UPC CTTC

# A hierarchical parallel implementation model for algebra-based CFD simulations on hybrid supercomputers

Centre Tecnològic de Transferència de Calor
Departament de Màquines i Motors Tèrmics
Universitat Politècnica de Catalunya

Xavier Álvarez Farré

Doctoral Thesis

# A hierarchical parallel implementation model for algebra-based CFD simulations on hybrid supercomputers

Xavier Álvarez Farré

TESI DOCTORAL

presentada al

Departament de Màquines i Motors Tèrmics
ESEIAAT
Universitat Politècnica de Catalunya

per a l'obtenció del grau de

Doctor per la UPC

Terrassa, May 12, 2022

# A hierarchical parallel implementation model for algebra-based CFD simulations on hybrid supercomputers

Xavier Álvarez Farré

**Director de la tesi**

Dr. Assensi Oliva Llena

**Co-directors de la Tesi**

Dr. Francesc Xavier Trias Miquel
Dr. Andrey Gorobets

**Tribunal Qualificador**

Dr. Antonio Pascau Benito
Universidad de Zaragoza

Dr. Charles Moulinec
Science and Technology Facilities Council

Dr. Carlos David Pérez Segarra
Universitat Politècnica de Catalunya

*A l'Elisabeth*

*Als meus pares*

# Agraïments

Podria dir que aquesta tesi té un inici ben clar, durant la primera meitat de l'any 2017. Tanmateix, la guspira que esdevingué passió per la recerca s'encengué molt temps abans.

En finalitzar el batxillerat, vaig matricular-me a Enginyeria Industrial amb un gran interès per la tecnologia en general, però sense cap entusiasme concret. Un bon dia, durant el tercer curs, assistí a una classe magistral del professor Assensi Oliva qui, durant dues hores guixant la pissarra, revisà la història de la física de fluids, des d'Arquimedes fins a Reynolds, en el que seria una ponència tant interessant i engrescadora com desordenada i divergent. Pocs minuts abans de concloure, el professor s'assegué sobre l'escriptori que hi havia a l'estrada i, amb un cert aire presumit, explicà que al seu laboratori s'hi estava desenvolupant un programari de càlcul numèric que resoldria les equacions de Navier–Stokes en un ordinador basat en processadors gràfics, de mida semblant a l'escriptori però més potent que deu mil ordinadors personals, i que serviria per donar resposta als problemes més complexos de l'enginyeria. Un instant més tard va sonar la campana i jo havia trobat, per fi, la raó per què volia començar aquest llarg viatge. Gràcies, Assensi, per la incessant empenta, passió i energia que transmets. Sense dubte has sigut un dels tres pilars del meu doctorat.

Al professor, company, i amic Francesc Xavier Trias, qui m'ha acompanyat durant tot el trajecte, vull dedicar-li un especial agraïment. Ell és qui millor representa els valors de la recerca, i a qui més admiro. Ha confiat en mi, m'ha fet prendre responsabilitats, i m'ha recolzat sempre. Amb ell he après molt. Es podrien comptar en milers els cafès d'idees que hem fet junts, i les hores que hem discutit i treballat. Però també hem gaudit i hem brindat cada repte que hem superat.

A tres mil quilòmetres de distància m'ha acompanyat l'Andrey Gorobets, un mestre de la computació científica que, a base d'hòsties (virtuals) i sempre amb el seu particular sentit del (mal) humor, m'ha ensenyat a ser crític i exigent amb mi mateix, també a qüestionar-me sense por, i a formular i respondre, amb contundència, la pregunta: *"what for?"* I tot i que procura camuflar-se rere una actitud desinteressada, amb ell he compartit les jornades de treball més llargues i intenses de tota aquesta aventura.

El pas pel CTTC ha sigut, en part, com un llarg viatge en tren. Cadascú amb un origen i destinació diferents però tots amb un camí en comú. Quan vaig començar, el tren ja anava ple de bona gent a qui vull dedicar unes paraules. Als professors Carles David Pérez-Segarra i Joaquim Rigola per la confiança dipositada en mi. Als companys Ricard Borrell i Guillermo Oyarzún, amb qui vaig publicar el meu primer article. Al mestre Arnau Pont, amb qui vaig compartir vagó durant molts anys i vaig poder gaudir inoblidables parades a Terrassa, Tàrrega o Barcelona. També a l'Enrique Gutiérrez, l'Eduard Bartrons, l'Alireza Naserí, el Jesús Ruano, la Roser Capdevila, el Joan Calafell, el Carles Oliet, el Dani Martínez, el Santiago Torras, o el Nicolás Ablanque; tots ells han sigut magnífics companys de tren i han fet d'aquest un viatge molt agradable. Als genis Guillem Colomer, Jordi Chiva i Ramiro Alba, que van resoldre tots els meus mentre m'endinsava en aquest món tan hostil que és la programació de baix nivell. A poc a poc els passatgers han anat canviant, però el tren segueix en marxa. Al *nouvingut* Àdel Alsalti, un gran matemàtic que m'ha fet suar amb les seves formulacions tan elegants com ininteligibles, un amic amb qui hem treballat de valent per fer els desenvolupaments més fascinants i potents del nostre codi numèric. Ara bé, si hi ha algú amb qui he estat durant tot aquest temps és la Nina Morozova, companya de dinars, cafès i cerveses; una amiga amb qui no hem perdut l'oportunitat de planejar una última parada a Vladivostok en diverses ocasions. Tant de bo! I què dir del maquinista d'aquest tren i mestre de cerimònies, l'Octavi Pavon, qui fa que tot funcioni bé tant en el plà professional com en el personal.

No puc acabar aquests agraïments sense mencionar al millor company de sala que hom podria desitjar. Un d'aquells companys de sala que a mitja tarda et demana cinc minuts i no saps com però no arribes a casa per sopar. I no pas per haver baixat al bar (que també), sinó perquè t'ha tornat a explicar, per enèsima vegada, la topologia algebraica des del conjunt fins a la cohomologia de De Rham i com tot això lliga amb les condicions de contorn. Gràcies, Nicolás Valle. Ets un grandíssim referent.

La recerca és un camí internacional i gràcies a això he pogut conèixer persones arreu. Els congressos per Europa m'han fet coincidir en diverses ocasions amb un home admirable, en Charles Moulinec, a qui vull agrair el suport i afecte que m'ha lliurat sempre que ens hem trobat, així com les oportunitats que m'ha brindat per ampliar el meu coneixement. *Cheers!* Per l'altra banda, una de les millors experiències, tant de recerca com vitals, la vaig viure al Japó. Durant aquell temps vaig poder gaudir experiències locals, naturals i divertides al costat d'un bon amic, en Yasutaka Kojima. Gràcies per enriquir i fer-me costat en una etapa tan bonica. També m'hi van acompanyar el Takeshi Kasuya, el Ken Ochiai, el Gerard Rovira i l'Uri Font; sense ells, l'estada no hagués sigut igual. I vull agrair especialment al professor Takayuki Aoki el privilegi d'haver pogut realitzar l'estada internacional al seu laboratori, a l'institut tecnològic de Tòquio. Allà vaig aprendre diferents formes de treballar i vaig enfrontar el meu treball a nous punts de vista, crítics però constructius, que sens dubte van fer-lo millorar. 青木先生、東京工業大学のゲスト研究員として先生の研究室にお迎えいただき、また、長らくのご指導ご鞭撻に感謝いたします。

De vegades els trens tornen a casa, i aquest no és una excepció.

Als meus amics, pels incomptables moments en què he necessitat un suport addicional i l'he trobat. I en especial al Javier Arribas, perquè ha valorat i estimat el meu treball com qui més, i perquè m'ha ajudat a tirar-lo endavant en inumerables ocasions.

A la meva família, pel seu suport incondicional. I en especial a la meva mare, pel seu amor i sacrifici infinits. Gràcies per donar-me la millor vida que podria desitjar.

A l'Elisabeth, per creure sempre en mi i donar-me forces quan no en trobava. Gràcies per fer-me, cada dia, una miqueta més feliç.

A tothom, gràcies de tot cor.

# Contents

# Abstract

Continuous enhancement in hardware technologies enables scientific computing to advance incessantly and reach further aims. Since the start of the global race for exascale high-performance computing (HPC), massively-parallel devices of various architectures have been incorporated into the newest supercomputers, leading to an increasing hybridization of HPC systems. In this context of accelerated innovation, software portability and efficiency become crucial.

Traditionally, scientific computing software development is based on calculations in iterative stencil loops over a discretized geometry—the mesh. Despite being intuitive and versatile, the interdependency between algorithms and their computational implementations in stencil applications usually results in a large number of subroutines and introduces an inevitable complexity when it comes to portability and sustainability. An alternative is to break the interdependency between algorithm and implementation to cast the calculations into a minimalist set of kernels.

The portable implementation model that is the object of this thesis is not restricted to a particular numerical method or problem. However, owing to the CTTC's long tradition in computational fluid dynamics (CFD) and without loss of generality, this work is targeted to solve transient CFD simulations. By casting discrete operators and mesh functions into (sparse) matrices and vectors, it is shown that all the calculations in a typical CFD algorithm boil down to the following basic linear algebra subroutines: the sparse matrix-vector product, the linear combination of vectors, and the dot product.

The proposed formulation eases the deployment of scientific computing software in massively parallel hybrid computing systems and is demonstrated in the large-scale, direct numerical simulation of transient turbulent flows.

# 1

# Introduction

Physics is the natural science that involves the study of matter and its motion through space and time. Engineering is the application of mathematics, empirical evidence, and scientific knowledge to research, improve or invent things. To describe a system or a phenomenon, scientists and engineers design and use mathematical models that are eventually solved on massively parallel supercomputers.

## 1.1 Brief historical review

This thesis is devoted to the modern computational challenges of solving complex fluid motion phenomena. Yet let me guide a journey through the history of the study of fluids first.

Mankind has been eager to understand nature since the dawn of civilization. For centuries, sometimes out of curiosity, sometimes out of necessity, people have observed the universe bumping into the questions and answers that build knowledge and science as we comprehend it today. Albeit many breakthroughs have recognized authorship, we must acknowledge that all understanding arises from a constantly evolving socio-cultural context, a stream of wisdom in which all humanity is involved, a flow of knowledge.

*Panta rhei*, or everything flows. This aphorism is credited to the ancient Greek philosopher Heraclitus of Ephesus (c. 535 BC – c. 475 BC). It characterizes his thoughts about the changing nature of things with the flow of time. Heraclitus

opposed the idea of a changeless *being*—a principle of his contemporary philosopher Parmenides of Elea (c. 515 BC – c. 470 BC)—claiming,

> *No man ever steps in the same river twice, for it's not the same river and he's not the same man.*

This duality between *becoming* and *being* influenced Plato (c. 427 BC – c. 347 BC) in formulating his theory of *Forms*. He distinguished two worlds or realms, the visible and the intelligible. In this way, *Forms*, or ideas, are the non-physical, absolute essences of things, the basis of science that must allow describing the world as we perceive it, which is subject to constant change.

Among many others, the great philosophers aforementioned founded the principles of thinking. However, their contributions were not recognized in a practical or scientific way. Plato's theory of *Forms* was harshly criticized by his student Aristotle (c. 384 BC – c. 322 BC). Contrary to Platonism, Aristotle developed an empirical philosophy where the experience was the source of knowledge and introduced the Aristotelian physics that would reign for eighteen centuries until the rise of Classical physics. Meanwhile, Archimedes of Syracuse (c. 287 BC – c. 212 BC)—considered the greatest mathematician of ancient history—introduced the field of hydrostatics and the notion of pressure. Legend has it that King Hiero II of Syracuse asked him to determine whether his golden crown was made of pure gold or alloyed with some other metal. *Eureka!* The question gave rise to discovering the principle of buoyancy and a method for determining the volume of an object with an irregular shape.

Centuries later, during the Renaissance, the formal ideas of fluid dynamics would be revisited in detail, ushered in the hand of Leonardo da Vinci (1452 – 1519). Albeit da Vinci did not conduct any remarkable mathematical studies on paper (probably due to a lack of analytical tools yet to come), he observed, studied, and comprehended fluid flow phenomena. Da Vinci experimented with and accurately illustrated waves, jets, or eddies. He was presumably the first to introduce the term *turbolenza* in a scientific sense describing two types of motion.

> *Observe the motion of the surface of the water, which resembles that of hair, which has two motions, of which one is caused by the weight of the hair, the other by the direction of the curls; thus, the water has eddying motions, one part of which is due to the principal current, the other to the random and reverse motion.*

**Figure 1.1:** Leonardo da Vinci's illustration of the swirling water flow. Credit: Leonardo da Vinci, Studies of Turbulent Water, RCIN 912660, Royal Collection Trust / © Her Majesty Queen Elizabeth II 2022.

Following the great scientists of the Renaissance, Sir Isaac Newton (1643 – 1727) culminated the scientific revolution with the publication of his work *Principia* [1]. His contributions, including the conservation of momentum, $\boldsymbol{F} = m\boldsymbol{a}$, or the concept of viscous flow, laid the foundation for the theoretical description of fluid flows and paved the way for the next two centuries. While Daniel Bernoulli (1700 – 1782) established a relationship between pressure and velocity based on simple energy principles, Leonhard Euler (1707 – 1783) formulated the equations of inviscid fluid flow, the Euler equations. Indeed, he is sometimes considered the father of fluid dynamics as a mathematical discipline.

Our journey ends with the Frenchman Claude Louis Marie Henry Navier (1785 – 1836) and the Irishman George Gabriel Stokes (1819 – 1903). They introduced the viscous terms in the nowadays well-known Navier–Stokes equations. These two-hundred-year-old differential equations, which include expressions for the conservation of mass, momentum, energy, or species, are the basis of the contemporary computational fluid dynamics (CFD) disciplines, that is, is the branch of fluid mechanics

that aims to solve fluid flow problems using numerical analysis. Indeed, Navier–Stokes equations are so difficult to solve—it is designated one of the seven Millenium Prize Problems, and its solution is worth 1 million dollars—that solutions for real flow problems were not feasible until the advent of modern digital computers.

## 1.2 Evolution of (high-performance) computer systems

Many engineering studies lead to mathematical models whose analytical resolution is unknown. The need to provide a practical solution to these problems promotes the research and development of numerical methods. Numerical analysis is the study of algorithms or numerical methods to obtain an approximate solution to complex mathematical models. The fact that numerical methods calculate an approximate solution to the problem implies that there is always an associated error. The two main factors related to the quality of the numerical analysis results are accuracy and precision. On the one hand, better accuracy is obtained using a better mathematical model. On the other hand, better precision is obtained using a better numerical method.

Interest in numerical analysis grew close together with the advent of computers. Nowadays, it is obvious what a computer looks like; however, before the emergence of digital computers, the term could refer to computer machines as devices used to aid computations or even to people who carried out calculations.

Lewis Fry Richardson (1881 – 1953), an English mathematician, physicist, and pacifist, was an early advocate of the approach to solving large-scale computing problems. He developed the first numerical weather prediction system based on dividing the physical domain into grid cells. His attempt to simulate weather for a single eight-hour period took six weeks of work and resulted in a dismal failure. Not deterred, Richardson described a fantasy weather forecast "*factory*" of 64,000 human computers working in "*a large hall like a theatre*" calculating the world's weather forecasts from meteorological data supplied by weather balloons spaced two hundred kilometers apart around the globe [2]. Using colored signal lights and telegraph communication, a leader in the center would coordinate the forecast.

> *Perhaps some day in the dim future it will be possible to advance the computations faster than the weather advances at a cost less than the saving to mankind due to the information gained. But that is a dream.*

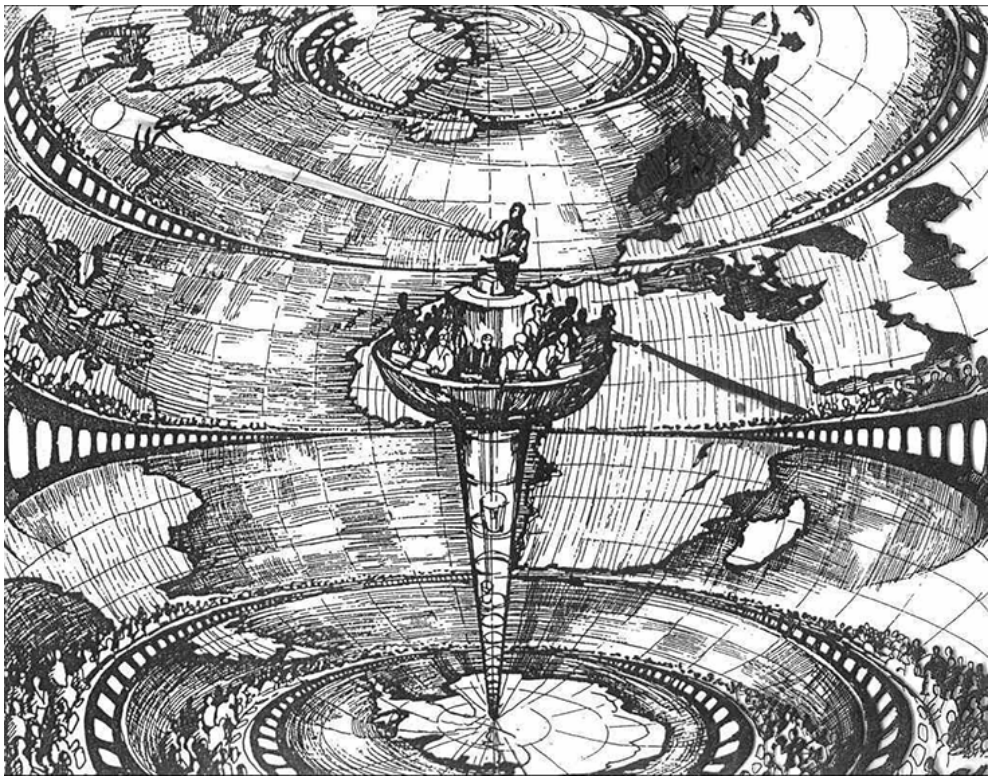

**Figure 1.2:** Illustration from "*Weather Prediction by Numerical Process*" by Lewis Fry Richardson, published in 1922 [2].

However, Richardson was proposing a very rudimentary CFD calculation. Along the same line, A. Thom reported the numerical solution to the flow past circular cylinders at low speeds in 1933 [3]. It is also worth mentioning the work by M. Kawaguti [4], who numerically solved the flow around a circular cylinder at $Re = 40$, working 20 hours per week for 18 months. It is clear that our colleagues from the early 19th century could hardly imagine that the use of digital computers would be so widespread a few years later. Nowadays, a smartphone would have solved Kawaguti's simulation in a fraction of a second.

The 1950s decade set a turning point in computer science after the development of the first computer languages, COBOL (1953) and FORTRAN (1954), the release of the first mass-produced computer in the world, IBM 650 (1954), or the Nobel Prize

worth unveiling of the integrated circuit (1958), among others events. Ever since, the interest in numerical disciplines such as CFD has increased exponentially.

Indeed, the emergence of digital computers capable of performing millions of simple binary operations per second boosted the use and development of numerical methods because together, they allow to quickly perform massive mathematical calculations. Computers reduce the need for analytical calculations or experimental results in the development of engineering projects such as thermal loads, fluid motion, and structural or financial calculations, among others. Moreover, such computations are increasingly fast and accurate thanks to the constant evolution of computers and numerical methods.

Gordon Moore, a co-founder of Intel, predicted in 1965 [5] that the number of transistors inside processors would double every year. Later on, this prediction was called *Moore's Law* by Caltech professor Carver Mead and became a self-fulfilled prophecy. For decades, frequency scaling and processor's core complexity were the dominant basis for progress in computer performance which, at that time, was the main bottleneck. Namely, an increase in frequency decreased the runtime of compute-bound programs. If this was insufficient, raw parallelism was used to engage multiple (equal) processors to solve a problem.

The main drawback of this constant growth is the energetic cost, which increases almost the same magnitude and constrains numerical calculations. Before a project is carried out, it is necessary to decide the mathematical model and numerical method depending on the desired accuracy and precision and assess whether the costs are worth it. Therefore, the efficiency of the code implemented for carrying out the simulation plays a key role: it is the only remaining factor to minimize the energetic cost of the computation.

The publication of Suhas V. Patankar's book [6] in 1980 was another key event in the CFD industry. It is probably the most influential book on CFD to date and the one that spawned most CFD codes. Hand in hand with the numerical studies and methods aforementioned, and recalling the compute-centric evolution in hardware and software technologies (it was considered the most precious resource on computer systems), the implementation of numerical simulation tools was naturally based on iterative stencil loops (ISL) over discretized geometries. In this way, complex codes and algorithms were designed to minimize the number of computations. However, the energy efficiency of data movement did not improve as fast as floating-point operations

(flop)'s energy efficiency [7].

> *Therein lies the problem with our overvaluation of preserving flops, because data movement is the dominant factor.*

To finish it off, a divergence in hardware architectures started back in 2004 when the increase in core's complexity and clock speeds reached a plateau, and therefore the demands for more computing power had to be met by other means. Rapidly, systems based on multicore processors or multi-socket configurations seized the top supercomputers list, adding additional levels of parallelism. The default message-passing interface (MPI) parallel models assuming data equidistance between processes ceased to be valid. Hybrid approaches combining MPI and open multi-processing (OpenMP) appeared in response, although they could easily end up delivering even worse performance on complex non-uniform memory access (NUMA) configurations. The divergence intensified in 2008 when devices of various architectures introducing completely different parallel paradigms came into play, such as many integrated cores (MICs) or graphics processing units (GPUs). Ever since the scientific computing community has been facing significant challenges.

## 1.3 Enabling scientific computing on modern supercomputers

Continuous enhancement in hardware technologies enables scientific computing to advance incessantly and reach further aims. Nowadays, the use of high-performance computing (HPC) systems is rather common in the solution of both industrial and academic scale problems. Many algorithms employed in scientific computing have a very low arithmetic intensity (AI), which is the ratio of computing work in flop to memory traffic in bytes. Hence numerical simulation codes are usually memory-bounded, making processors suffer from severe data starvation [7–9]. To top it off, the calculations often result in irregular, non-coalescing memory access patterns, reducing the memory access efficiency. Ironically, the memory bandwidth of computing hardware grows much slower than its peak performance, aggravating the problem. All this motivates the introduction of new parallel architectures with faster and more complex memory configurations into HPC systems.

To take advantage of the increasing variety of hardware architectures and the hybridization of HPC systems, the computing subroutines that form the algorithms, the

so-called kernels, must be adapted to complex paradigms such as distributed-memory (DM) and shared-memory (SM) multiple instruction, multiple data (MIMD) parallelism, and stream processing (SP). This encourages the demand for portable and sustainable implementations of scientific simulation codes [10]. While portability is an intangible characteristic of software (it cannot be measured nor qualified), it may be easy for a developer to know how difficult it is to rewrite, debug and verify a specific code on its adaptation to a new architecture. On the other hand, sustainability refers to developing reusable and resilient codes that last for long periods of time. The way a code is conceived at its inception enormously determines the degree to which both properties can be attained.

The use of GPUs in scientific computing is nowadays rather mature, and there are many successful examples in the literature [11–14]. For instance, the early GPU implementations in [15], extended in [16], proved to be two orders of magnitude faster than its central processing unit (CPU) counterpart. Moreover, the solution of two-phase flows on multi-GPU systems [17] was not only faster but also more energy-efficient. An example of a GPU porting of an open-source Navier–Stokes solver, the AFiD code, is found in [18]. Further examples of multi-GPU simulations of supersonic and hypersonic flows can be found in [19]. One of the most impressive GPU-based simulations is found in [20], after [21], on the solution of turbulent flows, reporting a sustained performance of 13.7Pflop/s.

Traditionally, scientific computing software development is based on calculations in ISL over a discretized geometry—the mesh. In this work, this implementation approach is referred to as *stencil* or *stencil-based*. Despite being intuitive and versatile, the interdependency between algorithms and their computational implementations in stencil applications usually results in a large number of subroutines and introduces an inevitable complexity when it comes to portability and sustainability [22].

The complexity of stencil applications motivates the adoption of conservative porting strategies, which consist of porting (rewriting) the most time-consuming part of an existing code, or even the entire code, to a new architecture but minimizing the structural modifications. In other words, it leads to a partial or complete reimplementation of an existing code. These strategies were common during the rise of general-purpose computing on GPUs because they allow for direct comparison studies of both numerical and performance results versus the legacy versions. Well-known commercial CFD codes and open-source platforms offer GPU extensions for solvers

of systems of linear algebra equations (SLAE), which represent a significant part of the overall computing time. This provides substantial acceleration with compactly localized changes in the code. Such an example can be found in [23], 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.

Implementing new physics or numerical methods in a stencil-based framework or its specialization for different mesh types usually requires the design of new computing subroutines and data structures. This is the main drawback of such an approach because the effort is not necessarily accumulative and thus reduces the software's sustainability. Some authors propose domain-specific tools to address this, generalizing the stencil computations for specific fields. For instance, a framework that automatically translates stencil functions written in C++ to both CPU and GPU codes is proposed in [24]. However, these generalizations are still heavily restricted by the shape of the stencil they target.

An alternative to stencil implementations is to break the aforementioned interdependency between algorithm and implementation so that the calculations are cast into a minimalist set of universal kernels. In other words, the idea is to use the classical ISL just for data building and leave the calculations to a reduced set of basic operations; in this way, legacy codes can be used and maintained indefinitely as preprocessing tools, and the calculation engines become easy to port and optimize.

By casting discrete operators and mesh functions into sparse matrices and vectors, it is shown that all the calculations in a typical CFD algorithm for the direct numerical simulation (DNS) and large-eddy simulation (LES) of incompressible turbulent flows boil down to the following basic linear algebra subroutines: sparse matrix-vector product (SpMV), linear combination of vectors (axpy) and dot product (dot) [25–28]. From now on, we refer to this implementation based on algebraic subroutines as *algebraic* or *algebra-based*. In this algebraic approach, the kernel code shrinks to dozens of lines; the portability becomes natural, and maintaining OpenMP, open computing language (OpenCL), or compute unified device architecture (CUDA) implementations takes minor effort. Besides, standard libraries optimized for particular architectures (*e.g.*, cuSPARSE [29], clSPARSE [30]) can be easily linked in addition to specialized in-house implementations. A similar approach is found in PyFR [21], where the majority of operations are cast in terms of matrix-matrix multiplications linking with appropriate BLAS libraries. In the context of the DNS, the preconditioned conjugate

gradient (CG) method following such an algebraic approach was implemented in [31], and its potential was exploited in [32] to perform petascale CFD simulations. Using an algebra-based formulation provided robust, portable, and optimized implementations in all cases. Consequently, the design of algebra-based algorithms for its use in massively parallel architectures seems a smart strategy for the efficient solution of both industrial and academic scale problems.

The rest of the work is organized as follows. Chapter 2 reviews the symmetry-preserving discretization of the Navier–Stokes equations to provide some context, and outlines an algebra-based algorithm for solving the governing equations. Chapter 3 describes the implementation model of the hierarchical parallel code for high-performance computing ($\text{HPC}^2$) framework in detail. Chapter 4 presents an exhaustive study of performance of the algebraic kernels implemented in $\text{HPC}^2$ on different computer systems. Chapter 5 demonstrates the capabilities of $\text{HPC}^2$ in dealing with large-scale CFD simulations. Finally, the conclusions and future work are outlined in Chapter 6.

## References

[1] S. Newton, *Philosophiæ Naturalis Principia Mathematica*. Londini, Jussu Societatis Regiæ ac Typis Josephi Streater. Prostat apud plures Bibliopolas, 1687.

[2] L. F. Richardson, *Weather Prediction by Numerical Process*. Cambridge Mathematical Library, Cambridge University Press, 2 ed., 1922.

[3] A. S. Thom, "The flow past circular cylinders at low speeds," *Proceedings of The Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 141, pp. 651–669, 1933.

[4] M. Kawaguti, "Numerical solution of the Navier–Stokes equations for the flow around a circular cylinder at Reynolds number 40," *Journal of the Physical Society of Japan*, vol. 8, no. 6, pp. 747–757, 1953.

[5] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, April 1965.

[6] S. V. Patankar, *Numerical Heat Transfer and Fluid Flow*. CRC Press, 1980.

[7] P. M. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the "new normal" for computer architecture," *Computing in Science Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[9] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.

[10] M. A. Farhan, A. Abdelfattah, S. Tomov, M. Gates, D. Sukkari, A. Haidar, R. Rosenberg, and J. Dongarra, "MAGMA templates for scalable linear algebra on emerging architectures," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 645–658, 2020.

[11] J. Romero, J. Crabill, J. E. Watkins, F. D. Witherden, and A. Jameson, "ZEFR: A GPU-accelerated high-order solver for compressible viscous flows using the flux reconstruction method," *Computer Physics Communications*, vol. 250, p. 107169, 2020.

[12] F. Jiang, K. Matsumura, J. Ohgi, and X. Chen, "A GPU-accelerated fluid–structure-interaction solver developed by coupling finite element and lattice Boltzmann methods," *Computer Physics Communications*, vol. 259, p. 107661, 2021.

[13] S. Watanabe and T. Aoki, "Large-scale flow simulations using lattice boltzmann method with AMR following free-surface on multiple GPUs," *Computer Physics Communications*, vol. 264, p. 107871, 2021.

[14] S. Ha, J. Park, and D. You, "A multi-GPU method for ADI-based fractional-step integration of incompressible Navier–Stokes equations," *Computer Physics Communications*, vol. 265, p. 107999, 2021.

[15] 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.

[16] S. Sakane, T. Takaki, M. Ohno, Y. Shibuta, and T. Aoki, "Acceleration of phase-field lattice Boltzmann simulation of dendrite growth with thermosolutal convection by the multi-GPUs parallel computation with multiple mesh and time step method," *Modelling and Simulation in Materials Science and Engineering*, vol. 27, p. 054004, jul 2019.

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

[18] X. Zhu, E. Phillips, V. Spandan, J. Donners, G. Ruetsch, J. Romero, R. Ostilla-Mónico, Y. Yang, D. Lohse, R. Verzicco, M. Fatica, and R. J. Stevens, "AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters," *Computer Physics Communications*, vol. 229, pp. 199–210, aug 2018.

[19] A. N. Bocharov, N. M. Evstigneev, P. V. Petrovskiy, O. I. Ryabkov, and I. O. 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.

[20] P. Vincent, F. Witherden, B. Vermeire, J. S. Park, and A. Iyer, "Towards green aviation with Python at petascale," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, (Salt Lake City), IEEE, nov 2016.

[21] F. Witherden, A. M. Farrington, and P. E. Vincent, "PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach," *Computer Physics Communications*, vol. 185, pp. 3028–3040, nov 2014.

[22] T. Gysi, T. Grosser, and T. Hoefler, "MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, (New York, NY, USA), pp. 177–186, ACM, jun 2015.

[23] B. I. 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.

[24] T. Shimokawabe, T. Aoki, and N. Onodera, "High-productivity framework for large-scale GPU/CPU stencil applications," *Procedia Computer Science*, vol. 80, pp. 1646–1657, 2016.

[25] 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, no. 9, pp. 396–411, 2017.

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

[27] X. Álvarez-Farré, A. Gorobets, and F. X. Trias, "A hierarchical parallel implementation for heterogeneous computing. Application to algebra-based CFD simulations on hybrid supercomputers," *Computers & Fluids*, vol. 214, p. 104768, 2021.

[28] N. Valle, X. Álvarez-Farré, A. Gorobets, J. Castro, A. Oliva, and F. X. Trias, "On the implementation of flux limiters in algebraic frameworks," *Computer Physics Communications*, vol. 271, p. 108230, 2022.

[29] "cuSPARSE: The API reference guide for cuSPARSE, the CUDA sparse matrix library," Tech. Rep. March, NVIDIA Corporation, 2020.

[30] J. L. Greathouse, K. Knox, J. Poła, 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*, (New York, NY, USA), ACM, apr 2016.

[31] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva, "MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner," *Computers & Fluids*, vol. 92, pp. 244–252, mar 2014.

[32] R. Borrell, J. Chiva, O. Lehmkuhl, G. Oyarzun, I. Rodríguez, and A. Oliva, "Optimising the Termofluids CFD code for petascale simulations," *International Journal of Computational Fluid Dynamics*, vol. 30, pp. 425–430, jul 2016.

# 2

# Mathematical framework

This chapter aims to describe the mathematical framework underlying the implementation of the HPC$^2$ framework. Although it is not restricted to a particular numerical method or physical problem, owing to the Centre Tecnològic de Transferència de Calor (CTTC)'s long tradition in CFD, and without loss of generality, this work is targeted to solve transient CFD simulations. By casting discrete operators and mesh functions into (sparse) matrices and vectors, it is shown that all the calculations in a typical CFD algorithm boil down to the following basic linear algebra subroutines: the sparse matrix-vector product, the linear combination of vectors, and the dot product. From now on, we refer to this implementation based on basic linear algebra subroutines as *algebraic* or *algebra-based*.

## 2.1 Algebraic formulation of discrete Navier–Stokes equations

Let us consider the numerical simulation of turbulent, incompressible flows of Newtonian fluids in the absence of external forces. Assuming constant physical properties, the dimensionless governing equations in primitive variables read

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{u} = \frac{1}{Re} \nabla^2 \boldsymbol{u} - \nabla p, \tag{2.1a}$$

$$\nabla \cdot \boldsymbol{u} = 0, \tag{2.1b}$$

where $Re$ is the dimensionless Reynolds number. The non-linear convective term is given by $(\boldsymbol{u} \cdot \nabla)\phi = \mathcal{C}(\boldsymbol{u}, \phi)$.

The essence of turbulence are the smallest scales of motion. They result from a subtle balance between convective transport and diffusive dissipation. Mathematically, these terms are governed by two differential operators differing in symmetry: 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.

In this section, we review the well-know conservative, symmetry-preserving, finite-volume discretization of Navier–Stokes equations on unstructured grids introduced by Trias *et al.* in [1], and introduce an algebra-based algorithm for the numerical simulation of turbulent flows. Thus, this work intends to lead to a generalization of Verstappen and Veldman [2] on unstructured grids. As it is shown, the algorithm relies on only three basic linear algebra subroutines. The underlying algebra-based formulation fits perfectly in the numerical simulation framework presented later, in Chapter 3.

### 2.1.1 Symmetry-preserving discretization on collocated grids

A collocated arrangement (see Figure 2.1) is preferred over staggered due to its more straightforward form for unstructured grids despite the intrinsic errors due to improper pressure gradient formulation [3–5]. Thus, both the pressure and velocities are stored at the center of the control volume, whereas a secondary, face-centered velocity field is defined to enforce mass conservation in the cells.

In a matrix-vector notation, the finite-volume discretization of the Navier–Stokes and continuity equations on an arbitrary collocated mesh reads:

$$\Omega_c^{3d}\frac{d\boldsymbol{u}_c}{dt} + \mathsf{C}_c^{3d}\left(\boldsymbol{u}_s\right)\boldsymbol{u}_c + \mathsf{D}_c^{3d}\boldsymbol{u}_c + \Omega_c^{3d}\mathsf{G}_c\boldsymbol{p}_c = \mathbf{0}_c, \tag{2.2a}$$

$$\mathsf{M}\boldsymbol{u}_s = \mathbf{0}_c, \tag{2.2b}$$

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

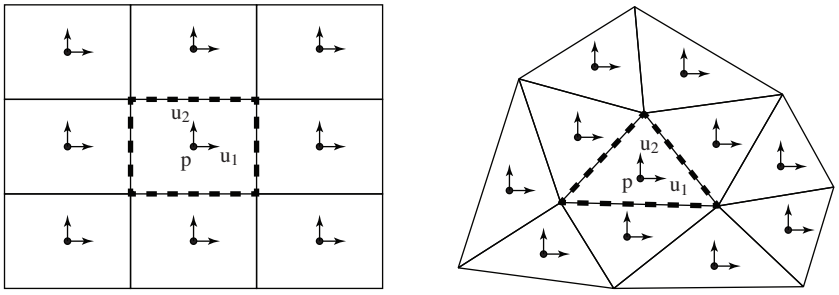


**Figure 2.1:** Variable arrangement for a collocated-mesh scheme.

$\boldsymbol{u}_i = ((u_i)_1, (u_i)_2, \ldots, (u_i)_n)^T$ 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 interpolation $\Gamma_{c\to s} \in \mathbb{R}^{m\times 3n}$, $\boldsymbol{u}_s \equiv \Gamma_{c\to s}\boldsymbol{u}_c$. 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 $\Omega_c^{3d} \in \mathbb{R}^{3n\times 3n}$, $\mathsf{C}_c^{3d}(\boldsymbol{u}_s) \in \mathbb{R}^{3n\times 3n}$ and $\mathsf{D}_c^{3d} \in \mathbb{R}^{3n\times 3n}$ are block diagonal matrices given by

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

where $\mathsf{I}_3 \in \mathbb{R}^{3\times 3}$ is the identity matrix and $\Omega_c \in \mathbb{R}^{n\times n}$ is a diagonal matrix with the cell-centered control volumes. $\mathsf{C}_c(\boldsymbol{u}_s) \in \mathbb{R}^{n\times n}$ and $\mathsf{D}_c \in \mathbb{R}^{n\times n}$ are the collocated convective and diffusive operators, respectively. Note the $\boldsymbol{u}_s$-dependence of the convective operator (non-linear operator). Finally, $\mathsf{G}_c \in \mathbb{R}^{3n\times n}$ represents the discrete gradient operator and the matrix $\mathsf{M} \in \mathbb{R}^{n\times m}$ is the face-to-center discrete divergence operator.

The conservative nature of the Navier–Stokes equations is intimately tied up with the symmetries of the differential operators (see [2, 6], for instance). This section reviews that retaining such symmetries leads to spatial discretizations that exactly conserve the total kinetic energy for inviscid flows. The forthcoming analysis does not consider the effect of external sources and is restricted to impervious or periodical boundary conditions. Following the same criterion as in [2], the discrete inner-product

reads

$$(\boldsymbol{v}_c, \boldsymbol{u}_c) = \boldsymbol{v}_c^T \Omega_c^{3d} \boldsymbol{u}_c, \tag{2.3}$$

then, the global discrete kinetic energy is given by $||\boldsymbol{u}_c||^2 \equiv \boldsymbol{u}_c^T \Omega_c^{3d} \boldsymbol{u}_c$ and its temporal evolution equation can be obtained by left-multiplying Equation 2.2a by $\boldsymbol{u}_c^T$ and summing the resulting expression with its transpose:

$$\begin{aligned} \frac{d}{dt}||\boldsymbol{u}_c||^2 &= -\boldsymbol{u}_c^T \left(\mathsf{C}_c^{3d}(\boldsymbol{u}_s) + (\mathsf{C}_c^{3d}(\boldsymbol{u}_s))^T\right) \boldsymbol{u}_c - \boldsymbol{u}_c^T \left(\mathsf{D}_c^{3d} + (\mathsf{D}_c^{3d})^T\right) \boldsymbol{u}_c \\ &\quad -\boldsymbol{u}_c^T \Omega_c^{3d} \mathsf{G}_c \boldsymbol{p}_c - \boldsymbol{p}_c^T \mathsf{G}_c^T (\Omega_c^{3d})^T \boldsymbol{u}_c. \end{aligned} \tag{2.4}$$

In absence of diffusion, that is $\mathsf{D} = \mathsf{0}$, the global kinetic energy $||\boldsymbol{u}_c||^2$ is conserved if both, the convective and pressure terms, vanish (for any $\boldsymbol{u}_c$, $\mathsf{M}\boldsymbol{u}_c = \mathbf{0}_c$) in the discrete kinetic energy equation,

$$\boldsymbol{u}_c^T \left(\mathsf{C}_c^{3d}(\boldsymbol{u}_s) + (\mathsf{C}_c^{3d}(\boldsymbol{u}_s))^T\right) \boldsymbol{u}_c = 0, \tag{2.5a}$$

$$\boldsymbol{u}_c^T \Omega_c^{3d} \mathsf{G}_c \boldsymbol{p}_c - \boldsymbol{p}_c^T \mathsf{G}_c^T (\Omega_c^{3d})^T \boldsymbol{u}_c = 0. \tag{2.5b}$$

The first equality is held if the discrete convective operator is skew-symmetric, whereas defining the negative transpose of the discrete gradient operator to be exactly equal to the divergence operator, *i.e.*,

$$\mathsf{C}_c^{3d}(\boldsymbol{u}_s) = -(\mathsf{C}_c^{3d}(\boldsymbol{u}_s))^T, \tag{2.6a}$$

$$-\left(\Omega_c^{3d} \mathsf{G}_c\right)^T = \mathsf{M}\Gamma_{c\to s}, \tag{2.6b}$$

guarantees that the contribution of the pressure term also vanishes. In this way, the skew-symmetry of the continuous operators is preserved.

A classical fractional step projection method [7–9] is used to solve the velocity-pressure coupling. For the staggered velocity field, $\boldsymbol{u}_s$, this projection is *naturally* derived from the Helmholtz-Hodge vector decomposition theorem [10], whereby a velocity $\boldsymbol{u}_s^p$ can be uniquely decomposed into a solenoidal vector, $\boldsymbol{u}_s^{n+1}$, and a curl-free vector, expressed as the gradient of a scalar field, $\mathsf{G}\tilde{\boldsymbol{p}_c}'$. This decomposition is written as

$$\boldsymbol{u}_s^p = \boldsymbol{u}_s^{n+1} + \mathsf{G}\tilde{\boldsymbol{p}_c}', \tag{2.7}$$

where $\mathsf{G} \in \mathbb{R}^{m\times n}$ is the center-to-face staggered gradient operator, which is related to

the divergence operator via

$$\mathsf{G} \equiv -\Omega_s^{-1}\mathsf{M}^T, \tag{2.8}$$

where $\Omega_s \in \mathbb{R}^{m\times m}$ is a diagonal matrix with the staggered control volumes. Then, taking the divergence of Equation (2.7) yields a discrete Poisson equation for $\tilde{\boldsymbol{p}}_c{}'$

$$\mathsf{M}\boldsymbol{u}_s^p = \mathsf{M}\boldsymbol{u}_s^{n+1} + \mathsf{M}\mathsf{G}\tilde{\boldsymbol{p}}_c{}' \quad \longrightarrow \quad \mathsf{M}\mathsf{G}\tilde{\boldsymbol{p}}_c{}' = \mathsf{M}\boldsymbol{u}_s^p. \tag{2.9}$$

Finally, using the definition of $\mathsf{G}$ given in Equation (2.8) the previous equation becomes

$$\mathsf{L}\tilde{\boldsymbol{p}}_c{}' = \mathsf{M}\boldsymbol{u}_s^p \qquad \text{with} \qquad \mathsf{L} \equiv -\mathsf{M}\Omega_s^{-1}\mathsf{M}^T, \tag{2.10}$$

where the discrete Laplacian operator $\mathsf{L} \in \mathbb{R}^{n\times n}$ is, by construction, a symmetric negative-definite matrix.

### 2.1.2 Constructing the discrete operators

The discretization of the operators preserving the global properties is outlined in this section (for a more detailed description of the spatial discretization and the operator properties, the reader is referred to [1]).

In general, the constraints imposed by the operator (skew-)symmetries strongly restrict the form of the local approximations limiting, in some cases, the local truncation error.

To prepare for the finite volume symmetry-preserving discretization, we recall the Reynolds transport theorem for a smooth function $\phi$:

$$\frac{d}{dt}\int_{[\Omega_c]_{k,k}} \phi dV = \int_{[\Omega_c]_{k,k}} \frac{\partial\phi}{\partial t} dV + \int_{\partial[\Omega_c]_{k,k}} \phi\boldsymbol{u}\cdot\boldsymbol{n} dS, \tag{2.11}$$

on an arbitrary centered cell $k$ of volume $[\Omega_c]_{k,k}$. Note that the function $\phi$ can have several physical meanings depending on what is being transported.

**Collocated convective operator**

The second term of the right-hand side (RHS) of Equation (2.11) can be exactly expressed as

$$\int_{\partial[\Omega_c]_{k,k}} \phi \boldsymbol{u} \cdot \boldsymbol{n} dS = \sum_{f \in F_f(k)} \int_{S_f} \phi \boldsymbol{u} \cdot \boldsymbol{n} dS, \tag{2.12}$$

where $F_f(k)$ is the set of faces bordering the cell $k$. Assuming that the discrete normal velocities, $[\boldsymbol{u}_s]_f \approx \boldsymbol{u}_f \cdot \boldsymbol{n}_f$, are located at the centroid of the face, then a second-order discretization of the integral (2.12) is given by

$$\int_{\partial[\Omega_c]_{k,k}} \phi \boldsymbol{u} \cdot \boldsymbol{n} dS \approx \sum_{f \in F_f(k)} \phi_f [\boldsymbol{u}_s]_f A_f, \tag{2.13}$$

where $A_f$ is the area of the face $f$. Hence, the collocated convective operator is defined by its action on an arbitrary cell-centered scalar field $\boldsymbol{\phi}_c \in \mathbb{R}^n$ at some cell $k$ as

$$\left[\mathsf{C}_c\left(\boldsymbol{u}_s\right)\boldsymbol{\phi}_c\right]_k = \sum_{f \in F_f(k)} \phi_f [\boldsymbol{u}_s]_f A_f. \tag{2.14}$$

The convective contribution to the global kinetic energy vanishes if the convective operator is skew-symmetric (Equation 2.6a). Such a condition is easily verified in two steps [2]. Firstly, we consider the off-diagonal elements. The matrix $\mathsf{C}_c\left(\boldsymbol{u}_s\right) - diag\left(\mathsf{C}_c\left(\boldsymbol{u}_s\right)\right)$ is skew-symmetric if the interpolation weights of the adjacent discrete variables are taken equal to $1/2$, hence independent of their spatial coordinates:

$$\phi_f \approx \left[\Pi_{c\to s}\boldsymbol{\phi}_c\right]_f = \frac{\phi_{c_1} + \phi_{c_2}}{2}, \tag{2.15}$$

where $c_1$ and $c_2$ are the cells adjacent to the face $f$ (see Figure 2.2, left) and $\Pi_{c\to s} \in \mathbb{R}^{m\times n}$ is the interpolation operator that interpolates a cell-centered scalar field to the faces. Then, for the skew-symmetry of the collocated convective operator, $\mathsf{C}_c\left(\boldsymbol{u}_s\right)$, the diagonal elements must be zero:

$$\left[\mathsf{C}_c\left(\boldsymbol{u}_s\right)\right]_{k,k} = \frac{1}{2} \sum_{f \in F_f(k)} [\boldsymbol{u}_s]_f A_f = 0. \tag{2.16}$$

In conclusion, the collocated convective operator $\mathsf{C}_c\left(\boldsymbol{u}_s\right)$ is skew-symmetric if the

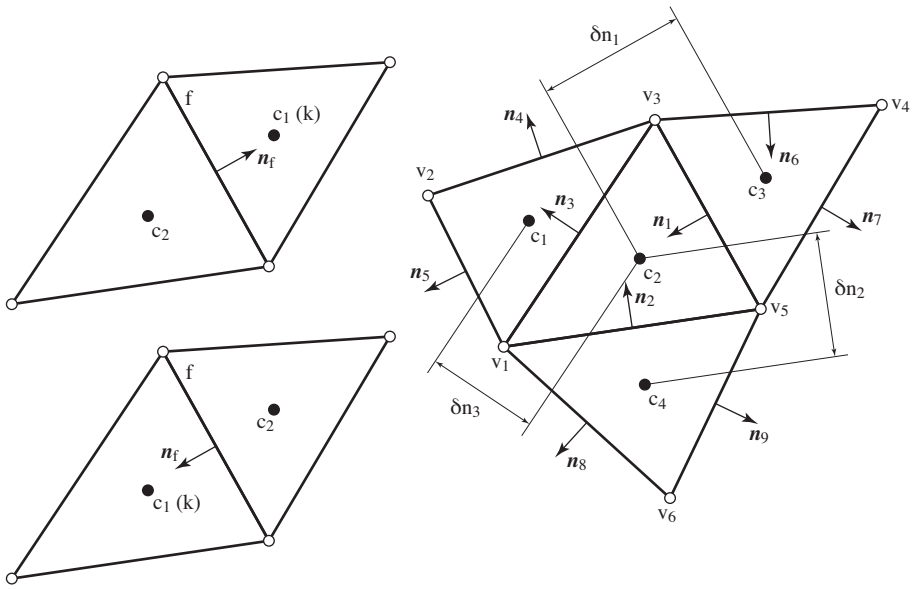


**Figure 2.2:** Left: face normal and neighbor labeling criteria. Right: definition of the volumes of the face-normal velocity cell.

discrete centered variable $\phi$ is interpolated to the faces of the control volumes using the rule defined in Equation (2.15), and the Equation (2.16) is accomplished. The following section shows that the latter condition holds if the divergence operator, $\mathsf{M}$, is consistently defined.

### Gradient, divergence and Laplacian operators

Integrating the continuity equation (2.1b) over an arbitrary centered cell $k$ of volume $[\Omega_c]_{k,k}$ yields

$$\int_{[\Omega_c]_{k,k}} \nabla \cdot \boldsymbol{u} dV = \int_{\partial[\Omega_c]_{k,k}} \boldsymbol{u} \cdot \boldsymbol{n} dS = \sum_{f \in F_f(k)} \int_{S_f} \boldsymbol{u} \cdot \boldsymbol{n} dS. \tag{2.17}$$

Note that taking $\phi$ equal to the unity, the Reynolds transport theorem (2.11) also gives the continuity equation in integral form. Therefore, Equation (2.13) is particularized

to define the properly integrated divergence operator,

$$[\mathsf{M}\boldsymbol{u}_s]_k = \sum_{f \in F_f(k)} [\boldsymbol{u}_s]_f A_f = 0. \tag{2.18}$$

Doing so, we are forcing the diagonal elements of the collocated convective operator to be equal to zero (see Equation 2.16). At this stage, it must be noted that the collocated convective operator defined in Equation (2.14) can be rewritten by using more basic operators as follows:

$$\mathsf{C}_c\left(\boldsymbol{u}_s\right)\boldsymbol{\phi}_c = \mathsf{M}\left(\mathrm{diag}\left(\Pi_{c\to s}\boldsymbol{\phi}_c\right)\boldsymbol{u}_s\right). \tag{2.19}$$

In Section 2.1.1, we have defined (see Equation 2.8) the integrated pressure gradient operator, $\Omega_s\mathsf{G}$, to be equal to the negative transpose of the divergence operator , $-\mathsf{M}^T$. Hence, the discretization of the pressure gradient at the face $f$ follows from Equation (2.18),

$$[\Omega_s\mathsf{G}\boldsymbol{p}_c]_f = \left(p_{c_1} - p_{c_2}\right)A_f, \tag{2.20}$$

where $c_1$ and $c_2$ are the cells adjacent to the face $f$ (see Figure 2.2, left). The order of accuracy of the discretization of the gradient operator defined in Equation (2.20) is, in general, $\mathcal{O}\left(1\right)$. Since the role of the pressure gradient is to project the velocity field into a divergence-free space for incompressible flows, this lack of accuracy becomes irrelevant in this context. Also, note that because the discrete gradient inherits the boundary conditions from the discrete divergence operator, we need not specify the pressure's boundary conditions. Finally, we compute the pressure from a Poisson equation, which arises from the incompressibility constraint. The Laplacian operator is approximated by the matrix

$$\mathsf{L} = -\mathsf{M}\Omega_s^{-1}\mathsf{M}^T, \tag{2.21}$$

which is symmetric and negative-definite, like the continuous Laplacian operator, $\nabla^2 \equiv \nabla\cdot\nabla$.

### Diffusive operator

Again, a diffusive operator is easily constructed on a collocated mesh. The same method for the discretization of the Laplacian operator is also applied. The diffusive operator as the product of two first-derivative-based operators: the divergence, $\mathsf{M}$, of a gradient, $\mathsf{G}$,

$$\mathsf{D}_c = -\frac{1}{Re}\mathsf{M}\mathsf{G} = \frac{1}{Re}\mathsf{M}\Omega_s^{-1}\mathsf{M}^T. \tag{2.22}$$

The Reynolds number has been introduced in order to simplify the forthcoming notation. Note that the collocated diffusive operator is, by definition (see Equation 2.22), symmetric and positive-definite, and its action on a cell-centered variable is given by

$$[\mathsf{D}_c\phi_c]_k = \frac{1}{Re}\sum_{f\in F_f(k)}\frac{(\phi_{c_2}-\phi_{c_1})\,A_f}{\delta n_f}, \tag{2.23}$$

where the length $\delta n_f$ is an approximation of the distance between the centroids of the cells $c_1$ and $c_2$ given by $\delta n_f = |\boldsymbol{n}_f \cdot \overrightarrow{c_1c_2}|$. Then, the volume of the face-normal velocity cell at the face $f$ is defined as $(\Omega_s)_f = \delta n_f A_f$ (see Figure 2.2, right).

### Interpolation operators

In a collocated formulation, the actual velocity is $\boldsymbol{u}_c$ and, therefore, linear interpolation operators are needed to relate the cell-centered velocity fields to the staggered ones and vice versa. Namely, the linear interpolation operator, $\Gamma_{c\to s} \in \mathbb{R}^{m\times 3n}$, transforms a cell-centered velocity field into a staggered one:

$$\boldsymbol{u}_s = \Gamma_{c\to s}\boldsymbol{u}_c, \tag{2.24}$$

whereas the cell-centered fields are related to the staggered ones via the linear interpolation $\Gamma_{s\to c} \in \mathbb{R}^{3n\times m}$,

$$\boldsymbol{u}_c = \Gamma_{s\to c}\boldsymbol{u}_s. \tag{2.25}$$

Notice that, in general, $\Gamma_{s\to c}\Gamma_{c\to s} = \mathsf{I}$ holds only approximately, *i.e.*, $\boldsymbol{u}_c \approx \Gamma_{s\to c}\Gamma_{c\to s}\boldsymbol{u}_c$. More importantly, recalling the definition for the staggered gradient operator given in

Equation 2.8, the cell-centered discrete gradient operator results in

$$\mathsf{G}_c = -\Gamma_{s\to c}\Omega_s^{-1}\mathsf{M}^T, \tag{2.26}$$

and, therefore, the face-to-cell interpolation operator $\Gamma_{s\to c}$ is restricted by Equation 2.6b:

$$\left(\Omega_c^{3d}\Gamma_{s\to c}\Omega_s^{-1}\mathsf{M}^T\right)^T = \mathsf{M}\Gamma_{c\to s} \longrightarrow \Gamma_{s\to c} = (\Omega_c^{3d})^{-1}\Gamma_{c\to s}^T\Omega_s, \tag{2.27}$$

to force that the pressure gradient contribution to the global kinetic energy exactly vanishes.

The linear interpolation, $\Gamma_{c\to s} \in \mathbb{R}^{m\times 3n}$, is given by

$$\left[\Gamma_{c\to s}\boldsymbol{u}_c\right]_f = \left[\mathsf{N}_s\left(\Pi\boldsymbol{u}_c\right)\right]_f = \frac{1}{2}\left(\left[\boldsymbol{u}_c\right]_{c1} + \left[\boldsymbol{u}_c\right]_{c2}\right)\cdot\boldsymbol{n}_f, \tag{2.28}$$

where matrices $\mathsf{N}_s \in \mathbb{R}^{m\times 3m}$ and $\Pi \in \mathbb{R}^{3m\times 3n}$ are respectively given by

$$\mathsf{N}_s = \begin{pmatrix} \mathsf{N}_{s,1} & \mathsf{N}_{s,2} & \mathsf{N}_{s,3} \end{pmatrix} \quad \text{and} \quad \Pi = \mathsf{I}_3 \otimes \Pi_{c\to s}, \tag{2.29}$$

where $\mathsf{N}_{s,i} \in \mathbb{R}^{m\times m}$ are diagonal matrices containing the $x_i$-spatial components of the face normal vectors, and $\Pi_{c\to s} \in \mathbb{R}^{m\times n}$ is the operator that interpolates a cell-centered scalar field to the faces defined in Equation (2.15). Finally, face-to-cell interpolation, $\Gamma_{s\to c} \in \mathbb{R}^{3n\times m}$, follows straightforwardly from Equation (2.27).

### 2.1.3 Algebra-based algorithm for the solution of Navier–Stokes equations

The algorithm to solve one time-integration step is outlined in Algorithm 1. Note that the particular choice of the time-integration scheme is not relevant to this chapter's scope. Here, for the sake of simplicity, we have adopted a second-order Adams-Bashforth (step 2 in Algorithm 1) although depending on the balance between convection and diffusion, schemes with a more appropriate stability region may be required [11]. Similarly, 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 [12]. Nevertheless, most existing sparse linear solvers algorithms rely on basic linear algebraic operations.

At this point, it is noted that except for the non-linear convective term, $\mathsf{C}_c^{3d}\left(\boldsymbol{u}_s^n\right)\boldsymbol{u}_c^n$,

**Algorithm 1** Algorithm to solve one time-integration step of Navier–Stokes equations.

1: Compute the convective and the diffusive terms of momentum Equation (2.2a):
$\mathsf{R}\left(\boldsymbol{u}_s^n, \boldsymbol{u}_c^n\right) \equiv -\mathsf{C}_c^{3d}\left(\boldsymbol{u}_s^n\right)\boldsymbol{u}_c^n - \mathsf{D}_c^{3d}\boldsymbol{u}_c^n$
2: Compute the predictor velocity:
$\boldsymbol{u}_c^p = \boldsymbol{u}_c^n + \Delta t\left\{\frac{3}{2}\mathsf{R}\left(\boldsymbol{u}_s^n, \boldsymbol{u}_c^n\right) - \frac{1}{2}\mathsf{R}\left(\boldsymbol{u}_s^{n-1}, \boldsymbol{u}_c^{n-1}\right)\right\}$
3: Solve the Poisson equation given in Equation (2.9):
$\mathsf{L}\tilde{\boldsymbol{p}}_c^{n+1} = \mathsf{M}\boldsymbol{u}_s^p$ 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 $\mathsf{G}_c = -\Gamma_{s\to c}\Omega_s^{-1}\mathsf{M}^T$

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

$$\mathsf{C}_c\left(\boldsymbol{u}_s\right) = \mathsf{MU}\Pi_{c\to s}, \tag{2.30}$$

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 $\Pi_{c\to s}$ is the above-mentioned cell-to-face scalar field interpolation. Computing the convective term using this form seems inefficient since three consecutive SpMV are required. However, this naive approach can be easily improved by noticing that $\mathsf{MU}$ can be precomputed since $\mathsf{U}$ is a diagonal matrix (that changes every time-step); therefore, the product $\mathsf{MU}$ is simply a re-scaling of columns. Moreover, this new matrix is shared by all the convective operators regardless of the quantity being advected. Finally, the cell-to-face interpolation operator, $\Pi_{c\to s}$, will depend on the particular choice for the spatial numerical scheme.

In summary, the method is based on only five basic (linear) operators: the cell-centered and staggered control volumes, $\Omega_c$ and $\Omega_s$, the matrix containing the face

normal vectors, $\mathsf{N}_s$, the cell-to-face scalar field interpolation, $\Pi_{c\to s}$ and the divergence operator, $\mathsf{M}$. Once these operators are constructed, all the calculations in a typical CFD algorithm boil down to the following basic linear algebra subroutines: the sparse matrix-vector product (SpMV), the linear combination of vectors (axpy), and the dot product (dot).

Hereafter, we adopt an algebraic implementation 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.

## References

[1] 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 meshes," *Journal of Computational Physics*, vol. 258, pp. 246–267, 2014.

[2] R. W. C. P. Verstappen and A. E. P. Veldman, "Symmetry-preserving discretization of turbulent flow," *Journal of Computational Physics*, vol. 187, pp. 343–368, 2003.

[3] Y. Morinishi, T. S. Lund, O. V. Vasilyev, and P. Moin, "Fully conservative higher order finite difference schemes for incompressible flow," *Journal of Computational Physics*, vol. 143, pp. 90–124, 1998.

[4] F. E. Ham and G. Iaccarino, "Energy conservation in collocated discretization schemes on unstructured meshes," *Center for Turbulence Research, Annual Research Briefs*, pp. 3–14, 2004.

[5] F. N. Felten and T. S. Lund, "Kinetic energy conservation issues associated with the collocated mesh scheme for incompressible flow," *Journal of Computational Physics*, vol. 215, pp. 465–484, 2006.

[6] U. Frisch, *Turbulence. The Legacy of A. N. Kolmogorov*. Cambridge University Press, 1995.

[7] A. J. Chorin, "Numerical solution of the Navier–Stokes equations," *Mathematics of Computation*, vol. 22, no. 104, pp. 745–762, 1968.

[8] N. N. Yanenko, *The Method of Fractional Steps*. Springer-Verlag, 1971.

[9] J. B. Perot, "An analysis of the fractional step method," *Journal of Computational Physics*, vol. 108, pp. 51–58, 1993.

[10] A. J. Chorin and J. E. Marsden, *A Mathematical Introduction to Fluid Mechanics*. Springer, third ed., 1993.

[11] F. X. Trias and O. Lehmkuhl, "A self-adaptive strategy for the time integration of Navier–Stokes equations," *Numerical Heat Transfer, Part B: Fundamentals*, vol. 60, pp. 116–134, 2011.

[12] 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.

# 3

# The HPC$^2$ framework

This chapter aims to describe the implementation model of the HPC$^2$, the fully-portable framework object of this thesis, designed to efficiently execute numerical simulations on hybrid supercomputers. Although it is committed to the algebraic formulation and computation of transient CFD simulations, the implementation model described here is not limited to these numerical methods. A hierarchical parallel implementation combined with a communication hiding approach is proposed to minimize the overhead of data exchanges in DM parallelism. A NUMA-aware SM parallelization is proposed to properly engage all cores in multi-socket configurations. Finally, two rather simple generalizations of the kernels that improve the capabilities the code are described.

## 3.1 Motivation

Let us assume there is a framework that provides us with discrete differential operators (matrices) and fields (vectors), without going into detail about the numerical method used to discretize. Matrices are provided in standard compressed sparse row (CSR) format, while vectors are given as one-dimensional arrays. No workload distribution or partitioning is regarded yet. Following the nomenclature introduced in Chapter 2, consider, for instance, the evaluation of the heat flux as

$$\boldsymbol{q}_s = -k\mathsf{G}\boldsymbol{T}_c. \tag{3.1}$$

Presuming that the discrete gradient operator and temperature field are given and, therefore, the size of both the input and output vector spaces are known, we want to write Equation 3.1 in our computer program as

```
1 q = - k*G*T;
```

where `T` is the discrete temperature field, built as an element of the vector space the gradient maps from, `G` is the discrete gradient operator, `k` is the thermal conductivity and `q` is the discrete heat flux, built as an element of the vector space the gradient maps to.

The HPC² project aims at computing any algebra-based model on, say, both a laptop and a hybrid supercomputer without changing any line of the code. Namely, it would be interesting for a research laboratory to execute a large-scale DNS of turbulent flow on a massively parallel supercomputer (*e.g.*, via a PRACE project on MareNostrum 4) and then to compute multiple overnight industrial simulations on different GPU-accelerated nodes. Thus, such a framework must rely on data structures and kernels that are appropriately managed at a lower-level code block, a black box that is simply configured through command-line parameters.

## 3.2 Abstract modeling of hybrid supercomputers

Before entering into detail, hybrid supercomputers' configuration and parameters that will directly impact code development and performance will be discussed from an abstract point of view. Thus, such systems will be considered black-boxes capable of performing the required calculations, regardless of the internal operations or instructions at a very hardware level.

### 3.2.1 Overview of hybrid supercomputers

In solving major challenges, scientific computing relies on HPC systems, also known as supercomputers. A quick look at the world's fastest supercomputers today [1] reveals the huge variety of hardware architectures and system configurations competing in the race for exascale computing.

In general, hybrid supercomputers consist of multiple nodes interconnected via a high-bandwidth network (see Figure 3.1). An efficient DM MIMD parallelization

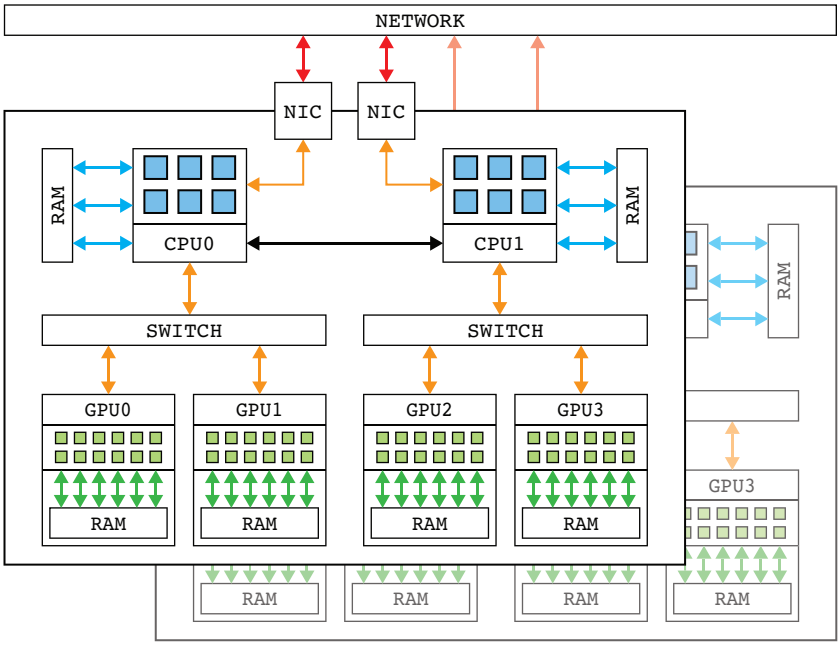


**Figure 3.1:** Example of a hybrid supercomputer consisting of two six-core CPUs and four GPUs.

capable of hiding the inter-node communication overhead is required to engage the nodes of an HPC system. The MPI standard is typically used at this level. In turn, hybrid nodes combine several computing units of various architectures that feature different parallel paradigms. Indeed, there are still many powerful *traditional* supercomputers based on single-processor nodes. However, these are considered just a particular case of the more general hybrid nodes described from now on.

The CPU, the so-called *host*, consists of a pool of cores packed into NUMA nodes: different CPU sockets, even groups of cores within sockets, with separate memory banks and controllers. The non-uniform memory access allows for faster access to local memory at the expense of slower access to remote memory. A fine-tuned, NUMA-aware SM MIMD parallelization is required to engage all the CPU cores, ensuring thread affinity and local memory access. Moreover, modern manycore CPUs integrate dozens of cores and allow for simultaneous multithreading (SMT), from two threads per core in Intel CPUs up to eight in IBM ones. In most cases, SMT is intended

for hiding memory latency, thus increasing the throughput delivered per core. The OpenMP interface is a common choice for multithreaded programming on CPUs, despite that it offers very little facility to express information about data locality or data movement. Special emphasis is also given to the increasingly larger vector registers such as advanced vector extensions (AVX), which introduce the single instruction, multiple data (SIMD) paradigm.

On the other hand, massively-parallel coprocessors, also known as *accelerators*, integrate an on-chip memory space separated from the host. Such devices are independent processors in their own right, although they are not intended for general-purpose programming. Indeed, coprocessors usually feature a limited instruction set focused on accelerating specific tasks and thus have to be driven by the host processor. For this reason, it is common to find the terms *master* and *slave* in the literature referring to such a workflow. Nowadays, the GPU is the most common accelerator, and algorithms must be compatible with the SP paradigm to deal with such processors efficiently. There are two common choices for implementing SP algorithms: the vendor-locked platform from NVIDIA, CUDA, and the open-source application programming interface (API) developed by Khronos, OpenCL. While the latter is a highly portable option compatible with virtually any hardware device, CUDA is geared towards NVIDIA GPUs only. However, in NVIDIA environments, it is usually preferred because it is easier to master and, in most cases, delivers higher performance.

Supercomputers' topology, the way computing units and nodes are interconnected, is also paramount in code development and decision making. In computer architecture, any communication system that transfers data between components inside a node or between nodes is called a bus. Different buses are used depending on which components connect, and their specifications can vary in orders of magnitude. Complex topologies introduce an important non-uniform input-output access (NUIOA) factor because I/O devices and accelerators are directly connected to single processor sockets, and hence accessing such devices from remote sockets results slower.

In conclusion, the system depicted in Figure 3.1 is, therefore, nothing but an example of a hybrid supercomputer. The number of computing units per node, their architecture, or the bus types used varies from one system to another, making the development and optimization of numerical simulation codes terribly cumbersome.

### 3.2.2 System parametrization

Broadly speaking, three hardware specifications are relevant in the performance analysis of numerical simulations: the maximum speed of calculations or peak performance, $\pi$, expressed in flop/s, the memory bandwidth, $\beta$, in bytes per second, and memory latency, $\lambda$, in seconds. Either could be the bottleneck of a numerical simulation, depending on the computational nature of the algorithm.

The peak performance is unique for every computing unit. It is obtained as the product of other primary specifications: the processor's clock speed, in Hertz, its number of physical cores, floating-point units (FPUs) per core and operations per unit, and the width of the vector units, in bits, divided by the size of the data, also in bits:

$$\mathrm{Hz} \cdot \mathrm{cores} \cdot \frac{\mathrm{units}}{\mathrm{core}} \cdot \frac{\mathrm{flop}}{\mathrm{unit}} \cdot \frac{\mathrm{bits}}{\mathrm{bits}} = \mathrm{flop/s}.$$

However, peak performance is not absolute but must be considered along with the algorithm that is executed. For instance, an algorithm performing only additions is, by nature, unable to use the fused multiply-add (FMA) unit that performs both operations simultaneously, and hence its maximum achievable performance is directly divided by two. Similarly, some algorithms cannot be vectorized and hence cannot use vector extensions such as AVX. Therefore, developers must be aware of the actual limits of an application to evaluate its efficiency on a specific computing unit properly.

In contrast, memory bandwidth and latency are not unique but depend on *who* is accessing *which* data, and *how*. For instance, to move a data set from a GPU to another computing node, data must travel from the source GPU to the source host, then the send buffer is processed by the host, and finally sent through the network. At best, all data traffic is directed through the shortest and fastest path or bus. However, it may happen that data is unnecessarily transferred through less efficient channels, especially in implementations that do not take NUMA and NUIOA factors into account.

### 3.2.3 Performance estimation

On the software side, three parameters are evaluated to estimate the maximum achievable performance of an algorithm: the number of operations, $W$, in flops, the memory traffic, $Q$, in bytes, and the data exchanges, $\boldsymbol{S}$ and $\boldsymbol{R}$, in bytes also. In

DM parallel programming, the latter are lists of message sizes that must be sent to and received from every computing unit participating in the execution. For instance,

$$\boldsymbol{S}_u = (S_1, ..., S_U)$$

is the list of message sizes that computing unit $u$ must send to all other computing units.

Scientific software developers long for well-balanced HPC systems to leverage hardware specifications for their applications. The harsh reality is that the achievable performance by most numerical algorithms or linear solvers is usually reduced to a tiny fraction of the peak performance, as evidenced by the HPCG benchmark [2]. Namely, the AI of a CFD code is typically in the order of 0.2 flop/byte, while the $\pi/\beta$ ratio in current computing units is in the order of 5 flop/byte. Ironically, this ratio has been growing since decades ago; and still grows.

At this point, we want to introduce some expressions that estimate the behavior of a hybrid computing system when dealing with a given algorithm or subroutine. Before doing so, let us outline some assumptions that are relevant for our developments hereinafter:

**Assumption 1.** *Kernels perform a vast number of independent operations, and therefore can be parallelized, vectorized and pipelined taking advantage of vector registers and instruction level parallelism (ILP).*

**Assumption 2.** *Kernels perform a vast number of independent, direct and unit-strided memory requests which can be parallelized and pipelined, and therefore memory latency can be neglected.*

**Assumption 3.** *The problem size is large enough to ignore the effects of temporal locality in cache memory, and therefore calculations require continuously accessing the computing unit's main memory.*

**Assumption 4.** *The implementation grants that all memory traffic ($Q$) required for computations is directed through the fastest path between a computing unit and its main memory, hence NUMA is not an issue.*

**Assumption 5.** *The implementation grants that all data exchanges ($\boldsymbol{S}$ and $\boldsymbol{R}$) required in parallel executions are directed through the fastest path between different computing units and nodes, hence NUIOA is not an issue.*

### Single device studies

Under these assumptions, the elapsed time of a kernel running on a single computing unit is estimated as follows:

$$t_k = min\left(\frac{W_k}{\pi_u}, \frac{Q_k}{\beta_u}\right), \tag{3.2}$$

where $\pi_u$ is the unit's peak performance, $\beta_u$ its main memory bandwidth, and $W_k$ and $Q_k$ are the kernel's work and memory traffic, respectively.

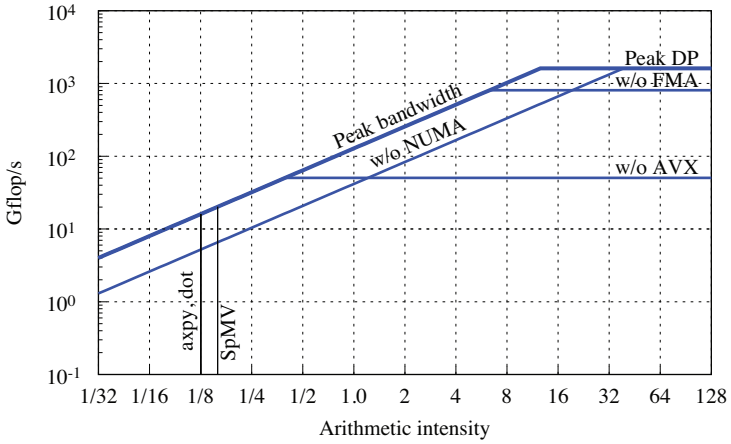


**Figure 3.2:** Example of a roofline model for an Intel Xeon 8160 processor.

Sometimes efficiency is more relevant than elapsed time itself in performance evaluation. In this regard, the roofline model [3] is used to estimate the maximum achievable performance of an algorithm running on single computing units as follows:

$$\pi_k = min(\pi_u, AI_k \beta_u), \tag{3.3}$$

where $AI_k$ is the kernel's arithmetic intensity, that is, the ratio of work to memory traffic: $AI_k = W_k/Q_k$.

The roofline is shown in Figure 3.2. It allows to visually identify the bottleneck of an algorithm and easily evaluate its efficiency by dividing the measured performance by the maximum achievable, $\pi_k$. It is straightly observed that algebraic kernels are

strongly memory-bound (and hence Equation 3.2 reduces to $t_k = Q_k/\beta_u$). However, it does not provide any information regarding DM parallelization.

### Multiple device studies

Given a hybrid supercomputer consisting of $N$ compute nodes and $U$ computing units or devices, the following expression evaluates the overhead in seconds introduced by data exchanges in DM configurations (per device):

$$t_x = \sum_{u' \in U} \left( \frac{S_{u,u'}}{\beta_{u,h}} + 3\frac{S_{u,u'}}{\beta_h} + \frac{R_{u,u'}}{\beta_x} + 3\frac{R_{u,u'}}{\beta_h} + \frac{R_{u,u'}}{\beta_{h,u}} \right), \tag{3.4}$$

where $u$ and $u'$ are the source and destination computing units, respectively. There are five distinct terms in Equation 3.4. The first term evaluates the time required for copying the data in $u$ to be sent to $u'$, $S_{u,u'}$, through a device-to-host link of bandwidth $\beta_{u,h}$. The second term estimates the time required for packing the sending buffer by means of a memory mapping or reordering. This reordering is handled by the host processor and implies reading and writing the entire buffer and a pair of integer lists or a map; hence its weight is 3. Next, the time required for receiving the data in $u'$ required by $u$ through the network of bandwidth $\beta_x$. Finally, the fourth and fifth terms are equivalent to the second and first (unpacking receive buffers on host and copying to device), respectively.

Equation 3.4 applies to all kinds of data exchanges. However, in each particular case, some terms might be canceled. For instance, in a CPU-based supercomputer, the first and fifth terms are canceled because $S_{u,u'}$ and $R_{u,u'}$ are already located in the appropriate memory space. On the other hand, intra-node data exchanges within a node with multiple accelerators do not involve the third and fourth terms. It is noted that the intra-node packing of sending buffer results in the unpacked receiving buffer and can be directly copied to the accelerator.

In CFD simulations, it is common that *halo*, or data exchanges, are symmetric: $S_{u,u'} = R_{u,u'} = X_{u,u'}$. Then, let $U_n$ be the subset of units in node $n$, thus $X_u^i = \sum_{u' \in U_n} X_{u,u'}$ and $X_u^e = \sum_{u' \in U \setminus U_n} X_{u,u'}$ are the aggregated number of bytes that computing unit $u$ exchanges with intra- and inter-node computing units $u'$, respectively. Considering that device-to-host link buses are also symmetric, $\beta_{u,h} = \beta_{h,u}$, Equation

3.4 is rewritten as follows:

$$t_x = X_u^i \left( \frac{2}{\beta_{u,h}} + \frac{3}{\beta_h} \right) + X_u^e \left( \frac{2}{\beta_{u,h}} + \frac{6}{\beta_h} + \frac{1}{\beta_x} \right). \tag{3.5}$$

Now, we define $X_u = X_u^i + X_u^e$ as the total number of bytes that computing unit $u$ exchanges, and $\chi_u = X_u^e / X_u$ as the ratio of inter-node to total data exchanges, and reformulate:

$$t_x = X_u \left( \frac{2}{\beta_{u,h}} + \frac{3 + 3\chi_u}{\beta_h} + \frac{\chi_u}{\beta_x} \right). \tag{3.6}$$

In DM parallelization, Equation 3.6 allows estimating the overhead introduced by data exchanges and predicting the behavior of an application in a parallel environment. More precisely, the theoretical parallel efficiency is estimated as the ratio $t_k/(t_k + t_x)$ and, in the particular case where an application can handle kernel execution and data exchanges simultaneously, as $t_k/max(t_k, t_x)$. It is worth formulating an expression that estimates some theoretical maximum ratio of computing to communicating work that allows an adequate overlap dividing the memory-bound form of Equation 3.2 by 3.6:

$$\frac{t_k}{t_x} = \frac{Q_k}{X_u} \cdot \frac{\dfrac{1}{\beta_u}}{\dfrac{2}{\beta_{u,h}} + \dfrac{3 + 3\chi_u}{\beta_h} + \dfrac{\chi_u}{\beta_x}} \geq 1. \tag{3.7}$$

All expressions above evaluate the behavior of parallel processes handling single computing units. In our approach, we are also interested in the overall behavior of hybrid nodes in parallel environments. Considering a memory-bound application, the kernel execution time per node becomes:

$$t_k = \frac{Q_k}{\beta_n}, \tag{3.8}$$

where $\beta_n = \sum_{u \in U_n} \beta_u$ represents the aggregated (main) memory bandwidth per node. It is noted that having each computing unit its own main memory, resources are independent and can be directly aggregated. However, buses might be shared by multiple devices during node data exchanges or accessed differently. Moreover, the parameter $\chi_u$ may vary between computing units. It is noted that in most cases,

all computing units will feature similar $\beta_{u,h}$ and $\chi_u$ (*e.g.*, this occurs in fat nodes with multiple equal devices; moreover, in CPU-based clusters, $\beta_{u,h} = \infty$ and $\chi_u = 1$). Indeed, only in hybrid systems will these parameters vary significantly. In that case, the workload assigned to accelerators is larger than that of the host (due to workload balancing). Therefore, let us define an effective aggregated device-to-host link bandwidth, $\beta_l = \sum_{u \in U_n \setminus h} \beta_{u,h} X_n / X_u$ (note that its value results slightly higher than the sum of device-to-host links if there are data exchanges in the host side), and an average ratio of inter-node exchanges, $\bar{\chi} = \sum_{u \in U_n} \chi_u X_u / X_n$. Now, the overhead introduced by data exchanges in hybrid nodes reads

$$t_x = X_n \left( \frac{2}{\beta_l} + \frac{3 + 3\bar{\chi}_u}{\beta_h} + \frac{\bar{\chi}_u}{\beta_x} \right). \tag{3.9}$$

Finally, we generalize Equation 3.7 to obtain the theoretical maximum ratio of computing to communicating work per node:

$$\frac{t_k}{t_x} = \frac{Q_k}{X_n} \cdot \frac{\dfrac{1}{\beta_n}}{\dfrac{2}{\beta_l} + \dfrac{3 + 3\bar{\chi}_u}{\beta_h} + \dfrac{\bar{\chi}}{\beta_x}} \geq 1, \tag{3.10}$$

The conclusions of this section allow estimating the theoretical parallel behavior of a hybrid node during the execution of a specific application given the set of $X_u^i$ and $X_u^e$ per computing unit, and the memory traffic requirements, $Q_k$.

## 3.3 Multilevel workload distribution

To execute a numerical simulation on a hybrid HPC system as in Figure 3.1, the workload distribution is addressed by a multilevel approach, a distribution technique very suitable for hierarchical parallel implementations [4, 5]. Essentially, it aims at decomposing the computational load in two levels at least. First, to assign the load per computing node; second, the load per computing unit. Further levels are required to target, for instance, processors with complex NUMA configurations [6] ensuring thread affinity and local memory access.

In mesh methods, the computational domain arises from the spatial discretization

of a physical domain. Given whatever space of interest, $\Omega$, we can equip it with a partition of unity, namely a mesh $M$, by bounding the set of cells, $C$, with faces, $F$; those with the set of edges, $E$, and finally those with the set of vertices, $V$. Such a mesh can be represented with graphs, which are discrete mathematical structures representing pairwise relations between objects. In turn, graphs can be represented with adjacency and incidence matrices.

The HPC$^2$ library deals with vectors and matrices representing discrete variables and operators, respectively. In this context, the workload distribution consists in distributing vector elements and matrix rows into subvectors and submatrices assigned to different computing units. Besides, it is not restricted to mesh methods but graphs in general. Indeed, any matrix is a graph. Therefore, let us make it simpler and consider whatever pair of sets, U and V. The adjacency matrices, $A_U$ and $A_V$, are square matrices representing the connectivity between adjacent elements of the same set. Whenever there is a non-zero entry in $(A_U)_{ij}$, then $i$th and $j$th elements of U are adjacent. The incidence matrices, $E_{U\to V}$ and $E_{V\to U}$, represent the relationship between elements of different sets. Similarly, a non-zero entry in $(E_{U\to V})_{ij}$ represents an incidence relation between the $i$th element of V and the $j$th element of U.

Figure 3.3 illustrates two different multilevel workload distributions of a mesh (left), and the resulting cell-adjacency matrices (right) are also decomposed. The upper partitioning minimizes the number of edge cuts (*i.e.*, the couplings between elements located in different subdomains), while the lower one isolates some partitions from inter-node data exchanges.

The first-level decomposition is represented in Figure 3.3 by a gray division. The initial workload is distributed among computing nodes (*i.e.*, labeled in Figure 3.3 from 0 to 22 and from 23 to 45) using a parallel graph partitioning tool, such as the ParMETIS [7] library that fulfills the requested load balancing and minimizes the number of couplings between partitions.

The second-level partitioning is represented in Figure 3.3 by colored divisions and shadings. The local workload of each hybrid node is distributed among the available computing units by either using a graph partitioning tool again or switching to a custom tool that isolates accelerators from inter-node exchanges. The latter approach is particularly useful in heterogeneous computing, because accelerators introduce a larger overhead at inter-node parallelization (recall Equation 3.6). It is noted that the second-level partitioning must conform to the actual performance of the available

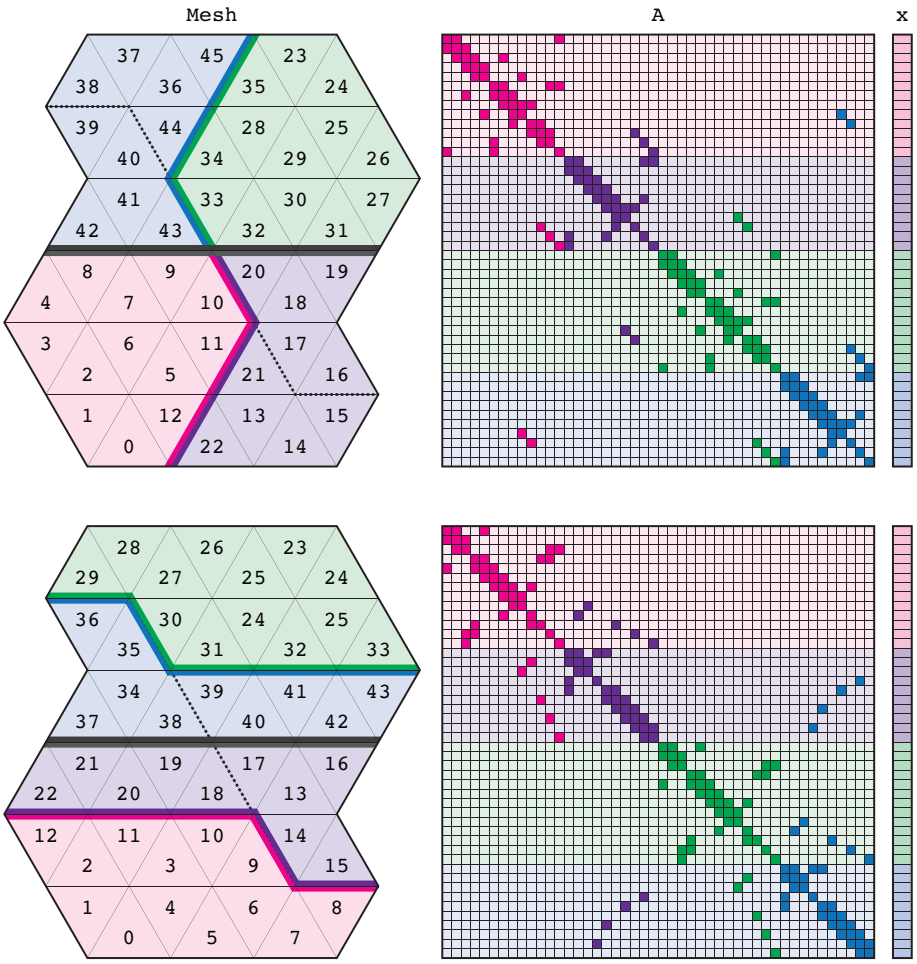


**Figure 3.3:** Representation of a row-major, multilevel workload distribution. Vector elements and matrix rows are divided into two first-level chunks for two compute nodes, represented by gray divisions, which are further divided into second-level chunks, represented by colored divisions, for two computing units. Some of the second-level chunks are implicitly organized into third-level intervals represented by a dashed division.

computing units for the sake of load balancing.

The third-level partitioning is represented in Figure 3.3 by dashed contours. This third-level applies to units with NUMA memory configurations where the local memory

access rate is faster than remote. In this case, a rather simple arrangement of the second-level partition among computing threads is made such that a pair of integers, *offset* and *size*, define the data intervals for each thread. In contrast with the first- and second-level decomposition in which data is explicitly distributed, this implicit third-level arrangement exploits NUMA configurations at its best. Threads are bound to NUMA groups using thread affinity. The data is initialized through a predictive first-touch policy so that each third-level interval resides in the thread's local memory bank. A similar approach is found, for instance, in [8]. Thus, this approach still benefits from SM parallelism.

Note that the HPC$^2$ is designed to work with any number of sets and perform coherent multilevel decompositions of each set given the corresponding adjacency or incidence matrices. Roughly, an initial set is decomposed using either its adjacency matrix or any incidence. Then, given any incidence matrix between an already partitioned set and another blank, the blank set is coherently decomposed following a rather simple heuristic: an element from the blank set will belong to the same partition as its incident element with the smallest index. In other words, if $u_n$ is incident to $v_i$, $v_j$ and $v_k$, and $i < j < k$, then $u_n$ will be assigned to $v_i$'s partition.

Compared to other implementations, which assign an MPI rank to each computing unit [9–12], this approach minimizes the number of processes participating in MPI exchanges, as well as the global size of the messages. It takes full advantage of the intra-node topology and the shared-memory parallel processors and minimizes inter-node communications. Moreover, this advantage will only strengthen as the memory hierarchies of modern supercomputers become more complex, increasing the number of accelerators and NUMA groups per node.

## 3.4 Communication hiding strategies

In the multilevel decomposition approach, partitions may be identified by a pair of integers $(p, q)$, that is, their first- and second-level partition IDs. The elements in $(p, q)$ partition become a set of contiguous indices, the *own* set of $(p, q)$. By the same token, non-own elements become the *outer* set of $(p, q)$.

Any non-zero entry, $[\mathsf{A}]_{ij}$, represents a coupling between the $i$th element in $V$ and the $j$th element in $U$. There is no guarantee that the $i$th and $j$th elements of vector spaces $V$ and $U$ are located in the same memory space in distributed parallel

processing. Hence, parallel routines such as SpMV induce data exchanges between processes. For instance, the fifth row in Figure 3.3, located in $(p_0, q_0)$, is coupled with an element of $(p_0, q_1)$ and an element of $(p_1, q_1)$. The communication stage affects the performance and limits the scalability of the operation, so necessary in large-scale applications. Therefore, the DM MIMD parallelisation of the SpMV must minimize the communication overhead.

It is noted that some elements in the own set have no couplings with any outer element. Therefore, the own set is further organized into *inner* and *interface* categories. The inner set consists of those elements of the own set, which are coupled with own elements only. Conversely, interface set consists of those elements coupled with an element of the outer set. The outer elements of $U$ required by the interface couplings are denoted as halo.

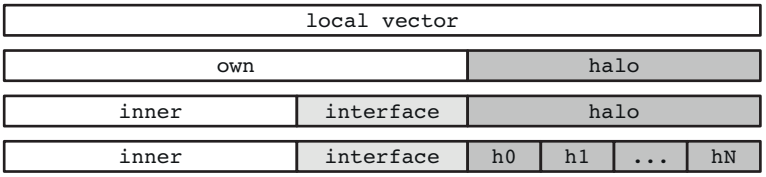


**Figure 3.4:** Ordering of vector elements according to their role in the parallel execution.

Given that the inner calculations are independent of the halo, this allows computing the calculations of inner elements simultaneously with data exchanges, required by the interface elements only. In this regard, elements are reordered locally as follows: first the inner set, then the interface set, and finally the halo (*cf.* Figure 3.4). The halo set is further reordered in ascending order of the owning subdomain numbers (computing units) to simplify the processing of communications.

Recalling the performance estimation developments in Section 3.2, the execution time in a synchronous implementation is estimated as follows:

$$t_{syn} = t_k + t_x, \tag{3.11}$$

where $t_k$ and $t_x$ are the kernel and data exchange times (Equations 3.8 and 3.9). In contrast, the time in an overlapping implementation reads

$$t_{ovl} = max(t_k^{inn}, t_x) + t_k^{ifc}. \tag{3.12}$$

Also, considering the partitioning described in Figure 3.3's bottom, which isolates the accelerators from inter-node exchanges, a double overlap is possible by exchanging the intra- and inter-node halo simultaneously. This double overlapping is estimated as follows:

$$t_{dov} = max(t_k^{inn}, max(t_x^i, t_x^e)) + t_k^{ifc}, \tag{3.13}$$

where $t_x^i$ and $t_x^e$ correspond to the times of intra- and inter-node exchanges, respectively. More precisely, in reference to Equation 3.5:

$$t_x^e = X_h^e \left( \frac{6}{\beta_h} + \frac{1}{\beta_x} \right), \tag{3.14}$$

$$t_x^i = t_x - t_x^e. \tag{3.15}$$

But a word of caution: this type of partition enlarges the size of the internal halo, $X^i$; therefore, the implementation must ensure that $t_x^i$ is kept smaller than the original $t_x$ of the optimal decomposition minimizing the number of couplings.

Figure 3.5 illustrates the NUMA-aware execution diagram enabling simple and double overlapping. The diagram boxes are described in Table 3.1. A flat, fixed-size OpenMP region is initialized before the kernel in the diagram. OpenMP threads are assigned different roles, either computing threads or management threads. 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. The workload is coherently distributed among CPU cores using third-level mesh partitioning (see Section 3.3) instead of loop parallelism. The management subgroup, which can be implemented via OpenMP tasking or nested multithreading, handles data exchanges and accelerator queues in a hybrid node. This NUMA-aware parallelization has significantly improved performance on multiprocessor nodes.

Namely, in our previous implementation [13], 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.

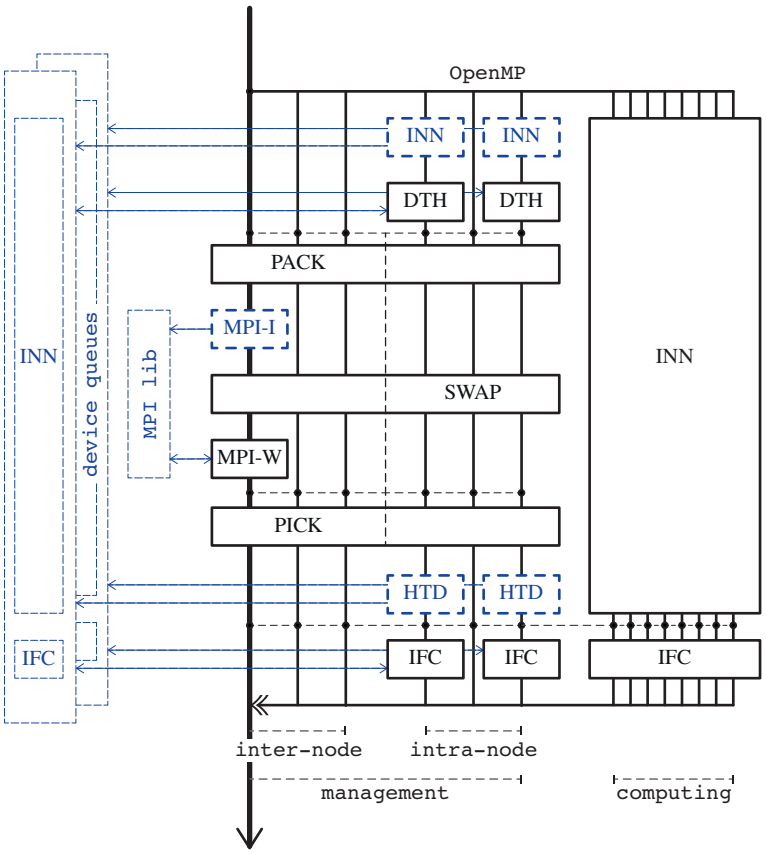


**Figure 3.5:** Execution diagram for the overlap mode using a flat OpenMP region (dashed boxes denote non-blocking calls).

Considering the workload distribution in Figure 3.3's top, the single overlap is fulfilled straightforward. Firstly, management threads wait at a synchronization barrier while one thread per accelerator enqueues the inner computations and device-to-host data copies. Then, management threads pack the MPI send buffer and perform the intra-node data exchanges (SWAP). Meanwhile, master thread enqueues the MPI non-blocking send and receive messages and joins the others in swapping. After swapping, management threads pause at a synchronization barrier while the master thread waits MPI messages to finish, and then all together unpack the receive buffers. Finally,

**Table 3.1:** Stages of the distributed memory, parallel SpMV execution (per computing unit).

| Stage | Description | Time |
|---|---|---|
| INN | Kernel execution for the inner subset | $Q_k^{inn}/\beta_u$ |
| DTH | Transfer the interface subset data from devices to host | $X_u/\beta_{u,h}$ |
| PACK | Put the external interface elements of devices into MPI send buffers | $3X_u^e/\beta_h$ |
| MPI-I | Initialize the MPI communications with non-blocking calls | – |
| SWAP | Copy the internal interface elements into halo buffers of devices | $3X_u^i/\beta_h$ |
| MPI-W | Wait for MPI communications to finish | $X_u^e/\beta_x$ |
| PICK | Get the external halo elements of devices from MPI receive buffers | $3X_u^e/\beta_h$ |
| HTD | Transfer the halo subset from host to devices | $X_u/\beta_{u,h}$ |
| IFC | Kernel execution for the interface subset | $Q_k^{ifc}/\beta_u$ |

management threads assigned to accelerators enqueue host-to-device copies and, after a global OpenMP barrier, the interface computations are processed.

The double overlap concerning Figure 3.3's bottom partition is achieved by splitting the PACK, SWAP, and PICK boxes, and the synchronization points, by the vertical dashed line as shown in Figure 3.5. More precisely, the group of inter-node management threads will be in charge of packing, sending, and unpacking MPI messages while intra-node management threads enqueue kernels, perform device-to-host copies, and swap intra-node data.

## 3.5 Portable implementation model

In a nutshell, our portable implementation model introduces three types of objects: *actuator*, *container* and *shaper*. A shaper is some sort of instructions manual that allows transforming any initial sequential input into a set of hierarchical partitions enabled for parallel processing. Containers are the data storage objects that stock such hierarchical partitions. Finally, the actuator is the object that provides with methods and functions to firstly create shapes and then manipulate and operate containers. This portable implementation model will be described throughout the section.

The solution we propose is depicted in Figure 3.6. Namely, there are four objects conceived for developers and four for users. The set of objects intended for users are nothing but high-level handlers or *wrappers* that contain lists of their low-level counterparts. Recall that the hierarchical parallel implementation of HPC$^2$ is designed to work with multilevel partitions as described in Section 3.3. Therefore, each high-level object operates with the required number of $n$th-level partitions on the backs of the

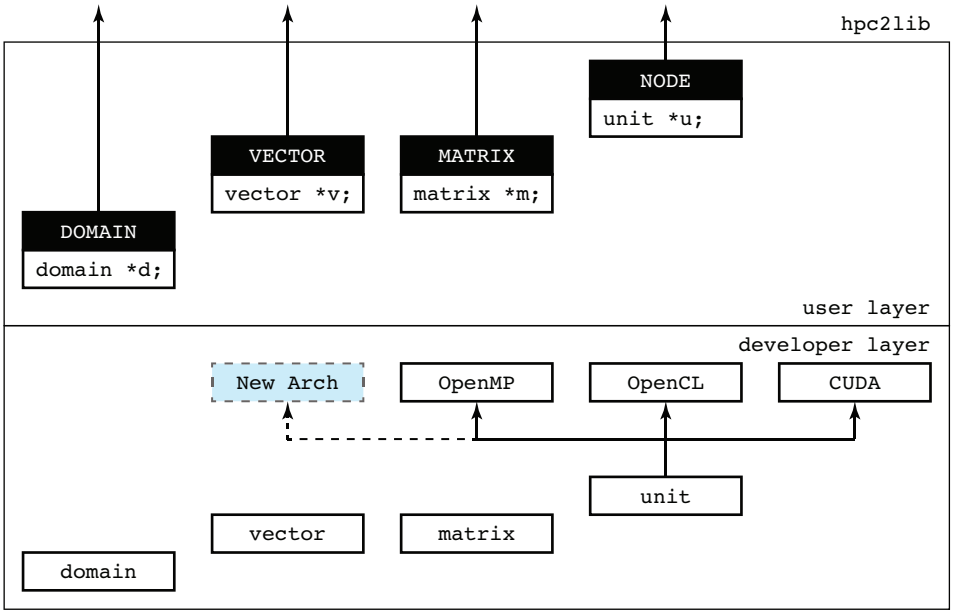


**Figure 3.6:** Diagram of classes in our pure virtual approach for managing platform portability.

users. In other words, no specification of the parallel environment or configuration is required in deploying an algorithm using HPC$^2$ library (the user can use HPC$^2$ objects in a way as simple as `std::vector`). This is illustrated in Listing 3.1 following the example in Equation 3.1.

**Listing 3.1:** Heat flux computation using hpc2lib.

```
#include "hpc2lib.h"

/* NODE object is a singleton and is accessed through a pointer */
NODE *Node;

int main(int argc, char** argv){
    /* initialize MPI stuff */
    int prov, req = MPI_THREAD_MULTIPLE;
    MPI_Init_thread(&argc, &argv, req, &prov);

    /* initialize NODE with command line arguments */
```

```
12      Node->Init(&argc, &argv);
13
14      /* initialize HPC2 objects using plain sequential data in binary files */
15      DOMAIN Cells, Faces;
16      Node->CreateDomain(Cells, Faces, "input_G.bin");
17
18      VECTOR q = Node->BuildVector(Faces, 0.0);
19      VECTOR T = Node->BuildVector(Cells, "input_T.bin");
20      MATRIX G = Node->BuildMatrix(Cells, Faces, "input_G.bin");
21
22      /* compute the gradient */
23      double k = 0.598;
24      Node->SpMV(G, T, q, -k);
25
26      MPI_Finalize();
27  }
```

The code in the example above (Listing 3.1) can be compiled and executed on virtually any modern computing environment. By way of example, Listing 3.2 shows three command-line configurations to execute the simulation on: 1) 200 nodes with 48 cores each, 2) a fat node with 4 NVIDIA GPUs, 3) two hybrid nodes with one 14-core CPU and one AMD GPU each.

**Listing 3.2:** Execution of .

```
1  mpirun -np 200 ./heat -devices=1 -imp=openmp -thr=48
2  mpirun -np 1 ./heat -devices=4 -imp=cuda,cuda,cuda,cuda
3  mpirun -np 2 ./heat -devices=2 -imp=openmp,opencl -thr=13,1 -wgt=68,288
```

The parameter `devices` determines the number of *virtual* devices to be used, that is, the number of second-level partitions. We specify virtual devices to highlight that a physical computing unit or device can be assigned to multiple virtual devices and, therefore, to multiple second-level partitions. This is particularly interesting in units with device queues such as GPUs. Then, at least one processor thread is assigned to each virtual device. It is possible to assign multiple threads to a virtual device using the parameter `thr`, which is particularly necessary in multi-core units. To deal with load balancing, the parameter `wgt` determines the relative workload for each virtual device. Last, but not least, the parameter `imp` determines the implementation, or backend, assigned to a virtual device. In this portable implementation, all architecture-specific implementations is encapsulated in a single, pure virtual class (the actuator).

### Unit and node

The node is the actuator provided to the user. As a singleton class, only one instance exists during the execution; and it always exist [14]. Node must be initialized at the beginning of the program using command-line parameters (line 3 of Listing 3.1) to determine the number of virtual devices, their implementation and relative weight, or the number of threads that operate. Then, if the user wants to create a domain, he uses the node (line 16 of Listing 3.1); if he wants to build vectors or matrices using plain binary files, he uses the node (lines 18–20 of Listing 3.1); if he wants to execute kernel, he uses the node (line 24 of Listing 3.1). At a glance it may seem a sequential application, but a powerful hierarchical parallel implementation is managed in the background.

In this approach, all the architecture-specific implementation is encapsulated in a single low-level, pure virtual class, the virtual unit, which can be specialized for different architectures. Currently, there are three different implementations of virtual unit: OpenMP, OpenCL, and CUDA. Each implementation is designed to allocate and operate second-level partitions of vectors and matrices. In each execution, the node will generate as many derived instances of virtual units as requested by the parameters, one per second-level partition. If a new architecture or parallel paradigm comes into play, the implementation of a new virtual unit is sufficient to port the entire numerical simulation framework.

### Vector and matrix

This pair of objects are containers used to store discrete mesh functions and operators. Vectors must be bound to a domain so that the user-given plain data, the input binary file, is transformed and distributed accordingly among the required compute nodes and units. Moreover, matrices are bound to two domains representing the input and output vector spaces. The former is used to renumber column indices, while the latter is used to transform and distribute matrix rows among the hardware.

There are multiple sparse matrix storage formats deployed (*e.g.*, the diagonal format, the standard CSR, or the ELLPACK [15] and its variants [16]), and more can be added easily.

**Domain**

This object describes a (computational) vector space, and contains the information required to transform such space from the user-given plain or sequential form, the input binary file, into the HPC$^2$-based multilevel form. It is required for allocating and operating containers, which are vector and matrix. Therefore, the domain is the shaper that describes the size of second- and third- level partitions of the vector space and the size and offset of its subsets. Besides, it contains maps to reorder the data back and forth, and to perform the required data exchanges in DM parallelization.

## 3.6 Challenges and opportunities

The fully-portable, algebra-based implementation model proposed in this chapter is promising, but also challenging. In this section, we outline some of the challenges we face and propose valid solutions and opportunities.

### 3.6.1 Exploiting the Kronecker product

SpMV is the most computationally expensive routine in many large-scale simulations relying on iterative methods. Namely, it is a strongly memory-bound kernel with a very low AI, which is the ratio of computing work in flop to memory traffic in bytes (its value is around 1:8 flop per byte), and requires irregular memory accessing to the input vector harming the memory access efficiency. To top it off, in distributed-memory parallel processing, vector elements and matrix rows are distributed among a group of processes inducing data exchanges between them. Therefore, the efficient execution of SpMV requires a fine-tuning process (*e.g.*, right choice of the sparse matrix storage format, proper workload balancing, reordering of unknowns to reduce matrix bandwidth, optimizing memory access to minimize cache misses).

Significant effort is devoted to studying and optimizing SpMV for different applications and state-of-the-art computing environments. The introduction of the GPUs into HPC systems motivated the research of new sparse matrix storage formats and SpMV implementations, as reviewed by Filippone et al. in [16]. The continuous evolution of CPUs also motivates the research for efficient SpMV kernels on such architectures [17, 18]. However, the AI still limits all these efforts as discussed in Section 3.2.

In some cases, a sparse matrix is to be multiplied by a set of vectors. For instance, in the evaluation of the diffusive term in collocated formulation as in Section 2.1.1:

$$\mathsf{D}_c^{3d}\boldsymbol{u}_c = -\left(\nu \mathsf{I}_3 \otimes \mathsf{L}\right)\boldsymbol{u}_c = -\left(\begin{pmatrix}\nu & 0 & 0\\ 0 & \nu & 0\\ 0 & 0 & \nu\end{pmatrix}\otimes \mathsf{L}\right)\boldsymbol{u}_c = -\begin{pmatrix}\nu\mathsf{L} & 0 & 0\\ 0 & \nu\mathsf{L} & 0\\ 0 & 0 & \nu\mathsf{L}\end{pmatrix}\boldsymbol{u}_c. \quad (3.16)$$

where, recall, $\boldsymbol{u}_c \in \mathbb{R}^{3n}$ is the cell-centered velocity field defined as a column vector and arranged as $\boldsymbol{u}_c = (\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3)^T$, $\mathsf{D}_c^{3d} \in \mathbb{R}^{3n\times 3n}$ is the block diagonal diffusive matrix and $\mathsf{L} \in \mathbb{R}^{n\times n}$ the discrete Laplacian operator. Moreover, in some cases it could be interesting to execute two simultaneous simulations with different parameters [19], $\nu_1$ and $\nu_2$. Then:

$$\begin{pmatrix}(\mathsf{D}_c^{3d})_1 & 0\\ 0 & (\mathsf{D}_c^{3d})_2\end{pmatrix}\begin{pmatrix}(\boldsymbol{u}_c)_1\\ (\boldsymbol{u}_c)_2\end{pmatrix} = -\begin{pmatrix}\nu_1\mathsf{L} & 0 & 0 & 0 & 0 & 0\\ 0 & \nu_1\mathsf{L} & & 0 & 0 & 0\\ 0 & 0 & \nu_1\mathsf{L} & 0 & 0 & 0\\ 0 & 0 & 0 & \nu_2\mathsf{L} & 0 & 0\\ 0 & 0 & 0 & 0 & \nu_2\mathsf{L} & 0\\ 0 & 0 & 0 & 0 & 0 & \nu_2\mathsf{L}\end{pmatrix}\begin{pmatrix}(\boldsymbol{u}_c)_1\\ (\boldsymbol{u}_c)_2\end{pmatrix}. \quad (3.17)$$

Such formulation applies to several scenarios in numerical algorithm implementations that are increasingly common. Examples are symmetric grids [20], parallel in time methods [21], multiple transport equations or multiple parameter simulations [19], among others. The nested combination of this approaches in a single framework can be expressed as follows:

$$\boldsymbol{y} = (diag(\boldsymbol{c}_1) \otimes ... \otimes diag(\boldsymbol{c}_n) \otimes \mathsf{A})\boldsymbol{x}, \quad (3.18)$$

where $\mathsf{A} \in \mathbb{R}^{m\times n}$ is a sparse matrix, $diag(\boldsymbol{c_i}) \in \mathbb{R}^{d_i\times d_i}$ is a diagonal matrix containing the entries in $\boldsymbol{c_i} \in \mathbb{R}^{d_i}$, and $\boldsymbol{y} \in \mathbb{R}^{Dm}$ and $\boldsymbol{x} \in \mathbb{R}^{Dn}$ are sets of vectors with $D = \prod_{i=1}^{n} d_i$ elements each.

In this context, the sparse matrix-matrix product (SpMM) kernel is described in Algorithm 2. It represents the product of a sparse matrix by a set of dense vectors. It results in significantly greater data reuse: in line 5, the coefficients of the sparse matrix are reused as many times as blocks are in the input vector. Needless to say, SpMM is applicable to matrices factored as the Kronecker product of a diagonal matrix times another as in Equation 3.18.

**Algorithm 2** SpMM implementation using the standard CSR matrix format.

**Require:** $\mathsf{A}$, $\boldsymbol{x}$, $\boldsymbol{c}$
**Ensure:** $\boldsymbol{y}$
1: **for** $i \leftarrow 1$ to $m$ **do**
2: $\quad \boldsymbol{s} \leftarrow \text{zeros}(d)$
3: $\quad$ **for** $j \leftarrow A.ptr[i]$ to $A.ptr[i+1]$ **do**
4: $\quad\quad$ **for** $k \leftarrow 1$ to $d$ **do**
5: $\quad\quad\quad \boldsymbol{s}[k] \leftarrow \boldsymbol{s}[k] + A.val[j] \cdot \boldsymbol{x}[A.idx[j]][k]$
6: $\quad$ **for** $k \leftarrow 1$ to $d$ **do**
7: $\quad\quad \boldsymbol{y}[i][k] \leftarrow \boldsymbol{c}[k] \cdot \boldsymbol{s}[k]$

According to Algorithm 2, and considering double-precision, standard CSR sparse matrix format, and ideal temporal locality, the AI of the SpMM reads:

$$\text{AI}_{\text{SpMM}}(d) = \frac{(2nnz(\mathsf{A}) + 1) \cdot d}{8nnz(\mathsf{A}) + 4nnz(\mathsf{A}) + 4(m+1) + (8m + 8n + 8) \cdot d}. \tag{3.19}$$

where $nnz(\mathsf{A})$, $m$ and $n$ are the number of non-zero elements, rows and columns in the matrix, respectively, and $d$ is the number of vectors. Consequently, the maximum speed-up achievable by replacing $d$ recursive SpMV calls with a single SpMM equals $\text{AI}_{\text{SpMV}}(d)/\text{AI}_{\text{SpMM}}(1)$. This upper-bound is plotted in Figure 3.7. The lower-bound is also given considering zero temporal locality (*i.e.*, accounting for the total number of memory accesses to the input vector, $8nnz(A) \cdot d$, instead of $8n \cdot d$). It is noteworthy that, being the upper-bound proportional to the average number of non-zeros per row, $nnz(A)/m$, the use of high order schemes may strengthen the benefits of the SpMM.

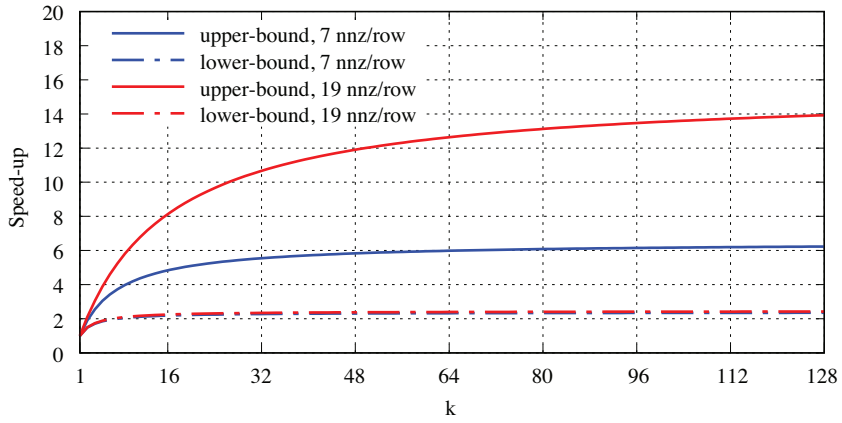


**Figure 3.7:** Different data layouts for sets of vectors.

### 3.6.2 Implementation of non-linear terms

From the formulation and algorithm implementation in Section 2.1.3, it is easily observed that non-linear terms are not efficiently introduced. Moreover, it may seem that it is not possible to introduce more complex non-linear terms such as high-resolution schemes or flux limiters.

Nevertheless, instead of being an inconvenience, this encouraged us to demonstrate the high potential of our algebraic strategy again. In our previous work [22], we proposed the generalization of the linear combination of vectors via the introduction of a generalized binary operator (kbin) that performs any given pointwise arithmetic calculation such that:

$$y_i \leftarrow y_i \circ f(x_i). \tag{3.20}$$

This binary operator can easily map to the required non-linear kernels by defining $\circ$ and $f(x_i)$ as outlined in Table 3.2. Similarly, the dot product kernel can be turned into a generalized reduction operator (kred),

$$r \leftarrow r \circ f(x_i), \tag{3.21}$$

which can easily represent any required reduction operation such as the calculation of the norm of a vector or the Courant–Friedrichs–Lewy (CFL) condition.

**Table 3.2:** Particularizations of the operator $\circ$ and pointwise function $f(x_i)$ to represent various kernels using the generalized binary operator described in Equation 3.20. The *superbeexty* corresponds to the SUPERBEE flux limiter [23].

| $\circ$ | $f(x_i)$ | AI | Resulting Kernel |
|---|---|---|---|
| $+$ | $ax_i$ | 1/12 | axpy |
| $\times$ | $ax_i$ | 1/12 | axty |
| $/$ | $ax_i$ | 1/12 | axdy |
| $\times$ | $x_i > 0? + 1 : -1$ | 1/12 | signxty |
| $\times$ | $0.5(max(0, max(min(1, 2x_i), min(x_i, 2))) - 1)$ | 7/24 | superbeexty |

From a computational point of view, this kernel generalization does not alter the implementation of the original axpy: it still performs simple, pointwise arithmetic operations over the vector elements and provides uniform, aligned and coalescing memory accesses which suits the SIMD and SP paradigms perfectly. Therefore, having already efficient implementations of axpy for different architectures, the implementation of kbin is straightforward (*e.g.*, consider the use of function pointers, templates, macros, among others). Indeed, the calculation of the convective term in Equation 5.5 can be computed using the *axty* form of kbin, avoiding to waste time on building the U matrix at each time-step.

On the other hand, the arithmetic intensity of this new kernel is not a fixed value anymore, as shown in Table 3.2. While the AI of the axpy is 1/12 flop per byte (one product and one addition per three double-precision values), that of the kbin depends on the specific arithmetic calculations involved in the function $f(x_i)$. This allows us to significantly increase the AI in our calls by means of kernel fusion, reduce the number of intermediate results, and thus reduce the time-to-solution.

The application of this approach to a canonical case, a three-dimensional deformation problem, has been validated and its performance studied in detail in Appendix A.

## References

[1] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "Top500 list – November 2021." `https://www.top500.org/lists/top500/list/2021/11/`, November 2021.

[2] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.

[3] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[4] A. Gorobets, S. A. Soukov, and P. B. Bogdanov, "Multilevel parallelization for simulating compressible turbulent flows on most kinds of hybrid supercomputers," *Computers & Fluids*, vol. 173, pp. 171–177, 2018.

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

[6] X. Álvarez-Farré, A. Gorobets, and F. X. Trias, "A hierarchical parallel implementation for heterogeneous computing. Application to algebra-based CFD simulations on hybrid supercomputers," *Computers & Fluids*, vol. 214, p. 104768, 2021.

[7] G. Karypis and V. Kumar, "Parallel multilevel $k$-way partitioning scheme for irregular graphs," *SIAM Review*, vol. 41, no. 2, pp. 278–300, 1999.

[8] C. Alappat, J. Laukemann, T. Gruber, G. Hager, G. Wellein, N. Meyer, and T. Wettig, "Performance modeling of streaming kernels and sparse matrix-vector multiplication on a64fx," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 1–7, 2020.

[9] X. Guo, M. Lange, G. Gorman, L. Mitchell, and M. Weiland, "Developing a scalable hybrid MPI/OpenMP unstructured finite element model," *Computers & Fluids*, vol. 110, pp. 227–234, 2015.

[10] 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, 2015.

[11] B. A. Page and P. M. Kogge, "Scalability of hybrid sparse matrix dense vector (SpMV) multiplication," in *2018 International Conference on High Performance Computing Simulation (HPCS)*, pp. 406–414, 2018.

[12] G. Xiao, K. Li, Y. Chen, W. He, A. Y. Zomaya, and T. Li, "CASpMV: A customized and accelerative SpMV framework for the Sunway TaihuLight," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 131–146, 2021.

[13] X. Álvarez-Farré, A. Gorobets, and F. X. 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*, ECCOMAS 2018, (Glasgow, UK), pp. 2021–2031, International Centre for Numerical Methods in Engineering, CIMNE, June 2020.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2009.

[15] D. R. Kincaid, T. C. Oppe, and D. M. Young, "ITPACKV 2D user's guide," tech. rep., Center for Numerical Analysis, University of Texas, 1989.

[16] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, jan 2017.

[17] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), pp. 273–282, Association for Computing Machinery, June 2013.

[18] X. Álvarez-Farré, A. Gorobets, F. X. Trias, and A. Oliva, "NUMA-aware strategies for the heterogeneous execution of SpMV on modern supercomputers," in *World Congress in Computational Mechanics and ECCOMAS Congress*, January 2021.

[19] S. Imamura, K. Ono, and M. Yokokawa, "Iterative-method performance evaluation for multiple vectors associated with a large-scale sparse matrix," *International Journal of Computational Fluid Dynamics*, vol. 30, no. 6, pp. 395–401, 2016.

[20] X. Álvarez-Farré, À. Alsalti-Baldellou, A. Gorobets, A. Oliva, and F. X. Trias, "Enabling larger and faster simulations from mesh symmetries," in *2nd High-Fidelity Industrial LES/DNS Symposium*, (Toulouse), September 2021.

[21] B. I. Krasnopolsky, "An approach for accelerating incompressible turbulent flow simulations based on simultaneous modelling of multiple ensembles," *Computer Physics Communications*, vol. 229, pp. 8–19, 2018.

[22] N. Valle, X. Álvarez-Farré, A. Gorobets, J. Castro, A. Oliva, and F. X. Trias, "On the implementation of flux limiters in algebraic frameworks," *Computer Physics Communications*, vol. 271, p. 108230, 2022.

[23] P. K. Sweby, "High resolution schemes using flux limiters for hyperbolic conservation laws," *SIAM J. Numer. Anal.*, vol. 21, pp. 995–1011, oct 1984.

# 4

# Performance analysis

This chapter studies the performance of the three types of algebraic kernels implemented in HPC$^2$ (*cf.* Chapter 3), which are the basis of our DNS algorithms (*cf.* Chapter 2), in different computing environments. Specifically, the study covers the sparse matrix multiplication (SpMV), the pointwise binary operator (axpy), and the reduction operator (dot).

## 4.1 Introduction

Namely, several HPC clusters, which are listed below, have been used during the development of the HPC$^2$ framework:

- **JFF third-generation** cluster at Heat and Mass Transfer Technological Center. 40 compute nodes with two AMD Opteron 6272 (16 cores, 2.1 GHz, 4 DDR3-1600 memory channels, 51.2 GB/s memory bandwidth, 16 MB L3 cache), interconnected via QDR Infiniband (32 Gb/s).
- **JFF fourth-generation** cluster at Heat and Mass Transfer Technological Center. 21 compute nodes with two Intel Xeon 6230 (20 cores, 2.1 GHz, 6 DDR4-2933 memory channels, 140.8 GB/s memory bandwidth, 27.5 MB L3 cache), interconnected via FDR Infiniband (56 Gb/s).
- **MareNostrum 3** supercomputer at Barcelona Supercomputing Center. 3,056 compute nodes with two Intel Xeon E5-2670 (8 cores, 2.6 GHz, 4 DDR3-1600

memory channels, 51.2 GB/s memory bandwidth, 20 MB L3 cache), interconnected via FDR10 Infiniband (40 Gb/s).

- **MareNostrum 4** supercomputer at Barcelona Supercomputing Center. 3,456 compute nodes with two Intel Xeon 8160 (24 cores, 2.1 GHz, 6 DDR4-2666 memory channels, 128 GB/s memory bandwidth, 33 MB L3 cache), interconnected via Intel Omni-Path (100 Gb/s).

- **MinoTauro** cluster at Barcelona Supercomputing Center. 64 compute nodes with two Intel Xeon E5649 (6 cores, 2.53 GHz, 3 DDR3-1333 memory channels, 32 GB/s memory bandwidth), two NVIDIA M2090 (666.1 Gflop/s peak performance, 6 GB GDDR5 memory, 177.4 GB/s memory bandwidth, PCIe 3.0 x16 at 16 GB/s), interconnected with QDR Infiniband (40 Gb/s).

- **Lomonosov 2** supercomputer at Moscow State University. 1,696 compute nodes with one Intel Xeon E5-2697 v3 (14 cores, 2.6 GHz, 4 DDR4-2133 memory channels, 68 GB/s memory bandwidth, 35 MB L3 cache), one NVIDIA Tesla K40M GPU (1.43 Tflop/s peak performance, 12 GB of GDDR5 memory, 288 GB/s memory bandwidth, PCIe 3.0 x16 at 16 GB/s), interconnected via InfiniBand FDR network (56 Gb/s).

- **K60** cluster at Keldysh Institute of Applied Mathematics. 10 compute nodes with two Intel Xeon 6142 (16 cores, 2.6 GHz, 6 DDR4-2666 memory channels, 128 GB/s memory bandwidth, 22 MB L3 cache), eight NVIDIA V100 (7 Tflop/s peak performance, 32 GB HBM2 memory, PCIe 3.0 x16 at 16 GB/s), interconnected via two FDR Infiniband (56 Gb/s).

- **Titan** supercomputer at Oak Ridge National Laboratory. 18,688 compute nodes with one AMD Opteron 6274 (16 cores, 2.2 GHz, 4 DDR3-1600 memory channels, 51.2 GB/s memory bandwidth, 16 MB L3 cache), one NVIDIA K20X (1.31 Tflop/s peak performance, 6 GB DDR5 memory, 250 GB/s memory bandwidth, PCIe 3.0 x16 at 16 GB/s), interconnected via Cray Gemini 3D-Torus (51.2 Gb/s).

- **Mira** supercomputer at Argonne National Laboratory. 49,152 compute nodes with one IBM PowerPC A2 (16 cores, 1.6 GHz, 4 DDR3-1333 memory channels,

42.6 GB/s memory bandwidth, 32 MB L2 cache), interconnected via IBM 5D-Torus (320 Gb/s).

- **Cori** supercomputer at National Energy Research Scientific Computing Center. 9,688 compute nodes with one Intel Xeon Phi KNL 7250 (68 cores, 1.4 GHz, 6 DDR4-2400 memory channels, 115.2 GB/s memory bandwidth, 34 MB L2 cache), interconnected via Cray Aries (42 Gb/s).
- **TSUBAME3.0** supercomputer at Tokyo Institute of Technology. 540 compute nodes with two Intel Xeon E5-2680 v4 (14 cores, 2.4 GHz, 4 DDR4-2400 memory channels, 77 GB/s memory bandwidth, 35 MB L3 cache), four NVIDIA P100 (5.3 Tflop/s peak performance, 16 GB HBM2 memory, 732 GB/s memory bandwidth, PCIe 3.0 x16 at 16 GB/s), interconnected via four Intel Omni-Path (100 Gb/s).
- **Marconi100** supercomputer at Common Infrastructure for National Cohorts in Europe, Canada, and Africa. 980 compute nodes with two IBM Power9 AC922 (16 cores, 2.6 GHz, 8 DDR4-2666 memory channels, 170.6 GB/s memory bandwidth, 160 MB L3 cache), four NVIDIA V100 (7.8 Tflop/s peak performance, 16 GB HBM2 memory, 900 GB/s memory bandwidth, NVLINK 2.0 at 300 GB/s), interconnected via InfiniBand EDR (100 Gb/s).

The author thankfully acknowledge all the institutions for the computational resources and technical support given.

By way of introduction, Figure 4.1 shows the results of a preliminary performance study that was carried out prior to the large-scale DNS presented in Chapter 5. Briefly, the results evidence the performance-portability of the HPC$^2$ framework, although its most recent version has further improved. For instance, the former SpMV performed deficiently in both dual-socket nodes (first and second histograms) due to the NUMA factor not being taken into account, while the current NUMA-aware implementation does not suffer anymore in such configurations.

### Testing conditions

For the sake of clarity, the performance study shown in this chapter targets only three of the systems mentioned above. MareNostrum 4 and TSUBAME3.0 are chosen for

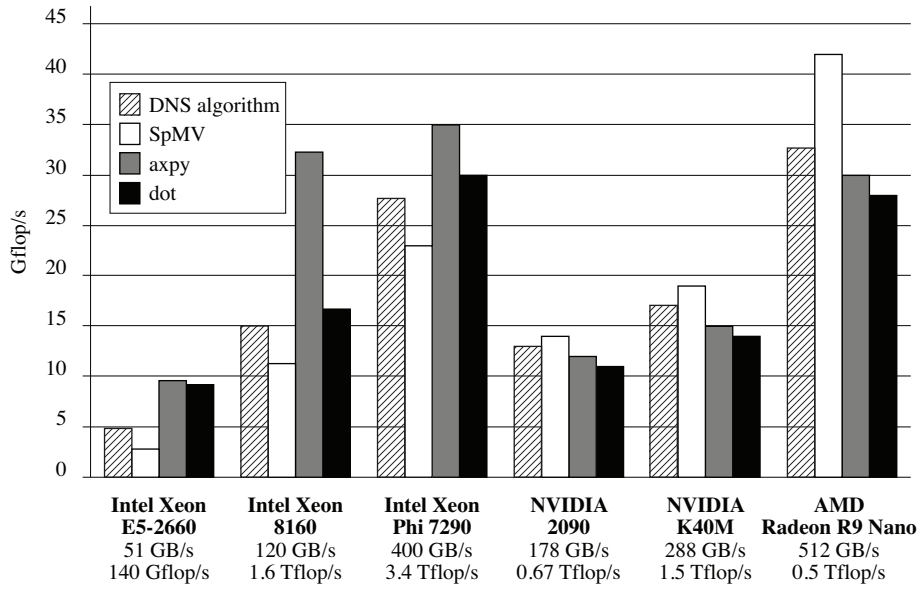


**Figure 4.1:** Preliminary performance study on multiple devices of different architectures. The discrete Laplacian operator resulting from the symmetry-preserving discretization [1] on an unstructured hex-dominant grid of 1M mesh cells is used. The ELLPACK sparse storage format and its block-transposed variant are used in CPU and GPU computations, respectively.

CPU- and GPU-only analysis, respectively, and Lomonosov 2 for the heterogeneous CPU + GPU performance study. The three systems are shown in Figure 4.2.

Time measurements are repeated in a loop one thousand times for each kernel to average the results. All tests are carried out using double-precision floating-point values. In single-node studies, the size of the problem is chosen large enough to avoid disturbances caused by temporal locality in CPU cache or low occupancy in GPUs (for a detailed performance study with smaller loads, the reader is referred to the author's master thesis [2]).

The sparse matrices arise from the symmetry-preserving discretization [1] 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 variant are used for CPU and GPU computations, respectively


**Figure 4.2:** Pictures of the three supercomputers used in this chapter's performance study. From left to right: MareNostrum 4, TSUBAME3.0, Lomonosov 2.

(see [3] for details). This formats provide uniform, unit-stride memory access with optimal coalescence of memory transactions.

As discussed in Chapter 3, the algebraic kernels are strongly memory-bound. In this regard, the theoretically achievable performance is estimated as $\pi_k = AI_k \times \beta_d$, where $AI_k$ and $\beta_d$ refer to the kernel's arithmetic intensity and device's memory bandwidth, respectively. The arithmetic intensities of SpMV, axpy, and dot, are 0.15, 0.125, and 0.125, respectively.

In the DM parallel execution of SpMV, two parallel modes are evaluated, which are overlapping and synchronous (*cf.* Chapter 3). Recall that the latter is to measure the time of communications and computations separately, while the overlapping measures both subroutines executed simultaneously.

At this point, it is worth specializing Equation 3.10 into an expression that estimates the theoretical maximum ratio of halo to inner mesh elements, $\rho = N^{hal}/N^{inn}$, that allow an adequate overlap in SpMV. For each mesh element, the exchanges of SpMV involve 8 bytes, while computations require accessing twelve bytes per non-zero element as well as reading both $x$ and $y$ and writing $y$, hence the memory traffic per element is $12nnz + 24$ bytes. Thus, considering seven non-zeros per row, we replace $Q_k = 108N^{inn}$ and $X_u = 8N_u^{hal}$:

$$\rho \leq \frac{108}{8} \frac{1}{\beta_n \left( \frac{2}{\beta_l} + \frac{3+\bar{\chi}}{\beta_h} + \frac{1}{\beta_x} \right)}. \tag{4.1}$$

This expression is used in this chapter's multi-node studies to discuss the results of strong and weak scaling.

## 4.2 MareNostrum 4

MareNostrum 4 is a CPU-based supercomputer at the Barcelona Supercomputing Center [4] consisting of 3,456 compute nodes and providing 10.30 Pflop/s of theoretical peak performance. Its dual-socket 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 via the Intel Omni-Path network (100 Gb/s). There are 216 nodes with 384 GB of memory (8 GB/core) and 3,240 with 96 GB (2 GB/core). The system architecture is illustrated in Figure 4.4.

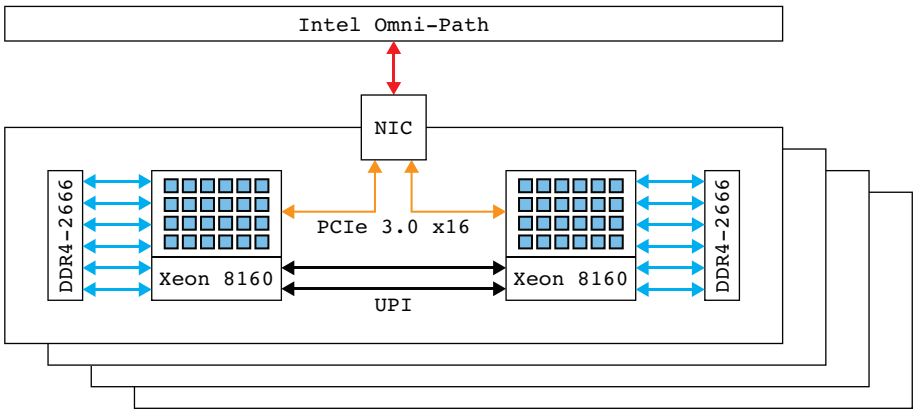


**Figure 4.4:** Topology and configuration of MareNostrum 4 supercomputer (3,456 compute nodes) at Barcelona Supercomputing Center (BSC).

Dual-socket configurations introduce an important NUMA factor. In this work, the performance on such configurations is enhanced through a NUMA-aware OpenMP multithreaded implementation that allows assigning a single MPI process per node while taking care of data locality according to the first-touch rule and thread affinity, preventing migration of threads between NUMA nodes, as detailed in Chapter 3.

### 4.2.1 Single-node study

The single-node performance of SpMV (left), axpy (middle), and dot (right) in different parallel execution modes on a mesh of 17 million cells are shown in Figure 4.5. Green horizontal lines represent the theoretically achievable performance. Also, the most relevant results are detailed in Table 4.1.

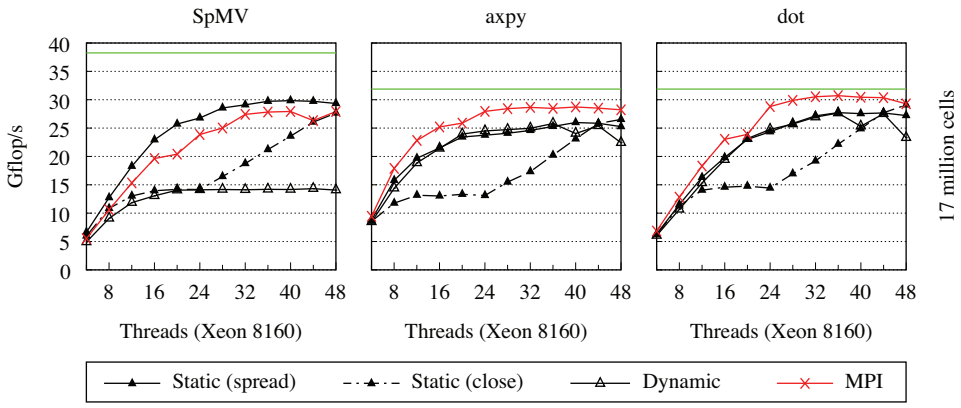


**Figure 4.5:** Single-node performance of HPC$^2$ kernels in different parallel execution modes on a mesh of 17 million cells, on MareNostrum 4. Both *Static* and *Dynamic* keywords refer to OpenMP multithreaded implementations.

In line with the preliminary results in Figure 4.1, the former implementation (labeled as *Dynamic* in Figure 4.5), lacking a NUMA-aware OpenMP implementation taking thread affinity and data locality into account, performs deficiently, especially with SpMV.

Both the MPI mode and the NUMA-aware OpenMP implementation with spread thread binding (*i.e.*, assigning threads to sockets or NUMA nodes equitably) perform similarly, rapidly achieving a high fraction of the theoretically achievable performance. The MPI mode, which leads to the most compact data placement, slightly outperforms OpenMP with axpy and dot but falls behind with SpMV. While axpy and dot are pure element-wise kernels that do not require heavy synchronization points on a single node, the SpMV introduces a noticeable overhead in DM parallel execution because processes must communicate and synchronize in addition to computing.

The NUMA-aware implementation without spread thread binding, *Static (close)*,

**Table 4.1:** Detailed single-node performance of HPC$^2$ kernels versus number of cores in different parallel execution modes on a mesh of 17 million cells, on MareNostrum 4.

| Kernel | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 | 44 | 48 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NUMA-aware OpenMP static (spread) | | | | | | | | | | | | |
| SpMV | 6.64 | 12.73 | 18.26 | 22.93 | 25.72 | 26.78 | 28.55 | 29.10 | 29.72 | **29.84** | 29.72 | 29.33 |
| axpy | 8.80 | 15.78 | 19.68 | 21.51 | 23.43 | 23.75 | 24.08 | 24.59 | 25.29 | **25.97** | 25.85 | 25.26 |
| dot | 6.55 | 11.52 | 16.31 | 19.81 | 23.00 | 24.35 | 25.82 | 27.15 | **27.73** | 27.55 | 27.67 | 27.21 |
| MPI | | | | | | | | | | | | |
| SpMV | 5.53 | 10.60 | 15.32 | 19.62 | 20.41 | 23.87 | 25.01 | 27.41 | 27.82 | **27.93** | 26.31 | 27.91 |
| axpy | 9.45 | 17.89 | 22.82 | 25.20 | 25.88 | 27.94 | 28.42 | 28.64 | 28.48 | **28.72** | 28.51 | 28.22 |
| dot | 6.84 | 12.82 | 18.30 | 23.01 | 23.90 | 28.80 | 29.88 | 30.53 | **30.71** | 30.44 | 30.36 | 29.28 |

reveals the importance of choosing a correct thread affinity policy. In this mode, the first 24 threads are placed into one socket, and the second 24 threads into the another. This leads to unbalanced use of memory resources and reduces the benefits of using the second socket into a linear expression, as shown in the plot.

### 4.2.2 Multi-node study

While the NUMA-aware implementation does not improve performance on a single node compared to the MPI-only (both implementations achieve a high fraction of the theoretically achievable performance), it positively impacts parallel efficiency and scalability. Here, we study the behavior of SpMV, axpy, and dot kernels in large DM parallel environments.

Firstly, Figure 4.6 shows the increase in halo to inner mesh elements ratio, $\rho$, as the number of compute nodes grows on initial meshes of 17 and 134 million cells. A blue line of value 0.5098 shows the theoretical maximum ratio estimated after Equation 3.10.

Strong scaling of SpMV (left), axpy (middle), and dot (right) from 48 to 9,600 cores (1 to 200 dual-socket nodes) on meshes of 17 and 134 million cells in different parallel execution modes are shown in Figure 4.7. The NUMA-aware, hybrid MPI + OpenMP implementation with spread thread binding clearly outperforms the MPI only. Indeed, the quasi ideal SpMV results on the mesh of 134 million cells demonstrate that the HPC$^2$'s SpMV scales on 200 nodes (9,600 cores) with a parallel efficiency of 96.5%, obtaining an effective speed up of $\times$193.

A superlinear scaling behavior is observed in all cases starting at around 100,000

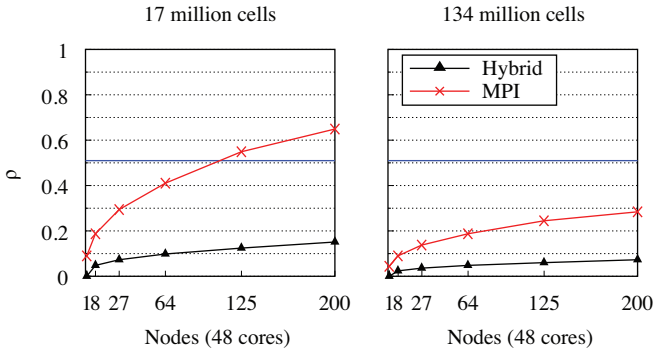


**Figure 4.6:** Halo to inner mesh elements ratio, $\rho$, versus number of compute nodes engaged in different parallel execution modes on MareNostrum 4.

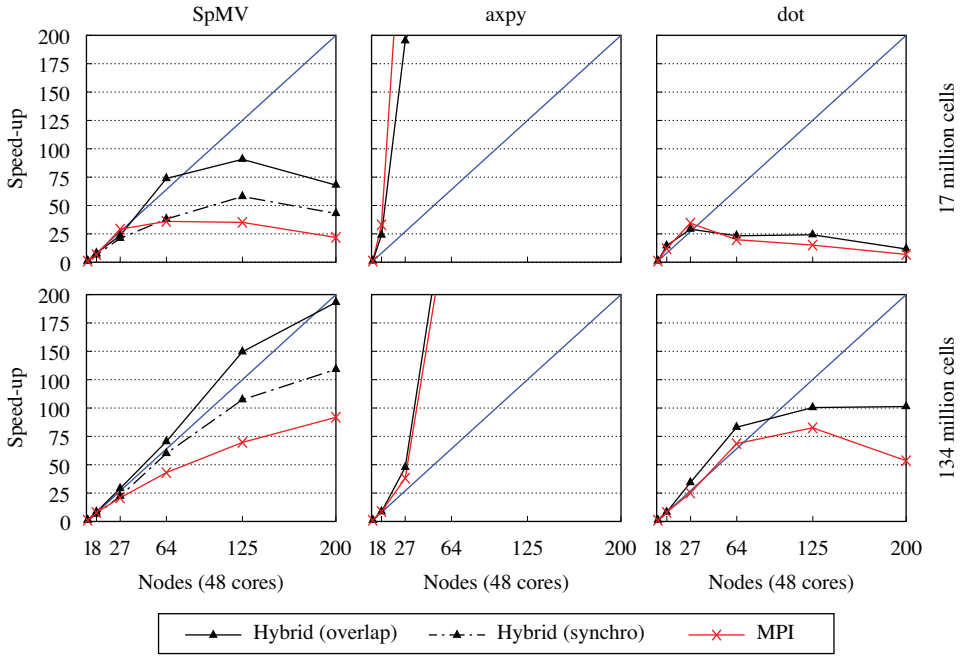


**Figure 4.7:** Strong scaling of HPC$^2$ kernels in different parallel execution modes on meshes of 17 and 134 million cells, on MareNostrum 4. The *Hybrid* keyword refers to the NUMA-aware OpenMP implementation with spread thread binding.

mesh cells per core due to the increase of temporal locality (cache reuse). Remark that axpy is a purely element-wise operation that does not require any involvement in parallel computing, and hence this superlinear scaling holds up indefinitely. However, this behavior degrades in SpMV and dot as the workload becomes smaller because the data exchange and synchronization overheads become unavoidable.

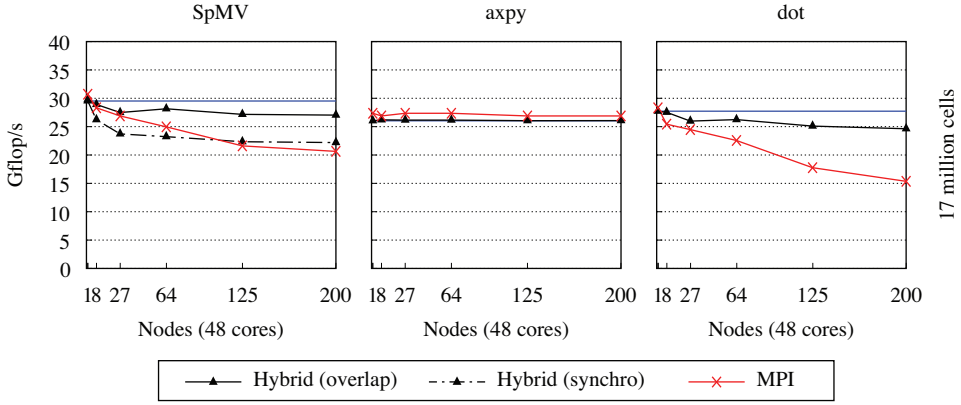


**Figure 4.8:** Weak scaling of HPC$^2$ kernels in different parallel execution modes on a mesh of 17 million cells, on MareNostrum 4. The *Hybrid* keyword refers to the NUMA-aware OpenMP implementation with spread thread binding.

Weak scaling of SpMV (left), axpy (middle), and dot (right) from 48 to 9,600 cores (1 to 200 dual-socket nodes) in different parallel execution modes are shown in Figure 4.8. A fixed workload per node of 17 million mesh cells is chosen as in single-node studies. Again, the NUMA-aware, hybrid MPI + OpenMP implementation outperforms the MPI-only with SpMV and dot because of the smaller number of MPI processes participating in reductions and data exchanges but performs similarly with axpy as expected.

The results indicate that the NUMA-aware implementation might be able to execute large-scale simulations on CPU-based supercomputers sustaining a parallel efficiency near 100%. Hybrid implementations are parallel-efficient due to the hierarchical parallelization, which allows exploiting both distributed- and shared-memory parallelism to minimize the number of processes participating in reductions or data exchanges and the number and size of the messages required for DM parallelization. It is noted that the NUMA-aware hybrid implementation allows engaging 9,600 cores

with 200 MPI processes, while the MPI-only approach requires 9,600 MPI processes to achieve the same level of parallelism.

## 4.3 Lomonosov 2

Lomonosov 2 is a hybrid supercomputer at Moscow State University [5] consisting of 1,696 compute nodes and providing 4.95 Pflop/s of theoretical peak performance. Its hybrid nodes are equipped with one Intel Xeon E5-2697 v3 (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 (1.43 Tflop/s peak performance, 12 GB of GDDR5 memory, 288 GB/s memory bandwidth, PCIe 3.0 x16 at 16 GB/s), and are interconnected via InfiniBand FDR network (56 Gb/s). The system architecture is illustrated in Figure 4.9.

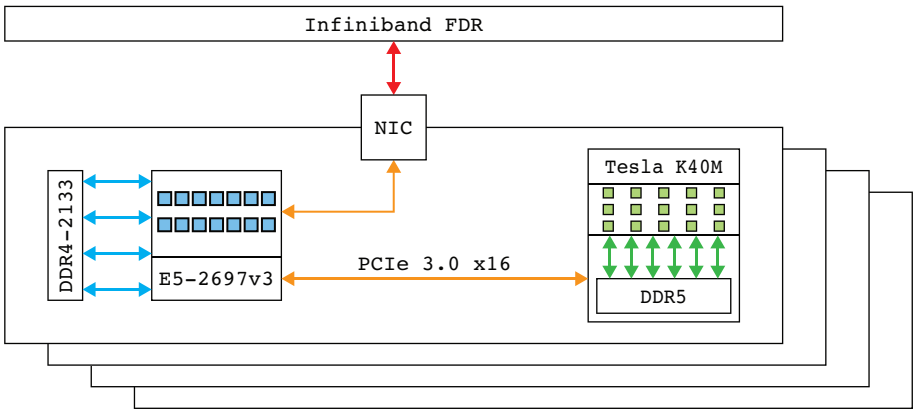


**Figure 4.9:** Topology and configuration of Lomonosov 2 supercomputer (1,696 compute nodes) at Moscow State University.

Hybrid nodes long for codes capable of heterogeneous computing that efficiently engage the variety of computing hardware. More precisely, a heterogeneous application targeting Lomonosov 2 must be able to combine CPU and GPU implementations simultaneously. The GPU to CPU memory bandwidth ratio in Lomonosov 2, which is 4.2, allows heterogeneous applications to benefit from a theoretical gain of $\times 1.24$ with respect to GPU-only implementations. In doing so, a proper intra-node load balancing is required because different devices feature different performance specifications. In this

work, the performance on such a configuration is approached through a hierarchical parallel implementation that allows assigning a single MPI process per node, further distributing the computational workload among multiple devices at lower parallelization levels, as detailed in Chapter 3.

### 4.3.1 Single-node study

The single-node performance of SpMV (left), axpy (middle), and dot (right) in different parallel execution modes on meshes of 2 and 17 million cells are shown in Figure 4.10. In single-device results, sheer green boxes on the bars represent the theoretically achievable performance. Also, the most relevant results are detailed in Table 4.2.

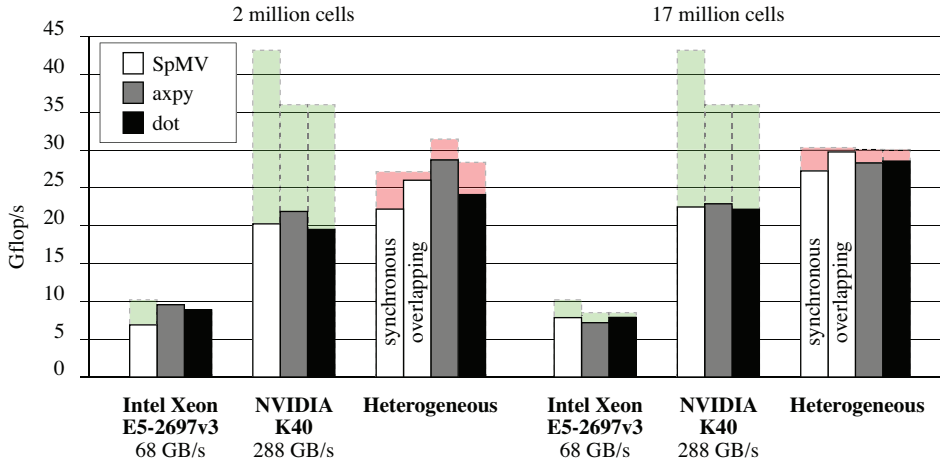


**Figure 4.10:** Single-node performance of HPC$^2$ kernels in different parallel execution modes on meshes of 2 and 17 million cells, on Lomonosov 2.

In line with the preliminary results in Figure 4.1, the single-device performance achieves a rather high fraction of the peak performance. On the CPU side, slightly better performance is obtained on the 2 million cells mesh due to the increase in temporal locality (in this case, the problem size was insufficient to avoid cache reuse). In contrast, the GPU improves on the 17 million cells mesh due to higher occupancy, hiding memory latency overheads.

The results of the heterogeneous co-execution mode demonstrate the capabilities

**Table 4.2:** Detailed single-node performance of HPC$^2$ kernels in different parallel execution modes on Lomonosov 2.

| Kernel | CPU | GPU | Het/ovl | Het/syn | Sum | Gain | Efficiency |
|---|---|---|---|---|---|---|---|
| 2 million mesh cells | | | | | | | |
| SpMV | 6,89 | 20,25 | 26,02 | 22,20 | 27,14 | 1,28 | 0,96 |
| axpy | 9,56 | 21,89 | 28,70 | — | 31,45 | 1,31 | 0,91 |
| dot | 8,87 | 19,51 | 24,12 | — | 28,38 | 1,24 | 0,85 |
| 17 million mesh cells | | | | | | | |
| SpMV | 7,85 | 22,48 | 29,76 | 27,25 | 30,33 | 1,32 | 0,98 |
| axpy | 7,18 | 22,90 | 28,30 | — | 30,08 | 1,24 | 0,94 |
| dot | 7,88 | 22,18 | 22,18 | — | 30,07 | 1,29 | 0,95 |

of HPC$^2$ on hybrid systems. In the heterogeneous histograms, sheer red boxes on top of the bars represent the performance lost compared to the sum of CPU- and GPU-only results. In contrast with the single-node study on MareNostrum 4, intra-node communications appear in this case due to the presence of different virtual memory spaces. Hence, two results of heterogeneous SpMV execution are given: overlapping and synchronous parallel modes. The heterogeneous efficiency is above 90% in all cases, and SpMV achieves the best result. This is because the workload is balanced considering the ratio of SpMV performances.

The benefits of collaborating CPU and GPU in Lomonosov 2 nodes are evident: the performance gain obtained is around $\times 1.3$ in all cases. This gain is even higher than the theoretical because, in this case, the CPU implementation performs more efficiently than its GPU counterpart.

### 4.3.2 Multi-node study

Firstly, Figure 4.11 shows the increase in the halo to inner mesh elements ratio, $\rho$, as the number of compute nodes grows on initial meshes of 17 and 134 million cells. Three blue lines of values 0.8591, 0.1316, and 0.1226 show the theoretical maximum ratio estimated after Equation 3.10 for CPU-only, GPU-only, and heterogeneous execution modes, respectively.

Strong scaling of SpMV (left), axpy (middle), and dot (right) from 1 to 64 hybrid nodes on a mesh of 17 million cells in different parallel execution modes are shown in

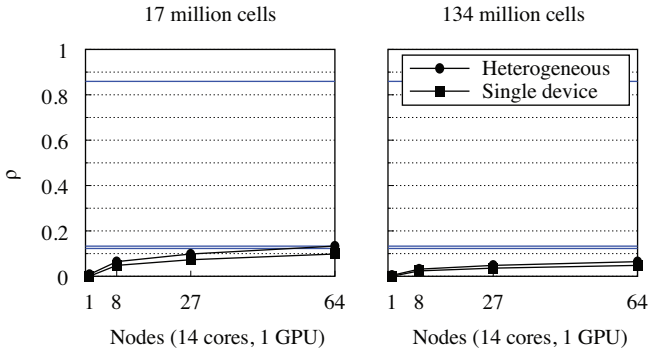


**Figure 4.11:** Halo to inner mesh elements ratio, $\rho$, versus number of compute nodes engaged in different parallel execution modes on Lomonosov 2.

Figure 4.12.

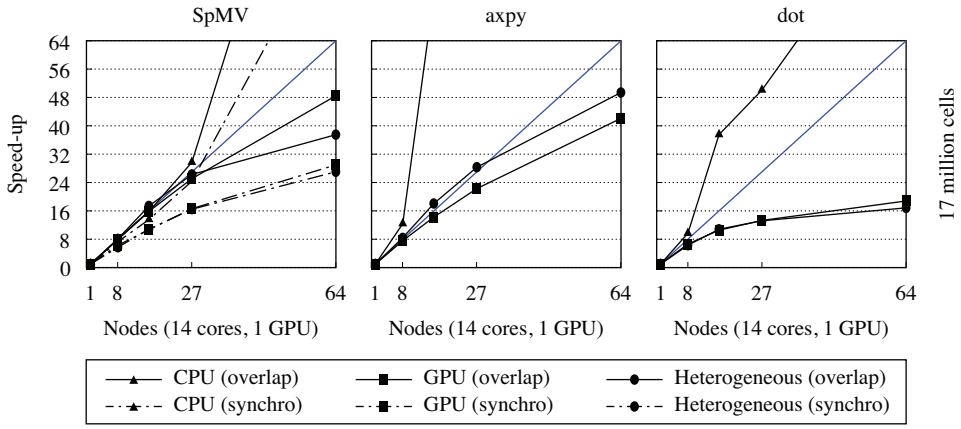


**Figure 4.12:** Strong scaling of HPC$^2$ kernels in different parallel execution modes on Lomonosov 2.

Like MareNostrum 4, the CPU-only mode results in a superlinear behavior starting at around 150,000 mesh cells per core. This superlinear speed up does not vanish in the range from 1 to 64 nodes. It is noted that the network to CPU's main memory bandwidth ratio is very similar in both systems (near 0.10) and that Lomonosov 2's CPU features a larger L3 cache per core. In contrast, the GPU-only mode

presents increased difficulties. While CPUs perform better on smaller loads due to the temporal locality (*i.e.*, lower cache miss ratio), GPU devices worsen due to the lower occupancy not allowing stream multiprocessors to hide memory latency overhead through hardware multi-threading. To top it off, the lower network to GPU's memory bandwidth ratio (near 0.025) limits the scalability of overlapping SpMV to a smaller number of processes or partitions.

Apart from the GPU's performance decay, the heterogeneous mode also suffers from increased complexity in data exchanges. In this mode, the intra-node communications (*i.e.*, data exchanges between devices on the same compute node) resulting from the second-level domain decomposition come into play and have to be handled simultaneously with inter-node exchanges and computations. Therefore, the heterogeneous computing mode is the least scaling mode in terms of number of nodes. However, given the correct operational conditions, which are found at 27 nodes, the heterogeneous mode allows to speed up the execution by a factor of $\times 25$ while taking advantage of all the computing hardware available, gaining an additional factor of $\times 1.3$ with respect to the GPU-only execution.

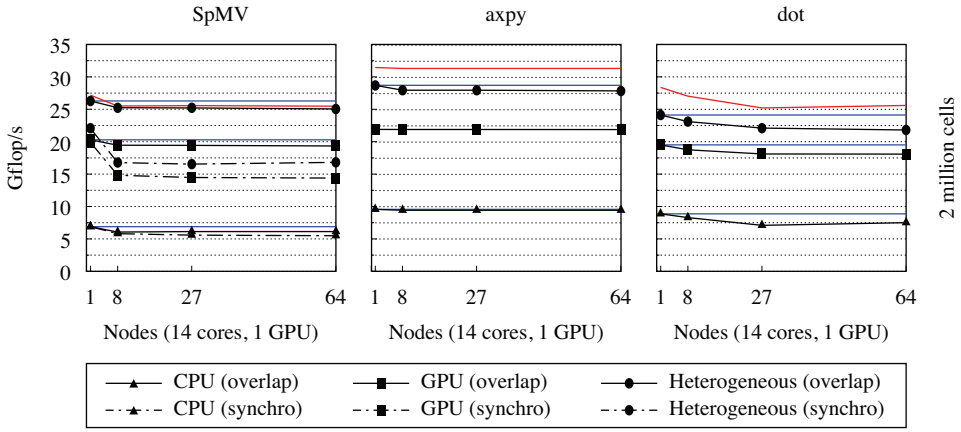


**Figure 4.13:** Weak scaling of HPC$^2$ kernels in different parallel execution modes on Lomonosov 2.

Weak scaling of SpMV (left), axpy (middle), and dot (right) from 1 to 64 hybrid nodes in different parallel execution modes are shown in Figure 4.13. A fixed workload per node of 2 million mesh cells is chosen as in single-node studies. After the occurrence

of inter-node data exchanges, the parallel efficiency decays in all modes. However, keeping the problem size constant also keeps the parallelization overheads constant. Therefore, the results indicate that the HPC$^2$ might be able to execute large-scale simulations on such hybrid supercomputers obtaining a parallel efficiency above 90%.

## 4.4 TSUBAME3.0

TSUBAME3.0 is a GPU-based supercomputer at Tokyo Institute of Technology [6] consisting of 540 compute nodes and providing 12.13 Pflop/s of theoretical peak performance. Its nodes are equipped with two Intel Xeon E5-2680 v4 (14 cores, 2.4 GHz, 4 DDR4-2400 memory channels, 77 GB/s memory bandwidth, 35 MB L3 cache) and four NVIDIA P100 (5.3 Tflop/s peak performance, 16 GB HBM2 memory, 732 GB/s memory bandwidth, PCIe 3.0 x16 at 16 GB/s), and are interconnected via four Intel Omni-Path networks (100 Gb/s). The system architecture is illustrated in Figure 4.14.

The extremely large ratio of aggregated GPU to CPU memory bandwidth per node in TSUBAME3.0, which is above 19, leaves no space for heterogeneous computing: the theoretical gain is $\times 1.05$ in this case, while the intra-node communications are even more complex than in Lomonosov 2 (there are 6 devices per node). Also the ratio of network to GPU's memory bandwidth is the lowest amongst our test machines, 0.017, posing a great challenge to achieving reasonable multi-node scalability.

### 4.4.1 Single-node study

The single-node performance of the HPC$^2$ kernels in different execution modes is shown in Figure 4.15 and detailed in Table 4.3 on a mesh of 17 million cells.

In line with the preliminary results in Figure 4.1, the single-device performance achieves a rather high fraction of the peak performance above 60% with SpMVand near 75% with axpy and dot. Again, intra-node communications appear in this case due to the presence of different virtual memory spaces (one per device).

Even in a single node (with four GPU devices), the synchronous execution struggles in achieving a reasonable throughput due to the GPUs' extremely high memory bandwidth, and achieves a parallel efficiency of only 50%. In other words, the time required for intra-node data exchanges is already equivalent to that of computations,

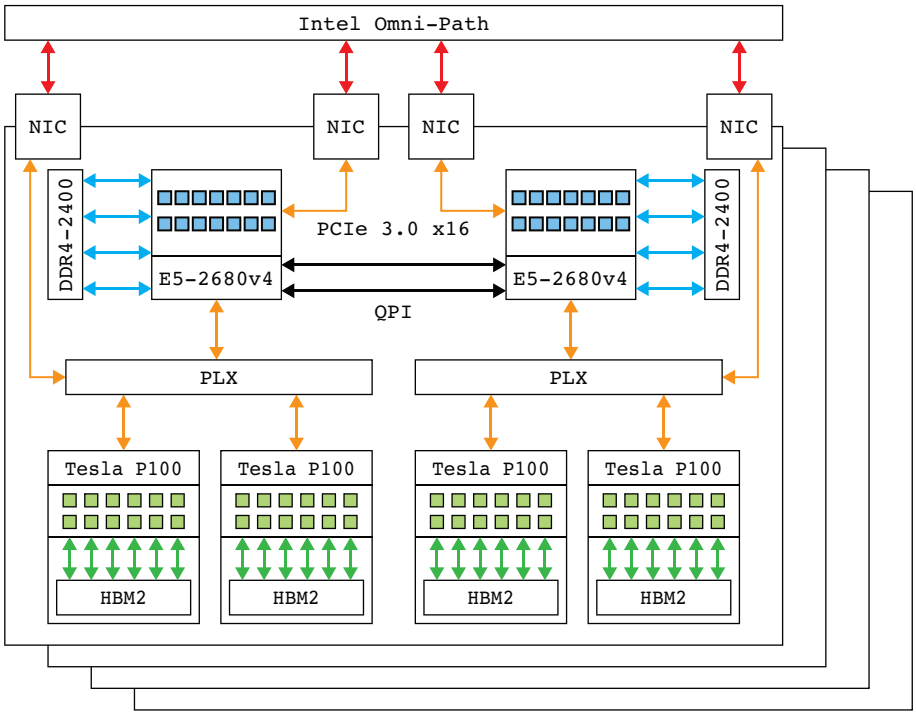


**Figure 4.14:** TSUBAME3.0

and hence it will be even higher in multi-node executions preventing an adequate overlap.

### 4.4.2 Multi-node study

Firstly, Figure 4.16 shows the increase in the halo to inner mesh elements ratio, $\rho$, as the number of compute nodes grows on initial meshes of 17 and 134 million cells. A blue line of values 0.0444 shows the theoretical maximum ratio estimated after Equation 3.10.

Strong scaling of SpMV (left), axpy (middle), and dot (right) from 4 to 256 GPUs (1 to 64 nodes) on a mesh of 134 million cells in different parallel execution modes are shown in Figure 4.17. At first glance, the results reveal that high-bandwidth memory (HBM) is a double-edged sword: computations execute so fast that any other overhead

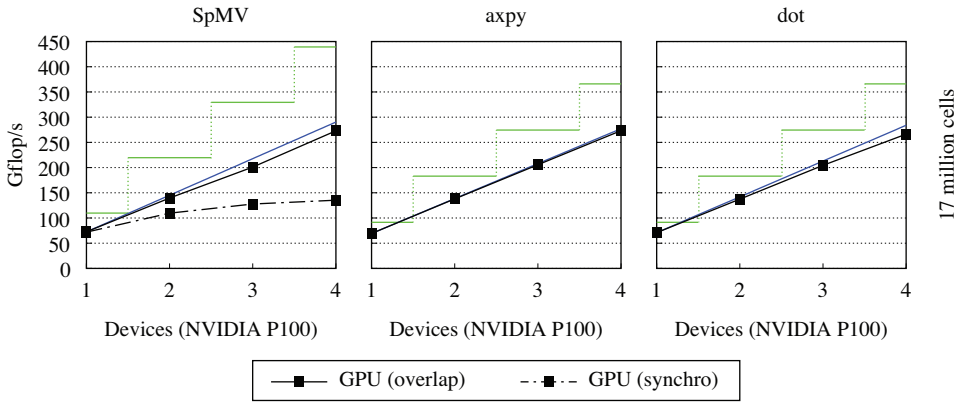


**Figure 4.15:** Single-node performance of $HPC^2$ kernels in different parallel execution modes on TSUBAME3.0

**Table 4.3:** Detailed single-node performance of $HPC^2$ kernels versus number of GPUs in different parallel execution modes on a mesh of 17 million cells, on TSUBAME3.0.

| Kernel | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| SpMV/ovl | 72.571 | 139.532 | 200.915 | 272.928 |
| SpMV/syn | 72.493 | 109.597 | 127.775 | 135.283 |
| axpy | 69.290 | 137.903 | 205.583 | 273.241 |
| dot | 70.983 | 137.410 | 204.391 | 266.333 |

becomes visibly heavier. For instance, the performance of both element-wise kernels, axpy and dot, decays faster than in Lomonosov 2. Even worse is the overlapping SpMV, which does not reach a parallel efficiency of 15%.

However, the results obtained are worse than expected. Considering the system's $\rho$, 0.0444, the SpMV should scale well on 8 nodes and reasonably well on 27, as the curve labeled as *GPU (predicted) indicates*. Therefore, the results suggest that our current implementation suffers from additional penalties not taken into account, such as NUIOA. It is noted that the specifications of other supercomputers such as MareNostrum 4 or Lomonosov 2 did not expose this issue, and hence it was not yet studied.

Weak scaling of SpMV (left), axpy (middle), and dot (right) from 1 to 64 hybrid nodes in different parallel execution modes are shown in Figure 4.18. A fixed workload

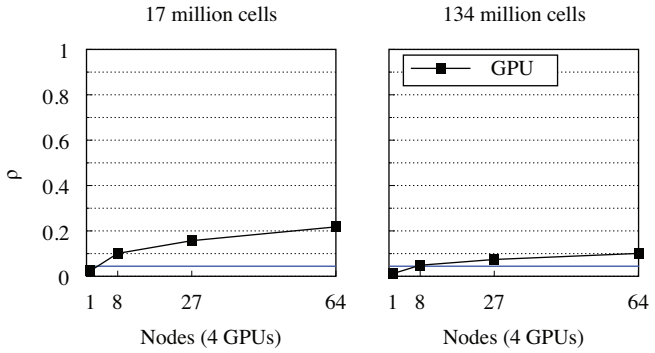


**Figure 4.16:** Halo to inner mesh elements ratio, $\rho$, versus number of compute nodes engaged in different parallel execution modes on TSUBAME3.0.

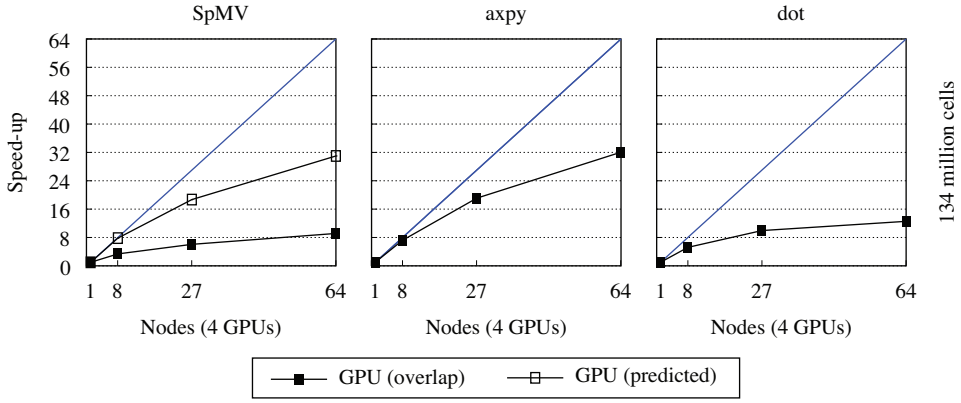


**Figure 4.17:** Strong scaling of HPC$^2$ kernels in different parallel execution modes on TSUBAME3.0.

per node of 17 million mesh cells is chosen, as in single-node studies. In this case, the element-wise kernels scale properly, but the SpMV still struggles and hardly sustains a 55% parallel efficiency. Again, considering the system's $\rho$, the *GPU (predicted)* curve shows the performance that the HPC$^2$ could attain with an optimized data exchange protocol. The prediction indicates that the HPC$^2$ might be able to execute large-scale simulations on such GPU-based supercomputers obtaining a parallel efficiency above 90%.

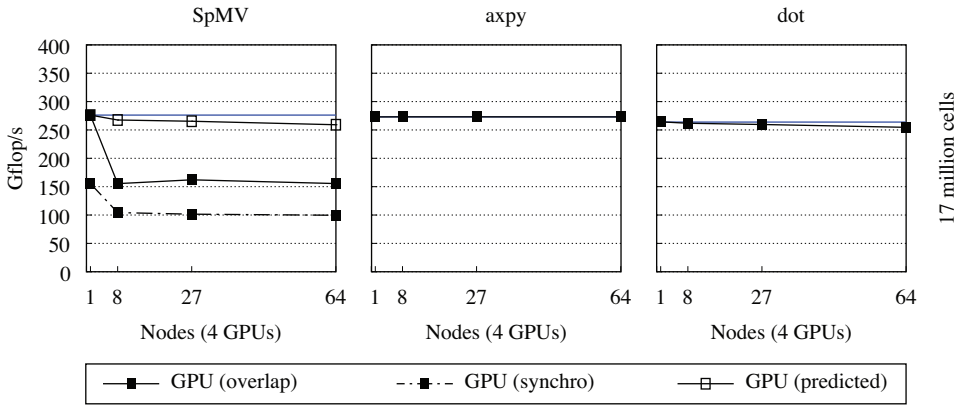


**Figure 4.18:** Weak scaling of $HPC^2$ kernels in different parallel execution modes on TSUBAME3.0.

## References

[1] 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 meshes," *Journal of Computational Physics*, vol. 258, pp. 246–267, 2014.

[2] X. Álvarez-Farré, *Study of an Efficient Parallel Implementation of a CFD Code Based on Algebraic Operators*. PhD thesis, Universistat Politècnica de Catalunya, 2016.

[3] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, jan 2017.

[4] B. S. Center, "Marenostrum iv supercomputer." `https://www.bsc.es/ca/marenostrum/marenostrum/informacio-tecnica`, 2017.

[5] L. M. S. University, "Lomonosov-2 supercomputer." `http://hpc.msu.ru/node/159`, 2014.

[6] G. S. Information and C. Center, "TSUBAME3.0 supercomputer." `https://www.t3.gsic.titech.ac.jp/en/hardware`, 2017.

# 5

# Direct numerical simulation of buoyancy-driven turbulent flows

This chapter demonstrates the capabilities of HPC$^2$ in dealing with large-scale transient CFD simulations.

## 5.1 Air-filled differentially heated cavity at Rayleigh $1.2 \times 10^{11}$

*Main contents of this section are published in [1].*

A DNS of a turbulent air-filled differentially heated cavity (DHC) is chosen to demonstrate the capabilities of the HPC$^2$ framework in dealing with large-scale CFD simulations. Firstly, the case description in conjunction with the numerical methods is detailed. Then, the DHC results are briefly presented.

### 5.1.1 Mathematical model and numerical method

Consider a cavity of height $H$, width $L$, and depth $D$ filled with an incompressible Newtonian viscous fluid of kinematic viscosity $\nu$, thermal diffusivity $\alpha$, and density $\rho$. The geometry of the problem is displayed in Figure 5.1 (left). The Boussinesq approximation is used to account for the density variations. Thermal radiation is neglected. Under these assumptions, the velocity, $\boldsymbol{u}$, and the temperature, $\theta$, are

governed by the following set of dimensionless partial differential equations (PDEs):

$$\partial_t \boldsymbol{u} + (\boldsymbol{u}\cdot\nabla)\boldsymbol{u} = PrRa^{-1/2}\nabla^2\boldsymbol{u} - \nabla p + \boldsymbol{f}, \tag{5.1}$$

$$\partial_t \theta + (\boldsymbol{u}\cdot\nabla)\theta = Ra^{-1/2}\nabla^2\theta, \tag{5.2}$$

where $Pr = \nu/\alpha$, $Ra = (g\beta\Delta\theta H^3)/(\nu\alpha)$, and $\boldsymbol{f} = (0, Pr\theta, 0)$ (Boussinesq approximation) are the Prandtl and Rayleigh number (based on the cavity height), and the body force vector, respectively. Notice that with the reference quantities, $L_{ref} = H$ and $t_{ref} = (H^2/\alpha)Ra^{-1/2}$, the vertical buoyant velocity, $Pr^{1/2}$, and the characteristic dimensionless Brunt-Väisälä frequency, $N$, are independent of the $Ra$. The configuration considered here resembles the experimental set-up performed by D. Saury *et al.* [2] and P. Belleoud *et al.* [3]: the height, $H/L$, and depth, $D/L$, aspect ratios are 3.84 and 0.86, whereas the Rayleigh and Prandtl numbers are $Ra = 1.2\times10^{11}$ and $Pr = 0.71$ (air), respectively. The cavity is subjected to a temperature difference $\Delta\theta$ across the vertical isothermal walls ($\theta(0, y, z) = 0.5$, $\theta(L/H, y, z) = -0.5$). The temperature at the rest of the walls is given by the "fully realistic" boundary conditions proposed in [4]. They are time-independent analytical functions that fit the experimental data of J. Salat *et al.* [5]. The no-slip boundary condition is imposed on the walls.

The governing equations (5.1) and (5.2) are discretized using a symmetry-preserving discretization [6]. Shortly, the temporal evolution of the spatially discrete velocity vector, $\boldsymbol{u}_c$, is governed by the following operator-based finite-volume discretization of Eqs. (5.1):

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

where $\boldsymbol{p}_c \in \mathbb{R}^n$ 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) $\Gamma_{c\to s} \in \mathbb{R}^{m\times 3n}$, $\boldsymbol{u}_s \equiv \Gamma_{c\to s}\boldsymbol{u}_c$. The dimensions of these vectors, $n$ and $m$, are the number of control volumes and faces in the computational domain, respectively. The subindices $c$ and $s$ refer to whether the variables are cell-centered or staggered at the faces. The matrices $\Omega_c^{3d} \in \mathbb{R}^{3n\times 3n}$,

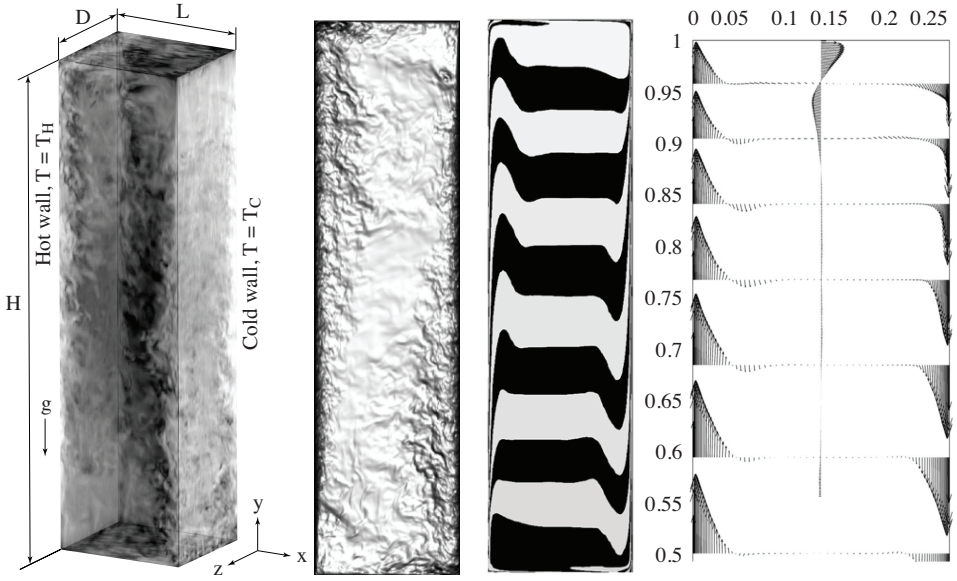


**Figure 5.1:** From left to right: DHC schema, instantaneous schlieren-like snapshot from the DNS and the averaged temperature field at the cavity mid-depth (the isotherms are uniformly distributed between $-0.5$ and $0.5$), the airflow map in the upper part of the cavity.

$\mathsf{C}_c^{3d}(\boldsymbol{u}_s) \in \mathbb{R}^{3n\times 3n}$ and $\mathsf{D}_c^{3d} \in \mathbb{R}^{3n\times 3n}$ are block diagonal matrices given by

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

where $\mathsf{I}_3 \in \mathbb{R}^{3\times 3}$ is the identity matrix. $\mathsf{C}_c(\boldsymbol{u}_s) \in \mathbb{R}^{n\times n}$ 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 $\boldsymbol{\theta}_c \in \mathbb{R}^n$ (see Eq. 5.2) is discretized in the same vein. For a detailed explanation of the spatial discretization, the reader is referred to [6]. Regarding the temporal discretization, a second-order explicit one-leg scheme is used for both the convective and the diffusive terms [7]. Finally, the pressure-velocity coupling is solved by means of a classical fractional step projection method [8]: a predictor velocity, $\boldsymbol{u}_s^p$, is explicitly evaluated without considering the contribution of the pressure gradient. Then, imposing the incompressibility constraint, $\mathsf{M}\boldsymbol{u}_s^{n+1} = \mathbf{0}_c$,

leads to a Poisson equation for $\tilde{\boldsymbol{p}}_c^{n+1}$ to be solved once each time-step,

$$\mathsf{L}\tilde{\boldsymbol{p}}_c^{n+1} = \mathsf{M}\boldsymbol{u}_s^p \qquad \text{with} \qquad \mathsf{L} = -\mathsf{M}\Omega_s^{-1}\mathsf{M}^T, \tag{5.4}$$

where $\tilde{\boldsymbol{p}}_c = \Delta t \boldsymbol{p}_c$ and the discrete Laplacian operator, $\mathsf{L}$, is represented by a symmetric negative semi-definite matrix.

In summary, the method is based on only five basic (linear) operators: the cell-centered and staggered control volumes, $\Omega_c$ and $\Omega_s$, the matrix containing the face normal vectors, $\mathsf{N}_s$, the cell-to-face scalar field interpolation, $\Pi_{c\to s}$ and the divergence operator, $\mathsf{M}$. Once these operators are constructed, the rest follows straightforwardly.

The algorithm to solve one time-integration step is outlined in Algorithm 3. Regarding the time-integration scheme (steps 2 and 7 in Algorithm 3), and without loss of generality, a second-order Adams–Bashforth has been adopted here. Since it is a fully explicit scheme, a CFL condition-like condition is required in order to keep the numerical scheme inside the stability region [7]. This necessarily leads to rather small time-steps, $\Delta t$, and subsequently to a good initial guess for the Poisson equation justifying the use of a relatively simple linear solver for the Poisson equation (step 3 in Algorithm 3) to maintain the norm of the divergence of the velocity field, $\mathsf{M}\boldsymbol{u}_s^{n+1}$, at a low enough level [9]. Furthermore, since the matrix, $-\mathsf{L}$, is symmetric and positive-definite, a preconditioned CG is used with a simple SpMV-based preconditioner (either the Jacobi or the approximate inverse). In conclusion, the overall Algorithm 3 relies on the set of three basic linear algebra operations: SpMV, axpy, and dot.

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

$$\mathsf{C}_c\left(\boldsymbol{u}_s\right) = \mathsf{MU}\Pi_{c\to s}, \tag{5.5}$$

**Algorithm 3** Time-integration step

1: Compute the convective, the diffusive and the source term of momentum Eq.(5.1):
$\mathsf{R}\left(\boldsymbol{u}_s^n, \boldsymbol{u}_c^n, \boldsymbol{\theta}_c^n\right) \equiv -\mathsf{C}_c^{3d}\left(\boldsymbol{u}_s^n\right)\boldsymbol{u}_c^n - \mathsf{D}_c^{3d}\boldsymbol{u}_c^n + \boldsymbol{f}_c(\boldsymbol{\theta}_c^n)$
2: Compute the predictor velocity:
$\boldsymbol{u}_c^p = \boldsymbol{u}_c^n + \Delta t\left\{\frac{3}{2}\mathsf{R}\left(\boldsymbol{u}_s^n, \boldsymbol{u}_c^n\right) - \frac{1}{2}\mathsf{R}\left(\boldsymbol{u}_s^{n-1}, \boldsymbol{u}_c^{n-1}\right)\right\}$
3: Solve the Poisson equation given in Eq.(5.4):
$\mathsf{L}\tilde{\boldsymbol{p}}_c^{n+1} = \mathsf{M}\boldsymbol{u}_s^p$ 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.(5.2):
$\mathsf{R}_\theta\left(\boldsymbol{u}_s^n, \boldsymbol{\theta}_c^n\right) \equiv -\mathsf{C}_c\left(\boldsymbol{u}_s^n\right)\boldsymbol{\theta}_c^n - Pr^{-1}\mathsf{D}_c\boldsymbol{\theta}_c^n$
7: Compute temperature at the next time-step:
$\boldsymbol{\theta}_c^{n+1} = \boldsymbol{\theta}_c^n + \Delta t\left\{\frac{3}{2}\mathsf{R}_\theta\left(\boldsymbol{u}_s^n, \boldsymbol{\theta}_c^n\right) - \frac{1}{2}\mathsf{R}_\theta\left(\boldsymbol{u}_s^{n-1}n, \boldsymbol{\theta}_c^{n-1}\right)\right\}$

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 $\Pi_{c\to s}$ is the above-mentioned cell-to-face scalar field interpolation. Computing the convective term using this form seems inefficient since three consecutive SpMV are required. However, this naive approach can be easily improved by noticing that $\mathsf{MU}$ can be precomputed since $\mathsf{U}$ is a diagonal matrix (that changes every time-step); therefore, the product $\mathsf{MU}$ is simply a re-scaling of columns. Moreover, this new matrix is shared by all the convective operators regardless of the quantity being advected. Finally, the cell-to-face interpolation operator, $\Pi_{c\to s}$, will depend on the particular choice for the spatial numerical scheme.

Table 5.1 sums up the number of times that each kernel is called at the different steps of Algorithm 3.

### 5.1.2 Execution of a large-scale simulation

Since no subgrid-scale model is used in the computation, the grid resolution and the time-step, $\Delta t$, have to be fine enough to resolve all the relevant turbulence scales. Furthermore, the starting time for averaging, $t_0$, and the time-integration period, $\Delta t_{avg}$, must also be long enough to properly evaluate the flow statistics. The procedure followed to verify the simulation results is analogous to our previous DNS work [10,11].

**Table 5.1:** Number of times that each basic operation is performed in the numerical algorithm.

| Step of Algorithm 3 | SpMV | axpy | dot |
|---|---|---|---|
| 1. Compute the convective–difusive terms | 12 | 0 | 0 |
| 2. Predictor velocity | 0 | 6 | 0 |
| 3. Poisson equation (r.h.s) | 4 | 2 | 0 |
| *. Poisson equation (per iteration) | 2 | 3 | 2 |
| 4. Velocity correction (staggered) | 1 | 1 | 0 |
| 5. Velocity correction (centred) | 3 | 3 | 0 |
| 6. Compute the convective–difusive terms | 4 | 0 | 0 |
| 7. Compute temperature at next time step | 0 | 2 | 0 |
| Total outside Poisson solver | 24 | 14 | 2 |

In this case, the averages over the three statistically invariant transformations (time, mid-depth plane and central point symmetry) have been carried out for all fields and the grid points in the three wall-normal directions are distributed using a hyperbolic-tangent function, *i.e.* for the $x$-direction $x_i = \frac{1}{2}\left(1 + \frac{\tanh\{\gamma_x(2(i-1)/N_x - 1)\}}{\tanh\gamma_x}\right)$. For details about the physical and numerical parameters see Table 5.2. Hereafter, the angular brackets operator $\langle\cdot\rangle$ denotes averaged variables.

**Table 5.2:** Physical and numerical simulation parameters of the DNS of the turbulent DHC displayed in Figure 5.1. From left to right: number of control volumes and concentration factors for each spatial direction, the size of the first off-wall control volume in the $x$-direction (also in wall-units), the non-dimensional time-step, the starting time for averaging and the time-integration period.

| $N_x$ | $N_y$ | $N_z$ | $\gamma_x$ | $\gamma_y$ | $\gamma_z$ | $(\Delta x)_{\min}$ | $(\Delta x)^+_{\min}$ | $\Delta t$ | $t_0$ | $\Delta t_{avg}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 450 | 900 | 256 | 2 | 1 | 1 | $4.28 \times 10^{-5}$ | $\lesssim 0.5$ | $3.65 \times 10^{-4}$ | $\approx 300$ | $\approx 300$ |

The described algorithm has been implemented into the HPC$^2$ framework using only three basic linear algebraic kernels: sparse matrix-vector product, linear combination of vectors, and dot product. Prior to the execution of such a large simulation, a performance study has been conducted on MareNostrum 4 supercomputer at Barcelona Supercomputing Center (BSC). Specifically, a strong scaling analysis from 1 to 200 nodes consisting of measuring 40 time-steps only. It is noted that the entire study requires less than 100 core hours. Figure 5.2 shows the results of the analysis and indicates that the optimal number of nodes is 120. In this conditions, the superlinear

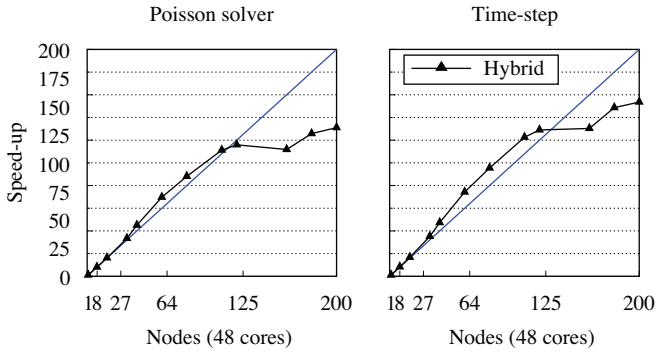


**Figure 5.2:** Strong scaling of the Poisson solver (left) and outside it (right) on a mesh of 103 million cells, on MareNostrum 4.

behavior is still observed and allows the HPC$^2$ to sustain 3.24 Tflop/s, and to achieve an effective $\times 129$ speed up.

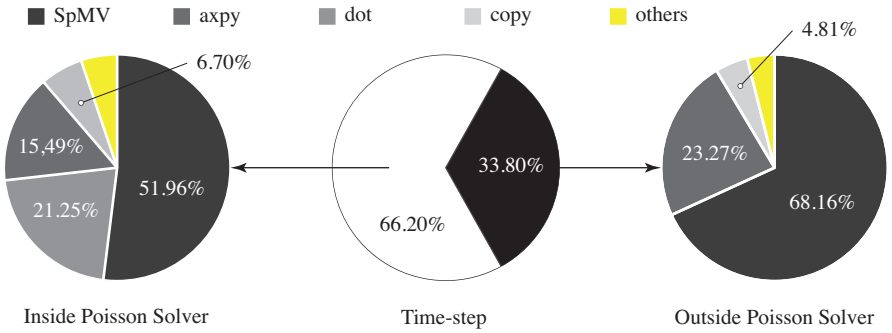


**Figure 5.3:** Relative time spent per kernel and time-step inside the Poisson solver (left) and outside it (right)

The simulation has been successfully executed on 120 nodes (5,760 Intel Xeon 8160 cores). The average elapsed time per time-step is $0.07175s$, thus the total simulation requires around 27 hours to finish (1,369,864 time-steps), or 155,520 core hours. Figure 5.3 shows the relative time spent in each operation and time-step (Algorithm 3), and confirms that the overall computing cost is only depending on three algebraic kernels. Hence, the cost and efficiency estimation for large-scale simulations on virtually any modern supercomputer can be directly extrapolated from the performance studies

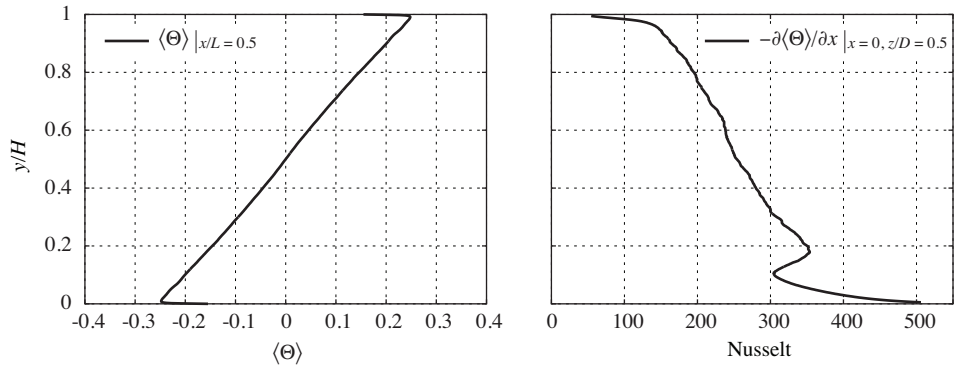


**Figure 5.4:** Left: average temperature profiles at the cavity mid-depth at mid-width. Right: averaged Nusselt number at the cavity mid-depth.

in our previous works [1, 12, 13], or in Chapter 4. This allows avoiding nonsense waste of computational resources in repetitive performance studies. Our fully-portable, algebra-based framework for heterogeneous computing has demonstrated its great potential for the simulation of turbulent flows on virtually any modern supercomputer.

### 5.1.3 Results and discussion

Instantaneous flow fields displayed in Figure 5.1 illustrate the inherent flow complexity of this configuration. Namely, the vertical boundary layers remain laminar only in their upstream part up to the point where the waves traveling downstream grow up enough to disrupt the boundary layers ejecting large unsteady eddies. An accurate prediction of the flow structure in the cavity lies on the ability to correctly locate the transition to turbulence while the high sensitivity of the thermal boundary layer to external disturbances makes it difficult to predict (see results for a similar DHC configuration in [14] and [11], for instance). In this case, the transition occurs around $y \approx 0.2$ (see the peak of the averaged local Nusselt number displayed in the right part of Figure 5.1).

The average temperature field and the airflow map are displayed in Figure 5.1 (right). The cavity is almost uniformly stratified with a dimensionless stratification of $S \approx 0.45$ (see Figure 5.4, left). This value is in a rather good agreement with the experimental results obtained by D. Saury *et al.* [2] ($S = 0.44 \pm 0.03$ with $\epsilon = 0.1$ and

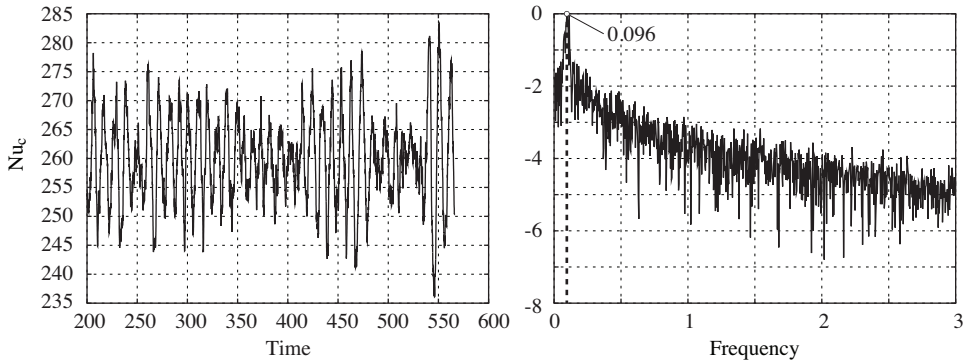


**Figure 5.5:** Time evolution of the Nusselt number at the vertical mid-plane (left) and its normalized density power spectrum (right).

$S = 0.57 \pm 0.03$ with $\epsilon = 0.6$, where $\epsilon$ is the wall emissivity). The averaged Nusselt number at the cavity mid-depth is displayed in Figure 5.4 (right). The profile is again rather similar to the experimental results obtained by D. Saury *et al.* [2]. In this case, the transition point occurs at a slightly more upstream position. The peak of the averaged local Nusselt number is located at $y \approx 0.2$ whereas in the experimental results this point is at $y \approx 0.3$. Integrating the averaged local Nusselt number over the $y$-direction, the overall Nusselt is determined. In this case, $\langle Nu \rangle = 259.2$, a slightly higher value that the one obtained by D. Saury *et al.* [2], *i.e.*, $\langle Nu \rangle = 231 \pm 30$ but very similar to the value obtained by means of LES, *i.e.*, $\langle Nu \rangle = 254$ (see Figure 11 in [2]).

Another important feature of this kind of configuration is the presence of internal waves. Although in the cavity core the averaged velocity (and its fluctuations) are much smaller compared with those observed in the vertical boundary layers, simulations show that in this region isotherms oscillate around the mean horizontal profile. As mentioned-above, the cavity core remains well stratified (see Figure 5.1 and Figure 5.4, left); therefore, this phenomenon can be attributed to internal waves. This can be confirmed by analyzing the Nusselt number through the vertical mid-plane, $Nu_c$. The time evolution and the normalized density power spectrum are respectively displayed in Figures 5.5. The peak in the spectrum is located at 0.096 which is in a good agreement with the dimensionless Brunt-Väisälä frequency, $N = (SPr)^{0.5}/(2\pi)$, where $S$ is the dimensionless stratification of the time-averaged temperature, *i.e.* $N \approx 0.09$.

Both values are very similar confirming that internal waves are permanently excited by the eddies ejected form the vertical boundary layer. Detailed results including turbulent statistics can be downloaded in the following link [15].

## 5.2 Towards exascale simulations of the ultimate regime

Buoyancy-driven flows have always been an important subject of scientific studies with numerous applications in environment and technology. The most famous example thereof is the thermally driven flow developed in a fluid layer heated from below and cooled from above, *i.e.* the Rayleigh-Bénard convection (RBC). It constitutes a canonical flow configuration that resembles many natural and industrial processes, such as solar thermal power plants, indoor space heating and cooling, flows in nuclear reactors, electronic devices, and convection in the atmosphere, oceans and the deep mantle.

In the last decades significant efforts, both numerically and experimentally, have been directed at investigating the mechanisms and the detailed scaling behavior of the Nusselt numbers as a function of $Ra$ and $Pr$ in the general form $Nu \propto Ra^{\gamma} Pr^{\beta}$. In this regard, Figure 5.6 shows the predictions of the $Nu$-number based on the classical Grossmann-Lohse (GL) theory [16] and its subsequent corrections [17, 18] where different scaling regimes, characterized by their corresponding exponents $\gamma$ and $\beta$, are identified. Assuming this power-law scalings and following the same reasonings as in Ref. [19] leads to the estimations for the number of grid points shown in Figure 5.7 (top). This corresponds to mesh resolution requirements for DNS and clearly explain why nowadays DNS of RBC is still limited to relatively low $Ra$-numbers. However, many of the above-mentioned applications are governed by much higher $Ra$ numbers, located in the region of the $\{Ra, Pr\}$ phase space where the thermal boundary layer becomes turbulent (see the black dash-dotted line in Figure 5.7). This region corresponds to the so-called asymptotic Kraichnan or ultimate regime of turbulence, with $\gamma = 1/2$. Experimentally, power-law dependencies of heat flux with exponents between $1/4$ and $1/3$ have been measured; however, despite the great efforts devoted, no clear evidence of the Kraichnan regime has been observed yet. Reaching such $Ra$-numbers while keeping the basic assumptions (Boussinesq approximation, adibaticity of the closing walls, isothermal horizontal walls, perfectly smooth surfaces...) is a very hard task. To that end, experimental set-ups using cryogenic helium gas as the working have been

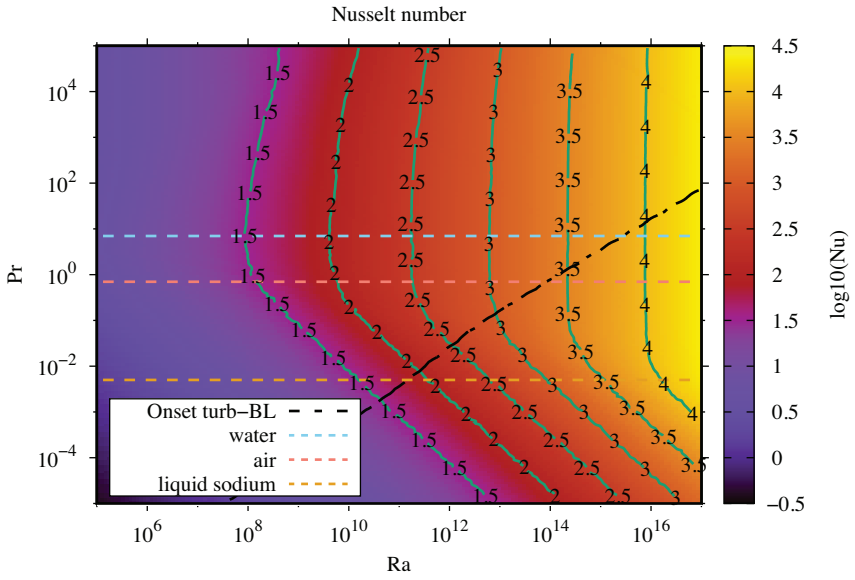


**Figure 5.6:** Estimation of the Nusselt number of a RBC in the $\{Ra, Pr\}$ phase space given by the classical GL theory [16] and its subsequent corrections [17]. Green solid isolines represent the $log10$ of the Nusselt. Three dashed horizontal lines correspond to three different working fluids: water ($Pr = 7$), air ($Pr = 0.7$) and liquid sodium ($Pr = 0.005$). Black dash-dotted line is an estimation for the onset of turbulence in the thermal boundary layer.

conducted. However, despite the great effort devoted their conclusions are unclear due to the highly scattered results [17, 18].

In this context, we may turn to large-eddy simulation (LES) to predict the large-scale behavior of incompressible turbulent flows driven by buoyancy at very high $Ra$-numbers. In LES, the large-scale motions are explicitly computed, whereas the effects of small-scale motions are modeled. Since the advent of CFD, many subgrid-scale (SGS) models have been proposed and successfully applied to a wide range of flows. However, there still exits inherent difficulties in the proper modelization of the SGS heat flux. This was analyzed in detail in the PRACE project entitled "Exploring new frontiers in Rayleigh-Bénard convection" (33.1 millions of CPU hours on MareNostrum4 in 2018-2019), where DNS simulations of air-filled ($Pr = 0.7$) RBC up to $Ra = 10^{11}$ were carried out using meshes up to $5.6 \times 10^9$ grid points (see dots displayed in Figure 5.7, top). These results shed light into the flow topology and

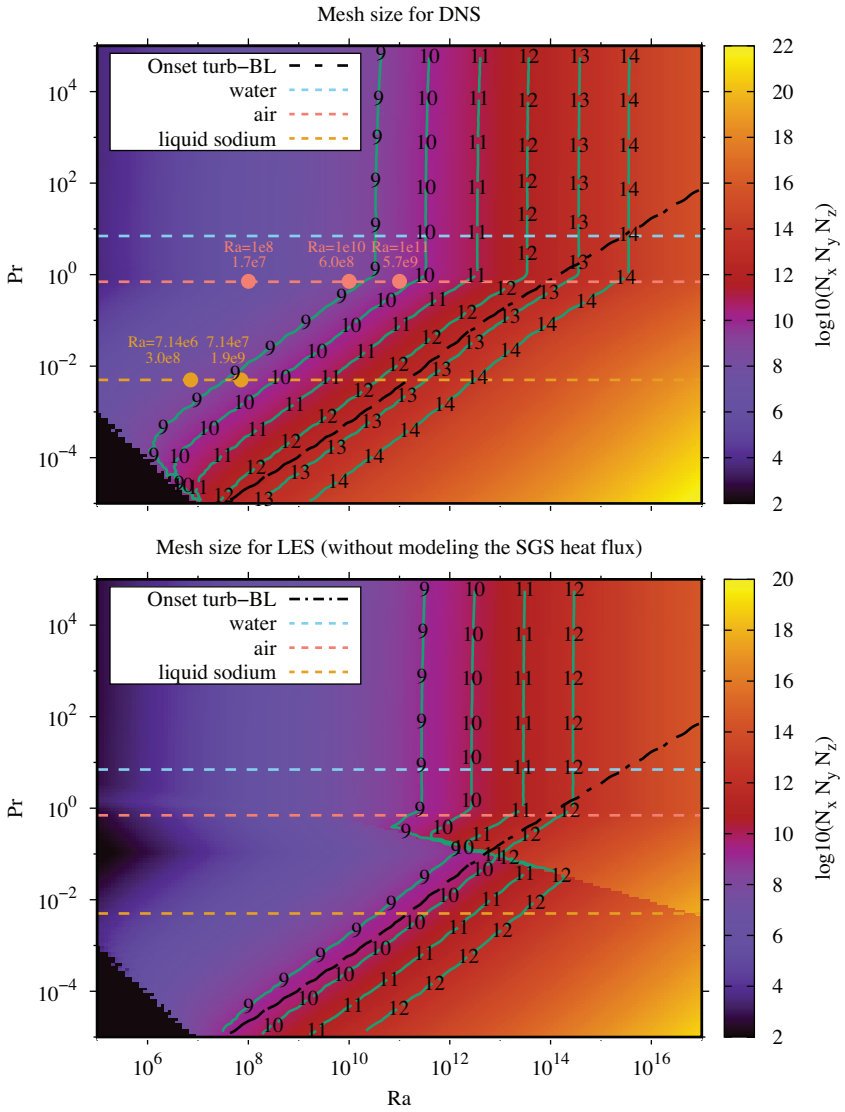


**Figure 5.7:** Estimation of the mesh sizes for DNS (top) and LES (bottom) simulations of RBC in the $\{Ra, Pr\}$ phase space. Estimations for LES assume that thermal scales are fully resolved, *i.e.* no SGS heat flux model is needed. Green solid isolines represent the $log10$ of the total number of grid points. Three dashed horizontal lines correspond to three different working fluids: water ($Pr = 7$), air ($Pr = 0.7$) and liquid sodium ($Pr = 0.005$). Dots displayed on top of these lines correspond to the DNS simulations carried out in previous studies [19–21]. Black dash-dotted line is an estimation for the onset of turbulence in the thermal boundary layer.

the small-scale dynamics which are crucial in constructing the turbulent wind and energy budgets [20]. Moreover, it also provided new insights into the preferential alignments of the SGS and its dependence with the $Ra$-numbers [21], highlighting that the modelization of the SGS heat flux is the main difficulty that (still) precludes reliable LES of buoyancy-driven flows at (very) high Ra-numbers. This inherent difficulty can be by-passed by carrying out simulations at low-Prandtl numbers. In this case, the ratio between the Kolmogorov length scale and the Obukhov-Corrsin length scale (the smallest scale for the temperature field) is given by $Pr^{3/4}$; therefore, for instance, at $Pr = 0.005$ (liquid sodium) we have a separation of more than one decade. Hence, it is possible to combine an LES simulation for the velocity field (momentum equation) with the numerical resolution of all the relevant scales of the thermal field. Results obtained in Ref. [21] suggest that accurate predictions of the overall $Nu$ can be obtained with meshes significantly coarser than for DNS (*e.g.* in practice for $Pr = 0.005$ we can expect mesh reductions in the range $10^2$-$10^3$ for the total number of grid points). This can be clearly observed in Figure 5.7 (bottom), where mesh resolution for LES is given with the assumption that thermal scales are fully resolved. In conclusion, in the near future, we plan to carry out LES simulations at low-$Pr$ with the final goal to hit the ultimate regime of turbulence with meshes in the range from $10^{10}$ to $10^{11}$ using the HPC$^2$ framework developed in the context of this thesis.

## References

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

[2] D. Saury, N. Rouger, F. Djanna, and F. Penot, "Natural convection in an air-filled cavity: Experimental results at large Rayleigh numbers," *International Communications in Heat and Mass Transfer*, vol. 38, pp. 679–687, 2011.

[3] P. Belleoud, D. Saury, and D. Lemonnier, "Coupled velocity and temperature measurements in an air-filled differentially heated cavity at Ra=1.2E11," *International Journal of Thermal Sciences*, vol. 123, pp. 151–161, 2018.

[4] A. Sergent, P. Joubert, S. Xin, and P. L. Le Quéré, "Resolving the stratification discrepancy of turbulent natural convection in differentially heated air-filled cavities Part II: End walls effects using large eddy simulation," *International Journal of Heat and Fluid Flow*, vol. 39, pp. 15–27, 2013.

[5] J. Salat, S. Xin, P. Joubert, A. Sergent, F. Penot, and P. L. Le Quéré, "Experimental and numerical investigation of turbulent natural convection in a large air-filled cavity," *International Journal of Heat and Fluid Flow*, vol. 25, pp. 824–832, 2004.

[6] 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 meshes," *Journal of Computational Physics*, vol. 258, pp. 246–267, 2014.

[7] F. X. Trias and O. Lehmkuhl, "A self-adaptive strategy for the time integration of Navier–Stokes equations," *Numerical Heat Transfer, Part B: Fundamentals*, vol. 60, pp. 116–134, 2011.

[8] A. J. Chorin, "Numerical solution of the Navier–Stokes equations," *Mathematics of Computation*, vol. 22, no. 104, pp. 745–762, 1968.

[9] 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.

[10] F. X. Trias, A. Gorobets, M. Soria, and A. Oliva, "Direct numerical simulation of a differentially heated cavity of aspect ratio 4 with $Ra$-number up to $10^{11}$ – Part I: Numerical methods and time-averaged flow," *International Journal of Heat and Mass Transfer*, vol. 53, pp. 665–673, 2010.

[11] F. X. Trias, A. Gorobets, C. D. Pérez-Segarra, and A. Oliva, "DNS and regularization modeling of a turbulent differentially heated cavity of aspect ratio 5," *International Journal of Heat and Mass Transfer*, vol. 57, pp. 171–182, 2013.

[12] X. Álvarez-Farré, A. Gorobets, and F. X. Trias, "A hierarchical parallel implementation for heterogeneous computing. Application to algebra-based CFD simulations on hybrid supercomputers," *Computers & Fluids*, vol. 214, p. 104768, 2021.

[13] N. Valle, X. Álvarez-Farré, A. Gorobets, J. Castro, A. Oliva, and F. X. Trias, "On the implementation of flux limiters in algebraic frameworks," *Computer Physics Communications*, vol. 271, p. 108230, 2022.

[14] D. G. Barhaghi and L. Davidson, "Natural convection boundary layer in a 5:1 cavity," *Physics of Fluids*, vol. 19, no. 12, p. 125106, 2007.

[15] X. Álvarez-Farré, A. Gorobets, F. X. Trias, R. Borrell, and G. Oyarzún, "The DNS results presented in this paper are publicly available." `http://www.cttc.upc.edu/downloads/DHC_Ra1_2e11`, May 2017.

[16] S. Grossmann and D. Lohse, "Scaling in thermal convection: a unifying theory," *Journal of Fluid Mechanics*, vol. 407, pp. 27–56, 2000.

[17] R. J. A. M. Stevens, E. P. van der Poel, S. Grossmann, and D. Lohse, "The unifying theory of scaling in thermal convection: the updated prefactors," *Journal of Fluid Mechanics*, vol. 730, pp. 295–308, 2013.

[18] S. Bhattacharya, M. K. Verma, and R. Samtaney, "Revisiting Reynolds and Nusselt numbers in turbulent thermal convection," *Physics of Fluids*, vol. 33, p. 015113, 2021.

[19] 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.

[20] F. Dabbagh, F. X. Trias, A. Gorobets, and A. Oliva, "Flow topology dynamics in a three-dimensional phase space for turbulent Rayleigh-Bénard convection," *Physical Review Fluids*, vol. 5, p. 024603, 2020.

[21] F. Trias, F.Dabbagh, A.Gorobets, and C.Oliet, "On a proper tensor-diffusivity model for large-eddy simulation of buoyancy-driven turbulence," *Flow, Turbulence and Combustion*, vol. 105, pp. 393–414, 2020.

# 6

# Conclusions

Inspired by the formulation reviewed in Chapter 2, an algebra-based implementation model for numerical simulation frameworks is presented. By casting discrete operators and mesh functions into sparse matrices and vectors, it has been shown that all the calculations in a typical CFD algorithm for the DNS and LES of incompressible turbulent flows boil down to the following basic linear algebra subroutines: SpMV, axpy, and dot. In this algebraic approach, the kernel code shrinks to dozens of lines; the portability becomes natural, and maintaining OpenMP, OpenCL, or CUDA implementations takes minor effort.

The DNS conducted in Chapter 5 has validated the approach and demonstrated the computational capabilities of the framework. Using HPC$^2$ library, the implementation of Algorithm 3 is reduced to a few lines of code that could seem similar to the friendly style of MATLAB, but internally these are capable of massively parallel heterogeneous computing. It is noted that the particular choice of the time-integration scheme or the SLAE solver is not relevant to the scope of this thesis. Indeed, the HPC$^2$ is not a particular CFD solver but a framework designed for implementing algorithms and solvers.

During the design and development process, we have faced some challenges and obstacles to overcome—namely, the low arithmetic intensity of algebraic kernels and the problem of implementing non-linear terms. After relatively simple generalizations, we have surpassed the challenges giving rise to a much more competitive numerical simulation framework for heterogeneous (super)computing that applies to many fields

of computational physics and mathematics.

**Implementation model**

At first, an abstract model of hybrid supercomputers has been presented. This model has taken five assumptions that apply to most scientific computing codes to simplify the parametrization. As a result, handy expressions to estimate the parallel behavior of numerical applications on hybrid supercomputers have been derived (see Equations 3.8 and 3.9) following reasoning similar to the classical roofline. The implementation highlights are outlined below:

- A hierarchical parallel implementation has been proposed to deal with hybrid supercomputers. In this approach, the multilevel workload distribution described in Section 3.3 minimizes the number of processes participating in MPI exchanges and the global size of the messages, taking full advantage of the intra-node topology and the shared-memory parallel processors. Moreover, this advantage will only strengthen as the memory hierarchies of modern supercomputers become more complex.

- Single and double overlapping execution schemes have been described in detail in Section 3.4. These execution schemes allow for processing kernel execution and halo exchanges in DM parallelization simultaneously.

- A portable implementation model has been proposed in Section 3.5. It is based on three types of objects: actuator, container, and shaper. In this approach, all the architecture-specific implementation is encapsulated in a single low-level, pure virtual class, the virtual unit, which can be specialized for different architectures. It has been shown that implementing a new virtual unit is sufficient to port the entire numerical simulation framework.

- A novel approach based on exploiting nested Kronecker products using the SpMM kernel has been proposed in Section 3.6.1. Given a sufficient number of vectors, theoretical speed-ups up to $\times 15$ are expected from using the SpMM instead of SpMV, as shown in Figure 3.7.

- A rather simple but effective generalization of the axpy and dot kernels has been proposed to enable the implementation of non-linear terms as shown in

Section 3.6.2. The application of this approach to a canonical case, a three-dimensional deformation problem, has been validated and its performance studied in detail in Appendix A.

**Performance study**

An in-depth performance study on several modern supercomputers has been conducted in Chapter 4. Recall that any algorithm based on HPC$^2$ relies only on three types of algebraic routines: SpMV, kbin (the generalization of axpy), and kred (the generalization of dot). The roofline model has been proven accurate in predicting the performance of such algebraic routines, in contrast to the expression to evaluate the maximum ratio of halo to inner mesh elements (*cf.* Equation 4.1). While there have been no cases in which good performance exceeding the theoretical maximum has been obtained, which is fine, the parallel efficiency in some cases has been much lower than expected. Probably, this is due to poor implementation of the data exchange protocol and is left for future work. The performance highlights are listed below:

- MareNostrum 4 has been found the best performing supercomputer among the three we have studied. The new NUMA-aware implementation with proper thread binding and NUMA-placement of data has increased the performance about $\times 1.8$ times compared to the previous implementation with dynamic scheduling. The hierarchical parallel implementation has demonstrated the best parallel efficiency in both strong and weak scaling tests. It allows exploiting both distributed- and shared-memory parallelism to minimize the number of processes participating in reductions or data exchanges and the number and size of the messages required for DM parallelization.

- On Lomonosov 2, the heterogeneous co-execution mode has demonstrated nearly 100% heterogeneous performance on single-node tests. The benefits of collaborating CPU and GPU in Lomonosov 2 nodes are evident: the performance gain obtained is around $\times 1.3$ in all cases. Besides, weak scaling results for 2 million cells per node suggest that our framework is able to execute extreme-scale simulations on such hybrid supercomputers sustaining a parallel efficiency above 90%. However, the heterogeneous mode struggles in strong scaling tests. On the one hand, the CPU is more involved in data exchanges than in CPU-only

modes. On the other hand, the performance of GPUs on smaller loads decreases due to lower occupancy.

- TSUBAME3.0 has been erected as the Achilles heel of algebraic implementations. Its fat nodes feature the largest ratio of aggregated GPU to CPU memory bandwidth per node, 19, and also the lowest ratio of network to GPU's memory bandwidth, 0.017, posing a great challenge to achieving reasonable multi-node scalability. Our predictions after Equation 4.1 indicate that the SpMV might be able to sustain rather good performance in weak scaling tests, suggesting that our current implementation suffers from additional penalties not taken into account, such as NUIOA

This thesis is has been conducted in the CTTC laboratory, at the Technical University of Catalonia. Owing to CTTC's acknowledged tradition in CFD, and without loss of generality, most of this work has been targeted to solve large-scale transient CFD simulations of incompressible turbulent flows. In this regard, the lines of future work are clear: to enable larger and faster simulations by means of code optimization. First, we want to explore in depth the use of Kronecker-based formulations (see Section 3.6.1) applied to mesh symmetries, parallel-in-time methods, or multiple transport equations, among others. Then, we want to revisit the data exchange protocol in $HPC^2$ and design a new one that exploits all the buses efficiently while taking care of the NUIOA-factor. We believe that the upcoming HPC systems will feature even more complex memory configurations. Finally, we have to prepare the code for dealing efficiently with different types of discretizations and meshes, integrating more sparse matrix storage formats and reordering methods.

# Appendix A

# Implementation flux limiters in HPC$^2$

*Main contents of this appendix are published in [1].*

This appendix extends the implementation of non-linear terms introduced in Section 3.6.2 detailing the implementation of a flux limiter scheme in HPC$^2$ and studying its computational performance on different supercomputers to demonstrate its portability.

## A.1 Algebraic implementation of flux limiters

Flux limited schemes can be stated in the following form [2]:

$$\theta_f = \frac{\theta_C + \theta_D}{2} + \frac{\Psi(r) - 1}{2} \left(\theta_D - \theta_C\right), \tag{A.1}$$

where $\theta_C$ and $\theta_D$ stand for the centered and downwind values of $\theta$ according to the velocity field $\boldsymbol{u}$, $\Psi(r)$ is the flux limiter function, and $r$ is the discontinuity sensor, which, following the nomenclature in Figure A.1, is defined as follows [2]:

$$r_f = \frac{\Delta_U \theta}{\Delta_u \theta} = \frac{\theta_C - \theta_U}{\theta_D - \theta_C}, \tag{A.2}$$

where $\Delta_U \theta$ is the gradient of $\theta$ at the upwind face and $\Delta_u \theta$ the gradient at the face of interest. Both differences are taken as positive in the flow direction, defined by the sign of the velocity field $\boldsymbol{u}$.

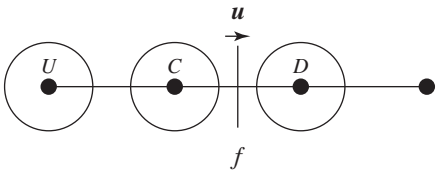


**Figure A.1:** Classical stencil for the computation of the gradient ratio at face $f$. $U$, $C$ and $D$ correspond to the upstream, centered and downstream nodes.

The discretization of Equation A.1 may benefit from the adoption of an algebraic approach. In this regard, it can be easily extended to the whole computational domain as:

$$\boldsymbol{\theta}_s = (\Pi_{c\to s} + \mathsf{F}(\boldsymbol{r}_s)\mathsf{Q}(\boldsymbol{u}_s)\Delta_{c\to s})\,\boldsymbol{\theta}_c, \tag{A.3}$$

where $\boldsymbol{\theta}_s \in \mathbb{R}^m$ and $\boldsymbol{\theta}_c \in \mathbb{R}^n$ are the vectors holding all the values of $\theta_f$ and $\theta_c$, $\Pi_{c\to s} \in \mathbb{R}^{m\times n}$ is the cell-to-face interpolation defined in Equation 2.15, $\mathsf{F}(\boldsymbol{r}_s) \in \mathbb{R}^{m\times m}$ is the diagonal matrix absorbing the artificial diffusion introduced in Equation A.1, $\mathsf{Q}(\boldsymbol{u}_s) \in \mathbb{R}^{m\times m}$ is the diagonal matrix taking the proper sign of the velocity at the faces, $[diag\,(\mathsf{Q}(\boldsymbol{u}_s))]_f = sign([\boldsymbol{u}_s]_f)$, and $\Delta_{c\to s} \in \mathbb{R}^{m\times n}$ takes the difference across the face following the nomenclature in Figure 2.2, left.

The construction of the gradient ratio will proceed first by the separate calculation of both the numerator ($\Delta_U\theta$) and the denominator ($\Delta_u\theta$) of Equation A.2, then computed as:

$$[\boldsymbol{r}_s]_f = \frac{[\boldsymbol{d_U}\boldsymbol{\theta}_s]_f}{[\boldsymbol{d_u}\boldsymbol{\theta}_s]_f} = \frac{[(\mathsf{Q}(\boldsymbol{u}_s)\mathsf{S}_{c\to s} + \mathsf{T}_{c\to s})\,\boldsymbol{\theta}_c]_f}{[\mathsf{Q}(\boldsymbol{u}_s)\Delta_{c\to s}\boldsymbol{\theta}_c]_f}, \tag{A.4}$$

where we introduce the new matrices $\mathsf{S}_{c\to s} = \frac{1}{2}\mathsf{N}_s\left(\mathsf{I}_d \otimes \mathsf{A}_s^D\right)\mathsf{N}_s^T\mathsf{E}_{c\to s}$ and $\mathsf{T}_{c\to s} = \frac{1}{2}\mathsf{N}_s\left(\mathsf{I}_d \otimes \mathsf{A}_s\right)\mathsf{N}_s^T\mathsf{E}_{c\to s}$, where $\mathsf{A}_s^D$ and $\mathsf{A}_s$ are the face adjacency and *directed* adjacency matrices following Figure A.2 and $\mathsf{E}_{c\to s}$ is the cell-to-face incidence (*cf.* see [1]). It is noted that $\mathsf{S}_{c\to s}$ and $\mathsf{T}_{c\to s}$ can be precomputed at the beginning of the simulation.

In this way, the final algorithm for the deployment of a flux limiter in the reconstruction of the variable at faces, $\theta_f$, within our algebra-based framework is described in Algorithm 4.

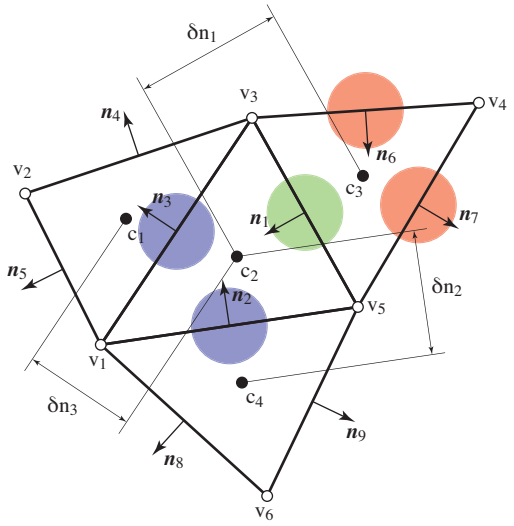


**Figure A.2:** Upstream (red) and downstream (blue) adjacent faces for face 1 with respect to a positive component of velocity. The selection of the right ones in $\mathsf{A}_s$ will ultimately depend on $\mathsf{Q}(\boldsymbol{u}_s)$.

**Algorithm 4** Algorithm for reconstruction of a scalar field at faces, $\boldsymbol{\theta}_s$, using the algebraic implementation of a flux limiter.

**Require:** $\boldsymbol{\theta}_c$, $\boldsymbol{u}_s$, $\Delta_{c\to s}$, $\mathsf{S}_{c\to s}$, $\mathsf{T}_{c\to s}$, $\Pi_{c\to s}$
**Ensure:** $\boldsymbol{\theta}_s$

1: $\boldsymbol{d_u\theta}_s \leftarrow \Delta_{c\to s}\boldsymbol{\theta}_c$ ▷ SpMV
2: $\boldsymbol{d_u\theta}_s \leftarrow \mathsf{Q}(\boldsymbol{u}_s)\boldsymbol{d_u\theta}_s$ ▷ signxty
3: $\boldsymbol{r}_s \leftarrow \mathsf{S}_{c\to s}\boldsymbol{\theta}_c$ ▷ SpMV
4: $\boldsymbol{r}_s \leftarrow \mathsf{Q}(\boldsymbol{u}_s)\boldsymbol{r}_s$ ▷ signxty
5: $\boldsymbol{r}_s \leftarrow \boldsymbol{r}_s + \mathsf{T}_{c\to s}\boldsymbol{\theta}_c$ ▷ SpMV
6: $\boldsymbol{r}_s \leftarrow \boldsymbol{r}_s/\boldsymbol{d_u\theta}_s$ ▷ axdy
7: $\boldsymbol{\theta}_s \leftarrow \mathsf{F}\boldsymbol{d_u\theta}_s$ ▷ superbeexty
8: $\boldsymbol{\theta}_s \leftarrow \boldsymbol{\theta}_s + \Pi_{c\to s}\boldsymbol{\theta}_c$ ▷ SpMV

## A.2 Computational comparison with stencil-based approaches

From a purely computational point of view, the algebraic approach seems inefficient. On the one hand, all three types of generalized algebraic kernels (*i.e.*, SpMV, kbin and kred) presented in Section 3.6.2 feature a very low AI, and this restricts their performance

to small fractions of the peak performance (this is further discussed in Chapters 3 and 4). On the other hand, this approach requires a higher number of simpler kernel calls, which results in producing more intermediate results. For instance, four independent kernel calls are required to compute $\boldsymbol{r}_s$ in lines 3 to 6 in Algorithm 4.

Therefore, the algebraic implementation for the appropriate reconstruction of the variables at faces, $\boldsymbol{\theta}_s$, is compared with classical, stencil-based approaches. This comparison is conducted from a theoretical point of view, assessing the minimum number of flop and memory traffic (in bytes) required in different scenarios. Note that the actual number of memory accesses during kernel execution depends not only on the algorithm but also on hardware and software features. Therefore, regarding the memory traffic, two different values are estimated considering the *full-hit* and *full-miss* caseloads. The former refers to the best scenario with an ideal temporal locality: multiple accesses to a particular data element are so close in time that its value is always reused from cache. Conversely, the latter considers the worst scenario with a null temporal locality so that every repeated access results in cache-miss and requires a memory load from memory. Thus, these two values result in the interval of effective AI of each kernel.

For the sake of clarity, in this comparison we only consider $k$-regular, periodic meshes composed of convex polygons which accomplish the following equality:

$$m \approx \frac{k}{2} n, \tag{A.5}$$

where $n$ and $m$ represent the number of cells and faces, respectively, and $k$ is the degree of the mesh elements (*i.e.*, the number of neighboring cells). The resulting requirements are listed in Table A.1, which is described throughout the section.

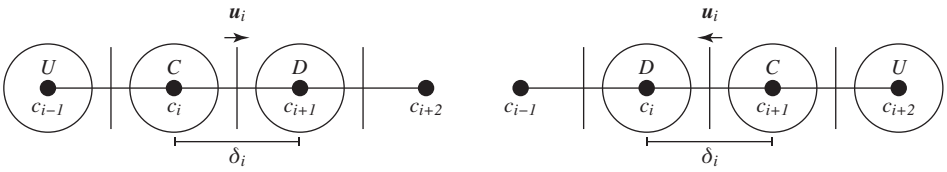


**Figure A.3:** Example of stencils in a one-dimensional Cartesian grid according to the classical flux limiter approach. The stencil topology varies according to the sign of the velocity field, $\boldsymbol{v}$.

**Table A.1:** Minimum number of flop and memory traffic (in bytes) required per mesh cell for computing the variable at the faces in different scenarios: stencil- and algebra-based implementations on uniform, non-uniform and unstructured one-dimensional ($k = 2$) and three-dimensional ($k = 6$) meshes.

| k | Approach | FLOP | Bytes | | AI | |
|---|---|---|---|---|---|---|
| | | | full-hit | full-miss | full-hit | full-miss |
| 2 | Uniform | 11 | 32 | 48 | 0.344 | 0.229 |
| 2 | Non-uniform | 13 | 40 | 56 | 0.325 | 0.232 |
| 2 | Unstructured | 20 | 84 | 132 | 0.238 | 0.152 |
| 2 | Algebraic | 28 | 288 | 352 | 0.097 | 0.080 |
| 6 | Uniform | 33 | 80 | 240 | 0.413 | 0.138 |
| 6 | Non-uniform | 39 | 104 | 360 | 0.375 | 0.108 |
| 6 | Unstructured | 168 | 228 | 1020 | 0.737 | 0.165 |
| 6 | Algebraic | 180 | 1408 | 1984 | 0.128 | 0.091 |

Let us start from the analysis of the simplest case: the stencil-based calculation of $\theta_f$ in a one-dimensional Cartesian grid, depicted in Figure A.3 and described in Algorithm 5. Every $i$th face is surrounded by two cells, $c_f$ and $c_{i+1}$. For each $i$th face in $m$, the sign of the velocity determines the upstream, centered and downstream values of $\theta$, as well as the centered and upstream distances. Following the algorithm, $5m$ flop are required for computing the gradient ratio in line 8, $1m$ for computing the limiter function in line 9 (this value may vary depending on the limiter function; in this example we have considered the *superbee* limiter [2]), and $7m$ for computing the value at the face in line 10. In the algorithm, three discrete fields are required for the computations: the initial scalar field, $\boldsymbol{\theta}_c \in \mathbb{R}^n$, the velocity field, $\boldsymbol{u}_s \in \mathbb{R}^m$ and the distances between cells, $\boldsymbol{d}_s \in \mathbb{R}^m$. Besides, the algorithm ensures the calculation of the discrete scalar field at faces, $\boldsymbol{\theta}_s \in \mathbb{R}^m$. Thus, considering double-precision values, and given that the number of faces is equal to the number of cells in the one-dimensional case ($k = 2 \rightarrow m = n$), the minimum flop and bytes required are $5m + 1m + 7m = 13n$ and $8(n + m + m + m) = 32n$, respectively. The total memory traffic in the full-miss caseload would rise to $48n$ because two different values of $\boldsymbol{d}_s$ and $\boldsymbol{\theta}_c$ are accessed for every face. Note that this values are slightly reduced in the particular case of uniform meshes: neither the distances array nor its quotient are required, thus omitting $2m$ flop in line 8 and the access to $\boldsymbol{d}_s$. On the other hand, the generalization of Algorithm 5 for three-dimensional Cartesian grids ($k = 6 \rightarrow m = 3n$) is straightforward and

rises the computational requirements to $39n$ flop and 80–$336n$ bytes for non-uniform meshes, or $33n$ flop and 56–$216n$ bytes in the uniform case.

**Algorithm 5** Stencil-based calculation of $\theta_f$ in a one-dimensional Cartesian grid.

**Require:** $\boldsymbol{\theta}_c$, $\boldsymbol{u}_s$, $\boldsymbol{d}_s$ ▷ 8C + 8F + 8F bytes
**Ensure:** $\boldsymbol{\theta}_s$ ▷ 8F bytes

1: **for** $i \leftarrow 1$ to $m$ **do**
2: **if** $\boldsymbol{u}_s[i] > 0$ **then**
3: $\theta_C = \boldsymbol{\theta}_c[i]$, $\theta_U = \boldsymbol{\theta}_c[i-1]$, $\theta_D = \boldsymbol{\theta}_c[i+1]$
4: $d_i = \boldsymbol{d}_s[i]$, $d_U = \boldsymbol{d}_s[i-1]$
5: **else**
6: $\theta_C = \boldsymbol{\theta}_c[i+1]$, $\theta_U = \boldsymbol{\theta}_c[i]$, $\theta_D = \boldsymbol{\theta}_c[i+2]$
7: $d_i = \boldsymbol{d}_s[i]$, $d_U = \boldsymbol{d}_s[i+1]$
8: $r_i \leftarrow (d_i/d_U)(\theta_C - \theta_U)/(\theta_D - \theta_C)$ ▷ 5F flops
9: $\Psi_i \leftarrow limiter(r_f)$ ▷ 1F flops
10: $\boldsymbol{\theta}_s[i] \leftarrow (\theta_C + \theta_D)/2 + (\Psi_i - 1)(\theta_D - \theta_C)/2$ ▷ 7F flops

A generalization of the stencil calculation for unstructured meshes is outlined in Algorithm 6. In contrast with the structured algorithm, the indices of neighboring nodes are not predictable, so the incidence graphs are required. For each $i$th face in $m$, the sign of the velocity determines the indices of the centred and downstream cells, $c_C$ and $c_D$, according to the cell-to-face incidence graph. Then, for each $j$th face incident to $c_C$ (except $f_i$), its contribution to the upstream gradient, projected over the normal of $f_i$, is accumulated, accounting for $5(k-1)m$ and $4(k-1)m$ flop in lines 18 and 19, respectively. The calculation of $\theta_f$ in lines 20 to 22 follows similarly as in Algorithm 5, and adds $13m$ operations. In this case, six discrete fields, one integer list and two incidence graphs are required for the computations. The specific requirements for two k-regular unstructured meshes, $k = 2$ (one-dimensional mesh) and $k = 6$ (three-dimensional hexaedral mesh), are listed in Table A.1.

Finally, in the algebraic implementation outlined in Algorithm 4, it can be observed how it is completely independent of the mesh type and the numerical method: these characteristics only affect the matrices. The number of calls to SpMV and kbin kernels is readily deduced from the algorithm: 4 times each. Four matrices are required for the computations: the differences at faces, $\Delta_{c\to s}$, with $2m$ non-zero elements, the oriented and unoriented differences, $\mathsf{S}_{c\to s}$ and $\mathsf{T}_{c\to s}$, with $2km$ each, and the cell-to-face interpolation, $\Pi_{c\to s}$, with $2m$. In this example, we consider the use of the

**Algorithm 6** Stencil-based calculation of $\theta_f$ in a generic unstructured grid based on incidence graphs.

**Require:** $\boldsymbol{\theta}_c$, $\boldsymbol{u}_s$, $\boldsymbol{d}_s$, $\boldsymbol{nx}_s$, $\boldsymbol{ny}_s$, $\boldsymbol{nz}_s$, ▷ 8C + 40F bytes
$\boldsymbol{k}_c$, $\Delta_{c\to s}$, $\mathsf{E}_{c\to s}$ ▷ 4C + 8F + 8F bytes
**Ensure:** $\boldsymbol{\theta}_s$ ▷ 8F bytes

1: **for** $i \leftarrow 1$ to $m$ **do**
2: **if** $\boldsymbol{u}_s[i] > 0$ **then**
3: $c_C = \Delta_{c\to s}[i][0]$
4: $c_D = \Delta_{c\to s}[i][1]$
5: **else**
6: $c_C = \Delta_{c\to s}[i][1]$
7: $c_D = \Delta_{c\to s}[i][0]$
8: $\Delta_U = 0$
9: **for** $k \leftarrow 1$ to $\boldsymbol{k}_c[c_C]$ **do**
10: $j = \mathsf{E}_{c\to s}[c_C][k]$
11: **if** $j \neq i$ **then**
12: **if** $\Delta_{c\to s}[j][0] \neq c_C$ **then**
13: $c_U = \Delta_{c\to s}[j][0]$
14: **else**
15: $c_U = \Delta_{c\to s}[j][1]$
16: $d_j = \boldsymbol{d}_s[j]$
17: $\theta_U = \boldsymbol{\theta}_c[c_U]$
18: $n_k \leftarrow \boldsymbol{nx}_s[i]\boldsymbol{nx}_s[j] + \boldsymbol{ny}_s[i]\boldsymbol{ny}_s[j] + \boldsymbol{nz}_s[i]\boldsymbol{nz}_s[j]$ ▷ 5(k-1)F flops
19: $\Delta_U \leftarrow \Delta_U + n_k(\theta_C - \theta_U)/d_j$ ▷ 4(k-1)F flops
20: $r_i \leftarrow \Delta_U/((\theta_D - \theta_C)/d_i)$ ▷ 3F flops
21: $\Psi_i \leftarrow superbee(r_i)$ ▷ 1F flops
22: $\boldsymbol{\theta}_s[i] \leftarrow (\theta_C + \theta_D)/2 + (\Psi_i - 1)(\theta_D - \theta_C)/2$ ▷ 7F flops

ELLPACK format [3] in which each non-zero element accounts for 12 bytes (*i.e.*, 8 bytes for the coefficient and 4 bytes for the column index). The specific requirements for two k-regular meshes are listed in Table A.1.

## A.3 Three-dimensional deformation problem

The algebraic implementation of a flux limiter in HPC$^2$ is applied to a canonical benchmark. In particular, the deformation (advection) of a sharp profile, which has been tested on three-dimensional hexahedral meshes of $72^3$, $144^3$, $288^3$, $432^3$ and

$576^3$ cells following Algorithm 7, where $\mathsf{M} \in \mathbb{R}^{n\times m}$ is the divergence operator and $\mathsf{U}(\boldsymbol{u}_s) \in \mathbb{R}^{m\times m}$ is a diagonal matrix containing the velocities at faces (*cf.* Section 2.1.1). Recall we evaluate the products by diagonal matrices by means of kbin calls (*cf.* Section 3.6.2).

**Algorithm 7** Algorithm for the advection of a scalar field with a 1st order Euler method, using the algebraic implementation of a flux limiter. The nomenclature and algorithm formulation follows that of Chapter 2.

**Require:** $\boldsymbol{\theta}_c^n$, $\boldsymbol{u}_s$, $dt$, $\Delta_{c\to s}$, $\mathsf{S}_{c\to s}$, $\mathsf{T}_{c\to s}$, $\Pi_{c\to s}$, $\mathsf{M}$
**Ensure:** $\boldsymbol{\theta}_c^{n+1}$
1: $\boldsymbol{d_u\theta}_s \leftarrow \mathsf{Q}(\boldsymbol{u}_s)\Delta_{c\to s}\boldsymbol{\theta}_c^n$
2: $\boldsymbol{r}_s \leftarrow (\mathsf{Q}(\boldsymbol{u}_s)\mathsf{S}_{c\to s} + \mathsf{T}_{c\to s})\boldsymbol{\theta}_c^n / \boldsymbol{d_u\theta}_s$
3: $\boldsymbol{\theta}_s \leftarrow \Pi_{c\to s}\boldsymbol{\theta}_c^n + \mathsf{F}(\boldsymbol{r}_s)\boldsymbol{d_u\theta}_s$
4: $\boldsymbol{\theta}_c^{n+1} \leftarrow \boldsymbol{\theta}_c^n + dt\mathsf{MU}(\boldsymbol{u}_s)\boldsymbol{\theta}_s$

The sharp profile is initialized in a physical domain of $[0,1]\times[0,1]\times[0,1]$ as a sphere of radius $r = 0.15$, located at $(0.35, 0.35, 0.35)$, and subject to a divergence-free velocity field:

$$u = 2sin^2(\pi x)sin(2\pi y)sin(2\pi z)cos(\pi t/T), \tag{A.6}$$

$$v = -sin(2\pi x)sin^2(\pi y)sin(2\pi z)cos(\pi t/T), \tag{A.7}$$

$$w = -sin(2\pi x)sin(2\pi y)sin^2(\pi z)cos(\pi t/T), \tag{A.8}$$

during 3.0 time-units, $T$ [4].

The results of the profile on meshes of $72^3$, $144^3$, $288^3$, $432^3$ and $576^3$ cells are shown in Figure A.4 for the slices in $x = 0.35$, $y = 0.35$ and $z = 0.35$ planes. Also, the three-dimensional temporal evolution of the sphere on a mesh of $576^3$ cells is shown in Figure A.5. As in [4], the resulting shapes after the deformation are satisfactory, and mass is precisely conserved.

This benchmark has been deployed in the HPC². To demostrate its portability, the simulations have been run on different supercomputers. Before going into details, we recall from Section 3.2 the theoretically achievable performance, $\pi_k$, which reads

$$\pi_k = min(\pi_u, AI_k\beta_u),$$

where $\pi_u$ is the peak performance of the computing device in double-precision, $\beta_u$ is

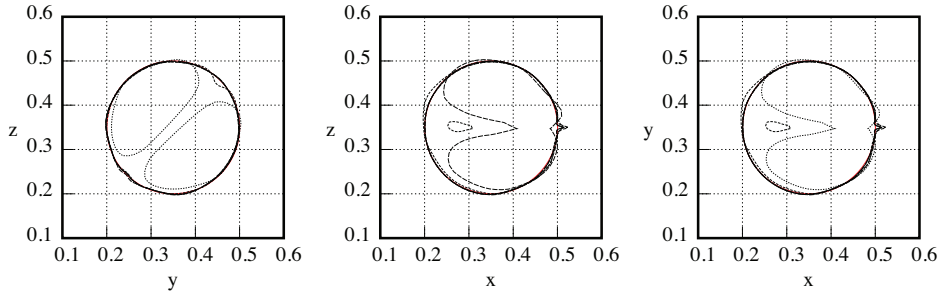


**Figure A.4:** Contours for $\theta = 0.5$ in $x = 0.35$, $y = 0.35$ and $z = 0.35$ planes after 3.0 time-units for meshes of $72^3$, $144^3$, $288^3$, $432^3$ and $576^3$ cells.

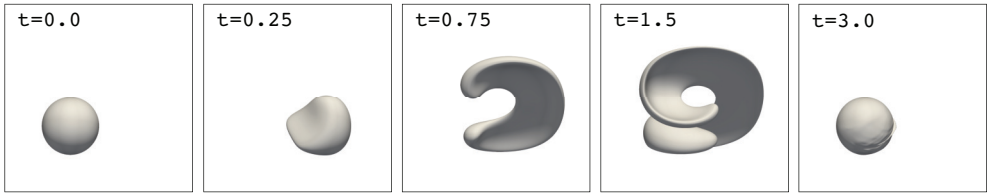


**Figure A.5:** Time evolution of the $\theta = 0.5$ contour for $t = 0$, 0.25, 0.75, 1.5, 2.25, 2.75 and 3.0 time-units for mesh of $576^3$ cells.

the peak memory bandwidth and $AI_k$ is the maximum AI of the kernel, taking the full-hit scenario as described in Section A.2. Then, we define the performance and memory efficiencies as the ratio of measured performance to $\pi_k$ and measured memory traffic to full-hit, respectively.

The simulations on meshes of $72^3$–$432^3$ cells have been executed on up to 64 nodes (3,072 cores) of the CPU-based MareNostrum 4 supercomputer at the Barcelona Supercomputing Center. Its nodes are equipped 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), interconnected through the Intel Omni-Path network (12.5 GB/s). The application achieved a sustained performance of up to 1.6 Tflop/s, corresponding to nearly 0.8 of performance efficiency.

The simulation on a mesh of $576^3$ cells has been executed on 27 nodes of the Lomonosov-2 hybrid 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), interconnected via InfiniBand FDR network (7 GB/s). The application achieved a sustained performance of 0.9 Tflop/s, corresponding to nearly 0.75 of performance efficiency and 98.5% of the heterogeneous efficiency (*i.e.*, the ratio of the heterogeneous performance to the sum of CPU-only and GPU-only performances).

Finally, we aim at completing the performance analysis, and the theoretical comparison in Section A.2, by comparing the actual performance of the algebraic and stencil approaches. Following Algorithm 6, a stencil kernel has also been implemented in HPC². The tests have been carried out on the CPU-based JFF fourth-generation cluster at the Heat and Mass Transfer Technological Center. Its nodes are equipped with two Intel Xeon Gold 6230 CPUs (20 cores, 2.1 GHz, 6 DDR4-2933 memory channels, 140 GB/s memory bandwidth, 27.5 MB L3 cache). Considering that the DM parallelization is equivalent in both approaches since the data exchanges are the same, only single-node comparisons have been conducted.

The results are shown in a roofline plot [5] in Figure A.6 for both single- and dual-socket executions using a NUMA-aware, SM parallelization on meshes of $144^3$ (A) and $288^3$ (B) cells. We have discarded the smallest mesh of $72^3$ to ensure a memory-bounded behavior and the biggest meshes of $432^3$ and $576^3$ because they do not fit in a single node. In the plot, two vertical lines represent the minimum and maximum values of the AI for each kernel as estimated in Section A.2 and outlined in Table A.1. Then, the roofline curve is calculated as follows:

$$\Pi_u(\pi_u, \beta_u) = min(\pi_u, AI\beta_u).$$

In this particular test, $\beta_u$ and $\pi_u$ are 140 GB/s and 1344 Gflop/s per socket, respectively. The real number of memory accesses to main memory have been measured using a profiling tool to calculate the effective AI of each execution.

In single-socket execution, the stencil kernel performs nearly twice faster than the algebraic one (stencil: 33.02 (A) and 24.52 (B) Gflop/s, algebraic: 13.89 (A) and 13.54 (B) Gflop/s), even though its lower performance efficiency (stencil: 0.32 and 0.24, algebraic: 0.78 and 0.76). However, this gap is reduced to approximately $\times 1.35$ in the dual-socket case (stencil: 37.01 and 33.82 Gflop/s, algebraic: 27.16 and 25.17 Gflop/s). The algebraic kernels feature a regular unit-strided memory access everywhere except the input vector in SpMV. In contrast, the stencil kernel leads to irregular accesses

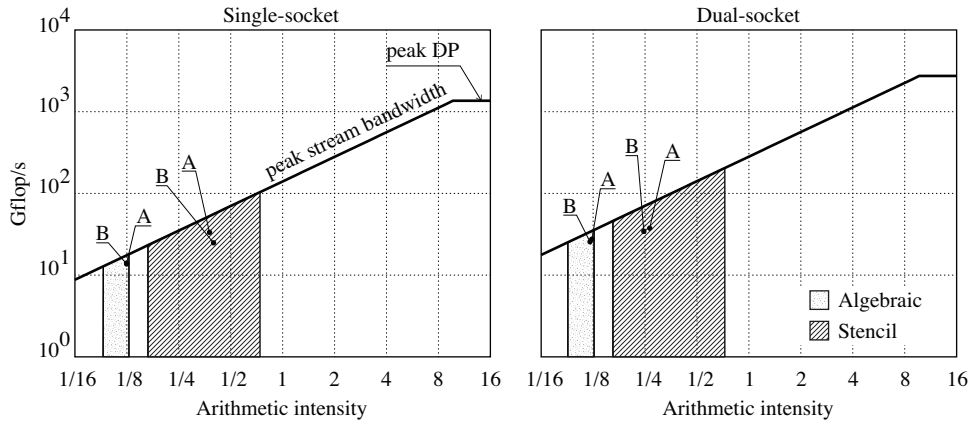


**Figure A.6:** Roofline models representing the ranges of achievable performance for the stencil- and algebra-based implementations of the flux limiter on three-dimensional unstructured meshes. Results on Intel Xeon Gold 6230 are shown for meshes of $144^3$ (A) and $288^3$ (B).

to $\mathsf{E}_{s\to c}$, $\Delta_{c\to s}$, $\boldsymbol{d}_s$, $\boldsymbol{\theta}_c$ and $\boldsymbol{n}$, resulting in higher cache miss rates and reducing the memory efficiency, especially in dual-socket configurations (stencil: 0.36 and 0.33, algebraic: 0.96 and 0.94). Therefore, the actual performance gap is far from the $\times 5.75$ of the (worst) theoretical scenario.

## References

[1] N. Valle, X. Álvarez-Farré, A. Gorobets, J. Castro, A. Oliva, and F. X. Trias, "On the implementation of flux limiters in algebraic frameworks," *Computer Physics Communications*, vol. 271, p. 108230, 2022.

[2] P. K. Sweby, "High resolution schemes using flux limiters for hyperbolic conservation laws," *SIAM J. Numer. Anal.*, vol. 21, pp. 995–1011, oct 1984.

[3] D. R. Kincaid, T. C. Oppe, and D. M. Young, "ITPACKV 2D user's guide," tech. rep., Center for Numerical Analysis, University of Texas, 1989.

[4] K. Yang and T. Aoki, "Weakly compressible Navier–Stokes solver based on evolving pressure projection method for two-phase flow simulations," *Journal of Computational Physics*, vol. 431, p. 110113, 2021.

[5] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.