UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

UPCommons

Portal del coneixement obert de la UPC

http://upcommons.upc.edu/e-prints

Aquesta és una còpia de la versió *author's final draft* d'un article publicat a la revista *Computers & Fluids.*

http://hdl.handle.net/2117/335542

Article publicat / Published paper:

Alvarez, X.; Gorobets, A.; Trias, F.X. A hierarchical parallel implementation for heterogeneous computing. Application to algebra-based CFD simulations on hybrid supercomputers. "Computers and fluids", 2021, vol. 214, p. 104768/1-104768/13. DOI: <10.1016/j.compfluid.2020.104768>.

©2021. This manuscript version is made available under the CC-BY-NC-ND 4.0 license http://creativecommons.org/licenses/by-nc-nd/4.0/

A hierarchical parallel implementation for heterogeneous computing. Application to algebra-based CFD simulations on hybrid supercomputers

Xavier Álvarez-Farré[a], Andrey Gorobets[b], F. Xavier Trias[a,*]

[a] *Technical University of Catalonia, Heat and Mass Transfer Technological Center, Carrer Colom 11, 08222 Terrassa (Barcelona), Spain*
[b] *Keldysh Institute of Applied Mathematics of Russian Academy of Sciences, Miusskaya Sq. 4, 125047 Moscow, Russia*

Abstract

The quest for new portable implementations of simulation algorithms is motivated by the increasing variety of computing architectures. Moreover, the hybridization of high-performance computing systems imposes additional constraints, since heterogeneous computations are needed to efficiently engage processors and massively-parallel accelerators. This, in turn, involves different parallel paradigms and computing frameworks and requires complex data exchanges between computing units. Typically, simulation codes rely on sophisticated data structures and computing subroutines, so-called kernels, which makes portability terribly cumbersome. Thus, a natural way to achieve portability is to dramatically reduce the complexity of both data structures and computing kernels. In our algebra-based approach, the scale-resolving simulation of incompressible turbulent flows on unstructured meshes relies on three fundamental kernels: the sparse matrix-vector product, the linear combination of vectors and the dot product. It is noteworthy that this approach is not limited to a particular kind of numerical method or a set of governing equations. In our code, an auto-balanced multilevel partitioning distributes workload among computing devices of various architectures. The overlap of computations and multistage communications efficiently hides the data exchange overhead in large-scale supercomputer simulations. In addition to computing on accelerators, special attention is paid at efficiency on manycore processors in multiprocessor nodes with significant non-uniform memory access factor. Parallel efficiency and performance are studied in detail for different execution modes on various supercomputers using up to 9600 processor cores and up to 256 graphics processor units. The heterogeneous implementation model described in this work is a general-purpose approach that is well suited for various subroutines in numerical simulation codes.

Keywords: Parallel CFD, SpMV, Heterogeneous computing, Hybrid

*Corresponding author: F. Xavier Trias, xavi@cttc.upc.edu

supercomputer, CPU+GPU, MPI+OpenMP+OpenCL, CUDA

1. Introduction

Modern high-performance computing (HPC) systems consist of multiple hybrid computing nodes interconnected via a communication infrastructure. Hybrid nodes combine computing devices of different architectures, such as many-core central processing unit (CPU) and graphics processing unit (GPU), among others. To take advantage of such systems, the computing subroutines that form the algorithms, the so-called kernels, must be compatible with distributed- and shared-memory multiple instruction, multiple data parallelism, and more importantly, with stream processing, which is a very restrictive parallel programming paradigm. Therefore, a complex hierarchical parallel implementation model is required for combining the different parallel paradigms and the corresponding computing frameworks.

Many algorithms employed in scientific computing, especially in computational fluid dynamics (CFD), can greatly benefit from using massively parallel accelerators with high computing capabilities and fast integrated memory. Even those algorithms that have low arithmetic intensity, that is, the ratio of computing work in floating-point operations (FLOPs) to memory traffic in bytes. Nowadays, the use of GPU architectures in scientific computing has become rather mature, and there are many successful examples in the literature. For instance, the GPU-accelerated, compute-intensive phase-field simulation in [1] proved to be two orders of magnitude faster than its CPU counterpart. The solution of incompressible two-phase flows on multi-GPU clusters in [2] shown not only the GPU was quicker in memory-bounded applications but also more energy- and cost-efficient. An example of a multi-GPU simulation of supersonic and hypersonic flows can be found in [3]. One of the most notable GPU-based, large-scale turbulent flow simulations is presented in [4], reporting an impressive sustained performance of 13.7 PFLOPs per second (PFLOPS), 58% of peak, by using a very compute-intensive flux reconstruction approach [5]. Well-known commercial CFD codes and open-source platforms with large user bases often implement GPU extensions for solvers of systems of linear algebraic equations (SLAE), which may represent a significant part of the overall computing time. This approach can provide substantial acceleration with compactly localized changes in the code. Such an example can be found in [6], where the authors coupled a GPU-accelerated library for solving large sparse SLAE with the OpenFOAM platform and demonstrated performance on up to 128 nodes of a GPU-based cluster.

The popularity of the GPU-only implementation model is probably based on the assumption that GPUs are fast enough to neglect the computing power of CPUs. However, both architectures are still progressing, and their competition is far from over. On the one hand, traditional memory of the DDR4 standard cannot compete with on-chip high-bandwidth memory. It is unclear whether the next generations of CPUs are going to integrate fast on-chip memory, likewise

the Intel Xeon Phi series, or use external memory controllers connected via fast channels, such as the IBM open memory interface. Either way, the gap in memory bandwidth between architectures can be reduced. On the other hand, the CPU vendors also tend to unreasonably raise the peak performance, for instance, by increasing the number of fused multiply-add (FMA) units per core and by widening them (*e.g.*, Intel's SkyLake started to place two 512-bit FMA units per core). This, of course, has minimal impact on sustained performance in real-life memory-bound CFD applications. Shall the next generation place four 512-bit FMA units per core, the parity in peak performance may be easily reached.

Therefore, if either CPUs or GPUs can be used efficiently in CFD simulations, why not to use them both together? If we assume that the ratio between CPU's and GPU's bandwidth remains around 1:3, then the heterogeneous parallel models may result in about 30% gain in sustained performance, as shown in [7] on up to 640 hybrid nodes of the Lomonosov-2 supercomputer. Furthermore, developing an efficient heterogeneous implementation to maximize performance on all major computing architectures is beneficial regardless of the results of the competition between them. However, fully-heterogeneous applications that engage CPUs and accelerators as equal partners, co-execution, in other words, are still relatively rare, compared to GPU-only implementations. For instance, the phase-field simulation in [8] reported up to 1.017 PFLOPS in single-precision combining 4000 GPUs with 16000 CPU cores. The PyFR framework in [9] is written in Python and incorporates backends for OpenMP, CUDA and OpenCL frameworks that provide portability and heterogeneous computing capability. In [10], the authors apply the OpenACC directive standard in the Fortran code SENSEI for simulation of compressible flows on structured meshes, showing heterogeneous computing capability on a single workstation with two GPUs. Borrell et al. describe in [11] the multi-code co-execution approach implemented in the Alya framework and assess its performance on 40 hybrid nodes of the CTE-POWER cluster at Barcelona Supercomputing Center.

In our previous work [12], we proposed an algebra-based framework for heterogeneous computing as a portable solution for the scale-resolving simulation of incompressible turbulent flows on unstructured meshes. Roughly, the CFD algorithm relies on a set of only three algebraic kernels: a sparse matrix-vector product (SpMV), a linear combination of two vectors and a dot product. Note that these linear algebraic operations are simple and fully compatible with the stream processing paradigm. Thus, the size of the kernel code shrinks to hundreds of lines; portability becomes natural, and maintaining the OpenMP, OpenCL and CUDA implementations requires little effort. Besides, we can easily use standard libraries (*e.g.*, cuSPARSE [13] of NVIDIA, clSPARSE [14] of AMD) optimized for particular architecture, in addition to our specialized in-house implementations.

Among the aforementioned algebraic kernels, the SpMV is of the keenest interest in scientific computing. Large sparse matrices often appear in the numerical solution of partial differential equations. The sparsity pattern of these matrices (*i.e.*, the distribution of the non-zero coefficients) arises from the spa-

tial discretization, which depends on the numerical method employed. SpMV performance is very sensitive to the sparsity pattern, the storage format and the hardware architecture. Indeed, the SpMV has a very low arithmetic intensity (around 1/8 FLOP per byte in double precision), and indirect memory access to the input vector with unavoidable cache misses. Significant effort has been devoted to adapt sparse matrix storage formats to various computing architectures and matrix properties (*e.g.*,see [15, 16, 17]). Besides, in distributed-memory parallel processing, vector elements and matrix rows are distributed among a group of processes, and this operation requires communications between them. Therefore, efficient execution of the SpMV kernel on hybrid supercomputers requires a fine-tuning process: the right choice of the storage format, workload balancing, reordering of unknowns (to reduce the matrix bandwidth), memory access optimization (to minimize cache misses), overlapping of communication and computations (to hide data exchange overhead).

The heterogeneous execution of the SpMV on hybrid CPU+GPU systems was studied, for instance, in [18], showing a notable gain on a single node. In our previous works [19, 12], we studied heterogeneous SpMV performance on the ARM-based system of the Mont-Blanc project [20] and on various hybrid clusters. However, our previous multilevel parallel implementation had some weakness on multi-socket CPU systems, since it did not take into account the effects of non-uniform memory access (NUMA). In the present work, we addressed this issue and significantly improved performance on CPUs. Consequently, the gain of heterogeneous computing has grown. Besides, we notably reduced the communication overhead by multithreaded processing of data exchange between multiple devices inside hybrid nodes. It must be noted that the presented heterogeneous execution algorithm is, of course, not limited only to SpMV. It is a general-purpose approach that is well suited for various subroutines in numerical simulation codes, especially in CFD codes based on mesh methods. Therefore, there are no assumptions as to what exactly is computed in the kernels.

The rest of paper is organized as follows. In Section 2, a mathematical model and an algorithm for modeling incompressible turbulent flows are presented as a representative application example. It is shown how this algorithm ends with algebraic objects. Section 3 is devoted to multilevel parallelization and implementation details. In Section 4, performance is studied on various supercomputers, including the CPU-based MareNostrum 4, the hybrid Lomonosov-2, and the GPU-based TSUBAME3.0 and K-60. Finally, the results are summarized and the conclusions are given in Section 5.

2. Mathematical model and numerical algorithm

In this section, the simulation of turbulent flows is chosen as a representative example of the kind of applications for which the HPC^2 framework [12] is designed. Figure 1 illustrates several large-scale simulations that have been carried out with the original CPU-only version of the presented algorithm. The HPC^2 is

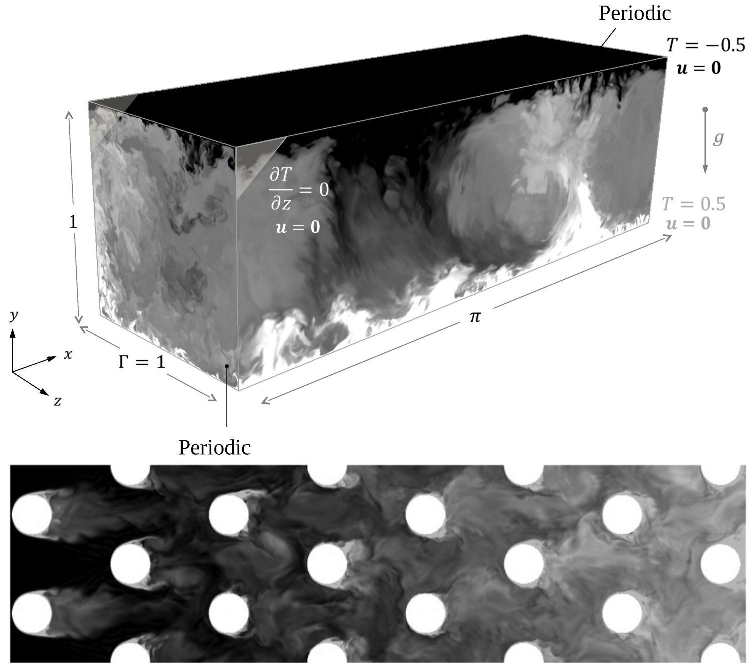

Figure 1: Top: schema of the Rayleigh-Bénard configuration studied in [24] displayed together with the instantaneous temperature field corresponding to the air-filled ($Pr = 0.7$) DNS at $Ra = 10^{10}$. Bottom: LES simulation of a wall-bounded pin matrix at $Re = 10000$ [25].

not limited to a particular kind of numerical method (*e.g.*, finite-volume, finite-difference, finite-element) or set of governing equations. For instance, problems that require additional transport equations such as combustion, multiphase flows, magnetohydrodynamics, transport of pollutants, turbulence modeling, can be cast into this framework in a similar manner. Also worth mentioning that, in some particular cases, an additional effort may be required to reformulate existing algorithms to fit with the matrix-vector paradigm. The implementation of flux limiters [21] (for problems with discontinuities such as interface tracking for multiphase flows), the discretization of the viscous term with spatially varying (eddy-)viscosity [22] and the computation of the surface tension term for multiphase flows [23] are examples thereof.

We consider the simulation of turbulent, incompressible flows of Newtonian fluids. The Boussinesq approximation is used to account for the density variations, the thermal radiation is neglected. Under these assumptions, the velocity field, $\boldsymbol{u}$, and the temperature, θ, are governed by the incompressible Navier-Stokes equations coupled with the temperature transport equation:

$$\partial_t \boldsymbol{u} + \mathcal{C}(\boldsymbol{u}, \boldsymbol{u}) = \nu \nabla^2 \boldsymbol{u} - \nabla p + \boldsymbol{f}, \qquad \nabla \cdot \boldsymbol{u} = 0, \tag{1}$$

$$\partial_t \theta + \mathcal{C}(\boldsymbol{u}, \theta) = \alpha \nabla^2 \theta, \tag{2}$$

where p represents the kinematic pressure, ν is the kinematic viscosity, and α is the thermal diffusivity. The non-linear convective term is given by $\mathcal{C}(\boldsymbol{u}, \phi) = (\boldsymbol{u} \cdot \nabla)\phi$ whereas for cases with buoyancy effects the body force vector is given by $\boldsymbol{f} = \boldsymbol{g}\beta\theta$ where $\boldsymbol{g}$ is the gravitational force and β is the thermal expansion coefficient.

The convective operator is skew-symmetric, $(\phi_1, \mathcal{C}(\boldsymbol{u}, \phi_2)) = -(\phi_2, \mathcal{C}(\boldsymbol{u}, \phi_1))$, whereas the diffusive is symmetric and positive-definite, *i.e.*, $\left(\phi_1, \nabla^2\phi_2\right) = \left(\phi_2, \nabla^2\phi_1\right)$ and $\left(\phi, \nabla^2\phi\right) \leq 0, \forall\phi$. Here, the inner-product of functions is defined in the usual way: $(a, b) = \int_\Omega a \cdot b d\Omega$. On the other hand, accuracy and stability need to be reconciled for numerical simulations of turbulent flows around complex configurations. With this in mind, a fully-conservative discretization method for general unstructured grids was proposed in [26]: it exactly preserves the symmetries of the underlying differential operators on a collocated unstructured mesh. Shortly, the temporal evolution of the spatially discrete velocity vector, $\boldsymbol{u}_c$, is governed by the following operator-based finite-volume discretization of (1)

$$\Omega_c^{3d}\frac{d\boldsymbol{u}_c}{dt} + \mathsf{C}_c^{3d}\left(\boldsymbol{u}_s\right)\boldsymbol{u}_c + \mathsf{D}_c^{3d}\boldsymbol{u}_c + \Omega_c^{3d}\mathsf{G}_c\boldsymbol{p}_c = \boldsymbol{f}_c, \quad \mathsf{M}\boldsymbol{u}_s = \boldsymbol{0}_c,$$

where the $\boldsymbol{p}_c \in \mathbb{R}^n$ and $\boldsymbol{u}_c \in \mathbb{R}^{3n}$ are the cell-centered pressure and velocity fields. For simplicity, $\boldsymbol{u}_c$ is defined as a column vector and arranged as $\boldsymbol{u}_c = (\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3)^T$, where $\boldsymbol{u}_i = ((u_i)_1, (u_i)_2, \ldots, (u_i)_n)^T$ are the vectors containing the velocity components corresponding to the x_i-spatial direction. The auxiliary discrete staggered velocity $\boldsymbol{u}_s = ((u_s)_1, (u_s)_2, \ldots, (u_s)_m)^T \in \mathbb{R}^m$ is related to the centered velocity field via a linear shift transformation (interpolation) $\Gamma_{c\rightarrow s} \in \mathbb{R}^{m\times 3n}$, $\boldsymbol{u}_s \equiv \Gamma_{c\rightarrow s}\boldsymbol{u}_c$. The dimensions of these vectors, n and m, are the number of control volumes and faces on the computational domain, respectively. The sub-indices c and s refer to whether the variables are cell-centered or staggered at the faces. The matrices $\Omega_c^{3d} \in \mathbb{R}^{3n\times 3n}$, $\mathsf{C}_c^{3d}\left(\boldsymbol{u}_s\right) \in \mathbb{R}^{3n\times 3n}$ and $\mathsf{D}_c^{3d} \in \mathbb{R}^{3n\times 3n}$ are block diagonal matrices given by

$$\Omega_c^{3d} = \mathsf{I}_3 \otimes \Omega_c, \qquad \mathsf{C}_c^{3d}\left(\boldsymbol{u}_s\right) = \mathsf{I}_3 \otimes \mathsf{C}_c\left(\boldsymbol{u}_s\right), \qquad \mathsf{D}_c^{3d} = \mathsf{I}_3 \otimes \mathsf{D}_c,$$

where $\mathsf{I}_3 \in \mathbb{R}^{3\times 3}$ is the identity matrix. $\mathsf{C}_c\left(\boldsymbol{u}_s\right) \in \mathbb{R}^{n\times n}$ and $\mathsf{D}_c \in \mathbb{R}^{n\times n}$ are the collocated convective and diffusive operators, respectively. The temporal evolution of the discrete temperature $\boldsymbol{\theta}_c \in \mathbb{R}^n$ in (2) is discretized in the same vein. For a detailed explanation of the spatial discretization, the reader is referred to [26]. Regarding the temporal discretization, a second-order explicit scheme is used for both the convective and the diffusive terms. Finally, the pressure-velocity coupling is solved by means of a classical fractional step projection method [27]: a predictor velocity, $\boldsymbol{u}_s^p$, is explicitly evaluated without considering the contribution of the pressure gradient. Then, by imposing the incompressibility constraint, $\mathsf{M}\boldsymbol{u}_s^{n+1} = \boldsymbol{0}_c$, it leads to a Poisson equation for $\tilde{\boldsymbol{p}}_c^{n+1}$ to be solved once each time-step,

$$\mathsf{L}\tilde{\boldsymbol{p}}_c^{n+1} = \mathsf{M}\boldsymbol{u}_s^p \qquad \text{with} \qquad \mathsf{L} = -\mathsf{M}\Omega_s^{-1}\mathsf{M}^T, \tag{4}$$

Algorithm 1 Time-integration step

1. Compute the convective, the diffusive and the source term of momentum Eq.(1): $\mathsf{R}(\boldsymbol{u}_s^n, \boldsymbol{u}_c^n, \boldsymbol{\theta}_c^n) \equiv -\mathsf{C}_c^{3d}(\boldsymbol{u}_s^n)\boldsymbol{u}_c^n - \mathsf{D}_c^{3d}\boldsymbol{u}_c^n + \boldsymbol{f}_c(\boldsymbol{\theta}_c^n)$
2. Compute the predictor velocity: $\boldsymbol{u}_c^p = \boldsymbol{u}_c^n + \Delta t\left\{\frac{3}{2}\mathsf{R}(\boldsymbol{u}_s^n, \boldsymbol{u}_c^n) - \frac{1}{2}\mathsf{R}\left(\boldsymbol{u}_s^{n-1}, \boldsymbol{u}_c^{n-1}\right)\right\}$
3. Solve the Poisson equation given in Eq.(4): $\mathsf{L}\tilde{\boldsymbol{p}}_c^{n+1} = \mathsf{M}\boldsymbol{u}_s^p$ where $\boldsymbol{u}_s^p = \Gamma_{c\to s}\boldsymbol{u}_c^p$
4. Correct the staggered velocity field: $\boldsymbol{u}_s^{n+1} = \boldsymbol{u}_s^p - \mathsf{G}\tilde{\boldsymbol{p}}_c^{n+1}$ where $\mathsf{G} = -\Omega_s^{-1}\mathsf{M}^T$
5. Correct the cell-centered velocity field: $\boldsymbol{u}_c^{n+1} = \boldsymbol{u}_c^p - \mathsf{G}_c\tilde{\boldsymbol{p}}_c^{n+1}$ where $\mathsf{G}_c = -\Gamma_{s\to c}\Omega_s^{-1}\mathsf{M}^T$
6. Compute the convective and the diffusive terms in temperature transport Eq.(2): $\mathsf{R}_\theta(\boldsymbol{u}_s^n, \boldsymbol{\theta}_c^n) \equiv -\mathsf{C}_c(\boldsymbol{u}_s^n)\boldsymbol{\theta}_c^n - Pr^{-1}\mathsf{D}_c\boldsymbol{\theta}_c^n$
7. Compute temperature at the next time-step: $\boldsymbol{\theta}_c^{n+1} = \boldsymbol{\theta}_c^n + \Delta t\left\{\frac{3}{2}\mathsf{R}_\theta(\boldsymbol{u}_s^n, \boldsymbol{\theta}_c^n) - \frac{1}{2}\mathsf{R}_\theta\left(\boldsymbol{u}_s^{n-1}, \boldsymbol{\theta}_c^{n-1}\right)\right\}$

where $\tilde{\boldsymbol{p}}_c = \Delta t\boldsymbol{p}_c$ and the discrete Laplacian operator, L, is represented by a symmetric negative semi-definite matrix. The particular choice of the Poisson solver will eventually depend on many factors such as the size of the problem, the computational architecture, the presence of periodic direction(s) or mesh symmetries [28]. Nevertheless, most of the existing sparse linear solvers algorithms rely on basic linear algebraic operations; therefore, they fit well the algebra-based HPC^2 framework.

In summary, the method is based on only five basic (linear) operators: the cell-centered and staggered control volumes, Ω_c and Ω_s, the matrix containing the face normal vectors, N_s, the cell-to-face scalar field interpolation, $\Pi_{c\to s}$ and the divergence operator, M. Once these operators are constructed, the rest follows straightforwardly. The algorithm to solve one time-integration step is outlined in Algorithm 1. Notice that the particular choice of the time-integration scheme is not relevant for the scope of this paper. Here, for the sake of simplicity, we have adopted a second-order Adams-Bashforth (steps 2 and 7 in Algorithm 1) although depending on the balance between convection and diffusion, schemes with a more appropriate stability region may be required [29].

At this point, it is noteworthy that, except for the non-linear convective term, $\mathsf{C}_c^{3d}(\boldsymbol{u}_s^n)\boldsymbol{u}_c^n$ and $\mathsf{C}_c(\boldsymbol{u}_s^n)\boldsymbol{\theta}_c^n$, all the operations directly correspond to sparse matrix-vector products (SpMV), most of them sharing the same matrix portrait. Regarding the convection (steps 1 and 6 in Algorithm 1), it can be reduced to an SpMV operation by simply noticing that the coefficients of the convective operator, $\mathsf{C}_c(\boldsymbol{u}_s^n)$, must be re-computed accordingly to the adopted numerical schemes [26]. However, it is rather common for many CFD applications to use different numerical schemes (*e.g.*, central difference, upwind, hybrid schemes,...) for each transport equation. In this case, different convective operators, $\mathsf{C}_c(\boldsymbol{u}_s)$, need to be recomputed at each time-step. Alternatively, the convective operator, $\mathsf{C}_c(\boldsymbol{u}_s)$, can be represented using more basic operators. Namely,

$$\mathsf{C}_c(\boldsymbol{u}_s) = \mathsf{M}\mathsf{U}\Pi_{c\to s}, \tag{5}$$

where $\mathsf{U} \equiv diag(\boldsymbol{u}_s) \in \mathbb{R}^{m\times m}$ is the diagonal arrangement of the face velocities,

$\boldsymbol{u}_s$, and $\Pi_{c \rightarrow s}$ is the above-mentioned cell-to-face scalar field interpolation. At first sight, computing the convective term using this form may look very inefficient since three consecutive SpMV are required. However, this naïve approach can be easily improved noticing that MU can be pre-computed since U is a diagonal matrix (that changes every time-step); therefore, the product MU is simply a re-scaling of columns. Moreover, this new matrix is shared by all the convective operators regardless the quantity that it is being advected. Finally, the cell-to-face interpolation operator, $\Pi_{c \rightarrow s}$, will depend on the particular choice for the spatial numerical scheme.

In conclusion, the overall Algorithm 1 relies on the set of three basic algebraic operations: an SpMV, a dot product and a linear combination of two vectors, the so-called AXPBY ($\mathbf{x} = \alpha \mathbf{x} + \beta \mathbf{y}$). Hereafter, we adopt an algebraic approach for the sake of code portability. Namely, the traditional stencil data structures and sweeps are replaced by algebraic data structures and kernels, and the discrete operators and mesh functions are then stored as sparse matrices and vectors, respectively.

3. Hierarchical parallel implementation model

Modern HPC systems consist of multiple hybrid nodes interconnected via a high-bandwidth network (Fig. 2). Hybrid nodes combine different architectures, such as manycore CPUs and GPUs, among others. Efficient distributed memory parallelization able to hide communication overhead behind computations is required for efficient use of an HPC cluster. We use the MPI at this level. Inside nodes, different devices require different parallel paradigms. On the one

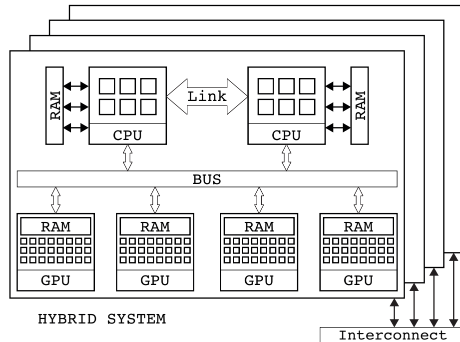

Figure 2: Hybrid HPC system diagram.

hand, the CPU side, the so-called host, consists of a pool of cores grouped into NUMA nodes (*i.e.*, different CPU sockets, or even clusters of cores inside each socket). Modern manycore CPUs integrate dozens of cores and allow for simultaneous multithreading, from 2 threads per core in Intel CPUs up to 8 in IBM ones. Fine-tuned shared memory parallelization is required to engage all the CPU cores. The NUMA factor must be taken into account to ensure the

correct affinity of threads and the proper placement of data in memory. We use the OpenMP standard at this level.

On the other hand, the massively-parallel devices of various architectures, also known as computing accelerators, integrate on-chip memory separated from the memory space of the host. Nowadays, the GPU is the most common kind of accelerator. The algorithms must be compatible with the stream processing paradigm to deal with such devices. We primarily use the OpenCL standard for various architectures, including GPU of AMD and NVIDIA. The multilevel MPI + OpenMP + OpenCL parallelization is sufficient for most of the existing HPC systems. Besides, our framework includes CUDA version of kernels to target systems based on IBM CPUs, which fully support the NVIDIA NVLINK interconnect (OpenCL drivers do not seem to work there).

3.1. Multilevel workload distribution

In a mesh method, the computational domain is decomposed in order to distribute the workload among computing devices. In our algebraic approach we deal with vectors and matrices representing mesh functions and discrete operators, respectively. The decomposition implies distributing vector elements and matrix rows among subdomains.

For a given subdomain, its vector elements and matrix rows form its *own* set. The portrait of a sparse matrix, $A \in \mathbb{R}^{n\times m}$, gives rise to other sets. A non-zero entry in the j-th column of the i-th row of a matrix, A_{ij}, defines a coupling between the i-th element of the output space and the j-th element of the input space. Thus, the *halo* set is composed of those elements from other subdomains, which are coupled with its own elements. Besides, the own set is further divided into two subsets. The *interface* set consists of those own elements that are coupled with halo elements. The remaining own elements form the *inner* set. In this distributed processing form, the vector elements are reordered locally as follows: first the inner set, then the interface set, then the halo (see Figure 3). The halo set is further reordered in ascending order of the owning subdomain numbers to simplify the processing of communications.

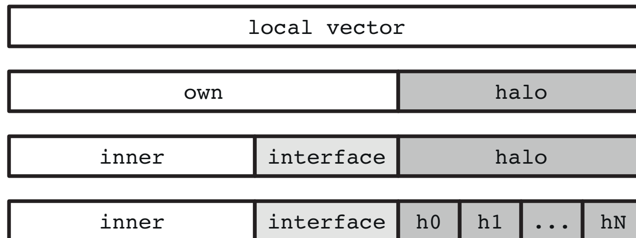

Figure 3: Ordering of vector elements according to their role in the parallel execution.

Let us consider now a parallel CFD simulation, as described in Section 2. In our algorithm, we have mesh functions defined in cells and faces. The mesh is represented by its connectivity graph, in which vertices correspond to mesh cells

and edges correspond to common faces between cells. In accordance with the terminology of graph theory, we construct the *adjacency* and *incidence* matrices, $A \in \mathbb{R}^{n\times n}$ and $E \in \mathbb{R}^{m\times n}$. The former represents the connectivity between adjacent mesh cells, and the latter represents the connectivity between faces and cells. Note that the matrices that correspond to discrete operators in our algorithm share their portraits either with the adjacency or incidence matrices.

The first-level partitioning, represented in Figure 4 by a gray contour, distributes the mesh cells among computing nodes (*i.e.*, labeled in Figure 4 as `0` and `1`). In doing so, the mesh is decomposed using a partitioning tool, such as the ParMETIS [30] library or a space-filling curve method [31], which fulfills the requested load balancing and minimizes the number of couplings between partitions. Then, the incidence matrix determines the corresponding partitioning for the mesh faces.

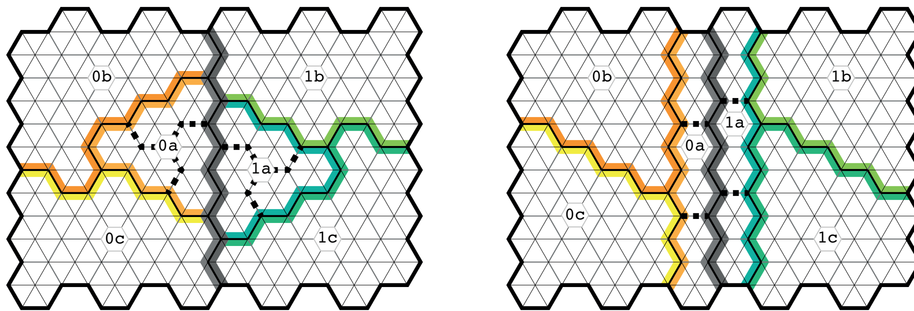

Figure 4: Representation of the multilevel domain decomposition: usual decomposition (left) to minimize the number of edgecuts, special decomposition (right) to isolate partitions `b` and `c`, the accelerators, from the inter-node exchanges. First-level subdomains for two cluster nodes are denoted as `0` and `1`. Each first-level subdomain is divided among a manycore CPU, `a`, and two accelerators, `b` and `c`. The second-level subdomains of the CPUs are split further among three NUMA nodes.

The second-level partitioning, represented in Figure 4 by colored contours, distributes the local workload of each hybrid node among the available devices (*i.e.*, labeled in Figure 4 as `a` for the CPU, `b` and `c` for the GPUs). This partitioning must conform to the actual performance of devices for the sake of load balancing. As a result, the interface and halo sets are further classified into *external* and *internal* categories according to whether they represent couplings between first- or second-level subdomains, respectively.

The third-level partitioning, represented in Figure 4 by dashed contours, distributes workload among CPU cores in a shared memory space (therefore, there is no interface and no halo at this level).

3.2. Overlap of computations and communications

The SpMV kernel needs the halo elements of the input vector to be updated; that is, the values of halo elements must be transferred from the owning subdomains. This update needs data exchanges between the cluster nodes and

Operations		Description
INN		Kernel execution for the inner set.
IFC		Kernel execution for the interface set.
UPD		Part of halo update with transfers inside nodes.
	DTH	Transfer interface data from devices to the host.
	COPY	Copy internal interface data to halo buffers of devices.
	HTD	Transfer halo data from host to devices.
MPI		MPI part of exchange for external interface and halo.
	PUT	Put external interface data of devices into MPI send buffers.
	MPI-I	Initialization by non-blocking MPI calls (MPI_Isend, MPI_Irecv).
	MPI-W	Wait external exchanges to finish (MPI_Waitall).
	GET	Get external halo data from MPI receive buffers in buffers of devices.

Table 1: Stages of the heterogeneous kernel execution combined with the halo update

the devices with separate memory space inside nodes. Such a costly multi-stage "device → host → MPI → host → device" communication is called halo update. The multilevel partitioning allows us to significantly reduce communication overhead by excluding the internal subdomain boundaries from network traffic. Furthermore, we use a communication-hiding approach, which consists in overlapping computations and communications. Since inner matrix rows need no halo, this allows us to perform computations for inner elements simultaneously with communications needed for halo elements only. Similarly, many other kernels of a numerical algorithm that need a halo update can be represented in a similar way if decomposed into inner and interface parts.

Finally, we can separate subdomains of devices from the external interface zone (see the right part of Figure 4). In this case, MPI exchanges can be separated, and all three components overlapped: inner computations, intra-node exchanges, inter-node MPI communications. The number of MPI processes is equal to the number of cluster nodes that are involved in a simulation. These processes distribute workload among various computing devices inside their nodes. Each process breaks apart into multiple OpenMP threads that control the execution on devices, perform data exchange and computations. Different strategies can be used for managing the intra-node workload. Consider a heterogeneous execution of a kernel in combination with a halo update in a general form. The notation in Table 1 is used hereinafter. In the baseline synchronous execution scheme, all the components are performed consecutively, and the total execution time is a sum of times per each component:

$$t_{sync} = t_{INN} + t_{UPD} + t_{MPI} + t_{IFC}.$$

Note that t_{INN} and t_{IFC} are the maximum times over the computing devices used.

In the overlapped execution, communications are performed simultaneously with INN computations. Hence, the execution time reduces to:

$$t_{ovl} = max(t_{INN}, t_{UPD} + t_{MPI}) + t_{IFC}.$$

If devices are isolated from external boundaries, as shown in Figure 4 (right),

the stages of the halo update related with MPI communications can be performed simultaneously with intra-node host-device exchanges. This can be called a double overlap:

$$t_{dovl} = max(t_{INN}, max(t_{UPD}, t_{MPI})) + t_{IFC}.$$

3.3. Multithreaded execution

In our previous implementation [32], we used nested OpenMP regions for distributing roles between groups of threads: device management threads, communication processing threads, computing threads. For instance, the computing thread spawned a nested region to compute in parallel CPU's workload. There were parallel dynamically scheduled loops inside those nested OpenMP computing regions. For that reason, firstly, there were problems with affinity because of the nested regions, and, secondly, we could not force the data locality due to dynamic scheduling.

In a multiprocessor node, access from one CPU to the memory on the controller of another CPU is notably slower, since the traffic goes via the interconnect of narrower bandwidth between the CPUs. Additionally, there is some internal NUMA factor inside a manycore CPU because its cores are not equally distant from its multiple memory controllers (*e.g.*, CPUs with ring topology, such as Intel Xeon of Broadwell microarchitecture with two rings of cores, each with its own memory controller, or Intel Xeon Scalable with mesh topology and two controllers at opposite sides of the mesh). Finally, in order to increase the core count, the CPUs are often made as a multi-chip package of, in fact, several separate CPUs, such as Intel Xeon Platinum 9200 series made of two CPUs of 8200 series, or 64-core AMD EPYC of 7700 series made of eight separate 8-core compute dies. Therefore, if we engage such CPUs by means of OpenMP from one MPI process, then we need to prevent migration of threads and take good care on data locality.

The new version of multithreaded execution avoids nested parallelism for computations. A flat fixed-size OpenMP region manages computing devices (second-level partitioning) of a hybrid node. The workload is distributed among CPU cores using third-level mesh partitioning instead of loop parallelism. The roles are assigned to OpenMP threads: computing threads and management threads. The management subgroup can be implemented either via OpenMP tasking or nested multithreading. Computing threads are properly bound to NUMA nodes, and data locality is granted by the first-touch rule at the initialization stage, since the data set of each thread now remains constant. This NUMA-aware parallelization has significantly improved performance on multiprocessor nodes.

Another improvement is related with the halo update part. Each of the stages DTH, PUT, COPY, GET, and HTD of the communication algorithm (see Table 1) is now performed in parallel by multiple threads. When host operates with multiple powerful GPUs, these copy operations appeared to produce notable overhead. The use of multithreaded processing has reduced this overhead several times. Note that the MPI exchanges occur in the background

simultaneously with the COPY stage. In order to ensure direct memory access transfers from and to devices simultaneously with computations on these devices, we use two OpenCL queues (two streams in case of CUDA) and pinned memory buffers mapped between the host and each device for the interface and halo data. The execution diagram for the overlap mode is shown in detail in Figure 5. The PUT, COPY and GET stages are handled by all the management threads. The double overlap mode is derived by splitting the management threads into inter-node and intra-node subgroups. In this case, the PUT and GET stages are processed by the inter-node threads, the COPY stage is handled by the intra-node ones (note the dashed lines that divide those boxes in Figure 5). Thus, inter-node and intra-node parts occur independently and simultaneously.

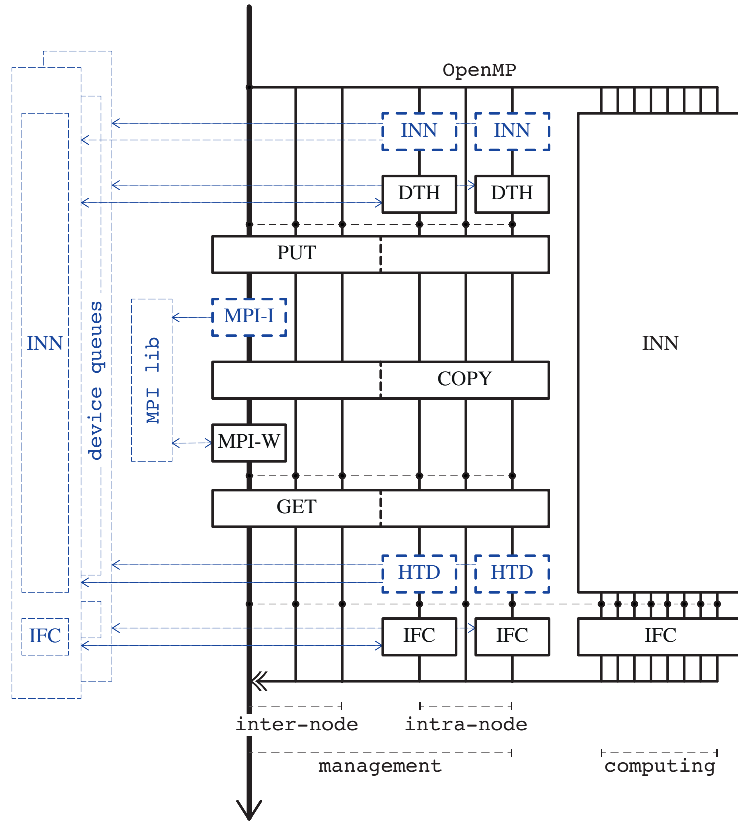

Figure 5: Execution diagram for the overlap mode using a flat OpenMP region (dashed boxes denote non-blocking calls).

4. Performance analysis

4.1. Testing conditions

In this section, an in-depth performance study of our basic kernels on several supercomputers is presented. The sparse matrices used in this study arise from the symmetry-preserving discretization [26] of the Laplacian operator on unstructured hex-dominant meshes. Therefore, the majority of rows contain seven non-zero coefficients. The ELLPACK sparse storage format and its block-transposed version are used for CPU and GPU computations, respectively (see [33] for details). This format provides a uniform, aligned memory access with coalescence of memory transactions on GPUs. The arithmetic intensity of this operation is about 1/8 FLOP per byte (7 products and 6 additions per row give 13 FLOP; 7 8-byte coefficients and 7 4-byte integer column indices per row, plus two real vectors of 8-byte values give us 100 bytes, assuming full reuse of the input vector values), therefore, it is a strongly memory-bound operation. In addition to the SpMV, we study the dot product kernel and the AXPBY kernel, which represents a linear combination of two vectors ($\mathbf{x} = \alpha\mathbf{x} + \beta\mathbf{y}$).

All the tests have been carried out using double-precision real values, and a large enough data set so that the data does not fit into the cache. The tests for all the kernels were repeated in a loop several hundred times in order to average the measurements.

4.2. Performance study on a CPU-based supercomputer

In this paper, the performance on multiprocessor nodes has been enhanced through a NUMA-aware multithreaded implementation, which takes care of data locality according to the first-touch rule and prevents migration of threads between NUMA nodes, as detailed in Section 3.

The benefits of the new version have been tested on MareNostrum 4 supercomputer at the Barcelona Supercomputing Center. Its nodes with two Intel Xeon 8160 CPUs (24 cores, 2.1 GHz, 6 DDR4-2666 memory channels, 128 GB/s memory bandwidth, 33 MB L3 cache) are interconnected through the Intel Omni-Path network (12.5 GB/s).

The single-node results for the SpMV in different parallel execution modes are shown in Figure 6 for a mesh of 17 million cells. The MPI mode, which leads to the most compact data placement, is not different from the OpenMP mode with NUMA placement and thread binding to cores (denoted as `init=thread, bind=core`). In this OpenMP mode threads are equally distributed among the two CPU sockets. The previous version of implementation (`init=master, bind=none`) with loop-based parallelism and without thread binding performs notably worse. Similarly, the new version with disabled NUMA placement (`init=master, bind=core`) is not far from the old one. The maximal performance obtained for the dot product, AXPBY and SpMV kernels in the new OpenMP mode is 29.66, 27.57 and 24.15 GFLOPS, respectively, which corresponds to 93%, 86% and 75% of the theoretically achievable performance (*i.e.*, the product of the arithmetic intensity by memory bandwidth, denoted

in Figure 6 as memory-bound). The old version of OpenMP parallelization reaches 27.65, 25.80 and 12.53 GFLOPS, respectively.

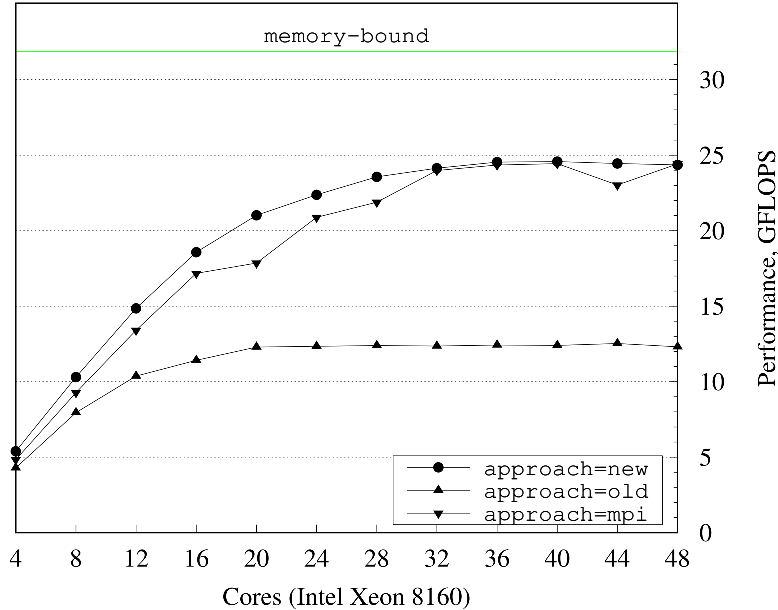

Figure 6: Comparison of different parallel modes of the *SpMV* kernel on a dual-processor node with two 24-core Intel Xeon 8160 CPUs. The matrix corresponds to an unstructured hex-dominant mesh of 17 million cells.

The strong and weak scaling results for the SpMV kernel on up to 9600 cores are shown in Figures 7 and 8, respectively. The OpenMP parallelization with NUMA placement and thread binding is used to compare both the overlap and the synchronous execution modes.

The results in Figure 7 show the speedup for two matrices derived from meshes of 17 million and 110 million cells. For the smaller matrix, parallelism runs out already on 6000 cores. The overlap mode significantly outperforms the synchronous version. For the bigger matrix, the parallel efficiency on 9600 cores for the overlap and synchronous modes is 97% and 65%, respectively. As the load per CPU goes down with the growing number of cores involved, a super-linear behavior of the speedup plot is observed due to the increase of cache reuse. However, as the workload becomes too small, the communication overhead becomes unavoidable.

The plot in Figure 8 shows the weak scaling in terms of relative performance compared to a single node for a constant workload of 17 million mesh cells per node. The performance of the overlap and synchronous modes drops to 96% and 67%, respectively, already at 384 cores due to the presence of communications. However, the communication overheads remain nearly constant until 9600 cores.

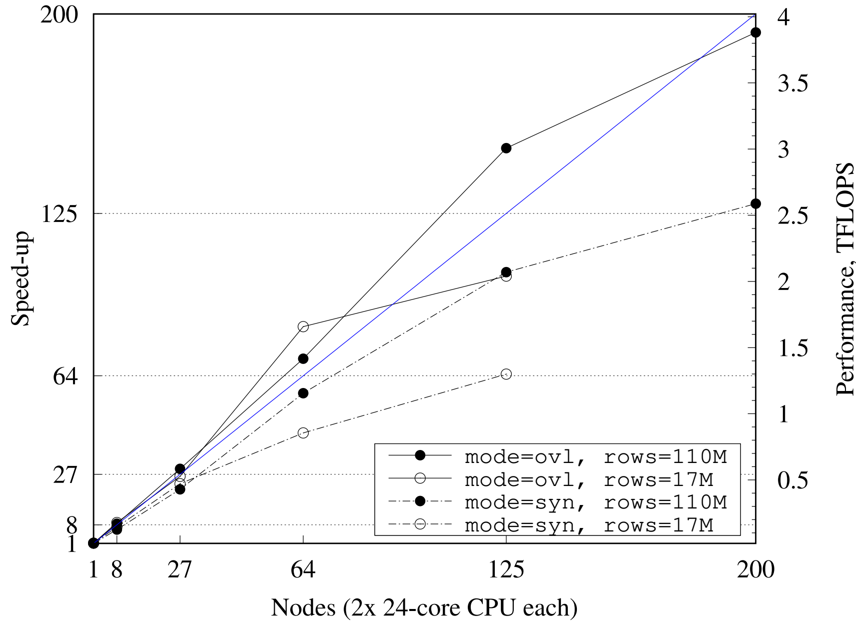

Figure 7: Speedup of the SpMV kernel in the overlap (ovl) and synchronous (syn) execution modes on MareNostrum 4 supercomputer for meshes of 17 million and 110 million cells

This plot shows very well that the overlap hides most of the communications.

4.3. Performance study on GPU-based supercomputers

The new implementation has been tested on the TSUBAME3.0 supercomputer at Tokyo Institute of Technology. Its nodes consist of two Intel Xeon E5-2680 v4 CPUs (14 cores, 2.4 GHz, 4 DDR4-2400 memory channels, 77 GB/s memory bandwidth, 35 MB L3 cache) and four Tesla P100 (16 GB HBM2, 732 GB/s memory bandwidth , PCIe 3.0 x16 – 16GB/s) interconnected via four Intel Omni-Path network cards (4×12.5 GB/s). In this system, the ratio of total memory bandwidth between the CPUs and the GPUs is around 1:20. It is worth mentioning that the CPU is responsible for handling immense amount of communications of 4 devices with a total memory bandwidth of 3 TB/s. Thus, we have neglected the CPU's computing power and studied the GPU-only mode.

The single-node performance results for SpMV are shown in Figure 9 for a mesh of 17 million cells. As it can be seen, computations are too fast to hide the halo update by the overlap. Sequential processing of communications is too slow, so that the resulting speedup is only $\times 2.2$ on 4 devices. The acceleration of intra-node interface–halo permutations between pinned buffers of devices by means of multithreaded parallelization has significantly improved performance, raising the speed-up to $\times 3.80$ and reaching 238 GFLOPS on 4 GPUs, which corresponds to 66% of theoretically achievable performance.

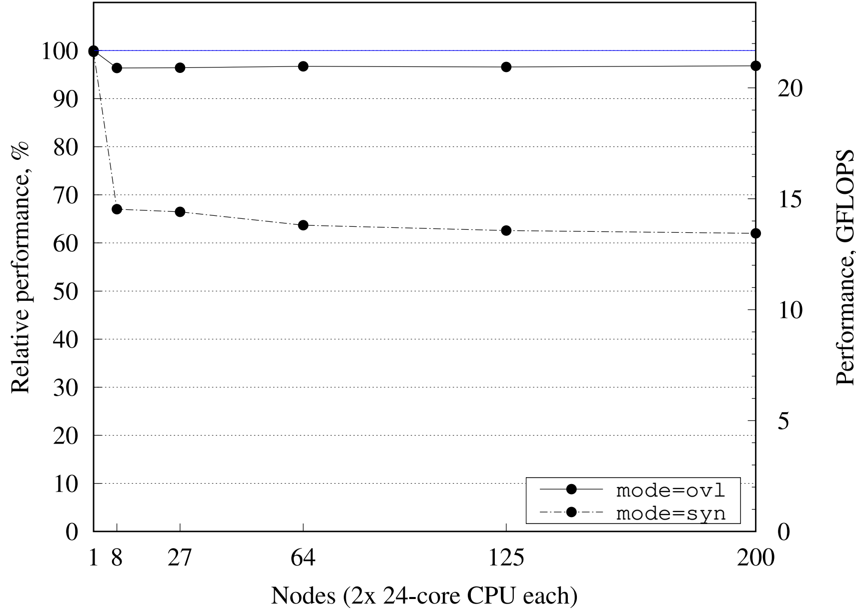

Figure 8: Weak scaling of the SpMV kernel in the overlap (ovl) and synchronous (syn) execution modes on MareNostrum 4 supercomputer in terms of relative performance compared to a single node for a constant workload of 17 million mesh cells per node.

From the problems with hiding intra-node exchanges, it is clear that with MPI communications between multiple nodes the situation will become worse. Indeed, the weak scaling results in Figure 10 confirm our expectations that computations are too fast to hide communications even for a big workload of 110 million cells per node. Weak scaling efficiency appeared to be 57% on 64 nodes in the overlap mode and about 37% in synchronous mode. This led to the estimation that communications take about 1.5 times longer than computations.

This fact motivated further study of communications. Additional tests with more detailed measurements have been performed on K-60 hybrid cluster: two Intel Xeon 6142 v4 (16 cores, 2.6 GHz, 6 DDR4-2666 memory channels, 128 GB/s memory bandwidth, 22 MB L3 cache) and four NVIDIA V100 GPU (32 GB HBM2, 900 GB/s, PCIe 3.0 x16 – 16GB/s) per node, interconnected with two InfiniBand FDR network cards (2×7 GB/s).

Timings for the SpMV kernel are shown in Table 2 for different mesh sizes, for the synchronous and overlap execution modes, sequential and parallel handling of communications. It can be seen that parallel processing of intra-node communications accelerates it up to $\times 4.5$. Overlapping communications and computations hides one of the two components nearly completely. Comparing the results for a mesh of 110 million cells on 8 nodes and a 14 million cells mesh on 1 node, we can conclude that the intra-node part takes about 1/3 (with

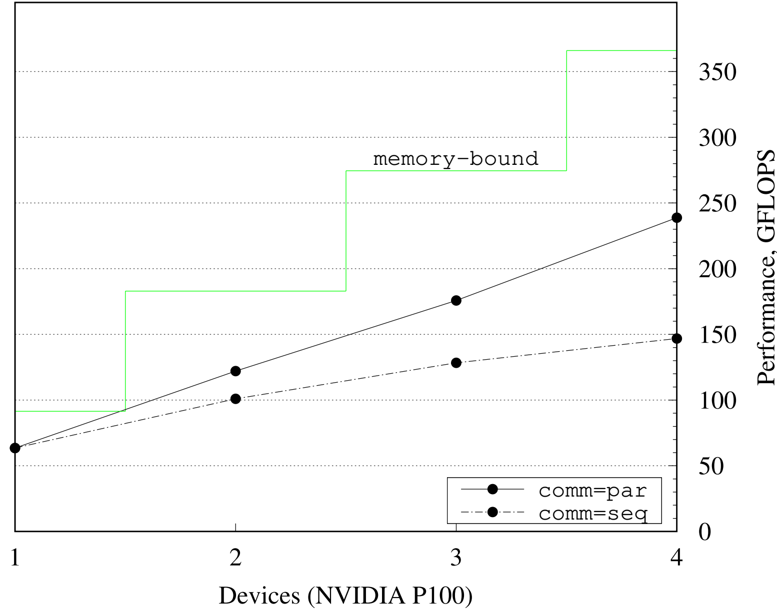

Figure 9: Comparison of sequential and multithreaded communication handling in the SpMV kernel execution on a node with four NVIDIA P100 GPUs. The sparse matrix corresponds to an unstructured hex-dominant mesh of 17 million cells.

parallel handling, of course), and the MPI transfers consume 2/3 of the overall communication overhead.

A closer look at communications is presented in Table 3. Detailed timings are given for the overlap execution mode with parallel handling of communications. The labels of stages correspond to Figure 5 in Section 3.3. Note that the COPY stage (memory copy between interface and halo buffers of devices inside nodes) is overlapped with the MPI communications. As we can see from these results, the MPI communications are the dominating overhead. Indeed, in the test on 8 nodes for 885 million cells, the throughput of the MPI communications is about 2.3 GB/s per node, which is below our expectations for the case of two 56G network cards per node (14 GB/s). Further improvements of MPI communications are needed, as well as NUMA-placement of the MPI buffers and buffers of devices.

4.4. Performance study on a heterogeneous supercomputer

The tests have been carried out on the Lomonosov-2 supercomputer at Lomonosov Moscow State University. Its hybrid nodes are equipped with one Intel Xeon E5-2697 v3 CPU (14 cores, 2.6 GHz, 4 DDR4-2133 memory channels, 68 GB/s memory bandwidth, 35 MB L3 cache) and one NVIDIA Tesla K40M GPU (12 GB of GDDR5 memory, 288 GB/s, PCIe 3.0 x16 – 16GB/s), and are interconnected via InfiniBand FDR network (7 GB/s).

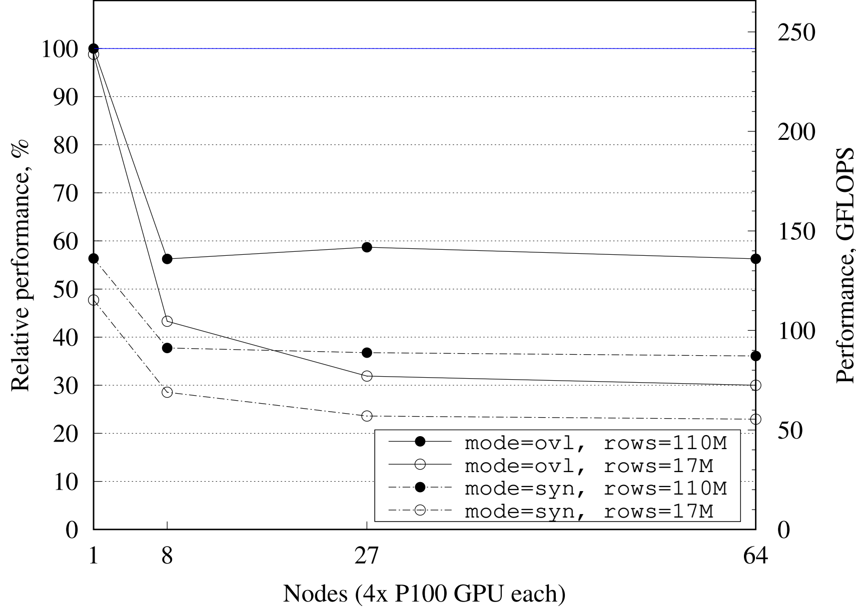

Figure 10: Weak scaling of the SpMV kernel in the overlap and synchronous execution modes on TSUBAME3.0 supercomputer in terms of relative performance compared to a single node for a constant workload of 17 and 110 million mesh cells per node.

Weak scaling results for a fixed workload of 2 million cells per hybrid node are shown in Figure 11. On a single node, sustained performance for the SpMV kernel is 6.9, 20.3, and 26.3 GFLOPS in CPU-only, GPU-only, and heterogeneous modes, respectively. On 8, 27 and 64 nodes, the heterogeneous mode outperformed GPU-only mode by a factor of $\times 1.3$ (wall clock time is about 1 ms), which corresponds to 98.5% of the sum of CPU-only and GPU-only performances. For the AXPBY and dot product kernels this factor is $\times 1.28$ and $\times 1.22$, respectively. These values are lower, because the workload was balanced for the SpMV kernel.

Strong speedup results for a mesh of 14 million cells are shown in Fig. 12. Results until 27 nodes demonstrate near linear speedup, and heterogeneous mode outperforms GPU-only mode by the factor of about $\times 1.29$. At 64 nodes, the situation changes a lot. Firstly, CPU-only mode demonstrates intensive super-linear speedup and becomes as fast as the GPU-only mode, since the data set fully fits in the L3 cache. This should have led to a much greater gain from the heterogeneous mode, but the gain on the contrary has decreased to $\times 1.04$. This can be explained by the fact that SpMV becomes too fast at that workload (around 200 thousand cells per node), about 0.2 ms, so the constant overheads from the intra-node decomposition outweigh the performance benefits.

4-GPU nodes	Mesh, million cells	Execution mode	Comm. mode	Comm. time, ms	Overall time, ms	GFLOPS
1	14	sync.	sequential	1.52	2.07	87
1	14	sync.	parallel	0.49	1.06	170
1	14	overlap	parallel	0.5	0.63	285
1	64	sync.	sequential	4.8	7.29	115
1	64	sync.	parallel	1.06	3.56	234
1	64	overlap	parallel	1.08	2.57	324
8	110	overlap	parallel	1.7	1.84	781
8	512	overlap	parallel	4.46	4.76	1398
8	885	overlap	parallel	7.81	7.98	1441

Table 2: Timings on K-60 cluster (four NVIDIA V100 per node) of the SpMV kernel for different mesh sizes, for the synchronous and overlap execution modes, sequential and parallel handling of communications

Stage	1 node time, ms	8 nodes time, ms
DTH (HTD)	0.51	1.5
PUT	–	0.74
COPY	0.71	0.68
MPI-W	–	4.2
GET	–	0.54
Overall	4.4	7.98

Table 3: Detailed timings of communications on K-60 cluster (four NVIDIA V100 per node) for a constant workload of 110 million cells per node.

5. Conclusions

In this work, a hierarchical parallel implementation for heterogeneous computing on hybrid supercomputers has been presented. Its application to algebra-based, scale-resolving modeling of incompressible turbulent flows with heat transfer has been described. It is noteworthy that this approach is not limited to a particular kind of numerical method or a set of governing equations.

OpenMP parallelization has been significantly improved over the previous version. The NUMA-aware implementation with proper thread binding and NUMA-placement of data has increased performance on dual-CPU nodes of the MareNostrum 4 supercomputer about $\times 1.8$ times. Parallel performance has been demonstrated on up to 9600 cores.

Furthermore, the multithreaded parallelization of memory copy operations has reduced the overhead of updating the halo. This upgrade is important when the computing workload is insufficient to fully hide the communications behind the computations using an overlapping approach. On hybrid nodes with multiple powerful GPUs, such as NVIDIA V100, hiding communications is not an easy task for a lightweight kernel. For instance, the SpMV kernel execution is too fast to hide the communications it needs, even using multithreaded processing. This fact has been demonstrated on the TSUMABE3.0 and K-60 hybrid

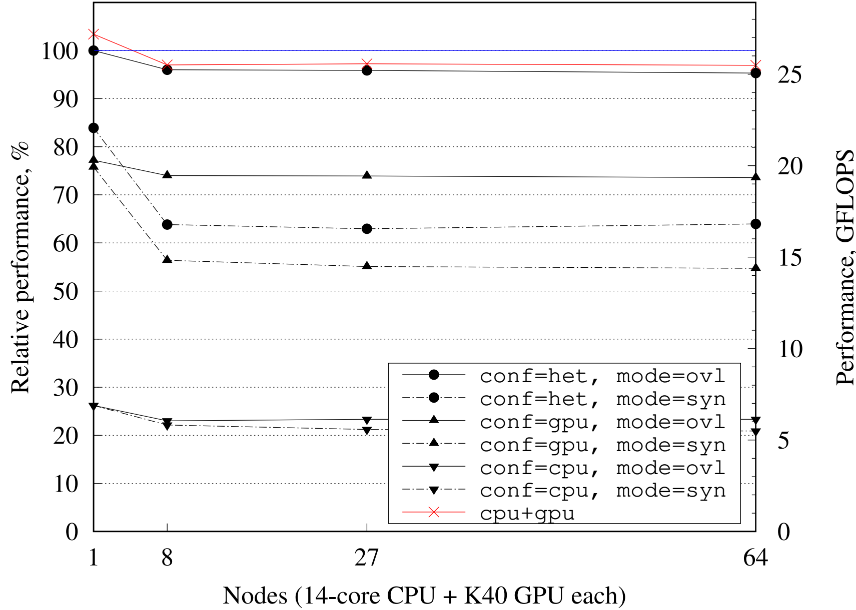

Figure 11: Weak scaling of the SpMV kernel in CPU-only, GPU-only and heterogeneous (het) modes in synchronous (syn) and overlapped (ovl) execution, constant workload of 2 million mesh cells per node. The red line denotes the sum of the CPU and GPU performance.

supercomputers, in which the host side is not fast enough managing communications of 4 devices. In this regard, we need to find further ways to reduce communication overheads.

Finally, the benefits of heterogeneous computing capability have been demonstrated on up to 64 hybrid nodes of the Lomonosov-2 supercomputer with a rather good weak scaling. The use of CPUs together with GPUs resulted in acceleration by a factor of about $\times 1.3$ compared to the GPU-only execution. It should be noted that the same execution pattern that we demonstrated on the example of the SpMV kernel can be used for various routines in numerical simulation codes that require a halo update and can be decomposed into inner and interface parts.

In our future work, we plan to focus on accelerating the communications (for instance, by NUMA-placement of exchange buffers of devices and MPI messages, binding of management threads, taking into account intra-node PCIe connection topology, etc.) and extending the applicability in CFD simulations.

Acknowledgments

The work of A. G. has been funded by the Russian Science Foundation, project 19-11-00299. The work of X. Á. F. and F. X. T. has been financially

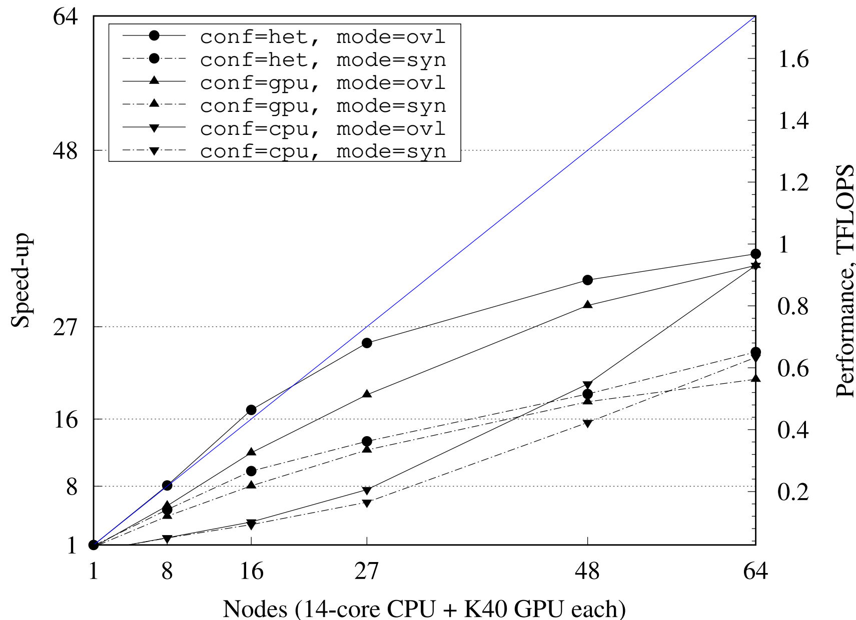

Figure 12: Speedup of the SpMV kernel in CPU-only, GPU-only and heterogeneous (het) modes in synchronous (syn) and overlapped (ovl) execution for a mesh of 14 million cells.

supported by a competitive R+D project (ENE2017-88697-R) by the Spanish Research Agency. X. Á. F. is supported by a predoctoral contract (2019FI_B2-00076) by the Government of Catalonia. The work has been carried out using the MareNostrum 4 supercomputer of the Barcelona Supercomputing Center; the TSUBAME3.0 supercomputer of the Global Scientific Information and Computing Center at Tokyo Institute of Technology; the Lomonosov-2 supercomputer of the shared research facilities of HPC computing resources at Lomonosov Moscow State University; the K-60 hybrid cluster of the collective use center of the Keldysh Institute of Applied Mathematics. The authors thankfully acknowledge these institutions.

References

[1] A. Yamanaka, T. Aoki, S. Ogawa, and T. Takaki, "GPU-accelerated phase-field simulation of dendritic solidification in a binary alloy," *Journal of Crystal Growth*, vol. 318, no. 1, pp. 40–45, 2011.

[2] P. Zaspel and M. Griebel, "Solving incompressible two-phase flows on multi-GPU clusters," *Computers & Fluids*, vol. 80, no. 1, pp. 356–364, 2013.

[3] A. Bocharov, N. Evstigneev, V. Petrovskiy, O. Ryabkov, and I. Teplyakov, "Implicit method for the solution of supersonic and hypersonic 3D flow

problems with Lower-Upper Symmetric-Gauss-Seidel preconditioner on multiple graphics processing units," *Journal of Computational Physics*, vol. 406, p. 109189, apr 2020.

[4] P. E. Vincent, F. Witherden, B. Vermeire, J. S. Park, and A. Iyer, "Towards Green Aviation with Python at Petascale," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, no. November, pp. 1–11, IEEE, nov 2016.

[5] H. T. Huynh, "A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods," in *18th AIAA Computational Fluid Dynamics Conference*, vol. 1, (Reston, Virigina), pp. 698–739, American Institute of Aeronautics and Astronautics, jun 2007.

[6] B. Krasnopolsky and A. Medvedev, "Acceleration of large scale openfoam simulations on distributed systems with multicore cpus and gpus," *Advances in Parallel Computing*, vol. 27, pp. 93 – 102, 2016.

[7] S. A. Soukov and A. V. Gorobets, "Heterogeneous Computing in Resource-Intensive CFD Simulations," *Doklady Mathematics*, vol. 98, pp. 472–474, sep 2018.

[8] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka, "Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 11, (New York, NY, USA), Association for Computing Machinery, 2011.

[9] F. D. Witherden, B. C. Vermeire, and P. E. Vincent, "Heterogeneous computing on mixed unstructured grids with PyFR," *Computers & Fluids*, vol. 120, pp. 173–186, oct 2015.

[10] W. Xue and C. J. Roy, "Heterogeneous Computing of CFD Applications on CPU-GPU Platforms using OpenACC Directives," in *AIAA Scitech 2020 Forum*, no. January, (Reston, Virginia), pp. 1–12, American Institute of Aeronautics and Astronautics, jan 2020.

[11] R. Borrell, D. Dosimont, M. Garcia-Gasulla, G. Houzeaux, O. Lehmkuhl, V. Mehta, H. Owen, M. Vzquez, and G. Oyarzun, "Heterogeneous cpu/gpu co-execution of cfd simulations on the power9 architecture: Application to airplane aerodynamics," *Future Generation Computer Systems*, vol. 107, pp. 31–48, 6 2020.

[12] X. Álvarez, A. Gorobets, F. X. Trias, R. Borrell, and G. Oyarzun, "HPC^2 – A fully-portable, algebra-based framework for heterogeneous computing. Application to CFD," *Computers & Fluids*, vol. 173, pp. 285–292, sep 2018.

[13] "cuSPARSE. The API reference guide for cuSPARSE, the CUDA sparse matrix library," *NVIDIA Corporation.*

[14] J. L. Greathouse, K. Knox, J. Pola, K. Varaganti, and M. Daga, "clSPARSE: A Vendor-Optimized Open-Source Sparse BLAS Library," in *IWOCL '16: Proceedings of the 4th International Workshop on OpenCL. Article No. 7.*, pp. 1–4, April 2016.

[15] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 781–792, IEEE, nov 2014.

[16] J. L. Greathouse and M. Daga, "Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 769–780, IEEE, nov 2014.

[17] W. Liu and B. Vinter, "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing - ICS '15*, (New York, New York, USA), pp. 339–350, ACM Press, mar 2015.

[18] W. Yang, K. Li, and K. Li, "A hybrid computing method of SpMV on CPU–GPU heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 104, pp. 49–60, jun 2017.

[19] G. Oyarzun, R. Borrell, A. Gorobets, F. Mantovani, and A. Oliva, "Efficient cfd code implementation for the arm-based mont-blanc architecture," *Future Generation Computer Systems*, vol. 79, pp. 786 – 796, 2018.

[20] N. R. et al., "The mont-blanc prototype: An alternative approach for hpc systems," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT*, pp. 444–455, Nov. 2016.

[21] N. Valle, X. Álvarez, F. X. Trias, J. Castro, and A. Oliva, "Algebraic implementation of a flux limiter for heterogeneous computing," in *Tenth International Conference on Computational Fluid Dynamics (ICCFD10)*, (Barcelona, Catalonia), July 2018.

[22] F. X. Trias, A. Gorobets, and A. Oliva, "A simple approach to discretize the viscous term with spatially varying (eddy-)viscosity," *Journal of Computational Physics*, vol. 253, pp. 405–417, 2013.

[23] N. Valle, F. X. Trias, and J. Castro, "An energy-preserving level set method for multiphase flows," *Journal of Computational Physics*, vol. 400, no. 1, p. 108991, 2020.

[24] F. Dabbagh, F. X. Trias, A. Gorobets, and A. Oliva, "On the evolution of flow topology in turbulent Rayleigh-Bénard convection," *Physics of Fluids*, vol. 28, p. 115105, 2016.

[25] L. Paniagua, O. Lehmkuhl, C. Oliet, and C. D. Pérez-Segarra, "Large eddy simulations (LES) on the flow and heat transfer in a wall-bounded pin matrix," *Numerical Heat Transfer, Part B: Fundamentals*, vol. 65, no. 2, pp. 103–128, 2014.

[26] F. X. Trias, O. Lehmkuhl, A. Oliva, C. D. Pérez-Segarra, and R. W. C. P. Verstappen, "Symmetry-preserving discretization of Navier–Stokes equations on collocated unstructured grids," *Journal of Computational Physics*, vol. 258, pp. 246–267, feb 2014.

[27] A. J. Chorin, "Numerical Solution of the Navier-Stokes Equations," *Mathematics of Computation*, vol. 22, pp. 745–762, 1968.

[28] A. Gorobets, F. X. Trias, M. Soria, and A. Oliva, "A scalable parallel Poisson solver for three-dimensional problems with one periodic direction," *Computers & Fluids*, vol. 39, pp. 525–538, 2010.

[29] F. X. Trias and O. Lehmkuhl, "A self-adaptive strategy for the time-integration of Navier-Stokes equations," *Numerical Heat Transfer, part B*, vol. 60, no. 2, pp. 116–134, 2011.

[30] D. Lasalle and G. Karypis, "Multi-threaded Graph Partitioning," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 225–236, may 2013.

[31] R. Borrell, J. Cajas, D. Mira, A. Taha, S. Koric, M. Vázquez, and G. Houzeaux, "Parallel mesh partitioning based on space filling curves," *Computers & Fluids*, vol. 173, pp. 264–272, sep 2018.

[32] X. lvarez, A. Gorobets, and F. Xavier Trias, "Strategies for the heterogeneous execution of large-scale simulations on hybrid supercomputers," in *Proceedings of the 6th European Conference on Computational Mechanics: Solids, Structures and Coupled Problems, ECCM 2018 and 7th European Conference on Computational Fluid Dynamics, ECFD 2018*, pp. 2021–2031, 2020.

[33] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva, "Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers," *International Journal of Computational Fluid Dynamics*, vol. 31, pp. 396–411, oct 2017.