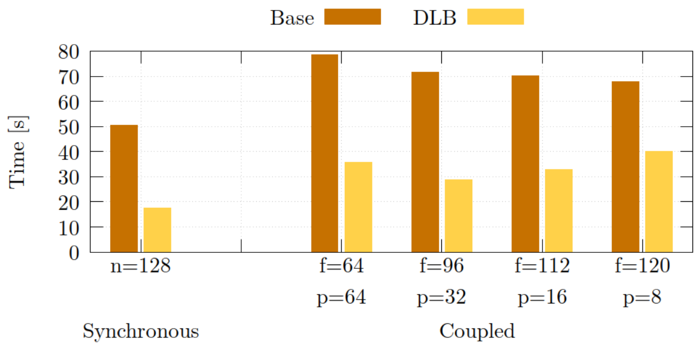

Figure 12: *Simulation of* $7 \cdot 10^6$ *particles in Dibona*

In Figure 12 we can see the execution time when simulating $7 \cdot 10^6$ particles in the Dibona cluster. We can observe that the trend in the performance of the original execution when changing the number of MPI processes is different respect to the simulation of $4 \cdot 10^5$ particles, and also when compared to the same simulation in the Intel-based system. This means that users can not rely on a single configuration as the optimum one. The optimum configuration depends in fact on the simulation (simulating $4 \cdot 10^5$ or $7 \cdot 10^6$ particles implies a different behavior), on the mode and distribution of MPI processes chosen and also on the underlying architecture.

The execution with DLB, in this case, speeds the simulation up between 2× and 3× with respect to the original execution. Moreover, the performance when using DLB is independent of the decision taken by the user in the mode and distribution of MPI processes between codes.

Figure 13 shows the elapsed time when simulating the transport of $7 \cdot 10^6$ particles in the Power9 cluster. We can observe again that the trend when changing f and p is different from the Arm and Intel-based systems. Nevertheless, DLB improves the performance in all the cases and maintains an almost constant performance independently

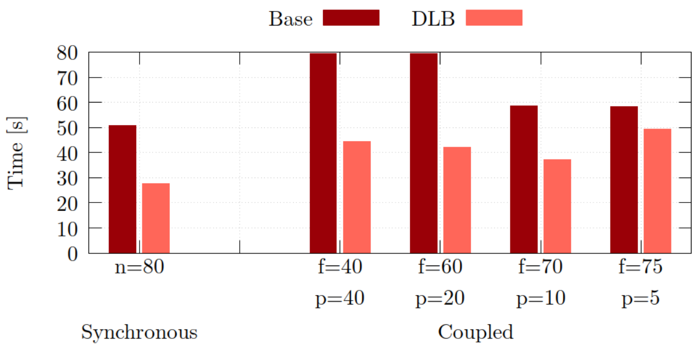

Figure 13: *Simulation of* $7 \cdot 10^6$ *particles in Power9*

of the coupling configuration chosen.

VI. Conclusions

In this paper, we analyzed the performance of a simulation tracking the transport of particles within the human respiratory system. We showed that the performance of this kind of simulations is affected by factors going from the simulation parameters (e.g., the number of particles injected) to the underlying architecture of the HPC cluster. For this reason, we rely on runtime techniques that will improve the performance of CFPD simulations independently from the simulation parameters and the architecture details.

One of the techniques that we evaluated are the iterators over dependences that will be added in the new release of OpenMP (5.0). Using these iterators, we can define multidependences among tasks (i.e., the number of dependences is decided at runtime, not compile time). We take advantage of the early implementation of multidependences in the OmpSs programming model to evaluate it in a CFPD simulation on three different architectures: an Intel-based, an Arm-based and an IBM-based system.

We have seen that the use of multidependences can improve the performance of the matrix assembly phase when using a hybrid parallelization. We have also observed that its impact depends on the architecture we are using. In the Intel-based cluster, the performance of multidependences achieves a 1.7× speedup respect to the implementation using `omp atomic` pragmas. In the Arm-based cluster the speedup obtained by the multidependences version is 1.4× respect to the `atomic` version and in the IBM-based cluster it is 2.0×.

The DLB library offers a load balancing mechanism transparent to the application and the architecture. DLB relies on the broadly adopted programming model OpenMP to improve resource utilization. In this paper, we have analyzed its performance when applied to a CFPD simulation. One of the main characteristics of CFPD simulations is that they must solve two different physic problems: the velocity of the fluid and the transport of the particles. The users running these simulations can decide whether to run them in a synchronous mode or in a coupled mode and, when running the coupled mode, how many computational resources to assign to each of the physics problems.

We have shown that the execution time of the simulation can be doubled if a bad decision is taken. Also, that the best decision is not easy to find without a previous performance analysis of the simulation. We have demonstrated that using DLB improves the performance of the execution in all the cases, independently on the architecture and the configuration chosen by the user. We obtained a speedup of up to 2× with respect to the original code using the same number of resources.

Moreover, using DLB relieves the user of deciding which configuration for the coupling of physics (synchronous or coupled, number of processes solving the fluid, f and number of processes solving the particle transport, p) to choose for his simulation. The performance of the simulation when using DLB is, in fact, less dependent on the chosen configuration.

Finally, the recommendation for code developers and users is to rely on runtime techniques to avoid architectural differences. In the case we have presented, the best option is to use *Multidependences* when assembling the matrix and the *Atomics* when computing the subgrid scale, and DLB for all the runs.

Acknowledgments

This work is partially supported by the Spanish Government (SEV-2015-0493), by the Spanish Ministry of Science and Technology project (TIN2015-65316-P), by the Generalitat de Catalunya (2017-SGR-1414), and by the European Mont-Blanc projects (288777, 610402 and 671697).

References

[1] Marco Baity-Jesi, Rachel A Baños, Andres Cruz, Luis Antonio Fernandez, José Miguel Gil-Narvión, Antonio Gordillo-Guerrero, David Iñiguez, Andrea Maiorano, Filippo Mantovani, Enzo Marinari, et al. Janus II: A new generation application-driven computer for spin-system simulations. *Computer Physics Communications*, 185(2):550–559, 2014.

[2] Fabio Banchelli, Marta Garcia, Marc Josep, Filippo Mantovani, Julian Morillo, Kilian Peiro, Guillem Ramirez, Xavier Teruel, Giacomo Valenzano, Joel Wanza Weloli, Jose Gracia, Alban Lumi, Daniel Ganellari, and Patrick Schiffmann. MB3 D6.9 – Performance analysis of applications and mini-applications and benchmarking on the project test platforms. Technical report, 2019. Version 1.0.

[3] Milind Bhandarkar, Laxmikant V Kalé, Eric de Sturler, and Jay Hoeflinger. Adaptive load balancing for mpi programs. In *International Conference on Computational Science*, pages 108–117. Springer, 2001.

[4] Luca Biferale, Filippo Mantovani, Marcello Pivanti, Mauro Sbragaglia, Andrea Scagliarini, Sebastiano Fabio Schifano, Federico Toschi, and Raffaele Tripiccione. Lattice Boltzmann fluid-dynamics on the QPACE supercomputer. *Procedia Computer Science*, 1(1):1075–1082, 2010.

[5] Hadrien Calmet, Alberto M Gambaruto, Alister J Bates, Mariano Vázquez, Guillaume Houzeaux, and Denis J Doorly. Large-scale cfd simulations of the transitional and turbulent regime for the large human airways during rapid inhalation. *Computers in biology and medicine*, 69:166–180, 2016.

[6] Enrico Calore, Alessandro Gabbana, Jiri Kraus, E Pellegrini, Sebastiano Fabio Schifano, and Raffaele Tripiccione. Massively parallel lattice Boltzmann codes on large GPU clusters. *Parallel Computing*, 58:1–24, 2016.

[7] Enrico Calore, Filippo Mantovani, and Daniel Ruiz. Advanced performance analysis of hpc workloads on cavium thunderx. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 375–382. IEEE, 2018.

[8] Enrico Calore, Sebastiano Fabio Schifano, and Raffaele Tripiccione. Energy-performance tradeoffs for HPC applications on low power processors. In *European Conference on Parallel Processing*, pages 737–748. Springer, 2015.

[9] DLB library online, March 2018.

[10] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[11] Charbel Farhat and Luis Crivelli. A general approach to nonlinear FE computations on shared-memory multiprocessors. *Computer Methods in Applied Mechanics and Engineering*, 72(2):153–171, 1989.

[12] Árpád Farkas and István Szöke. Simulation of bronchial mucociliary clearance of insoluble particles by computational fluid and particle dynamics methods. *Inhalation toxicology*, 25(10):593–605, 2013.

[13] Yu Feng, Zelin Xu, and Ahmadreza Haghnegahdar. Computational fluid-particle dynamics modeling for unconventional inhaled aerosols in human respiratory systems. In *Aerosols-Science and Case Studies*. InTech, 2016.

[14] Gary H. Ganser. A rational approach to drag prediction of spherical and nonspherical particles. *Powder Technology*, 77:143–152, 1993.

[15] Marta Garcia, Jesus Labarta, and Julita Corbalan. Hints to improve automatic load balancing with LeWI for hybrid applications. *Journal of Parallel and Distributed Computing*, 74(9):2781–2794, 2014.

[16] Marta Garcia-Gasulla, Guillaume Houzeaux, Roger Ferrer, Antoni Artigues, Victor López, Jesús Labarta, and Mariano Vázquez. MPI+X: task-based parallelization and dynamic load balance of finite element assembly. Technical report, 2018.

[17] Marta Garcia-Gasulla, Marc Josep-Fabrego, Beatriz Eguzkitza, and Filippo Mantovani. Computational Fluid and Particle Dynamics Simulations for Respiratory System: Runtime Optimization on an Arm Cluster. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18, pages 11:1–11:8. ACM, 2018.

[18] Ebrahim Ghahramani, Omid Abouali, Homayoon Emdad, and Goodarz Ahmadi. Numerical analysis of stochastic dispersion of micro-particles in turbulent flows in a realistic model of human nasal/upper airway. *Journal of Aerosol Science*, 67:188–206, 2014.

[19] Siegfried Höfinger and Ernst Haunschmid. Modelling parallel overhead from simple run-time records. *The Journal of Supercomputing*, 73(10):4390–4406, 2017.

[20] Guillaume Houzeaux, Ricard Borrell, Yvan Fournier, Marta Garcia-Gasulla, Jens Henrik Göbbert, Elie Hachem, Vishal Mehta, Youssef Mesri, Herbert Owen, and Mariano Vázquez. High-Performance Computing: Dos and Don'ts. In *Computational Fluid Dynamics-Basic Instruments and Applications in Science*. InTech, 2018.

[21] Guillaume Houzeaux and Javier Principe. A variational subgrid scale model for transient incompressible flows. *International Journal of Computational Fluid Dynamics*, 22(3):135–152, 2008.

[22] George Karypis. Metis: Unstructured graph partitioning and sparse matrix ordering system. *Technical report*, 1997.

[23] Cédric Lachat, Cécile Dobrzynski, and François Pellegrini. Parallel mesh adaptation using parallel graph partitioning. In *5th European Conference on Computational Mechanics (ECCM V)*, volume 3, pages 2612–2623. CIMNE-International Center for Numerical Methods in Engineering, 2014.

[24] Filippo Mantovani and Enrico Calore. Performance and power analysis of hpc workloads on heterogeneous multi-node clusters. *Journal of Low Power Electronics and Applications*, 8(2):13, 2018.

[25] Mont-Blanc Project, November 2018.

[26] OmpSs Specification, March 2018.

[27] OpenMP Architecture Review Board. OpenMP technical report 6: Version 5.0 preview 2. Technical report, November 2017.

[28] G Oyarzun, Ricard Borrell, Andrey Gorobets, Filippo Mantovani, and A Oliva. Efficient CFD code implementation for the ARM-based Mont-Blanc architecture. *Future Generation Computer Systems*, 79:786–796, 2018.

[29] Paraver, March 2018.

[30] Nikola Rajovic, Alejandro Rico, et al. The Mont-Blanc prototype: an alternative approach for HPC systems. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 444–455. IEEE, 2016.

[31] Sandia National Laboratory, November 2018.

[32] Harald Servat, Germán Llort, Kevin Huck, Judit Giménez, and Jesús Labarta. Framework for a productive performance optimization. *Parallel Computing*, 39(8):336–353, 2013.

[33] TOP500.org, November 2018.

[34] Jiyuan Tu, Kiao Inthavong, and Goodarz Ahmadi. *Computational fluid and particle dynamics in the human respiratory system*. Springer Science & Business Media, 2012.

[35] Mariano Vázquez, Guillaume Houzeaux, Seid Koric, Antoni Artigues, Jazmin Aguado-Sierra, Ruth Arís, Daniel Mira, Hadrien Calmet, Fernando Cucchietti, Herbert Owen, et al. Alya: Multiphysics engineering simulation toward exascale. *Journal of Computational Science*, 14:15–27, 2016.

[36] Raul Vidal, Marc Casas, Miquel Moretó, Dimitrios Chasapis, Roger Ferrer, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Evaluating the impact of OpenMP 4.0 extensions on relevant parallel workloads. In *International Workshop on OpenMP*, pages 60–72. Springer, 2015.