Tenth International Conference on Computational Fluid Dynamics (ICCFD10), Barcelona, Spain, July 9-13, 2018

ICCFD10-2018-0353

Implementation of a flux limiter into a fully-portable, algebra-based framework for heterogeneous computing

X. Álvarez*, N. Valle*, A. Gorobets**, F. X. Trias*
Corresponding author: xavier@cttc.upc.edu

* Heat and Mass Transfer Technological Center, Technical University of Catalonia, C/ Colom 11, Terrassa (Barcelona), 08222, Spain
** Keldysh Institute of Applied Mathematics, Miusskaya Sq. 4, Moscow, 125047, Russia

Abstract: During the last years, there has been a significant increment in the variety of hardware to overcome the power constraint in the context of the exascale challenge. This progress is leading to an increasing hybridisation of high-performance computing (HPC) systems, making the design of computing applications a rather complex problem, and is affecting most of the fields that rely on large-scale simulations. Many scientific computing applications have been partially ported, even rewritten entirely, to take advantage of the coprocessor devices (*e.g.* GPUs or MICs). This porting process usually becomes very costly for the traditional stencil-based implementations as they involve a large number of functions (*i.e.* every numerical method requires specific functions and data-sets). In contrast, algebra-based implementations have shown to provide with a high level of abstraction and portability because they simplify the calculations into universal algebraic kernels only. In this work, we present an algebraic formulation of a high-resolution scheme, a flux limiter, and detail its implementation into the fully-portable, algebra-based HPC^2 framework. As a result, the solution of the advection of a scalar field with sharp discontinuities relies on a reduced set of algebraic kernels. Therefore, the algebra-based approach combined with a multilevel MPI + OpenMP + OpenCL parallelisation naturally provides modularity and portability.

Keywords: High-resolution schemes, Flux limiter, Parallel CFD, Portability, Heterogeneous computing.

1 Introduction

Massively-parallel devices of various architectures are being incorporated to the modern supercomputers to overcome the power constraint in the context of the exascale challenge [1], increasing the variety of hardware significantly. This progress is leading to an increasing hybridisation of high-performance computing (HPC) systems and making the design of computing applications a rather complex problem. To take advantage of the most efficient HPC systems, the computing operations that form the algorithms, the so-called kernels, must be compatible with distributed- and shared-memory SIMD and MIMD parallelism and, more importantly, with stream processing (SP), which is a more restrictive parallel paradigm. Many scientific computing applications have been partially ported, even rewritten entirely, to take advantage of the coprocessor devices (*i.e.* GPUs or MICs). For instance, in [2] the reader can find the solution of incompressible two-phase flows on multi-GPU platforms. Furthermore, examples of heterogeneous implementations of CFD algorithms for hybrid CPU+GPU supercomputations can be found in [3, 4], and an example of a petascale CFD simulation on 18.000 K20X GPUs in [5].

In this context of accelerated innovation, making an effort to design modular applications composed of a reduced number of independent and well-defined code blocks is worth it. On the one hand, this helps

to reduce the generation of errors and facilitates debugging. On the other hand, modular applications are user-friendly and more comfortable for porting to new architectures (the fewer the kernels of an application and its dependencies, the easier it is to provide portability). Furthermore, if the majority of computing kernels represent universal algebraic operations, then both the standard optimised libraries (*e.g.* ATLAS, clBLAST) and the specific in-house implementations can be used and easily switched.

In previous work, Oyarzun et al. [6] proposed a portable implementation model for direct numerical simulations (DNS) and large eddy simulations (LES) of incompressible turbulent flows on unstructured meshes. Roughly, the method consists of replacing traditional stencil data structures and sweeps by algebraic data structures and kernels. As a result, the algorithm of the time-integration phase relies on a reduced set of only three basic algebraic operations: the sparse matrix-vector product, the linear combination of vectors and the dot product. Consequently, this approach combined with a multilevel MPI + OpenMP + OpenCL parallelisation naturally provides modularity and portability.

Inspired by the compelling results of the algebraic implementation, we increased the level of abstraction and presented in [7] the HPC^2 (Heterogeneous Portable Code for HPC), a fully-portable, algebra-based framework with many potential applications in the fields of computational physics and mathematics. The strategies for the heterogeneous execution of the HPC^2 kernels were improved and detailed in [8] reporting satisfying strong scalability results on up to 32 nodes of a hybrid supercomputer equipped with a 14-core Intel E5-2697v3 CPU and an NVIDIA Tesla K40M GPU.

In this work, we aim at extending the scientific applications of the HPC^2 framework. Specifically, our challenge is to implement a high-resolution scheme (specifically the SUPERBEE flux limiter in [9]) into our heterogeneous computing framework for solving hyperbolic problems in the presence of sharp discontinuities or shocks. For this purpose, the flux limiter must be somehow rewritten in an algebraic form so that stencil operations are avoided during the computation. Therefore, the algebraic formulation of a SUPERBEE flux limiter is described in Section 2. Such a formulation of the flux limiter becomes more challenging in comparison with that of a DNS because of its non-linearity. However, instead of being an inconvenience, this encourages us to demonstrate in Section 3 the high potential of our algebra-based implementation strategy again, showing that only the addition of a few simple algebraic kernels is required. Finally, we challenge in Section 4 the algebraic implementation of the flux limiter for the advection of scalar field with a sharp discontinuity using 2D structured grids in both CPU and GPU platforms.

2 Algebra-based formulation of a flux limiter

Flux limiters are non-linear functions commonly used for solving hyperbolic problems in the presence of sharp discontinuities or shocks. Specifically, flux limiters are used to construct high-resolution schemes with the aim of obtaining second-order approximations and avoiding oscillations near the interface [10]. For instance, a flux limiter scheme is introduced in [11] for the evaluation of the convective terms in the simulation of two-phase flows using the conservative level-set method on unstructured grids.

Let us consider the typical form of a flux limiter for finite volume methods [11],

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

where θ_f is the value of the scalar θ at the face of interest, θ_U and θ_D are upwind and downwind values of θ according to the velocity field u, and $\Psi(r)$ stands for the flux limiter function. The argument r, namely the discontinuity sensor, is chosen as the gradient ratio and is defined as

$$r_f = \frac{\Delta_U \theta}{\Delta_u \theta},$$

where $\Delta_U\theta$ is the gradient of θ at the upwind face and $\Delta_u\theta$ is the gradient at the face of interest. Finally, to facilitate the casting of the flux limiter into an algebraic form, we rewrite the Equation (1) in the less common form:

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

The operator-based, finite volume discretisation of the Equation (2) is written as follows (the details of the mathematical background and the construction of the matrices below can be found in Trias et al. [12], Valle et al. [13]):

$$\boldsymbol{\theta}_s = (\mathsf{\Pi}_{c\to s} + \Omega(\boldsymbol{r}_s)\cdot \mathsf{Q}(\boldsymbol{u}_s)\cdot \mathsf{\Delta}_{c\to s})\,\boldsymbol{\theta}_c, \tag{3}$$

where $\boldsymbol{\theta}_s \in \mathbb{R}^m$ and $\boldsymbol{\theta}_c \in \mathbb{R}^n$ are the staggered and centred scalar fields respectively, $\boldsymbol{r}_s \in \mathbb{R}^m$ is the gradient ratio at the faces, and $\boldsymbol{u}_s = ((u_s)_1, (u_s)_2, ..., (u_s)_m)^T \in \mathbb{R}^m$ is the auxiliary discrete staggered velocity which is related to the centered velocity field via a linear interpolation $\mathsf{\Gamma}_{c\to s} \in \mathbb{R}^{m\times dn}$ such that $\boldsymbol{u}_s \equiv \mathsf{\Gamma}_{c\to s}\boldsymbol{u}_c$. The size of these vectors, n and m, are the number of control volumes and faces on the computational domain respectively, and d is the number of dimensions of the simulation. The subindices c and s refer to whether the variables are cell-centred or staggered at the faces. The matrices $\mathsf{\Pi}_{c\to s}$ and $\mathsf{\Delta}_{c\to s}$ are constant and represent the scalar cell-to-face interpolator and the scalar cell-to-face difference operator respectively. The matrix $\mathsf{Q}(\boldsymbol{u}_s)$ is a variable and diagonal matrix which holds the sign of the velocity relative to the normal of the face $\boldsymbol{u}_s$. The elements in the diagonal of $\mathsf{Q}(\boldsymbol{u}_s)$ are recomputed in each time-step as

$$diag\,(\mathsf{Q}(\boldsymbol{u}_s)) = sign(\boldsymbol{u}_s). \tag{4}$$

The gradient ratio $\boldsymbol{r}_s$, which is the argument for computing $\Omega(\boldsymbol{r}_s)$, is measured as

$$\boldsymbol{r}_s(\boldsymbol{\theta}_c) = \frac{(\mathsf{Q}(\boldsymbol{u}_s)\mathsf{UUD}_{c\to s} + \mathsf{OUD}_{c\to s})\,\boldsymbol{\theta}_c}{(\mathsf{Q}(\boldsymbol{u}_s)\mathsf{\Delta}_{c\to s})\,\boldsymbol{\theta}_c}. \tag{5}$$

The matrices $\mathsf{OUD}_{c\to s}$ and $\mathsf{UUD}_{c\to s}$ are the oriented and unoriented cell-to-face difference operators respectively [13]. The matrix $\Omega(\boldsymbol{r}_s)$ is a variable and diagonal which represents the term $(\Psi(r)-1)/2$ of Equation (2). The elements in its diagonal depend on the chosen flux limiter function. Then, considering the SUPERBEE flux limiter scheme [9], the elements in the diagonal of $\Omega(\boldsymbol{r}_s)$ become

$$diag\,(\Omega(\boldsymbol{r}_s)) = \frac{max(0, max(min(1, 2\boldsymbol{r}_s), min(\boldsymbol{r}_s, 2))) - 1}{2}. \tag{6}$$

Finally, note that the implementation of different flux limiter functions would only affect the values in the diagonal of $\Omega(\boldsymbol{r}_s)$.

3 Implementation of the flux limiter into the HPC^2

In our previous works [6, 8], we proposed a portable implementation model for direct numerical simulations (DNS) and large eddy simulations (LES) of incompressible turbulent flows. As a result, the algorithm for the time-integration relies on a reduced set of only three basic algebraic operations: the sparse matrix-vector product (`SpMV`), the linear combination of vectors (`axpy`) and the dot product (`ddot`). However, it can be deduced from the Equations (4), (5) and (6) that some new kernels are required to perform element-wise operations over the vectors (*e.g.* an element-wise division is required for computing the gradient ratio as in Equation (5)). Nevertheless, instead of being an inconvenience, this encourages us to demonstrate the high potential of our algebra-based implementation strategy again, showing that only the addition of six simple algebraic kernels of element-wise operations over the vectors is required to embed the algebraic flux limiter into our fully-portable, algebra-based framework. These new kernels are described below.

`y = axdy(y, x, a)`	$\longrightarrow y_i = ay_i/x_i,$
`y = shft(y, a)`	$\longrightarrow y_i = y_i - a,$
`y = scal(y, a)`	$\longrightarrow y_i = ay_i,$
`y = vmax, vmin(y, x)`	$\longrightarrow y_i = max, min(y_i, x_i),$
`y = smax, smin(y, a)`	$\longrightarrow y_i = max, min(y_i, a),$
`y = sign(x)`	$\longrightarrow y_i = \{-1 \text{ if } x_i < 0,\ 1 \text{ otherwise}\}.$

The six kernels above do not show appreciable differences regarding their computational behaviour com-

pared with the `axpy`. On the one hand, they are simple element-wise operations over the vectors; hence there is no need for communications in distributed-memory parallelisation. Besides, they provide a uniform aligned memory access with coalescing of memory transactions which suit the stream processing paradigm perfectly. On the other hand, the arithmetic intensity of this new kernels (*i.e.* the number of FLOP per byte) is very low, likewise to that of the `axpy`; thus their performance is also memory-bounded (see Table 1). Therefore, having already efficient OpenMP, OpenCL and CUDA implementations of `axpy`, implementing the six new kernels above is straightforward.

Table 1: Analysis of the arithmetic intensity (FLOP per byte). The value n refers to the number of elements in the vectors, and nnz the number of non-zero elements in the matrices. The amount of bytes is scaled by 8 assuming the usage of double-precision data types.

	`SpMV`	`axpy`	`axdy`	`shft`	`scal`	`vmax`, `vmin`	`smax`, `smin`	`sign`
Number of FLOP	$2nnz - n$	$2n$	$2n$	n	n	n	n	n
Amount of bytes moved	$8(nnz+2n)$	$8(3n)$	$8(3n)$	$8(2n)$	$8(2n)$	$8(3n)$	$8(3n)$	$8(2n)$
Arithmetic intensity	*variable*	0.083	0.083	0.0625	0.0625	0.042	0.042	0.0625

Finally, the implementation of the Equations (4), (5) and (6) into the HPC^2 framework becomes a simple combination of algebraic kernels which reads

$$\begin{aligned}
\texttt{Q.diag} &= \texttt{sign(u)},\\
\texttt{r} &= \texttt{axdy(axpy(spmv(Q, spmv(UUD, t)), spmv(OUD, t), 1), spmv(Q, spmv(D, t)), 1)},\\
\texttt{F.diag} &= \texttt{scal(shft(max(0, max(min(1, scal(r, 2)), min(r, 2))), -1), 0.5)},
\end{aligned}$$

where the matrices `Q`, `UUD`, `OUD`, `D` and `F` are $\mathsf{Q}(\boldsymbol{u}_s)$, $\mathsf{UUD}_{c\to s}$, $\mathsf{OUD}_{c\to s}$, $\Delta_{c\to s}$ respectively, and the vectors `u`, `r` and `t` are $\boldsymbol{u}_s$, $\boldsymbol{r}_s$ and $\boldsymbol{\theta}_c$ respectively. In comparison with the stencil-based, the algebra-based implementation is very similar regarding performance. The distributed-memory parallelisation remains the same since the size of the *halo* (*i.e.* the adjacent elements owned by different MPI processes) depends directly on the partitioning of the computational domain. On the other hand, the stencil-based approach could appear to be slightly more efficient regarding the use of data in some numerical schemes since they can discriminate some values with conditional statements, for instance when locating upwind values. However, these conditional statements may harm the memory access and the coalescing of memory transactions, which is very important in some parallel paradigms such as stream processing. Nevertheless, the data oversize affect only a few operators by a small percentage in the algebra-based implementation.

4 Numerical tests

In this section, the algebraic implementation of the flux limiter is tested for a canonical case. We consider the simulation of the advection of a scalar field with a sharp discontinuity using a high-resolution scheme. The complete algorithm for the time-integration of the advection equation using the algebraic formulation of a flux limiter is described in Algorithm 1.

Algorithm 1 Time-integration step for the advection of a scalar field with the SUPERBEE flux limiter

1. Compute the matrix $\mathsf{Q}(\boldsymbol{u}_s)$ as $diag(\mathsf{Q}(\boldsymbol{u}_s)) = sign(\boldsymbol{u}_s)$.
2. Compute the vector $\boldsymbol{r}_s(\boldsymbol{\theta}_c) = ((\mathsf{Q}(\boldsymbol{u}_s)\mathsf{UUD}_{c\to s} + \mathsf{OUD}_{c\to s})\,\boldsymbol{\theta}_c) \,/\, ((\mathsf{Q}(\boldsymbol{u}_s)\Delta_{c\to s})\,\boldsymbol{\theta}_c)$.
3. Compute the matrix $\Omega(\boldsymbol{r}_s)$ as $diag(\Omega(\boldsymbol{r}_s)) = (max(0, max(min(1, 2\boldsymbol{r}_s), min(\boldsymbol{r}_s, 2))) - 1)\,/2$.
4. Calculate $\boldsymbol{\theta}_c^{n+1}$ with 1st order Euler method: $\boldsymbol{\theta}_c^{n+1} = \boldsymbol{\theta}_c^{n} - dt \cdot \mathsf{DIV} \cdot \mathsf{U}_s\,(\Pi_{c\to s} + \Omega(\boldsymbol{r}_s)\mathsf{Q}(\boldsymbol{u}_s)\Delta_{c\to s})\,\boldsymbol{\theta}_c$

The marker function is initialised in a 2D domain with the shape of a rhodonea [14] for three different structured grids of 32x32, 128x128 and 512x512 cells. Besides, two different fixed velocity fields are evaluated. Namely, a rotating field $\boldsymbol{u} = (y, -x)$, together with Dirichlet boundary conditions in all the boundaries, and a flat field $\boldsymbol{u} = (0, 1)$, combined with periodic boundary conditions on top and bottom. The Figure 1 shows various plots of the marker function for the 32x32 grid (top) and the 128x128 (bottom). On the left hand side, the initial state of $\boldsymbol{\theta}_c$ is shown. In the center, the final state of $\boldsymbol{\theta}_c$ after one cicle under the flat velocity field with periodic boundary conditions. On the right hand side, the final state of $\boldsymbol{\theta}_c$ after one complete revolution under the rotating velocity field.

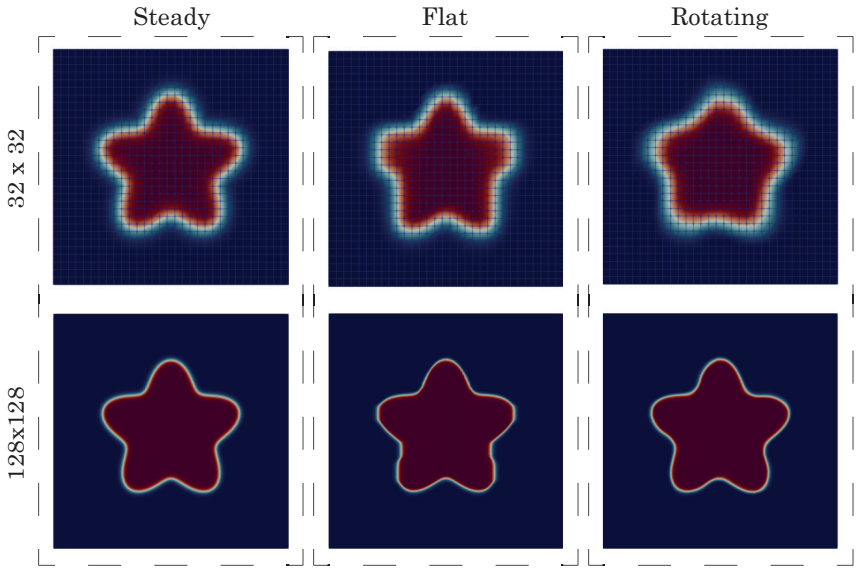

Figure 1: Plots of the marker function $\boldsymbol{\theta}_c$ for different grids and cases.

The flux limiter implementation has been tested on a single node equipped with an Intel i5-2300 and an Nvidia GTX 590. To demonstrate the effectiveness of the algebra-based approach, we show in Table 2 the number of times that each algebraic kernel is called every time-step in the numerical Algorithm 1. Then, the comparison of the relative time spent in each operation in both CPU and GPU is shown in Figure 2 (for simplicity, the vector kernels have been grouped). The results in Figure 2 confirm again that the overall performance is only depending on the performance of the kernels. The 96% and 88% of the computational time are employed for running kernels in the CPU and GPU respectively. Hence, the cost estimation of large-scale simulations of multiphase flows on hybrid supercomputers can be extrapolated from the performance study in our previous work [8] to avoid a nonsense waste of computational resources in repetitive performance study.

Table 2: Number of times that each kernel is executed per time-step

Step of Algorithm 1	`SpMV`	`axpy`	`axdy`	`shft`	`scal`	`vmax, vmin`	`smax, smin`	`sign`
1 – Compute matrix $\mathsf{Q}(\boldsymbol{u}_s)$	0	0	0	0	0	0	0	1
2 – Compute gradient ratio	5	1	1	0	0	0	0	0
3 – Compute matrix $\Omega(\boldsymbol{r}_s)$	0	0	0	1	2	1	3	0
4 – 1st order Euler	6	2	0	0	0	0	0	0
Total number of executions	11	3	1	1	2	2	2	1

The performance wasted in the *others* group is due to operations that are not directly involved with the algorithm such as the printing of simulation outputs. Given the fact that in the GPU-only execution the host (CPU) must download the device's (GPU) data, and the PCIe bus is slower than the main memory

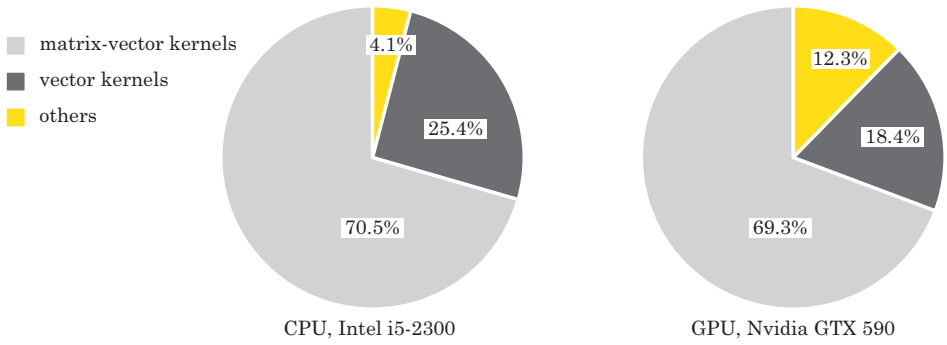

Figure 2: Comparison of the computational cost of the operations in one time step.

bandwidth, it is consistent that the simulation spends more time in the *others* group when it is executed on GPUs.

5 Conclusion

An algebraic formulation of a high resolution scheme has been presented. A flux limiter has been implemented into the HPC^2 framework using only simple vector operations. We have shown that the addition of six simple algebraic kernels is sufficient to implement high-resolution, non-linear schemes into our framework. The advection of a scalar field with a sharp discontinuity has been simulated using different 2D structured grids, velocity fields and boundary conditions, and the simulations have been run on both CPU and GPU. Hence, our fully-portable, algebra-based framework for heterogeneous computing has shown a great potential for the simulation of multiphase flows on hybrid supercomputers.

Acknowledgments

The work has been financially supported by the *Ministerio de Economía y Competitividad*, Spain (ENE2017-88697-R and ENE2015-70672-P). X. Á. is supported by a *FI AGAUR* predoctoral contract (2018FI_B1_00081). N. V. is supported by a *FI AGAUR* predoctoral contract (2018FI_B1_000109). F. X. T. is supported by a *Ramón y Cajal* postdoctoral contract (RYC-2012-11996).

References

[1] Jack Dongarra et al. The International Exascale Software Project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, feb 2011.

[2] Peter Zaspel and Michael Griebel. Solving incompressible two-phase flows on multi-GPU clusters. *Comput. Fluids*, 80(1):356–364, 2013.

[3] Andrey Gorobets, F. Xavier Trias, and Assensi Oliva. A parallel MPI+OpenMP+OpenCL algorithm for hybrid supercomputations of incompressible flows. *Comput. Fluids*, 88:764–772, dec 2013.

[4] Chuanfu Xu, Xiaogang Deng, Lilun Zhang, Jianbin Fang, Guangxue Wang, Yi Jiang, Wei Cao, Yonggang Che, Yongxian Wang, Zhenghua Wang, Wei Liu, and Xinghua Cheng. Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer. *J. Comput. Phys.*, 278(1):275–297, dec 2014.

[5] Peter E. Vincent, Freddie Witherden, Brian Vermeire, Jin Seok Park, and Arvind Iyer. Towards Green Aviation with Python at Petascale. In *SC16 Int. Conf. High Perform. Comput. Networking, Storage Anal.*, number November, pages 1–11. IEEE, nov 2016.

[6] Guillermo Oyarzun, Ricard Borrell, Andrey Gorobets, and Assensi Oliva. Portable implementation

model for CFD simulations. Application to hybrid CPU/GPU supercomputers. *Int. J. Comut. Fluid Dyn.*, 31(9):396–411, oct 2017.

[7] Xavier Álvarez, Andrey Gorobets, F. Xavier Trias, Ricard Borrell, and Guillermo Oyarzun. HPC^2–A fully-portable, algebra-based framework for heterogeneous computing. Application to CFD. *Comput. Fluids (published online)*, 2018.

[8] Xavier Álvarez, Andrey Gorobets, and F. Xavier Trias. Strategies for the heterogeneous execution of large-scale simulations on hybrid supercomputers. In *7th Eur. Conf. Comput. Fluid Dyn.*, 2018.

[9] P. K. Sweby. High Resolution Schemes Using Flux Limiters for Hyperbolic Conservation Laws. *SIAM J. Numer. Anal.*, 21(5):995–1011, 1984.

[10] Charles Hirsch. *Numerical Computation of Internal and External Flows. Volume 1*. John Wiley & Sons, 2007.

[11] Néstor Balcázar, Lluís Jofre, Oriol Lehmkuhl, Jesús Castro, and Joaquim Rigola. A finite-volume/level-set method for simulating two-phase flows on unstructured grids. *Int. J. Multiph. Flow*, 64:55–72, sep 2014.

[12] F. Xavier Trias, Oriol Lehmkuhl, Assensi Oliva, C. D. Pérez-Segarra, and R. W. C. P. Verstappen. Symmetry-preserving discretization of Navier–Stokes equations on collocated unstructured grids. *J. Comput. Phys.*, 258:246–267, feb 2014.

[13] Nicolas Valle, Xavier Alvarez, F. Xavier Trias, Jesus Castro, and Assensi Oliva. Algebraic implementation of a flux limiter for heterogeneous computing. In *Tenth Int. Conf. Comput. Fluid Dyn.*, 2018.

[14] M. Oevermann and R. Klein. A Cartesian grid finite volume method for elliptic equations with variable coefficients and embedded interfaces. *J. Comput. Phys.*, 219(2):749–769, dec 2006.