Computational Fluid and Particle Dynamics Simulations for Respiratory System: Runtime Optimization on an Arm Cluster

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

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

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

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

ABSTRACT

Computational fluid and particle dynamics simulations (CFPD) are of paramount importance for studying and improving drug effectiveness. Computational requirements of CFPD codes involves 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 both Intel- and Arm-based HPC clusters 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×, keeping the computational resources constant.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Applied computing** → *Systems biology*; • **Computer systems organization** → *Multicore architectures*; • **Hardware** → *Emerging architectures*;

1 INTRODUCTION AND RELATED WORK

Aerosolized delivery of drugs to the lungs is used to treat a number of respiratory diseases, such as asthma, chronic obstructive pulmonary disease, cystic fibrosis and pulmonary infections. But, it is well known that a large 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, e.g., [8, 24]. CFPD simulations can be used to help scientists to reproduce and reduce the lost aerosol fraction and improve the overall performance of the drug [9, 13].

Validated CFPD methods offer a powerful tool to predict the airflow and localized deposition of drug particles in the respiratory airways, in order 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 key to predict therapeutic responses related with chronic obstructive pulmonary disease. Deposition maps generated via CFPD simulations and their integration into clinical practice is a critical point to develop such a model. The understanding of these kind of dynamics can in fact 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 the aerosol physics. Due to the complexity and the 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.

Therefore it is clear that a successful study of aerosol dynamic strongly depends on two challenges: on one hand we have the physics of the problem, that needs to be translated into more and more precise models in order to increase 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 in order to reduce the time to obtain a solution.

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 technologies). As proven by previous studies on other fluid dynamics codes [1, 4], it is important to be able to efficiently exploit state-of-the-art architectures maintaining at the same time correctness and portability of the code.

In view of these observations, we consider in this paper Alya [25], a simulation code for high performance computational mechanics developed at the Barcelona Supercomputing Center (BSC). Alya is part of the UEABS (Unified European Applications Benchmark Suite), 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.

Using Alya we tested runtime mechanisms to mitigate load imbalance penalties on an Intel-based HPC cluster [12]. We extend here our previous work including the evaluation of those techniques on emerging Arm system-on-chips (SoCs) by Cavium.

Considering the momentum that Arm technology is gaining in the HPC panorama, we consider it is important to understand the behaviour of such technology under a production HPC workload. There have been previous evaluations of relevant scientific codes on

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology through TIN2015-65316-P project, by the Generalitat de Catalunya (contract 2017-SGR-1414) and from the European Union's Horizon 2020 under grant agreement No 785907 (HBP SGA2)

Arm-based platforms targeting both performance improvement [18, 21] and energy reduction [5, 17], however we want to stress the fact that we study in our paper a production code evaluating a real use case on a server grade Arm platform targeting data centers and code portability.

As already mentioned, the complexity and the size of production CFPD codes do not allow machine dependent fine tuning for each platform used to perform studies. For this reason we propose here two software techniques: *multidependences*, for improving performance avoiding atomic operations while running on a shared memory system [26] and *DLB*, a dynamic load balancing library able to detect load imbalance within a parallel system and redistribute workload in order to minimize the inefficiency [11, 16]. The idea is to demonstrate how these tools, deployed at 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.

The main contributions of this paper are: *i)* we provide the evaluation of a production use case of real biological HPC simulation on a Arm-based HPC cluster. *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: Section 2 introduces Alya simulation infrastructure as well as its computational profiling. Section 3 introduces the key ideas of the runtime techniques that we evaluate. In Section 4 we briefly introduce the hardware platforms, the software configurations used for our tests and the results of our evaluation. In Section 5 we discuss the lessons learned and next steps for our work.

2 PARTICLE TRACKING IN THE RESPIRATORY SYSTEM

2.1 Simulation details

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 a clinical practice to personalize therapies involving treatments with aerosol.

In this work we simulate the transport of particles injected in an unsteady flow in the human large airways during a rapid inhalation. The use of massive computational resources allows to capture 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 [3]. In particular, the mesh is hybrid and composed of 17.7 million elements with different geometry: *prisms*, to resolve accurately the boundary layer; *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.

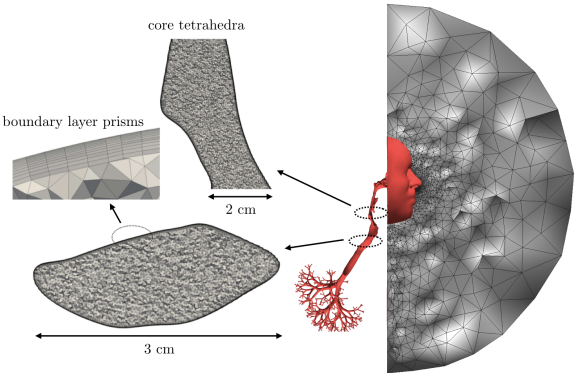

Figure 1: Mesh representing the human respiratory system.

As introduced in Section 1, the application used in this document is the high performance computational mechanics code, Alya [25]. Alya is parallelized using MPI and OpenMP, but, production runs are usually performed using a pure MPI parallelization approach.

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 the Newton's second law for a fluid continuous 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

$$\rho_f \frac{\partial \boldsymbol{u}_f}{\partial t} + \rho_f(\boldsymbol{u}_f \cdot \nabla)\boldsymbol{u}_f - \nabla \cdot [2\mu_f \boldsymbol{\varepsilon}(\boldsymbol{u}_f)] + \nabla p_f = \boldsymbol{0}, \quad (1)$$

$$\nabla \cdot \boldsymbol{u}_f = 0, \quad (2)$$

together with initial and boundary conditions. The velocity strain rate is defined as $\boldsymbol{\varepsilon}(\boldsymbol{u}_f) := \frac{1}{2}(\nabla \boldsymbol{u}_f + \nabla \boldsymbol{u}_f^t)$. 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 [15].

Particles are transported solving the 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 the 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

$$\boldsymbol{F}_g = m_p \boldsymbol{g}, \quad (4)$$

$$\boldsymbol{F}_b = -m_p \boldsymbol{g} \rho_f / \rho_p, \quad (5)$$

$$\boldsymbol{F}_D = (\pi/8)\mu_f d_p C_d \mathrm{Re}_p (\boldsymbol{u}_f - \boldsymbol{u}_p), \quad (6)$$

respectively, where Re and C_D are the particle Reynolds number and drag coefficient given by Ganser's formula[10], respectively:

$$\mathrm{Re}_p = \rho_f d_p |\boldsymbol{u}_f - \boldsymbol{u}_p| / \mu_f, \quad (7)$$

$$C_D = \frac{24}{\mathrm{Re}_p}[1 + 0.1118(\mathrm{Re}_p)^{0.65657}] + \frac{0.4305}{1 + \frac{3305}{Re_p}}. \quad (8)$$

The time integration is based on Newmark's method. The time step used in this paper is 10^{-4} seconds.

2.2 Profile and performance analysis

In this section we study a trace of the Alya simulation gathered on one node of the Thunder cluster, introduced in Section 4.2 (i.e., running with 96 MPI processes in the same cluster evaluated in Section 4).

We use Extrae [22] to obtain a performance trace and then Paraver [19] 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, pink and blue are algebraic solvers to compute the momentum and continuity of the fluid and purple represents the 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 black on the right part 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:

$$L_n = \frac{\sum_{i=1}^{n} t_i}{n \cdot \max_{i=1}^{n} t_i} \quad (9)$$

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 lo load imbalance.

Phase	L_{96}	% Time
Matrix assembly	0.66	40.84%
Solver1	0.90	16.13%
Solver2	0.89	4.20%
SGS	0.61	21.43%
Particles	0.02	3.37%

Table 1: Load balance and percentage of the total execution time for different phases of the respiratory simulation executed with 96 MPI processes.

In Table 1, 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 [12].

It is important to note that the percentage of time spent in the particle phase is directly proportional to the amount of particles injected in 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 amount 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 will 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.

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 will 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 will assign to solve the fluid and how many processes will solve the particles. Depending on this decision and the amount of particles injected, the more loaded processes will be the ones solving the fluid or the ones solving the particles. 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 will be explained in detail in Section 3.2.

3 RUNTIME TECHNIQUES

3.1 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,

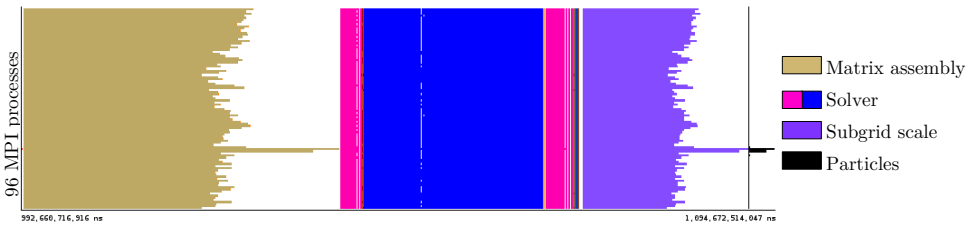

Figure 2: Trace of respiratory simulation with 96 MPI processes gathered on a node of the Thunder cluster

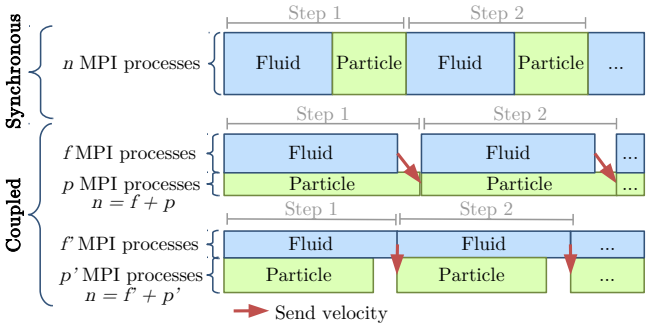

Figure 3: Execution modes for CFPD simulations with Alya

a set of local operations is performed in order to assemble the local matrices. More details can be found in [12, 14].

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

(1) On one hand, the algebraic system assembly over the elements is performed locally to each MPI process, thus, no MPI communication are 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 implementations.

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 in 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.

To avoid the use of an atomic operation, a coloring technique can be used [7]. The coloring technique, as can be seen in Figure 4, consists in assigning a color to each element. Elements that share a

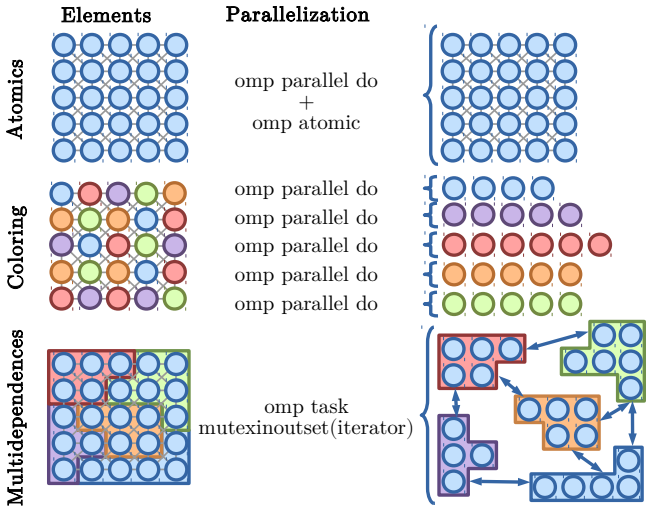

Figure 4: Parallelization approaches for the matrix assembly

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.

The third approach that we consider is *multidependences*. This approach consists in partitioning each MPI domain in different subdomains using Metis. 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.

In the multidependences version we used two new features that will be introduced in the OpenMP standard in the 5.0 release [2]. On one hand, the `iterators` to define a list of dependences, this feature allow to define a dynamic number of dependences between tasks that is 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.

These two new features of the standard were first implemented in the OmpSs programming model [6, 23]. 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 [26]. We will take advantage of these early implementations tot test these two new features in a real code.

An important added value of the multidependences version is that its implementation does not requires significant changes in the code and leaves the parallelization quite clean and simple. 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.

3.2 Dynamic Load Balancing (DLB)

Dynamic Load Balancing (DLB) is a library that aims to improve the load balance of hybrid applications [16]. 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 [11].

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()`.

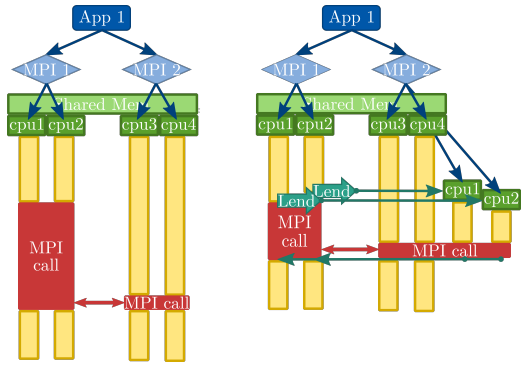

Figure 5: Left: Unbalanced hybrid application. Right: Hybrid application balanced with DLB

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 2 MPI processes and 2 OpenMP threads per process. On the right side we can see the same execution when load balanced with DLB. We can observe that when MPI process 1 reaches a blocking MPI call, it lends its resources to MPI process 2. At this point MPI process 2 will be able to use 4 OpenMP threads and finish so its computation faster. When finishing the MPI blocking call, each MPI process recovers its original resources.

4 PERFORMANCE EVALUATION

4.1 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 explained in Section 2.1 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 two different platforms that will be explained in the following section. All the experiments have been executed in two nodes of each platform.

4.2 Platforms

The tests presented in this work have been executed on *MareNostrum4*, the PRACE Tier-0 Supercomputer hosted at BSC and *Thunder*, a small cluster deployed at BSC within the framework of the EU Mont-Blanc project.

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 (8GB/core), while 3240 nodes are equipped with 12 × 8 GB DDR4-2667 DIMMS (2GB/core).

The Thunder cluster is composed of four computational nodes integrated in a 2U server box. Each node features a dual socket motherboard housing 2× Cavium ThunderX CN8890 Pass1 SoCs with 48× custom Arm-v8 cores at 1.8 GHz each socket, 128 GB DDR3 at 2.1 GHz per node and 256 GB SSD for local storage. Nodes are air cooled and interconnected using a single 40 GbE link. The SoC double precision peak performance is 172.80 GFlops while the SoC aggregate memory bandwidth is 51.20 GB/s.

The Thunder cluster is a state-of-the-art Arm-based test platform as *i)* it embeds 48 Armv8 custom designed cores per SoC, *ii)* it features a fairly high number of cores (96) running a single instance of Linux, *iii)* it is one of the first off-the-shelves platform based on Arm technology targeting data centers [20].

On both clusters we use the same version of the OmpSs programming model composed by the *Mercurium* 2.1.0 source to source compiler and the *Nanox* 15a compiler. On MareNostrum4 we leverage the *Intel MPI* distribution together with the *Intel compiler* 2017.4. On Thunder we use *OpenMPI* 3.0.1 for message passing flavour and the *GNU compiler suite* 5.3.0 as compiler.

4.3 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 of using `atomic` pragmas. Nevertheless, we will evaluate the performance in this phase in order 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 or 4 threads per MPI process. In the charts the combination is shown as: *Total number of MPI processes* × *Number of OpenMP threads per MPI process*.

In this section we show the speedup obtained by each hybrid execution with respect to the pure MPI version using the same number of nodes in each cluster (i.e., running with 96 MPI processes in MareNostrum4 and 192 MPI processes in Thunder). 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 a pure MPI implementation.

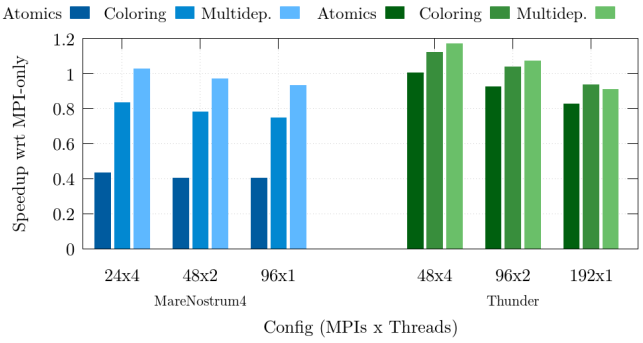

Figure 6: Speedup of hybrid assembly *wrt* the MPI-only code.

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 a worse performance than the pure MPI version (the speedup is in fact below one in almost all the cases). Although this observation is true for both clusters, the negative impact of the `atomic` in the performance is much higher in MareNostrum4 (based on Intel technology) than in Thunder (based on Arm technology). This difference can be explained by the higher instruction level parallelism (ILP) exposed by the Intel architecture, compared to the Arm one (e.g., Intel leverages out-of-order cores, while the Cavium implementation of the Arm core is in-order). These architectural differences are reflected in the Instruction Per Cycle (IPC) achieved by the different versions in this phase. When running the MPI-only version the average IPC in the Thunder cluster is ~ 0.49, while with the atomic version the IPC is ~ 0.42 (a reduction of 14%). On the other hand, in MareNostrum4 the IPC of the MPI-only version is ~ 2.25 and the IPC in the matrix assembly when using atomics is ~ 1.15 (corresponding to a reduction of 50%).

The coloring version achieves a better performance than the atomics version on both architectures, in the case of Thunder achieving the same performance or better than the MPI-only code.

Nevertheless, the best version in all the 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 both clusters IPC is in fact 94% to 96% of the one achieved by the MPI-only version.

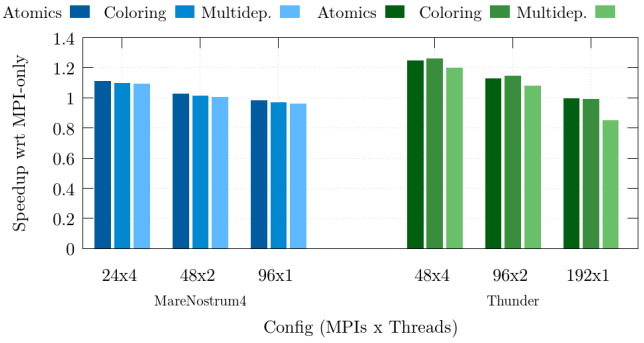

Figure 7: Speedup of hybrid SGS *wrt* the MPI-only code.

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.

In both clusters, MareNostrum4 and Thunder, we can observe an overhead below 10% associated with the use of coloring and multidependences, because its performance is lower than the one obtained with the atomic version. We can observe that the MPI+OpenMP versions outperforms the MPI-only execution.

4.4 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 atomics 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.

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

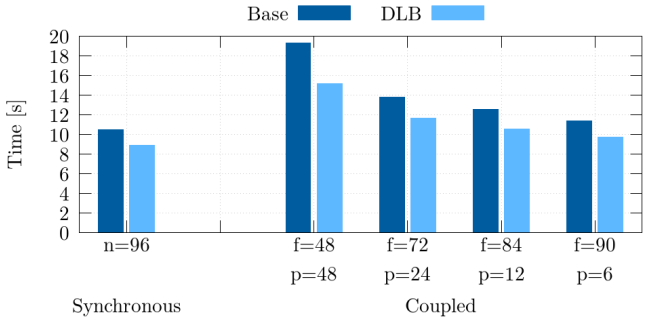

Figure 8: Simulation of $4 \cdot 10^5$ particles in MareNostrum4

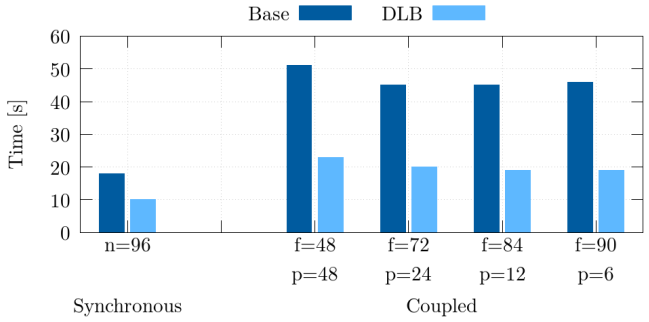

Figure 10: Simulation of $7 \cdot 10^6$ particles in MareNostrum4

combination of MPI processes the execution time can change up to 2× compared to the original code. The use of DLB improve all the executions of the original code. The improvement obtained by DLB respect the same type of execution of the original code depends on the mode and combination of MPI processes.

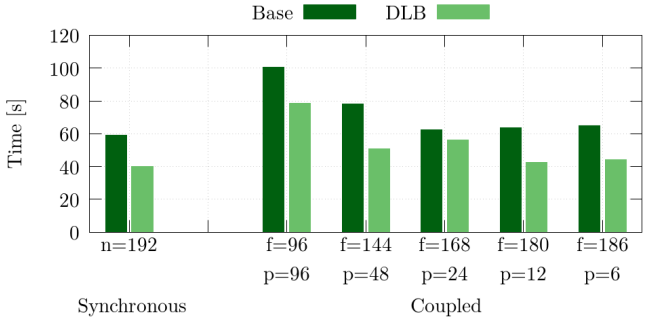

Figure 9: Simulation of $4 \cdot 10^5$ particles in Thunder

In Figure 9 is shown the execution time for simulating $4 \cdot 10^5$ particles in Thunder. 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 96 MPI processes for the fluid and 96 MPI processes for the particles) the simulation can be 2× slower than running the best configuration (e.g., synchronous execution). Also in Thunder the use of DLB improves the performance of all the configurations, and minimize the effect of choosing a bad combination of MPI processes.

The execution time when simulating the transport of $7 \cdot 10^6$ of particles in MareNostrum4 can be seen in Figure 10. 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.

In Figure 11 we can see the execution time when simulating $7 \cdot 10^6$ particles in the Thunder 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

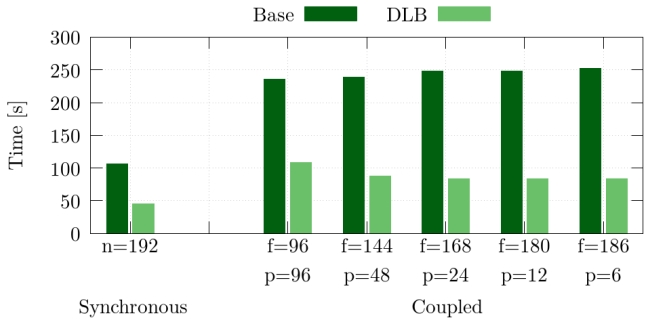

Figure 11: Simulation of $7 \cdot 10^6$ particles in Thunder

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 from the decision taken by the user in the mode and distribution of MPI processes between codes.

5 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 these kind of simulations is affected by factors going from the simulation parameters (i.e. 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 are able to define multidependences between 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 an Intel-based and an Arm-based cluster.

We have seen that the use of multidependences can improve the performance of the matrix assembly phase when using a hybrid parallelization. We have observed also that its impact depends

on the architecture we are using. In the Intel-based cluster the performance of multidependences achieves a 2.5 speedup respect to the implementation using `omp atomic` pragmas. In the Arm-based cluster the speedup obtained by the multidependences version is 1.2 respect to the `atomic` version.

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 the 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.

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 of OpenMP thread per MPI process to choose for his simulation. The performance of the simulation when using DLB is in fact less dependent on the chosen configuration.

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

[2] OpenMP Architecture Review Board. November 2017. *OpenMP Technical Report 6: Version 5.0 Preview 2.* Technical Report. http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf

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

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

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

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

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

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

[9] Yu Feng, Zelin Xu, and Ahmadreza Haghnegahdar. 2016. Computational Fluid-Particle Dynamics Modeling for Unconventional Inhaled Aerosols in Human Respiratory Systems. In *Aerosols-Science and Case Studies*. InTech.

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

[11] Marta Garcia, Jesus Labarta, and Julita Corbalan. 2014. Hints to improve automatic load balancing with LeWI for hybrid applications. *J. Parallel and Distrib. Comput.* 74, 9 (2014), 2781–2794.

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

[13] Ebrahim Ghahramani, Omid Abouali, Homayoon Emdad, and Goodarz Ahmadi. 2014. 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 (2014), 188–206.

[14] Guillaume Houzeaux, 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 *Computational Fluid Dynamics-Basic Instruments and Applications in Science*. InTech.

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

[16] DLB library. 2018. https://pm.bsc.es/dlb. (March 2018).

[17] Filippo Mantovani and Enrico Calore. 2018. Multi-Node Advanced Performance and Power Analysis with Paraver. In *Parallel Computing is Everywhere*, Vol. 32. IOS Press, 723–732. https://doi.org/10.3233/978-1-61499-843-3-723

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

[19] Paraver. 2018. https://tools.bsc.es/paraver. (March 2018).

[20] Miloš Puzović, Srilatha Manne, Shay GalOn, and Makoto Ono. 2016. Quantifying energy use in dense shared memory HPC node. In *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing*. IEEE Press, 16–23.

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

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

[23] OmpSs Specification. 2018. https://pm.bsc.es/ompss-docs/spec/. (March 2018).

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

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

[26] Raul Vidal, Marc Casas, Miquel Moretó, Dimitrios Chasapis, Roger Ferrer, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Evaluating the impact of OpenMP 4.0 extensions on relevant parallel workloads. (2015).