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 and making the design of computing applications a rather complex problem. Many scientific computing applications have been partially ported (even rewritten entirely) to take advantage of the coprocessor devices (i.e. GPUs or MICs). Therefore, in this context of accelerated innovation, we developed the HPC^2 (Heterogeneous Portable Code for HPC). It is a portable, algebra-based framework for heterogeneous computing with many potential applications in the fields of computational physics and mathematics. In this work, we present an algebraic implementation of a flux limiter and define its implementation into the HPC^2 framework. As a result, in the evaluation of the advection of a scalar field, the algorithm of the time-integration phase relies on a reduced set of algebraic operations. This algebraic approach combined with a multilevel MPI+OpenMP+OpenCL parallelisation naturally provides modularity and portability. The advection of a rhodorea in a 2D domain with different velocity fields and boundary conditions has been simulated to validate the implementation of the flux limiter on both CPU and GPU.

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

1 Introduction

During the last years, there has been a significant increment in the variety of hardware. Massively-parallel devices of various architectures have been incorporated to the modern supercomputers 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 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. Consequently, 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 [1] the reader can find the solution of incompressible two-phase flows on multi-GPU. Furthermore, examples of heterogeneous implementations of CFD algorithms for hybrid CPU+GPU supercomputations can be found in [2, 3], and an example of a petascale CFD simulation on 18.000 K20X GPUs in [4].

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 kernels represent linear algebra 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. [5] 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 exciting results of the algebraic implementation, we increased the level of abstraction and presented in [6] 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 [7] 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 by adding flux limiters to construct high-resolution schemes to obtain second-order approximations and avoid oscillations near discontinuities or shocks. The algebraic formulation of a 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 the high potential of this implementation approach again showing that only the addition of six simple algebraic kernels is required.

2 High resolution schemes for the HPC^2

Flux limiters are non-linear functions commonly used to construct high-resolution schemes with the aim of obtaining second-order approximations and avoiding oscillations near discontinuities or shocks. In this section, we describe the integration of a flux limiter into the HPC^2, a fully-portable, algebra-based framework for heterogeneous computing. For this purpose, the flux limiter must be somehow rewritten in an algebraic form so that stencil operations are avoided during the computation. Then, if the majority of kernels represent linear algebra operations, both the standard optimised libraries (*e.g.* ATLAS, clBLAST) and the specific in-house implementations can be used and easily switched.

The algebraic formulation of a flux limiter

Let us consider the typical form of a flux limiter for finite volume methods [8],

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

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 (for a detailed explanation of the mathematical background and formulation below, the reader is referred to Valle et al. [9] and Trias et al. [10]):

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

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

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

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 [9]. The matrix $\Omega(\boldsymbol{r}_s)$ is a variable and diagonal which represents the term $(\Psi(r)-1)/2$ of Equation (2). Then, considering the SUPERBEE flux limiter scheme [8], the elements in the diagonal of $\Omega(\boldsymbol{r}_s)$ become

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

Finally, 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 of the advection with the SUPERBEE flux limiter

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

3 Implementation of the flux limiter into the HPC^2

In our previous works [5, 7], we proposed a portable implementation model for direct numerical simulations (DNS) and large eddy simulations (LES) of incompressible turbulent flows on unstructured meshes. As a result, the algorithm of the time-integration step 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 this implementation approach again showing that only the addition of six simple algebraic kernels is required for the integration of the flux limiter into our

fully-portable, algebra-based framework. These new kernels are described below.

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

The six new kernels above do not show appreciable differences regarding their computational behaviour respect to 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 similar to that of the `axpy`, so they are memory-bounded too. Therefore, having already efficient OpenMP, OpenCL and CUDA implementations of `axpy`, that of this six kernels is straightforward.

We show in Table 3 the number of times that each algebraic kernel is executed in every time-step in the numerical Algorithm 1. In Figure 3 a comparison of the relative time taken by the kernels (for simplicity, the vector kernels have been grouped) in both CPU and GPU is shown. This comparison demonstrates that our implementation model relies almost completely on the algebraic kernels. The 96% and 88% of the computational time is spent running kernels in the CPU and GPU respectively. The loss in performance is due to the *others* group, which encompresses operations that are not directly involved with the algorithm such that copying intermediate vectors or updating the coefficients of the variable matrices. Therefore, it is necessary to reduce the cost of this extra operations introducing generalised kernels to avoid intermediate data storage.

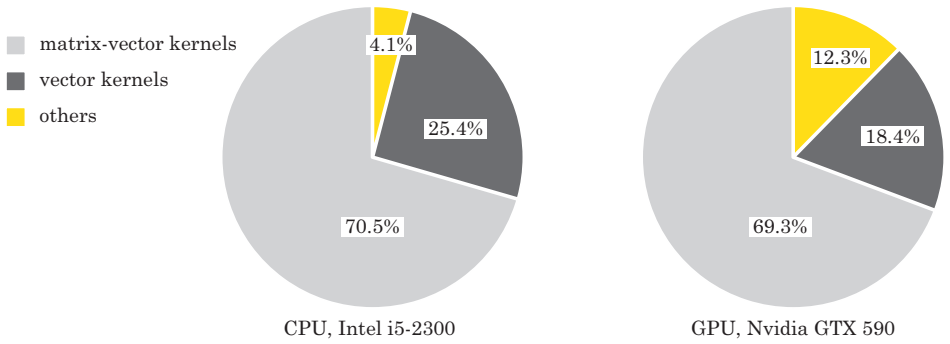

Figure 1: Comparison of the computational cost of the operations in one time step.

Table 1: 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 $\mathbb{Q}(\boldsymbol{u}_s)$	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 flux limiter implementation has been tested on a single node equipped with an Intel CPU and an

NVIDIA GPU, and the results in 3 confirm again that the overall performance is only depending on the performance of the kernels. Hence, the performance presented in this paper can be extrapolated to that of the study in [7]. Nevertheless, instead of being an inconvenience, this encourages us to demonstrate again the high potential of this implementation approach showing that only the addition of six simple algebraic kernels is required for the integration of the flux limiter into our fully-portable, algebra-based framework.

4 Numerical results

The advection of a scalar field with a high resolution scheme (see Algorithm 1 is evaluated. The marker function is initialised in a 2D domain with the shape of a rhodorea [11] for three different structured grids: 32x32, 128x128 and 512x512. Two different fixed velocity fields are evaluated: the rotating field $\boldsymbol{u} = (y, -x)$, together with dirichlet boundary conditions in all the boundaries, and the flat $\boldsymbol{u} = (0, 1)$ combined with periodic boundary conditions on top and bottom.

The Figure 4 shows various plots of the marker function for the 32x32 grid (top) and the 512x512 (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 two cicles under the flat velocity field. On the right hand side, the final state of $\boldsymbol{\theta}_c$ after one revolution under the rotating velocity field.

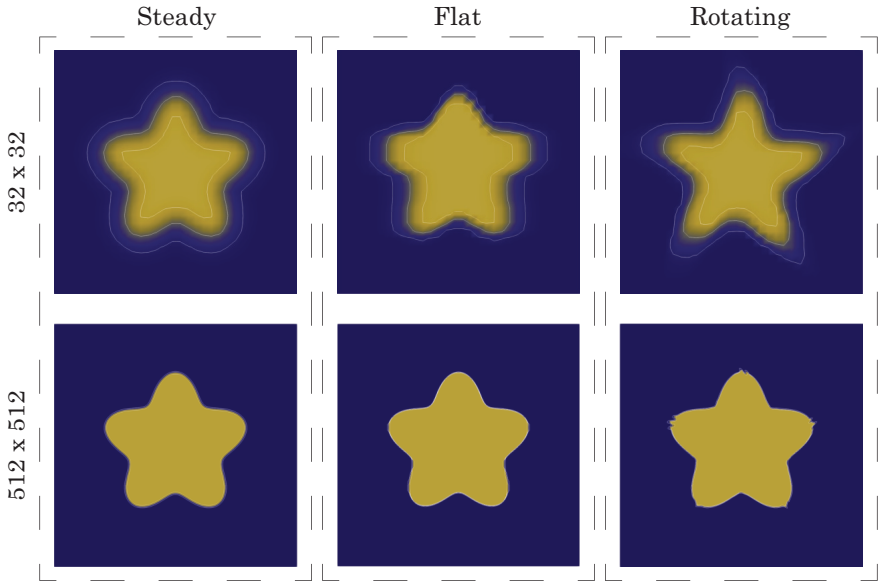

Figure 2: Plots of the marker function $\boldsymbol{\theta}_c$ for different grids and cases.

Finally, we have studied the evolution of the error, measured as follows:

$$\epsilon = \frac{||\boldsymbol{\theta}_f - \boldsymbol{\theta}_i||}{||\boldsymbol{\theta}_i||}. \tag{7}$$

The error of both the flat and the rotating simulations is shown in Figure 4 for the three different grids.

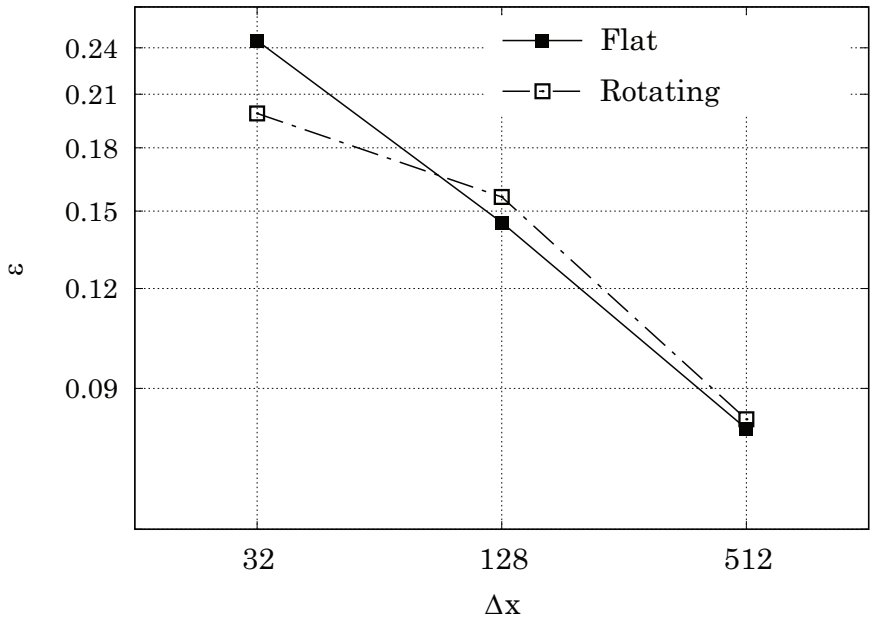

Figure 3: Error versus number of control volumes.

5 Conclusion and Future Work

An algebraic formulation of a high resolution scheme has been presented. The flux limiter has been implemented into the HPC^2 framework. We have shown that the addition of only six algebraic kernels into our framework is sufficient for simulating the advection of a scalar field with a flux limiter. The advection of a 2D scalar field has been computed using different grids, velocity fields and boundary conditions. The simulations have been run on both CPU and GPU. Hence, our fully-portable, algebra-based framework for heterogeneous computing has demonstrated its great potential for large-scale simulations 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] Peter Zaspel and Michael Griebel. Solving incompressible two-phase flows on multi-GPU clusters. *Computers and Fluids*, 80(1):356–364, 2013.

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

[3] 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. *Journal of Computational Physics*, 278(1):275–297, dec 2014.

[4] Peter E. Vincent, Freddie Witherden, Brian Vermeire, Jin Seok Park, and Arvind Iyer. Towards Green Aviation with Python at Petascale. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, number November, pages 1–11. IEEE, nov 2016.

[5] Guillermo Oyarzun, Ricard Borrell, Andrey Gorobets, and Assensi Oliva. Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers. *International Journal of Computational Fluid Dynamics*, 31(9):396–411, oct 2017.

[6] Xavier Álvarez, Andrey Gorobets, F. Xavier Trias, Ricard Borrell, and Guillermo Oyarzun. HPC2âĂŤA fully-portable, algebra-based framework for heterogeneous computing. Application to CFD. *Computers and Fluids*, 0:1–8, 2018.

[7] Xavier Álvarez, Andrey Gorobets, and F. Xavier Trias. Strategies for the heterogeneous execution of large-scale simulations on hybrid supercomputers. In *7th European Conference on Computational Fluid Dynamics*, 2018.

[8] P. K. Sweby. High Resolution Schemes Using Flux Limiters for Hyperbolic Conservation Laws. *SIAM Journal on Numerical Analysis*, 21(5):995–1011, 1984.

[9] Nicolas Valle, Xavier Alvarez, F. Xavier Trias, Jesus Castro, and Assensi Oliva. Algebraic implementation of a flux limiter for heterogeneous computing. In *Tenth International Conference on Computational Fluid Dynamics*, 2018.

[10] F. Xavier Trias, Oriol Lehmkuhl, Assensi Oliva, C. D. Pérez-Segarra, and R. W. C. P. Verstappen. Symmetry-preserving discretization of NavierâĂŞStokes equations on collocated unstructured grids. *Journal of Computational Physics*, 258:246–267, feb 2014.

[11] M. Oevermann and R. Klein. A Cartesian grid finite volume method for elliptic equations with variable coefficients and embedded interfaces. *Journal of Computational Physics*, 219(2):749–769, dec 2006.