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

Xavier Álvarez[1], Nicolás Valle[1], Andrey Gorobets[2], F. Xavier Trias[1]

In the tenth International Conference on Computational Fluid Dynamics (ICCFD10)
9–13 June 2018, **Barcelona**

[1] Heat and Mass Transfer Technological Center, **Technical University of Catalonia** (UPC)
[2] Keldysh Institute of Applied Mathematics, **Russian Academy of Science** (RAS)

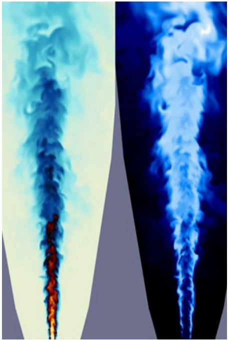
Centre Tecnològic de Transferència de Calor
UNIVERSITAT POLITÈCNICA DE CATALUNYA

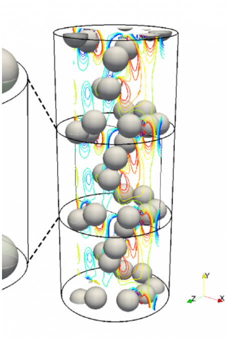
Motivation

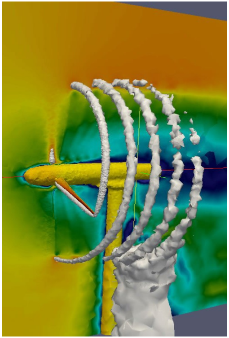
The CTTC research group

The Heat and Mass Transfer Technological Center (CTTC) has been working on CFD for more than 20 years:

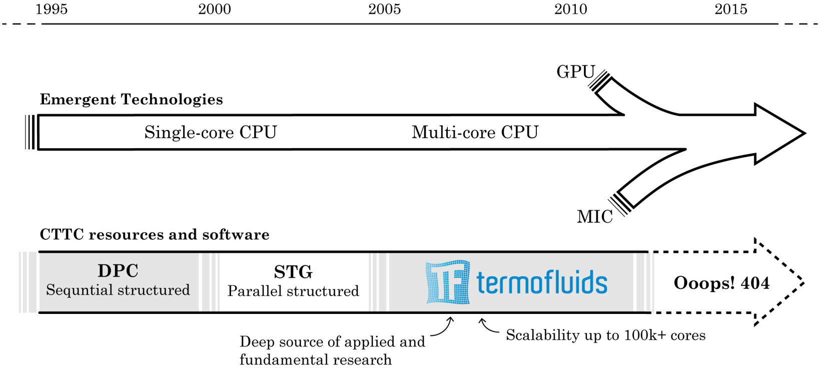
- **Fundamental research** on fluid dynamics and heat and mass transfer phenomena.
- **Applied research** on thermal and fluid dynamic optimization of thermal system and equipment.







CTTC's historical background in HPC



1995 2000 2005 2010 2015

Emergent Technologies

Single-core CPU

Multi-core CPU

GPU

MIC

CTTC resources and software

DPC
Sequntial structured

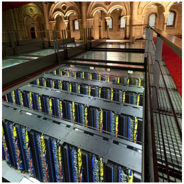
STG
Parallel structured

termofluids

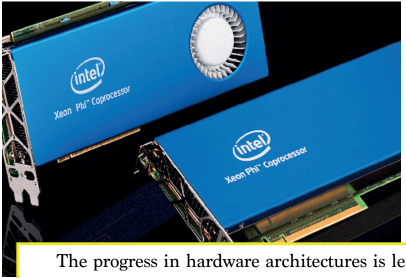
Ooops! 404

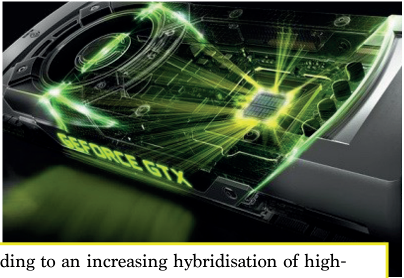
Deep source of applied and fundamental research

Scalability up to 100k+ cores

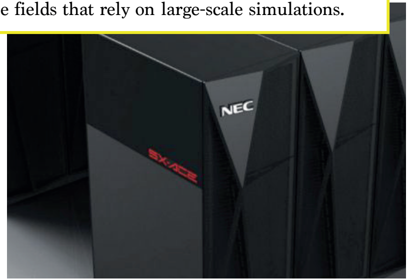


Divergence of HPC systems





The progress in hardware architectures 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.



Divergence of HPC systems

Fully-portable implementation models

Is it **necessary** to use the new hardware architectures?

- In our opinion, **yes**. New hardware is designed to overcome the power constraint in the context of the exascale challenge.

Fully-portable implementation models

Is it **necessary** to use the new hardware architectures?

- In our opinion, **yes**. New hardware is designed to overcome the power constraint in the context of the exascale challenge.

Do the **traditional** implementation models facilitate code portability?

- In our opinion, **no**. The legacy software was not designed for providing portability simply because it was not necessary. Hence, they involve a large number of functions and data structures that only fit properly into the CPU-only approach.

Fully-portable implementation models

Is it **necessary** to use the new hardware architectures?

- In our opinion, **yes**. New hardware is designed to overcome the power constraint in the context of the exascale challenge.

Do the **traditional** implementation models facilitate code portability?

- In our opinion, **no**. The legacy software was not designed for providing portability simply because it was not necessary. Hence, they involve a large number of functions and data structures that only fit properly into the CPU-only approach.

Do we need to **change** the way we look at scientific computing in general?

- In our opinion, **yes**. Making an effort to design modular applications composed of a reduced number of independent and well-defined code blocks helps to reduce the generation of errors and facilitates debugging. Furthermore, modular applications are user-friendly and more comfortable for porting to new architectures.

Fully-portable implementation models

Stencil-based

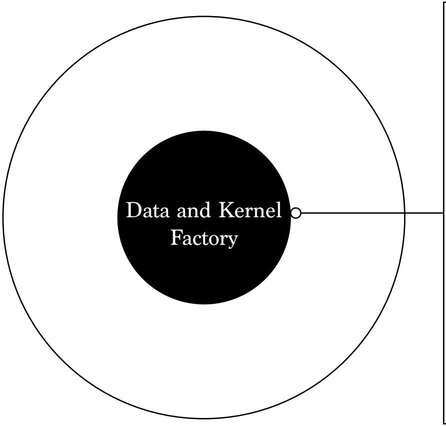
Traditionally, the stencil-based implementations are used by the scientific computing community. These implementations arise straightforward from the formulation of the numerical method. However, they require **specific stencil sweeps and data structures** for each numerical method.

Algebra-based

Algebra-based implementations only rely on a reduced number of **universal algebraic kernels and data structures,** allowing the use of standard optimised libraries and, therefore, providing portability. As a counterpart, the formulation of the numerical method becomes more complex and could even lead to an increase in the number of operations.

The HPC^2 fully-portable, algebra-based framework

The HPC^2 (Heterogeneous Portable Code for HPC) is a fully-portable, algebra-based framework with many potential applications in the fields of computational physics and mathematics. Its algebraic approach combined with a multilevel MPI + OpenMP + OpenCL + CUDA parallelisation naturally provides modularity and portability.



Data and Kernel Factory

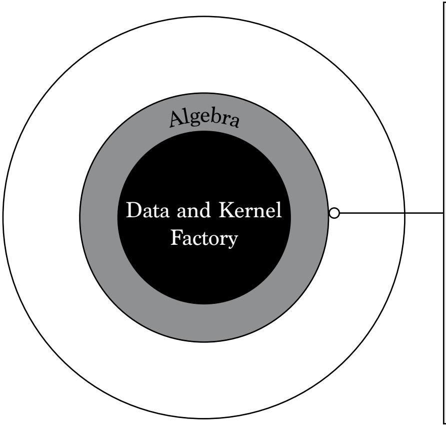
The Data and Kernel Factory layer holds the data containers and the computing kernels.

```
SpMV(A, u, y)
axpy(u, v, y)
ddot(u, v)
```

Its layout, inspired in the **abstract factory** design pattern, employs virtual objects for minimising the dependencies. Thus, it allows adding a new implementation of data containers and computing kernels easily.

The HPC^2 fully-portable, algebra-based framework

The HPC^2 (Heterogeneous Portable Code for HPC) is a fully-portable, algebra-based framework with many potential applications in the fields of computational physics and mathematics. Its algebraic approach combined with a multilevel MPI + OpenMP + OpenCL + CUDA parallelisation naturally provides modularity and portability.



Algebra

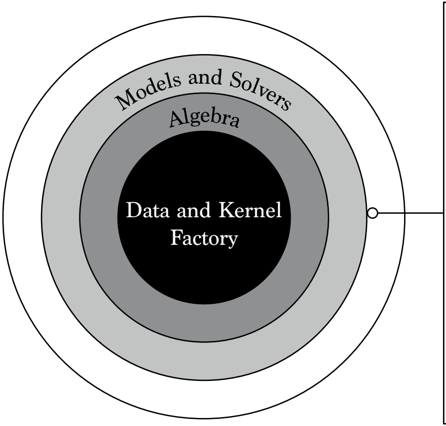
The Algebra layer contains classes and objects that mimic algebraic structures. They wrap data containers and computing kernels from the abstract factories, which makes them entirely independent of the implementation.

```
Set
VectorSpace
DifferentialManifold
```

Differential operators (i.e. DIV, GRAD) can be generated by means of whatever numerical method such as FEM, FVM. They're just matrices.

The HPC^2 fully-portable, algebra-based framework

The HPC^2 (Heterogeneous Portable Code for HPC) is a fully-portable, algebra-based framework with many potential applications in the fields of computational physics and mathematics. Its algebraic approach combined with a multilevel MPI + OpenMP + OpenCL + CUDA parallelisation naturally provides modularity and portability.



Models and Solvers

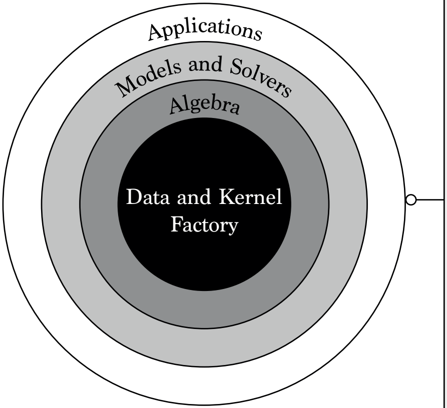
The Models and Solvers layer engages the algebraic objects for generating both physical models and solver methods.

```
NavierStokes
RungeKutta
ConjugateGradient
```

Recall that the numerical method (i.e. FEM, FVM, FDM) is used only at the preprocessing stage for building the matrices.

The HPC^2 fully-portable, algebra-based framework

The HPC^2 (Heterogeneous Portable Code for HPC) is a fully-portable, algebra-based framework with many potential applications in the fields of computational physics and mathematics. Its algebraic approach combined with a multilevel MPI + OpenMP + OpenCL + CUDA parallelisation naturally provides modularity and portability.



Applications

The Application layer is where the user can combine models and solvers to set-up the case for running a simulation.

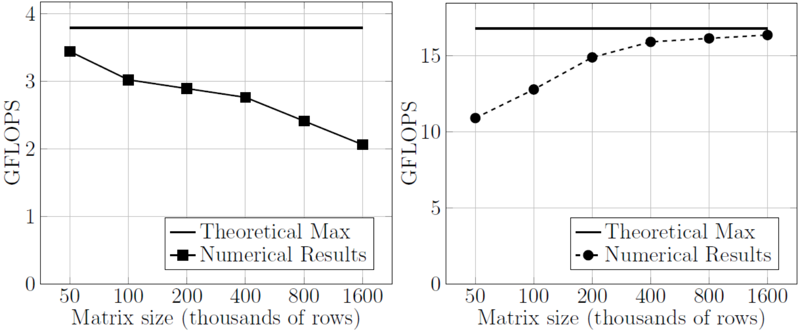
```
DrivenCavity
RisingBubble
FallingFilm
```

In summary, everything that can be expressed by means of algebraic structures such as Matrix and Vector, it can be run within this framework.

A performance overview of the HPC^2

Study case 1

Single-device performance of the SpMV kernel *vs* the matrix size on an Intel Xeon E5649 (left) and Nvidia M2090 (right) for a matrix derived from a symmetry-preserving discretisation[1] on an unstructured hex-dominant mesh.

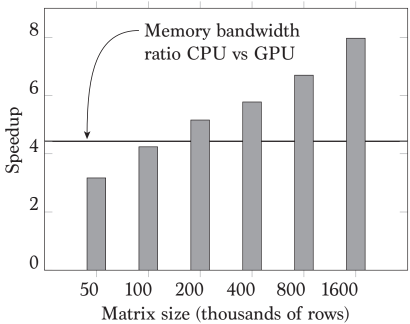


[1] F.X. Trias et al., Symmetry-preserving discretization of Navier-Stokes equations on collocated unstructured grids, **J.Comp.Phys.**, 258, 246-267, 2014.

A performance overview of the HPC^2

Study case 1

Single-device performance of the SpMV kernel *vs* the matrix size on an Intel Xeon E5649 (left) and Nvidia M2090 (right) for a matrix derived from a symmetry-preserving discretisation[1] on an unstructured hex-dominant mesh.



CPU vs GPU

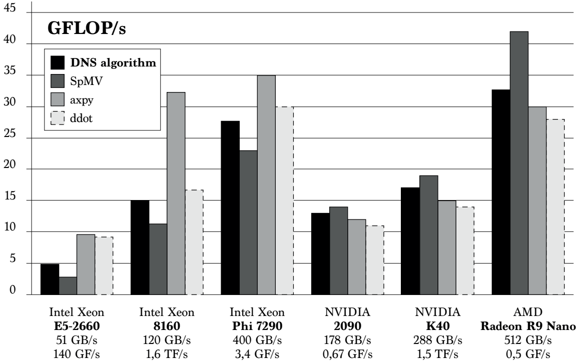
In memory-bounded applications, the performance depends on the memory bandwidth. However, the GPU relative performance improves with the size of the matrix, in contrast with that of the CPU. Hence, the speedup depends on both the matrix size and the memory bandwidth.

[1] F.X. Trias et al., Symmetry-preserving discretization of Navier-Stokes equations on collocated unstructured grids, **J.Comp.Phys.**, 258, 246-267, 2014.

A performance overview of the HPC[2]

Study case 2

Single-device performance comparison of the algebraic DNS algorithm using the symmetry-preserving discretisation[2] on an unstructured hex-dominant mesh of 1M cells.



Hardware Architectures

In line with the conclusion in the Study case 1, the performance comparison for fixed mesh sizes is not fair enough; we do not observe a linear growth respect to the memory bandwidth because the relative performance of each hardware architecture depends on the mesh size.

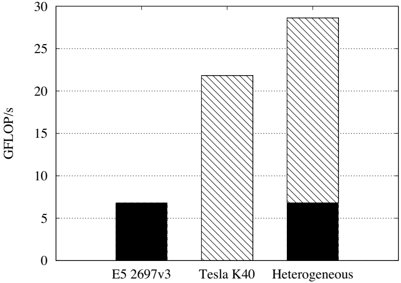
To hit the optimal heterogeneous performance, it is necessary to assign to each device the domain partition that maximises its performance.

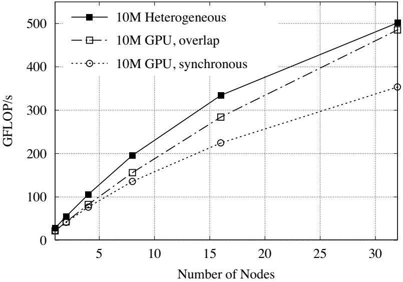
[2] F.X. Trias et al., Symmetry-preserving discretization of Navier-Stokes equations on collocated unstructured grids, **J.Comp.Phys.**, 258, 246-267, 2014.

A performance overview of the HPC^2

Study case 3

Heterogeneous performance study of the SpMV kernel on a hybrid node equipped with an Intel E5 2697v3 and an Nvidia Tesla K40 for a matrix derived from a symmetry-preserving discretisation[3] on an unstructured hex-dominant mesh of 10M cells. On the left, the sinlge-node performance study. On the right, the strong-scaling study.





[3] F.X. Trias et al., Symmetry-preserving discretization of Navier-Stokes equations on collocated unstructured grids, **J.Comp.Phys.**, 258, 246-267, 2014.

Implementation of a flux limiter into the HPC^2

Algebraic 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. The typical form of a flux limiter for finite volume methods reads:

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

Algebraic 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. The typical form of a flux limiter for finite volume methods reads:

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

Similarly expressed in the less common form:

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

Algebraic 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. The typical form of a flux limiter for finite volume methods reads:

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

Similarly expressed 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 above is written as follows:

$$\boldsymbol{\theta}_s = \left(\Pi_{c\to s} + \Omega(\boldsymbol{r}_s)\mathsf{Q}(\boldsymbol{u}_s)\Delta_{c\to s}\right)\boldsymbol{\theta}_c \tag{3}$$

Detail of the algebraic implementation

Equivalent terms between the analytical and the operator-based, discrete form.

$$\begin{aligned} \theta_f &= \frac{\theta_U + \theta_D}{2} + \frac{\Psi(r) - 1}{2}(\theta_D - \theta_U) \\ \boldsymbol{\theta}_s &= \Pi_{c \to s}\boldsymbol{\theta}_c + \Omega(r_s) \qquad \mathsf{Q}(u_s)\Delta_{c \to s}\boldsymbol{\theta}_c \end{aligned}$$

Definition

$\Pi_{c \to s}$ is the the cell-to-face scalar interpolator[4].

[4] F.X. Trias et al., Symmetry-preserving discretization of Navier-Stokes equations on collocated unstructured grids, **J.Comp.Phys.**, 258, 246-267, 2014.

Detail of the algebraic implementation

Equivalent terms between the analytical and the operator-based, discrete form.

$$\begin{aligned} \theta_f &= \frac{\theta_U + \theta_D}{2} + \frac{\Psi(r) - 1}{2} (\theta_D - \theta_U) \\ \boldsymbol{\theta}_s &= \Pi_{c \to s} \boldsymbol{\theta}_c + \Omega(\boldsymbol{r}_s) \qquad \mathsf{Q}(\boldsymbol{u}_s) \Delta_{c \to s} \boldsymbol{\theta}_c \end{aligned}$$

Definition

$\boldsymbol{r}_s$ is the the discontinuity sensor, chosen here as the gradient ratio.

The gradient ratio, $\boldsymbol{r}_s$, is given by[5]:

$$\boldsymbol{r}_s(\boldsymbol{\theta}_c) = \frac{\left(\mathsf{Q}(\boldsymbol{u}_s)\mathsf{UUD}_{c \to s} + \mathsf{OUD}_{c \to s}\right) \boldsymbol{\theta}_c}{\left(\mathsf{Q}(\boldsymbol{u}_s)\Delta_{c \to s}\right) \boldsymbol{\theta}_c} \tag{4}$$

[5] N. Valle et al., Algebraic implementa- tion of a flux limiter for heterogeneous computing, **Tenth International Conference on Computational Fluid Dynamics**, Barcelona, 2018.

Detail of the algebraic implementation

Equivalent terms between the analytical and the operator-based, discrete form.

$$\begin{aligned} \theta_f &= \frac{\theta_U + \theta_D}{2} + \frac{\Psi(r) - 1}{2}(\theta_D - \theta_U) \\ \boldsymbol{\theta}_s &= \Pi_{c \to s}\boldsymbol{\theta}_c + \Omega(\boldsymbol{r}_s) \quad\quad \mathsf{Q}(\boldsymbol{u}_s)\Delta_{c \to s}\boldsymbol{\theta}_c \end{aligned}$$

Definition

$\Omega(\boldsymbol{r}_s)$ represents the flux limiter term.

The diagonal of $\Omega(\boldsymbol{r}_s)$, considering a SUPERBEE flux limiter[6], is obtained as:

$$diag(\Omega(\boldsymbol{r}_s)) = \frac{max(0, max(min(1, 2\boldsymbol{r}_s), min(\boldsymbol{r}_s, 2))) - 1}{2} \tag{5}$$

[6] P. K. Sweby, High resolution schemes using flux limiters for hyperbolic conservation laws, **SIAM Journal on Numerical Analysis**, 21(5), 995-1011, 1984.

Detail of the algebraic implementation

Equivalent terms between the analytical and the operator-based, discrete form.

$$\begin{aligned}\theta_f &= \frac{\theta_U + \theta_D}{2} + \frac{\Psi(r) - 1}{2}(\theta_D - \theta_U) \\ \boldsymbol{\theta}_s &= \Pi_{c\rightarrow s}\boldsymbol{\theta}_c + \Omega(\boldsymbol{r}_s) \qquad \mathsf{Q}(\boldsymbol{u}_s)\Delta_{c\rightarrow s}\boldsymbol{\theta}_c\end{aligned}$$

Definition

$\Delta_{c\rightarrow s}$ is the scalar cell-to-face difference operator.

Definition

$\mathsf{Q}(\boldsymbol{u}_s)$ 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 computed as follows:

$$diag(\mathsf{Q}(\boldsymbol{u}_s)) = sign(\boldsymbol{u}_s) \tag{6}$$

Definition of the required new kernels

Six new kernels are required for the implementation of the flux limiter. They are simple **element-wise operations over the vectors** (like `axpy`) hence they are not involved in distributed-memory communications. Besides, they provide a **uniform aligned memory access** with coalescing of memory transactions which suit the stream processing paradigm perfectly. Its arithmetic intensity is very low (*i.e.* the number of FLOP per byte); thus they are **memory-bounded kernels** too. Therefore, having already efficient MPI + OpenMP + OpenCL + CUDA implementations of `axpy`, implementing the six new kernels below is straightforward.

`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)` $\longrightarrow y_i = \{-1 \text{ if } x_i < 0,\ 1 \text{ otherwise}\}.$

Numerical results

Time-integration algorithm for the advection of a scalar field

We consider the simulation of the **advection of a scalar field with sharp discontinuities** for different shapes, velocity fields and mesh sizes, using the algebraic implementation of the flux limiter.

The complete algorithm for the time-integration of the advection equation using the algebraic formulation of the SUPERBEE flux limiter and a 1st order Euler method is described below.

Algorithm 1 Time-integration step

```
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. Integrate $\boldsymbol{\theta}_c^{n+1} = \boldsymbol{\theta}_c^{n} - dt \cdot \mathsf{DIV}_{s\to c}\mathsf{U}_s\,(\Pi_{c\to s} + \Omega(\boldsymbol{r}_s)\mathsf{Q}(\boldsymbol{u}_s)\Delta_{c\to s})\,\boldsymbol{\theta}_c$
```

Analysis of kernel calls

We consider the simulation of the **advection of a scalar field with sharp discontinuities** for different shapes, velocity fields and mesh sizes, using the algebraic implementation of the flux limiter.

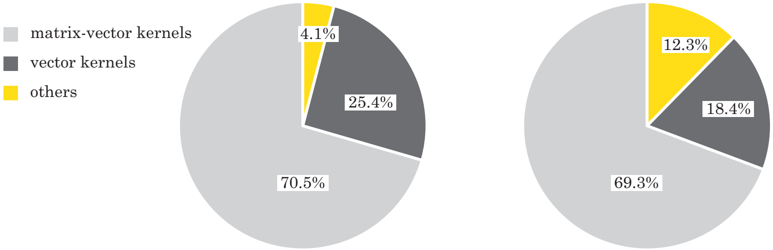
The table below shows the number of times that each algebraic kernel is called in every time-step of our simulation.

Step of Algorithm	`SpMV`	`axpy`	`axdy`	`shft`	`scal`	`vmax,`	`smax,`	`sign`
						`vmin`	`smin`	
1 – Compute $Q(u_s)$	0	0	0	0	0	0	0	1
2 – Compute r_s	5	1	1	0	0	0	0	0
3 – Compute $\Omega(r_s)$	0	0	0	1	2	1	3	0
4 – 1st order Euler	6	2	0	0	0	0	0	0
Total	11	3	1	1	2	2	2	1

Analysis of kernel calls

We consider the simulation of the **advection of a scalar field with sharp discontinuities** for different shapes, velocity fields and mesh sizes, using the algebraic implementation of the flux limiter.

The simulations after **have been run in both CPU and GPU**. The comparison of the relative time spent in each operation in both CPU and GPU is shown in the figure below (for simplicity, the vector kernels have been grouped). The *others* group refers to operations that are not directly involved with the algorithm such as the printing of simulation outputs.



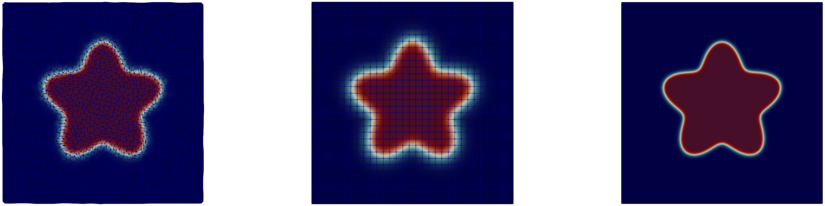
CPU, Intel i5-2300

GPU, Nvidia GTX 590

Pure advection simulation

Study case 4

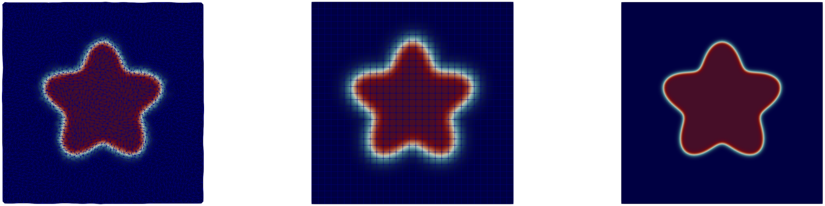
Simulation of the **advection of a 2D rhodonea** with a flat velocity field for an unstructured mesh of 1K cells (left), a structured mesh of 32x32 (center) and a structured mesh of 128x128 cells (right).



Pure advection simulation

Study case 5

Simulation of the **advection of a 2D rhodonea** with a rotating velocity field for an unstructured mesh of 1K cells (left), a structured mesh of 32x32 (center) and a structured mesh of 128x128 cells (right).

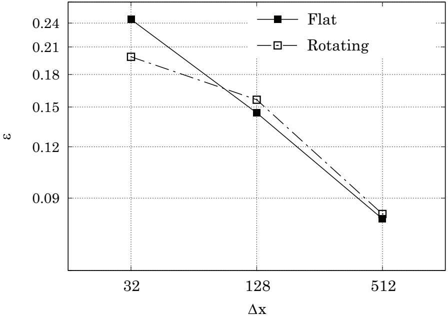


Pure advection simulation

A preliminary study of the error of both the flat and the rotating simulations is shown in the figure below for the structured mesh of 32x32, 128x128 and 512x512 cells.

The error has been computed as the norm of the difference between the final and initial values of the marker function, normalised with the norm of the initial value as follows:

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



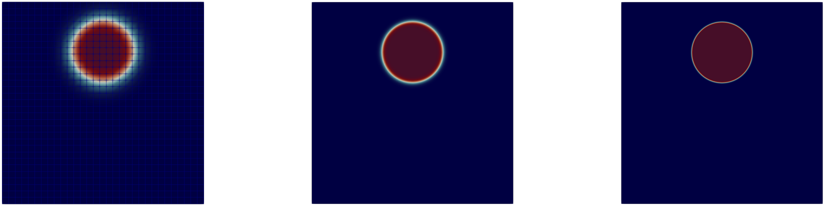
Level-set simulation

Study case 6

Simulation of the **advection of a 2D circle** using the **level-set method** with a vortical velocity field for a structured mesh of 32x32 cells (left), a structured mesh of 128x128 (center) and a structured mesh of 512x512 cells (right).

A third-order Runge-Kutta time-integration scheme is used for the reinitialisation of the interface for three pseudo time-steps.

NOTE: Our algebraic implementation of the level-set method is recent and still under analysis. We will extend the details of the implementation and the results soon.



Conclusions and future work

Conclusions and future work

In this work...

- An algebraic formulation of a high-resolution scheme has been presented.
- A flux limiter has been implemented into the HPC^2 framework.
- We have shown that the addition of only six simple algebraic kernels is sufficient to implement high-resolution, non-linear schemes into our framework.
- The simulation of the advection of different 2D marker functions in both CPU and GPU has been shown, including the preliminary results with the level-set method.
- **We have shown that the algebra-based approach naturally provides with modularity and portability**.

In future, we aim at...

- Optimising the in-house algebraic kernels for specific architectures.
- Extending the implementation to new hybrid architectures.
- Improving the parallel and the heterogeneous efficiency.
- Addapting more challenging numerical methods to the algebra-based framework.
- Coupling optimal multilevel meshing and partitioning tools.
- Enhancing the user interface.