



DAAA, ONERA, CHATILLON

Rapport Technique

Rapport final PRF CHAMPION

RT 5/25097 DAAA – février 2020

Ivan Mary, Nicolas Alferez, Guillaume
Jeanmasson

FÉVRIER 2020

TABLE DES MATIÈRES

1.	INTRODUCTION	3
1.1.	Rappel des objectifs et moyens du projet	3
1.2.	Les choix techniques non négociables	3
2.	NEXT GENERATION CFD SOLVERS ARCHITECTURE	5
2.1.	Onera's Open system software implementation	5
2.1.1.	<i>Standard Data Model</i>	5
2.1.2.	<i>Software components policy</i>	6
2.2.	Simple exemples of end-user script	10
2.2.1.	<i>Flat plate computation with structured FastS module</i>	10
2.2.2.	<i>Data access in the tree variable</i>	11
2.3.	Schematic decomposition of one Navier-Stokes solver iteration	12
2.4.	Solver data structures	14
2.4.1.	<i>FastS data structures</i>	15
2.4.2.	<i>FastP data structures</i>	17
2.4.3.	<i>Connector : tc data structures</i>	19
3.	HPC LAYER	29
3.1.	factorization of a Navier-Stokes workflow	29
3.2.	Vectorization	31
3.3.	Cache Blocking technique	33
3.4.	Parallelisation	35
3.4.1.	<i>Parallelisation by OpenMP threads</i>	35
3.4.2.	<i>Parallelisation by MPI</i>	39
3.5.	Automatic code generation	40
3.5.1.	<i>Source sharing within a module : flux assembly and balance function</i>	40
3.5.2.	<i>Source sharing between Navier-Stokes modules : FastS, FastP, ...</i>	47
3.5.3.	<i>Adjoint by automatic differentiation</i>	52
3.6.	Performance results	52
3.6.1.	<i>Test case : 3D vortex advection and performance model</i>	52
3.6.2.	<i>Hardware architecture</i>	53
3.6.3.	<i>Importance of NUMA at the node level</i>	53
3.6.4.	<i>Performance comparisons at fully loaded node</i>	54
3.6.5.	<i>Strong problem size dependency : a cache-associativity issue ?</i>	56
3.6.6.	<i>Typical Cells Update Per Second</i>	57
3.6.7.	<i>Scalability at the cluster level : PRACE preparatory</i>	57

4.	EFFICACITÉ DES MÉTHODES NUMÉRIQUES	61
4.1.	Introduction	61
4.2.	Local time stepping scheme for unsteady computation	61
4.2.1.	<i>Local time stepping scheme and Partitioned Runge-Kutta methods</i>	63
4.2.2.	<i>New schemes</i>	69
4.2.3.	<i>Numerical tests</i>	74
4.2.4.	<i>Conclusion</i>	84
5.	INTERFACES	87
6.	APPLICATIONS	89
7.	CONCLUSIONS	91
8.	BIBLIOGRAPHIE	95

FÉVRIER 2020

1. INTRODUCTION

1.1. Rappel des objectifs et moyens du projet

Ce rapport synthétise les actions menées au cours du PRF CHAMPION (janvier 2016 à décembre 2018) dont le but était de préparer l'offre ONERA en matière de simulation numérique multi-physique pour les années 2020-2030 par le biais d'un nouveau prototype logiciel. Celui-ci devait s'appuyer sur l'expérience acquise dans les logiciels Cedre et elsA et permettre de concevoir une architecture ouverte, d'effectuer les choix algorithmiques adaptés au nouveau contexte HPC et de développer les méthodes numériques manquantes. La souplesse d'utilisation, l'évolutivité et l'efficacité constituaient les grands axes de travail de ce PRF, avec comme cible des simulations nouvelles et d'exception.

Les ressources humaines initialement affectées au PRF étaient de 7500h/an. Ces activités ont dûes être revues très significativement à la baisse faute de ressource disponible pour 2 raisons principales :

- plan de charge contractuel contraignant.
- sous l'impulsion d'Airbus, le renouvellement du grand code Aéro Elsa a pris un sérieux coup d'accélérateur en 2017-2018 avec l'accord de développement du code RHEA entre l'ONERA, le DLR et AIRBUS. Le DLR ayant réussi à imposer ses choix technologiques pour ce nouveau logiciel, le PRF CHAMPION a perdu une grande partie de son intérêt en cours de route. De plus, l'existence de plusieurs prototypes ONERA plus ou moins officiels développés sans trop de concertation a certainement contribué à brouiller le message interne vis à vis des unités applicatives et contribué à leur faible investissement sur le projet. Le PRF ODYSSEY, démarré en janvier 2019 et qui possède de nombreux objectifs communs avec le PRF CHAMPION, apportera certainement des clarifications. On peut toutefois regretter que les principaux acteurs du PRF ODYSSEY n'est pas pu s'impliquer davantage dans le PRF CHAMPION, et vice versa, compte tenu du plan de charge respectif de chacun.

1.2. Les choix techniques non négociables

Au cours des 20 dernières années, l'Onera, à travers ses logiciels *elsA* et *Cassiopee*, a mis en place en environnememnt de simulation basé sur 2 éléments techniques importants :

- le langage Python sert d'interface utilisateur
- l'adoption du standard Python/CGNS comme modèle de données

Ces choix, très avant-gardistes en 2000, sont aujourd'hui plutôt bien partagés par la communauté. Ainsi le groupe Safran a récemment retenu les mêmes options pour développer sa plate-forme industrielle de simulation multi-physiques MOSAIC. Airbus a également retenu le langage python comme interface utilisateur, mais a choisi un modèle de donnée qui lui est propre. Un des objectifs du PRF étaient de démontrer que ces choix étaient pleinement compatible avec l'environnement HPC.

FÉVRIER 2020

FÉVRIER 2020

2. NEXT GENERATION CFD SOLVERS ARCHITECTURE

2.1. Onera's Open system software implementation

Over the 10 last years, Onera has created, developed and delivered its own Open System Software Components approach based on choices which are presented in the following subsections with all components presented previously (*elsA*, *Cassiopee*, *FastP*, *FastS*, *Aghora* and next *NeoN-Solver*). It also important to notice that this Open System Software architecture implementation has been also chosen by Safran Group for its industrial platform, the equivalent of *FlowSimulator*.

2.1.1. Standard Data Model

The standard data model is based on the CFD General Notation System (CGNS, see [1]) which provides a general, portable and extensible standard for the storage and retrieval of computational fluid dynamics (CFD) analysis data. It consists of a collection of conventions and free and open software implementing those conventions. It is self-descriptive, machine-independent, well-documented and administered by an international steering committee to which Airbus and Onera belong.

As a standard CGNS does not define 100% of the data but only the definitions which are agreed between people/organizations. However, the CGNS standard provides users with the ability to add new, not yet standardized, data called *UserDefinedData*. This allows any user to adapt the data structure to specific use cases or components. Any standard modification or new data structure can be proposed, discussed and accepted by the steering committee.

The CGNS data model defines a tree-like specification of information. A tree data structure which can be compared to an enriched file system tree is very common in software applications, it is simple and powerful. A CGNS tree is shown on the figure 2.1 where a consensus was found to describe the main principles of mesh topology by :

- *Base_t* is a logical partition of the mesh (fuselage, wind, etc...)
- *Zone_t* is a sub mesh partition of the *Base_t*
 - *GridCoordinates_t* describes the grid coordinates of the *Zone_t*
 - *Element_t* describes the grid mesh connectivity of the *Zone_t*
 - *ZoneGridConnectivity_t* describes joins between *Zone_t*
 - *ZoneBC_t* describes boundary condition for the *Zone_t*

An exhaustive description of CGNS data representation is available in [2] where all the data necessary for the computation (initial solution, wall distance, mesh coordinates, boundary condition, reference state, governing equations,...), except for the numerical parameters provided in the end-user script can be included.

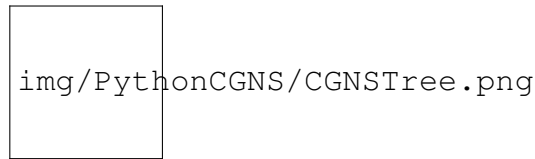


Fig. 2.1 –: CGNS tree description.

This hierarchical description given by the standard is available for structured meshes, unstructured meshes defined by elements or by faces including polyedrical/polygonal representation, hybrid structured/unstructured meshes, overset meshes. A standard is currently under discussion to define high-order meshes and degrees of freedom data structure representation used by Discontinuous Galerkin numerical methods.

The mapping of the CGNS tree, as standard data model in memory uses both Python native list and numpy array which can be considered today as a standard which is extremely portable on all HPC architectures. Each CGNS tree node is fully defined by a name, a type, a data and a list of children as shown on figure 2.2.

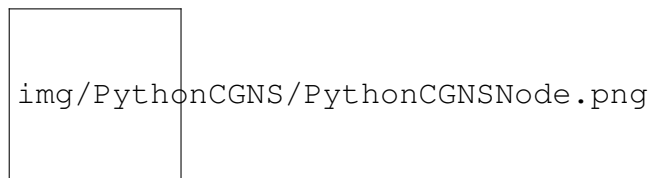


Fig. 2.2 –: Python/CGNS node description.

On figure 2.3, the tree schematically described on the figure 2.1 is now mapped into the Python/CGNS representation.



Fig. 2.3 –: Python/CGNS tree description.

Because in CFD simulations, the number of degree of freedom can be huge, the mapping of the CGNS tree is stored on the file system into the largely used HDF5 format which has been chosen for its efficiency, portability and the high volume and complex data management capabilities.

2.1.2. *Software components policy*

In the scope of the Onera's Open System Software, all components have to be a Python module or package.

FÉVRIER 2020

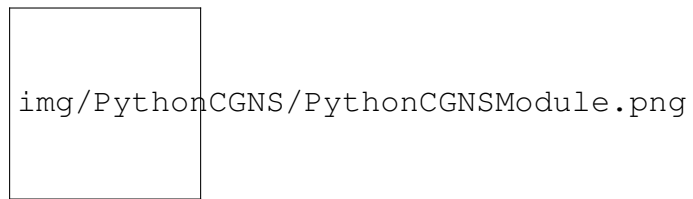


Fig. 2.4 –: Python/CGNS module description.

As expressed in the general definition, the software component is a Python module schematized by the figure 2.4 which is defined by an interface *i.e.* the required/optional input and output values and the dedicated service *i.e.* the core code. This core code can be itself Python module assembly with a defined sequence or a single service.

- *Software components as a single service*

As an example, on the listing 2.1, a single service Python module is implemented as a Python function to scale the coordinates of any meshes defined by the CGNS standard. This module is written in pure Python and takes as input parameters a Python/CGNS tree and a *float* factor. It returns a Python/CGNS tree as output. As one can observe, a Python module called Converter from *Cassiopee* has been developed where functions to load and write a large set of existing formats in particular the HDF5 format are available but also a large set of functions to manipulate Python/CGNS trees and fund nodes by name or by type.

Listing 2.1: example of Python/CGNS module

```
import Converter.PyTree as C
import Converter.Internal as Internal

def scale(t, factor):
    # Access to all grid coordinates for all zones
    for gridCoordsNode in Internal.getNodesFromType(t, "GridCoordinates_t"):
        # Scale Coordinates from millimeters to meters
        for coordsNode in Internal.getNodesFromName(gridCoordsNode, "Coordinate*"):
            coordsNumpy = coordsNode[1]
            coordsNumpy *= factor
    return t

# Read HDF5 file converted to Python/CGNS Tree t
t = C.convertFile2PyTree("DPW-mm.hdf")
# Scale Coordinates from millimeters to meters
t = scale(t, factor=0.001)
# Save Python/CGNS Tree t to HDF5 file
C.convertPyTree2File(t, "DPW-m.hdf")
```

In the general case, thanks to the standard CGNS data model and the Python/CGNS implementation, the tree is directly available from an HDF5/CGNS file or from memory and it can be exploited at the Python or at another code level (Fortran, C or C++).

- *Software components as multi-granularity assembly*

The very simple Python module or component presented in listing 2.1 could be a candidate for an assembly with any other modules to create new larger components. Always based on the same idea, one can build very large user applications from defined module sequences or workflows : this defines the multi-granularity concept. On figure 2.5, four different assembly examples are described :

- one for pre-treatment where two modules are connected : the first one is responsible for partitioning the meshes, that is to say for splitting meshes into subdomains distributed in order to achieve the best load balancing on the distributed HPC architecture, the second could be the Multigrid agglomerator for multigrid acceleration techniques used by the Finite Volume solvers. The first module responsible for the mesh partitioning called *PyPart* is able to split and merge topologically both structured and unstructured domains on distributed parallel execution. Because the splitting and merging algorithms are executed in a distributed parallel environment, this component is really powerful and today it is largely used by any other component on distributed parallel computation. This second component can be a solver or a component dedicated to the pre- or post-treatment.
- one for a solver canvas as *FastS* where 3 different modules are connected : at each time step, a component is responsible for join mesh partition treatment especially the MPI exchanges, followed by a component responsible of the Rhs and Lhs computation where asynchronous OpenMP techniques are implemented and one for the computation of the Flux Balance,
- one for the construction of the Immersed boundary solver based on the sequencing of the structured solver *FastS* or next with *NeoN-S* and the methods from *Connector* module responsible for the immersed boundary condition calculation.
- one for the co-processing, where a Python/CGNS tree communicates from a light client/server between a HPC architecture dedicated to the solver and a distant computer dedicated to post-treatment where a slice can be applied on the wind for example to display the static pressure repartition.

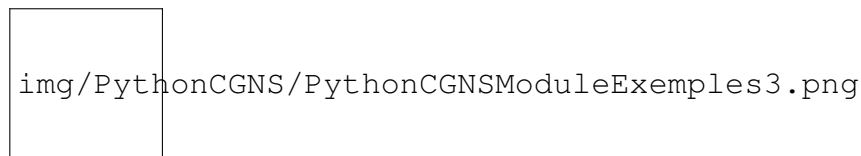


Fig. 2.5 –: Python/CGNS module assembly examples.

These 4 examples show a very small part of the possibilities offered by the architecture of an Open System, because thanks to this approach it is also possible to replace one module by another if it proves more relevant or more efficient, to factorize module as explained for the use of *PyPart* (the partitioning module) or to create of an infinity of user applications ranging from academic to industrial simulations.

- *Software components genericity in HPC context*

Because the target is HPC simulations, where the key point is the capability of handling very large configurations requiring very large resources within an affordable computation time, the core code can be also implemented in appropriated low-level programming language like C, C++, Fortran or Cython and even redefine its own data structure to best fit the required HPC context like the hybrid OpenMP/MPI architecture. Those different low-level programming languages can be easily wrapped to the Python language thanks to

FÉVRIER 2020

numpy/f2py for Fortran, Cython or directly through the Python API C for C or C++ languages and all these wrapping techniques are perfectly mastered at Onera especially in an HPC context.

As discussed in the previous section, to solve the contradiction existing between cross functionalities and HPC requirements, developments have been done to create code generator thanks to Python facilities called Python templating. The Python/template technique allows to generate easily understandable back-end Fortran optimized routines from templates where the loop implementation that can be optimized for a particular HPC architecture is separated from the "inside loop" physical/numerical implementation. Depending on the treated problem, for example Euler or Navier-Stokes, laminar or RANS turbulent problems or even choices with finer granularity, the code added inside the optimized template loops is then fully adapted to generated specialized and optimized code with the least possible calculation.

The Python/template technique advantages are numerous :

- grouping code from different loops to hardly optimize code on HPC architecture with memory band approach is very easy,
- generate Fortran back-end that everybody, numerical experts, HPC architects or software architects can easily read and understand,
- offer the possibility to generate code "on-the-fly" and create only a small number of routines that are compiled at the beginning of the computation and used dynamically by the Python code thanks to numpy/f2py package.

Thanks to this new approach, cache blocking solutions, implicit solvers, some boundary conditions or even turbulence models are explored to be shared by different solvers *FastS*, *FastP* in the first place and next with *Aghora* without loss of genericity and efficiency. In the short term, it will be possible to take advantage of the factorization work carried out on the 3 prototypes in view of the future generation development of the *NeoN-Solver*.

- *Onera's framework as Python software components packages*

Finally, because of the multi-granularity offered by the Open System, it is hard to deliver an exhaustive panel of all current available components developed at Onera, but the figure 2.6 tries to summarize the different relevant coarse-grain packages developed by Onera based on the Python/CGNS implementation which could interest Airbus for the Next Generation CFD software where some components dedicated to meshing, coupling, pre and post-treatment have been gathered in the *Cassiopee* suite and those dedicated to numerical solver prototypes (*Fast*, *Fast* and *Aghora*) gathered into the Navier-Stokes equation resolution packages. As explained in the introduction 1, in concordance with the CFD2030 Onera's project [3], these 3 prototypes have the vocation to become the basis of the architecture of the future Onera generation solver.

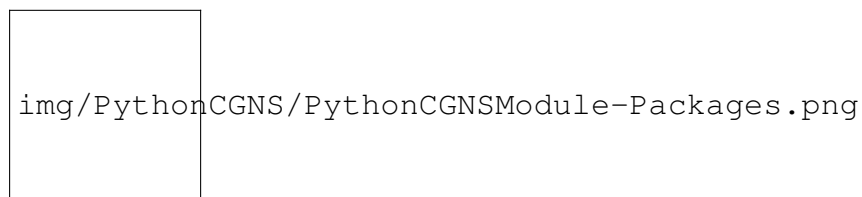


Fig. 2.6 –: Python/CGNS packages developed by Onera in the Open System Architecture based on the Python/CGNS implementation.

Inside those 2 suites, each module is a python module that can be produced and validated autonomously. Thanks to this hierarchical approach, development and validation take place at three levels : modules, suites and CFD platforms. This decentralized approach makes it possible to divide the work among several specialized teams, facilitates the maintenance and validation works, **whereas specialized product delivery can be realized to different customers**. Moreover this organization allows a better involvement of the teams and consequently a better appropriation of the final product.

All components belonging to the future Orion suite will be constructed following the same model. The main functions of a Foo Python module are located in the file Foo/PyTree.py. If the function requires an intensive computation phase, it will be redirected to the C layer, in which memory pointers are retrieved and temporary work array are created. Depending on the modules considered, the operations requiring CPU intensity are carried out either in the C layer or in a Fortran layer called from C in function of both the development teams preference and source sharing constraint between modules.

2.2. Simple exemples of end-user script

2.2.1. Flat plate computation with structured FastS module

In order to illustrate later in the document the interactions between the structures of the solvers and the end-user python script interface, a trivial script of flat plate computation is presented in the listing 2.2. In this example, two different Python modules are imported : *Cassiopee/Converter* to read and write data solutions and *Fast* module to resolve the Navier-Stokes equations. The set of nodes in the Fig. 2.3 are represented by a unique variable, *t*, which represents the python/CGNS tree of the simulation. Informations about grid connectivity, chimera, immersed boundary condition and wall function are stored in a different tree, *tc*. The general parameters of the calculation are specified in the dictionary **numb**, while those specific to a zone are defined in **numz**. Once defined, all these calculation parameters are placed in nodes of the tree *t* thanks to the *setNum2Zones* functions. A warmup function is called to compute the metrics linked to the grid contained in the tree and after, 500 iterations of the Navier-Stokes solver are computed.

Listing 2.2: example of Python script for Flat plate computation

```
import Converter.PyTree as C
import Fast.PyTree as Fast

t = C.convertFile2PyTree('FlatPlate.cgns')
tc = C.convertFile2PyTree('Connectivity.cgns')

numb = { 'temporal_scheme':'explicit', 'ss_iteration':1 }
numz = { 'cfl': 7, 'scheme':'ausm', 'slope':'minmod' }
#
Fast.__setNum2Zones(t, numz); Fast.__setNum2Base(t, numb)
#
(t, metrics) = Fast.warmup(t)
#
FastS.__applyBC(t, metrics)
#
```

FÉVRIER 2020

```

nit = 500; time = 0.
for it in xrange(nit):
    Fast._compute(t, metrics, it, tc)
    if (it%50 == 0):
        Fast.display_temporal_criteria(t, metrics, it)

C.ConvertPyTree2File(t, 'result.cgns')

```

2.2.2. Data access in the tree variable

This example illustrates how the variables stored in the Python tree can be accessed from the python or C layer. In the listing 2.3, the CGNS file described in Fig. 2.3 is first loaded in memory in a tree variable, `t`, and next all values of the Density variable are printed to the screen two times. In the first one only the Python Interface is used thanks to Internal functions of Converter module : the list of zones contained in `t` is retrieved using the function `getNodeFromType`, then the address of the numpy array associated with the density is obtained via the function `getNodeFromName`. In the second one the print command is realized from the C layer thanks to the C function `displayVar.cpp` illustrated in the listing 2.4. This function has 2 arguments : a python object linked to the zone node and a string, which represents the name of the variable to be printed. Functions of the Kcore library (`K_PYTREE :: ...` belonging to the *Cassiopee* kernel) are then used to extract the pointer of the array of the variable named `varname` from the Python object

Listing 2.3: example of Python script for Data access in the Python tree variable

```

# - access to Pytree variable -
import Converter.PyTree as C
import Converter.Internal as Internal
import FastS.PyTree as FastS
import numpy

t = C.convertFile2PyTree('FlatPlate.cgns')

zones = Internal.getNodesFromType2(t0, 'Zone_t')

var = 'Density'

#Python layer interface
for z in zones:
    Density = Internal.getNodeFromName2(z, var)[1]
    size= numpy.shape( Density )
    print 'In the zone', z[0], 'density is equal to:'
    for k in range( size[2] ):
        for j in range( size[1] ):
            for i in range( size[0] ):
                print 'i,j,k=', i, j, k, ' Density=', Density[i, j, k]

#C layer interface
for z in zones:
    FastS.fasts.displayVar( z, var)

```

Listing 2.4: example of Data access to the Python tree variable from the C layer

```

#include "fastS.h"
#include <string.h>
using namespace std;
using namespace K_FLD;

PyObject* K_FASTS::displayVar(PyObject* self, PyObject* args)
{
    PyObject* zone;
    char varname;

    if (!PyArg_ParseTuple(args, "Os", &zone, &varname)) return NULL;

    E_Float* iptro; vector<PyArrayObject*> hook;

    // Get size of structured zone
    size = K_PYTREE::getDimZone(zone);

    // Get pointer of the variable varname
    PyObject* sol_center; PyObject* t;
    sol_center = K_PYTREE::getNodeFromName1(zone, "FlowSolution#Centers");
    t = K_PYTREE::getNodeFromName1(sol_center, varname);
    iptro = K_PYTREE::getValueAF(t, hook);

    E_Int c= 0;
    for (E_Int k = 0; k < size[2]; k++)
    for (E_Int j = 0; j < size[1]; j++)
    for (E_Int i = 0; i < size[0]; i++)
    { printf("i,j,k=,%d,%d,%d,Density=%\n", i,j,k, iptro[ c ]) ; c+=1; }

    RELEASEHOOK(hook)
    Py_INCREF(Py_None);
    return Py_None;
}

```

2.3. Schematic decomposition of one Navier-Stokes solver iteration

In the listing 2.2, the Navier-Stokes equations are solved 500 times thanks to the Python function *Fast._compute*. This function, which is shown in the listing 2.5, is used both for implicit Gear or explicit Runge-Kutta time integration. This function relies on the possibility of creating sub-regions of the tree (sub-zone_list) to allow the use of consistant multi-rate explicit RK scheme with a local time step or to fine-tune the local convergence of the Newton process Gear. In the future, the coupling between explicit and implicit temporal integrations will also be made possible by this functionality. On each sub-iteration, a C function *fast_computePT*, is called to solve the equations of the fluid mechanics according to the numerical options chosen. Once the iterative process is completed, the solutions to the different instants are updated with a copy of pointer.

FÉVRIER 2020

Listing 2.5: *Fast* Python function for the temporal integration algorithm.

```

def _compute(t, metrics, nitrun, tc=None, graph=None):
    global HOOK

    dtloc = Internal.getNodeFromName1(t, '.Solver#dtloc') # noeud
    zones = Internal.getNodesFromType1(t, 'Zone_t')
    zones_tc = Internal.getNodesFromType1(tc, 'Zone_t')

    dtloc = Internal.getValue(dtloc) # tab numpy
    nitmax = int(dtloc[0])

    # RK step or Newton Gear subiteration
    for nstep in xrange(1, nitmax+1):
        # determination of the size zone for local Newton process
        # or constant RK scheme with space dependant time step
        hook1 = HOOK + fast.souszones_list(zones, metrics, HOOK, nitrun, nstep)
        nidom_loc = hook1[11]

        skip = 0
        if (hook1[13] == 0 and nstep == nitmax and nitmax > 3): skip = 1

        # Navier_stokes computation + BC + grid transfer
        if nidom_loc > 0 and skip == 0:
            fast._computePT(zones, metrics, nitrun, nstep, hook1, zones_tc, graph)

    # switch pointers between solutions at time N+1, N and N-1
    Fast.Internal.switchPointers__(zones)
    return None

```

The C function for solving the Navier-Stokes equations is described in listing 2.6. For the sake of clarity, the function reproduced in this document has been simplified by removing the parts for retrieving data from the Python CGNS tree and creating temporary work arrays. The architecture is made to be able to mix if necessary in the same simulation structured, polyhedral unstructured, or high-order (DG) solvers. The tools for computing on shared and distributed memory machines are the two most commonly used standards in the HPC environment : MPI library to link distributed memory nodes and Openmp library on shared memory nodes. The calculation of one sub-iteration is done in a parallel openmp zone, which is divided into 4 parts :

- Non blocking MPI Comm.
- Subzones computations, which do not depend on MPI Comm.
- MPI Comm. synchronisation and chimera/IBC treatments
- Subzones computations, which depend on MPI Comm.

Listing 2.6: *Fast* C layer for the computation of one sub-iteration.

```

#include "fast.h"
#include "param_solver.h"
#include <string.h>
#include <omp.h>
using namespace K_FLD;
using namespace std;

E_Int K_FAST::_computePT(PyObject* self, PyObject* args)

{
    PyObject* zones, metrics, nitrun, nstep, hook, zones_tc, graph; E_Int nitrun, nstep;

    if (!PyArg_ParseTuple(args, "OOiOOO", &zones, &metrics, &nitrun, &nstep, &hook, &zones_tc, &graph)) return NULL;

    //
    // data pointer access to the different variable as in listing. 5
    // Temporary work array creation
    //
    // Not shown for readability
    //

    //-- start zone // omp
#pragma omp parallel default(shared)
    {
        E_Int ithread = omp_get_thread_num() + 1;
        E_Int Nbre_thread_actif = omp_get_num_threads();

        // ----- MPI asynchronous transfert for N layers of primitives variables (Density, VelocityX,..., VelocityY, Temperature,...)
        Connector.connector.setInterpTransfer_iSend( zones, zones_tc, graph, buffer_s, etiquette);
        Connector.connector.setInterpTransfer_iRecv( zones, zones_tc, graph, buffer_r, etiquette);

        // ----- loop over list of subzones which does not depend on grid data transfert
        for (E_Int nd = 0; nd < nidom_internal; nd++)
        {
            if ( solver[nd] == 1 ) { FastS.fasts.BCZone( x[nd], ro[nd], .... );
            FastS.fasts.navier_stokes_struct_( x[nd], ro[nd], .... );}
            else if ( solver[nd] == 2 ) { FastP.fastp.BCZone( x[nd], ro[nd], .... );
            FastP.fastp.navier_stokes_unstruct_( x[nd], ro[nd], .... );}
            else if ( solver[nd] == 3 ) { FastDG.fastdg.BCZone( x[nd], ro[nd], .... );
            FastDG.fastdg.navier_stokes_dg_( x[nd], ro[nd], .... );}
        }

        // ----- MPI synchronisation
        Connector.connector.setInterpTransfer_synchro( zones, zones_tc, graph, buffer_r);

        // ----- loop over list of subzones which depend on MPI transfert
        for (E_Int nd = 0; nd < nidom_external; nd++)
        {
            if ( solver[nd] == 1 ) { FastS.fasts.BCZone( x[nd], ro[nd], .... );
            FastS.fasts.navier_stokes_struct_( x[nd], ro[nd], .... );}
            else if ( solver[nd] == 2 ) { FastP.fastp.BCZone( x[nd], ro[nd], .... );
            FastP.fastp.navier_stokes_unstruct_( x[nd], ro[nd], .... );}
            else if ( solver[nd] == 3 ) { FastDG.fastdg.BCZone( x[nd], ro[nd], .... );
            FastDG.fastdg.navier_stokes_dg_( x[nd], ro[nd], .... );}
        }
    }
    // end zone // omp
    return ierr;
}

```

For each subzone to be calculated, there is a tag in the Python/CGNS tree that identifies which solver to call. The boundary conditions are then applied to fill ghostcells whose depth depends on the numerical schemes. Then the Navier-Stokes solver is finally executed. The memory structure of the arguments of this function (incomplete list for clarity), is very important. Indeed, if FlowSimulator does not provide the same type of storage for these arguments, it would imply to have very different solvers for Safran or Airbus, or to go through a copy to the right format of the data that goes against HPC spirit.

2.4. Solver data structures

To solve the conservative Navier-Stokes equations, the choice was made to work with ghostcells. The number of layers of these cells depends, on the one hand, on the stencil of the scheme and on the possible use of overlapping techniques for implicit methods, and on the other hand, the choice of transferring only the basic variables (not the gradients) to simplify code sources and limit the cost of MPI transfers.

FÉVRIER 2020

2.4.1. *FastS data structures*

2.4.1.1. *State vector*

The choice has been made to work with the following primitive variables to integrate the conservative Navier-Stokes equations :

$$(Density, VelocityX, VelocityY, VelocityZ, Temperature)$$

This choice was guided by HPC considerations, since it allows to maximize the number of multiplications (cheap) and minimize the number of divisions (costly) in the workflow. For RANS computations, additional variables are added to the list above. Datas at a given iteration are stored in a 64Bits array of the following size : $rop(ndimdx * neq)$, where neq is the number of variables (5 + RANS vars) and $ndimdx$ the size needed to store one variable. In practice, $ndimdx = n_i \times n_j \times n_k + pad$, where n_i is the number of cells in the I direction of the structured grid (ghostcells included) and pad a padding variable introduced to limit cache eviction, which is processor and (n_i, n_j, n_k) dependant. For vectorization purpose, the variables are stored in a Structure of Array mode to guarantee contiguous access and limit scatter/gather operand :

$$rop(1 :: n_i \times n_j \times n_k) = Density; \quad rop(1 + ndimdx :: ndimdx + n_i \times n_j \times n_k) = VelocityX; \dots$$

Depending on temporal scheme, 2 or 3 time levels ($rop^{n+1}, rop^n, rop^{n-1}$) are needed to resolve the problem. If required by the solver, viscosity variable, chimera properties and Wall distance are also stored in three 64Bits arrays of size $ndimdx$. An array is also needed to store the RHS, $drom(ndimdx * neq)$.

2.4.1.2. *mesh coordinates and metrics*

Mesh coordinates, CoordinateX, CoordinateY and CoordinateZ are stored in 3 different 64Bits arrays of size $(n_i + 1) \times (n_j + 1) \times (n_k + 1)$. Informations about the Cell face normals and surface are stored in an array for each topological direction of the grid. For example, in the I direction,

$$ti(:, 1) = n_x \times S \quad ; \quad ti(:, 2) = n_y \times S \quad ; \quad ti(:, 3) = n_z \times S$$

, where S denotes the surface of the face. For HPC optimization reason, 4 solvers are present in *FastS* in order to take advantage of specific grid topology :

- 2d
- 3dcart : 3d cartesian with constant spacing (usefull for IBM approach)
- 3dhomo : curvilinear mesh in the (x,y) plane, linked with (I,J) direction and cartesian in the z direction, linked to K direction (LES/DNS)
- 3dfull : general 3d curvilinear mesh

In general, 3dhomo and 3dcart solvers are respectively 2 and 2.5 times faster than the 3dfull one. Before the computation of the metric, the nature of the grid is automatically analysed (2d, 3dcart, 3dhomo, 3dfull), a tag is added in the Python tree and the size of the different arrays ti,tj,tk is adjusted :

$$3dfull : size = (n_i + 1) \times (n_j + 1) \times (n_k + 1); \quad ti(size, 3), \quad tj(size, 3), \quad tk(size, 3) \quad (2.1)$$

$$3d_{homo} : size = (n_i + 1) \times (n_j + 1) \times (n_k + 1); ti(size, 2), tj(size, 2), tk(size, 1) \quad (2.2)$$

$$3d_{cart} : size = 1; ti(size, 1), tj(size, 1), tk(size, 1) \quad (2.3)$$

$$2d : size = (n_i + 1) \times (n_j + 1); ti(size, 2), tj(size, 2), tk(size, 0) \quad (2.4)$$

Following the same logic, the cell volume is stored in an 64Bits array : vol(size).

2.4.1.3. Zonal parameters

For HPC reason, zonal parameter are stored in a flat way in 2 arrays : Parameter_int for int*4 and Parameter_real for float64. These arrays are stored in python/CGNS node, located in each zone node, and named Parameter_int and Parameter_real, respectively.

The main integer4 datas are summarized here :

- Parameter_int[NIJK :NIJK+5]= (ni, nj, nk, ijfic, kfic) : total number of cells by direction (included ghostcells) and number of ghost layers in IJ and K directions, respectively.
- Parameter_int[NIJK_MTR :NIJK_MTR+5]= (ni_mtr, nj_mtr, nk_mtr, ijfic_mtr, kfic_mtr) : total number of faces by direction (included ghostcells) for metrics and number of ghost layers in IJ and K directions, respectively.
- Parameter_int[NIJK_XYZ :NIJK_XYZ+5]= (ni_xyz, nj_xyz, nk_xyz, ijfic_xyz, kfic_xyz) : total number of vertex by direction (included ghostcells) for grid coordinates and number of ghost layers in IJ and K directions, respectively.
- Parameter_int[IJKV :IJKV+3]= (iv, jv, kv) : total number of cells by direction (without ghostcells) for grid coordinates and number of ghost layers in IJ and K directions, respectively.
- Parameter_int[ITYPEZONE]= 0 :3dfull, 1 : 3dhomo; 2 : 3dcart; 3 : 2d.
- Parameter_int[IFLOW]= 1 :Euler, 2 : NSLaminar; 3 : NSTurbulent
- Parameter_int[ITYPCP]= 1 :implicit, 2 : explicit
- Parameter_int[LALE]= 0 :no motion, 1 : rigid motion
- Parameter_int[NEQ]= number of variable in state vector
- Parameter_int[NEQ_IJ]= number of components in metrics (face I and J)
- Parameter_int[NEQ_K]= number of component in metrics face K
- Parameter_int[NEQ_COE]= number of variable in coe array
- Parameter_int[NDIMDX]= shift to adress the different variable of the state vector. Usually, the parameter is equal to the total number of cells in the zone (included ghostcells). But, sometimes this number can be slightly larger for HPC efficiency reason.
- Parameter_int[NDIMDX_XYZ]= the total number of vertices in the zone (included ghostcells) which describe the mesh.
- Parameter_int[NDIMDX_MTR]= the total number of faces needed to describes the face normales (included ghostcells)
- Parameter_int[KFLUDOM]= 1 :ausmpred, 2 : senseur, 5 : Roe
- Parameter_int[SLOPE]= 1 : o1, 2 : o3, 3 : minmod
- Parameter_int[NSSITER]= number of subiteration (for RK or implicit)

The main float64 datas are summarized here :

- Parameter_real[STAREF :STAREF+7]= ($\gamma, C_v, \rho_\infty, P_\infty, V_\infty, T_\infty, Mach_\infty, \rho \tilde{v}_\infty$) : reference state
- Parameter_real[VISCO :VISCO+4]= ($Re, Prandtl, \mu_{sutherland}, T_{sutherland}, C_{sutherland}$)

FÉVRIER 2020

- `Parameter_real[PRANDT]`= Prandlt number
- `Parameter_real[PRANDT_TUR]`= Turbulent Prandlt number
- `Parameter_real[DTC]`= timestep

Moreover, information of Python/CGNS boundary condition nodes are also stored in `Parameter_real` and `Parameter_integer` arrays, without duplication of the information, in order to allow computation of all the BC at the C or Fortran level without return to the Python layer. This storage is based on the number $NbBC$ of nodes of type `BC_t` contained in the tree :

$$Parameter_integer[BC_NBBC : *] = (NbBC, \underbrace{Ptr_BC^1, \dots, Ptr_BC^{NbBC}}_{\text{address of BC in Parameter_int}}, \underbrace{Ptr_Data^1, \dots, Ptr_Data^{NbBC}}_{\text{address of BC data in Parameter_real}}, \underbrace{BC^1}_{\text{First BC}}, \dots, \underbrace{BC^{NbBC}}_{\text{Last BC}})$$

To access to the i th BC, you must point at the following pointer :

$$BC^{i*} = parameter_int + parameter_int[BC_NBBC + i]$$

$$BC^i = (type, idir, PointRange^*, Nb_data, size_data^*)$$

with :

- `type`= 0 (BCExtrapolate); 1 (BCFarfield);.....
- `idir`= 1 (imin); 2 (imax);.....; 6 (kmax)
- `Nb_data` : number of float64 dataArray stored in node `BC_t`
- `size_data*` = (`size_data`¹, ..., `size_data` ^{Nb_data}) : size of the float64 dataArray mentioned above

To retrieve the first float64 dataArray of the i th BC in `Parameter_real`, one must use the following pointer :

$$data^* = Parameter_real^* + Parameter_int[BC_NBBC + 1 + i + NbBC]$$

whereas the remaining dataArray (if any) can be accessed thanks to a shift based on `size_data*`.

2.4.2. FastP data structures

This solver is dedicated to generalized polyhedric meshes and Finite volume. In this section `Nvtx`, `Nfaces` and `Nelts` are respectively the number of vertex, faces and elements (ghostcells included). Unlike *FastS*, it requires information about connectivity, which are stored in 32-bit integer arrays :

- `NGON_ElementConnectivity` : give for each face, the number of vertices and their identity

$$(\underbrace{Nvtx^1, id_1^1, id_2^1, \dots, id_{Nvtx^1}^1}_{\text{face} = 1}, \dots, \underbrace{Nvtx^{Nfaces}, id_1^{Nfaces}, id_2^{Nfaces}, \dots, id_{Nvtx^{Nfaces}}^{Nfaces}}_{\text{face} = Nfaces})$$

— NGON_ParentElement : give for each face, the right and left elements

$$\underbrace{(id_L^{face=1}, id_L^{face=2}, \dots, id_L^{face=Nfaces})}_{\text{Left elements}}, \underbrace{(id_R^{face=1}, id_R^{face=2}, \dots, id_R^{face=Nfaces})}_{\text{Right elements}}$$

— NFACE_ElementConnectivity : give for each element, the number of faces and their identity

$$\underbrace{(Nface^1, id_1^1, id_2^1, \dots, id_{Nface^1}^1, \dots)}_{\text{elt} = 1}, \underbrace{(Nface^{Nelts}, id_1^{Nelts}, id_2^{Nelts}, \dots, id_{Nface^{Nelts}}^{Nelts})}_{\text{elt} = Nelts}$$

In order to share a maximum of "functions" between *FastS* and *FastP*, the choice has been made to work with the same primitive variables than those used in *FastS* to integrate the conservative Navier-Stokes equations :

$$(Density, \quad VelocityX, \quad VelocityY, \quad VelocityZ, \quad Temperature)$$

For the same reason data storage is also performed by an array of structure identical to that of *FastS*, in a 64Bits array of the following size : $rop(Nelts * neq)$, where neq is the number of variables (5 + RANS vars).

$$rop(1 :: Nelts) = Density \quad ; \quad rop(1 + Nelts :: 2 \times Nelts) = VelocityX; \dots$$

Depending on temporal scheme, 2 or 3 time levels ($rop^{n+1}, rop^n, rop^{n-1}$) are needed to resolve the problem. If required by the solver, viscosity variable, chimera properties and Wall distance are also stored in three 64Bits arrays of size $Nelts$. An array is also needed to store the RHS, $drodm(Nelts * neq)$. Mesh coordinates, CoordinateX, CoordinateY and CoordinateZ are stored in 3 different 64Bits arrays of size $Nvtx$. Informations about the Cell face normals, surface and center are stored in an 64Bits array $ti(Nfaces, 6)$:

$$ti(:, 1) = n_x \times S \quad ; \quad ti(:, 2) = n_y \times S \quad ; \quad ti(:, 3) = n_z \times S$$

$$ti(:, 4) = x_f \quad ; \quad ti(:, 5) = y_f \quad ; \quad ti(:, 6) = z_f$$

, where S denotes the surface of the face, and (x_f, y_f, z_f) are the coordinates of the face center.

The volume and center position of each element is stored in an 64Bits array $vol(Nelts, 4)$:

$$vol(:, 1) = volume \quad ; \quad vol(:, 2) = x_c \quad ; \quad vol(:, 3) = y_c \quad ; \quad vol(:, 4) = z_c$$

, where (x_c, y_c, z_c) are the coordinates of the element center.

Another important point concerns the order of arrangement of elements and faces with respect to the boundary conditions, grid connection, and ghostcells :

— The first elements are the internal elements, then come those of the first neighborhood, then those of the second, For instance, the density is stored as followed :

$$\underbrace{(\rho^1, \rho^2, \dots, \rho^{Nelts0})}_{\text{Interior elements}}, \underbrace{\rho^{Nelts0+1}, \rho^{Nelts0+2}, \dots, \rho^{Nelts0+Nelts1}}_{\text{first layer ghostcells}}, \dots$$

— For interface data, the first faces are the internal faces, then come those linked to grid connectivity, then those internal for the first layer of ghostcell, then those of boundary condition For instance, the x coordinate of face center is stored as followed for a two ghostlayers case :

$$\underbrace{(x_f^1, \dots, x_f^{Nface0})}_{\text{Interior}}, \underbrace{(x_f^{Nface0+1}, \dots, x_f^{Nface1})}_{\text{connectivity}}, \underbrace{(x_f^{Nface1+1}, \dots, x_f^{Nface2})}_{\text{Interior for first layer}}, \underbrace{(x_f^{Nface2+1}, \dots, x_f^{Nface3})}_{\text{BC for first layer}}, \underbrace{(x_f^{Nface3+1}, \dots, x_f^{Nface4})}_{\text{BC for internal element}}, \dots$$

FÉVRIER 2020

2.4.3. Connector : tc data structures

For HPC efficiency reason, the connectivity tree, tc, is stored in a specific way in 2 integer32 and float64 arrays, but tc still remains fully compatible with CGNS/Python.

2.4.3.1. Integer datas

A node, named `Parameter_int` of type `DataArray_t`, is stored at the top of the tree. This array of integer is structured in function of the number of Point to Point MPI communications, $NbP2P$.

2.4.3.1.1. Structure of `parameter_int` for explicit global time stepping scheme and implicit schemes

$$parameter_int = (NbP2P, NbcomIBC, destIBC^1, \dots, destIBC^{NbcomIBC}, NbcomID, destID^1, \dots, destID^{NbcomID}, \underbrace{Ptr_P2P^1, \dots, Ptr_P2P^{NbP2P}}_{\text{address of P2P exchange in Parameter_int}}, \underbrace{P2P^1}_{\text{First P2P exchange}}, \dots, \underbrace{P2P^{NbP2P}}_{\text{Last P2P exchange}})$$

$NbcomIBC$ is the number of Point 2 Point communications for IBC joins. This parameter is available at the following address :

$$NbcomIBC = parameter_int + parameter_int[1] \quad (2.5)$$

The parameters $destIBC^1, \dots, destIBC^{NbcomIBC}$ denote respectively the number of the first, ..., the $NbcomIBC^{th}$ MPI processus for the point to point communication. These parameters are obtained at the addresses :

$$destIBC^1, \dots, destIBC^{NbcomIBC} = parameter_int + parameter_int[1 + i] \quad (1 \leq i \leq NbcomIBC)$$

$NbcomID$ is the number of Point 2 Point communications for ID joins. To access this parameter, you must go to the following address :

$$NbcomID = parameter_int + parameter_int[2 + NbcomIBC], \quad (2.6)$$

where the parameter $NbcomIBC$ is available at the adress (2.5). The parameters $destID^1, \dots, destID^{NbcomID}$ denote the number of the first, ..., the $NbcomID^{th}$ MPI processus for the point to point communication. These parameters are available at the addresses :

$$destID^1, \dots, destID^{NbcomID} = parameter_int + parameter_int[2 + NbcomIBC + i] \quad (1 \leq i \leq NbcomID)$$

To access to the i th Point to Point exchange, you must point at the following pointer :

$$P2P^{i*} = parameter_int + parameter_int[2 + NbcomIBC + NbcomID + i]$$

,where $NbcomIBC$ and $NbcomID$ are obtained respectively at the addresses (2.5) and (2.6). For each P2P exchange, the data are structured as followed :

$$P2P^i = (dest, Nrac, Nrac_inst, timelevels, UnsteadyJoin^*, Nbpts^*, Nbpts_InterpD^*, Nbtype^*, IBC^*, ZoneRole^*, NozoneD^*, Ptr_Pointlist^*, Ptr_InterpolantsType^*, Ptr_InterpolantsDonor^*, Nrac, GridLocation^*, Nbpts_D^*, NozoneR^*, Ptr_PointlistDonor^*, Neq^*, Rotation^*, Skip^*, PointlistDonor^*, InterpolantsType^*, Pointlist^*)$$

with :

- $dest$, the number of the MPI processus for the point to point communication
- $Nrac$, the total number of joins between the current MPI processus and the $dest$ processus
- $Nrac_inst$, the total number of unsteady joins between the current MPI processus and the $dest$ processus
- $timelevels$, the total number of timesteps stored for unsteady joins
- $UnsteadyJoin^* = (\underbrace{No_begin^1, \dots, No_begin^{timelevels}}_{\text{first join for a given timelevel}}, \underbrace{No_end^1, \dots, No_end^{timelevels}}_{\text{last join for a given timelevel}})$.

For a given timelevel, unsteady joins are stored contiguously ; therefore a loop between No_begin^i and No_end^i allows to deal with all the joins of timelevel i .

- $Nbpts^* = (\underbrace{Nbpts^1, \dots, Nbpts^{Nrac}}_{\text{size of the PointList numpy}})$.

This array has the following pointer address :

$$Nbpts^* = P2P^{i*} + 4 + 2 \times timelevels$$

- $Nbpts_InterpD^* = (\underbrace{Nbpts_InterpD^1, \dots, Nbpts_InterpD^{Nrac}}_{\text{size of the InterpolantsDonor numpy}})$

The pointer of this array is : $Nbpts_InterpD^* = P2P^{i*} + 4 + 2 \times timelevels + Nrac$

- $Nbtype^* = (\underbrace{Nbtype^1, \dots, Nbtype^{Nrac}}_{\text{number of different interpolantsTypes}})$

The pointer of this array is :

$$Nbtype^* = P2P^{i*} + 4 + 2 \times timelevels + 2 \times Nrac$$

- $IBC^* = (\underbrace{IBC^1, \dots, IBC^{Nrac}}_{\text{binary value : 1= IBC join, 0= ID join}})$

The pointer of this array is :

$$IBC^* = P2P^{i*} + 4 + 2 \times timelevels + 3 \times Nrac$$

- $ZoneRole^* = (\underbrace{ZoneRole^1, \dots, ZoneRole^{Nrac}}_{\text{binary value : 1= Receiver, 0= Donor}})$

The pointer of this array is :

$$ZoneRole^* = P2P^{i*} + 4 + 2 \times timelevels + 4 \times Nrac$$

- $NozoneD^* = (\underbrace{NozoneD^1, \dots, NozoneD^{Nrac}}_{\text{Donor zone number in the tree list}})$

The pointer of this array is :

$$NozoneD^* = P2P^{i*} + 4 + 2 \times timelevels + 5 \times Nrac$$

- $Ptr_Pointlist^* = (\underbrace{Ptr_Pointlist^1, \dots, Ptr_Pointlist^{Nrac}}_{\text{pointer address of PointList numpy}})$

The pointer of this array is :

$$Ptr_Pointlist^* = P2P^{i*} + 4 + 2 \times timelevels + 6 \times Nrac$$

FÉVRIER 2020

$$— \text{Ptr_InterpolantsType}^* = \underbrace{(\text{Ptr_InterpolantsType}^1, \dots, \text{Ptr_InterpolantsType}^{N_{rac}})}_{\text{pointer address of InterpolantsType numpy}}$$

The pointer of this array is :

$$\text{Ptr_InterpolantsType}^* = P2P^{i*} + 4 + 2 \times \text{timelevels} + 7 \times N_{rac}$$

$$— \text{Ptr_InterpolantsDonor}^* = \underbrace{(\text{Ptr_InterpolantsDonor}^1, \dots, \text{Ptr_InterpolantsDonor}^{N_{rac}})}_{\text{pointer address of InterpolantsDonor numpy}}$$

The pointer of this array is :

$$\text{Ptr_InterpolantsDonor}^* = P2P^{i*} + 4 + 2 \times \text{timelevels} + 8 \times N_{rac}$$

— N_{rac} , the total number of joins between the current MPI processus and the *dest* processus. Data duplicated for contiguous send in MPI buffer. May change soon.

$$— \text{GridLocation}^* = \underbrace{(\text{GridLocation}^1, \dots, \text{GridLocation}^{N_{rac}})}_{\text{binary value : 1= vertex, 0= center}}$$

The pointer of this array is :

$$\text{GridLocation}^* = P2P^{i*} + 5 + 2 \times \text{timelevels} + 9 \times N_{rac}$$

$$— \text{Nbpts}_D^* = \underbrace{(\text{Nbpts}_D^1, \dots, \text{Nbpts}_D^{N_{rac}})}_{\text{size of the PointListDonor numpy}}$$

This array has the following pointer address :

$$\text{Nbpts}_D^* = P2P^{i*} + 5 + 2 \times \text{timelevels} + 10 \times N_{rac}$$

$$— \text{NozoneR}^* = \underbrace{(\text{NozoneR}^1, \dots, \text{NozoneR}^{N_{rac}})}_{\text{Receiver zone number in the tree list}}$$

The pointer of this array is :

$$\text{NozoneR}^* = P2P^{i*} + 5 + 2 \times \text{timelevels} + 11 \times N_{rac}$$

$$— \text{Ptr_PointlistD}^* = \underbrace{(\text{Ptr_PointlistD}^1, \dots, \text{Ptr_PointlistD}^{N_{rac}})}_{\text{pointer address of PointListDonor numpy}}$$

The pointer of this array is : $\text{Ptr_PointlistD}^* = P2P^{i*} + 5 + 2 \times \text{timelevels} + 12 \times N_{rac}$

$$— \text{NEQ}^* = \underbrace{(\text{NEQ}^1, \dots, \text{NEQ}^{N_{rac}})}_{\text{number of variables to be send}}$$

The pointer of this array is :

$$\text{NEQ}^* = P2P^{i*} + 5 + 2 \times \text{timelevels} + 13 \times N_{rac}$$

$$— \text{Rotation}^* = \underbrace{(\text{Rotation}^1, \dots, \text{Rotation}^{N_{rac}})}_{\text{binary value : 1= azimuthal periodicity , 0= no periodicity}}$$

The pointer of this array is :

$$\text{Rotation}^* = P2P^{i*} + 5 + 2 \times \text{timelevels} + 14 \times N_{rac}$$

$$— \text{Skip}^* = \underbrace{(\text{Skip}^1, \dots, \text{Skip}^{N_{rac}})}_{\text{binary value : -1= allways, i= active if nitrn==i}}$$

The pointer of this array is :

$\text{Skip}^* = P2P^{i*} + 5 + 2 \times \text{timelevels} + 15 \times N_{rac}$; It will be usefull for local unsteady timestepping.

$$— \text{PointlistDonor}^* = \underbrace{(\text{PointlistDonor}^1, \dots, \text{PointlistDonor}^{N_{rac}})}_{\text{PointListDonor numpy}}$$

The pointer of this array is :

$$\text{PointlistDonor}^* = \text{parameter_int} + \text{parameter_int}[\text{Ptr_PointlistD}^i]$$

$$— \text{InterpolantsType}^* = \underbrace{(\text{InterpolantsType}^1, \dots, \text{InterpolantsType}^{N_{rac}})}_{\text{InterpolantsType numpy}}$$

The pointer of this array is :

$$\text{InterpolantsType}^* = \text{parameter_int} + \text{parameter_int}[\text{Ptr_InterpolantsType}^i]$$

$$\text{--- } Pointlist^* = \underbrace{(Pointlist^1, \dots, Pointlist^{N_{rac}})}_{\text{PointList numpy}}$$

The pointer of this array is :

$$Pointlist^* = parameter_int + parameter_int[Ptr_Pointlist^i]$$

2.4.3.1.2. Structure of *parameter_int* for the explicit local time stepping scheme

$$parameter_int = (NbP2P, size_IBC, Ptr_comIBC^{it1}, \dots, Ptr_comIBC^{itf}, comIBC^{it1}, \dots, comIBC^{itf}, \\ size_ID, Ptr_comID^{it1}, \dots, Ptr_comID^{itf}, comID^{it1}, \dots, comID^{itf}, \\ \underbrace{Ptr_P2P^1, \dots, Ptr_P2P^{NbP2P}}_{\text{address of P2P exchange in Parameter_int}}, \underbrace{P2P^1}_{\text{First P2P exchange}}, \dots, \underbrace{P2P^{NbP2P}}_{\text{Last P2P exchange}})$$

The parameter *size_IBC* is the length of the following array :

$$(\underbrace{Ptr_comIBC^{it1}, \dots, Ptr_comIBC^{itf}}_{\text{adress of informations about P2P communication at each it.}}, \underbrace{comIBC^{it1}}_{\text{Informations about P2P comm. at it1}}, \dots, \underbrace{comIBC^{itf}}_{\text{Informations about P2P comm. at itf}})$$

The array which contains all informations about the Point 2 Point communications for IBC joins at iteration *i* ($1 \leq i \leq itf$), *comIBC^{iti}*, starts at the address :

$$comIBC^{iti*} = parameter_int + parameter_int[2 + Ptr_comIBC^{iti}], \quad (2.7)$$

whith *Ptr_comIBC^{iti}* available at the adress :

$$Ptr_comIBC^{iti} = parameter_int + parameter_int[1 + i]. \quad (2.8)$$

The array *comIBC^{iti}* is structured as follow :

$$comIBC^{iti} = (NbcomIBC^{iti}, dest^{1,iti}, \dots, dest^{NbcomIBC^{iti},iti}), \quad (2.9)$$

where :

- *NbcomIBC^{iti}* is the number of point to point communications for IBC joins at iteration *i*. It is available at the address : $NbcomIBC^{iti} = parameter_int + parameter_int[2 + Ptr_comIBC^{iti}]$.
- $dest^{k,iti}$ is the number of the k^{th} ($1 \leq k \leq NbcomIBC^{iti}$) MPI processus for point to point communication at iteration *i*. It is available at the address : $dest^{k,iti} = parameter_int + parameter_int[2 + Ptr_comIBC^{iti} + k]$.

FÉVRIER 2020

The parameter $size_ID$ is the length of the following array :

$$\left(\underbrace{Ptr_comID^{it1}, \dots, Ptr_comID^{itf}}_{\text{adress of informations about P2P communication at each it.}}, \underbrace{comID^{it1}}_{\text{Informations about P2P comm. at it1}}, \dots, \underbrace{comID^{itf}}_{\text{Informations about P2P comm. at itf}} \right)$$

The array which contains all informations about the point to point communications for ID joins at iteration i ($1 \leq i \leq itf$), $comID^{iti}$, starts at the address :

$$comID^{iti*} = parameter_int + parameter_int[2 + size_IBC + Ptr_comID^{iti}], \quad (2.10)$$

whith Ptr_comID^{iti} available at the address :

$$Ptr_comID^{iti} = parameter_int + parameter_int[2 + size_IBC + i]. \quad (2.11)$$

The array $comID^{iti}$ is structured as follow :

$$comID^{iti} = (NbcomID^{iti}, dest^{1,iti}, \dots, dest^{NbcomID^{iti},iti}), \quad (2.12)$$

where :

- $NbcomID^{iti}$ is the number of point to point communications for ID joins at iteration i . It is available at the address : $NbcomID^{iti} = parameter_int + parameter_int[2 + size_IBC + Ptr_comID^{iti}]$.
- $dest^{k,iti}$ is the number of the k^{th} ($1 \leq k \leq NbcomID^{iti}$) MPI processus for point to point communication at iteration i . It is available at the address : $dest^{k,iti} = parameter_int + parameter_int[2 + size_IBC + Ptr_comID^{iti} + k]$.

To access to the i th Point to Point exchange, you must point at the following pointer :

$$P2P^{i*} = parameter_int + parameter_int[2 + NbcomIBC + NbcomID + i]$$

,where $NbcomIBC$ and $NbcomID$ are obtained respectively at the adresses (2.5) and (2.6). For each P2P exchange, the data are structured as followed :

$$P2P^i = (dest, Nrac, Nrac_inst, timelevels, UnsteadyJoin^*, Nbpts^*, Nbpts_InterpD^*, Nbtype^*, IBC^*, ZoneRole^*, NozoneD^*, Ptr_Pointlist^*, Ptr_InterpolantsType^*, Ptr_InterpolantsDonor^*, Nrac, GridLocation^*, Nbpts_D^*, NozoneR^*, Ptr_PointlistDonor^*, Neq^*, Rotation^*, Skip^*, PointlistDonor^*, InterpolantsType^*, Pointlist^*, \underbrace{infosdtloc^1, \dots, infosdtloc^{Nrac}}_{\text{arrays of informations for local time stepping}})$$

with :

- *dest*, the number of the MPI processus for the point to point communication
- *Nrac*, the total number of joins between the current MPI processus and the *dest* processus
- *Nrac_inst*, the total number of unsteady joins between the current MPI processus and the *dest* processus
- *timelevels*, the total number of timesteps stored for unsteady joins
- $UnsteadyJoin^* = (\underbrace{No_begin^1, \dots, No_begin^{timelevels}}_{\text{first join for a given timelevel}}, \underbrace{No_end^1, \dots, No_end^{timelevels}}_{\text{last join for a given timelevel}})$.

For a given timelevel, unsteady joins are stored contiguously ; therefore a loop between No_begin^i and No_end^i allows to deal with all the joins of timelevel i .

- $Nbpts^* = (\underbrace{Nbpts^1, \dots, Nbpts^{Nrac}}_{\text{size of the PointList numpy}})$.

This array has the following pointer address :

$$Nbpts^* = P2P^{i*} + 4 + 2 \times timelevels$$

- $Nbpts_InterpD^* = (\underbrace{Nbpts_InterpD^1, \dots, Nbpts_InterpD^{Nrac}}_{\text{size of the InterpolantsDonor numpy}})$

The pointer of this array is : $Nbpts_InterpD^* = P2P^{i*} + 4 + 2 \times timelevels + Nrac$

- $Nbtype^* = (\underbrace{Nbtype^1, \dots, Nbtype^{Nrac}}_{\text{number of different interpolantsTypes}})$

The pointer of this array is :

$$Nbtype^* = P2P^{i*} + 4 + 2 \times timelevels + 2 \times Nrac$$

- $IBC^* = (\underbrace{IBC^1, \dots, IBC^{Nrac}}_{\text{binary value : 1= IBC join, 0= ID join}})$

The pointer of this array is :

$$IBC^* = P2P^{i*} + 4 + 2 \times timelevels + 3 \times Nrac$$

- $ZoneRole^* = (\underbrace{ZoneRole^1, \dots, ZoneRole^{Nrac}}_{\text{binary value : 1= Receiver, 0= Donor}})$

The pointer of this array is :

$$ZoneRole^* = P2P^{i*} + 4 + 2 \times timelevels + 4 \times Nrac$$

- $NozoneD^* = (\underbrace{NozoneD^1, \dots, NozoneD^{Nrac}}_{\text{Donor zone number in the tree list}})$

The pointer of this array is :

$$NozoneD^* = P2P^{i*} + 4 + 2 \times timelevels + 5 \times Nrac$$

- $Ptr_Pointlist^* = (\underbrace{Ptr_Pointlist^1, \dots, Ptr_Pointlist^{Nrac}}_{\text{pointer address of PointList numpy}})$

The pointer of this array is :

$$Ptr_Pointlist^* = P2P^{i*} + 4 + 2 \times timelevels + 6 \times Nrac$$

- $Ptr_InterpolantsType^* = (\underbrace{Ptr_InterpolantsType^1, \dots, Ptr_InterpolantsType^{Nrac}}_{\text{pointer address of InterpolantsType numpy}})$

The pointer of this array is :

$$Ptr_InterpolantsType^* = P2P^{i*} + 4 + 2 \times timelevels + 7 \times Nrac$$

- $Ptr_InterpolantsDonor^* = (\underbrace{Ptr_InterpolantsDonor^1, \dots, Ptr_InterpolantsDonor^{Nrac}}_{\text{pointer address of InterpolantsDonor numpy}})$

The pointer of this array is :

$$Ptr_InterpolantsDonor^* = P2P^{i*} + 4 + 2 \times timelevels + 8 \times Nrac$$

- *Nrac*, the total number of joins between the current MPI processus and the *dest* processus. Data

FÉVRIER 2020

— duplicated for contiguous send in MPI buffer. May change soon.

$$\text{— } GridLocation^* = \underbrace{(GridLocation^1, \dots, GridLocation^{N_{rac}})}_{\text{binary value : 1= vertex, 0= center}}$$

The pointer of this array is :

$$GridLocation^* = P2P^{i*} + 5 + 2 \times timelevels + 9 \times N_{rac}$$

$$\text{— } Nbpts_D^* = \underbrace{(Nbpts_D^1, \dots, Nbpts_D^{N_{rac}})}_{\text{size of the PointListDonor numpy}}$$

This array has the following pointer address :

$$Nbpts_D^* = P2P^{i*} + 5 + 2 \times timelevels + 10 \times N_{rac}$$

$$\text{— } NozoneR^* = \underbrace{(NozoneR^1, \dots, NozoneR^{N_{rac}})}_{\text{Receiver zone number in the tree list}}$$

The pointer of this array is :

$$NozoneR^* = P2P^{i*} + 5 + 2 \times timelevels + 11 \times N_{rac}$$

$$\text{— } Ptr_PointlistD^* = \underbrace{(Ptr_PointlistD^1, \dots, Ptr_PointlistD^{N_{rac}})}_{\text{pointer address of PointListDonor numpy}}$$

The pointer of this array is : $Ptr_PointlistD^* = P2P^{i*} + 5 + 2 \times timelevels + 12 \times N_{rac}$

$$\text{— } NEQ^* = \underbrace{(NEQ^1, \dots, NEQ^{N_{rac}})}_{\text{number of variables to be send}}$$

The pointer of this array is :

$$NEQ^* = P2P^{i*} + 5 + 2 \times timelevels + 13 \times N_{rac}$$

$$\text{— } Rotation^* = \underbrace{(Rotation^1, \dots, Rotation^{N_{rac}})}_{\text{binary value : 1= azimuthal periodicity , 0= no periodicity}}$$

The pointer of this array is :

$$Rotation^* = P2P^{i*} + 5 + 2 \times timelevels + 14 \times N_{rac}$$

$$\text{— } Skip^* = \underbrace{(Skip^1, \dots, Skip^{N_{rac}})}_{\text{binary value : -1= allways, i= active if nitrun==i}}$$

The pointer of this array is :

$$Skip^* = P2P^{i*} + 5 + 2 \times timelevels + 15 \times N_{rac}; \text{ It will be usefull for local unsteady timestepping.}$$

$$\text{— } PointlistDonor^* = \underbrace{(PointlistDonor^1, \dots, PointlistDonor^{N_{rac}})}_{\text{PointListDonor numpy}}$$

The pointer of this array is :

$$PointlistDonor^* = parameter_int + parameter_int[Ptr_PointlistD^i]$$

$$\text{— } InterpolantsType^* = \underbrace{(InterpolantsType^1, \dots, InterpolantsType^{N_{rac}})}_{\text{InterpolantsType numpy}}$$

The pointer of this array is :

$$InterpolantsType^* = parameter_int + parameter_int[Ptr_InterpolantsType^i]$$

$$\text{— } Pointlist^* = \underbrace{(Pointlist^1, \dots, Pointlist^{N_{rac}})}_{\text{PointList numpy}}$$

The pointer of this array is :

$$Pointlist^* = parameter_int + parameter_int[Ptr_Pointlist^i]$$

— $infosdtloc^{k*}$ is an array of size 27 that contains all informations to ensure a correct transition for local time stepping at join k .

The pointer of this array is :

$$infosdtloc^{k*} = P2P^{i*} + 5 + 2 \times timelevels + 16 \times N_{rac} + 27 \times k$$

This array is structured as follow :

$$infosdtloc^{k*} = (PtRange^{k*}, JoinDirection^k, PtRangeDonor^{k*}, JoinDirectionDonor^k, Transfo^{k*}, PtPivot^{k*}, Profondeur^k, Ratio^{k*}, LevelRcv^k, LevelDnr^k, StartStorage^k),$$

where :

- $PtRange^{k*}$ is an array of size 6 which contains the point ranges for the receiving zone ($imin, imax, jmin, jmax, kmin, kmax$).
- $JoinDirection^k$ is the join direction from the point of view of the receiving zone.
- $PtRangeDonor^{k*}$ is an array of size 6 which contains the point ranges for the donor zone ($imin, imax, jmin, jmax, kmin, kmax$).
- $JoinDirectionDonor^k$ is the join direction from the point of view of the donor zone.
- $Transfo^{k*}$ is an array of size 3 which describes the geometrical transformation between the axes (x, y, z) of the receiving zone and the axes (x, y, z) of the donor zone.
- $PtPivot^{k*}$ is an array of size 3 which is used to identify adjacent cells on both sides of the join k .
- $Ratio^{k*}$ is an array of size 3 which contains the ratio between the grid spacing of the receiving zone and the donor zone in each spatial direction.
- $LevelRcv^k$ is the time level of the receiving zone.
- $LevelDnr^k$ is the time level of the donor zone.
- $StartStorage^k$ is the adress to start the storage of all values for join k .

2.4.3.2. Float datas

A node, named `Parameter_real` of type `DataArray_t`, is also stored at the top of the tree. This array of `float64` is structured as followed :

$$parameter_real = (\underbrace{InterpolantsDonor^*, xc^*, \dots, zw^*, Density^*, Pressure^*, utau^*, y+^*, Rotation^*}_{\text{first join}}, \underbrace{\dots}_{\text{ith join}}, \underbrace{InterpolantsDonor^*, xc^*, \dots, zw^*, Density^*, Pressure^*, utau^*, y+^*, Rotation^*}_{\text{last join}})$$

For the i th join, the pointers of these arrays are recover as follow :

FÉVRIER 2020

```

— InterpolantsDonor* = Parameter_real + Parameter_int[Ptr_InterpolantsDonor[i]]
  with Ptr_InterpolantsDonor* =  $P2P^{i*} + 4 + 2 \times \text{timelevels} + 8 \times \text{Nrac}$ 
— xc* = Parameter_real + Parameter_int[Ptr_InterpolantsDonor[i]] + Nbpts_D[i]
— yc* = Parameter_real + Parameter_int[Ptr_InterpolantsDonor[i]] +  $2 \times \text{Nbpts\_D}[i]$ 
— ....
— xi* = Parameter_real + Parameter_int[Ptr_InterpolantsDonor[i]] +  $4 \times \text{Nbpts\_D}[i]$ 
— ....
— zw* = Parameter_real + Parameter_int[Ptr_InterpolantsDonor[i]] +  $9 \times \text{Nbpts\_D}[i]$ 
— Density* = Parameter_real + Parameter_int[Ptr_InterpolantsDonor[i]] +  $10 \times \text{Nbpts\_D}[i]$ 
— Pressure* = Parameter_real + Parameter_int[Ptr_InterpolantsDonor[i]] +  $11 \times \text{Nbpts\_D}[i]$ 
— utau* = Parameter_real + Parameter_int[Ptr_InterpolantsDonor[i]] +  $12 \times \text{Nbpts\_D}[i]$ 
— y+* = Parameter_real + Parameter_int[Ptr_InterpolantsDonor[i]] +  $13 \times \text{Nbpts\_D}[i]$ 
— rotation* = ( $\theta_x; \theta_y, \theta_z, c_x, c_y, c_z$ )
  The pointer of this array is :
  rotation* = Parameter_real + Parameter_int[Ptr_InterpolantsDonor[i]] +  $14 \times \text{Nbpts\_D}[i]$ 

```

FÉVRIER 2020

FÉVRIER 2020

3. HPC LAYER

Modern computing platforms are composed of several thousands of computing nodes. Each computing node is now able to deal with several multi-core sockets with several many-core accelerators (GPUs, MICs). On such heterogeneous systems, Non-Uniform Memory Access (NUMA) effects play a crucial role on the performance of the applications and must be taken into account in the design of algorithms for next generation software.

The choice is made to work as much as possible with standard tools to manage the HPC layer. This promotes both the portability of the solvers and the ability of a new developer to be quickly operational. The calculation on nodes with distributed memory uses functions of the *Cassiopee/Connector* module, based on MPI asynchrone point to point tranfers in order to allow compute and communication overlapping (see listing 2.6). This section only focuses on the HPC layer dedicated to the shared memory node. Indeed the MPI transfers are not really managed by the different solvers and it is relatively easy to obtain a good scalability at a cluster level. Moreover the impact of the hardware modernization on the evolution of the software is much more important for the software layers interacting with the shared memory nodes. Thus the ability to evolve the HPC layer dedicated to the shared memory node is undoubtedly one of the most critical points for the future competitiveness of an HPC software.

A good HPC layer must be able to :

- exploit as much as possible the increasing number of computing cores and vector computing units. This is achieved by using standard OpenMP ≥ 4.0 library (part of C/Fortran compiler) to distribute the work among the cores (threading) and activate efficient vectorization thanks to OpenMP directive applied to internal loop.
- overcome the difficulties associated with memory access (limited size of fast memory, non-uniform access bandwidth among the cores, ...). This is achieved by strongly factoring functions requiring loop over elements or faces, by introducing blocking techniques and by relying on the "first touch" policy of OS system to guarantee a good DRAM memory localization on Non-Uniform-Memory-Access node.

In an Intel ecosystem [4], performance analysis tools such as Trace Analyzer, Vtune-Amplifer, XE-Inspector and XE-Advisor are often used either to detect new performance bottlenecks or to validate new algorithmic adaptations. These analyses can contribute to the programming rules for the sake of code quality and performance.

3.1. factorization of a Navier-Stokes workflow

By factorization, we mean the minimization of the number of functions needed to solve the Navier-Stokes equations illustrated in Eqs. 3.1-3.2 at continuous and discret levels. For the sake of clarity this process is illustrated for an explicit time integration, since the implicit phase does not add any particular difficulty. This factorisation is done to improve the cache temporality of the workflow and minimize the size of temporary work arrays.

$$\frac{\partial Q(\mathbf{x}, t)}{\partial t} + \iint_{\partial\Omega(t)} \mathbf{F}(\mathbf{x}, t) \cdot \mathbf{n} d\Sigma(t) = \mathbf{S} \quad (3.1)$$

$$Q^{n+1} = Q^n + \frac{\Delta t}{\vartheta} \underbrace{\sum_{l=1}^{N_{face}} (F_{Euler}^l(Q^n) - F_{Viscous}^l(Q^n, \mu_t^n))}_{drod m} n + S \quad (3.2)$$

In Eq.3.2, S represents the source terms (RANS, acoustic,...), whereas F_{Euler} and $F_{Viscous}$ are the convective and diffusive fluxes. In *elsA*, a large number of functions (≥ 20) are called step by step to build all the terms of Eq.3.2. This number has been reduced to 4 calls of subroutine :

1. the viscosity μ_t is assigned to the corresponding array.
2. $\Delta t/vol$ stored in coe array
3. the source term, S, is assigned to the RHS array
4. the fluxes balance (viscous and convective) are added on the fly to the RHS array.
5. update of the solution

Listing 3.1: simplified version of *FastS* Navier-Stokes_struct subroutine.

```

subroutine navier_stokes_struct(x,y,z,cellN,rop,param_int,..)

implicit none
#include "FastS/param_solver.h"

INTEGER_E ind_coe(6),ind_grad(6),ind_sdm(6), param_int(0:*)
REAL_E rop(*),ti(*),tj(*),tk(*),vol(*),x(*),y(*),z(*)

ind_zone= index_range for interior element
! (1,100,1,100,1,100) for 100^3 grid

ind_grad= index_range for interior element + 1 layer of ghostcells
! (0,101,0,101,0,101) for 100^3 grid

ind_coe = index_range for interior element + 2 layer of ghostcells
! (-1,102, -1,102, -1,102) for 100^3 grid

ind_sdm = ind_zone
ind_rhs = ind_zone
ind_mjr = ind_zone

!Laminar/turbulent viscosity: step (i)
call invist( param_int, ind_grad, rop, xmut )

! Timestep/vol computation (ii)
call cptst3(ind_sdm, xmut,rop, cellN, coe, ti,tj,tk, vol)

!RHS init + source term S: step (iii)
call src_term(ind_sdm, ind_rhs, ind_grad, rop, xmut, drodm)

! euler and viscous fluxes computation and balance: step (iv)
call fluxAUSM_slp03_lamin_3dfull( ind_sdm, rop, drodm, xmut, ti)

! solution update: step (v)

```

FÉVRIER 2020

```

||      call core3ark3(param_int, ind_mjr, rop_tmp, rop, drodm, coe)
||
||      end

```

For unstructured grid, a step must be added to compute the gradients of the primitive variables before the viscosity. The step (iv) is in general responsible for $\approx 80\%$ of the overall simulation time (explicit case), (v) $\approx 10\%$, whereas (i)-(iii) are around $\approx 3 - 4\%$ each. The difficulty of this approach is that it is necessary to generate a large number of functions for step (iv) to take into account the different model and numerical options (Euler, RANS, convective scheme, slope limiter, ...). For each option there is only one piece of code for maintenance efficiency reason, and all the necessary functions are generated automatically (This point is explained in a paragraph below dedicated to code generation). A simplified version of *FastS.navier – stokes_struct* function, used in listing 2.6, is shown in listing 3.1 to illustrate the (i)-(v) steps.

3.2. Vectorization

Modern processors have few Single Instruction Multiple Data units per core that can perform operations by packet of 4 or 8 for the price of one for the internal loop. In theory the compiler is supposed to do the job, but often it is necessary to instrument the internal loop with a directive to help the compiler to generate an efficient assembly code. To illustrate how the vectorization is activated, the function that calculates the flux balance is presented in the listing 3.2 in a slightly simplified version for sake of clarity.

Listing 3.2: *FastS* : step (iv) for AUSM laminar computation.

```

subroutine fluAUSM_slp3_lamin_3dfull(ind_loop, rop, drodm,xmut...)
implicit none
#include "FastS/param_solver.h"

INTEGER_E ind_loop(6), param_int(0:*)

REAL_E xmut( param_int(NDIMDX) )
REAL_E rop( param_int(NDIMDX)      * param_int(NEQ)      )
REAL_E drodm( param_int(NDIMDX)     * param_int(NEQ)     )
#include "FastS/formule_param.h"
inci = 1
incj = param_int(NIJK)
inck = param_int(NIJK)*param_int(NIJK+1)

#include "pragma_align.for"

DO k = ind_loop(5), ind_loop(6)
  DO j = ind_loop(3), ind_loop(4)

#include      "loopI_begin.for"
      l0= 1 - inck
#include      "AUSM/3dfull/fluAUSM_slp3_3dfull_k.for"
#include      "fluvisq_3dfull_k.for"
#include      "assemble_drodm_plus_vec1.for"
      enddo

#include      "loopI_begin.for"
      l0= 1 - incj !on modifie rhs(j et j-1)
#include      "AUSM/3dfull/fluAUSM_slp3_3dfull_j.for"
#include      "fluvisq_3dfull_j.for"
#include      "assemble_drodm_plus_vec1.for"
      enddo

#include      "loopI_begin.for"
      l0= 1 - inci
#include      "AUSM/3dfull/fluAUSM_slp3_3dfull_i.for"
#include      "fluvisq_3dfull_i.for"
#include      "assemble_drodm_plus_vec1.for"
      enddo

      if(icorr.eq.1) then !flux at Imax interface
        i = ind_loop(2) + 1
        l = inddm( i, j, k)
        lt = indmtr( i, j, k)
#include      "AUSM/3dfull/fluAUSM_slp3_3dfull_i.for"
#include      "fluvisq_3dfull_i.for"
        ls = 1 -inci
#include      "flu_send_scater_vec1.for"
      endif !
      ENDDO !do j
      !Flux at Jmax interface omitted for sake of clarity

```

FÉVRIER 2020

```

||
||      !If(jcorr.eq.1)
||      ENDDO !do k
||      !Flux at Kmax interface omitted for sake of clarity
||      end

```

For structured mesh, one need to compute the fluxes on I, J and K interfaces. In order to improve cache reuse, the computation of these 3 fluxes are mixed for fixed values of k and j (external loops). For these values of the pair (j, k), we can distinguish a chain of 3 internal loops on the i variable to calculate the flux in the 3 directions. This internal loop, represented by an include, loopI_begin.for, in listing 3.3, allows to place directives for vectorization easily modifiable in the future (alignement information, dependency,...) For the time being, the construction of the most expensive flow functions (fluvisq_3dfull?, assemble_drodm, fluAUSM_slp3_3dfull_?) relies on the inclusion of pieces of code by the C PreProcessing tool, which automatically generates the final subroutine to be compiled. The OpenMP 4 standard allowing the declaration and the construction of vector functions, these pieces of code can eventually become true C or Fortran functions in a near future if this relatively new possibility of OpenMP proves to be effective.

Listing 3.3: LoopI_begin.for : include for vectorization of the internal loop

```

||
||      lij =      inddm( ind_loop(1) , j, k) -1
||      ltij = lij - indmtr(ind_loop(1) , j, k) +1
||      #ifdef _OPENMP4
||      !$OMP simd aligned(drodm,rop,xmut: CACHELINE)
||      #else
||      !DIR$ IVDEP
||      #endif
||      do l = lij+1, lij+1 + ind_loop(2) - ind_loop(1)
||
||          lt = l - ltij
||          lvo= lt

```

3.3. Cache Blocking technique

Even though the number of functions required to solve the equations has been limited to 5, the memory traffic can still be reduced by using a cache blocking technique [5]. For this purpose, the calculation area is subdivided into subparts, illustrated in Fig. 3.1 so that the data loaded in the cache L3 in step (i) can remain there until step (v). The size of the subpart is processor dependant and must be tuned during the porting phase on a new processor. A common size is worth about $size_{cache} = (\infty, 3, 3)$, since the internal loop (i) must be as large as possible for vectorization purpose. The listing 3.4 illustrates how cache blocking is introduced in HPC layer. This is done by means of 3 nested loops (icache,...), which make it possible to traverse all the sub-parts of Fig.3.1. A function is also added (index_loop_ssdome) to calculate the ranges of indices strictly necessary for the computation of steps (i)-(v) without adding redundant calculations.

Listing 3.4: Cache aware version of *FastS* Navier-Stokes_struct subroutine

```

subroutine navier_stokes_struct(x,y,z,cellN,rop,param_int,...)

implicit none
#include "FastS/param_solver.h"
  INTEGER_E ind_zone(6),ind_coe(6),ind_grad(6),ind_sdm(6),
&          param_int(0:*)
  REAL_E rop(*),ti(*),tj(*),tk(*),vol(*),x(*),y(*),z(*)

  ind_zone = index_range for interior element
  ! (1,100,1,100,1,100) for 100^3 grid

  do kcache = 1, max (1, ind_zone(6)/size_cache_loop(3) )
  do jcache = 1, max (1, ind_zone(4)/size_cache_loop(2) )
  do icache = 1, max (1, ind_zone(2)/size_cache_loop(1) )

    call indice_boucle_ssdom( icache, jcache, kcache, size_cache,
&                          ind_zone, ind_sdm, ind_coe,
&                          ind_grad, ind_rhs, ind_mjr)

    !Laminar/turbulent viscosity: step (i)
    call invist( param_int, ind_grad, rop, xmut )

    ! Timestep/vol computation (ii)
    call cptst3(ind_sdm, xmut,rop, cellN, coe, ti,tj,tk, vol)

    !RHS init + source term S: step (iii)
    call src_term(ind_sdm, ind_rhs, ind_grad, rop, xmut, drodm)

    ! euler and viscous fluxes computation and balance: step (iv)
    call fluxAUSM_slp03_lamin_3dfull( ind_sdm, rop, drodm, xmut, ti)

    ! solution update: step (v)
    call core3ark3(param_int, ind_mjr, rop_tmp, rop, drodm, coe)

  enddo
  enddo
  enddo
end

```

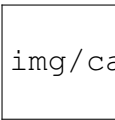

 img/cache.jpg

Fig. 3.1 –: Subzone partitioning for cache blocking

FÉVRIER 2020

3.4. Parallelisation

An hybrid parallelization technic has been retained. On a node with shared memory, the OpenMP library is used to generate threads and distribute the workload among the cores of the node, whereas MPI library are employed to transfer data between distributed memory nodes. The advantage of this hybrid approach is that it should have a better scalability for extreme HPC computation. The drawback relies on the fact that special efforts have to be done to obtain a good scalability at the node level. Moreover debugging an OpenMP code is always tedious even for an expert, which renders the use of such technic questionable in a large application due to maintenance and development issues.

3.4.1. Parallelisation by OpenMP threads

In order to exploit the growing number of cores in shared memory node, an OpenMP thread parallelization strategy has been adopted. For sake of efficiency and abstraction, a coarse grain approach is used. This means that the OpenMP directives are very localized in the code :

- the creation of the parallel zone is carried out very upstream in the C layer (see the directive `#pragma omp parallel` in Fig. 2.6).
- a function must be created to distribute the work among the thread
- a function must be created to synchronize threads, avoid races and flush their memory.

Each thread computes its own local subset of indexes corresponding to its position in the thread topology considered and the workload is split for a distribution among the cores of the node (typically, one thread per core). This process is illustrated in Fig. 3.2 : a color is attached to each thread and the work is distributed among 4×7 threads in the (J,K) plane. If the resulting workload is too large, cache blocking is also performed , see figure 3.3.

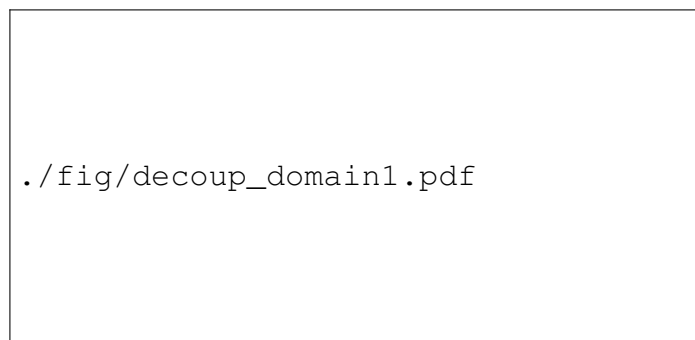


Fig. 3.2 –: zone decomposition strategy for threading.

Moreover, the cache blocking strategy provides a way to reduce synchronisation constraints at the border between 2 threads (see 3.3). Indeed, no synchronisation is needed unless the considered cache block shares a border between more than one thread. As a consequence, a minimum of 2 blocks per direction is imposed when more than 1 thread is found in that direction, the “independent” parts can then be computed safely. Each thread then computes the local indexes of the cells corresponding to its cache blocks. The flux balance (`fluAUSM_slp3_3dfull_`) is then computed for each cells. In order to prevent a double estimation of the fluxes at each interface between 2 threads, only 1 flux per direction is computed when the (i, j, k) cell is considered.

This flux corresponds to the one with the lowest index in that direction (*e.g.* the left and bottom flux for the j and k direction, respectively). The resulting distribution of faces treatments among threads is represented by the coloured edges in figure 3.3 (centre). The main drawback of this approach is the necessary synchronisation process to prevent race conditions on the balance result *drom*, which has been tedious to implement for a general topology (see the dependencies in Fig. 3.3, right).

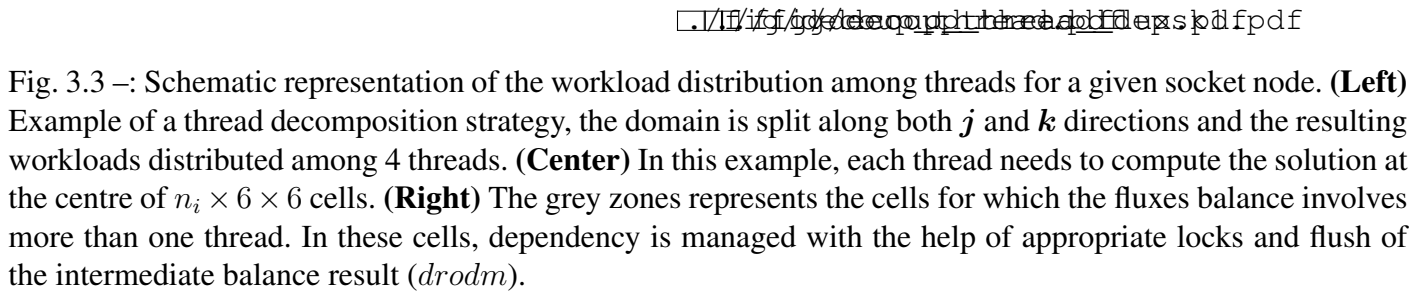


Fig. 3.3 –: Schematic representation of the workload distribution among threads for a given socket node. **(Left)** Example of a thread decomposition strategy, the domain is split along both j and k directions and the resulting workloads distributed among 4 threads. **(Center)** In this example, each thread needs to compute the solution at the centre of $n_i \times 6 \times 6$ cells. **(Right)** The grey zones represents the cells for which the fluxes balance involves more than one thread. In these cells, dependency is managed with the help of appropriate locks and flush of the intermediate balance result (*drom*).

The listing 3.5 illustrates how threading is introduced in the HPC layer. This is done by means of 2 functions :

- First the function `indice_loop_thread` distribute the global work (`ind_zone`) among the threads (`ind_zone_thread`)
- the `index_loop_ssdom` function now work with `ind_zone_thread` as input to calculate the ranges of indices strictly necessary for the computation of steps (i)-(v) without adding redundant calculations.
- the function `function` synchronizes threads and flush their memory if it is required by the (`icache`, `jcache`, `kcache`) indices and the threads topology.

FÉVRIER 2020

Listing 3.5: OMP cache aware version of *FastS* Navier-Stokes_struct subroutine

```

subroutine navier_stokes_struct(x,y,z,cellN,rop,param_int,...)
implicit none
#include "FastS/param_solver.h"
  INTEGER_E ind_zone(6),ind_coe(6),ind_grad(6),ind_sdm(6),
&      param_int(0:*)
  REAL_E rop(*),ti(*),tj(*),tk(*),vol(*),x(*),y(*),z(*)

  ithread          = omp_get_thread_num()
  Nbre_thread_activ = omp_get_num_threads()

  ind_zone = index_range for interior element
  ! (1,100,1,100,1,100) for 100^3 grid

  call indice_loop_thread( ithread, thread_parsock,      !In
&                          ind_zone,                    !In
&                          topo_thread, ind_zone_thread) !Out

  do kcache = 1, max (1, ind_zone(6)/size_cache_loop(3) )
  do jcache = 1, max (1, ind_zone(4)/size_cache_loop(2) )
  do icache = 1, max (1, ind_zone(2)/size_cache_loop(1) )

    call indice_boucle_ssdom( icache, jcache, kcache, size_cache,
&                          ind_zone_thread, ind_sdm , ind_coe,
&                          ind_grad, ind_rhs, ind_mjr)
    !OUT: ind_sdm (interior points)
    !OUT: ind_grad(interior points+1 layer)
    !OUT: ind_coe (interior points+2 layer)
    !OUT: .....
    call synchro_omp( ithread, topo_thread, lock,      !In
&                  icache, jcache, kcache,            !In
&                  OMP_WAIT,                          !In
&                  drodm, coe, xmut)                  !InOut (flush)

    !Laminar/turbulent viscosity: step (i)
    call invist( param_int, ind_grad, rop, xmut )

    ! Timestep/vol computation (ii)
    call cptst3(ind_sdm, xmut,rop, cellN, coe, ti,tj,tk, vol)

    !RHS init + source term S: step (iii)
    call src_term(ind_sdm, ind_rhs, ind_grad, rop, xmut, drodm)

    ! euler and viscous fluxes computation and balance: step (iv)
    call fluxAUSM_slp03lamin_3dfull( ind_sdm, rop, drodm, xmut, ti)

    ! solution update: step (v)
    call core3ark3(param_int, ind_mjr, rop_tmp, rop, drodm, coe)

    call synchro_omp( ithread, topo_thread, lock,      !In
&                  icache, jcache, kcache,            !In
&                  OMP_GO,                            !In

```

```

||
||      &                                drodm, coe, xmut)                !InOut (flush)
||      enddo
||      enddo
||      enddo
||      end

```

Finally, more complex work distribution has been implemented in order to improve L3 cache sharing between threads, as those presented in Fig. 3.4. The only thing to do is to adapt the thread synchronization (`synchro_omp`), index calculation (`index_loop_ssdome`) and work distribution (`index_loop_thread`) functions to a new scenario. These 3 functions in addition with the templated function presented in the listing 3.6 constitute the key elements to easily adapt the HPC layer to the future evolution of the hardware. With this approach, the developer of new functionalities (`foo1`, `foo2`...) must just provide the working index range he wants to parallelize, and then provide the functions he develops the index range they need (`interface`, `elements`, `neighborhoods`, etc.). ..).

The presentation of the HPC layer is based on the *FastS* solver. For *FastP*, the HPC layer strictly follows the same logic with its own thread synchronization (`synchro_omp`), index calculation (`index_loop_ssdome`) and work distribution (`index_loop_thread`) functions. For this, the internal faces are arranged so that it forms contiguous partitions which are associated with one thread as in Fig. 3.5. For *Aghora*, the parallel strategy relies mainly on a classic domain decomposition method using Message Passing Interface paradigm. Non-blocking synchronous send mode overlaps communications with calculations. In a more prospective way, OpenMP strategies are regularly evaluated with fine-grained or coarse-grained approaches. But also a parallel task-based strategy that relies on the StarPU library [7]. The goal is to estimate potential benefits of this runtime library for high-order CFD calculations in terms of performance (execution time, parallel efficiency) and adequation to hardware. A task is defined by a computational kernel with data dependencies and data privileges (read, write, read-write). An iterative time integration scheme can then be translated into a graph of tasks. Parallel tasks will be generated over all available computing supports (CPUs, GPUs, MICs) during the execution depending on scheduling policies. StarPU threads can be combined with MPI and OpenMP paradigms.

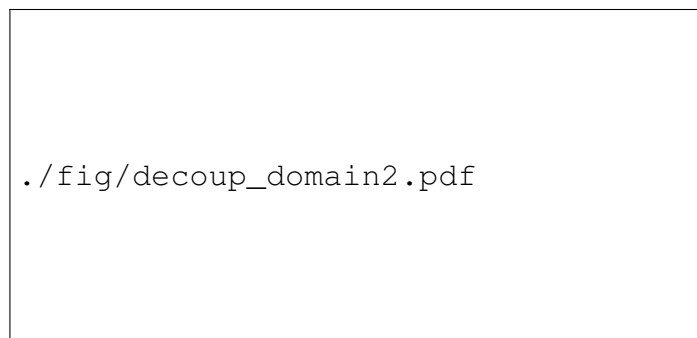


Fig. 3.4 –: Zone decomposition strategy for friendly L3 cache threading.

FÉVRIER 2020

Listing 3.6: HPC Template for a foo function with center and interface dependance.

```

subroutine foo(x,y,z,cellN,rop,param_int,...)
implicit none
#include "FastS/param_solver.h"
INTEGER_E ind_zone(6),ind_coe(6),ind_grad(6),ind_sdm(6),
&          param_int(0:*)
REAL_E rop(*),ti(*),tj(*),tk(*),vol(*),x(*),y(*),z(*)

ithread          = omp_get_thread_num()
Nbre_thread_activ = omp_get_num_threads()

ind_zone = index range of the work to be done

#include "FastS/HPC_LAYER/WORK_DISTRIBUTION_BEGIN.for"
#include "FastS/HPC_LAYER/LOOP_CACHE_BEGIN.for"

#include "FastS/HPC_LAYER/INDICE_RANGE.for"
!OUT: ind_sdm (interior points)
!OUT: ind_grad(interior points+1 layer)
!OUT: ind_coe (interior points+2 layer)
!OUT: .....
#include "FastS/HPC_LAYER/SYNCHRO_WAIT.for"

! step (i)
call foo1( ind_grad, param_int, rop, xmut )

!step (ii)
call foo2(ind_coe, xmut,rop, cellN, coe, ti,tj,tk, vol)

call ...
#include "FastS/HPC_LAYER/SYNCHRO_GO.for"
#include "FastS/HPC_LAYER/LOOP_CACHE_END.for"
#include "FastS/HPC_LAYER/WORK_DISTRIBUTION_END.for"
end

```

3.4.2. Parallelisation by MPI

To compute on distributed memory cluster, the MPI library has been retained. Call to the MPI functions can be made from the python layer thanks to the mpi4py module. For the present Fast solver, these calls to mpi functions are realized in the Connector module to allow exchange data between different stuctured/unstructured zones. The advantage of mpi4py relays on the fact that python type data (Dictionnary, list, numpy,...) can be directly tranferred for the programmer very conveniently. The main drawback of the mpi4py module concerns the efficiency. In a CFD application it is almost impossible to introduce overlapping between data transfer and CFD computation if the MPI call are done at the Python level. Moreover it has been shown recently that MPI transfer can be twice longer if done with MPI4py rather than with usual call to MPI Point to Point Communications realized at the C or Fortran level. To prevent this problem of efficiency, a library was developed by X. Juvigny to allow transfer of mixed data (integer, real, character) at the C level in order to make a compromise between the agility of MPI4Py and the efficiency of usual MPI Point to Point transfer functions. Non-blocking



Fig. 3.5 –: Illustration of cache aware work partitioning on unstructured grid (1 color= 1 thread).

transfert functions like `issend` or `irecv`, has been retained.

3.5. Automatic code generation

For maintenance efficiency reason and ease of code evolution, a large part of the source code must be generated automatically. Important work was undertaken last year to automatically share, using the Python templating technique, sequences of instructions between and inside the various Navier-stokes modules. For the time being, this sharing work mainly concerns *FastS* and *FastP* modules for the calculation of the 5 steps of the listing 3.1. In the near future, this work of sharing (when meaningful) will be extended to the *Aghora* solver, as well as to the management of implicit phases and boundary conditions. Concerning the adjoint method, a technique of automatic differentiation in tangent mode has been initiated in *FastS* during the project thanks to the Inria Tappenade library. This task is presently investigated more deeply in the PRF ODYSSEY. As long as the source code is c and/or Fortran, AD with Tappenade should be relatively straightforward.

3.5.1. Source sharing within a module : flux assembly and balance function

As mentioned in the previous section, the number of functions necessary to integrate Navier-Stokes equation has been reduced to five (*FastS*) or seven (*FastP*). To achieve that, the individual fluxes (viscous and convective) are added on the fly to the RHS array, which implies to generate a large number of functions to take into account the different model and numerical options (Euler, RANS, convective scheme, slope limiter...).

4 steps are necessary to generate this hotspot subroutine (80% of the total CPU time for explicit computation) :

FÉVRIER 2020

1. design a template compatible with all convective flux vector and difference splitting scheme targeted to compute the fluxes and the balance : **template_fluxAndbalance.for** in the listing 3.7. For HPC efficiency, this template can be tuned for each type of processor : for instance, viscous and convective fluxes can be evaluated in the same loop (as in listing 3.7) or in two different ones depending on the number of vectorial registers or the characteristics of the cache memory.
2. For each convective flux (Roe, AUSM, Jameson,...) a directory is created and templates are created to compute the flux : one for the 5 Euler equations **fluFaceEuler.for** illustrated in listing 3.8, and one more for each additional RANS model (for instance **fluFaceSA.for** for Spalart-Allmaras illustrated in listing 3.8 (right)). These functions are written for only one direction of a structured block, I, and the functions necessary in the 3 directions **fluFace_i**, **fluFace_j**, **fluFace_k** are generated automatically during the step (iv) described just below. In listing 3.8 for the *FastS* module, **TypeMesh** can be associated to a list of options [3dfull, 3dhomo, 3dcart,2d], which allow to generate the routines of the 4 solvers mentioned in section 2.2.1. **TypeSlope** represents the scheme employed to build the right and left slope [Minmod, 3rdorder, 2ndorder, 1storder,...] and **TypeMotion** determines if ALE is activated or not.
3. Create the common building blocks to handle in listing 3.8 includes linked to **TypeMesh**, **TypeSlope**, **TypeMotion**, viscous contributions (**fluViscLaminar**,...), right and left states reconstruction,..., which are shared by different numerical schemes.
4. Create a Python script, **generate_flux.py**, to generate all the back-end subroutines, the caller subroutines and the associated "makefile" thanks to Python templating. This generator, presented succinctly in listing 3.10, is applied to each type of flux (Roe, AUSM,...). A python dictionary is defined for each flux, which allow to generate very easily simplified version of the solver. For instance, if the model variable is restrained to "SA" for AUSM entry of the dictionary, then only the Spalart-Almaras routines will be generated (No Euler, No Laminar). This simple example illustrates Python's ability to generate dedicated solvers for a specific industry partner that do not contain all the calculation options, but only a subset. Examples of functions generated automatically from this script are presented in Fig.3.6 for Euler simulation with Roe scheme on 2D grid and Spalart-Almaras simulation with ALE and Ausm scheme on 3d grid.

Listing 3.7: Fortran template for flux and balance computation in *FastS*.

```

subroutine template_fluxAndbalance(ind_loop, rop, drodm,...)
implicit none
#include "FastS/param_solver.h"
INTEGER_E ind_loop(6), param_int(0:*)
REAL_E xmut( param_int(NDIMDX) )
REAL_E rop( param_int(NDIMDX)      * param_int(NEQ)      )
REAL_E drodm( param_int(NDIMDX)    * param_int(NEQ)    )
#include "FastS/formule_param.h"
inci = 1
incj = param_int(NIJK)
inck = param_int(NIJK)*param_int(NIJK+1)
#include "pragma_align.for"
DO k = ind_loop(5), ind_loop(6)
DO j = ind_loop(3), ind_loop(4)

#include      "loopI_begin.for"
l0= 1 - inck
#include      "FLUX_CONV/fluFaceEuler_k.for"
#include      "FLUX_CONV/fluFaceRans_k.for"
#include      "fluViscLaminar_k.for"
#include      "fluViscRans_k.for"
#include      "assemble_drodm_plus_vec1.for"
enddo

#include      "loopI_begin.for"
l0= 1 - incj
#include      "FLUX_CONV/fluFaceEuler_j.for"
#include      "FLUX_CONV/fluFaceRans_j.for"
#include      "fluViscLaminar_j.for"
#include      "fluViscRans_j.for"
#include      "assemble_drodm_plus_vec1.for"
enddo

#include      "loopI_begin.for"
l0= 1 - inci
#include      "FLUX_CONV/fluFaceEuler_i.for"
#include      "FLUX_CONV/fluFaceRans_i.for"
#include      "fluViscLaminar_i.for"
#include      "fluViscRans_i.for"
#include      "assemble_drodm_plus_vec1.for"
enddo
if(icorr.eq.1) then !flux manquant en I
i = ind_loop(2) + 1
l = inddm( i, j, k)
lt = indmtr( i, j, k)
#include      "FLUX_CONV/fluFaceEuler_i.for"
#include      "FLUX_CONV/fluFaceRans_i.for"
#include      "fluViscLaminar_i.for"
#include      "fluViscRans_i.for"
ls = 1 -inci
#include      "flu_send_scater_vec1.for"
endif !
ENDDO !do j

```

FÉVRIER 2020

```
! Flux at Jmax interface omitted for sake of clarity
ENDDO !do k
! Flux at Kmax interface omitted for sake of clarity
end
```

Listing 3.8: Fortran templates for AUSM flux include used in listing 3.7 for Euler Eqs (fluFaceEuler.for).

```

c.....Metric
#include "Normale/normale_TypeMesh_i.for"
    nm = 1 - inci
    nm2 = 1 -2*inci
    np = 1 + inci
! Right and left slope estimation
    vslp= v1
#include "Slope/TypeSlope_var.for"
    qm1 = qm
    qp1 = qp
    vslp= v2
#include "Slope/TypeSlope_var.for"
    qm2 = qm
    qp2 = qp
    vslp= v3
#include "Slope/TypeSlope_var.for"
    qm3 = qm
    qp3 = qp
    vslp= v4 !3D only
#include "Slope/TypeSlope_var.for" !3D only
    qm4 = qm !3D only
    qp4 = qp !3D only
    vslp= v5
#include "Slope/TypeSlope_var.for"
    qm5 = qm
    qp5 = qp
!Right and left state
#include "etat_GD.for"

!velocity at cell interface
#include "Vit_ent/qn_TypeMotion_TypeMesh_i.for"
    !Ausm dissipation
    c = rgp*gam*rop(1 +v5) !c^2
    sound = sqrt(qn1*qn1 / c)
    tam = c3*son+si
    tam1 = max(0.,tam)*c2
    !Numerical velocity at interface
    u =0.25*(qn1+qn2)-tam1*(p2-p1)
    !Numerical velocity for dissipation term
    tdu = max(abs(u),c1*si!)

#include "Vit_ent/fluvector_TypeMotion_TypeMesh_i.for"

```

Listing 3.9: Fortran templates for SA flux include used in listing 3.7 for RANS Eqs (fluFaceSA.for).

```

    vslp = v6
#include "Slope/TypeSlope_var.for"
    qp6 = qp*qp1
    qm6 = qm*qm1

    flu6 = u*(qm6+qp6) -tdu*(qm6 - qp6)

```

FÉVRIER 2020

Listing 3.10: Python script for code generation of flux and balance computation in *FastS*.

```

import sys

# python generate_flu.py fluxFolderName
rep = sys.argv[1]

dico= {}
dico["JAMESON"]={'name':'flu jameson', 'model':['lamin','SA','euler'], 'TypeMotion': ['','ale']}
dico["SENSOR"] ={'name':'flusenseur', 'model':['lamin','SA','euler'], 'TypeMotion': ['','ale']}
dico["AUSM"]     ={'name':'fluausm', 'model':['lamin','SA','euler'], 'TypeMotion': ['','ale']}
dico["ROE"]      ={'name':'fluroe', 'model':['lamin','SA','euler'], 'TypeMotion': ['','ale']}

dico["JAMESON"]={'TypeMesh':['3dfull','3dhomo','3dcart','2d'], 'TypeSlope':['o1']}
dico["SENSOR"] ={'TypeMesh':['3dfull','3dhomo','3dcart','2d'], 'TypeSlope':['o3']}
dico["AUSM"]     ={'TypeMesh':['3dfull','3dhomo','3dcart','2d'], 'TypeSlope':['o3','o1']}
dico["ROE"]      ={'TypeMesh':['3dfull','3dhomo','3dcart','2d'], 'TypeSlope':['minmod','o1','o2']}

Models      = dico[ rep ]['model']
TypeMotion  = dico[ rep ]['TypeMotion']
TypeMesh    = dico[ rep ]['TypeMesh']
TypeSlope   = dico[ rep ]['TypeSlope']
flux        = dico[ rep ]['name']

for motion in TypeMotion:
    for model in Models:
        for slope in TypeSlope:
            for mesh in TypeMesh:

                # input file
                f_in = open('template_FluxAndBalance.for','r')

                f_out = rep+'/'+mesh+'/'+flux+motion+model+'_'+slope+'_'+mesh+'.for'
                fout = open(f_out,"w")

                # Stuff to do
                # ....

                # for each option combination:
                # generate fortran Include
                # generate Fortran subroutine
                # modify makefile
                # modify flux selection function

                fout.close()

```

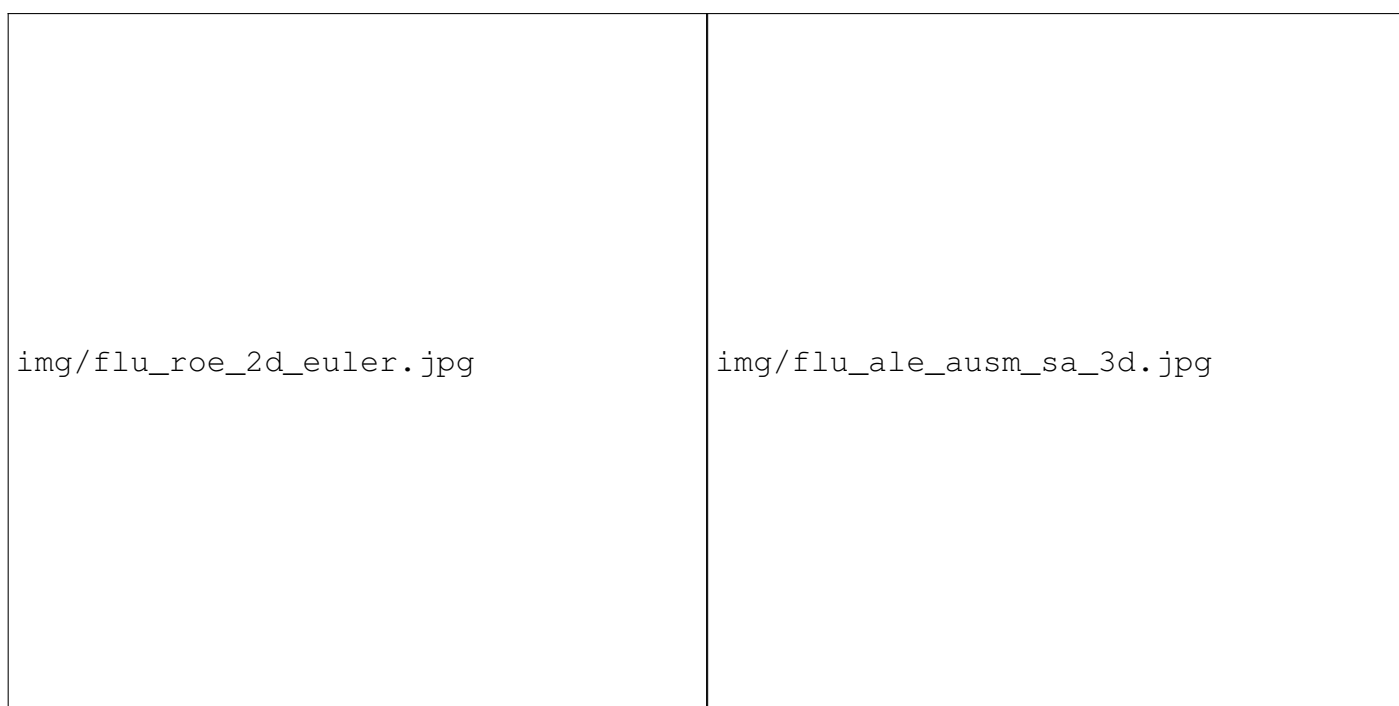


Fig. 3.6 –: Exemples of automatic function generated by Python templating (generate_flux.py). 2D Euler Eqs. with Roe scheme and minmod limiter (left). 3D ALE Spalart-Allmaras Eqs. with AUSM scheme and third order limiter (right)

FÉVRIER 2020

For a researcher involved in the development of numerical schemes, turbulence models or implicit phase, it is not necessary to know perfectly the process of Python templating. Indeed, the developer can make his own developments in a conventional way (valid for a limited number of options for example), whereas the integration of the method in the general workflow can be done in a second time by a person with a greater experience and knowledge of Python templates.

3.5.2. *Source sharing between Navier-Stokes modules : FastS, FastP, ...*

The structure of the *FastS*, *FastP*, ... solvers follows the same logic. This is shown in Fig.3.7 for *FastS* and *FastP*. Formally the functions differ by the addition of 2 additional steps in *FastP* to calculate the gradients at the center of the domains and in the ghostcells. For the "common" part (viscosity calculation, source term, solution update, ...), these functions are stored in the *Fast* module, dedicated to sharing "functions" between *FastS*, *FastP* (and *Aghora* in a near future). Then, thanks to a Python generator shown in the listing 3.11, these functions are duplicated and modified automatically in each module where their presence is necessary. This processus is illustrated in Fig. 3.8 for the viscosity computation.

img/comp_navier.jpg

Fig. 3.7 –: Comparaison of *FastS* and *FastP* routines for NavierStokes integration

FÉVRIER 2020

Listing 3.11: Python script for code generation of shared *FastS* and *FastP* routines involved in step (i)(ii)(iii)(v) of Fig.3.7

```

modules=['FastS', 'FastP']
files = []

files.append("Compute/invist.for")
files.append("Compute/cptst3.for")
files.append("Compute/core3ark3.for")
files.append("Compute/src_term.for")

for i in files:
    f = open(i, 'r')                                # ouvrir le fichier input
    lines = f.readlines()
    #loop solver
    for solver in modules:

        fout = '../..' + solver + '/' + solver + '/' + i
        print 'generation fichier:', fout, 'du module', solver

        fo = open(fout, "w")                        # ouvrir le fichier de sortie

        for ligne in lines:
            if solver == 'FastP':
                if 'INTEGER_E' not in ligne:
                    ligne=ligne.replace("Fast", "FastP").replace("ind_loop(3)", "1").replace
                        ("ind_loop(4)", "1").replace("ind_loop(5)", "1").replace("ind_loop(6)"
                        , "1")
                    ligne=ligne.replace("vol( param_int(NDIMDX_MTR) )", "vol( param_int(
                        NELTS) )")
                    ligne=ligne.replace("NDIMDX_MTR" , "NFACTES")
                    ligne=ligne.replace("NDIMDX_VENT", "NFACTES")
                    ligne=ligne.replace("NDIMDX" , "NELTS")
                    fo.write(ligne)
                else:
                    ligne=ligne.replace("Fast", "FastP")
                    ligne=ligne.replace("ind_loop(3)", "1")
                    ligne=ligne.replace("ind_loop(4)", "1")
                    ligne=ligne.replace("ind_loop(5)", "1")
                    fo.write(ligne)

            else:
                fo.write(ligne.replace("Fast", "FastS"))

        fo.close()                                # fermer le fichier output

f.close()                                        # fermer le fichier input

```



Fig. 3.8 –: Automatic generation of the subroutine involved in the laminar viscosity computation in *FastS* (left) and *fastP* (right) obtained from a single source code with the script 3.11

The sharing between modules is also realized for the flux and balance computation : the listing 3.12 shows the template used in *FastP* to generate all the combination of numerical and model parameter. This templates is very closed to those used in *FastS* (see listing 3.7). Indeed only the *face_i* contributions and internal loop has been kept, whereas information about right and left elements has been added.

FÉVRIER 2020

Listing 3.12: Fortran template for flux and balance computation in *FastP*.

```

subroutine template_fluxAndbalance(ind_loop, rop, drodm,...)
implicit none
#include "FastP/param_solver.h"
INTEGER_E ind_loop(6), param_int(0:*)
REAL_E xmut( param_int(NELTS) )
REAL_E rop( param_int(NELTS)      * param_int(NEQ)      )
REAL_E drodm( param_int(NELTS)    * param_int(NEQ)      )
#include "FastP/formule_param.h"

#include "pragma_align.for"
#include      "loopI_begin.for"
      ! > Parents Elements
      il = ng_pe(1 +v1mtr)      !left  element
      ir = ng_pe(1 +v2mtr)      !right element

#include      "FLUX_CONV/fluFaceEuler_i.for"
#include      "FLUX_CONV/fluFaceRans_i.for"
#include      "fluViscLaminar_i.for"
#include      "fluViscRans_i.for"
#include      "assemble_drodm_plus_vec1.for"
      enddo
end

```

Moreover a large part of code instructions necessary to compute Euler and Viscous fluxes can also be shared. This point is illustrated in 3.13, which represents the template used in *FastP* to compute the Euler flux with Ausm scheme. By comparing this listing with the listing 3.8 used in *FastS*, it is clear that about 80% of the instructions can be fully shared between the two modules for the calculation of the Euler flows, since the only difference concerns the estimation of the slopes. The same observation can be made for viscous terms where the assembly of the deformation tensor can also be totally shared.

Listing 3.13: Fortran template for AUSM flux include used in listing 3.12 for Euler Eqs (FluFaceEuler.for).

```

!.....Metric: full sharing between FastS and FastP
#include "Normale/normale_3dfull_i.for"

!Right and left slope estimation: specific to FastP
#include "Slope/TypeSlope_i.for"

!Right and left state: full sharing between FastS and FastP
#include "etat_GD.for"

!velocity at cell interface: full sharing between FastS and FastP
#include "Vit_ent/qn_3dfull_i.for"

! full sharing between FastS and FastP for the following instruction linked to
  Ausm dissipation
  c      = rgp*gam*(qm5+qp5)*0.5
  sound = sqrt(qn1*qn1 / c)
  tam    = c3*son+si
  tam1   = max(0.,tam)*c2 ! fct amortissement: c3*Mach+1
  ! Numerical velocity at interface

```

```

u      =0.25*(qn1+qn2)-tam1*(p2-p1)
! Numerical velocity for dissipation term
tdu = max(abs(u),c1*si)

!full sharing between FastS and FastP
#include "Vit_ent/fluvector_3dfull_i.for"

```

3.5.3. Adjoint by automatic differentiation

Onera has gained a significant experience in aerodynamic and aeroelastic discrete adjoint calculation within the framework of the *elsA* project. More precisely, as concerning the aerodynamic discrete adjoint, iterative solve with approximate Jacobian has been successfully implemented in *elsA* for the calculation of adjoint fields corresponding to functions of interest for rotors, blades or aircraft shape optimization.

$$\left(\frac{\partial R}{\partial W}\right)^{(APP)T} (\lambda_k^{(l+1)} - \lambda_k^{(l)}) = - \left(\left(\frac{\partial R}{\partial W}\right)^T \lambda_k^{(l)} + \left(\frac{\partial J_k}{\partial W}\right)^T \right) \quad k \in \{1, n_J\} \quad (3.3)$$

The corresponding scheme for the adjoint equation is given by equation (3.3) where, for the sake of completeness, we indicate that the inversion of the left hand side Jacobian $(\partial R / \partial W)^{(APP)T}$ has been carried out most often, with *elsA* by (LU) relaxation. The equivalent capability is to be provided in *FastS* and *FastP* in a near future.

However, in the context of *elsA*, the calculation of the flux balance was managed by as many C++ classes as physical terms in the PDE of interest, each of which calling several C++ methods corresponding to different set of indices (close to boundary conditions, interior of domain...) where finally Fortran routines are called. Unfortunately, the automatic differentiation of this C++/Fortran calculation chain appeared untraceable.

Conversely, the calculation of the flux balance in *Fast* - *R* in the equation (3.3), *droadm* from the equation 3.2 - is carried out by one or a very little number of single Fortran routine which are generated from the Python/template technique; this is extremely favorable for the use of automatic differentiation (AD) and automatic production of the routine calculating $(\frac{\partial R}{\partial W})^T \lambda_k^{(l)}$. Besides near field forces and moments are also calculated by single routines in *Fast* so that the calculation of the partial derivative of the function of interest $(\frac{\partial J_k}{\partial W})^T$ is also planned to be carried out by routines generated by (AD). This is the sketch of the first step of the development of *Fast-adjoint* (the approximate Jacobian, being first derived from *elsA* routines). The tool selected for Automatic Differentiation of the selected routines is *Tapenade* [6]. The evaluation of this library is presently under progress in the PRF ODYSSEY.

Future steps will include derivation of far-field function and other strategies for the resolution of the adjoint equation (most probably GMRES).

3.6. Performance results

3.6.1. Test case : 3D vortex advection and performance model

This test case performs the numerical integration of the Navier–Stokes equations in a three dimensional domain using a cartesian grid (which is around 2 times less expensive than the fully curvilinear formulation).

FÉVRIER 2020

The cartesian solver performance is relevant for the IBM approach. Periodicity is enforced in all three directions and the initial solution corresponds to the Lamb–Oseen vortex flow field. The test case is run on grids with 168^3 points (unless otherwise specified). This simplified problem may be representative of the workload one would typically find at the node level of a distributed-memory cluster when larger problem are considered. The same methodology has been followed for all the results : a series of 5 simulations consisting in 20 time iterations of the algorithm is averaged (standard deviations are also provided). For all the data presented in this paper, a compensation of the potential imbalance factor on row performance result is performed. Indeed, the OpenMP domain decomposition strategy may result in one or more threads performing more computations of fluxes than the others (*e.g.* see decomposition on figure 3.2 (right) for which the top-right thread computes more fluxes than the others). For this reason, it has been decided to compensate all the available results by the corresponding imbalance factor (*i.e.* $(6 \times 6)/(7 \times 7) \approx 74\%$ in the case illustrated in the figure 3.2). In practice for all the data presented hereafter, this factor is around 95% and always greater than 85%.

The metric used to measure the performances of our application is the time (in s) needed to perform a time-update of the solution divided by the number of degrees of freedom that characterize the problem (number of sub-steps and number of points). This performance ratio is expressed per number of threads considered :

$$\text{Effective time} = \frac{\text{elapsed CPU}}{\text{Nb. time steps} \times \text{Nb. cell}} \times \text{Nb. Threads}, \quad (3.4)$$

where Nb. time steps represents the cost of the time integration. This metric provides an estimation of a time-to-solution quantity which is particularly useful in the process of designing new cases (problem size and resources evaluation) on a given architecture [7].

3.6.2. Hardware architecture

The structured solver (FastS) is tested on multi-cores and many-cores architectures, the details of which are provided in the table 3.1. Two multi-cores (double sockets) architecture are considered. The first one features Haswell microprocessor (E5-2690v3) and the second one is Broadwell based (E5-2690v4). The (Knights Landing) model is the 7230 SKU which features 64 physical cores and 16GB of MCDRAM. The two multi-cores architectures feature the same amount of LLC memory per unit core (2.5MB). The amount of L2 cache in the Knights Landing architecture is equal to 512KB (1MB/tile), however this loss should be compensated by the on-chip 16GB of MCDRAM.

Name	Nb cores	L2 (KB)	L3 (MB)	Clock (GHz)
Haswell	12	256	30	2.6
Broadwell	14	256	35	2.6
Knights Landing	64	1000	-	1.3

Tableau 3.1 –: Hardware architectures (per sockets).

3.6.3. Importance of NUMA at the node level

We perform OpenMP parallelization at the node level to benefit from the total number of physical cores that share data in the DRAM. As a consequence, performances are sensitive to NUMA. Figure 3.9

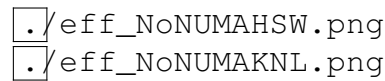


Fig. 3.9 –: Effects of NUMA-aware allocations on multi-cores node (Haswell, top) and many-cores node (Knights Landing, bottom). OS “first-touch” policy is applied to take NUMA into account. Knights Landing (Quadrant/Cache) is more flexible for the developers.

(top) illustrates the performance loss observed when allocation is performed sequentially (Not NUMA-aware) compare to NUMA-aware allocation using OS “first-touch” policy on bi-socket multi-cores node (Haswell). After a progressive loss of efficiency as the first socket is being populated (20% loss form 0 to 12 cores), a sudden burst of the effective time is observed (70% loss from 13 cores to full node). The first loss is due to a LLC saturation of the first socket, while the second loss is explained by a drastic increase of the QPI traffic between the two sockets. The Knights Landing node (booted in Quadrant/Cache mode) is not sensitive to NUMA which bring more flexibility to the developers.

3.6.4. Performance comparisons at fully loaded node

3.6.4.1. Core performance comparison on and



Fig. 3.10 –: Performance/threads comparison of multi- and many-core architectures. Knights Landing is 2 times slower/core at fully loaded node compared to multi-core architectures.

The unitary performance (*e.g.* available cache size, CPU clock speed...) of one core may be smaller for the 7230 compared with a classical architectures featuring a smaller number of faster cores like Haswell and Broadwell (see table 3.1). As a consequence, the effective time (*i.e.* expressed per thread count) may be larger for this architecture. However, it should be noted that it features the AVX512 vectors instructions and the fast MCDRM on-chip memory which may affect this conclusion. Nevertheless a slowdown in effective time is observed on Knights Landing compared to architectures (which are very similar, see figure 3.10). A factor 2 is indeed obtained between Knights Landing and fully populated Haswell and Broadwell. The KNL effective time features an important increase once one elementary tile is populated (a nearly 20% increase). This loss of efficiency corresponds to half of the total loss observed between 1 thread and 64 threads ($\approx 40\%$). Each tile features a 1MB L2-cache shared between 2 threads. Currently it is not possible to fit on it the memory needed by the 2 threads to update drodm (listing 3.2). An L2 cache saturation may be observed. The scattered (scattered in terms of KMP_AFFINITY) simulations (see figure 3.10), for which each tile is only populated by one thread up to 32 cores, show that the effective time obtained by a single thread simulation is maintained up to 32 cores, which confirms this hypothesis.

3.6.4.2. Effect of cache-blocking

The second-order finite volume approach algorithm used in the FastS module is convenient to take into account complex geometry. However, it suffers from a relatively low Arithmetic Intensity ($AI \approx 2$ on HSW in

FÉVRIER 2020



./speedup_blocked_fullPCSTR_COSC.png

Fig. 3.11 –: Effect of cache-aware algorithms. “3 scan” : three full scan of the domain (one per flux direction) in place of the chained inner loops in listing 3.2. “Chained” refers to the listing 3.2, written above.

cartesian coordinate) compared to higher order numerical methods for which peak performance may be easier to reach [8]. As a consequence, the algorithm is memory bounded and a careful attention must be paid at memory access. We opt for the cache-aware roofline in order to identify more precisely the effect of the memory hierarchy on performance [9]. Figure 3.11 reports the performance speedups obtained for two different implementations of the listing 3.2, with or without cache-blocking (cache blocks are of dimensions : (168, 3, 7)). As expected, few speedup is obtained when one thread/node is used (left). The “chained inner-loops” strategy offers a small speedup ($\approx 5\%$) compared to the three consecutive scan of the domain. Remarkably, not enforcing cache-blocking offers more performance on the Knights Landing node. Note that thread-blocking is however more pronounced with the 64 cores of KNL, (168, 21, 21) points/threads are obtained after the split of the domain (which is still significantly larger than the cache-block parameters used on). We recover the factor 2 slowdown when 1 thread/node is used on KNL compared to nodes. At full loaded nodes (figure 3.11, right) the cache-blocking strategy provides a factor 2.5 speedup for classical architectures. The multi-cores performances are satisfactory, 200GFLOPS are measured with Advisor, and the memory bandwidth of the application is around 80GB/s (which is closed to the system limit). Unfortunately, no cache-blocking improvement is observed for Knights Landing architecture. Without cache-blocking, a factor 3 speedup is obtained with KNL compared to classical . With cache-blocking, the is still 25% faster than the full Broadwell node.

3.6.4.3. Effect of vectorization



./speedup_vector_fullPCSTR_COSC.png

Fig. 3.12 –: Performance comparisons with and without vectorization. One thread/node (left). Fully loaded nodes (right).

The Knights Landing architecture features the new AVX512 vector instructions. Exploiting vectorization capability is therefore expected to be even more important compared with -based nodes with AVX2. The data layout combined with the inner-loop implementation (promotion of FMA operation and the use of explicit pre-processing directives to enable vectorization) make it possible to obtain a 58% vectorization efficiency on architectures (Haswell and Broadwell), see figure 3.12. The gain with vectorization is even larger when AVX512 instructions are used on (a 75% efficiency is observed). When the node is fully populated, a decrease of the vectorization gain is observed on (52% for Haswell node and 46% for the Broadwell). Remarkably, the gain of vectorization is kept when AVX512 is used on a fully loaded . With the 6x speedup provided by vectorization, Knights Landing performs better than classical nodes.

3.6.4.4. Scalability or vectorization ?

With the increased integration of modern microprocessors (e.g. Knights Landing), scalability is becoming fundamental to benefit from the increased performance capabilities. However, the overall performance is

also strongly affected by the capacity to achieving good vectorization efficiency. The figure 3.13 illustrates how disconnecting vectorization drastically improves strong scaling on both and architectures. Broadwell strong scaling is slightly less efficient than on Haswell node (this may be a consequence of a better performance when only one thread is used, possibly because of cache-blocking parameters, see figure 3.11). On Knights Landing a perfect scalability is recovered when disconnecting vectorization. However, overall time-to-solution is 6 times larger than when vectorization is activated and 70% scalability obtained.



Fig. 3.13 –: Scalability (strong scaling) observed on (left) and (right).

3.6.5. Strong problem size dependency : a cache-associativity issue ?

This section illustrates the performance sensitivity with respect to changes of the problem size. The dimensions of the structured block is gradually changed and the effective time reported.

3.6.5.1. Isotropic analysis

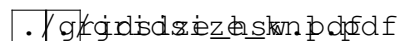
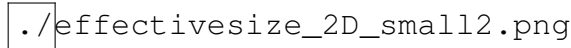


Fig. 3.14 –: Evolution of the effective time parameter with respect to change of the problem size for the Knights Landing (right) and Haswell (left) architectures. First, no padding between variables is performed (red circles). Then a 1000*64bit padding between variables (black line) is included. A series of 5 runs is shown for the two configurations (padding and no padding).

Isotropic changes of the problem dimensions are operated on each of the three directions. A unique parameter, noted *Size*, is then used to refer to these changes (*i.e.* the grid has $Size^3$ points). Figures 3.14 reports the grid sensitivity study for all the hardware architecture considered. Strong overshoots of performances are clearly identified (red circles) for specific choice of the parameter *Size*. In particular, for $Size = 65, 129, 193$ the CPU time may be doubled on (Haswell) while a factor 8 can occur on (Knights Landing). This strong dependency may have dramatic consequences on distributed MPI+OpenMP simulations commonly performed at the cluster scale (several hundreds nodes).

The hypothesis of a potential cache associativity issue is made and tested hereafter. Data padding is performed between each variables in memory (see the structure of array data layout in section ??) so as to prevent that the same cache way is used for the access in rop during the computation of drodm. The results obtained for a padding of 64000 bits are shown in black line on the figures 3.14. As can be seen, the performance loss has been fixed for the pathological sizes. However, it has not been possible to find a systematic process to chose the appropriate padding length for a given problem size and architecture. In particular, it may be possible that introducing padding will deteriorate the performances compare to zero padding simulations.

FÉVRIER 2020

Fig. 3.15 –: Effective time variations with respect to changes in I and J dimensions.

3.6.5.2. Anisotropic analysis

The performance variations with respect to independent changes of the domain size in i and j directions are reported in figure 3.15. Note that smaller effect on the effective time is observed when changing the grid size in k direction. Figure 3.15 reports results obtained on the HSW architecture using one thread. The results suggest that losses of performances occur in two different cases :

1. either the i or the j dimension is equal to $65 + N \times 64$ with N an arbitrary integer,
2. for certain specific values of the product $SizeI \times SizeJ$ (the corresponding curves in the (SizeJ, SizeI) plain are hyperbolas (three of them are reported in black in figure 3.15).

No systematic method to chose the appropriate padding size for a given problem size has been found. Instead, a practical solution has been retained. Footprints of performance sensitivity with respect to the grid size are done on preliminary jobs by scanning both i and j directions for different padding values. The best padding value is then collected for each problem size and used on production runs. Finally, as a by-product of the domain size dependency analysis, one can see that there exists periodic values of the domain size (in both I and J) for which a performance improvement is observed (white stripes in figure 3.15). The periodicity being equal to 4 (256bits) for Haswell and 8 (512bits) for KNL (figure 3.14), efficient data alignment for these particular sizes is likely to be responsible for the performance improvement.

3.6.6. Typical Cells Update Per Second

The HPC efficiency at the core level of the structured FastS solver is illustrated in table for different numerical (explicit or implicit time integration), grid types (see section 2.4.1) and model (Laminar, Spalart-Almaras or Euler) options for one subiteration of Navier-Stokes algorithm.

Tableau 3.2 –: Cells Update Per Second and per Core ($\times 10^6$) for E5-2690 v3 Haswell processor

Roughly implicit computations are two times more expensive than explicit ones. The same conclusions can be done between cartesian and full curvilinear grids, excepted for Euler computation for which cache eviction is less severe and the simplification of flux computation less important. The same kind of performance per core is obtained on Broadwell and Skylake processor. On the Xeon Phi KnightLanding, the performance per core is approximately decreased by a factor two.

3.6.7. Scalability at the cluster level : PRACE preparatory

In order to assess the ability of CGNS/Python composants to allow efficient computation on Tier0 supercomputer, assessment on two different cluster has been made :

- On the whole Sator onera cluster during the test phase of the supercomputer :
 - 620 nodes
 - Xeon E5 2680 node (broadwell)
 - 2 sockets of 14 cores
 - total number of cores = $620 \times 2 = 17360$
- On the Prace cluster Marenstrum Partition 4 in Barcelona :
 - 400 nodes
 - Xeon platinum 8160 node (Skylake)
 - 2 sockets of 24 cores
 - total number of cores = $400 \times 2 = 19200$

In a first step, data transfer between MPI processus (point to point communication) has been optimized on two nodes of Onera Sator cluster. The goal was to replace the use of Mpi4py python module employed at the Python level at every subiteration of the Navier-Stokes solver by a library of callable functions from layer C. This library, named COM, has been develop by X. Juvigny and allow transfers of mixed types (integer, real, character,...) in a single point to point message. This specification is necessary to avoid to generate 2 MPI messages (one for integers and one for floats), because the Connector module of Cassiopee requires interpolation coefficients (float) but also the list of the donors points (integer). Efficiency of the different approaches is compared on Fig. 3.16 for the Taylor Green vortex computed on a computational domain decomposed into 2 zones of 226^3 cells. The blue part corresponds to the time necessary to resolve the Navier-Stokes equation whereas the others parts deals with the different steps required by the data transfer between zones. Intra process determines the time to realize the transfer between zones belonging to the same MPI processus, whereas inter process, prepare buffer and Halo update are the different steps to allow transfer between zones computed by different MPI processus. From this figure, it appears clearly that the use of the C-COM library allow a reduction of a factor 2 of the time required to transfer data thanks to MPI. As the time necessary to compute the Navier-Stokes equation is not impacted by the choice of how MPI transfers are performed, the use of C-COM library allows a reduction of 15% of the total CPU time (compute + transfer) for this test case. It should be noticed that the speedup can be significantly larger for case with a large number of zones. Indeed in such case, returns from Python to C layer can impact the CPU because time is necessary to retrieve the data in the Python/CGNS tree :

- for a simple case with few zones (typically less than 100), 0.001s can be necessary for that purpose
- for a case with immersed boundary condition and octree cartesian grids (number of zone > 1000) , 0.01-0.05s can be spend in the tree search.

 ./mpi_mpi4py.png

Fig. 3.16 –: Comparisons of data transfer efficiency with MPI library between Mpi4py python module and C library.

Therefore, it is important to limit the number of returns between C and Python layer in case of large Python/CGNS tree to preserve a good parallel scalability (between 200 000 and 500 000 cells per core depending on the cluster). Thanks to the COM library, only one research in the tree by time step can be performed, whereas the use of Mpi4py requires one research by subiteration at least.

A Weak scalability study has been conducted on the Onera and MareNostrum clusters for the TGV test case thanks to a 226^3 affected to each node employed for the computation. The total number of cells

FÉVRIER 2020

computed on ONERA cluster is 7.1510^9 and 7.1510^9 on MareNostrum when all available CPU resources are employed. The results are depicted in Fig. 3.17. Almost 95% efficient scalability is obtained for this test, which demonstrates that Python/CGNS Navier-Stokes solver can be HPC compatible even on a large size cluster like those of the PRACE level.


./mpi_scalability.png

Fig. 3.17 –: Weak scalability for hybrid MPI/OpenMP computation of TGV test case (11.5×10^6 cells per nodes)

FÉVRIER 2020

FÉVRIER 2020

4. EFFICACITÉ DES MÉTHODES NUMÉRIQUES

4.1. Introduction

Cette tâche du PRF était décomposée en 2 parties consacrées au développement :

- d'une méthode d'intégration temporelle explicite consistante en temps à pas de temps local. Cette méthode est décrite dans la section 4.2.
- de techniques de maillage afin de rendre la prise en compte de géométrie complexe beaucoup plus rapide dans un workflow "bureau d'étude". Cette activité a essentiellement portée sur la mise au point d'une méthode de frontière immergée mise en oeuvre sur des maillages de type Octree cartésien générés automatiquement par la suite Cassiopée. Cette activité, qui a fait l'objet d'un rapport en année 1 du PRF [10] et de 2 exposés en conférence AIAA [11, 12], est prometteuse. Comme le maillage cartésien ne permet pas de résoudre les écoulements de type couche limite avec des résolutions de l'ordre de $y^+ = 1$, car cela réclamerait des maillages de plusieurs centaines de milliards de cellules. Pour éviter cela, des lois de parois sont développées/validées dans un contexte RANS stationnaire ou LES instationnaires. De nombreux cas ont déjà été étudiés (2D, 3D, subsonique, transonique, supersonique, RANS stationnaire, LES instationnaire,...) et ces premiers résultats obtenus dans le PRF CHAMPION sont très encourageants. La méthode est toujours en phase de développement/validation dans le PRF MEMFIS, qui a débuté début 2019, principalement dans le domaine des lois de parois et pour renforcer sa souplesse en permettant un mélange entre les approches maillage curviligne et Octree cartésien.

4.2. Local time stepping scheme for unsteady computation

Numerical simulations are widely used to study fluid mechanics problems. Simulations are based on the discretized form of fluid mechanics equations (mainly Euler or Navier-Stokes equations). This discretized form is obtained by applying space and time integration methods. Finite volume approach is one of the most popular space integration method in CFD. For time integration, explicit Runge-Kutta methods have proven their efficiency for solving fluid mechanics problems. However, with these methods, the time step is strongly restricted by the numerical stability condition (CFL condition). This stability condition is defined in each cell and imposes that the time step is lower than the ratio between the smallest length in the cell and the magnitude of the wave speed in the cell. The time step for the entire domain is chosen to be lower than the smallest ratio in the mesh (generally imposed by the smallest cell in the mesh). Consequently, the time step may be much smaller than necessary in the other cells, which is not efficient. It is particularly true in most CFD computations involving walls, which present some zones of mesh refinements where the stability condition is much more restrictive than in the other areas.

Implicit, unconditionally stable timestepping methods allow to use larger time steps, which reduces the computational time as compared to explicit methods. Note that the time step can not be too large in order to keep a good accuracy of the solution. For a Large Eddy Simulation of turbulent flows, the ratio between the time step and each cell characteristic length can not exceed 2-3 [14]. Moreover, implicit methods seem to have two drawbacks. First, an efficient parallel implementation of these methods may be difficult, especially if a strong scalability is required. In addition, implicit methods can be less accurate than explicit methods due to a residual numerical noise.

The use of an explicit local time stepping scheme seems to be a good approach to reduce the computational cost of explicit methods keeping their advantages with respect to implicit methods (solution accuracy and good compatibility with parallel programming). Indeed, an explicit local time stepping scheme allows a time step adaptation on the mesh to satisfy local stability conditions. Small time steps are used on cells which present the most severe CFL and larger time steps are applied on cells where CFL is less restrictive. This should result in a reduction of the computational cost with respect to explicit methods using a uniform time step.

Two approaches are possible to construct explicit local time stepping schemes. In the first approach, each cell of the computational domain is integrated with its own maximal allowable time step [15, 16, 17]. In the second approach, each time step of the computational domain is defined as a fraction of the largest time step and each time step is associated to a group of contiguous cells [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]. In the literature, these schemes are frequently called "multirate schemes". All the local time stepping schemes presented in this article are constructed by following this second approach. Indeed, this approach seems well suited to High Performance Computations (HPC) since the memory accesses are much more efficient due to the regular predictable pattern of the cells groups. However, local time stepping schemes seem to have an important drawback : it is difficult to guarantee both high order time accuracy and mass conservation. In the 1980's, Osher and Sanders [19] proposed a locally varying time stepping scheme based on the forward Euler method. It is conservative but only first order accurate in time. Dawson and Kirby [26] extended the work of Osher and Sanders by performing a conservative local time stepping scheme based on a second order Runge-Kutta scheme. Tang and Warnecke [20] proved that the schemes in [19, 26] have a lack of consistency and proposed a new second order time accurate local time stepping scheme. However, their scheme is not mass-conservative. The local time stepping schemes presented in [30, 22, 23] are also second order time accurate but not mass-conservative. In [21], a comparison is made between different multirate Runge-Kutta schemes and a multirate time stepping scheme of order three in time is proposed. This third order local time stepping scheme borrows some ideas of implicit-explicit (IMEX) schemes [31]. However, it is not mass-conservative. Another third order non conservative explicit local time stepping scheme based on the theory of partitioned Runge-Kutta methods (PRK) is proposed in [27]. Recently, multirate Runge-Kutta schemes up to order four are developed by several authors [28, 29]. They are derived from the Multirate Infinitesimal Step method, initially introduced in [32]. These methods are promising, even though they are not mass-conservative when applied to solve partial differential equations.

The strategy proposed by Constantinescu and Sandu [18] to construct local time stepping schemes seems more promising. Indeed, their strategy allows to obtain second order accurate and mass conservative local time stepping schemes. Moreover, the overall stability of their original local time stepping scheme proposed in [18] can be improved using a higher order Runge-Kutta base method [21, 24]. Nevertheless, the overall method remains second order accurate in time [21, 24].

In this article, we propose two local time stepping schemes that are second order and third order time accurate. Consistent solutions are calculated at the interface between domains of different time steps like in [20, 21, 25, 22, 23] to obtain the desired orders of accuracy. Moreover, a correction stage is added to make these schemes mass-conservative. The first goal of this article is to make a comparison between our strategy to construct mass-conservative local time stepping schemes and the strategy of Constantinescu and Sandu. Moreover, in most of articles devoted to local time stepping methods, the schemes are validated on rather academic test cases (Burgers equation, advection equation...) [26, 20, 22, 23, 18, 21, 30]. The second goal of this article is then to show the ability of our third order local time stepping scheme to compute efficiently an industrial-like case : a 3D Large-Eddy Simulation (LES) over an airfoil.

In section 1, we present some theoretical concepts for the construction of local time stepping schemes

FÉVRIER 2020

and we introduce the strategy used by Constantinescu and Sandu. In section 2, we introduce our approach to construct mass-conservative local time stepping schemes of order two and three. In section 3, the comparison between the two strategies is performed with several academic test cases. The results of the LES over an airfoil computed with our third order local time stepping scheme are also presented in this final part.

4.2.1. Local time stepping scheme and Partitioned Runge-Kutta methods

4.2.1.1. Semi-discretization of a one-dimensional hyperbolic equation

We consider the one-dimensional hyperbolic equation :

$$\frac{\partial y(t, x)}{\partial t} + \frac{\partial f(y(t, x))}{\partial x} = 0, \quad (4.1)$$

with $y(0, x) = y^0(x)$, $x \in]-\infty, +\infty[$ and $t > 0$. We also consider a 1D computational domain divided into cells of variable length Δx_i . Cell i is bounded by points $x_{i-\frac{1}{2}}$ and $x_{i+\frac{1}{2}}$. The finite volume method method is applied to (4.1). After space integration over cell i , we obtain :

$$\frac{\partial y_i}{\partial t} = -\frac{1}{\Delta x_i}(F_{i+\frac{1}{2}} - F_{i-\frac{1}{2}}), \quad (4.2)$$

where $y_i = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} y(t, x) dx$. The numerical flux at point $x_{i+\frac{1}{2}}$ is denoted by $F_{i+\frac{1}{2}}$, and depends on several values around y_i . For instance, we have : $F_{i+\frac{1}{2}}(y_{i-1}, y_i, y_{i+1}, y_{i+2})$ in the case of a flux function with a four cell stencil. For the sake of clarity for the reader, $F_{i+\frac{1}{2}}(y_{i-1}, y_i, y_{i+1}, y_{i+2})$ is denoted by $F_{i+\frac{1}{2}}$. For the sake of convenience, we introduce D_i , the flux balance on cell i :

$$D_i = \frac{1}{\Delta x_i}(F_{i+\frac{1}{2}} - F_{i-\frac{1}{2}}). \quad (4.3)$$

The flux balance D_i is obviously a function of y_i , and several values around it ($y_{i-2}, y_{i-1}, y_{i+1}, \dots$). With this notation, (4.2) now reads :

$$\frac{\partial y_i}{\partial t} = -D_i. \quad (4.4)$$

Now (4.2) and (4.4) can be viewed as ordinary differential systems that can be solved with an explicit Runge-Kutta (RK) method, as described in the following section. This is the so called method of lines (MOL).

4.2.1.2. Explicit Runge-Kutta methods

We consider an ordinary differential equation :

$$\frac{dy}{dt} = f(t, y) \quad y(t = 0) = y^0 \quad (4.5)$$

The solution of (4.5) at time t^n is y^n . A s-step explicit Runge-Kutta method allows to compute y^{n+1} (solution of (4.5) at time $t^{n+1} = t^n + \Delta t$) by using y^n and $s-1$ intermediate values. In [33], a s-step explicit Runge-Kutta method is defined by the formulas :

$$t^{n,i} = t^n + c_i \Delta t, \quad (4.6)$$

$$y^{n,i} = y^n + \Delta t \sum_{j=1}^{i-1} a_{ij} K^{n,j}, \quad (4.7)$$

$$K^{n,i} = f(t^{n,i}, y^{n,i}), \quad (4.8)$$

where $1 \leq i \leq s$, and $K^{n,1} = f(t^n, y^n)$. The i – th intermediate time, the i – th intermediate value, and the i – th intermediate evaluation of the f function between t^n and $t^n + \Delta t$ are denoted by $t^{n,i}$, $y^{n,i}$ and $K^{n,i}$. The value at time $t^n + \Delta t$ is given by the following formula :

$$y^{n+1} = y^n + \Delta t \sum_{i=1}^s b_i K^{n,i} \quad (4.9)$$

The method is defined by its coefficients $A = \{a_{ij}\}$, $b = \{b_i\}$ and $c = \{c_i\}$ that can be represented in a Butcher tableau (see [33]) :

	$c_1 = 0$	0					
	c_2	a_{21}					
	c_3	$a_{31} \quad a_{32}$					
	\cdot	\cdot					
	\cdot	\cdot					
	\cdot	\cdot					
	c_s	$a_{s1} \quad a_{s2} \quad \cdot \quad \cdot \quad \cdot \quad a_{s,s-1}$					
$\frac{c}{b}$		$b_1 \quad b_2 \quad \cdot \quad \cdot \quad \cdot \quad b_{s-1} \quad b_s$					

Tableau 4.1 –: Butcher tableau of an s-stage explicit Runge-Kutta method

All Runge-Kutta methods in this article satisfy the property of consistency :

$$c_i = \sum_{j=1}^{i-1} a_{ij}. \quad (4.10)$$

4.2.1.3. Partitioned Runge-Kutta methods

In [18] and [30], Partitioned Runge-Kutta (PRK) methods are applied to temporal multiscale problems.

Let us consider a 1D computational domain composed of n cells (figure 4.1). The finite volume method applied to the one-dimensional hyperbolic equation (4.1) on the whole domain leads to the autonomous system of ordinary differential equations :

$$\frac{dy}{dt} = f(y), \quad y \in R^n. \quad (4.11)$$

As shown in figure 4.1, the computational domain is divided into two subdomains which have different characteristic times. The first subdomain has a slow characteristic time and contains cells from cell 1 to cell i_0

FÉVRIER 2020

(large cells). In the following, this subdomain is referred to as the "slow subdomain". The second subdomain has a fast characteristic time and contains cells from cell i_{0+1} to cell n (small cells). In the following, this subdomain is referred to as the "fast subdomain"

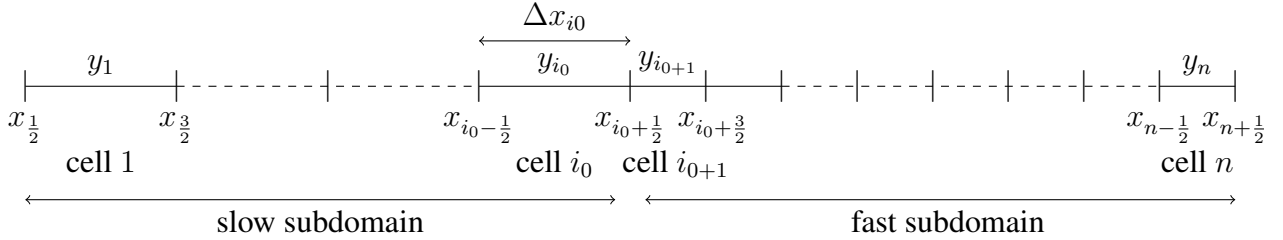


Fig. 4.1 –: Partitioned computational domain

We denote by $Y_S = [y_1, y_2, \dots, y_{i_0}]$ the vector of unknowns in the slow subdomain and by $Y_F = [y_{i_{0+1}}, y_{i_{0+2}}, \dots, y_n]$ the vector of unknowns in the fast subdomain. System (4.11) can be rewritten using the variables Y_S and Y_F :

$$\begin{cases} \frac{dY_S}{dt} = f_S(Y_S, Y_F) \\ \frac{dY_F}{dt} = f_F(Y_S, Y_F) \end{cases} \quad (4.12)$$

The dependency between vectors Y_S and Y_F in each equation of (4.12) occurs at the interface between the subdomains. The dissimilar time scale for Y_S and Y_F is a motivation for solving these equations with two Runge-Kutta methods that use different time steps. We denote by RKS the s -stage Runge-Kutta method for the slow subdomain. It is used to solve the first equation and its coefficients are : A^S, b^S, c^S . Similarly, we denote by RKF the s -stage Runge-Kutta method for the fast subdomain. It is used to solve the second equation and its coefficients are : A^F, b^F, c^F . According to [33], the PRK method using RKS and RKF is given by the following formulas :

$$\begin{aligned} Y_S^{n+1} &= Y_S^n + \Delta t \sum_{i=1}^s b_i^S K_S^{n,i} & Y_F^{n+1} &= Y_F^n + \Delta t \sum_{i=1}^s b_i^F K_F^{n,i} \\ Y_S^{n,i} &= Y_S^n + \Delta t \sum_{j=1}^{i-1} a_{ij}^S K_S^{n,j} & Y_F^{n,i} &= Y_F^n + \Delta t \sum_{j=1}^{i-1} a_{ij}^F K_F^{n,j} \\ K_S^{n,i} &= f_S(Y_S^{n,i}, Y_F^{n,i}) & K_F^{n,i} &= f_F(Y_S^{n,i}, Y_F^{n,i}) \end{aligned} \quad (4.13)$$

] where $1 \leq i \leq s$ with $K_S^{n,1} = f_S(Y_S^n, Y_F^n)$ and $K_F^{n,1} = f_F(Y_S^n, Y_F^n)$. We remind that Y_S^n and Y_F^n denote the vectors of unknowns at time t^n in the slow subdomain and in the fast subdomain, respectively. Y_S^{n+1} and Y_F^{n+1} are the vectors of unknowns at time $t^n + \Delta t$. The i -th intermediate slow solution and the i -th evaluation of the slow function f_S between t^n and $t^n + \Delta t$ are denoted by $Y_S^{n,i}$ and $K_S^{n,i}$. The i -th intermediate fast solution and the i -th evaluation of the fast function f_F between t^n and $t^n + \Delta t$ are denoted by $Y_F^{n,i}$ and $K_F^{n,i}$.

In most explicit local time stepping schemes (for instance [18], [20], [19] and [21]), RKS and RKF methods are based on the same RK method. RKF method is composed of two successive applications of the base method with the time step $\frac{\Delta t}{2}$, and some additional values to make the transition with the RKS method. RKS method is composed of the base method with the time step Δt , and some additional values to make the transition with the RKF method. With this strategy, an explicit local time stepping scheme with time steps Δt

in the slow subdomain and $\frac{\Delta t}{2}$ in the fast subdomain is obtained. To have a Partitioned Runge-Kutta method of order p and therefore, a local time stepping scheme of order p , both RKS and RKF methods must be of order p but this is not sufficient. The coefficients of the methods have to satisfy additional coupling conditions. Following the methodology of [33] we can derive the following conditions on the Runge-Kutta coefficients to obtain a PRK method of order two and three, respectively (tables 4.2 and 4.3).

RKS of order 2	RKF of order 2	Coupling conditions
$\sum_{i=1}^s b_i^S = 1$ $\sum_{i=1}^s b_i^S c_i^S = \frac{1}{2}$	$\sum_{i=1}^s b_i^F = 1$ $\sum_{i=1}^s b_i^F c_i^F = \frac{1}{2}$	$\sum_{i=1}^s b_i^S c_i^F = \frac{1}{2}$ $\sum_{i=1}^s b_i^F c_i^S = \frac{1}{2}$

Tableau 4.2 –: Conditions to obtain a second order accurate PRK method

RKS of order 3	RKF of order 3	Coupling conditions	
$\sum_{i=1}^s b_i^S = 1$ $\sum_{i=1}^s b_i^S c_i^S = \frac{1}{2}$ $\sum_{i=1}^s b_i^S (c_i^S)^2 = \frac{1}{3}$ $\sum_{i=1}^s \sum_{j=1}^{i-1} b_i^S a_{ij}^S c_j^S = \frac{1}{6}$	$\sum_{i=1}^s b_i^F = 1$ $\sum_{i=1}^s b_i^F c_i^F = \frac{1}{2}$ $\sum_{i=1}^s b_i^F (c_i^F)^2 = \frac{1}{3}$ $\sum_{i=1}^s \sum_{j=1}^{i-1} b_i^F a_{ij}^F c_j^F = \frac{1}{6}$	$\sum_{i=1}^s b_i^S c_i^F = \frac{1}{2}$ $\sum_{i=1}^s \sum_{j=1}^{i-1} b_i^S a_{ij}^F c_j^S = \frac{1}{6}$ $\sum_{i=1}^s \sum_{j=1}^{i-1} b_i^S a_{ij}^F c_j^F = \frac{1}{6}$ $\sum_{i=1}^s \sum_{j=1}^{i-1} b_i^S a_{ij}^S c_j^F = \frac{1}{6}$ $\sum_{i=1}^s b_i^S c_i^F c_i^S = \frac{1}{3}$	$\sum_{i=1}^s b_i^F c_i^S = \frac{1}{2}$ $\sum_{i=1}^s \sum_{j=1}^{i-1} b_i^F a_{ij}^S c_j^F = \frac{1}{6}$ $\sum_{i=1}^s \sum_{j=1}^{i-1} b_i^F a_{ij}^S c_j^S = \frac{1}{6}$ $\sum_{i=1}^s \sum_{j=1}^{i-1} b_i^F a_{ij}^F c_j^S = \frac{1}{6}$ $\sum_{i=1}^s b_i^F c_i^F c_i^S = \frac{1}{3}$

Tableau 4.3 –: Conditions to obtain a third order accurate PRK method

The number of conditions quickly increases with the order of accuracy which makes the implementation of high order local time stepping schemes difficult.

Constantinescu and Sandu strategy

In [18], Constantinescu and Sandu have developed a strategy, based on PRK methods, to obtain conservative and second order accurate local time stepping schemes. In this subsection, we briefly describe two schemes based on this strategy. The first one is the original scheme, based on a RK2 method [18]. The second one follows the same strategy but is based on a RK3 method (see [21, 24]). A comparison between these schemes and the schemes we propose (that are based on a different strategy, see part 3) is performed in part 4.

First scheme : original scheme of Constantinescu and Sandu

The original scheme developed by Constantinescu and Sandu is based on the second order Heun method :

0	0
1	1 0
	$\frac{1}{2}$ $\frac{1}{2}$

Tableau 4.4 –: Base method for the original scheme of Constantinescu and Sandu

The RKF method (Table 4.6) is obtained by applying twice successively the base method with the time step $\frac{\Delta t}{2}$. The RKS method (Table 4.5) is obtained by applying twice successively the base method with the time step Δt and dividing all b_i coefficients by 2.

FÉVRIER 2020

0	0
1	1 0
0	0 0 0
1	0 0 1 0
	$\frac{1}{4} \quad \frac{1}{4} \quad \frac{1}{4} \quad \frac{1}{4}$

Tableau 4.5 –: RKS method

0	0
$\frac{1}{2}$	$\frac{1}{2}$ 0
$\frac{1}{2}$	$\frac{1}{4}$ $\frac{1}{4}$ 0
1	$\frac{1}{4}$ $\frac{1}{4}$ $\frac{1}{2}$ 0
	$\frac{1}{4} \quad \frac{1}{4} \quad \frac{1}{4} \quad \frac{1}{4}$

Tableau 4.6 –: RKF method

Both RKS and RKF methods are second order accurate and we have $b_i^S = b_i^F$ for every i . These properties ensure that this local time stepping scheme is second order accurate [18]. The equality of coefficients b_i ensures that the scheme is mass conservative [18]. In [18], the authors demonstrate that the RK[S] method described in Table 4.5 is only applied in a buffer zone. It is an area of variable length (the length depends on the spatial stencil), adjacent to the interface. Outside this buffer zone, RKS reduces to the Heun method described in Table 4.4. Finally, our semi-discrete approximation (4.4) is solved by the algorithm shown in table 4.7. We introduce the following notations for this algorithm :

- for all n , the values at times t^n and t^{n+1} (with $t^{n+1} = t^n + \Delta t$, Δt corresponding to the largest time step) are denoted by y^n and y^{n+1} ,
- for all n , the k - th intermediate value between times t^n and t^{n+1} is denoted by $y^{n,k}$.

According to (4.3) and the notations given above :

- for all n , $D_i^n = \frac{1}{\Delta x}(F_{i+\frac{1}{2}}(y_i^n, y_{i+1}^n) - F_{i-\frac{1}{2}}(y_{i-1}^n, y_i^n))$,
- for all n , the k - th intermediate flux balance on cell i is given by : $D_i^{n,k} = \frac{1}{\Delta x}(F_{i+\frac{1}{2}}(y_i^{n,k}, y_{i+1}^{n,k}) - F_{i-\frac{1}{2}}(y_{i-1}^{n,k}, y_i^{n,k}))$.

	Large-time-scale-subdomain		Short-time-scale-subdomain
1	$y_i^{n,1} = y_i^n - \Delta t D_i^n$	$y_i^{n,1} = y_i^n - \Delta t D_i^n$	$y_i^{n,1} = y_i^n - \frac{\Delta t}{2} D_i^n$
2		$y_i^{n,2} = y_i^n$	$y_i^{n,2} = y_i^n - \frac{\Delta t}{4} [D_i^n + D_i^{n,1}]$
3		$y_i^{n,3} = y_i^n - \Delta t D_i^{n,2}$	$y_i^{n,3} = y_i^{n,2} - \frac{\Delta t}{2} D_i^{n,2}$
4	$y_i^{n+1} = y_i^n - \frac{\Delta t}{2} [D_i^n + D_i^{n,1}]$	$y_i^{n+1} = y_i^n - \frac{\Delta t}{4} [D_i^n + D_i^{n,1} + D_i^{n,2} + D_i^{n,3}]$	$y_i^{n+1} = y_i^{n,2} - \frac{\Delta t}{4} [D_i^{n,2} + D_i^{n,3}]$

Buffer zone

Tableau 4.7 –: Algorithm of the original scheme of Constantinescu and Sandu

We can notice that this scheme is internally inconsistent, since at the interface, some values estimated at different times are mixed in the flux balance D . For instance, at the interface, D_i^2 contains values estimated at time t^n (y_i^2 from the buffer zone) and values estimated at time $t^{n+\frac{1}{2}}$ (y_i^2 from the fast subdomain). Despite this inconsistency, the scheme is second order accurate and mass-conservative.

For each cell in the buffer zone, four flux balances have to be calculated, therefore the computational cost is the same as in the fast subdomain. In our applications, we use a flux function with a four cell stencil. In this case, it can be proven that the size of the buffer zone is equal to four cells.

Second scheme

In [21] and [24], a RK3 method is used as the base method and the strategy of Constantinescu and Sandu is applied. Here, we extend this approach using a RK3 low-storage scheme lowstorage as the base

method. Low-storage Runge-Kutta schemes require less storage than "classical" Runge-Kutta schemes. The RK3 low-storage method applied to (4.11) reads :

Step 1	$K^n = \Delta t f(t^n, y^n)$ $y^{n,1} = y^n + \beta_1 K^n$	$\beta_1 = \frac{1}{2}$
Step 2	$K^{n,1} = \alpha_2 K^n + \Delta t f(t^n + c_2 \Delta t, y^{n,1})$ $y^{n,2} = y^{n,1} + \beta_2 K^{n,1}$	$\alpha_2 = -0.6830127$ $c_2 = \frac{1}{2}$ $\beta_2 = 0.9106836$
Step 3	$K^{n,2} = \alpha_3 K^{n,1} + \Delta t f(t^n + c_3 \Delta t, y^{n,2})$ $y^{n+1} = y^{n,2} + \beta_3 K^{n,2}$	$\alpha_3 = -\frac{4}{3}$ $\beta_3 = 0.3660254$ $c_3 = 0.7886751$

Tableau 4.8 –: The RK3 low-storage scheme written in "low-storage form"

The values of the coefficients α and β have been proposed by Lowery and Reynolds loweryreynolds. Successive values of K and y overwrite the previous ones so that at any stage only $2N$ storage locations are required. The same method can be written in the "classical Runge-Kutta form" and a Butcher tableau can be associated :

Step 1	$K^n = \Delta t f(t^n, y^n)$ $y^{n,1} = y^n + a_{21} K^n$	$a_{21} = \beta_1$
Step 2	$K^{n,1} = \Delta t f(t^n + c_2 \Delta t, y^{n,1})$ $y^{n,2} = y^n + a_{31} K^n + a_{32} K^{n,1}$	$a_{31} = \beta_1 + \beta_2 \alpha_2$ $a_{32} = \beta_2$ $c_2 = \frac{1}{2}$
Step 3	$K^{n,2} = \Delta t f(t^n + c_3 \Delta t, y^{n,2})$ $y^{n+1} = y^n + b_1 K^n + b_2 K^{n,1} + b_3 K^{n,2}$	$b_1 = a_{31} + \beta_3 \alpha_2 \alpha_3$ $b_2 = \beta_2 + \beta_3 \alpha_3$ $b_3 = \beta_3$ $c_3 = 0.7886751$

0	0		
c_2	a_{21}		
c_3	a_{31}	a_{32}	
	b_1	b_2	b_3

Tableau 4.9 –: The RK3 low-storage scheme written in "classical form" and its associated Butcher tableau

It is to be noticed that $a_{31} (\approx -0.12)$ is negative, so this scheme is not total variation diminishing (TVD). In the following, we only use the "classical form" described in table 4.9 for our RK3 scheme. Indeed, its associated Butcher tableau is very useful for constructing and describing local time stepping schemes.

Following the strategy of Constantinescu and Sandu, the base method is duplicated to construct RKS (table 4.10) and RKF (table 4.11) methods.

FÉVRIER 2020

0	0					
c_2	a_{21}					
c_3	a_{31}	a_{32}				
0	0	0	0			
c_2	0	0	0	a_{21}		
c_3	0	0	0	a_{31}	a_{32}	
	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$

Tableau 4.10 –: RKS method

0	0								
$\frac{c_2}{2}$	$\frac{a_{21}}{2}$								
$\frac{c_3}{2}$	$\frac{a_{31}}{2}$	$\frac{a_{32}}{2}$							
$\frac{1}{2}$	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$						
$\frac{1+c_2}{2}$	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$	$\frac{a_{21}}{2}$					
$\frac{1+c_3}{2}$	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$	$\frac{a_{31}}{2}$	$\frac{a_{32}}{2}$				
	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$			

Tableau 4.11 –: RKF method

Even though both RKS and RKF methods are third order accurate, the local time stepping scheme is only second order accurate because some third order coupling conditions are not satisfied [18]. However, the local time stepping is mass conservative [24]. Like for the original scheme of Constantinescu and Sandu, there is a buffer zone at the interface, in the slow subdomain. Outside this buffer zone, the RKS method reduces to the base method described in table 4.9. Finally, our semi-discrete approximation (4.4) is solved by the algorithm shown in table 4.12, in which the notations introduced in subsection 4.2.1.3 are re-used.

	Large-time-scale-subdomain		Short-time-scale-subdomain
1	$y_i^{n,1} = y_i^n - a_{21}\Delta t D_i^n$	$y_i^{n,1} = y_i^n - a_{21}\Delta t D_i^n$	$y_i^{n,1} = y_i^n - \frac{a_{21}}{2}\Delta t D_i^n$
2	$y_i^{n,2} = y_i^n - a_{31}\Delta t D_i^n$ $- a_{32}\Delta t D_i^{n,1}$	$y_i^{n,2} = y_i^n - a_{31}\Delta t D_i^n$ $- a_{32}\Delta t D_i^{n,1}$	$y_i^{n,2} = y_i^n - \frac{a_{31}}{2}\Delta t D_i^n$ $- \frac{a_{32}}{2}\Delta t D_i^{n,1}$
3		$y_i^{n,3} = y_i^n$	$y_i^{n,3} = y_i^n - \frac{b_1}{2}\Delta t D_i^n$ $- \frac{b_2}{2}\Delta t D_i^{n,1} - \frac{b_3}{2}\Delta t D_i^{n,2}$
4		$y_i^{n,4} = y_i^n - a_{21}\Delta t D_i^{n,3}$	$y_i^{n,4} = y_i^{n,3} - \frac{a_{21}}{2}\Delta t D_i^{n,3}$
5		$y_i^{n,5} = y_i^n - a_{31}\Delta t D_i^{n,3}$ $- a_{32}\Delta t D_i^{n,4}$	$y_i^{n,5} = y_i^{n,3} - \frac{a_{31}}{2}\Delta t D_i^{n,3}$ $- \frac{a_{32}}{2}\Delta t D_i^{n,4}$
6	$y_i^{n+1} = y_i^n - b_1\Delta t D_i^n$ $- b_2\Delta t D_i^{n,1} - b_3\Delta t D_i^{n,2}$	$y_i^{n+1} = y_i^n - \frac{b_1}{2}\Delta t (D_i^n + D_i^{n,3})$ $- \frac{b_2}{2}\Delta t (D_i^{n,1} + D_i^{n,4}) - \frac{b_3}{2}\Delta t (D_i^{n,2} + D_i^{n,5})$	$y_i^{n+1} = y_i^{n,3} - \frac{b_1}{2}\Delta t D_i^{n,3}$ $- \frac{b_2}{2}\Delta t D_i^{n,4} - \frac{b_3}{2}\Delta t D_i^{n,5}$

Buffer zone

Tableau 4.12 –: Algorithm of the second local time stepping scheme

The algorithm is written in "classical form" but it can be easily implemented in the low storage form since these two formulations are equivalent. Note that with a flux function with a four cell stencil, the size of the buffer zone is equal to 6 cells.

4.2.2. New schemes

In this section, we present a different approach to construct local time stepping schemes. Like in the strategy of Constantinescu and Sandu, we use the theory of PRK methods but here we guarantee the internal consistency of the scheme. This allows to obtain higher order accuracy, since we are able to design a third order scheme. Note that our schemes require a correction step to make them mass conservative. [In these new schemes, the ratio between two adjacent time steps is limited to 2. Several authors develop local time stepping schemes that can handle random ratios. These ratios can be higher or equal to two (2,3,4,5...) and even lower

than 2 ($2/3$, $4/3$...). We remind that one of our goals is to use our local time stepping schemes to compute a large eddy simulation (LES) of a flow. In most meshes dedicated to LES or CFD in general, the grid size is rather continuous to guarantee the accuracy of the solution. Due to this continuity in cells size, the use of random ratios higher than 2 (3,4,5...) don't seem useful. The use of random ratios lower than 2 ($2/3$, $4/3$...) seems more useful and effective on LES meshes. However, it tends to increase the algorithmic complexity. Indeed, the algorithm must be able to handle all possible ratios between adjacent time step. The choice to limit the ratio between adjacent time steps to 2 is a compromise between efficiency and algorithmic complexity.]

4.2.2.1. First scheme : a second order time accurate scheme

Again, we use the second order Heun method (table 4.4) as the base method. It is applied twice with the time step $\frac{\Delta t}{2}$ to obtain the RKF method (table 4.14). Our RKS method is composed of the base method and some additional values to perform the transition with the RKF method and achieve the desired order of accuracy. Thus, both RKS and RKF methods are of order 2. Now, we impose internal consistency by the relation $c_i^S = c_i^F$, for every i . It can be easily seen that second order coupling conditions (see table 4.2) are satisfied with this condition. Note that the intermediate time coefficients of RKF are already fixed : we have $c^F = \{0; \frac{1}{2}; \frac{1}{2}; 1\}$. For RKS, we have $c_1^S = 0$ and $c_4^S = 1$, but c_2^S and c_3^S are still undetermined. Then the internal consistency is obtained with $c_2^S = c_3^S = \frac{1}{2}$. This requires to define additional intermediate values at time $t^n + \frac{1}{2}\Delta t$ at the interface, in the slow subdomain. The coefficients a_{ij} needed for the calculation of these additional value are determined by the consistency relation (4.10). We remind that these additional values are only calculated at the interface and are not used for the calculation of y^{n+1} in the slow subdomain. The Butcher tableau of RKS and RKF methods read :

0	0			
$\frac{1}{2}$	$\frac{1}{2}$	0		
$\frac{1}{2}$	$\frac{1}{2}$	0	0	
1	1	0	0	0
	$\frac{1}{2}$	0	0	$\frac{1}{2}$

Tableau 4.13 –: RKS method

0	0			
$\frac{1}{2}$	$\frac{1}{2}$	0		
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	0	
1	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$	0
	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

Tableau 4.14 –: RKF method

However, this local time stepping scheme defined by RKS and RKF is not mass conservative. It can be shown by applying RKS and RKF methods to (4.2) on cells i_0 and i_{0+1} , respectively. We remind that cell i_0 is the last cell of the slow subdomain while cell i_{0+1} is the first cell of the fast subdomain (see figure 4.1). The application of RKS and RKF methods to (4.2) on cells i_0 and i_{0+1} is shown in table 4.15. Note that to clarify the presentation, we assume that the flux function has a 2 cell stencil : $F_{i+\frac{1}{2}} = F_{i+\frac{1}{2}}(y_i, y_{i+1})$. The following notations are introduced for this algorithm :

- for all n , the values at times t^n and t^{n+1} (with $t^{n+1} = t^n + \Delta t$, Δt corresponding to the largest time step) are denoted by y^n and y^{n+1} ,
- for all n , the k -th intermediate value between times t^n and t^{n+1} is denoted by $y^{n,k}$.
- for all n , $F_{i+\frac{1}{2}}^n = F_{i+\frac{1}{2}}(y_i^n, y_{i+1}^n)$,
- for all n , the k -th intermediate flux at the interface $x_{i+\frac{1}{2}}$ is given by : $F_{i+\frac{1}{2}}^{n,k} = F_{i+\frac{1}{2}}(y_i^{n,k}, y_{i+1}^{n,k})$.

FÉVRIER 2020

step	Solution on cell i_0	Solution on cell i_{0+1}
1	$y_{i_0}^{n,1} = y_{i_0}^n - \frac{\Delta t}{2\Delta x_{i_0}}(F_{i_0+\frac{1}{2}}^n - F_{i_0-\frac{1}{2}}^n)$	$y_{i_{0+1}}^{n,1} = y_{i_{0+1}}^n - \frac{\Delta t}{2\Delta x_{i_{0+1}}}(F_{i_0+\frac{3}{2}}^n - F_{i_0+\frac{1}{2}}^n)$
2	$y_{i_0}^{n,2} = y_{i_0}^{n,1}$	$y_{i_{0+1}}^{n,2} = y_{i_{0+1}}^n - \frac{\Delta t}{4\Delta x_{i_{0+1}}}(F_{i_0+\frac{3}{2}}^n + F_{i_0+\frac{1}{2}}^{n,1} - F_{i_0+\frac{1}{2}}^n - F_{i_0+\frac{1}{2}}^{n,1})$
3	$y_{i_0}^{n,3} = y_{i_0}^n - \frac{\Delta t}{\Delta x_{i_0}}(F_{i_0+\frac{1}{2}}^n - F_{i_0-\frac{1}{2}}^n)$	$y_{i_{0+1}}^{n,3} = y_{i_{0+1}}^{n,2} - \frac{\Delta t}{2\Delta x_{i_{0+1}}}(F_{i_0+\frac{3}{2}}^{n,2} - F_{i_0+\frac{1}{2}}^{n,2})$
4	$y_{i_0}^{n+1} = y_{i_0}^n - \frac{\Delta t}{2\Delta x_{i_0}}(F_{i_0+\frac{1}{2}}^n + F_{i_0+\frac{1}{2}}^{n,1} - F_{i_0-\frac{1}{2}}^n - F_{i_0-\frac{1}{2}}^{n,1})$	$y_{i_{0+1}}^{n+1} = y_{i_{0+1}}^{n,3} - \frac{\Delta t}{4\Delta x_{i_{0+1}}}(F_{i_0+\frac{3}{2}}^{n,2} + F_{i_0+\frac{3}{2}}^{n,3} - F_{i_0+\frac{1}{2}}^{n,2} - F_{i_0+\frac{1}{2}}^{n,3})$

Tableau 4.15 –: RKS and RKF methods applied on cells i_0 and i_{0+1}

We analyze the expression of $y_{i_0}^{n+1}$. We notice that, between t^n and $t^n + \Delta t$, the total flux computed with the RKS method through the interface $x_{i_0+\frac{1}{2}}$ is :

$$tf_{i_0+\frac{1}{2}}^{RKS} = \frac{1}{2}[F_{i_0+\frac{1}{2}}^n + F_{i_0+\frac{1}{2}}^{[n,1]}] \quad (4.14)$$

The same analysis can be made on the other side of the interface. The total flux computed with the RKF method through the interface $x_{i_0+\frac{1}{2}}$ is :

$$tf_{i_0+\frac{1}{2}}^{RKF} = \frac{1}{4}[F_{i_0+\frac{1}{2}}^n + F_{i_0+\frac{1}{2}}^{[n,1]} + F_{i_0+\frac{1}{2}}^{[n,2]} + F_{i_0+\frac{1}{2}}^{[n,3]}] \quad (4.15)$$

It is clear that total fluxes $tf_{i_0+\frac{1}{2}}^{RKS}$ and $tf_{i_0+\frac{1}{2}}^{RKF}$ are different and consequently the local time stepping scheme is not mass-conservative at the interface between the subdomains.

We propose a correction step to make the scheme conservative. Following the notations given above, $y_{i_0}^{n+1}$ reads :

$$y_{i_0}^{n+1} = y_{i_0}^n - \frac{\Delta t}{\Delta x_{i_0}}tf_{i_0+\frac{1}{2}}^{RKS} + \frac{\Delta t}{2\Delta x_{i_0}}(F_{i_0-\frac{1}{2}}^n + F_{i_0-\frac{1}{2}}^{n,1}) \quad (4.16)$$

This value is corrected by adding the total flux difference. This leads to the following expression for the corrected value $y_{i_0}^{n+1,c}$:

$$y_{i_0}^{n+1,c} = y_{i_0}^{n+1} + \frac{\Delta t}{\Delta x_{i_0}}tf_{i_0+\frac{1}{2}}^{RKS} - \frac{\Delta t}{\Delta x_{i_0}}tf_{i_0+\frac{1}{2}}^{RKF} \quad (4.17)$$

We substitute $y_{i_0}^{n+1}$ by its expression given by (4.16). The corrected value $y_{i_0}^{n+1,c}$ now reads :

$$y_{i_0}^{n+1,c} = y_{i_0}^n - \frac{\Delta t}{\Delta x_{i_0}}tf_{i_0+\frac{1}{2}}^{RKF} + \frac{\Delta t}{2\Delta x_{i_0}}(F_{i_0-\frac{1}{2}}^n + F_{i_0-\frac{1}{2}}^{n,1}) \quad (4.18)$$

In (4.18), the total flux computed by the RKS method through the interface $x_{i_0+\frac{1}{2}}$, $tf_{i_0+\frac{1}{2}}^{RKS}$, has been replaced by the total flux computed by the RKF method through the interface $x_{i_0+\frac{1}{2}}$, $tf_{i_0+\frac{1}{2}}^{RKF}$. Consequently, the corrected scheme is mass conservative.

Time order and spatial order studies made on the different test cases in section 4.2.3 show that this correction step does not alter the time order of accuracy nor the spatial order of accuracy of the solution.

The algorithm of this local time stepping applied to our semi-discrete approximation (4.4) is shown in table 4.16. In this algorithm, the notations introduced in subsection 4.2.1.3 are re-used.

	Large-time-scale-subdomain		Short-time-scale-subdomain
1	$y_i^{n,3} = y_i^n - \Delta t D_i^n$	$y_i^{n,1} = y_i^n - \frac{\Delta t}{2} D_i^n$	$y_i^{n,1} = y_i^n - \frac{\Delta t}{2} D_i^n$
2		$y_i^{n,2} = y_i^{n,1}$	$y_i^{n,2} = y_i^n - \frac{\Delta t}{4} [D_i^n + D_i^{n,1}]$
3		$y_i^{n,3} = y_i^n - \Delta t D_i^n$	$y_i^{n,3} = y_i^{n,2} - \frac{\Delta t}{2} D_i^{n,2}$
4	$y_i^{n+1} = y_i^n - \frac{\Delta t}{2} [D_i^n + D_i^{n,3}]$	$y_i^{n+1} = y_i^n - \frac{\Delta t}{2} [D_i^n + D_i^{n,3}]$	$y_i^{n+1} = y_i^{n,2} - \frac{\Delta t}{4} [D_i^{n,2} + D_i^{n,3}]$
5		Correction step (last cell)	

Buffer zone

Tableau 4.16 –: Algorithm of the first proposed scheme (second order time accurate)

In our applications, with a four cell stencil flux function, the size of the buffer zone is equal to two cells, while it is equal to four cells for the original scheme of Constantinescu and Sandu. Moreover, only two flux evaluations (D_i^n and D_i^3) are needed in the buffer zone whereas the original scheme of Constantinescu and Sandu requires four flux evaluations in this zone.

However, unlike Constantinescu and Sandu scheme, our scheme requires an additional correction step to make it mass-conservative. Moreover, Constantinescu and Sandu schemes are TVB (Total Variation Bounded) if both temporal and spatial scheme are TVD (Total Variation Diminishing) [18]. We didn't study this property for this proposed scheme.

4.2.2.2. Second scheme : a third order time accurate scheme

The base method used to construct this local time stepping scheme is the RK3 low storage method described in table 4.9. The construction of RKF method is performed with two successive applications of the base method with the time step $\frac{\Delta t}{2}$. Consequently, RKF method has the following intermediate time coefficients : $c^F = \{0; \frac{c_2}{2}; \frac{c_3}{2}; \frac{1}{2}; \frac{1+c_2}{2}; \frac{1+c_3}{2}\}$. For the construction of RKS method, the base method is applied once with a time step Δt , which leads to the following intermediate time coefficients : $c^S = \{0; c_2 = \frac{1}{2}; c_3\}$. As for the first proposed scheme, we impose internal consistency by the relations $c_i^S = c_i^F$, for every i . Indeed, this property implies that some of the third order coupling conditions of table 4.3 are necessarily satisfied (namely that of lines 1,4,5). To satisfy internal consistency, an additional value at time $t + c_3 \Delta t$ must be calculated in RKF method. In RKS method, additional values at times $t + \frac{c_2}{2} \Delta t$, $t + \frac{c_3}{2} \Delta t$, $t + \frac{1+c_2}{2} \Delta t$ and $t + \frac{1+c_3}{2} \Delta t$ must be computed. RKS and RKF methods are now two third order Runge-Kutta methods with 7 stages. Their intermediate time coefficients are the same and read : $\{0; \frac{c_2}{2}; \frac{c_3}{2}; \frac{1}{2}; \frac{1+c_2}{2}; c_3; \frac{1+c_3}{2}\}$. The Runge-Kutta coefficients for the calculation of additional values, that are still to be determined, are denoted by α^S in RKS method and by α^F in RKF method. The Butcher tableaus of RKS and RKF methods are shown in tables 4.17 and 4.18, respectively.

FÉVRIER 2020

0	0						
$\frac{c_2}{2}$	α_1^S						
$\frac{c_3}{2}$	α_2^S	α_3^S					
$\frac{1}{2}$	a_{21}	0	0				
$\frac{1+c_2}{2}$	α_4^S	α_5^S	α_6^S	α_7^S			
c_3	a_{31}	0	0	a_{32}	0		
$\frac{1+c_3}{2}$	α_8^S	α_9^S	α_{10}^S	α_{11}^S	α_{12}^S	α_{13}^S	
	b_1	0	0	b_2	0	b_3	0

Tableau 4.17 –: RKS method

0	0								
$\frac{c_2}{2}$	$\frac{a_{21}}{2}$								
$\frac{c_3}{2}$	$\frac{a_{31}}{2}$	$\frac{a_{32}}{2}$							
$\frac{1}{2}$	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$						
$\frac{1+c_2}{2}$	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$	$\frac{a_{21}}{2}$					
c_3	α_1^F	α_2^F	α_3^F	α_4^F	α_5^F				
$\frac{1+c_3}{2}$	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$	$\frac{a_{31}}{2}$	$\frac{a_{32}}{2}$	0			
	$\frac{b_1}{2}$	$\frac{b_2}{2}$	$\frac{b_3}{2}$	$\frac{b_1}{2}$	$\frac{b_2}{2}$	0	$\frac{b_3}{2}$		

Tableau 4.18 –: RKF method

The unknown coefficients α_i^S and α_i^F will be determined so that the coupling conditions of lines 2 and 3 of table 4.3 are satisfied. Some of them are equivalent, therefore the independent coupling conditions read :

$$\sum_{i=1}^s \sum_{j=1}^s b_i^S a_{ij}^F c_j^F = \frac{1}{6}, \quad (4.19)$$

$$\sum_{i=1}^s \sum_{j=1}^s b_i^F a_{ij}^S c_j^S = \frac{1}{6}, \quad (4.20)$$

The coefficients for the computation of additional values in RKS method have to satisfy (4.20) and four relations given by the consistency property (4.10). This gives 5 relations that can be used to determine 5 unknowns only. The other eight coefficients can be set to any arbitrary value. A simple and natural choice is to set : $\alpha_3^S = \alpha_5^S = \alpha_6^S = \alpha_7^S = \alpha_9^S = \alpha_{10}^S = \alpha_{12}^S = \alpha_{13}^S = 0$, while the resolution of the system leads to the following values for the other coefficients :

$$\begin{cases} \alpha_1^S = \frac{c_2}{2} \\ \alpha_2^S = \frac{c_3}{2} \\ \alpha_4^S = \frac{1+c_2}{2} \\ \alpha_{11}^S = \frac{2}{3b_3} \approx 1.821 \\ \alpha_8^S = \frac{1+c_3}{2} - \frac{2}{3b_3} \approx -0.9270 \end{cases} \quad (4.21)$$

A similar analysis leads to the following values for the coefficients dedicated to the calculation of additional values in RKF method :

$$\begin{cases} \alpha_2^F = \left(\frac{2}{3} - \frac{b_2}{2}\right) \frac{1}{b_3} \approx 1.2440 \\ \alpha_1^F = c_3 - \alpha_2^F \approx -0.45534 \\ \alpha_3^F = 0 \\ \alpha_4^F = 0 \\ \alpha_5^F = 0 \end{cases} \quad (4.22)$$

As for our previous second order scheme, the local time stepping scheme defined by RKS and RKF methods described in tables 4.17 and 4.18 is not mass conservative. The strategy based on the Total Flux Lost and the Total Flux Received at the interface between subdomains can be applied to make the scheme conservative. The algorithm of this local time stepping scheme applied to solve (4.4) is shown in table 4.19. In this algorithm, the notations introduced in subsection 4.2.1.3 are re-used.

	Large-time-scale-subdomain	Short-time-scale-subdomain
1	$y_i^{n,1} = y_i^n - \alpha_1^L \Delta t D_i^n$	$y_i^{n,1} = y_i^n - \frac{a_{21}\Delta t}{2} D_i^n$
2	$y_i^{n,2} = y_i^n - \alpha_2^L \Delta t D_i^n$	$y_i^{n,2} = y_i^n - \frac{a_{31}\Delta t}{2} D_i^n - \frac{a_{32}\Delta t}{2} D_i^{n,1}$
3	$y_i^{n,3} = y_i^n - a_{21}\Delta t D_i^n$	$y_i^{n,3} = y_i^n - \frac{b_1\Delta t}{2} D_i^n - \frac{b_2\Delta t}{2} D_i^{n,1} - \frac{b_3\Delta t}{2} D_i^{n,2}$
4	$y_i^{n,4} = y_i^n - \alpha_4^L \Delta t D_i^n$	$y_i^{n,4} = y_i^{n,3} - \frac{a_{21}\Delta t}{2} D_i^{n,3}$
5	$y_i^{n,5} = y_i^n - a_{31}\Delta t D_i^n - a_{32}\Delta t D_i^{n,3}$	$y_i^{n,5} = y_i^n - \alpha_1^S \Delta t D_i^n - \alpha_2^S \Delta t D_i^{n,1}$
6	$y_i^{n,6} = y_i^n - \alpha_8^L \Delta t D_i^n - \alpha_{11}^L \Delta t D_i^4$	$y_i^{n,6} = y_i^{n,3} - \frac{a_{31}\Delta t}{2} D_i^{n,3} - \frac{a_{32}\Delta t}{2} D_i^4$
7	$y_i^{n+1} = y_i^n - b_1\Delta t D_i^n - b_2\Delta t D_i^{n,3} - b_3\Delta t D_i^{n,5}$	$y_i^{n+1} = y_i^{n,3} - \frac{b_1\Delta t}{2} D_i^{n,3} - \frac{b_2\Delta t}{2} D_i^{n,4} - \frac{b_3\Delta t}{2} D_i^{n,6}$
8	Correction step	
	Buffer zone	Buffer zone

Tableau 4.19 –: Algorithm of the second proposed scheme (third order time accurate)

This algorithm is written in the Runge-Kutta "classical form" but it can be implemented in the "low-storage form", since these forms are equivalent. In this scheme, there are two buffer zones because both RKS and RKF methods require one or several additional values. The size of each buffer zone is equal to two cells in our applications (flux function with a four cell stencil). We remind that for the scheme of Constantinescu and Sandu based on RK3, there is one buffer zone of six cells. However, the scheme of Constantinescu and Sandu based on RK3 does not require any correction step since it is mass conservative. Moreover, Constantinescu and Sandu schemes are TVB (Total Variation Bounded) if both temporal and spatial scheme are TVD (Total Variation Diminishing) [18]. We didn't study this property for this proposed scheme.

4.2.3. Numerical tests

In this section, some numerical test cases are performed to compare the new proposed schemes and the schemes proposed in the literature that are based on the strategy of Constantinescu and Sandu. The schemes based on RK2 and RK3 which follow the strategy of Constantinescu and Sandu are denoted by CSRK2 and CSRK3, respectively. The schemes that we propose, based on RK2 and RK3, are denoted by Prop. RK2 and Prop. RK3, respectively.

Two different schemes are used for the Euler flux discretization. The Roe scheme [34] with a minmod limiter [35] is retained for the Sod tube, whereas an hybrid centered/upwind version of the scheme AUSM+(P) proposed by Mary and Sagaut [36] is employed in the others test cases. Viscous fluxes are discretized with a classical second order scheme.

4.2.3.1. Two-dimensional vortex advection

This test case is based on 2D compressible Euler equations for a perfect gas :

FÉVRIER 2020

$$\begin{cases} \partial_t \rho + \partial_{x_j}(\rho u_j) = 0 \\ \partial_t \rho u_i + \partial_{x_j}(\rho u_i u_j + p \delta_{ij}) = 0 \\ \partial_t(\rho e) + \partial_{x_j}(\rho e u_j + p u_j) = 0 \end{cases} \quad (4.23)$$

with $1 \leq i, j \leq 2$. The variables ρ , u_1 , u_2 , p are the fluid density, the fluid velocity in directions \vec{x} and \vec{y} and the fluid pressure, respectively. Total energy is denoted by e and reads : $e = \frac{1}{\gamma-1} \frac{p}{\rho} + \frac{u_1^2 + u_2^2}{2}$, with $\gamma = 1.40$.

The initial state is a 2D uniform flow ($\rho = 1$, $u_1 = 1$, $u_2 = 0$, $p = 1$) with the superposition of perturbations $(\delta\rho, \delta u_1, \delta u_2, \delta p)$ which simulate a vortex. The perturbations read :

$$\begin{cases} \delta\rho = (1 + \delta T)^{\frac{1}{\gamma-1}} \\ \delta u_1 = \frac{\epsilon}{2\pi} e^{0.5(1-r^2)}(y_c - y) \\ \delta u_2 = \frac{\epsilon}{2\pi} e^{0.5(1-r^2)}(x - x_c) \\ \delta p = (1 + \delta T)^{\frac{\gamma}{\gamma-1}} \end{cases} \quad (4.24)$$

where $\delta T = -\frac{(\gamma-1)\epsilon^2}{8\gamma\pi^2} e^{1-r^2}$, is the temperature perturbation around the value $T = 1$. Note that the corresponding initial value and perturbation for the entropy are $S = 1$ and $\delta S = 0$, respectively. The coordinates (x_c, y_c) are the coordinates of the the vortex center, $r = \sqrt{(x - x_c)^2 + (y - y_c)^2}$ is the vortex radius and $\epsilon = 5$ is the vortex strength. The initial solution of this problem reads :

$$\begin{cases} \rho^0 = 1 + \delta\rho \\ u_1^0 = 1 + \delta u_1 \\ u_2^0 = 0 + \delta u_2 \\ p^0 = 1 + \delta p \end{cases} \quad (4.25)$$

The solution of this problem is a simple advection of the vortex in the \vec{x} direction. For more information about this test case, see [37].

The computational domain is rectangular (dimensions : 30×20) with periodic boundary conditions. It is divided into three zones. In the first and last zones, the grid spacings in directions \vec{x} and \vec{y} are $\Delta x = \Delta y = 0.05$. In the middle zone, the grid spacing in the \vec{x} direction is refined by a factor 2 ($\Delta x = 0.025$ and $\Delta y = 0.05$). Thus, the time step $\frac{\Delta t}{2}$ is used in this zone and the time step Δt is applied in the first and last zones. The mesh and the density field at initial time are shown in figure 4.2, while the density profile at initial time and $y = 10$ is plotted in Figure 4.3. The reference solution mentioned in figures 4.4 and 4.5 is obtained with a fourth order Runge-Kutta scheme (RK4) with $\Delta t = 0.0035$. The CFL number in the middle zone (the maximum CFL number over the mesh) is $CFL = 0.28$. For each local time stepping scheme, the time step used in the first and last zones (Δt) is the maximum allowable time step shown in the row " $CFL_{max} (\Delta t_{max})$ " of table 4.20.

Figures 4.4 and 4.5 show the density profile along $y = 10$ for each local time stepping scheme and for the reference solution at the final time 43.5. At this time, the vortex has returned approximately to its initial position (we remind that we are using periodic boundary conditions). A very good agreement between each scheme and the reference solution is observed.

Some characteristics studied at the final time are listed in table 4.20. The "speedup" refers to the relative difference between the CPU time of an explicit global time stepping scheme based on RK2 (which uses

a uniform time step satisfying the most restrictive CFL condition) and the CPU time of the local time stepping schemes. The row "Conservation defect" is the relative difference between the total fluid mass in the computational domain at final time $m(t_f)$ and at initial time $m(t_0)$:

$$\text{Conservation defect} = \frac{m(t_f) - m(t_0)}{m(t_0)}$$

The local time stepping schemes based on RK3 have a better stability than the local time stepping schemes based on RK2, which allows to use larger time steps. It is the reason why the local time stepping schemes based on RK3 (CS RK3 and Prop. RK3) provide a better speedup than the local time stepping schemes based on RK2 (CS RK2 and Prop. RK2). The conservation defects obtained with our local time stepping schemes are similar to the ones obtained with the schemes of Constantinescu that are conservative. It shows that our flux-correction strategy to make our schemes conservative is efficient.

Figure 4.6 shows the results of a time order study. For each local time stepping scheme, a finest solution is computed with the time step $\Delta t = 10^{-4}$ in the first and last zones. We remind that in the middle zone the grid spacing and the time step are refined by a factor 2, therefore the CFL number is uniform over the mesh ($CFL = 0.008$). Three other simulations, with time steps equal to $2\Delta t$, $4\Delta t$ and $8\Delta t$ for the first and last zones are also run. The L_2 errors on the density field between the finest solution and the three other solutions are computed. The slope of each convergence curve is in agreement with the theoretical temporal order of all the local time stepping schemes. We can notice that the L_2 errors of the original scheme of Constantinescu and Sandu and our proposed scheme based on RK2 are identical and show second order accuracy. For the schemes based on RK3, our proposed approach is really third order accurate, while CS RK3 is only a second-order accurate scheme leading to a higher level of L_2 error. 4.7 shows the results of a space order study, for each local time stepping scheme. For this study, the spatial steps Δx and Δy are uniform (no space refinement in the middle zone). There is only a temporal refinement in the middle zone. The finest solution is computed with $\Delta x = \Delta y = 0.013$. The time step is taken sufficiently small in order to compute only the spatial error. The time step $\Delta t = 9.10^{-4}$ ($CFL = 0.1$) is used in the first and last zones, while the time step $\Delta t = 4,5.10^{-4}$ is used in the middle zone. Three other solutions are computed keeping the same time steps and with the spacial steps $(2\Delta x, 2\Delta y)$, $(3\Delta x, 3\Delta y)$, $(4\Delta x, 4\Delta y)$. The L_2 errors on the density field between the finest solution and the three other solutions are computed. The results show that the four local time stepping schemes have the same spatial order of accuracy. This order of accuracy is close to 2, the theoretical spatial order of accuracy for the AUSM+P scheme. Figures 4.6 and 4.7 demonstrate that our flux correction stage used in schemes "Prop. RK2" and "Prop. RK3" does not alter the time order of accuracy nor the spatial order of accuracy.

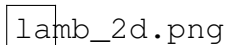
 lamb_2d.png

Fig. 4.2 –: Density field at initial time and view of the mesh partitioning for the local time stepping computation.


 courbes.png

Fig. 4.3 –: Initial density profile at $y = 10$

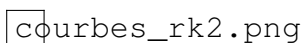
 courbes_rk2.png

Fig. 4.4 –: Density profiles at $y = 10$ and final time for the schemes Prop.RK2 and CS RK2 and the reference solution

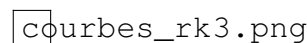
 courbes_rk3.png

Fig. 4.5 –: Density profiles at $y = 10$ and final time for Prop.RK3, CS RK3 schemes and the reference solution

FÉVRIER 2020

ordre_cas2.png

ordre_spatial_tourbi

Fig. 4.6 –: L_2 error as a function of the time step Δt for each local time stepping scheme

Fig. 4.7 –: L_2 error as a function of the spatial step Δx for each local time stepping scheme

	$CFL_{max}(\Delta t_{max})$	Speedup as compared to a RK2 global time stepping scheme	Conservation defect
CS RK2	0.25 ($\Delta t_{max} = 0.007$)	21%	$3.4 \cdot 10^{-16}$
Prop. RK2	0.25 ($\Delta t_{max} = 0.007$)	25%	$1.7 \cdot 10^{-16}$
CS RK3	1.1 ($\Delta t_{max} = 0.029$)	73%	$1.7 \cdot 10^{-16}$
Prop. RK3	1.1 ($\Delta t_{max} = 0.029$)	74%	$9.1 \cdot 10^{-16}$

Tableau 4.20 –: Comparison of efficiency and conservation property indicators for the local time stepping schemes

4.2.3.2. Sod shock tube

The Sod tube problem is used to test if the proposed schemes are able to predict accurate and stable solution for a flow containing discontinuities. Note that this 1D problem is solved here in 2D. At the initial time the velocity field is equal to zero, while the density and pressure profiles are given by :

- $\rho^0(x, y) = 1, p^0(x, y) = 1$ for $x \leq 9$,
- $\rho^0(x, y) = 0.125, p^0(x, y) = 0.1$ for $x > 9$,

The rectangular computational domain (18×0.6) is divided into three zones of length 6. In the first and last zones, the grid spacings in the \vec{x} direction (Δx) and in the \vec{y} direction (Δy) are : $\Delta x = \Delta y = 0.01$. In the middle zone, $\Delta x = 0.005$ and $\Delta y = 0.01$. Thus, the time step $\frac{\Delta t}{2}$ is used in the middle zone and the time step Δt is applied in the first and last zones. The mesh partitioning and the initial density profile at $y = 0.3$ are plotted in figure (4.8).

Figures 4.9 and 4.10 show the density profile at $y = 0.3$ for each local time stepping scheme and for the reference solution at the final time, $t = 4.8$. The reference solution mentioned in these figures is obtained by using the classical RK4 explicit scheme with the time step $\Delta t = 5 \cdot 10^{-4}$. The CFL number in the middle zone (the maximum CFL number over the mesh) is $CFL = 0.2$. The time step Δt of the local time stepping schemes (the time step in the first and last zones) is their maximal allowable time step shown in table 4.21 below. Figures 4.9 and 4.10 show a very good agreement between the reference solution and the solutions obtained with the local time stepping schemes. Table 4.21 also shows the speedup as compared to a RK2 global time stepping scheme, which uses a uniform time step satisfying the most restrictive CFL condition. Our local time stepping schemes have roughly the same speedup as their homologous schemes of Constantinescu and Sandu based on the same RK method.

The results of a time order study are plotted in figure 4.11. For each local time stepping scheme, a finest solution is computed with the time step $\Delta t = 2.5 \cdot 10^{-4}$ in the first and last zones. We remind that in the middle zone the grid spacing and the time step are refined by a factor 2, therefore the CFL number is uniform over the mesh ($CFL = 0.05$). Three other simulations, with time steps equal to $2\Delta t$, $4\Delta t$ and $8\Delta t$ for the first and last zones are also run. The L_2 errors committed on the density field between the finest solution and the three other solutions are computed. The slope of each convergence curve is in agreement with the theoretical

temporal order of the local time stepping schemes. Our local time stepping scheme "Prop. RK3" is third-order accurate and leads to the lowest error. Figure 4.12 shows the results of a space order study, for each local time stepping scheme. For this study, the spatial steps Δx and Δy are uniform (no space refinement in the middle zone). There is only a temporal refinement in the middle zone. The finest solution is computed with $\Delta x = \Delta y = 0.0057$. The time step is taken sufficiently small in order to compute only the spatial error. The time step $\Delta t = 5.10^{-4}$ ($CFL = 0.1$) is used in the first and last zones, while the time step $\Delta t = 2, 5.10^{-4}$ is used in the middle zone. Three other solutions are computed keeping the same time steps and with the spacial steps $(2\Delta x, 2\Delta y)$, $(3\Delta x, 3\Delta y)$, $(4\Delta x, 4\Delta y)$. The L_2 errors on the density field between the finest solution and the three other solutions are computed. The results show that the four local time stepping schemes have the same spatial order of accuracy. This order of accuracy is slightly superior to 1. This result is logical since the Roe scheme uses a slope limiter which reduces the order of accuracy at discontinuities. Figures 4.11 and 4.12 demonstrate that our flux correction stage used in schemes "Prop. RK2" and "Prop. RK3" does not alter the time order of accuracy nor the spatial order of accuracy.

choc_initial.png

Fig. 4.8 –: Initial density profile and view of the mesh partitioning for local time stepping scheme.

rk2.png

rk3.png

Fig. 4.9 –: Density profile at the final time for the CS RK2 and Prop. RK2 schemes

Fig. 4.10 –: Density profile at the final time for the CS RK3 and Prop. RK3 schemes.

	CFLmax (Δt_{max})	Speedup as compared to a RK2 global time stepping scheme
CS RK2	0.7 ($\Delta t_{max} = 0.0032$)	19%
Prop. RK2	0.7 ($\Delta t_{max} = 0.0032$)	19%
CS RK3	0.9 ($\Delta t_{max} = 0.004$)	28%
Prop. RK3	0.9 ($\Delta t_{max} = 0.004$)	26%

Tableau 4.21 –: Comparison of the maximum CFL and the speedup of the local time stepping schemes for the Sod shock tube.

ordre_choc.png

ordre_spatial_choc.png

Fig. 4.11 –: L_2 error as a function of the time step Δt for each local time stepping scheme

Fig. 4.12 –: L_2 error as a function of the spatial step Δx for each local time stepping scheme

4.2.3.3. Taylor-Green Vortex

This test case is used to validate the proposed schemes in the context of DNS or LES simulation of turbulent flows. At the initial time, large vortices are present in the computational box and during the simulation an energy transfer occurs from the large scales to smaller ones. When the energy cascade is sufficient to reach the Kolmogorov scale, the kinetic energy is dissipated. For that case, the compressible 3D Navier-Stokes

FÉVRIER 2020

equations are solved for a viscous fluid :

$$\begin{cases} \partial_t \rho + \partial_{x_j}(\rho u_j) = 0 \\ \partial_t \rho + \partial_{x_j}(\rho u_i u_j + p \delta_{ij} - \tau_{ij}) = 0 \\ \partial_t(\rho e) + \partial_{x_j}(\rho e u_j + p u_j - \tau_{jk} u_k + q_j) = 0 \end{cases} \quad (4.26)$$

with $1 \leq i, j \leq 3$. The variable ρ is the fluid density, u_i are the fluid velocity components, e is the total specific energy of the fluid, p is the fluid pressure (ideal gas law), q_j is the heat flux, and τ_{ij} is the tensor of viscous constraints.

The computational domain is a cube of dimension $2\pi L_0$ ($L_0 = 1$ is a reference length), with periodic boundary conditions. The initial conditions for variables u_i , p and ρ read :

$$\begin{cases} u_1^0 = U_0 \sin(x/L_0) \cos(y/L_0) \cos(z/L_0) \\ u_2^0 = -U_0 \cos(x/L_0) \sin(y/L_0) \cos(z/L_0) \\ u_3^0 = 0 \\ p^0 = P_0 + \rho_0 U_0^2 / 16 (\cos(x/L_0) + \cos(y/L_0)) (\cos(2z/L_0) + 2) \\ \rho^0 = p / (RT_0) \end{cases} \quad (4.27)$$

where $P_0 = 101183 Pa$, $\rho_0 = 1.2 kg.m^{-3}$, $T_0 = 294 K$ and $U_0 = 34.38 m.s^{-1}$. The simulation is performed during a period of $14L_0/U_0$.

The computational domain is divided into three zones of equal length, as shown in figure 4.13. In the first and last zones, the grid spacings in the \vec{x} direction (Δx), \vec{y} direction (Δy) and \vec{z} direction (Δz) are $\Delta x = \Delta y = \Delta z = 0.025$, respectively. In the middle zone, the mesh is refined by a factor two in the \vec{x} direction ($\Delta x = 0.0125$, $\Delta y = \Delta z = 0.025$). In this zone, the time step $\frac{\Delta t}{2}$ is used while the time step Δt is applied in the other zones. For each scheme, we compute at each time step the enstrophy ϵ , defined by

$$\epsilon = \frac{L_0}{U_0^2 \rho_0} \int_{\Omega} \rho \vec{w} \cdot \vec{w} d\Omega, \quad (4.28)$$

where \vec{w} denotes the vorticity, defined by : $\vec{w} = \vec{\nabla} \wedge \vec{V}$, with $\vec{V} = (u_1, u_2, u_3)$. The enstrophy is known to be highly sensitive to the numerical method [38].

Figures 4.14 and 4.15 show the time evolution of the enstrophy for each scheme. The reference solution mentioned in these figures is obtained with a classical RK4 explicit scheme used with the time step $\Delta t = 0.00015$. The CFL number in the middle zone (the maximum CFL number over the mesh) is $CFL = 0.03$. The time step Δt in the first and last zones of each local time stepping scheme is the maximal allowable time step, shown in the first row of table 4.22. A very good agreement is observed between the reference solution and the solutions obtained with the local time stepping schemes.

Table 4.22 also shows the speedup obtained with the local time stepping schemes as compared to an RK2 explicit global time stepping scheme, and the conservation defects. Our local time stepping schemes provide the same speedup as their homologous schemes based on the strategy of Constantinescu and Sandu. The conservation defects of our schemes are very close to the ones obtained with the schemes of Constantinescu and Sandu. It confirms that our flux-correction strategy is efficient.

Finally, figure 4.16 shows the results of a time order study. For each local time stepping scheme, the "finest" solution is computed with the time step $\Delta t = 7.10^{-5}$ in the first and last zones. We remind that in the

middle zone the grid spacing and the time step are refined by a factor 2, therefore the CFL number is uniform over the mesh ($CFL = 0.007$). In this figure, each mark is associated to the use of the time steps : $2\Delta t$, $4\Delta t$ and $8\Delta t$, for the first and last zones. Once again the slope of each curve is in agreement with the theoretical temporal order of each local time stepping scheme. Our proposed scheme based on RK3 has the lowest L_2 error. Figure 4.17 shows the results of a space order study, for each local time stepping scheme. For this study, the spatial steps Δx and Δy are uniform (no space refinement in the middle zone). There is only a temporal refinement in the middle zone. The finest solution is computed with $\Delta x = \Delta y = 0.022$. The time step is taken sufficiently small in order to compute only the spatial error. The time step $\Delta t = 15.10^{-4}$ ($CFL = 0.1$) is used in the first and last zones, while the time step $\Delta t = 7,5.10^{-4}$ is used in the middle zone. Three other solutions are computed keeping the same time steps and with the spacial steps $(2\Delta x, 2\Delta y)$, $(3\Delta x, 3\Delta y)$, $(4\Delta x, 4\Delta y)$. The L_2 errors on the density field between the finest solution and the three other solutions are computed. The results show that the four local time stepping schemes have the same spatial order of accuracy. This order of accuracy is slightly superior to 2, the theoretical spatial order of accuracy for the AUSM+P scheme. Figures 4.16 and 4.17 demonstrate that our flux correction stage used in schemes "Prop. RK2" and "Prop. RK3" does not alter the time order of accuracy nor the spatial order of accuracy.

tg_mv_maillage.png

Fig. 4.13 –: Mesh used for the Taylor-Green Vortex test case

tg_mv_rk2.png

tg_mv_rk3.png

Fig. 4.14 –: Enstrophy obtained with the schemes "CS RK2" and "Prop. RK2" Fig. 4.15 –: Enstrophy obtained with the schemes "CS RK3" and "Prop. RK3"

	$CFL_{max}(\Delta t_{max})$	Speedup as compared to a RK2 global time stepping scheme	Conservation defect
CS RK2	0.3 ($\Delta t_{max} = 0.0006$)	20%	$1,8.10^{-15}$
Prop. RK2	0.3 ($\Delta t_{max} = 0.0006$)	27%	$2,2.10^{-15}$
CS RK3	0.9 ($\Delta t_{max} = 0.0019$)	57%	$2,5.10^{-15}$
Prop. RK3	0.9 ($\Delta t_{max} = 0.0019$)	59%	$3,8.10^{-15}$

Tableau 4.22 –: Taylor-Green Vortex : comparison of some indicators for the different local time stepping schemes

tg_mv_ordre.png

ordre_spatial_tgv.png

Fig. 4.16 –: L_2 error as a function of the time step Δt for each local time stepping scheme Fig. 4.17 –: L_2 error as a function of the space step Δx for each local time stepping scheme

4.2.3.4. Turbulent flow over an airfoil

This test case is chosen to assess the interest of the local time stepping method for a flow configuration representative of current challenging configurations in the aerospace context. For that purpose a Large Eddy

FÉVRIER 2020

Simulation (LES) over a SD7003 profile is performed. Due to the angle of attack and the Reynolds number, a laminar-turbulent transition occurs in a laminar separation bubble (LSB). It is followed by a turbulent reattachment and a classical turbulent boundary layer subjected to an adverse pressure gradient. As this flow configuration is very sensitive to physical perturbations due to the laminar-turbulent transition process [39], it represents a [challenging case for the validation] relevant test case to highlight possible unwanted numerical perturbations introduced by our proposed temporal scheme. This test case is based on 3D Navier-Stokes equations (4.26). The Mach number, M , and the Reynolds number Re based on the airfoil chord c and the infinite velocity U_∞ are set to $M = 0.1$ and $Re = 60000$, respectively. The angle of attack α is set to $\alpha = 8^\circ$. This Reynolds number is not so challenging for nowadays simulations, but it is sufficient to test the numerical method with a reasonable CPU cost for a comparison between different schemes.

The computational domain is a 3D structured mesh with approximately 90.10^6 cells. To trigger the transition process in a deterministic way, the perturbation proposed in [39] is imposed on the suction side at $x/c = 0.03$ with an amplitude $A_v = 0.001U_\infty$. Figure 4.18 shows a mesh section in the (\vec{x}, \vec{y}) map with a reduced number of points (one point over 10). An extruded 2D RANS solution is used to initialize the 3D flow solution. No reliable experimental data are available to validate the simulation since the size of the LSB is strongly influenced by the level of the freestream turbulence and also by sidewall effects of the windtunnel. For a computational cost reason, taking into account properly these phenomena is out of the scope of the present simulation. The present wall mesh resolution, $\Delta x^+ \approx 25$, $\Delta y^+ \approx 1$, $\Delta z^+ \approx 8$, is finer than that used in [36] and equivalent to that used in [40, 41], in which grid converged results are presented for the same kind of flow : airfoil with laminar separation bubble and turbulent reattachment. As the numerical method employed in the present paper is the same as the one validated in [36, 42, 40, 41, 43, 44] thanks to windtunnel experiment comparisons, we are confident that our results obtained with the classical RK3 scheme and a global timestep are physically correct for the specific set of Reynolds number and boundary conditions. Therefore the solution obtained with the classical RK3 explicit scheme is considered as a reference to assess the accuracy and efficiency of the proposed temporal scheme.

Our most accurate local time stepping scheme "Prop. RK3" is used to compute this test case. It is compared to a classical explicit scheme (RK3) and one implicit scheme in terms of accuracy (see figures 4.23 - 4.28) and CPU cost (see table 4.24). The implicit scheme is based on the second order accurate Gear scheme [45] for the approximation of the time derivative. A self-adaptive Newton method is used to solve the non linear problem [43]. This method takes advantage of Newton convergence rate heterogeneities by using a spatially varying number of iterations for the Newton process. A user defined convergence criterion ϵ drives the spatially varying number of iterations needed to solve the Newton process. Two implicit simulations are performed with different values for the convergence criterion of the Newton process : $\epsilon = 0.1$, and $\epsilon = 0.06$ (see [43] for more information about this convergence criterion). [We consider that the reference solution of the problem is obtained with the classical explicit scheme (RK3).] Note that for this test case, all the schemes are implemented with parallel OpenMP directives and run on 64 threads on one Intel Xeon Phi 7230 node.

The initial 2D RANS solution is used for an automatic mesh partitioning based on the local value of the CFL number. Figure 4.19 shows a zoom on the airfoil to illustrate the spatial repartition of the different time steps for our local time stepping scheme. Only four time levels have been considered for this validation and the largest time step is such that $\Delta t \times U_\infty/c = 1.10^{-5}$. This table 4.23 shows the relative number of cells for each time step. This table 4.23 illustrates the interest of using a local time stepping scheme for this case. Indeed, it seems inefficient to restrict the time step for the whole domain to $\frac{\Delta t}{8}$, like in a classical explicit scheme, while only 4% cells require to use this time step.

The time step used for the classical RK3 scheme is such that $\Delta t \times U_\infty/c = 1.25.10^{-6}$ and the time

Time step	Δt	$\frac{\Delta t}{2}$	$\frac{\Delta t}{4}$	$\frac{\Delta t}{8}$
Relative number of cells (%)	61	23	12	4

Tableau 4.23 –: Relative number of cells for each time step

step of the implicit simulations is such that $\Delta t \times U_\infty / c = 6.10^{-5}$ (largest CFL number of roughly 30). The simulations with the different schemes are performed until the final time t_f such that $t_f \times U_\infty / c = 2$ is reached.

Figure 4.20 shows the iso-surface of the Q-criterion ($Q = 1.5$) coloured with the fluid velocity magnitude, obtained with our explicit local time stepping scheme at the final time t_f . The Q-criterion is an indicator of turbulent structures, defined by the formula :

$$Q = \frac{1}{2} \left[(tr(\nabla U))^2 - tr(\nabla U \cdot \nabla U) \right], \quad (4.29)$$

where $U = (u_1, u_2, u_3)$ is the fluid velocity vector. This figure shows that the flow is laminar at the leading edge, then a laminar-turbulent transition zone is observed. Finally, the flow is fully turbulent. Figures 4.21 and 4.22 show instantaneous isolines of the streamwise velocity component in the plane $z/c=0$. Figure 4.21 shows that the transition occurs in a laminar separation bubble, characterized by a fluid recirculation (negative streamwise velocity component). The turbulent structures created in the transition zone allow to reattach the fluid to the airfoil (reattachment zone). Figure 4.22 shows the location of the mid-chord zone, where the flow is fully turbulent. Pressure values are recorded at points 1 (end of the transition zone), 2 (reattachment zone) and 3 (mid-chord zone) to compute Power Spectral Density (PSD) spectrum. Figures 4.23 - 4.25 show the mean streamwise velocity profile $\langle u \rangle / u_e$ and two components of the Reynolds stress profiles ($\langle \rho u' u' \rangle / \rho u_e^2$ and $\langle \rho u' v' \rangle / \rho u_e^2$) for each scheme in the transition, reattachment and mid-chord zones. The velocity u_e refers to the streamwise velocity at the edge of the boundary layer. Pressure spectrums at points 1,2 and 3 are plotted in figures 4.26 - 4.28.

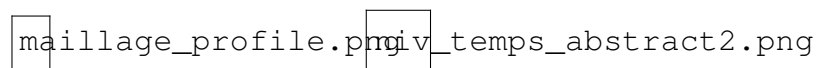


Fig. 4.18 –: 2D illustration of the structured mesh used for the computation of a flow over a SD7003 Wing. One point over 10 is shown.

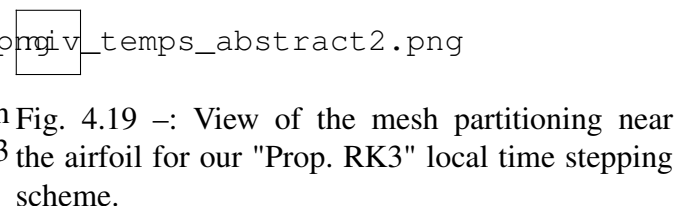


Fig. 4.19 –: View of the mesh partitioning near the airfoil for our "Prop. RK3" local time stepping scheme.

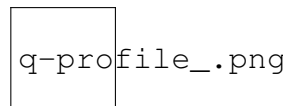


Fig. 4.20 –: Q-criterion iso-surface ($Q = 1.5$) coloured by the fluid velocity magnitude

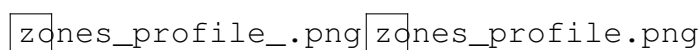


Fig. 4.21 –: Location of transition and reattachment zones with their pressure recording points

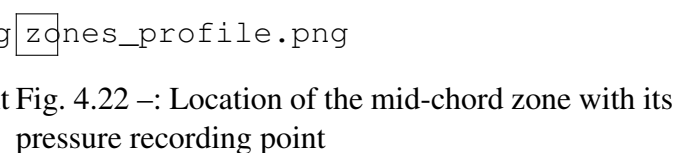


Fig. 4.22 –: Location of the mid-chord zone with its pressure recording point

FÉVRIER 2020

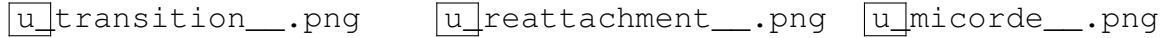


Fig. 4.23 –: Mean streamwise velocity profiles for each scheme in the transition, reattachment and mid-chord zones.

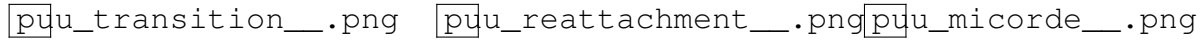


Fig. 4.24 –: Mean Reynolds stress profiles $\langle \rho u'u' \rangle / \rho u_e^2$ for each scheme in the transition, reattachment and mid-chord zones.



Fig. 4.25 –: Mean Reynolds stress profiles $\langle \rho u'v' \rangle / \rho u_e^2$ for each scheme in the transition, reattachment and mid-chord zones.

Figure 4.23 shows a perfect agreement between all schemes for the mean streamwise velocity profiles. The same perfect agreement is observed for the Reynolds stress profiles between the local time stepping scheme, the implicit scheme with $\epsilon = 0.06$ and the reference solution (global time stepping scheme). Small discrepancies between the reference solution and the implicit scheme run with $\epsilon = 0.1$ are present for the Reynolds stresses profiles. It shows that this value of ϵ is too large to prevent an interaction between the flow physics and the numerical error due to the incomplete convergence of the Newton process.



Fig. 4.26 –: PSD spectrum as a function of the Strouhal number St (based on U_∞) for each scheme in the transition zone (point 1)

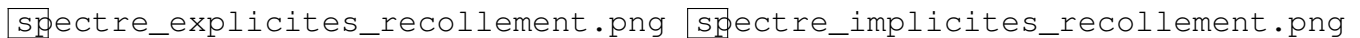


Fig. 4.27 –: PSD spectrum as a function of the Strouhal number St (based on U_∞) for each scheme in the reattachment zone (point 2)



Fig. 4.28 –: PSD spectrum as a function of the Strouhal number St (based on U_∞) for each scheme in the mid-chord zone (point 3)

No significant influence of the temporal scheme is observed on the pressure spectrums of figures 4.26 - 4.28. A slight offset between the implicit schemes spectrums and the explicit schemes spectrums is observed

for high Strouhal numbers ($St > 10^3$) at points (2) and (3). The implicit schemes seem slightly less accurate than the explicit schemes to compute high frequencies, certainly because of the use of a larger time step and the dissipative nature of the implicit Gear scheme.

Table 4.24 shows the CPU time obtained for each scheme and the time reduction as compared to the explicit global time stepping scheme. It appears that our local time stepping scheme "Prop. RK3" provides a significant CPU time reduction (69%), without altering the accuracy of the solution. Same order of CPU time reduction is obtained with the implicit scheme using $\epsilon = 0.1$ (73%), whereas the implicit simulation using $\epsilon = 0.06$ only provides a 47% CPU time reduction.

	Global time stepping (RK3)	Local time stepping (Prop. RK3)	Imp. scheme $\epsilon = 0.1$	Imp. scheme $\epsilon = 0.06$
CPU time (hours)	653	197	175	347
Time reduction (%)	0	69	73	47

Tableau 4.24 –: Temporal scheme influence on the CPU cost of the SD7003 airfoil simulation on a single Intel Xeon Phi 7230 node

This test case demonstrates that our local time stepping scheme "Prop RK3" is able to compute a challenging, industrial-like test case with a very good accuracy and for a competitive CPU time. In order to reach the accuracy of our local time stepping scheme, the implicit scheme convergence criterion ϵ has to be reduced from $\epsilon = 0.1$ to $\epsilon = 0.06$ (see figures 4.24 - 4.25). It leads to an important increase in the CPU time, which makes the implicit scheme less competitive than our explicit local time stepping scheme in this field.

4.2.4. Conclusion

The diversity of spatial scales in fluid mechanics problems leads to the use of several cell sizes in CFD meshes. It leads to a wide range of numerical stability conditions that must be satisfied by explicit time integration methods. Local time stepping schemes are well suited to these problems because the time step can vary across the mesh to satisfy several local stability conditions. It results in a computational speedup as compared to classical explicit schemes, where the time step for the entire domain is restricted by the most severe stability condition. However, the construction of local time stepping schemes requires a particular attention to ensure high order time accuracy and mass conservation.

Constantinescu and Sandu [18] have developed an interesting strategy which allows to construct mass conservative and second order accurate local time stepping schemes. Their strategy is based on the theory of Partitioned Runge-Kutta (PRK) methods. We also use the theory of PRK methods as well as the computation of consistent additional values at the interfaces to develop two new local time stepping schemes of order two and three. Our schemes are made conservative with an additional flux-correction step. The academic tests show that very similar results are obtained with our local time stepping schemes and the local time stepping schemes of [18]. Moreover, the tests confirm that our third order local time stepping scheme has a lower temporal error than our second order local time stepping scheme and than the local time stepping schemes of [18]. The final test case shows that our third order local time stepping scheme is able to compute complex, industrial-like test cases with a very good accuracy and for a CPU time close to the one obtained with our implicit scheme. Our strategy seems to be a good approach to construct efficient, accurate and conservative local time stepping schemes.

FÉVRIER 2020

We believe that our method will be more efficient than implicit schemes on a larger number of CPU cores, thanks to MPI or hybrid MPI/OpenMP parallelization. This requires a specific work on load balancing for our local time stepping scheme, like in [46] and [47]. This is a work in progress that will be the subject of a future publication.

FÉVRIER 2020

FÉVRIER 2020

5. INTERFACES

Cette tâche du PRF était décomposée en 5 parties :

- Les 3 premières devaient être portées par M. Poinot (DAAA/CLEF). Son départ en début de PRF pour Safran Tech et son profil particulier font que ces activités ont été abandonnées. Il s'agissait de mener une réflexion sur l'interface "idéale" de composant Python/CGNS pouvant traiter des physiques très variées (fluide, matériaux, ...). Il devait aussi poursuivre sa réflexion sur la manière de produire, documenter, valider des composants Python CGNS, ainsi qu'évaluer le comportement de Python dans la phase d'initialisation sur des très gros cluster HPC.
- Une tâche dédiée à l'interface CGNS/Python de différents outils ONERA. Compte tenu des ressources humaines disponibles dans les unités engagées sur cette tâche, seuls les modules FFX (extraction de trainée), move3D (déformation de maillage pour le couplage fluide structure) sont disponibles à la fin du PRF CHAMPION. Le code de propagation acoustique Sabrina_V0 n'a pas pu être interfacé jusqu'à présent ; il devrait l'être dans le cadre de la mise en place de la plateforme ORION/MOSAIC.
- Une dernière activité était consacrée au couplage multi-modèle, comme le couplage zonal RANS-LES. Pour cela l'approche proposée dans [13] a été mise en œuvre dans l'environnement FastS et Cassiopee/Connector. Son implémentation a été optimisée de manière à stocker en mémoire uniquement les variables nécessaires (5 pour la LES et 6 pour le RANS si le modèle de Spalart est retenu) et valider sur des cas relativement simple (profil d'aile sans flèche).

FÉVRIER 2020

FÉVRIER 2020

6. APPLICATIONS

Cette activité avait pour but de démontrer l'intérêt d'une plateforme faisant interagir des modules Python/CGNS pour réaliser des simulations CFD ambitieuses. Pour cela deux cas ont été initialement retenus :

- un à caractère académique visait à étudier le phénomène de tremblement sur voilure par des simulation LES/DNS. Des outils de posttraitement à la volée comme les filtres récursifs devaient être développés au Département d'Aérodynamique Fondamentale. Faute de ressources humaines suffisantes, l'activité a été fortement réduite et transférée dans l'unité DAAA/ACI (J. Dandois). Certaines configurations de tremblement ont été simulées avec succès et ont été valorisées par une publication [44]. Un dossier PRACE a été déposé pendant le PRF par J. Dandois pour poursuivre cette étude de manière plus ambitieuse sur des ordinateurs 10 à 20 fois plus gros que ceux de l'ONERA. Hélas, le sujet n'a pas été retenu car une équipe anglaise de l'université de Southampton a proposé la même étude au même moment.
- Un à caractère industriel autour de la configuration NOVA. Faute de ressource dans les départements DAAA et DADS, un cas test turbomachine (Configuration Create) lui a été substitué et l'activité partiellement transférée dans l'unité DAAA/H2T. Des validations du module Fast ont eu lieu sur des cas turbomachine académique en modélisation RANS (Rotor 37) au cours de la deuxième année du PRF. En 2018 cette activité n'a pu être poursuivie faute de ressource dans l'unité H2T. Les heures ont été basculées dans l'unité DEFI qui a préparé les modules FastS et Connector pour pouvoir faire des calcul d'interaction Rotor-Stator de manière efficace. Un calcul de faisabilité a été réalisé sur un étage de la configuration CREATE en modélisation URANS. Cette étude est en cours d'approfondissement en 2019 dans le cadre du Contrat DGAC ATOM pour lequel l'interaction rotor/stator sur 3 étages de la configuration CREATE doit être simulée avec le module Fast par un approche LES ou RANS/LES.

FÉVRIER 2020

FÉVRIER 2020

7. CONCLUSIONS

L'exécution du PRF CHAMPION a été significativement impactée par 2 évènements :

- la non disponibilité des équipes pressenties pour effectuer certaines tâches du projet. Le montage d'un PRF nécessite de consacrer un temps non négligeable à discuter avec différentes équipes de l'ONERA pour construire un programme scientifique cohérent et ambitieux et réclame souvent quelques long arbitrages pour maintenir le budget dans l'enveloppe d'un PRF. Aussi, il est assez regrettable que certaines équipes qui ont eu la chance de voir leur activité maintenue lors de la phase d'arbitrage (avec un volant supérieur à 1000 heures par an), se retirent 6 mois plus tard dès le début du projet sous couvert d'une charge contractuelle excessive. Il serait bon que les informations relatives à la possible non disponibilité des ressources humaines soit connues dès le montage d'un PRF afin de minimiser les risques de non exécution et de bâtir un programme scientifique plutôt avec des équipes qui ont une bonne chance de travailler pour le projet, qu'avec celles qui auront peu ou pas de temps à consacrer au PRF.
- la demande d'Airbus en 2017 quant au remplacement d'elsA et Tau par un code unique à l'horizon 2023 co-développé par l'ONERA et le DLR. Les travaux menés en 2016 dans les PRF CHAMPION et REALITY ont servi pour une bonne partie aux discussions ONERA/DLR/Airbus relatives à l'architecture de ce nouveau logiciel, tout particulièrement les ingrédients nécessaires au HPC et aux méthodes d'ordre élevé. L'approche technique préconisée par l'ONERA n'a pas été retenue et le DLR a donc fourni clés en main l'architecture de ce futur code. Même si ce choix peut apparaître plus politique que technique, il a entraîné une certaine perte de sens du PRF : l'architecture du futur code Airbus étant figée, quelle était la raison de travailler sur une nouvelle architecture ? La DSG a décidé de poursuivre en ciblant Safran au travers du PRF ODYSSEY, ce qui a conduit à l'apparition de nouveaux prototypes en plus de ceux développés dans le PRF CHAMPION. Dans cette période un peu confuse, il n'est pas étonnant que les équipes applicatives se soient peu investies pour réaliser les calculs démonstratifs prévus dans CHAMPION avec des prototypes à l'avenir très incertain.

Malgré ce contexte difficile, le PRF CHAMPION a permis de mettre en place une architecture de logiciel CFD basée sur l'interaction de modules Python/CGNS. L'objectif était de réduire la complexité des codes CFD en les séparant en plusieurs modules capables d'interagir efficacement. Pour cela les 2 boucles itératives d'un solveur CFD, celle liée à l'intégration temporelle et celle liée à l'algorithme associé (sous pas de méthode Runge-Kutta, sous-itération de Newton,...) ont été exposées au niveau de la couche Python. Ainsi, il devient possible de réaliser des couplages entre modules CFD (ou bien des modules simulant une autre physique) tant que les modules travaillent sur des données qui respectent le standard Python/CGNS en mémoire. L'avantage de cette approche est qu'elle permet d'introduire de la souplesse au niveau de chaque module sur les choix architecturaux tant que le standard Python/CGNS est respecté. Le prix à payer se situe sur la création de doublon et un très léger surcoût HPC. En effet, la récupération des pointeurs adressant les tableaux de flottants nécessite quelques micro seconde par pointeur. Ce coût est donc marginal, sauf si le nombre de pointeur à récupérer explose : grand nombre de zones et ou grand nombre de variables. Pour les variables, un stockage contigu des N variables des équations RANS a été retenu pour limiter d'un facteur N le nombre de pointeurs à récupérer. Un parcours parallèle de l'arbre pourrait permettre de rendre encore plus transparent cette étape du calcul, mais sa mise en oeuvre n'est pas triviale du fait du Global Interpreter Lock de Python. Pour un cas comportant une centaine de zones, il a été estimé que le parcours de l'arbre CGNS avait un

coût équivalent à celui requis pour la mise à jour de 10000 cellules avec les équations de Navier-Stokes. Tant que le nombre de cellules calculées par un coeur reste très supérieur à 10000, l'approche retenue pour coupler des modules Python demeure compétitive. Ceci est vrai dans l'écrasante majorité des cas de calcul issus de la pratique courante en calcul HPC. En effet, pour des raisons d'efficacité et de définition usuelle des classes de calcul sur les cluster HPC, la pratique veut qu'en général chaque coeur du processeur calcul au moins 100000 cellules, ce qui rend le surcoût de la couche Python indolore.

Un effort particulier a été entrepris pour atteindre un niveau de performance correct sur les processeurs récents d'Intel (10 à 20% de la puissance crête). Cela implique de limiter le trafic mémoire entre la DRAM et le processeur. Ceci conduit à limiter au maximum le recours à des tableaux de travail de grande taille et de parcourir un minimum de fois les tableaux où sont stockées les variables physiques du problème. Pour cela l'opérateur de Navier-Stokes a été décomposé en 5 ou 6 fonctions qui sont appelées successivement. Grâce à la technique de "cache bloquing" introduite dans le solveur Navier-Stokes, il a été mesurée qu'un seul transfert entre la DRAM et le processeur était nécessaire par itération pour calculer ces 5 ou 6 fonctions. Cette estimation, valable pour un calcul explicite, devient moins favorable pour un calcul implicite basé sur le solveur linéaire LU-SGS, puisqu'il faut alors 3 ou 4 va et viens entre la DRAM et le processeur par itération. Cette minimisation des transfert entre la DRAM et le processeur se fait au détriment de la simplicité du solveur. En effet, pour chaque combinaison de modèle physique et numérique (Turbulent, Laminaire, Euler, Spalart, schéma spatial, limiteur,...) une fonction doit être créée pour calculer les flux afférents ainsi que leur bilan. Cette tâche colossale, qui ne peut être réalisée qu'en s'appuyant sur un générateur de code automatique, a été réalisée dans le PRF grâce à des fonctions "patron" et des scripts de génération Python. D'autres choix sont possibles (voir CODA et ses templates C++14) pour générer les sous-routines cibles, mais le choix du python nous à sembler le plus pertinent pour cela :

- Le code fourni au compilateur étant trivial, celui peut faire un bon travail d'optimisation et l'efficacité HPC du code compilé s'en trouve renforcée.
- L'écriture en langage Python du générateur de code automatique n'est pas triviale, mais pour des développeurs maîtrisant les bases de ce langage, il demeure facilement compréhensible.

Au moins trois types de fonctions ont été identifiés comme candidats pour une génération automatique : le calcul des flux, les conditions aux limites et les éventuelles corrections de flux afférentes. Faute de moyen suffisant, la différentiation automatique du solveur n'a pas été évaluée de manière poussée dans le PRF. Cette activité est actuellement portée dans le PRF ODYSSEY, principalement autour de la librairie Tapenade de l'INRIA. Les choix réalisés dans le PRF CHAMPION pour générer des lignes de code automatiquement demeurent compatibles avec l'emploi de Tapenade.

Un effort important a été consacré à la mise en oeuvre d'une méthode de parallélisation hybride MPI-OpenMP. L'utilisation de la librairie OpenMP a rarement conduit à des accélérations de calcul significatives par rapport à des simulations 100% MPI. Des gains ont parfois été observés sur des processeurs à l'architecture particulière (Intel Xeon Phi), mais dans l'ensemble le niveau de performance est relativement identique entre un calcul hybride et un calcul 100% MPI. Cependant dans le cadre du PRF les plus grosses simulations réalisées étaient limitées à 20000 coeurs : il est possible que l'approche hybride soit pertinente sur des simulations plus ambitieuses (>100000 coeurs). De plus pour certains calculs instationnaires comme ceux utilisant la méthode explicite à pas de temps local, l'approche hybride demeure prometteuse, car elle permet d'introduire plus de souplesse dans le partage du travail, dont la quantité varie au cours du temps. Si la programmation hybride semble avoir quelques avantages, il est certain qu'elle introduit davantage de complexité dans le solveur. Cette dernière peut être relativement bien isolée du reste du solveur pour la rendre relativement transparente

FÉVRIER 2020

pour un développeur n'ayant pas d'intérêt pour la couche HPC. Néanmoins, la recherche d'erreur dans un code multi-threadé OpenMP demeure souvent un casse-tête même pour un spécialiste, dès lors qu'une "race condition" est en jeu.

La faible disponibilité des équipes applicatives pendant ce PRF a conduit à renforcer l'activité sur les méthodes de frontières immergées pour simuler des géométries complexes tout en utilisant une phase de maillage automatique à base d'Octree. Des progrès significatifs ont été réalisés dans ce PRF pour rendre plus robuste l'étape de pré-traitement qui permet de modéliser la paroi. La méthode a aussi été rendue plus précise en introduisant dans le module Cassiopee/Connector des lois de parois capables de reproduire des écoulements visqueux par des calcul stationnaires RANS ou instationnaires LES. Des progrès sont encore nécessaires pour rendre la méthode encore plus attrayante comme des lois de paroi plus précise ou la possibilité de mélanger l'approche IBM Octree avec des maillages "bodyfitted". Ces travaux de recherches sont actuellement en cours dans le PR MEMFIS débuté en 2018. Néanmoins la méthode fournit des résultats déjà prometteurs pour une large gamme d'écoulement :

- Pour les calculs RANS stationnaires, l'obtention de résultats quantitatifs nécessite l'emploi de maillage comportant plusieurs centaines de millions de cellules de calcul. Ce chiffre s'avère 1 à 2 ordres de grandeur plus grand que les pratiques usuelles en cours pour les calculs RANS bodyfitted. Néanmoins le solveur cartésien dédié à cette approche s'avère rapide et permet de passer des cas en moins d'une nuit. Comme la phase de maillage exige seulement quelques minutes, ce type de calcul peut avoir de l'intérêt pour des géométries difficiles à mailler par approche structurée ou non structurée.
- Pour les calculs instationnaires LES, l'obtention de résultats quantitatifs nécessite également l'emploi de maillage comportant plusieurs centaines de millions de cellules de calcul. Mais le surcoût en terme de cellules par rapport à un calcul sur un maillage "bodyfitted" n'est seulement que d'un facteur 5 à 10. Comme l'efficacité du solveur cartésien réduit la facture d'un facteur 2 et que l'emploi de lois de paroi permet de relacher les contraintes au niveau des résolutions pariétales nécessaires à l'obtention de résultats satisfaisants, l'approche IBM s'avère compétitive pour des calcul LES par rapport à l'approche "bodyfitted". La précision n'est pas encore du même niveau, mais il existe des pistes sérieuses pour l'améliorer (PR MEMFIS) qui pourrait faire de la méthode IBM LES un concurrent sérieux à l'approche LBM.

FÉVRIER 2020

FÉVRIER 2020

8. BIBLIOGRAPHIE

- [1] The CGNS Steering Committee.
<http://cgns.github.io>, 2011.
- [2] The CGNS Steering Committee.
Sids-to-python (cgns/python).
http://cgns.github.io/CGNS_docs_current/python/sidstopython.pdf, 2011.
- [3] DSG.
Orientations of future computational fluid dynamics activities at onera.
Technical report, ONERA, 2017.
- [4] <http://www.intel.fr/content/www/fr/fr/homepage.html>.
- [5] Amanda S.
Cache blocking techniques.
<https://software.intel.com/en-us/articles/cache-blocking-techniques>, 2013.
- [6] <http://www-sop.inria.fr/tropics/tapenade.html>.
- [7] I. Bermejo-Moreno, J. Bodart, J. Larsson, B. M Barney, J. W. Nichols, and S. Jones.
Solving the compressible navier-stokes equations on up to 1.97 million cores and 4.1 trillion grid points.
In High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for, pages 1–10. IEEE, 2013.
- [8] P. Vincent, F. Witherden, B. Vermeire, J. S. Park, and A. Iyer.
Towards green aviation with python at petascale.
In High Performance Computing, Networking, Storage and Analysis, SC16 : International Conference for, pages 1–11. IEEE, 2016.
- [9] S. Williams, A. Waterman, and D.s Patterson.
Roofline : an insightful visual performance model for multicore architectures.
Communications of the ACM, 52(4) :65–76, 2009.
- [10] S. Péron, I. Mary, T. Renaud, and C. Benoit.
Prf champion année 1 - méthode de frontières immergées sur maillage cartésien octree. validation sur un cas de profil 2d en rans avec modèle de paroi.
Technical report, ONERA, 2017.
- [11] S. Peron, T. Renaud, M. Terracol, C. Benoit, and I. Mary.
An immersed boundary method for preliminary design aerodynamic studies of complex configurations.
In 23rd AIAA Computational Fluid Dynamics Conference, Denvers, United States, 2017. AIAA paper 2017-3623.

- [12] T. Renaud, C. Benoit, S. Péron, I. Mary, and N. Alferez.
Validation of an immersed boundary method for compressible flows.
In *AIAA Scitech 2019*, San Diego, United States, 2019. AIAA paper 2019-2179.
- [13] G. Nolin.
Une méthodes de couplage RANS/LES zonal pour la simulation instationnaire d'écoulements turbulents complexes.
PhD thesis, Université Pierre et Marie Curie (Paris VI), 2007.
- [14] H. Choi and P. Moin.
Effects of the computational time step on numerical solutions of turbulent flow.
Journal of Computational Physics, 113(1) :1–4, 1994.
- [15] T. Unfer.
An asynchronous framework for the simulation of the plasma/flow interaction.
Journal of Computational Physics, 236 :229–246, 2013.
- [16] A. Toumi.
Asynchronous numerical scheme for modeling hyperbolic systems.
C. R. Acad. Sci. Paris, Ser I, 353(9) :843–847, 2015.
- [17] V. Semiletov and S. Karabasov.
Cabaret scheme for computational aero acoustics : Extension to asynchronous time stepping and 3d flow modelling.
International Journal of Aeroacoustics, 13(3-4) :321–336, 2014.
- [18] E. M. Constantinescu and A. Sandu.
Multirate timestepping methods for hyperbolic conservation laws.
Journal of Scientific Computing, 33(3) :239–278, 2007.
- [19] S. Osher and R. Sanders.
Numerical approximations to nonlinear conservation laws with locally varying time and space grids.
Mathematics of computation, 41(164) :321–336, 1983.
- [20] H. Tang and G. Warnecke.
A class of high resolution schemes for hyperbolic conservation laws and convection-diffusion equations with varying time and space grids.
Journal of Scientific Computing, 26(4) :1415–1431, 2005.
- [21] M. Schlegel, O. Knöth, M. Arnold, and R. Wolke.
Multirate runge-kutta schemes for advection equations.
Journal of computational and applied mathematics, 226(2) :345–357, 2009.
- [22] L. Liu, X. Li, and F.Q. Hu.
Nonuniform time-step runge-kutta discontinuous galerkin method for computational aeroacoustics.
Journal of Computational Physics, 229(19) :6874–6897, 2010.

FÉVRIER 2020

- [23] L. Liu, X. Li, and F.Q. Hu.
Nonuniform time-step explicit runge-kutta scheme for high-order finite difference method.
Computers and Fluids, 105 :166–178, 2014.
- [24] B. Seny, J. Lambrechts, R. Comblen, V. Legat, and J.-F. Remacle.
Multirate time stepping for accelerating explicit discontinuous galerkin computations with application to geophysical flows.
International Journal For Numerical Mehtods In Fluids, 71(1) :41–64, 2007.
- [25] V. Savcenco, W. Hundsdorfer, and J. G. Verwer.
A multirate time stepping strategy for stiff ordinary differential equations.
BIT numerical mathematics, 2006.
- [26] C. Dawson and R. Kirby.
High resolution schemes for conservation laws with locally varying time steps.
Society for Industrial and Applied Mathematics, 22(6) :2256–2281, 2001.
- [27] A. Kvaerno and P. Rentrop.
Low order multirate runge-kutta methods in electric circuit simulation.
Preprint No. 99/1, IWRMM, Universität Karlsruhe (TH), 1999.
- [28] Jean M. Sexton and Daniel R. Reynolds.
Relaxed mutlirate infinitesimal step methods for initial-value problems.
Preprint submitted to Journal of Computational and Applied Mathematics, 2019.
- [29] A. Sandu.
A class of multirate infinitesimal gark methods.
Technical report of the computational science laboratory (Virginia Tech), 1999.
- [30] W. Hundsdorfer, A. Mozartova, and V. Savcenco.
Analysis of explicit multirate and partitioned runge-kutta schemes for conservation laws.
Report of Centrum voor Wiskunde en Informatica, 2007.
- [31] O. Knuth and R. Wolke.
Implicit-explicit runge-kutta methods for computing atmospheric reactive flows.
Applied Numerical Mathematics, 28(2-4) :327–341, 1998.
- [32] J. Wensch, O. Knuth, and A. Galant.
Multirate infinitesimal step method for atmospheric flow simulation.
BIT Numerical Mathematics, 49, pp. 449-473, 2009.
- [33] E.Hairer, G.Wanner, and S.P. Norsett.
Solving ordinary differential equations I : Nonstiff Problems.
Springer, Berlin, 1993.

- [34] P. L. Roe.
Approximate riemann solvers, parameter vectors and difference schemes.
Journal of computational physics, 43(2) :357–372, 1981.
- [35] P. L. Roe.
Characteristic-based schemes for the euler equations.
Annu. Rev. Fluid Mech, 18(1) :337–365, 1986.
- [36] I. Mary and P. Sagaut.
Large eddy simulation of flow around an airfoil near stall.
AIAA Journal, 40(6) :1139–1145, 2002.
- [37] C. Hu and C. W. Shu.
Weighted essentially non-oscillatory schemes on triangular meshes.
Journal of Computational Physics, 150(1) :97–127, 1999.
- [38] J. R. DeBonis.
Solutions of the taylor-green vortex problem using high-resolution explicit finite difference methods.
51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, 2013.
- [39] O. Marxen and D.S Henningson.
The effect of small-amplitude convective disturbances on the size and bursting of a laminar separation bubble.
Journal of Fluid Mechanics, 671 :1–33, 2011.
- [40] N. Alferez, I. Mary, and E. Lamballais.
Study of stall development around an airfoil by means of high fidelity large eddy simulation.
Flow, Turbulence and Combustion, 91(3) :623–641, 2013.
- [41] C. Laurent, I. Mary, V. Gleize, A. Lerat, and D. Arnal.
Dns database of a transitional separation bubble on a flat plate and application to rans modeling validation.
Computers and Fluids, 61 :21–30, 2012.
- [42] L. Larchevêque, P. Sagaut, I. Mary, O. Labbé, and P. Comte.
Large-eddy simulation of a compressible flow past a deep cavity.
Physics of Fluids, 15(1) :193–210, 2003.
- [43] F. Daude, I. Mary, and P. Comte.
Self-adaptive newton-based iteration strategy for the les of turbulent multi-scale flows.
Computers and Fluids, 100 :278–290, 2014.
- [44] J. Dandois, I. Mary, and V. Brion.
Large-eddy simulation of laminar transonic buffet.
Journal of Fluid Mechanics, 850 :156–178, 2018.

FÉVRIER 2020

- [45] C. Gear.
Numerical initial value problems in ordinary differential equations.
Prentice Hall PTR, Upper Saddle River, New Jersey, USA, 1971.
- [46] B. Seny, J. Lambrechts, T. Toulorge, V. Legat, and J-F. Remacle.
An efficient parallel implementation of explicit multirate runge-kutta schemes for discontinuous galerkin computations.
Journal of Computational Physics, 256 :135–160, 2014.
- [47] M. Rietmann, D. Peter, O. Schenk, B. Uçar, and M.J. Grote.
Load-balanced local time stepping for large scale wave propagation.
2015 IEEE International Parallel and Distributed Processing Symposium, IEEE CPS, pages 925–935, 2015.