Development of Parallel CFD Applications with the Chapel Programming Language

Matthieu Parenteau * Simon Bourgault-Côté † Frédéric Plante‡
Polytechnique Montréal, Montréal, Québec, H3T 1J4, Canada

Engin Kayraklioglu§
Hewlett Packard Enterprise, San Jose, CA, 95002, USA

Éric Laurendeau¶
Polytechnique Montréal, Montréal, Québec, H3T 1J4, Canada

Traditionally, Computational Fluid Dynamics (CFD) software uses MPI (Message Passing Interface) to handle the parallelism over distributed memory systems and relies mostly on C, C++ and Fortran to ensure high performance. Consequently, the barrier of entry can be quite high for research and development, and productivity is therefore impacted. The Chapel programming language offers an interesting alternative tailored for research and development of CFD applications. In this paper, the developments of two CFD applications are presented: the first one as an experiment in rewriting a 2D structured flow solver and the second one as writing from scratch a 3D unstructured RANS simulation software named CHAMPS. Details are given on both applications with emphasis on the Chapel features which were used positively in the code design, in particular, to improve flexibility and extend the application from shared memory to distributed memory. Strong and weak scaling is evaluated up to 256 compute nodes on a Cray XC30 for a total of 9216 cores. Finally, CHAMPS is verified against well-established CFD software (FLO82, FUN3D, CFL3D and NSU3D).

I. Introduction

With the increase in computing power and more efficient algorithms, the field of Computational Fluid Dynamics (CFD) is now widely used in the analysis and design process for many industries, especially the aerospace industry. Even though this technology is more accessible, it still requires computing power and complex algorithms in order to solve the problems at hand. Consequently, the fidelity of CFD simulations is still limited by the available computing resources, thus overall performance remains a critical aspect for CFD codes.

The most common programming languages used for CFD software are Fortran, C and C++ to achieve high performance. As a result, parallelism is incorporated using external libraries such as MPI (Message Passing Interface) to handle the parallelism over distributed memory systems and OpenMP to handle parallelism over shared memory. The concept of Single Program Multiple Data (SPMD) for parallelism combined with complex algorithms to solve highly nonlinear equations makes the barrier of entry for CFD development high. In the context of an academic research laboratory, this barrier of entry has a significant impact on the overall science performed by the students that must complete their project within a time limit.

This challenge led the group to investigate other programming strategies and languages to develop CFD applications on distributed memory. The Chapel programming language[1] offers such an interesting alternative tailored for research and development of large distributed applications. Chapel provides a feature-rich language with native parallelism for both shared and distributed memory while maintaining impressive performance even when compared to equivalent C, as will be shown in Section III. Moreover, Chapel is a highly productive language, enabling quick implementation of complex algorithms on distributed memory. It also allows a flexible level of control between letting the language handle

*Ph.D. Candidate, Department of Mechanical Engineering, matthieu.parenteau@polymtl.ca
†Research associate, Department of Mechanical Engineering, simon.bourgault-cote@polymtl.ca
‡Ph.D. Candidate, Department of Mechanical Engineering, frederic.plante@polymtl.ca
§Software Engineer, Chapel team, engin@hpe.com
¶Professor, Department of Mechanical Engineering, eric.laurendeau@polymtl.ca, Senior AIAA Member

some operations, such as automatic reduction of a distributed array, or manual programming of such operations with different functions.

A preliminary study was therefore launched to investigate i) the performance a CFD code could achieve using Chapel and ii) the overall programming experience in comparison to other traditional languages used in CFD software (Fortran, C and C++). The CFD flow solver developed in this initial project is simply the conversion to Chapel of our in-house 2D structured CFD software written in C, using native Chapel features. The Chapel implementation along with performance comparisons between both codes are discussed in Section III.

The results of this first experience with Chapel for CFD applications led to the development of our laboratory's next-generation 3D multi-physics CFD software designed entirely with the Chapel language features for research and development. Consequently, the new software was named *Chapel Multi-Physics Simulation* (CHAMPS). In the remaining sections, the design choices and their implementation in CHAMPS are discussed. The efficiency and scalability of CHAMPS are presented with up to 256 compute nodes (36 cores per node) on a simple 3D Cartesian grid case. Finally, the flow solver in the software is formally verified on standard literature cases to assess its accuracy order and the turbulence modelling implementation, and on workshop cases to demonstrates its capability to simulate the flow around more complex geometries.

II. The Chapel Language

Chapel is a programming language aiming at increasing productivity for the development of parallel distributed applications. Developed since the early 2000's by Cray Inc., now a part of Hewlett Packard Enterprise, the Chapel language is inspired by several very common languages such as Python, Fortran, C and C++ and provides a high-level interface for parallel distributed programming while keeping high performance and allowing the developers to select different levels of control over the code. For example, whole array assignments can be used instead of loops to reduce the code length, but it comes with less control over the system resources.

Data distribution and task parallelism are native in Chapel. Based on a partitioned global address space (PGAS) model, it supports a global namespace that permits each variable within the lexical scope to be referenced regardless of whether it is stored locally or on a remote compute node. When executing a Chapel program, a single instance is launched on multiple compute nodes, which are called *locales*, and tasks can be spawned within the program to use the available resources. A *coforall* loop, for example, would spawn as many tasks as requested by the length of the loop. As an alternative, parallel data operations similar to what is seen in OpenMP applications can be performed by using a *forall* loop. These operations are further simplified by a unique feature of Chapel which is used to define a set of indices: the *domain*. By defining an array with a *domain*, its allocation, reallocation or deletion become entirely dependent on that *domain*, effectively reducing the risk of memory corruption and reducing the development time when dealing with arrays. As a *domain* can also be distributed, it allows the allocation of arrays on different compute nodes and to access their data directly without any special process such as would be required with MPI.

Chapel is also an object-oriented language featuring two types of objects, namely record and class. Class objects are further declared with a memory management strategy which allows the developer to let the language handle the destruction of the object instances at the end of their lifetime. Moreover, Chapel supports class inheritance, which allows creating object families for which the children can be further specialized by the use of generic types in the structure.

III. 2D CFD Solver

In this section, our in-house 2D Euler CODE (NSCODE) [2] written in C is converted to Chapel using some of Chapel's native features. NSCODE is parallelized on shared memory using OpenMP, thus the Chapel counterpart is developed for single *locale*. The main objective of this exercise is to quickly investigate the performance achieved with Chapel in comparison to C and to get a first impression of Chapel in the context of CFD.

NSCODE's flow solver implements a structured cell-centered finite volume scheme. The convective fluxes are discretized using a centered differential operator and an added artificial scalar dissipation, following the Jameson-Schmidt-Turkel scheme [3] with Martinelli scaling [4]. The equations are iterated in pseudo-time up to the convergence to a steady-state solution. A five-stage hybrid Runge-Kutta scheme [5] is used and the convergence is accelerated using local time steps and residual smoothing [6]. The cell-centered finite volume approach requires computations to be performed on the cell volume as well as on the edges or faces of those cells. As a result, the most effective way to incorporate parallelism is to divide the computational grid into blocks where a task or thread will be executed simultaneously on each block. It is an iterative process where each task exchanges its solution along the interfaces of the

block they share with another block at the end of each iteration. This is accomplished by using the concept of ghost cells.

The multi-block structure is built in NSCODE through the use of an array of *struct* where each *struct* contains all the information concerning its computational block. In the Chapel implementation, this is done through an array of *class* objects, called *Mesh*, using a Chapel *domain* to define the array. The parallelism on the block loops is achieved through the use of OpenMP for NSCODE, while the Chapel implementation uses Chapel's native parallelism over shared memory with the *forall* statement. These are the only structural differences between the C and the Chapel implementations. All algorithms remain identical.

A. Comparison Against NSCODE

A simple test case was performed to compare the performance of NSCODE with its Chapel counterpart. The real time to complete 100 iterations is evaluated for several numbers of cores up to 32 cores on a 2D structured grid of 1,024x1,024 elements. The computations were performed on a single node of the Compute Canada's Beluga cluster and the node's characteristics are presented in Table 1. As shown in Figure 1, the overall performance is very similar between the C and the Chapel implementation. Surprisingly, the Chapel implementation is almost two times faster on a single core. However, as the number of cores increases, the overall time to compute 100 iterations becomes equivalent. These results represent a first step in showing that the Chapel programming language is suitable for parallel CFD applications in terms of performance while offering a feature-rich language for efficient coding.

Table 1 Beluga node characteristics

Cores	40
Available memory	186G
CPU	2 x Intel Gold 6148 Skylake 2.4 GHz
Storage	1 x SSD 480G

In this exercise, we found that the Chapel *domain* type is very useful in defining a clear-solving process where different loops over the *i,j* directions are required for both the cells and the faces of those cells. In a C code, these various loops can become confusing, where some arrays of data are defined on the number of faces and others on the number of cells, which also contains two layers of ghost cells. Consequently, it is not always clear if a loop is accessing faces, cells or ghost cells and in which direction. This is where the *domain* type in Chapel is useful. For example, a *domain* can be created for the faces in the *j* direction as follows

```
var jFacesDomain : domain(2) = {2..rimax, 2..rjmax+1};
```

where *rimax* and *rjmax* are the number of nodes in each direction. Then, a loop over the faces in the *j* direction is performed as

```
for (i,j) in jFacesDomain
{
    var sx : real = snx[i,j];
    var sy : real = sny[i,j];
```

where *sx* and *sy* are the area vector components of the *i,j* face. With the definition of these *domains*, the code reads much more easily and is less prone to errors for new developers.

The overall programming experience with Chapel is satisfying. It is a feature-rich language allowing efficient implementation of algorithms for scientific computing. As an example, the Chapel implementation was easily extended to distributed memory with just a few small changes. The main modification consisted in defining the various *Mesh* instances over a *dmapped* domain as follows

```
var nBlock : int;
var blockSpace = {0..#nBlock};
var blockDomain : domain(1) dmapped Block(boundingBox=blockSpace) = blockSpace;
var blockData : [blockDomain] Mesh;
```

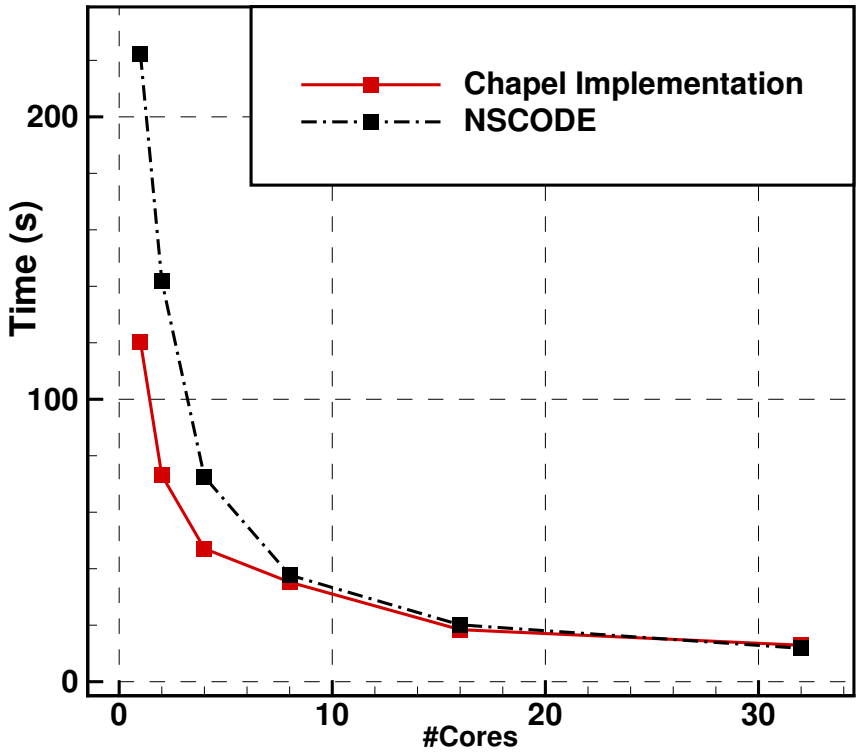


Fig. 1 NSCODE vs Chapel implementation - Real time in seconds to complete 100 iterations on a grid of 1,048,576 elements.

A *dmapped* domain is a distributed domain created with a specific distribution pattern. The *Block* distribution is used in this case, meaning that the index set is distributed with equal continuous blocks of data on each *locale*. Using such a distributed domain to define an array implies that the array is distributed using the same pattern and that accessing the element stored at an index associated with a different *locale* is possible as if shared memory was used, where the underlying remote communication is implemented by the compiler. Therefore, only a few lines of code are necessary to extend the Chapel implementation from shared memory to distributed memory. The same extension in the C code would require significant modifications to include MPI function calls within its structure. This example shows clearly how efficient programming with Chapel can be for distributed-memory applications.

IV. Chapel Multi-Physics Simulation (CHAMPS)

Since the results obtained with the 2D CFD flow solver written in Chapel were satisfactory, the development of a complete 3D unstructured Reynolds-Averaged Navier-Stokes (RANS) code in Chapel for distributed memory was initiated. This new software is named Chapel Multi-Physics Simulation (CHAMPS). The architecture of CHAMPS is designed to incorporate modules required to performed stability, ice-accretion and aeroelastic analyses, hence the name Multi-Physics. However, this paper only describes the development of the aerodynamic solver even though ice accretion and stability analysis are already available.

In the following sections, the design and implementation of CHAMPS using the Chapel programming language are discussed in detail and the code's strong and weak scalability is evaluated on a simple 3D cartesian grid. Furthermore, CHAMPS is verified against well-established CFD software for different flow conditions and different configurations.

A. Overview of CHAMPS

The aerodynamic solver is a cell-centered finite volume scheme on unstructured grids and the convective fluxes are discretized using the Roe scheme [7]. Second-order spatial accuracy is achieved using a piecewise linear reconstruction [8] of the solution at the grid facets using the gradient of the flow variables. The gradients are computed using the Green-Gauss [9] or Weigthed Least Square [9] methods, for which the Weights are computed as the inverse

distance between the cell-center and the center of the facets. Hybrid gradients (WLSQ and GLSQ) [10] are also available. The gradients can be limited using either the Barth and Jespersen [8] or Venkatakrishnan [11] limiters. Steady-state solutions are obtained by integrating the equations using a local pseudo-time step up to the point where the summation of the convective and viscous fluxes is equal to zero. This temporal integration can be carried out using an explicit five-stage hybrid Runge-Kutta scheme [5], a Block Symmetric Gauss-Seidel (SGS) scheme or a Flexible-GMRES. Solvers from the PETSC library are also available. For the simulation of turbulent flows, the Reynolds-Averaged Navier-Stokes equations are closed with the one equation Spalart-Allmaras (SA) model [12, 13] or the $K-\omega$ SST (SST-V) model [14]. The turbulence model is solved in a segregated way using the same numerical pseudo-time integration scheme as the flow variables and the convective fluxes are discretized with a first order upwind scheme.

The software relies on five external libraries. The CGNS library (CFD General Notation System) is used to read/write grids and flow solutions in the CGNS data format that defines a standard organization of the data for CFD within an HDF5 file [15]. The METIS library is used to perform automatic mesh splitting in order to distribute the grid over several compute nodes for parallelism. The Intel MKL package is also used for its high-performance linear algebra routines. Finally, the PETSc library is also linked into CHAMPS to provide some linear solvers and preconditioners. An interface is built for each of those libraries and Chapel made it very easy and efficient to do so with its C interoperability. By integrating with such well-known technologies, CHAMPS can easily fit in a scientist's workflow.

V. Data Structures and Type Aliases

As discussed in Section III, the parallelism is applied by dividing the grid into multiple zones, which in turn have boundary zones, and the same computation is carried on those grid zones in parallel with a solution exchange at interfaces at the end of each iteration. Therefore, three layers of data structures are of utmost importance in defining the code: i) the global level structure that controls the communication and the data exchange between the zones, ii) the zone-level structure that holds the information related to a volume grid zone and iii) the boundary-level structures that hold the information regarding the surface boundaries and interfaces of a volume zone. Note that a zone is the unstructured equivalent of a structured block. A significant difference is that cells cannot be accessed via implicit Cartesian (i,j,k) indices, but through the explicit definition of the connectivity.

A. Computational Grid

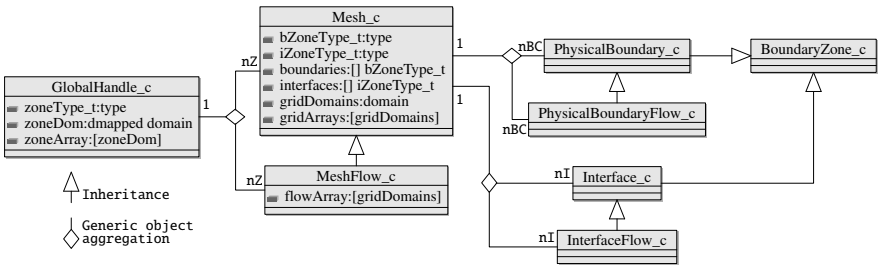


Fig. 2 Overview of the basic structural classes in CHAMPS.

The global level structure that distributes the computational grid is a crucial part of the software since it impacts how data is accessed. Moreover, a flow analysis requires a different type of grid than a Finite Element Analysis (FEA), thus a generic and reusable structure is needed for multi-physics simulation software. This is achieved through the use of generic classes and records.

For example, the distribution of the computational grid over the available compute nodes is handled by a generic class named *GlobalHandle_c*, defined as follows

```
class GlobalHandle_c
{
    type zoneType_t = Mesh_t;
    const localeSpace_ = {0..#numLocales};
    const localeDomain_ : domain(1) dmapped Block(boundingBox=localeSpace_) = localeSpace_;
    const numZones_ : int = 0;
    const zoneSpace_ = {0..#numZones_};
    const zoneDomain_ : domain(1) dmapped  Block(boundingBox=zoneSpace_) = zoneSpace_;
    var zones_ : [zoneDomain_] owned zoneType_t;
```

Essentially, the *GlobalHandle_c* object distributes any type of grid zone defined with the *zoneType_t* type field over the available Locales using a *Block* distribution pattern. By default, the type is a *Mesh_c* class containing only the grid coordinates and the grid connectivity. Depending on the main module being compiled, it can also be a *MeshFlow_c* class that inherits from *Mesh_c* for a flow simulation, as seen in Figure 2, or another type such as *MeshIcing_c* for icing simulations. This class hierarchy makes it very easy for any new developer to implement another computational grid/zone type and to use it on distributed memory.

As seen on Figure 2, the zone-level structure that contains the grid information, *Mesh_c*, is also a generic class that features inheritance and depends on the boundary zone types to define which physical boundaries and interfaces will be allocated within it. Inheritance was chosen to handle the various needs of the different computational grids (flow, structure, etc.) and the reason behind this choice is again to allow reusability. The base object, *Mesh_c*, is the simplest grid in the sense that it contains only metric information like elements connectivity, facets normal, elements volume and grid coordinates. These are essential information required for any analysis whereas specific data required for a simulation type is added within a derived class created from *Mesh_c*. For example, for a flow analysis, the *MeshFlow_c* class contains all the mandatory data about the grid from *Mesh_c* and also contains supplementary fields like pressure and density for the flow along with additional methods to solve the flow equations. Since all specialized grids are derived from the same object, namely *Mesh_c*, it allows the reusability of many procedures and it facilitates the communication between different grid types when performing multi-physics simulations. It is especially so since the different array fields are allocated using domains defined in *Mesh_c* which are thus common within all derived classes.

The last critical structural component of the code is how the boundary zones are defined. Physical boundaries represent a boundary where a physical condition is applied, like a no-slip wall or an inlet, whereas the interfaces are virtual boundaries connecting two zones, thus where communication between zones occurs. As seen in Figure 2, inheritance is used in two levels for the boundary zone classes : 1) the flow variants of the boundaries and interfaces inherit from the general ones (*PhysicalBoundary_c* and *Interface_c*), and 2) all these specialized classes inherit from a general *BoundaryZone_c* class that defines the grid boundary zone itself. This hierarchy again promotes code reuse.

These many layers of generic classes and inheritance increase the complexity of the code structure, especially since the generic types must be defined at compile time. To simplify the definition of the structure while programming, type aliases are predefined by including all related generic definitions and memory management specifications for a specific use. Such type aliases are thus widely used in CHAMPS. For example, there is a type alias defined specifically for a GlobalHandle object that will handle a *MeshFlow_t* zone type for a flow simulation

```
1  type GlobalHandleFlow_t = owned GlobalHandle_c(MeshFlow_t);
2
3  type MeshFlow_t = owned MeshFlow_c(PhysicalBoundaryZoneFlow_t, InterfaceZoneFlow_t);
```

where *MeshFlow_t* is a type alias defining the complete type of the zone on which to simulate on. At the boundary zone level, other type aliases define the type of physical boundaries and interfaces used within the zone along their memory management strategy, such as the *PhysicalBoundaryZoneFlow_t* and *InterfaceZoneFlow_t* type aliases. As a result, it eases the use of these complex generic objects and promotes code reuse of various procedures that initialize at run-time the structure, given a set of type aliases defined at compile time depending on the main module chosen by the user. For example, the same procedures are used to initialize and read a grid, whether it is for a flow, a structural or an ice accretion simulation. Consequently, it becomes very easy for a researcher or student to quickly develop a new application given the flexibility of the code structure. Note the use of the *owned* memory management strategy that states the allocated memory is owned by the variable that a particular instance is assigned to and will be freed when the variable goes out of scope. The memory deinitialization of a *GlobalHandleFlow_t* instance will trigger the memory deinitialization of all its zones and their boundary zones, thus limiting the risks of creating memory leaks while in development.

B. Interface Exchange

The interface exchange is another crucial portion of the code, because it involves communication between the interfaces of the various zones, thus between the compute nodes. The design is inspired by what would be done in a traditional SPMD CFD code with MPI. We believe it remains the most efficient approach for CFD codes as it limits the creation and destruction of tasks within the nonlinear iterative solver. Although Chapel can handle distributed communication easily, in CFD simulations the communication must be controlled closely in order to avoid poor scaling properties, since a large number of compute nodes are often required depending on the problem size.

The data exchange between two neighbouring zones is performed in two steps through the interface objects discussed earlier. First, each zone fills a dedicated buffer within all of its interfaces, which are linked to neighboring zones, with the data that need to be exchanged. An *allLocalesBarrier* is used to synchronize the compute nodes to ensure every

buffer is filled before the next step. The second and final step is fetching and reading the dedicated buffer from the corresponding interfaces of the neighboring zones, which is executed by the *exchangeInterfaces* procedure shown below.

```
proc performInterfaceExchanges(zone, exchangeType : ExchangeType_t)
{
    // Fill buffers
    local do zone.prepareExchange(exchangeType);
    allLocalesBarrier.barrier();

    // Read buffers
    zone.exchangeInterfaces(exchangeType);
    allLocalesBarrier.barrier();
}
```

The *ExchangeType_t* argument is an enumeration type used to define which data is exchanged between the zones. During the flow simulation, different fields are exchanged at different times, like the flow primitive variables or gradients. The use of a specific exchange type allows to perform the data exchange through a single common function. As a result, any new specific data exchange is easily added, allowing developers to quickly develop their scientific simulation over distributed memory. Moreover, with this design all major communications between the compute nodes are contained within this procedure, thus promoting the use of *local* statement elsewhere in the code to increase performance. Furthermore, the interface zone instance contains two objects for any given specific exchange type that will include a buffer and specific methods to fill or read that buffer. One is related to the local receiver and actually contains buffers for data exchange, while the other can be considered a pointer to the distant donor, which might be on another compute node, so that fetching the buffered data from the distant donor zone is done without having to perform communication at each level of the global structure. This is performed with the use of the *shared* memory management that allows the ownership of an instance to be shared among multiple object variables, even on different compute nodes.

VI. Parallelism Over Distributed Memory

The main loop concerns the iterative solving process that must be carried over the available compute nodes and synchronized between them. It is easily done using Chapel's *locale* type and *on* clauses, but it must be done with care to avoid accessing data stored on different *locale*, thus creating unwanted communication.

The initial grid is divided into multiple zones and then distributed over the available compute nodes through the *GlobalHandle_c* object. A single task is created for every zone through Chapel's *coforall* statement. Two *coforall* loops are shown below. The first *coforall* loop creates a task per node and starts executing them on the corresponding node by virtue of the *on* statement. The second *coforall* creates a task for each local zone allocated on the current node, then each task enters the iterative process until the maximum number of iterations is reached or a stopping criterion is triggered. The *coforall* loops are placed outside the main iterative process to reduce overhead caused by the *on* clauses and the *coforall* statements.

```
coforall loc in Locales do
on loc
{
    const localZonesIndices=globalHandle.zones_.localSubdomain();
    var localZones=globalHandle.zones_.localSlice(localZonesIndices);
    var modelHandle=globalHandle.modelHandle_;

    coforall (zone, localTaskID) in zip(localZones, 0..#localZones.size)
    {
        var numIterPerCycle : int = zone.flowSolver_.numIterPerCycle();
        var maxIter : int = NITER*numIterPerCycle;

        for i in 0..#maxIter
        {
            zone.flowSolver_.iterate(zone);

            const implicitCycleState : bool = zone.flowSolver_.inImplicitCycle();

            if implicitCycleState {
                zone.performInterfaceExchanges(ExchangeType_t.FLOW_DELTA_W);
            } else {}
```

However, in order to synchronize all tasks for data exchange, the *allLocalesBarrier* must be adjusted to account for the number of tasks per *locale*. As a consequence, every *locale* must have locally the same number of tasks/zones. This restriction can be managed fairly easily with unstructured grids, but the grid must still be divided specifically for the compute nodes on which it will be executed to ensure efficiency and that the entire nodes are used.

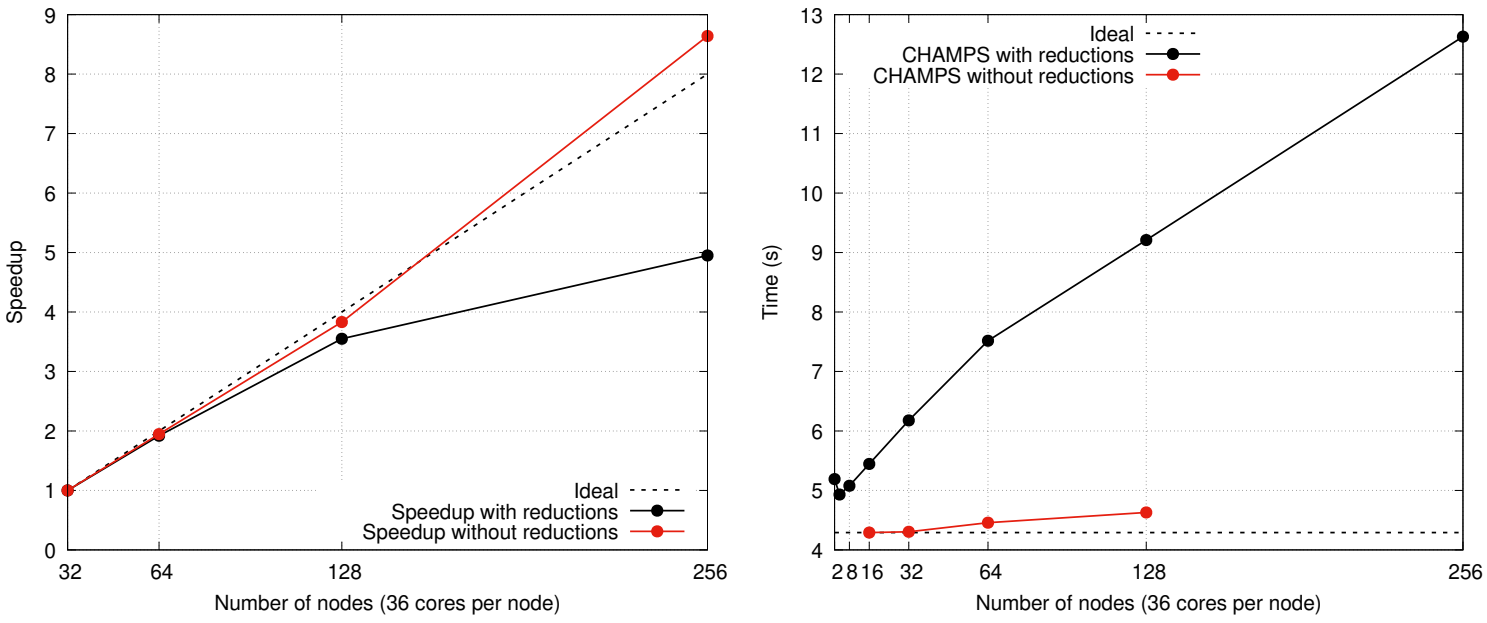
VII. Overall Performance

There are three typical components when it comes to assessing the performance of a CFD software: the solution convergence rate, the computational time per iteration and the scalability. The convergence rate mainly depends on the algorithms, flow conditions and mesh quality, while the computational time and scalability both depend on the hardware, the programming language and the implementation. In order to evaluate the performance of CHAMPS over distributed systems, the strong and weak scaling properties are estimated. These analyses are performed on a Cray XC30 with Aries interconnect with the following node characteristics

Table 2 Cray XC30 node characteristics

CPU	Dual Intel Xeon E5-2695 v4 2.1 GHz (Broadwell), 36 cores per node
Memory	128 GB DDR4-2400

A simple 3D Cartesian grid is used for both the strong and weak scaling. For the strong scaling, the mesh is discretized with 800 elements in each direction (i,j,k). For the weak scaling, the problem size is scaled with the number of compute nodes to maintain roughly 1M cells per core. The strong and weak scaling are presented in Figure 3a and 3b respectively. For the strong scaling, the compute time at 32 nodes is used as the reference, since a lower number of *locales* does not provide sufficient memory for this grid.



(a) Strong scaling **(b) Weak scaling**

Fig. 3 Strong and weak scaling using a 3D Cartesian grid with and without reductions.

During the nonlinear iterative process, the solution convergence is monitored using different values of the flow solution. These convergence values require a global reduction across all *locales*, thus impacting the scaling efficiency. As an example, the aerodynamic forces are monitored during the iterative process and since the surface of the geometry is divided between the different *locales*, the forces are computed locally on each *locale* and then a reduction is performed

across all *locales* to obtain the global value, even though some *locales* might not have any zones touching the solid surface. Consequently, it creates resource under-utilization, which, in turn, reduces the performance at scale. In order to evaluate the impact of these reductions, strong and weak scaling are presented with and without these reductions. However, communication is maintained everywhere else in the iterative process.

Figure 3a shows that a relatively good strong scaling is maintained up to 128 compute nodes with reductions. However, at 256 computes nodes (9216 *locales*), the impact of reductions is significant, whereas when these reductions are removed linear scaling is maintained. The same trend is observed for the weak scaling presented in Figure 3b, where the number of cells is increased at the same rate as the number of compute nodes. Ideally, the compute time should not increase, but there is a notable increase with the problem size when reductions are activated. Without reductions, the result is closer to an ideal scaling, as expected. These results indicate that further improvements of global reduction and communication management are necessary to achieve better performance.

However, the scaling performance without reductions shows that Chapel handles communication efficiently, since flow information is still exchanged between all *locales* at every iteration of the solving process. These communications include the exchange of the flow primitive values (density, velocity and pressure) and their gradients as well.

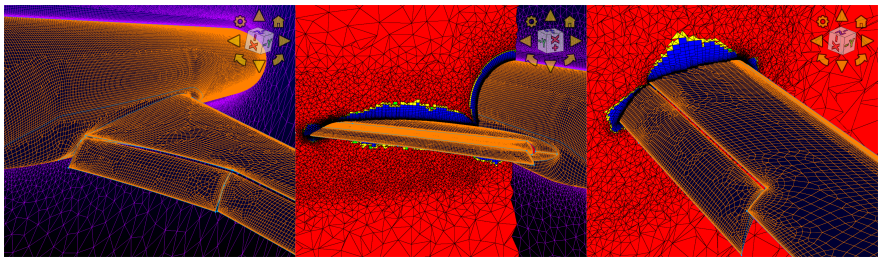


Fig. 4 Grid used for the Common Reasearch Model (CRM) high-lift configuration.

Finally, the Chapel programming language is actively developed and there are often improvements in overall performance between the various releases. As an example, Figure 5 presents the compute time to complete 10 iterations for CHAMPS compiled with Chapel 1.22 and Chapel 1.23, which is currently the latest release. A coarse mesh (Figure 4) of the CRM-HL configuration is used to produce these results. There is an average of 10% reduction in overall compute time with CHAMPS when using Chapel 1.23. The Chapel programming language is already providing impressive scaling capabilities and further improvements are expected from the next releases.

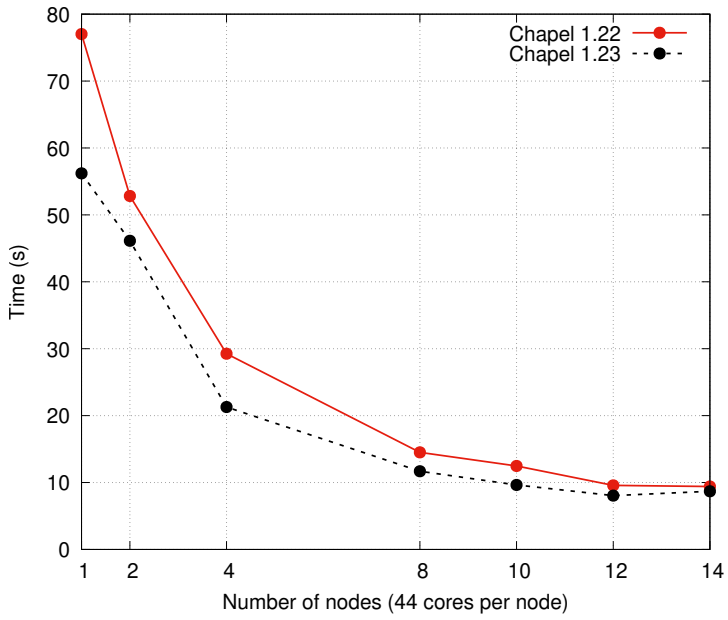


Fig. 5 Total compute time to complete 10 iterations using a coarse mesh of the CRM-HL configuration.

VIII. Numerical Results

This section provides verification that CHAMPS correctly solves the Euler and RANS equations. The first test case is the 2-D inviscid flow over a NACA0012 airfoil to verify the second order accuracy of the convective fluxes. The second test case is the turbulent flow over a flat plate, used to verify the implementation of the baseline versions of the Spalart-Allmaras and $K-\omega$ SST turbulence models. The third test case is the turbulent flow on a three-element airfoil to verify the Spalart-Allmaras model on a complex case. At last the solution of CHAMPS on a three-dimensional half wing-body aircraft geometry is analyzed. For all of these cases, a grid convergence study is performed and the solutions of CHAMPS are compared to solutions of state-of-the-art CFD solver on the same grid family. Note that 2-D cases are performed on planar grids, as CHAMPS can handle both 2-D and 3-D grids in CGNS unstructured format without requiring any special inputs from the user.

A. 2D inviscid NACA0012 - second order verification

The first verification case is the inviscid flow over a NACA0012 airfoil at a Mach number of 0.5 and an angle of attack of 1.25°. This case is intended to verify the grid convergence order of the solver. The grids of Vassberg and Jameson [16], who studied the grid convergence order of several CFD codes, are used as this family of grids is made of quadrilateral cells featuring aspect ratios close to unity. CHAMPS simulations are carried out using Roe convective fluxes, weighted least square gradients, no gradient limiter and a value of the Harten cut-off set to 0.1 times the local speed of sound. Pseudo-time integration is carried out using the SGS scheme with a relaxation factor of 0.9 and a CFL number of 100.

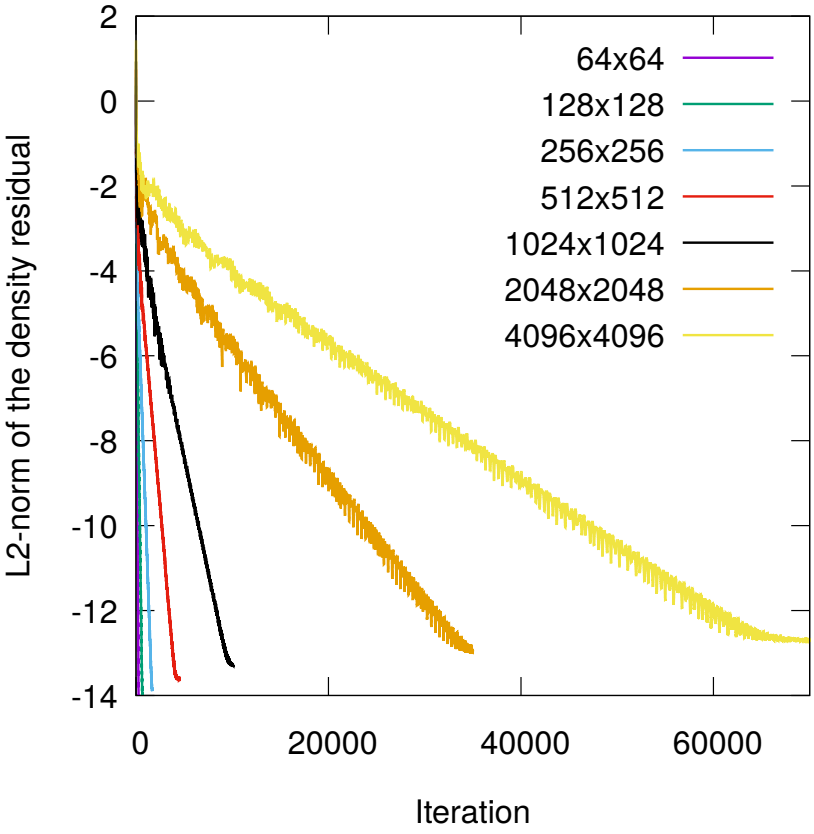
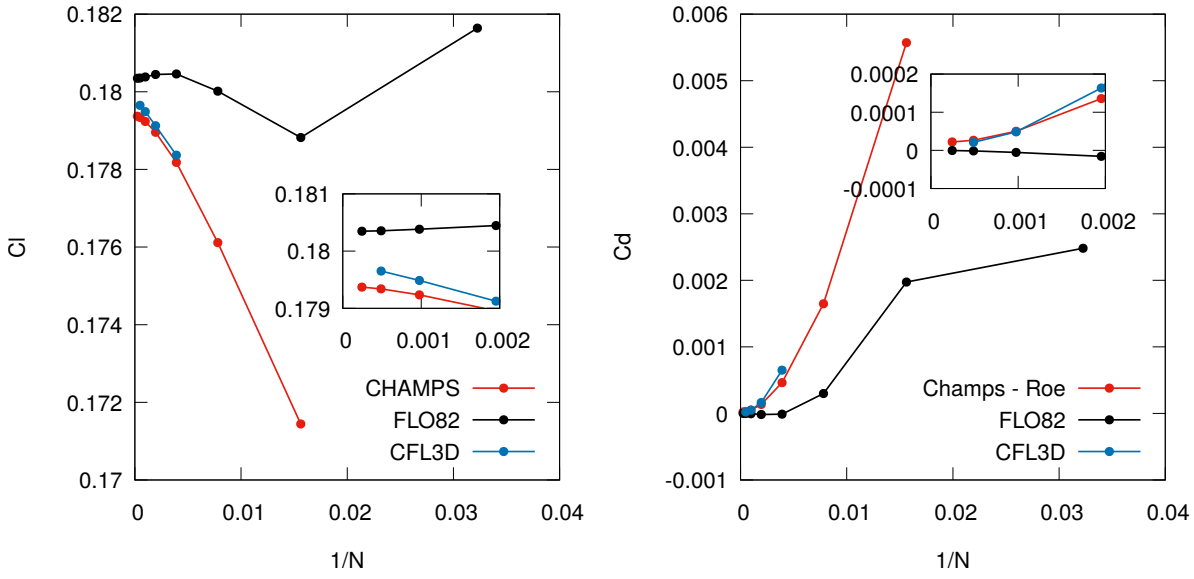


Fig. 6 Residual convergence of CHAMPS for the inviscid flow over a NACA0012 airfoil.

Figure 6 shows the iterative convergence of the density residual for all grids. A convergence to machine accuracy in 250 iterations is achieved for the coarse grid and the convergence rate slows down as the grid is refined. This behaviour is to be expected without the use of a multigrid algorithm.

Table 3 Grid convergence (NACA0012, $M = 0.5$, $\alpha = 1.25°$)

	CHAMPS - Roe		FLO82		CFL3D	
NC	C_l	C_d	C_l	C_d	C_l	C_d
32	-	-	0.1804581	0.0024821	-	-
64	0.1714422	0.0055709	0.1788225	0.0019746	-	-
128	0.1761140	0.0016481	0.1800143	0:0002984	-	-
256	0.1781812	0.0004631	0.1804582	-0.0000110	0.1783667	0.0006500
512	0.1789561	0.0001357	0.1804462	-0.0000155	0.1791250	0.0001634
1024	0.1792361	0.0000501	0.1803826	-0.0000053	0.1794876	0.0000485
2048	0.1793394	0.0000265	0.1803544	-0.0000015	0.1796505	0.0000209
4096	0.1793700	0.0000222	0.1803478	-0.0000004	-	-
Continuum	0.1793828	0.0000212	0.1803459	0.00000005	0.1797835	0.0000122
Order p	1.756	2.457	2.107	1.805	1.154	2.061
Order $\bar{p}$	-	0.256	-	1.932	-	1.213



(a) Lift coefficient **(b) Drag coefficient**

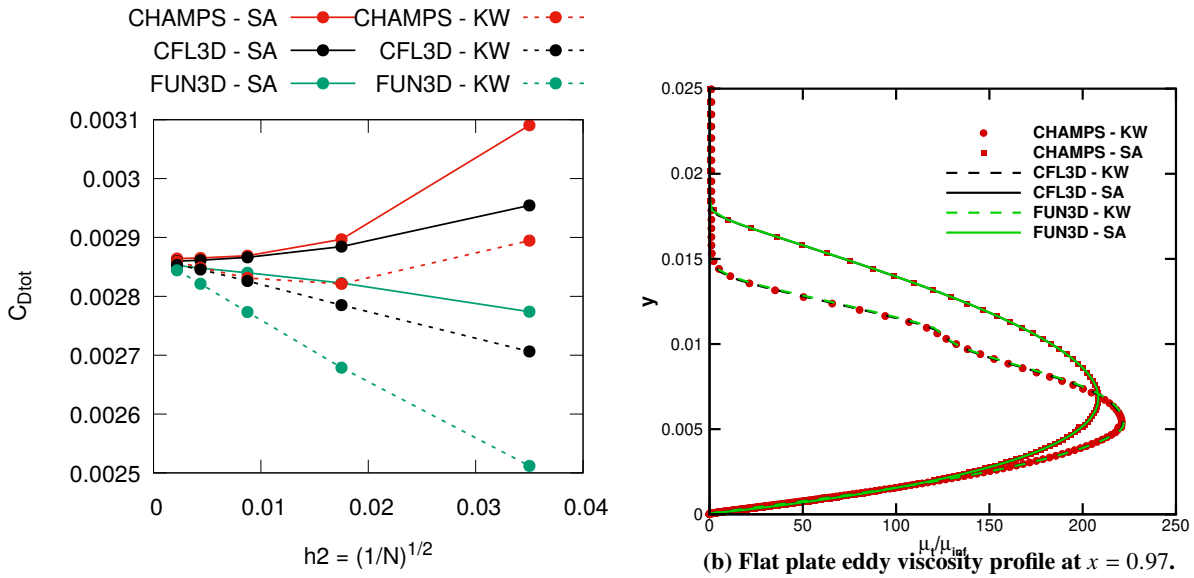
Fig. 7 Grid convergence of the forces coefficients on the NACA0012 airfoil.

Table 3 compares the grid convergence of CHAMPS to the results of FLO82 and CFL3D presented by Vassberg and Jameson [16]. One can see that an order of 1.76 and 2.46 are respectively obtained for the lift and drag coefficients, which is comparable to the convergence rate of the two reference codes. This verifies the second order accuracy of CHAMPS. Finally, Figure 7 shows the grid convergence of the lift and drag coefficients. The convergence of CHAMPS is similar to the one of CFL3D, which uses numerical methods similar to the one in CHAMPS. One should note that FLO82 used significantly different numerical methods and the difference in the convergence was to be expected.

B. 2D flat plate - turbulence model verification

The second test case is intended to verify the implementation of the Spalart-Allmaras and the $K - \omega$ SST turbulence models. More specifically, the SA model and the SST-V model are used [17]. A proper verification of these models

is desirable since the turbulence models have a great impact on the flow solutions and a proper identification of the turbulence model variant is now requested in multiple workshops. Variability in the turbulence model is detrimental to the reproducibility of numerical results from one CFD code to another. The case of the turbulent flow over a flat plate is analyzed here. The grid and flow conditions are the one of the VERIF/2DZP case of the NASA Turbulence Model Ressources [17]. The flat plate has a length of 2 and the flow conditions are a Mach number of 0.2 and a Reynolds number of 5 millions (based on a unitary length). The family of structured grids ranging from 32x25 to 545x385 provided by NASA is used. The convective fluxes are discretized with the Roe scheme and all gradients are computed using the Green-Gauss approach without any limiter for the piecewise linear reconstructions. The farfield boundary condition of the Spalart-Allmaras turbulence model is $\tilde{\nu}_\infty = 3\nu_\infty$. In the case of the $k-\omega$ SST, $k_\infty = 9 \times 10^{-9} a_\infty^2$ and $\omega_\infty = 1 \times 10^{-6} \frac{\rho_\infty a_\infty^2}{\mu_\infty}$.



(a) Flat plate drag coefficient.

(b) Flat plate eddy viscosity profile at $x = 0.97$.

Fig. 8 Flat plate verification results.

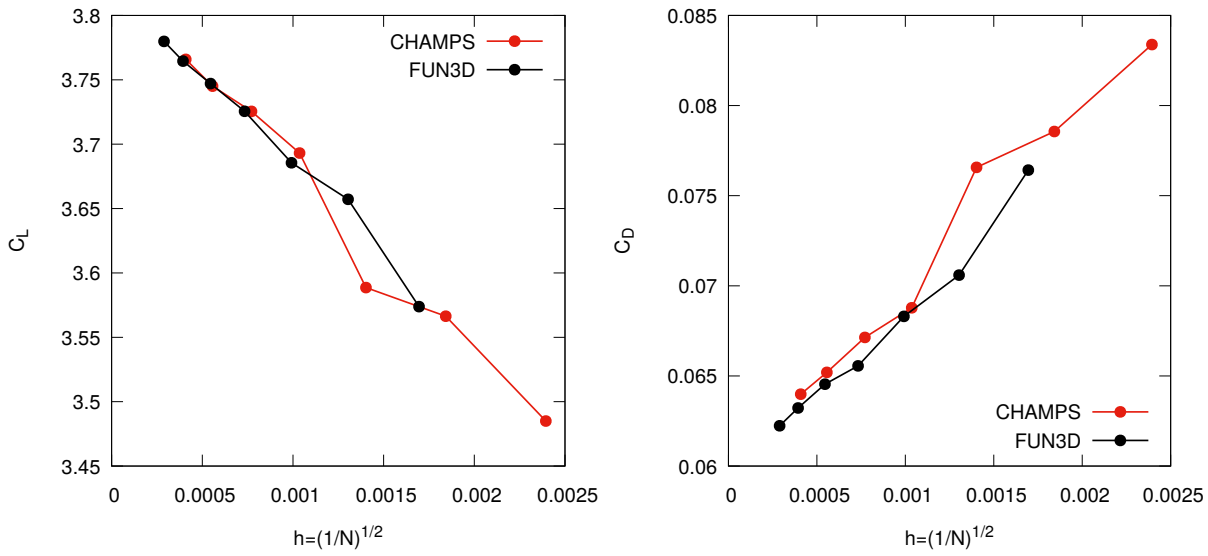
Figure 8a shows the grid convergence of the drag coefficient, compared to the CFD codes CFL3D and FUN3D. This figure shows that the drag coefficient converge towards the same value as the one of FUN3D and CFL3D as the grid gets refined. Moreover, the convergence of CHAMPS is similar to the one CFL3D for both turbulence models. The drag coefficient on the two coarse grid is higher that the one of CFL3D, but is well aligned for the finer grids.

Figure 8b shows the eddy viscosity profile at the location $x = 0.97$. One can observe the solution of CHAMPS is the same as the one of CFL3D and FUN3D for both the $k-\omega$ SST and Spalart-Allmaras turbulence models, thus confirming the proper implementation of both turbulence models in CHAMPS.

C. 2D high lift configuration

Another test case, which will be the (mandatory) test case 3a of the 4th AIAA CFD High Lift Prediction Workshop [18], is a 2-D three elements airfoil simulated with a Reynolds number of 5 million, an angle of attack of 16° and a Mach number of 0.2. The convective fluxes are again discretized with the Roe scheme, whereas the gradients for flow variables and the turbulence variables are computed with the GLSQ and Green-Gauss methods, respectively. No limiters are applied to the gradients in the piecewise linear reconstruction approach. Turbulence is modelled using the Standard SA model with $\tilde{\nu}_\infty = 3\nu_\infty$. The family of seven unstructured grids used in this paper is provided by the NASA Turbulence Model Ressources [19] and contains from around 174k to 5.98M elements.

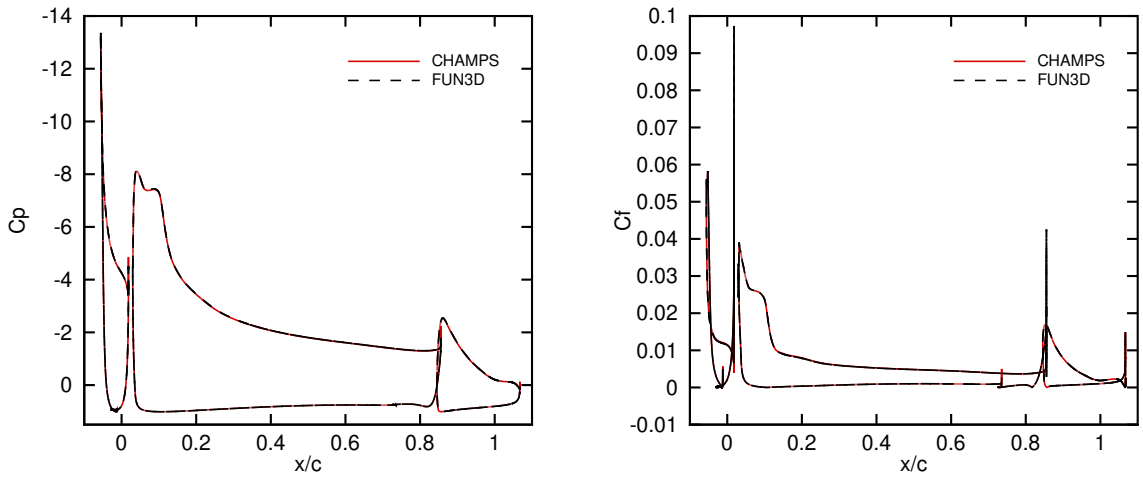
Figure 9 shows the grid convergence of the lift and drag coefficients, compared to the solution of FUN3D [19]. The data of FUN3D is modified so that the N is taken as the number of nodes because this solver is node-centered. In our opinion, this yields a fairer comparison because it is a measure of the actual number of degrees of freedom solved in the simulation. The solutions of CHAMPS are well aligned with the solutions of FUN3D.



(a) Lift coefficient **(b) Drag coefficient**

Fig. 9 Three elements airfoil grid convergence.

Figure 10 shows the surface pressure coefficient obtained with CHAMPS compared to the solution of FUN3D [19], on the finest grid. Again, the two solutions are almost identical visually. This finalize the verification of the Standard SA model.



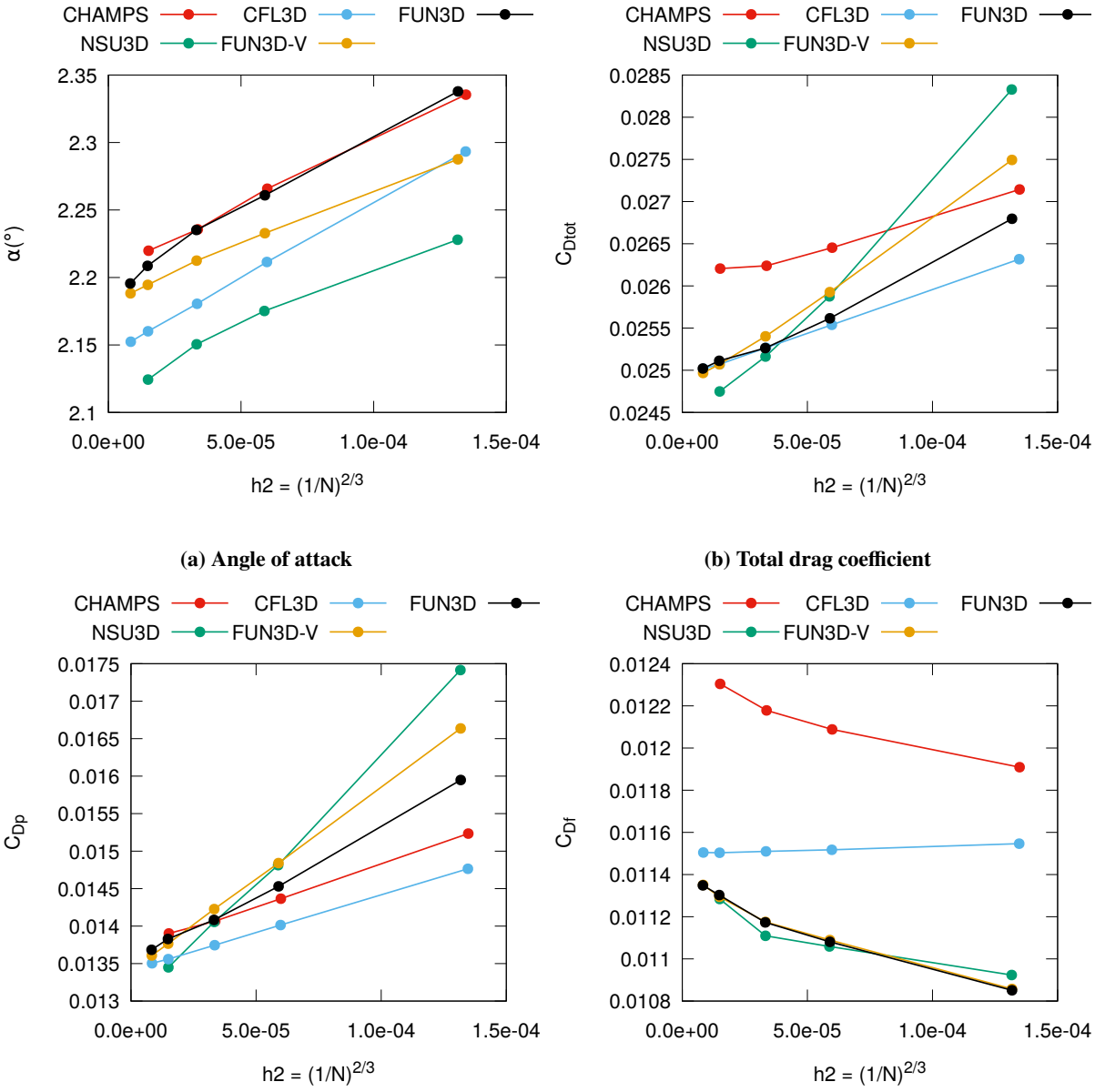
(a) Pressure coefficient **(b) Friction coefficient**

Fig. 10 Three elements airfoil surface coefficient.

D. Fifth drag prediction workshop

To finalize the demonstration of the capability of CHAMPS, the first case of the Fifth AIAA CFD Drag Prediction Workshop [20] has been selected. The geometry simulated is the NASA Common Research Model (CRM) in Wing-Body configuration. The family of hexahedral (structured) grids provided by the committee is used after conversion. This

family of grid contains six levels of grids from tiny to superfine. The four coarser levels with around 639k elements to around 17.25M elements are simulated. This test case is a grid convergence study at a Mach number of 0.85, a Reynolds number of 5 million and a constant lift coefficient of 0.5. The constant lift coefficient is achieved by using a proportional controller to modify the angle of attack while iterating. The convective fluxes are discretized with the Roe scheme. The gradients for the flow variables are computed using the weighted least-square approach, whereas, the Green-Gauss method is used to compute the gradients of the turbulence variables. The Venkatakrishnan limiter is applied to the gradient in the piecewise linear reconstruction. The K constant of the Venkatakrishnan limiter is increased to 100 to reduce the numerical dissipation while maintaining the stability of the solver. The Standard SA turbulence model is used with $\tilde{\nu}_\infty = 3\nu_\infty$.



(a) Angle of attack

(b) Total drag coefficient

(c) Pressure drag coefficient

(d) Friction drag coefficient

Fig. 11 CHAMPS DPW5 case 1 results.

Figure 11 shows the grid convergence of the angle of attack and the drag coefficient. The results of CHAMPS are

compared to the results of the CFD codes NSU3D, CFL3D and FUN3D presented by Park et al. [21]. The angles of attack required to yield a lift coefficient of 0.5 as well as the pressure drag coefficient are in line with the ones obtained with the other codes. The total drag is higher by ten drag counts for CHAMPS and this can be attributed to a higher friction drag coefficient. However, if one compares the solution of CHAMPS to the data of every participant to the DPW5 [22], one can notice a subset of results with a comparable friction drag and the total drag coefficient of CHAMPS is in the middle of the distribution. Hence, variability is observed in the results of the workshop and CHAMPS compares favorably with other state-of-the-art CFD solvers.

IX. Conclusion

In this paper, the use of the Chapel programming language to perform Computational Fluid Dynamics simulations is studied from the perspective of an academic research laboratory. Two different CFD applications, developed internally, were used to assess the performance and productivity of the Chapel programming language.

The first is a translation of a two-dimensional structured finite-volume code initially written in C. Only few basic features from the Chapel language were used to improve the code, like domains, iterators, and it resulted in at least the same efficiency in terms of overall simulation time for a shared-memory parallel flow simulation on a multi-block grid. Following this positive result, the second software, named CHAMPS, was developed as a new unstructured CFD development code that makes use of most of Chapel features, resulting in a flexible code structure that will be used for extensive multi-physics simulations.

Overall, the programming experience with Chapel is positive. Code prototyping is fast, which eases the addition of new features to a software such as CHAMPS. In an academic research lab, the added productivity from the programming language itself has a critical impact on how much science is performed by saving time in the initial development phase and the code maintenance phase as well. The achieved productivity with Chapel is similar to what is observed when using Python, but with performance comparable to C and C++, thus making the Chapel programming language well suited for the development of CFD software.

The weak and strong scaling analyses performed on a Cartesian grid show that CHAMPS scales well up to 256 compute nodes or 9216 cores. The scaling results without reductions confirm that the internode communication handled by Chapel scales well and that the main parallel work in CHAMPS is efficiently distributed on the available compute nodes.

The flow solver was verified on a case from the Fifth Drag Prediction Workshop (DPW5) against well-established CFD software, showing similar grid convergence for the lift and drag coefficient. The pressure distribution was also compared directly to FUN3D for a three elements airfoil, confirming the accuracy of CHAMPS. Other standard verification cases also confirm that the RANS equations and the turbulence models are correctly solved.

Future work will focus on multi-physics simulation for ice-accretion prediction and aeroelastic analyses. We believe that the Chapel programming language will facilitate the implementation of these complex models, while enabling new possibilities in terms of simulation.

X. Acknowledgements

This work benefited from the support of: Natural Sciences and Engineering Research Council of Canada (NSERC), MITACS and the Canada Research Chair Program. Calculations were performed on Compute Canada/Calcul Quebec clusters. Large-scale scalability simulations were performed on a cluster provided by Hewlett Packard. Enterprise.

References

[1] Chamberlain, B. L., "Chapel," *Programming Models for Parallel Computing*, 2015, pp. 129–159.

[2] Bourgault-Côté, S., Ghasemi, S., Mosahebi, A., and Laurendeau, E., "Extension of a Two-Dimensional Navier-Stokes Solver for Infinite Swept Flow," *AIAA Journal*, Vol. 55, No. 2, 2017, pp. 662–667. https://doi.org/10.2514/1.J055139.

[3] Jameson, A., Schmidt, W., and Turkel, E., "Numerical solution of the Euler equations by finite volume methods using Runge Kutta time stepping schemes," *14^{th} fluid and plasma dynamics conference*, AIAA Paper 1981-1259, Palo Alto, California, United States, 1981. https://doi.org/10.2514/6.1981-1259.

[4] Martinelli, L., "Calculations of viscous flows with a multigrid method," Thèse de doctorat, 1987. URL https://search.proquest.com/docview/303487785?accountid=40695, pp. 190.

[5] Mavriplis, D. J., and Jameson, A., "Multigrid Solution of the Navier-Stokes Equations on Triangular Meshes," *AIAA Journal*, Vol. 28, No. 8, 1990, pp. 1415–1425. https://doi.org/10.2514/3.25233.

[6] Jameson, A., and Baker, T., "Solution of the Euler equations for complex configurations," 6^{th} *Computational Fluid Dynamics Conference*, AIAA Paper 1983-1929, 1983. https://doi.org/10.2514/6.1983-1929.

[7] Roe, P., "Approximate Riemann solvers, parameter vectors, and difference schemes," *Journal of Computational Physics*, Vol. 43, No. 2, 1981, pp. 357–372. https://doi.org/10.1016/0021-9991(81)90128-5.

[8] Barth, T. J., and C., J. D., "The design and application of upwind schemes on unstructured meshes," 27^{th} *Aerospace Sciences Meeting*, AIAA Paper 1989-0366, 1989. https://doi.org/10.2514/6.1989-366.

[9] Blazek, J., *Computational Fluid Dynamics: Principles and Application*, 3rd ed., Elsevier, 2015.

[10] Shima, E., Kitamura, K., and Haga, T., "Green–Gauss/Weighted-Least-Squares Hybrid Gradient Reconstruction for Arbitrary Polyhedra Unstructured Grids," *AIAA Journal*, Vol. 51, No. 11, 2013, pp. 2740–2747. https://doi.org/10.2514/1.J052095.

[11] Venkatakrishnan, V., "Convergence to Steady State Solutions of the Euler Equations on Unstructured Grids with Limiters," *Journal of Computational Physics*, Vol. 118, No. 1, 1995. https://doi.org/10.1006/jcph.1995.1084.

[12] Spalart, P. R., and Allmaras, S. R., "A One-Equation Turbulence Model for Aerodynamic Flows," *Recherche Aerospatiale*, Vol. 94, No. 1, 1994, pp. 5–21.

[13] Spalart, P. R., and Allmaras, S. R., "A One-Equation Turbulence Model for Aerodynamic Flows," 30^{th} *Aerospace Sciences Meeting and Exhibit*, AIAA Paper 1992-0439, 1992. https://doi.org/10.2514/6.1992-439.

[14] Menter, F. R., "Improved Two-Equation k-omega Turbulence Models for Aerodynamic Flows," Tech. rep., NASA TM 103975, Oct. 1992.

[15] Poinot, M., and Rumsey, C. L., "Seven keys for practical understanding and use of CGNS," *2018 AIAA Aerospace Sciences Meeting*, AIAA Paper 2018-1503, 2018. https://doi.org/10.2514/6.2018-1503.

[16] Vassberg, J. C., and Jameson, A., "In Pursuit of Grid Convergence for Euler and Navier-Stokes Equations," *Journal of Aircraft*, Vol. 47, No. 4, 2010, pp. 1152–1166. https://doi.org/10.2514/1.46737.

[17] Rumsey, C., "NASA Langley Research Center Turbulence Modeling Resource," https://turbmodels.larc.nasa.gov/index.html, 2020.

[18] Rumsey, C., "4^{th} AIAA CFD High Lift Prediction Workshop (HLPW-4)," https://hiliftpw.larc.nasa.gov/, 2020.

[19] Rumsey, C., "VERIF/2DMEA: 2D Multielement Airfoil Verification Case - Intro Page," https://turbmodels.larc.nasa.gov/multielementverif.html, 2020.

[20] Morrison, J. H., "5^{th} AIAA CFD Drag Prediction Workshop," https://aiaa-dpw.larc.nasa.gov/Workshop5/workshop5.html, 2015.

[21] Park, M. A., Laflin, K. R., Chaffin, M. S., Powell, N., and Levy, D. W., "CFL3D, FUN3D, and NSU3D Contributions to the Fifth Drag Prediction Workshop," *Journal of Aircraft*, Vol. 51, No. 4, 2014, pp. 1268–1283. https://doi.org/10.2514/1.C032613.

[22] Ed Tinoco, O. B., David Levy, and the DPW Organizinf Committee, "DPW5 Summary of Participant Data," https://aiaa-dpw.larc.nasa.gov/Workshop5/presentations/DPW5_Presentation_Files/14_DPW5%20Summary-Draft_V7.pdf, 2012.