42nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit
9 - 12 July 2006, Sacramento, California

AIAA 2006-5274

CHIMPS: A High-Performance Scalable Module for Multi-Physics Simulations

J. J. Alonso, S. Hahn, F. Ham, M. Herrmann, G. Iaccarino. G. Kalitzin, P. LeGresley, K. Mattsson, G. Medic, P. Moin, H. Pitsch, J. Schlüter[*], M. Svärd, E. Van der Weide, D. You, X. Wu[†]

Center for Integrated Turbulence Simulations, Stanford University, Stanford, CA 94305, U.S.A.

As computational methods attempt to simulate ever more complex physical systems the need to couple independently-developed numerical models and solvers arises. This often results from the requirement to use different physical or numerical models for various portions of the domain of interest. In many situations it is also common to use different physical models that influence each other within the same domain of interest. The interaction between these models normally requires an exchange of information between the participating solvers. When the solvers that exchange information are distributed over a large number of processors in a parallel computer, the problem of exchanging information in an efficient and scalable fashion becomes complicated. This paper describes our efforts to develop a Coupler for High-Performance Integrated Multi-Physics Simulations library, CHIMPS, that can enable the exchange of information between solvers and that automates the search, interpolation and communication processes in order to allow the developer to focus on other matters of interest such as the appropriate strategies to couple the solvers in an accurate and stable fashion. Our basic approach, the underlying technology, the CHIMPS API, and a number of examples are presented. In addition, a series of appendices are included with actual sample code that can be used to become familiar with the CHIMPS library.

I. INTRODUCTION

Although during the last three decades there has been an explosive growth in the use of computational tools, the process of developing scientific software has not changed considerably. This process typically follows three phases: the first phase consists of the definition of the software requirements; next a variety of mathematical and physical models are identified and, finally, a single computational code is written to fulfill all requirements. With this approach, additional features can only be added if they are compatible with the overall structure of the software/algorithms already implemented. In addition, updates and modernization of the code structure typically require a complete rewrite and, therefore, a considerable investment. Moreover, portions of the code that may be considered to be *legacy* (for which little or no expertise is available) can be difficult to interface with the rest of the software.

An alternative approach is to build flexible computational infrastructures that are made up of several independent solvers that can be easily integrated. Each component performs a specific task and addresses a specific physical aspect of the problem. The ability to easily integrate these solvers ensures that new features or updated models can be included without disrupting the entire infrastructure and with a reasonable level of effort. An additional benefit of this strategy is that various existing component modules or solvers can be rapidly combined to solve new problems without the tremendous overhead needed to create a new environment from scratch.

Large multidisciplinary problems require the collaboration of a large group of scientists and the development of an extensive simulation environment. Within the Department of Energy Advanced Simulation

[*]Current address: Nanyang Technological University Singapore, Aerospace Division
[†]Authors are listed in alphabetical order.

Copyright © 2006 by the authors. Published by the American Institute of Aeronautics and Astronautics, Inc., with permission.

and Computing (ASC) program, the Center for Integrated Turbulence Simulations (CITS) at Stanford University is performing simulations of the flow through an entire jet engine. The radically different physics, length scales and dynamics of the problem prompt the use of a range of simulation technologies. In the compressor and turbine the flow is transonic, includes a very large number of moving parts and turbulence can be described using Reynolds averaging ideas. A compressible Unsteady Reynolds-Averaged Navier-Stokes (URANS), structured multiblock code is a good choice to capture the flow features in the turbomachinery components with reasonable cost. On the other hand, in the combustor, low speed air is mixed with liquid fuel and reactions take place. Consequently, an unstructured mesh Large-Eddy Simulation (LES), reactive flow solver is required in the combustor to reproduce the intricate injector passages and to account for the proper levels of mixing and cooling flows. In addition, the modeling of the liquid fuel atomization requires the detailed computation of the evolution of liquid sheets and their interaction with the gas phase that ends with atomization. Finally, if aero-structural phenomena are to be accounted for, it is necessary to couple the fluid and structural solvers through an appropriate fluid-structure interface.

The interaction between all of these solvers is usually accomplished via communication of information computed by each solver. This interaction can be based on artificially-imposed boundary conditions or simply on information interpolated between the grids that each solver computes on. Although the problem of interpolation may appear straightforward, it is in fact complicated by two fundamental issues. Firstly, in modern computing environments, the participating codes are usually domain decomposed into a large number of processors. The data to be interpolated can reside anywhere in the distributed memory of a parallel computer. The search process that identifies the mesh cells that provide information to a given solver (and the processor that they reside on) and the ensuing exchange of information have often been described as the $M \times N$ problem in the literature.[1] The number of processors that are typically involved in these search, interpolation, and communication patterns can be very large. The efficient and scalable (both in CPU time and in used memory) solution of the $M \times N$ problem is not straightforward. Secondly, although linear (bi-linear, tri-linear) interpolation is relatively straightforward to implement, it is unable to guarantee the accuracy, conservativeness, and stability of the coupled solution except in the limit of an infinitely fine mesh: the order of accuracy of a coupled solution can drop to first order if care is not taken to ensure accuracy.

The current implementation of CHIMPS addresses the first issue: the solution of the $M \times N$ problem using tri-linear mappings for the solution inside each element of a mesh. At the moment, CHIMPS does not provide any assistance with the second issue (it is left entirely to the user/developer) although future efforts will focus on addressing this issue within the context of the CHIMPS API.

This paper is organized to address the following goals:

- Provide an overview of the current and future functionality and of the performance of CHIMPS.
- Describe the full CHIMPS API and the underlying technology choices we have made to implement the functionality in the API.
- Provide some representative test cases (including the full source code in the appendices) so that this paper may be used as a Users' Manual for CHIMPS.
- Describe our experience during the development of CHIMPS and the ways in which our approach to code development and research have been changed.

The following sections focus on each of these goals and attempt to provide a detailed description and rationale for each of the topics.

II. CHIMPS Overview and Functionality

CHIMPS was conceived as a library (with its associated API) designed to facilitate the setup and execution of large-scale integrated simulations where interpolated information is requested from solver domains that are distributed (in the parallel computing, domain decomposition sense). As such, CHIMPS is merely an interpolation engine (albeit an efficient, distributed, and scalable one) that, together with some participating solvers and a main driver script, can be used to create integrated simulations. This efficiency is achieved by relieving the developer from the tedious task of figuring out how to interpolate and exchange information between a number of distributed-memory solvers: a task common to many integrated simulations that is, in principle, conceptually straightforward but, in practice, is difficult to do with reasonable performance and scalability. It is our intention to incorporate additional functionality of this kind into the CHIMPS library: operations that are common to a variety of integrated simulations and can be abstracted so that the user may use this functionality in new and unexpected ways. In the future this may include hole-cutting approaches for general unstructured overset computations, conservative interpolation techniques, and methodologies to guarantee the accuracy and stability of the participating simulations.

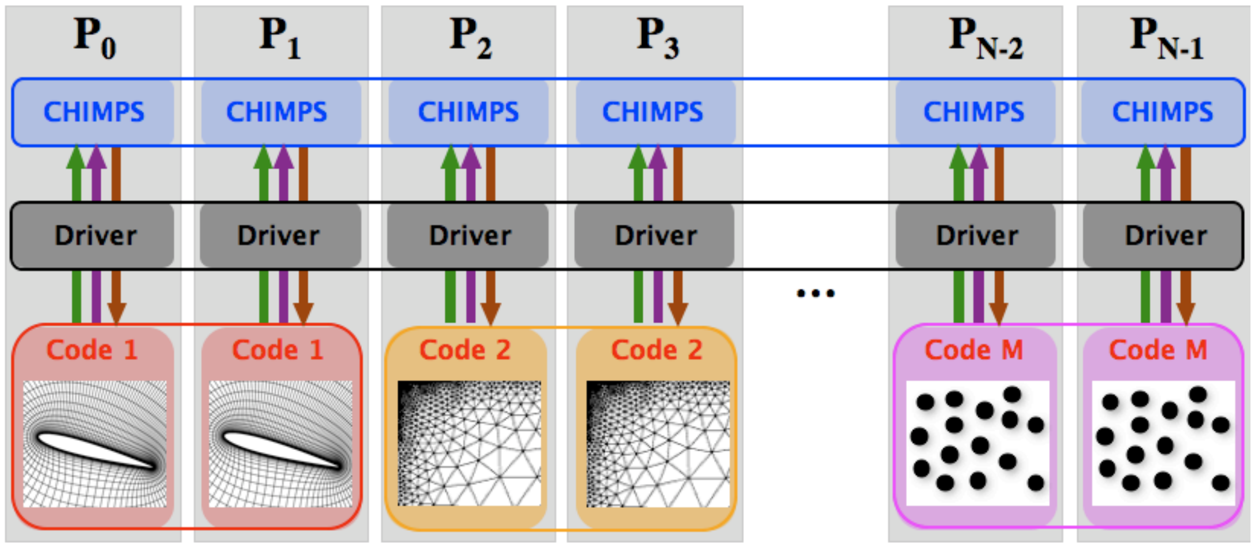

Figure 1. A schematic representation of the CHIMPS library, the necessary driver program/script, and the participating parallel solvers. The green arrows represent spatial locations at which one of the participating solvers requires interpolated information. The purple arrows represent the mesh and solution information that each solver provides so that CHIMPS may service interpolation requests from that or other solvers. The brown arrows represent the return transfer of interpolated information requested by the solvers from the CHIMPS library.

A schematic of a typical integrated simulation that uses CHIMPS is presented in Figure 1. It consists of three basic components:

- The participating solvers/applications (labeled Code 1 through Code M).
- The driver program / script (currently both Python, Fortran, and C/C++ are supported).
- The CHIMPS library / extension module.

The solvers/application are the basic building blocks providing the algorithms/physical models required to perform the simulations on arbitrary portions of the entire domain. These are typically (although not necessarily) parallel, domain decomposed applications that require information from each other in order to accomplish an integrated task. In order to interact with CHIMPS, these applications must be written in a language that can be called by either Fortran, C/C++, or Python. With these choices virtually any scientific computing application can interact with CHIMPS.

The driver program / script (depending on the programming language in which it is written), on the other hand, is responsible for coordinating the execution and information exchange between all the participating applications. It is truly the program / script that is meant to be written by the engineer or scientist in order to create an integrated simulation. In our view, the desirable properties of this driver program / script are as follows:

- It should be relatively concise.
- It should embody the overall strategy to couple multiple codes and, as such, it is responsible for such things as load balance, synchronization, information exchange and the stability and accuracy of the resulting integrated calculation.

- It should mainly consist of API calls (to the applications and to CHIMPS) that broker the information exchange.

Finally, the CHIMPS library / extension module, is the communication hub that enables scalable communication between multiple applications through the use of an efficient parallel search engine. Note that CHIMPS is provided as both a library (so that Fortran and C/C++ driver programs can link it) and a Python extension module that Python driver scripts can import.

For the remainder of this paper, we will use the terminology *driver program* and *driver script* interchangeably to indicate that our discussion is completely independent of whether the driver is written in traditional languages such as Fortran and C/C++ or in a scripting language such as Python. The structure of a typical driver program contains the following steps:

1. **Initialization**: the parallel environment for the integrated simulation is created and defined, the application codes are assigned to the available processors and the computation is properly load balanced.

2. **Registration**: each application defines its computational domain, grid and associated data, and interpolation locations for CHIMPS so that future exchanges of information (via interpolation) can take place.

3. **Interface set-up**: the logical connections between the locations where interpolated data are needed and the grid/data information from where the interpolated data may be obtained. Each such connection is termed an *interface*. Multiple interfaces can be created.

4. **Data exchange**: data at all interfaces is updated so that the interaction between solvers takes place.

5. **Application execution**: each component application performs the necessary solution steps before information is exchanged.

6. **Exit and clean up processes.** Termination of applications and CHIMPS and release of memory used during the integrated simulation.

CHIMPS has been conceived to operate in a completely dynamic environment and therefore, steps 2 to 6 may be repeated multiple times in typical integrated simulations. For example, simulations that use grids (possibly deforming) in relative motion may need to redefine these grids every time that the grid or interpolation location information changes.

The application codes are both *providers* and *receivers* of information. The transfer of information is accomplished through the *driver* script and, for that reason, the engineer / scientist writing the script must be able to *get* / *set* data from/to the participating solvers. CHIMPS imposes no restrictions on the form or structure of this interaction; it can happen through file I/O, by direct argument passing through a series of *helper* functions/subroutines or through a thin layer that interfaces the application codes with a scripting language such as Python, for example. In our work, the application codes have well-defined APIs (the second type of interaction mentioned above) and, therefore, the *driver* program consists of a series of API calls (to the application codes and to CHIMPS) that orchestrate the entire integrated simulations. Detailed examples of such *driver* programs are provided in Section V. A number of other examples are provided with the CHIMPS distribution.

It is important that the CHIMPS functionality be accessed from a well-defined, stable, and logical API so that codes that use CHIMPS can benefit from future improvements to the search, interpolation, and communication algorithms ("under the hood") without the need to modify the *driver* programs. For this reason, an initial version of the CHIMPS API was set up in 2004.[?] During the first three months of 2006, the CHIMPS task force was setup and a complete re-definition of the API was tackled based on lessons learned from the initial version and on a host of potential applications brought to the table by several developers and users. The result is an API that is flexible and upgradable. Although additional functionality may be included in the future, the existing API calls are not likely to be modified (except for the addition of a wider variety of flags for the various routines).

The fundamental concept used by CHIMPS for the exchange of information between participating solvers is that of an *interface*. An interface (in CHIMPS parlance) is the communication conduit between two user-defined *geometric* entities that exchange information via interpolation. These *geometric* entities can currently be either full unstructured meshes (structured and multi-block structured meshes can be provided to

CHIMPS by constructing an explicit connectivity) or list of points (to represent either interpolation locations or, possibly, particle locations). The current mesh definition is through an unstructured mesh format that supports the four basic three-dimensional elements primitives (tetrahedra, prisms, pyramids and hexahedra) and is expected to support in the near future the two fundamental two-dimensional primitives (triangles and quadrilaterals). Point lists are created with the Cartesian coordinates of the points themselves (internally CHIMPS provides the capability to transform these point coordinates to a Cylindrical coordinate system for purposes of interpolation). In CHIMPS, all geometric entities are assigned a name that acts as a handle for any future operations. There is no limit on the number of geometric entities that can be handled by CHIMPS.

After *geometric* entities are defined and named (see examples in Section V) they are grouped pairwise into the *interfaces*. An *interface* may have any type of *geometric* entity on either side. The kind of interpolation operation that will be performed across the *interface* is defined during the creation of the interface (using a call to `chimps_setInterface`) based on both the participating *geometric* entities and the interpolation operation flag. For example, if on both sides of the interface we have a mesh and a point list, the logical operations to be performed are either a failsafe search (the default) specified by the `CHIMPS_INTERPOLATE` flag to `chimps_setInterface` (this is essentially a containment search followed by a minimum distance search for those points that do not have a donor element, e.g. points that fall outside of all the elements in the mesh provided to that *interface*) or a containment search specified by the `CHIMPS_INTERPOLATE_CONTAINMENT` flag to `chimps_setInterface` (where the donor elements for the specified points are found). If, instead, we had two meshes on both sides of the *interface* then the logical interpolation operation may be a volume integration of the overlapping meshes as specified by the `CHIMPS_INTEGRATE` flag to the call to `chimps_setInterface`. For the time being, volume integrations (see Section E) are not supported in this fashion: they can only take place between a point list (the source of information) and a mesh (the destination/receiver of information).

Once the *interfaces* are created, it remains to carry out the interpolations requested. This is done through calls to `chimps_updateInterface` which can update any or all the interfaces currently registered into CHIMPS. Of course, prior to the actual interpolation and communication operations, CHIMPS must be aware of the latest donor data from which the interpolated information will be obtained. CHIMPS is made aware of this data by registering it with calls such as `chimps_setMeshData` and `chimps_setPointData`. The `chimps_updateInterface` call carries out all the necessary interpolations and communication. Once this call has concluded, the interpolated data can be retrieved from the CHIMPS internal data structures via calls to `chimps_getMeshData` and `chimps_getPointData`.

Some details associated with the implementation of these routines in the current CHIMPS library are discussed in Section IV, including how processor groups and their associated communicators are built and used, and which calls are globally collective vs. locally collective on one or a set of processor groups.

III. Technical Components

A. The parallel environment

CHIMPS assumes that the parallel environment is an MPI-based distributed-memory environment where the total number of available processors are broken up into a set of disjoint groups where each group has a unique user-specified name. A single group of processors can run one or more of the applications required by the coupled simulation and the group name should reflect these associated applications for clarity in the driver script.

The CHIMPS initialization routine `chimps_initialize` is one of only a few globally blocking routines, and must be called from all processors in the global communicator chimpsComm (typically `MPI_COMM_WORLD`). `chimps_initialize` splits the global communicator based on a case-sensitive comparison of the passed group names, and ultimately returns these split communicators as group communicators for the subsequent initialization of the participating applications. `chimps_initialize` also builds a matrix of possible interface communicators representing all possible unions of the groups. This particular choice is of course not scalable in terms of the number of groups, N_g, however it is efficient for the relatively small numbers of groups typically used, and allows the subsequent setting and updating of interfaces to be called by only those groups participating in a particular interface. In the future, if CHIMPS is used to couple much larger numbers of processor groups, some compile-time or run-time preprocessing of the driver script will be necessary to determine the interface communicators actually required during the coupled simulation. Alternatively we could have required the engineer/scientist writing the driver script to specify all interface pairings *a priori* as part of the initialization routine, in an input file for example. However, to keep the API as clean as possible, to avoid forcing the user to specify the same information more than once, and to keep the driver scripts as readable and self-explanatory as possible, this was not done.

The routines that register geometries and their associated data with CHIMPS (e.g. `setMeshGeom`, `setMeshData`, etc.) must be called collectively from only the processors in the group that owns the geometry. At present, these routines actually create a copy of that information in memory managed by CHIMPS. There is of course a memory overhead associated with this copy, however the choice to make a copy in a form suitable for CHIMPS means that we impose no restrictions or requirements on the data structures associated with the applications themselves. The only requirement is that they conform to the CHIMPS API in providing their mesh and data.

The interface routines must be called collectively from all processor groups participating in the interfaces being created or updated. In addition, the `updateInterface` command takes an array of interface names to allow the CHIMPS implementation to optimize the required communication when multiple, independent interfaces are updated at the same time (as is often the case, at the end of a global time step for example).

B. The search algorithm

Geometric searches are an important part of CHIMPS as data must be interpolated on several grids. Moreover, due to relative motion of the different parts, it is possible that these interpolations must be repeated every time step in an unsteady simulation. Hence an efficient search algorithm, whose local memory usage is limited, is a prerequisite. The reason for this attention to the memory requirement is that the total amount of memory available on modern MPP's is very large, but that is often caused by the large number of processors ($O(10,000)$); on a per processor basis and in extreme cases, only 256 MBytes may be available (as in the DoE Lawrence Livermore BlueGene/L computer).

For these reasons, searches are performed in a memory-scalable way on the processors containing the source geometry (e.g. the mesh) using a parallel binary tree structure called a parallel Alternating Digital Tree (ADT).[2] We note that this approach will only be scalable in terms of computation time when the requests are distributed more-or-less evenly over the source processors. For cases where all the points being requested are contained on only one or a small subset of the source processors, this approach is clearly not scalable and better alternatives exist that distribute the computational work more evenly amongst the participating processors; see for example the "rendezvous" approach described in Plimpton *et al.*,[1] where both source mesh and destination points are migrated to a new partition where the actual interpolation is performed. While the current implementation of CHIMPS does not use such an algorithm, the API is specifically designed such that it does not preclude the use of these more formally scalable approaches in the future.

The parallel ADT algorithm proceeds as follows. First the bounding boxes of the elements of the local grid are created. These bounding boxes are characterized by the lower left and the upper right coordinate and are therefore equivalent to a point in 6 space dimensions. Every processor creates a local 6-dimensional ADT of the bounding boxes (see figure 2). Logical rather than geometric splits are used to avoid degenerate trees. The root leaves of the local trees are made available to all processors, see figure 3, such that the geometric range covered by each local tree is known on every processor. Hence possible target trees can be

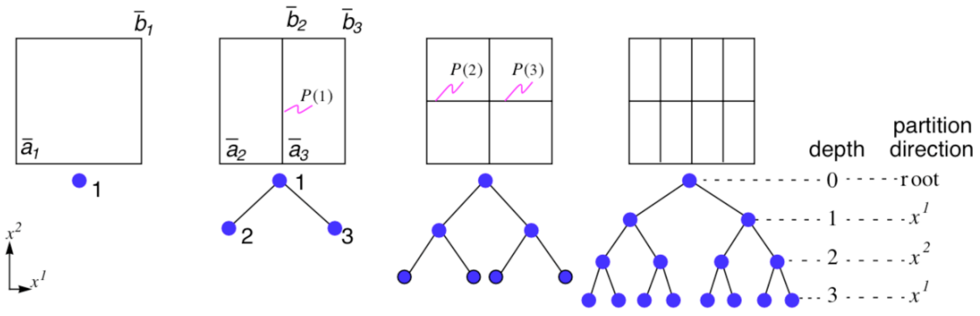

Figure 2. Building of the local ADT. Logical rather than geometrical splits are used to avoid degenerate trees. Figure taken from.[2]

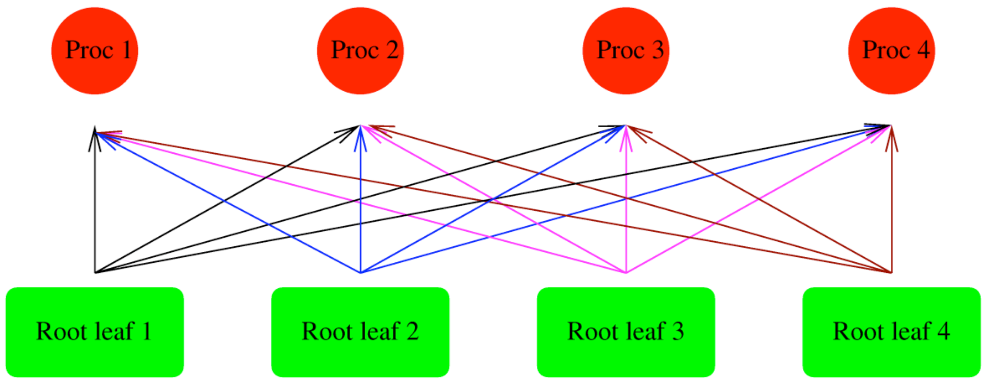

Figure 3. Gathering of the root leaves on all processors.

determined for every point to be searched.

The global algorithm proceeds as follows:

- Determine for every point to be searched the target trees and determine the number of local trees to be searched for the current set of points.
- Make this number available to all processors $\Longrightarrow$ Every processor can determine the total number of points it has to search in the local tree.
- Determine the number of search rounds to avoid a possible memory bottleneck.
- Determine the sending and receiving processors for every search round.
- Loop over the search rounds
  - MPI_Alltoall to request data from processors.
  - Send coordinates to be searched to the target processors.
  - Perform the interpolation in the local tree to overlap between communication and computation.
  - Loop over the number of processors for which I need to interpolate data in my local tree
    - Receive the coordinates.
    - Interpolate the data in my local tree.
    - Send interpolation data back to the requesting processor.
  - Loop over the number of processors to which I sent coordinates.
    - Receive the interpolation data.

The local search algorithm consists of a standard tree traversal, which leads to limited number of possible target elements. A simple linear search is then performed to find the element, which either contains the point to be searched or minimizes the distance to this point. For the actual interpolation the standard (tri-)linear interpolation formula using finite element base functions[3] are used.

C. Interface Coupling Numerical Issues

Although the stability and accuracy of integrated numerical simulations is currently left up to the user, our group has been conducting research into numerical coupling techniques that can guarantee these properties. As an example we consider the case of coupling two solution domains with a vortex convecting across the interface.

We evaluate and compare two different approaches to couple two computational domains. The first technique is an overlap technique, where we make use of ghost cells in the overlap region in combination with linear interpolation (the current practice using CHIMPS). The second is a non-overlap technique that uses Summation By Parts (SBP) operators (see for example[4]) in combination with the Simultaneous Approximation Term (SAT) technique [5] to impose the interface conditions. The non-overlap technique can be proven stable and accurate in the case of matching grid lines, but here we use linear interpolation to transfer the solutions along the line defining the common interface. In this test we use standard second-order finite difference schemes to discretize space and use the standard fourth-order accurate Runge-Kutta method for time integration.

We will compare the two methods for the 2-D compressible Euler equations on a 2-block domain. To evaluate accuracy and stability, we use an exact analytic solution describing an isentropic vortex (see for example[6]) as initial and boundary data. We run the vortex at Mach number 0.3 which means that we have both right- and left-going charecteristics. The computational grid and the initial solution are shown in Figure 4. The SAT method is remarkably robust and second-order accurate (design order). See Table 2 for more details. The corresponding convergence study for the overlap technique is found in Table 1 showing first-order accuracy. In these tables, N represents the number of points along the interface in the coarse side of the mesh, ρ, u, v, and e correspond to the fluid density, Cartesian velocity components and internal energy, and $q()$ indicates the order of accuracy obtained for each of the variables (this is more significant in the limit of a very fine mesh). By performing a long time integration we conclude that both methods are stable, but the non-overlap SAT technique is more accurate and certainly cheaper, since we only have to do interpolation along the interface mesh points. It must be understood that the two schemes differ only at the grid-points along the interface (at x=5 in Fig. 4). From this simple example we understand that merely providing linear interpolation capabilities is only a necessary condition to create high-fidelity integrated simulations: a significant amount of work must be done in both the *driver* program and the component applications / codes themselves in order to guarantee the accuracy of the resulting integrated simulation.

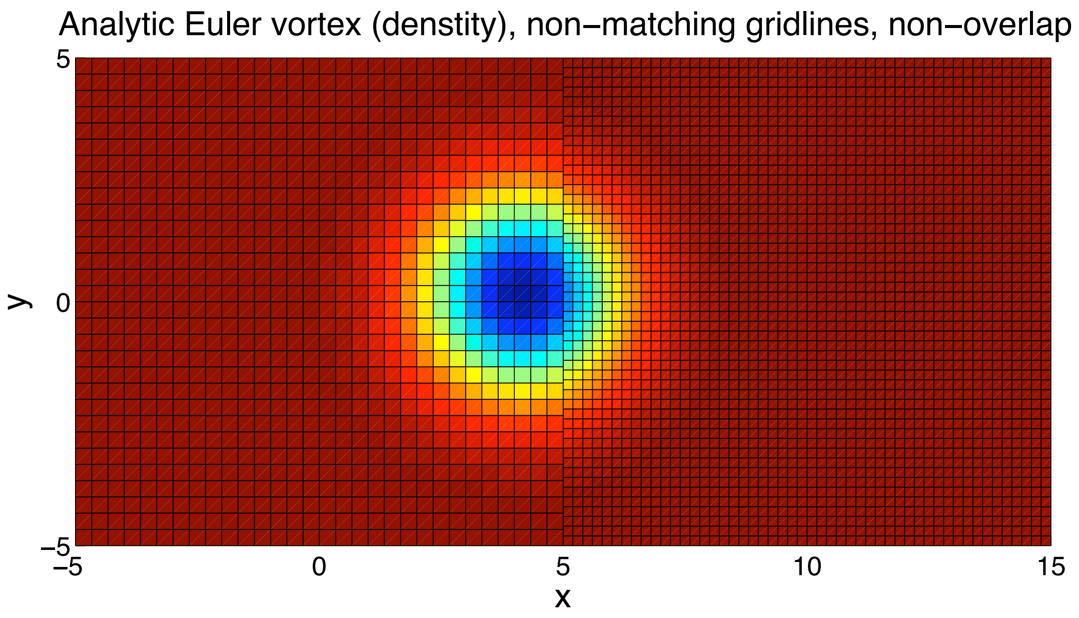

Figure 4. The Euler vortex problem for a SAT non-matching grid interface

D. Cartesian vs. Cylindrical coordinates

In theory, the current CHIMPS implementation can support the exchange of information between geometries described in any consistent orthogonal basis. Care must be taken, however, for cases where the mapping to physical space is not single-valued, such as cylindrical coordinates near the planes $\theta = 0, 2\pi$. Additionally, problems arise when vector data crosses periodic planes in a participating application and must be transformed appropriately. For example, the case of vector data with cylindrical periodicity arises in the annular simulations of turbomachinery components described below. To handle these and other cases, additional parameters can be specified after geometries are registered with CHIMPS using the `setMeshParam` and `setPointParam` routines. Some examples of their use are provided below.

N	$l_2(\rho)$	$q(\rho)$	$l_2(u)$	$q(u)$	$l_2(v)$	$q(v)$	$l_2(e)$	$q(e)$
41	-5.36		-4.29		-4.22		-3.90	
61	-5.59	1.30	-4.51	1.30	-4.53	1.76	-4.13	1.30
81	-5.74	1.25	-4.67	1.22	-4.73	1.65	-4.28	1.22
101	-5.86	1.22	-4.78	1.18	-4.88	1.55	-4.40	1.20
121	-5.96	1.19	-4.87	1.15	-5.00	1.46	-4.49	1.17
141	-6.04	1.17	-4.95	1.13	-5.09	1.39	-4.56	1.15

Table 1. $log(l_2 - error)$ and convergence, 2nd order case. Interface coupling for analytic Euler vortex. Overlapping technique on coarse to fine grid.

N	$l_2(\rho)$	$q(\rho)$	$l_2(u)$	$q(u)$	$l_2(v)$	$q(v)$	$l_2(e)$	$q(e)$
41	-5.50		-4.30		-4.14		-3.96	
61	-5.84	1.88	-4.64	1.90	-4.48	1.94	-4.29	1.89
81	-6.07	1.89	-4.88	1.93	-4.72	1.95	-4.53	1.91
101	-6.26	1.91	-5.07	1.94	-4.91	1.96	-4.72	1.93
121	-6.41	1.93	-5.22	1.95	-5.07	1.95	-4.87	1.94
141	-6.54	1.94	-5.35	1.96	-5.20	1.96	-5.01	1.95

Table 2. $log(l_2 - error)$ and convergence, 2nd order case. Interface coupling for analytic Euler vortex. Penalty SAT technique on coarse to fine grid.

E. Interpolation vs. Integration in CHIMPS

Volume integration interfaces play an important role in finite volume solvers. By definition, a stored quantity ϕ_i in a finite volume code represents a control-volume averaged quantity,

$$\phi_i = \int_{V_i} \phi d\boldsymbol{x}/V_i \ , \tag{1}$$

with V_i the cell control volume. This implies that if two codes are to be coupled that employ overlapping or intersecting grids, Eq. 1 must be solved to determine ϕ_i. For simplicity, it will be assumed in the following that two codes, code A and code B, are to be coupled, with code B consisting of a fine mesh (blue) that overlays parts of a code A coarse mesh (red), see left part of Fig. 5.

Let i be the code A cell in the lower right corner of Fig. 5. To calculate ϕ_i from code B data ϕ_j exactly, CHIMPS has to first identify all cells $\mathcal{G}$ of code B, that have any common volume with V_i (large green dots in center of Fig. 5), calculate the common volume $V_{i,j} = V_i \subset V_j$ (shaded blue areas in center of Fig. 5),

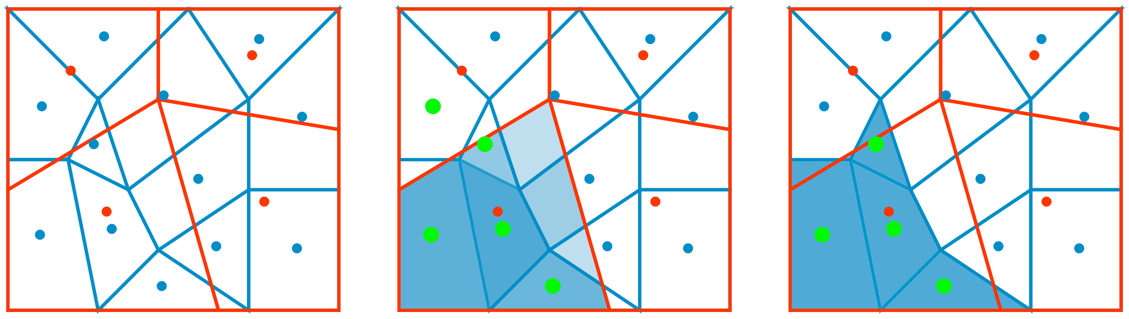

Figure 5. CHIMPS volume integration code B (blue grid) to code A (red grid), exact integration (center), approximate integration (right).

and then solve

$$\phi_i = \sum_{j \in \mathcal{G}} \phi_j V_{i,j} / V_i \ . \tag{2}$$

This exact formulation will be implemented in a future version of CHIMPS. The current version employs a simpler, faster, approximate scheme that reverts back to the exact formulation in the case of exactly overlapping grids, i.e. $V_{i,j} = V_j$ or $V_{i,j} = 0$. The right side of Fig. 5 shows the more general case of not exactly overlapping grids. In a first step, CHIMPS identifies all code B control volume centroids $\mathcal{F}$ that are within V_j (large green dots in right of Fig. 5) and then solves

$$\phi_i = \sum_{j \in \mathcal{F}} \phi_j V_j / \sum_{j \in \mathcal{F}} V_j \ . \tag{3}$$

To utilize the integration interface in a driver program, the mesh of code A has to be registered as a `mesh` entity and the cell centroids of code B have to be registered as a `point` entity. Furthermore, code B has to provide the cell control volumes V_j in the `codeB_getPointData` call. These then have to passed to CHIMPS in the `chimps_setPointData` with the variable name `CELL_VOLUME`. An example implementation of an integration interface in a driver program is listed in Appendix B.

The actual implementation of an integration interface in CHIMPS consists of the following steps:

1. generate ADT for code A's mesh
2. send code B's centroid coordinates $\boldsymbol{x}_j$ to code A's processors in a balanced way
3. find code A's mesh element $e(\boldsymbol{x}_j)$ that contains $\boldsymbol{x}_j$
4. send element information $e(\boldsymbol{x}_j)$ back to code B's processor that contains $\boldsymbol{x}_j$
5. build list of unique code A elements $e'(\boldsymbol{x}_j)$ on each code B processor
6. build communication structure for $e'(\boldsymbol{x}_j)$
7. on code B's processors calculate $\phi_{e'(\boldsymbol{x}_j)} = \sum_{j \in e'(\boldsymbol{x}_j)} \phi_j V_j$ and $V_{e'(\boldsymbol{x}_j)} = \sum_{j \in e'(\boldsymbol{x}_j)} V_j$
8. send $\phi_{e'(\boldsymbol{x}_j)}$ and $V_{e'(\boldsymbol{x}_j)}$ to code A processor containing $e'(\boldsymbol{x}_j)$
9. on code A processors calculate $\phi_i = \sum_{np \text{ codeB}} \phi_{e'(\boldsymbol{x}_j)} / \sum_{np \text{ codeB}} / V_{e'(\boldsymbol{x}_j)}$

Note, that step 1 can be skipped if code's A mesh has not changed, and steps 2-6 can be skipped, if neither code A's nor code B's mesh has changed. Furthermore, the ADT build in step 1 is actually the same as for an interpolation interface from code A to code B. Thus, in the common case that code B requests interpolated data from code A, and code A requests integrated data from code B, the ADT needs to be build only once, thereby reducing computational time, see also Sec. B.

The above algorithm ensures memory scalability even in the extreme case that all code B centroids are within a single code A mesh element. Also, the dual-step integration, first on code B's processors in step 7 and then on code A's processors in step 9 ensures that actual parallel communication is kept to a minimum, thereby ensuring excellent scalability.

IV. Software Verification

A. Analytical tests

The search and interpolation routines implemented in CHIMPS have been verified for all supported element types using analytic linear and non-linear functions (see figure 6). These tests confirm that the interpolations are linearly exact and second-order.

B. Scalability

One of the crucial features of the CHIMPS library is its ability to scale up well even on massively parallel, distributed memory computer systems. In order to quantify the scalability, a simple interpolation coupling of two codes, code A and code B is performed, see Appendix A. Both codes exist on the same group of processors. In code A, N hexahedral elements are defined in a unit cube, from which code B requests two interpolated scalar values at M randomly distributed points inside the unit cube. Both codes are themselves perfectly load-balanced.

Figure 7 shows the speed-up as a function of the number of processors for the first `updateInterface` and any subsequent `updateInterface` call. Note that for the first `updateInterface` call the ADT and communicators are build, the data is exchanged and the interpolation is performed, whereas for any subsequent `updateInterface` call, only the latter two steps have to be performed. To obtain a meaningful scaling relation, the problem size, that is N and M, is increased once the speed-up drops off due to an insufficient number of elements/points per processor. To insure compatibility, the speed-up of the larger problem is then scaled to the obtained speed-up of the smaller problem at the previous number of processors.

The observed speed-up for the initial `updateInterface` is slightly hyper-linear. This is due to the fact that the ADT build and search scales hyper-linearly due to the smaller tree sizes being built in each processor. The subsequent calls to `updateInterface` also show excellent scalability, with a noticeable drop-off in speed-up once the problem size becomes too small. However, in practical applications this drop-off is not significant, since in terms of wall-clock time, the subsequent calls to `updateInterface` are significantly faster than the initial call. For example, in the $N = 536.9 \cdot 10^6$ and $M = 134.2 \cdot 10^6$ case on 256 processors the wall-clock execution time for the initial call to `updateInterface` is 50.5s compared to 1.9s for any one of the subsequent calls.

V. APPLICATIONS

This section is meant to showcase a number of integrated simulations of increasing complexity that our group is carrying out with the help of CHIMPS. We start with a simple, static, three-code computation of the flow in an annular pipe. We then present a dynamic test case where two separate solvers are coupled to solve the flow around a pitching airfoil. The use of CHIMPS in this example occurs in the inner loop of the computation as the relative positions of the meshes change on every time step. We conclude this section with a brief description of our integration efforts in two major research programs: the DoE ASC program where we are trying to carry out computations of the flow through entire jet engines, and the DARPA Helicopter Quieting program where two solvers are being coupled to simulate the flow physics and

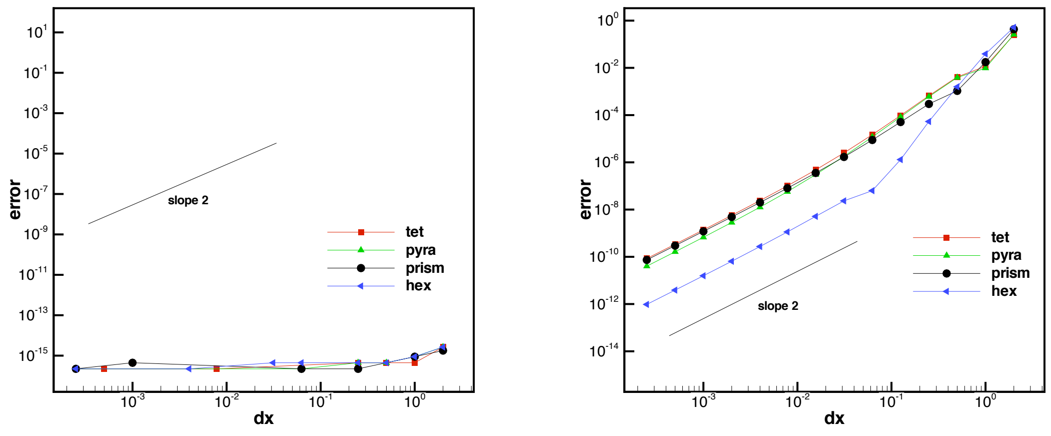

Figure 6. Verification of interpolation error for a linear function (left), where interpolations are exact, and for a non-linear function $f = sin(x)sin(y)sin(z)$ (right), where convergence is second-order for all element types. The significantly lower error coefficient associated with hex elements is an artifact of this particular analytic function, and not a general property of the hex interpolation.

acoustic noise generation on advanced helicopter rotor configurations. All of these examples use CHIMPS to exchange information between solvers.

A. Annular pipe test case

A simple test case that mimics the geometrical features of the jet-engine environment (compressible–incompressible coupling, rotational periodicity, swirling velocity) and which consists of three overlapping annular 90 degree pipe sectors has been used to validate CHIMPS. Two annular pipe sectors were computed with the compressible solver, SUmb (formerly known as TFLO; Yao *et al.* 2001; Kalitzin 2000) and the remaining one with the incompressible solver, CDP (Mahesh *et al.* 2004). Reynolds number for this flow was chosen as $Re = u_B D_h/\nu = 500$, with u_B, $D_h = (D_{inner} + D_{outer})/2$ and ν are bulk velocity, hydraulic diameter and kinematic viscosity, respectively. Mach number was chosen as $M = u_B/a = 0.3$. For the case with swirling velocity, its magnitude corresponded to 10% of the streamwise velocity component. The schematic of the computational domain for this coupled simulation is shown in Figure 8. The interface conditions used for this computation are depicted in Figure 9. Contrary to boundary conditions, the most ideal treatment of interface conditions would be to mimic the inter-block communications in the classical domain decomposition. In case of coupling between two compressible solvers, it means that all dependent variables at an interface should be exchanged at every inner iteration. However, the present test involves compressible–incompressible coupling and the following interface conditions had to be used at the beginning of every unsteady time step. SUmb1 provides CDP with the velocities u, v and w and SUmb2 provides CDP with the pressure (which is used to set the proper pressure level within CDP). CDP provides all the necessary variables for a subsonic inflow for SUmb2 (total pressure, p_t, total temperature, T_t, and the velocity angles, α_i). Note, that the density in CDP is set to a constant value.

Figures 10–11 show contours of streamwise velocity and the swirl velocity vectors at both SUmb1/CDP and CDP/SUmb2 interfaces. SUmb and CDP solutions agree very well in the overlap region.

B. Dynamic stall around a 2D airfoil

In order to validate the performance of CHIMPS in a more general moving-mesh situation, it has been applied to a coupled two-dimensional RANS simulation of dynamic stall using SUmb and CDP. The Reynolds and Mach numbers chosen are $\mathrm{Re} = u_\infty c/\nu_\infty = 10^6$ and $\mathrm{M} = u_\infty/a_\infty = 0.3$, where u_∞, ν_∞ and a_∞ are velocity, kinematic viscosity and speed of sound at the free stream, and c is the chord length. A NACA0012 airfoil

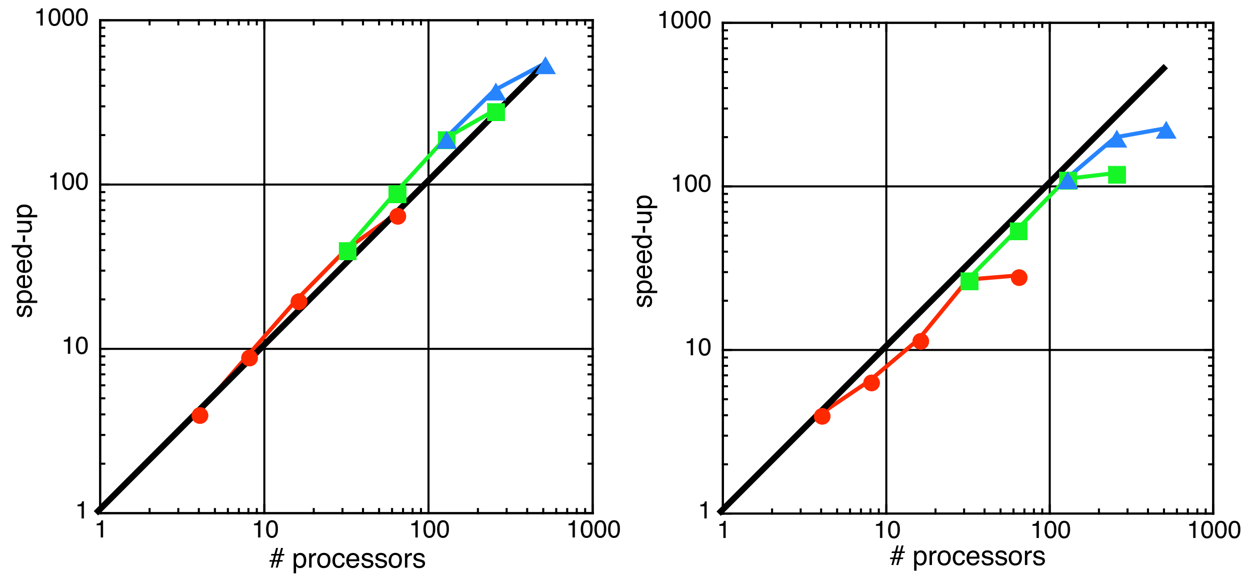

Figure 7. CHIMPS scalability for initial `updateInterface` call (left) and subsequent `updateInterface` calls (right) with $N = 16.8\cdot10^6$ and $M = 4.2\cdot10^6$ (red circles), $N = 134.2\cdot10^6$ and $M = 33.6\cdot10^6$ (green squares), and $N = 536.9\cdot10^6$ and $M = 134.2\cdot10^6$ (blue triangles).

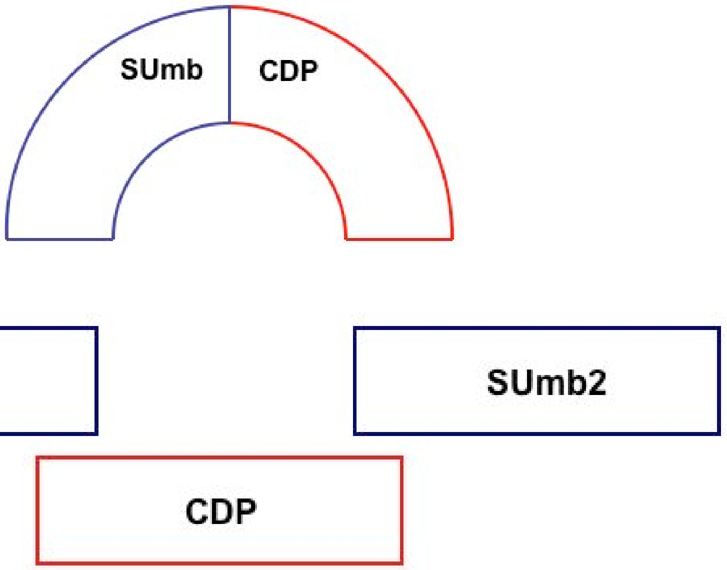

Figure 8. Schematic of the computational domain for a coupled simulation of three pipes, side view (left) and top view(right).

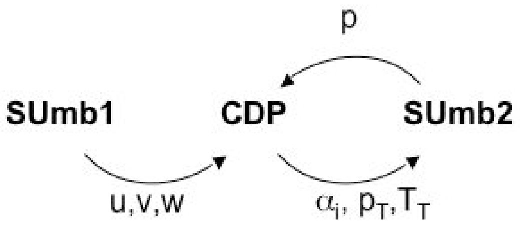

Figure 9. Interface conditions for a coupled simulation of three pipes.

is pitching about the quarter chord and the pitching condition is $\alpha(t) = \alpha_0 + \Delta\alpha \sin(\omega t)$, $\alpha_0 = 15.2°$, $\Delta\alpha = 4.2°$, $\beta = \omega c/2u_\infty = 0.5$, where $\alpha(t)$ is the angle of attack, ω is the pitching frequency, and β is the reduced frequency.

Figure 12 shows a schematic diagram of the computational domain. The inner near-blade region is solved by the fully compressible SUmb, while the outer domain is covered by the incompressible CDP. The inner SUmb domain has an elliptic shape with $-1.1 \lessapprox x/c \lessapprox 1.6$ and $-1.2 \lessapprox y/c \lessapprox 1.2$ (with the origin at the quarter chord), while the outer CDP domain has a square shape with a rectangular inner hole. Coordinates for the CDP inner hole and outer boundaries are $x/c = -0.5$, 1.25, $y/c = \pm 0.5$ and $x/c = y/c = \pm 15$, respectively. Note that there is an overlap region between the two domains, where the two solutions should be close to each other within the accuracy order of discretization and coupling schemes. The SUmb domain is pitching with the airfoil, whereas the CDP domain remains stationary in time. Therefore, the geometric information should be exchanged at every time step for this problem. For both solvers, v^2–f turbulence model (Durbin 1991, 1995) is used for turbulence closure. Both solvers are initialized with the uniform free stream at $t = 0$ and the size of computational time step is $\Delta t u_\infty/c = \pi/600$, which corresponds to 1200 time steps per pitching period and leads to CFL $= |u|\Delta t/\Delta x = 1 \sim 1.5$.

Along with the coupled simulation, we have additionally conducted two single-SUmb simulations with a large and a small computational domain, which will be denoted by LD and SD respectively, for the purpose of comparison. The LD case has a large circular computational domain ($-15 \lessapprox x/c, y/c \lessapprox 15$) with the usual far-field boundary condition applied at the outer boundary, while the grid resolution is maintained almost identical to that of the coupled simulation. On the other hand, the SD case uses exactly the same SUmb grids that are used for the coupled case (see figures 14–16). The only difference between SD and coupled cases is that the former imposes the far-field boundary condition on the outer boundary, while the latter uses the data transferred from CDP. Since both the domain size and grid resolution for the LD case

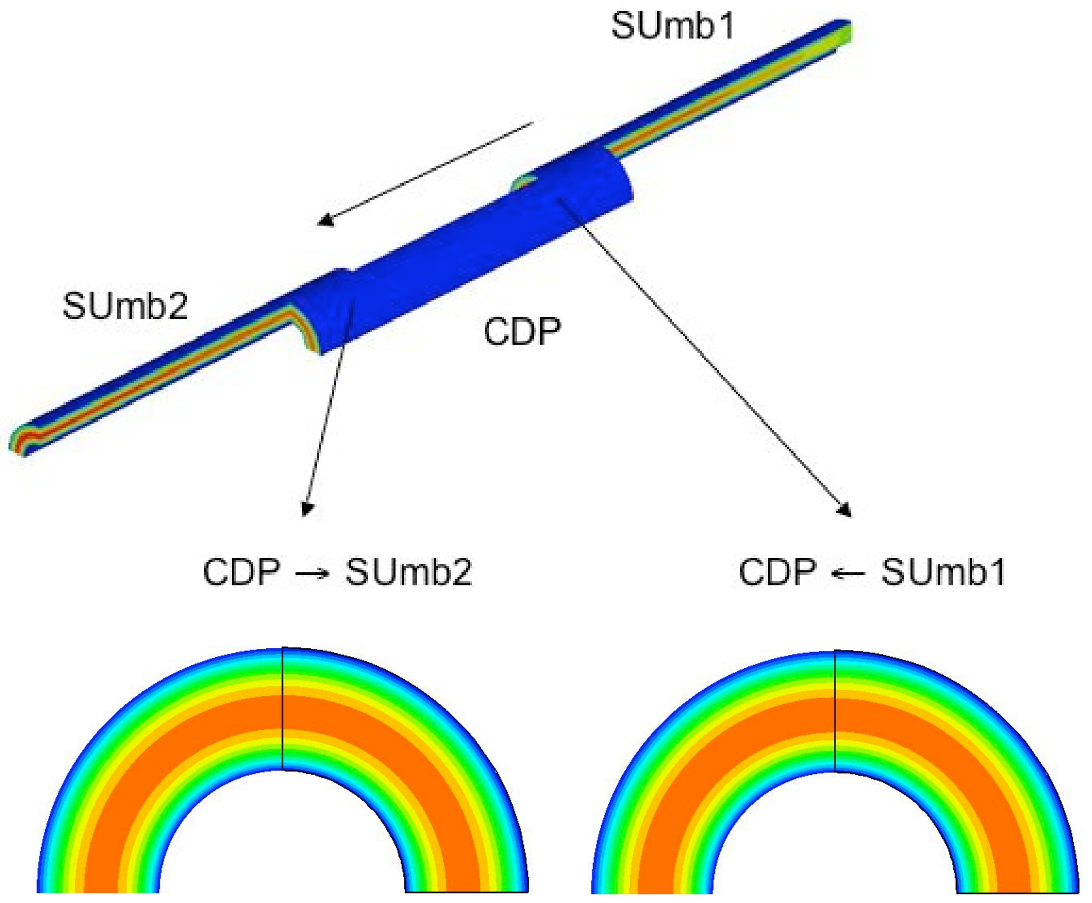

Figure 10. Contours of streamwise velocity at SUmb1/CDP (right) and CDP/SUmb2 (left) interfaces.

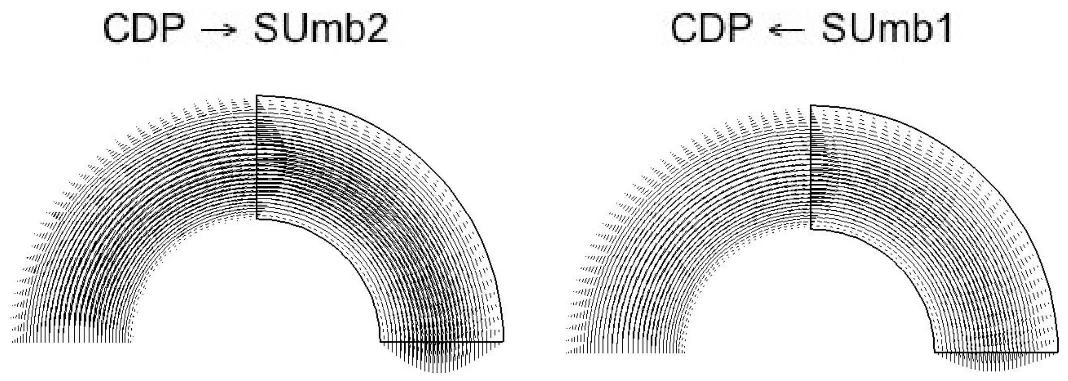

Figure 11. Contours of swirl velocity at SUmb1/CDP (right) and CDP/SUmb2 (left) interfaces.

are comparable to those for the coupled simulation, it is again expected that the two results should be close to each other within the accuracy order of discretization and coupling schemes used, as will be shown below.

As described in the previous section, the compressible–incompressible coupling for the present test has two main limitations. One stems from the fact that the incompressible solver cannot provide the thermodynamic pressure (or total energy, equivalently) required for the compressible one. The other is that SUmb and

CDP use different classes of inner-iteration schemes (multigrids combined with the dual time stepping for SUmb, whereas Poisson system combined with the fractional-step method for CDP), which makes it difficult to exchange data at every inner iteration. Therefore, we introduced the following approximations to the present coupled simulation:

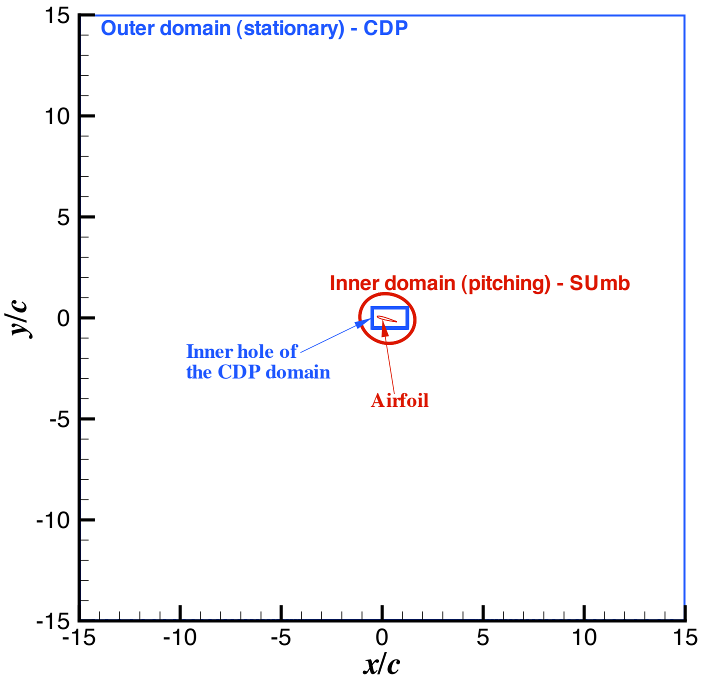
Outer domain (stationary) - CDP
Inner domain (pitching) - SUmb
Inner hole of the CDP domain
Airfoil
y/c
x/c
15
10
5
0
-5
-10
-15

Figure 12. Schematic of the computational domain for coupled simulation of dynamic stall.

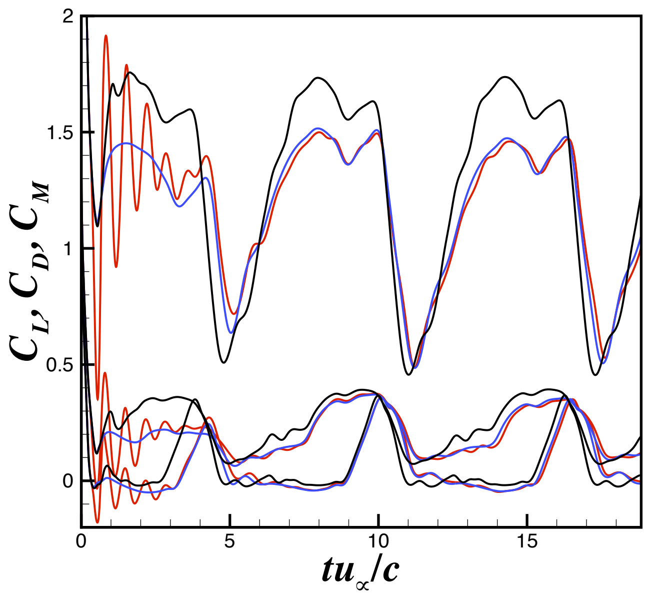

Figure 13. Time histories of lift, drag and moment coefficients. Red, blue, and black lines denote coupled, LD, and SD cases, respectively.

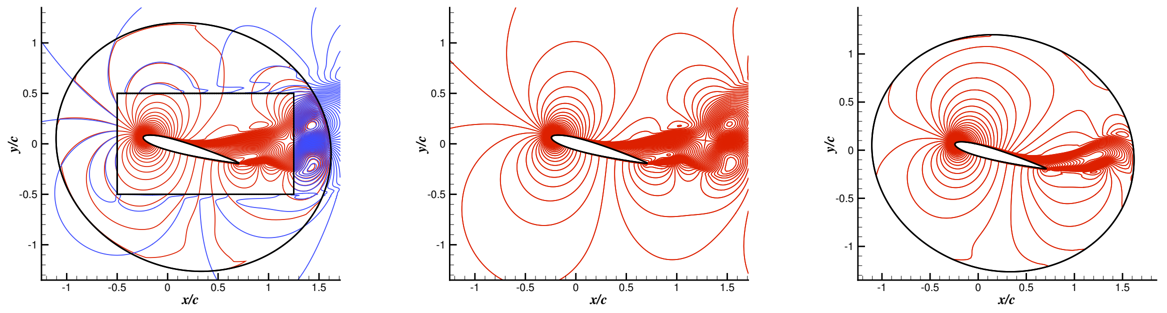

Figure 14. Contours of streamwise velocity at $t = 3T$ for coupled (left), LD (middle) and SD (right) cases. Red and blue lines are for SUmb and CDP contours, respectively. Denoted by black lines are the interfaces or domain boundaries.

(1) At interfaces, data are exchanged at the beginning of every unsteady time step and they are fixed during the inner iteration process of each solver.
(2) SUmb provides all the necessary dependent variables for CDP (i.e. u, v, w, k, ϵ, v^2 and f) via CHIMPS. Except the thermodynamic pressure, CDP also provides all the necessary dependent variables for SUmb (i.e. ρ, u, v, w, k, ϵ, v^2 and f) through CHIMPS. Meanwhile, SUmb extrapolates the thermodynamic pressure using one-dimensional Riemann invariant, instead of acquiring it from CHIMPS.

Note that CDP provides a constant free-stream density for the SUmb interface, which can be justified by the relatively low Mach number of the present case. Indeed, results of the LD case showed that the density variation is mostly within 1% of the free-stream value along the SUmb interface. Even the passage of shed vortices causes a density variation only less than 6% behind the trailing edge. The approximation (1) leads to a formally first-order accuracy in time. In the present test, this degradation in temporal accuracy is complemented by using a sufficiently small time step. Loss of temporal accuracy can be also avoided by introducing carefully designed subiterations that are performed between each pair of consecutive time stations, such as shown in Felippa *et al.* (2001).

Figure 13 shows time histories of lift, drag and moment coefficients for coupled, LD and SD cases. Contrary to single-SUmb cases, the coupled simulation initially shows spurious oscillations, which indicates that a certain amount of time should be passed for the two solutions in the overlap region to be fully adjusted to each other. The spurious oscillations result from artificial reflections occurring at the SUmb interface, since the interface condition (2) acts like an over-specified far-field boundary condition until the CDP solution is wholly modified from the initial uniform flow and fully adapted to the unsteadiness. As is expected, however, all the force and moment coefficients from LD and coupled cases show an excellent agreement with each other with a sufficient lapse of time ($t \gtrsim T/2$, where T is the pitching period). Note that the oscillatory initial transients can be evaded by using a face-to-face matching interface or a body-force-based volume coupling, which are currently under investigation. On the other hand, the SD case yields quite an erroneous airload prediction due to the insufficient domain size, which confirms that the present coupling plays a significant role in maintaining overall accuracy.

Figures 14–16 show contours of streamwise velocity (u), turbulent kinetic energy (k) and dissipation (ϵ) at $t = 3T$. As is expected, the coupled simulation shows mean-flow and turbulence evolutions very similar to those of the LD case. Furthermore, for the coupled simulation, it is also confirmed that SUmb and CDP solutions agree very well in the overlap region. However, the SD case exhibits a much thinner wake and a significantly incorrect flow development, which is consistent with the poor airload prediction shown in figure 13.

C. Full engine simulations in the DoE ASC program

Today's use of Computational Fluid Dynamics (CFD) in gas turbine design is usually limited to component simulations. The demand on the models to represent the large variety of physical phenomena encountered in the flow path of a gas turbine, mandates the use of a specialized and optimized approach for each component. The flow field in the turbomachinery portions of the domain is characterized by both high Reynolds-numbers

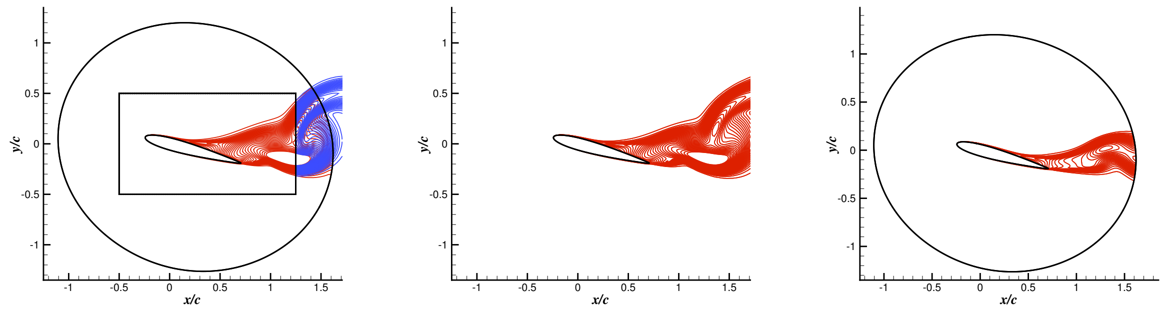

Figure 15. Contours of turbulent kinetic energy at $t = 3T$ for coupled (left), LD (middle) and SD (right) cases. Red and blue lines are for SUmb and CDP contours, respectively. Denoted by black lines are the interfaces or domain boundaries.

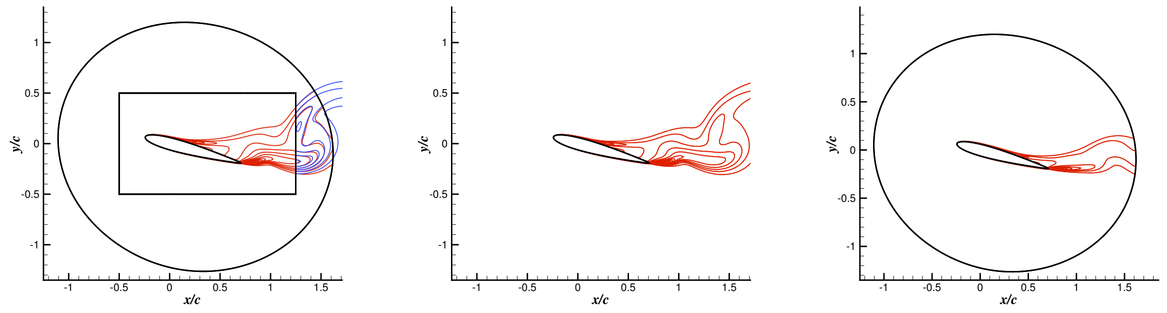

Figure 16. Contours of turbulent dissipation rate at $t = 3T$ for coupled (left), LD (middle) and SD (right) cases. Red and blue lines are for SUmb and CDP contours, respectively. Denoted by black lines are the interfaces or domain boundaries.

and high Mach-numbers. The accurate prediction of the flow requires the precise description of the turbulent boundary layers around the rotor and stator blades, including tip gaps and leakage flows. The flow solvers are typically based on the Reynolds-Averaged Navier-Stokes (RANS) approach. Here, the unsteady flow field is ensemble-averaged removing all dependence on the details of the small scale turbulence. A turbulence model becomes necessary to represent the portion of the physical stresses that has been removed during the averaging process. The flow in the combustor, on the other hand, is characterized by detached flows, chemical reactions and heat release. The prediction of detached flows and free turbulence is greatly improved using flow solvers based on Large-Eddy Simulations (LES). While the use of LES increases the computational cost, LES has been the only predictive tool able to simulate consistently these complex flows. LES resolves the large scale turbulent motions in time and space, and only the influence of the smallest scales, which are usually more universal and hence, easier to represent, has to be modeled. Since the energy containing part of the turbulent scales is resolved, a more accurate description of scalar mixing is achieved, leading to improved predictions of the combustion process.

Here, we present the results of an integrated multi-component simulation of a Pratt & Whitney aircraft engine presented in Figure 17. This simulation simultaneously computes the flow in the fan/compressor (SUmb), the combustor (CDP) and the turbine (SUmb), and each of the components exchanges flow data with its neighbors. The goal of this simulation is to demonstrate the ability to perform complex multi-physics multi-code simulations on a real world problem. The domain consists of a 20 degree sector of all the components; in view of the full engine simulation, this is the smallest sector that can be chosen, since it contains one fuel injector. The initial solution for the integrated simulation is provided by a combination of the component simulations. The operating conditions for the engine corresponds to cruise conditions. For the compressor (or the fan when the entire engine is computed), the boundary conditions have to be specified at the inlet. Here, total temperature, total pressure and the flow directions are imposed. At the

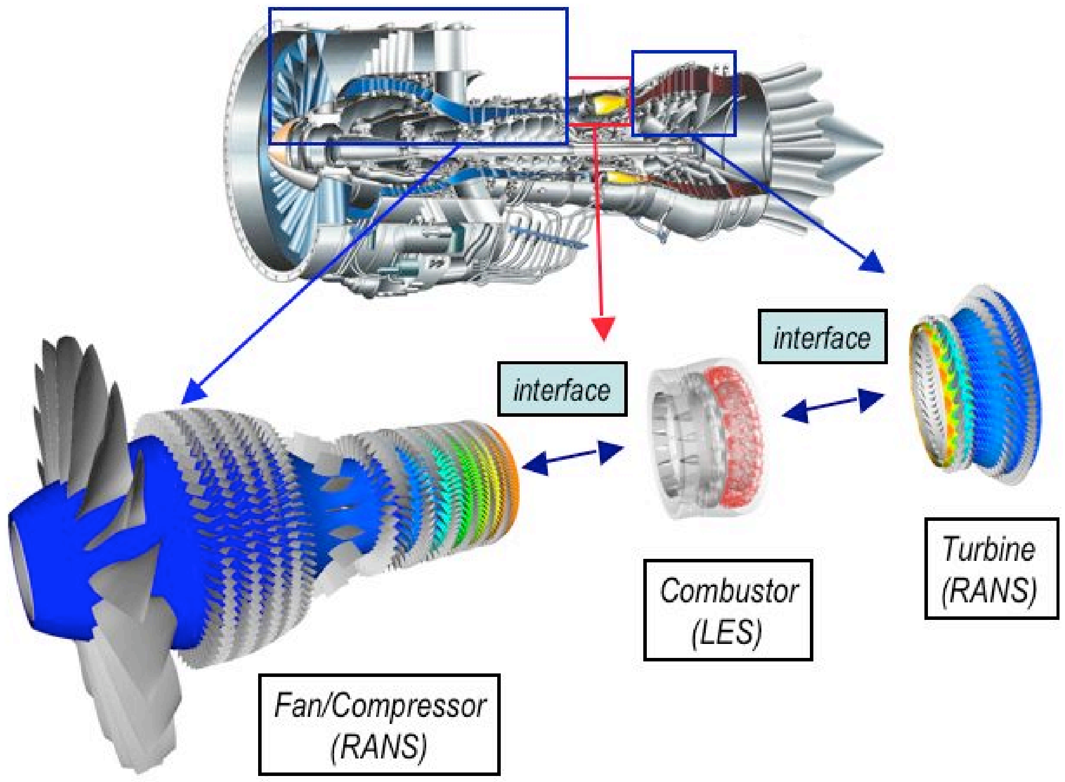

Figure 17. Decomposition of engine components for the coupled simulation.

outlet of the compressor, the static pressure is imposed as an interface condition (which can also be are provided by the downstream flow solver that is computing the combustor). The combustor receives at the inlet the mean flow velocities $[\bar{u}, \bar{v}, \bar{w}]$ from the compressor. Additional fluctuations $[u', v', w']$ generated in an auxiliary annular duct computation are then added to these mean velocities from SUmb. The fuel mass flow rate is defined as a boundary condition corresponds to the cruise operating conditions. The actual outlet of the combustor domain is far downstream in order to minimize the effect of the domain boundary and the convective outflow condition. The turbine receives at its inlet the total pressure, the total temperature and the flow directions from the combustor; the quantities that are transfered are time-averaged on the fly as the computation proceeds. At the outlet of the turbine, we specify the static pressure as a boundary condition.

The communication between the components is handled by CHIMPS. Since the turbomachinery meshes of each sector may not necessarily coincide with the sector mesh of the neighboring domain, the interface donor cells must be searched over the entire circumference of the engine. A fast search method has been developed to minimize the time spent on the sector searches. Vector components of exchanged flow variables are automatically rotated depending of the azimuthal offset of the neighboring domains.

We have performed two sets of integrated simulations for the 20^o sector. The first one concentrated on the high-pressure spool and included the high pressure compressor, the combustor and the turbine. In the second computation, we have also included the fan and the low pressure compressor (the remaining stages of the low pressure turbine and the exit nozzle are currently being added to the computational domain). In both computations, we considered two sets of grids for the compressor: a finer grid consisting of approximately 57 million cells for the entire fan/compressor ensemble and a coarser grid consisting of approximately 8 million cells. The combustor grid contains 3 million cells and the turbine grid consists of approximately 5 million cells. The time-step has been chosen to enure that in the turbomachinery components we use at least 50 time-steps for a blade passing in a blade row with the highest count and the highest rotational speed. This translates into about 6,300 time-steps needed for a full wheel revolution of the slower low pressure components and 2,700 time steps for the faster rotating high pressure components. In addition, estimates for the number of time-steps needed for a flow-through time range from 10,000 time steps for the high-pressure spool, to about 20,000 for the entire engine.

We have performed multiple simulations on a Xeon Linux cluster at the US Department for Energy. The simulations typically run for 2000 time-steps in 24 hours of wall clock-time on 700 processors, for the entire engine on the coarser grid for the compressor; the fan/compressor was run on 380 processors, the combustor on 80 processors and the turbine on 240 processors. To obtain the same amount of time-steps for the entire engine on the finer grid, approximately 3,500 processors would be needed. For the high-pressure spool, the computations with the coarser grid require only 400 processors, whereas the computations that employ the finer grid for the HPC require about 1200 processors. We estimate that a flow-through time of an entire high-spool of the engine can be computed within 5 days of uninterrupted running.

Our preliminary analysis of the results obtained so far (6,000 time-steps for the high pressure spool and 2,000 time-steps for the entire engine) has focused on the solution in the vicinity of the component interfaces. The axial velocity contours at the compressor/combustor interface plotted in the mid-passage radial plane are shown in Figure 18. The wakes from the last row of vanes in the high pressure compressor are propagating through the interface into the diffuser. Figure 19 presents the axial velocity and temperature contours for the combustor/turbine interface plotted for the mid-passage radial plane. Note that the time-averages of the flow variables from the combustor computation are passed to the turbine (total pressure, total temperature and flow angles). Interestingly, at the turbine inflow we have observed a strong variation of temperature and axial velocity in the circumferential direction.

D. Helicopter rotor simulations in the DARPA Helicopter Quieting program

The objective of Phase I of the DARPA Helicopter Quieting program (HQP) is to develop advanced simulation tools to accurately predict the performance and acoustic signature of helicopter rotors in forward flight. Our approach to the solution of this problem has been to adopt a two-flow-solver strategy to predict the flow features (transonic flows, dynamic stall, vortical wake, etc.). These two flow solvers (SUmb and CDP) are coupled to each other and to the University of Maryland's UMAC and UMARC packages for the prediction of the radiated noise field (UMAC), and for both the aeroelastic blade deflections and trim state of the helicopter rotor (UMARC). CHIMPS plays a key role in these interfaces between solvers as it is responsible for the exchange of information between SUmb and CDP, as well as the information extraction for the acoustic propagation module in our environment.

Figure 20 below shows a schematic view of the simulation environment and all of its components (including Large Eddy Simulations that are not discussed here) with CHIMPS at its center acting as an information broker for the various modules. Note that in this program, we have chosen to drive all modules (and CHIMPS) through high-level scripts in the Python scripting language that has been mentioned earlier in this paper: CHIMPS can be linked into a single executable (and used as a library) or it can be used as a runtime module that is imported and called from a Python *driver* script. Figure 21 below shows more details of the mesh geometries that the SUmb and CDP solvers use. Holes are cut into the CDP background mesh (red) and all interpolations are carried out through CHIMPS.

VI. Conclusions & Future Work

In this paper we have presented a description of the CHIMPS module/library for coupling of multiple parallel solvers into a single integrated simulation. As it stands, CHIMPS provides a fully parallel and scalable search and interpolation capability that can be used in multiple applications. Moreover, CHIMPS includes a first version of a volume integration capability for conservative interpolation of certain quantities. CHIMPS can be accessed as a library from a high-level language (Fortran 90/95 or C/C++) or as a module in the Python scripting language. The intent of the CHIMPS team is to provide this basic functionality to the community as an Open Source project and to enhance the current capabilities to include conservative interpolation, support for overset mesh computations (including hole cutting), and support for fluid-structure interfaces. We welcome the use of CHIMPS by members of the community and we look forward to their contributions to a constantly-improving CHIMPS library/module.

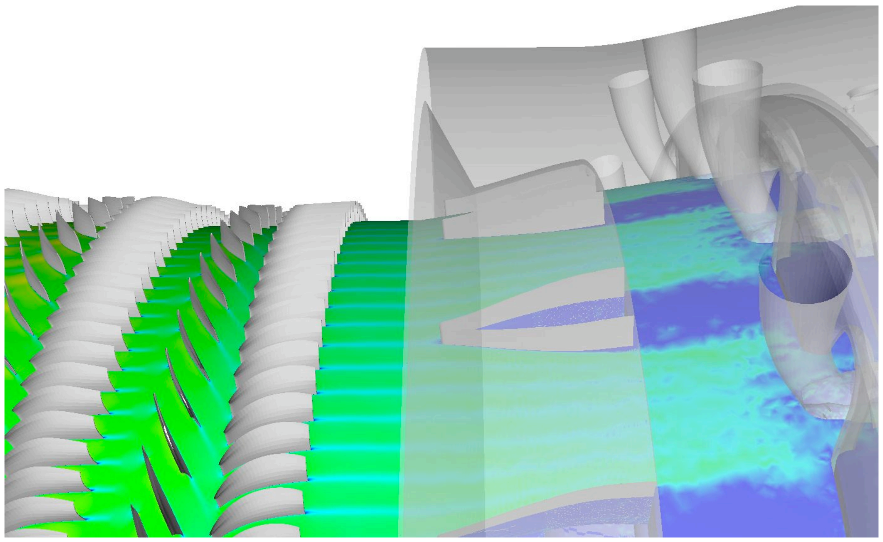

Figure 18. Compressor/combustor interface. Axial velocity, mid-passage.

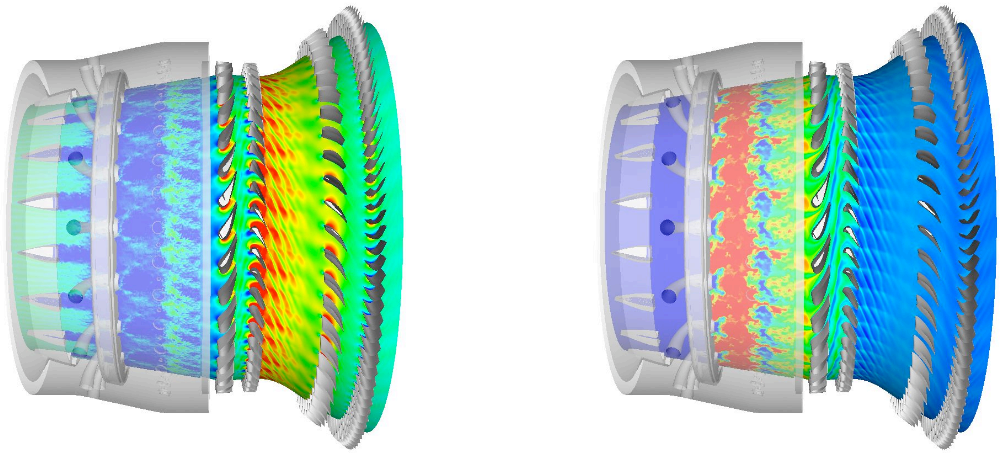

Figure 19. Combustor/turbine interface. Axial velocity (left) and temperature (right) contours, mid-passage.

VII. Acknowledgments

The majority of the work in the development of the CHIMPS module has been supported by the U.S. Department of Energy ASC program under contract number LLNL B341491 as part of the Advanced Simulation and Computing Program (ASC). Significant contributions and motivation for the development of the most recent version of CHIMPS have been provided by DARPA under the Helicopter Quieting program. This part of the work is sponsored by DARPA under ARO Contract # W911NF-04-1-0424. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government. In addition, the authors would like to acknowledge the contributions of Dr. Jaynarayanan Sitaraman from the University of Maryland for his use of the CHIMPS environment and his help in debugging the initial versions.

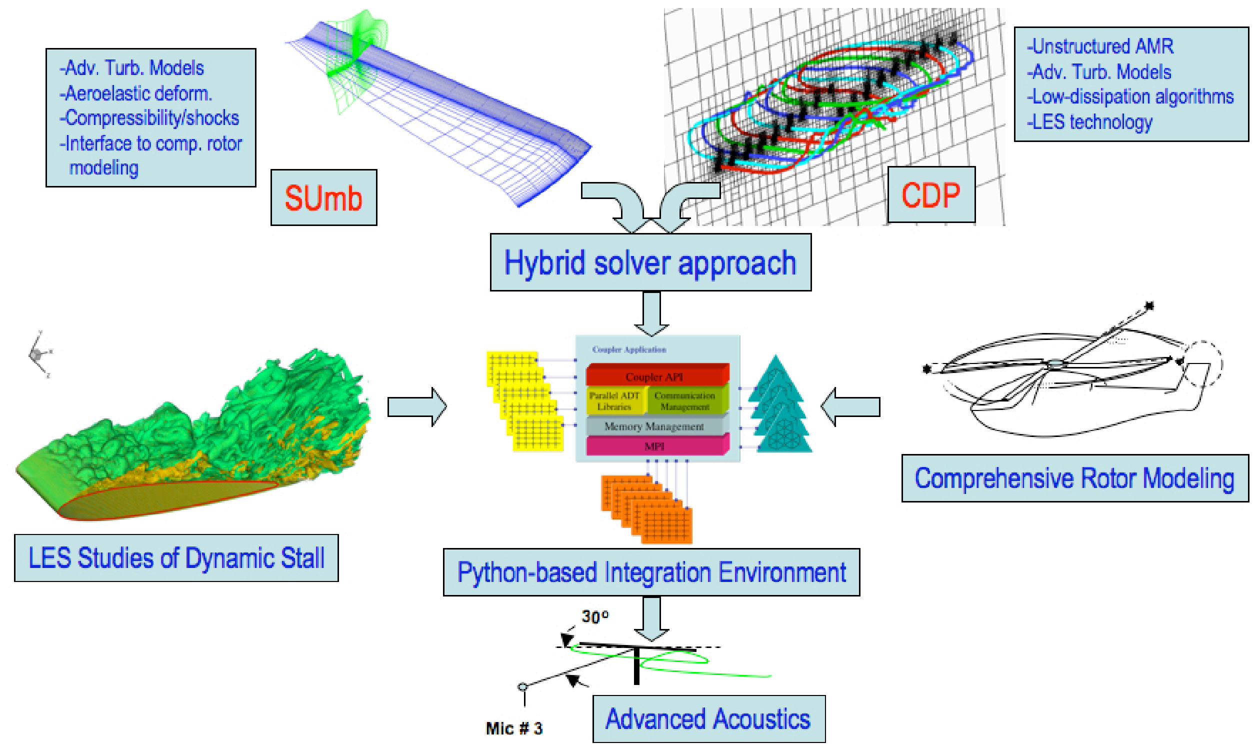

Figure 20. A schematic view of the HUSH (Hybrid Unsteady Simulations for Rotors) environment with the various participating modules and CHIMPS.

VIII. References

References

[1]Plimpton, S. J., Hendrickson, B., and Stewart, J. R., "A parallel rendezvous algorithm for interpolation between multiple grids," *J. Parallel Distrib. Comput.*, Vol. 64, 2004, pp. 266–276.

[2]Aftosmis, M., "Solution Adaptive Cartesian Grid Methids for Aerodynamic Flows with Complex Geometries," *VKI LS 1997-02, Computational Fluid Dynamics*, 1997.

[3]Zienkiewicz, O. and Taylor, R., *The Finite Element Method, Volume 1. Basic Formulation and Linear Problems*, McGraw-Hill Book Company Europe, 1997.

[4]Mattsson, K. and Nordström, J., "Summation by parts operators for finite difference approximations of second derivatives," *J. Comput. Phys.*, Vol. 199(2), 2004.

[5]Carpenter, M. H., Gottlieb, D., and Abarbanel, S., "Time-stable boundary conditions for finite-difference schemes so lving hyperbolic systems: Methodology and application to high-order compact s chemes," *J. Comput. Phys.*, Vol. 111(2), 1994.

[6]K.Mattsson, Svärd, M., and andJ. Nordström, M. C., "High-Order Accurate Computations for Unsteady Aerodynamics," *to appear in Computers and Fluids*, 2006.

IX. Appendices

Three separate appendices are included in this paper. These appendices show the detailed *driver* programs (written in Fortran 90/95) for different aspects of the current functionality of CHIMPS. These *driver* programs are meant to serve as examples to understand the level of programming that is needed to make use of the CHIMPS interface. In addition to these programs, the CHIMPS distribution contains a large number of test programs that can also be used to understand the functionality of CHIMPS.

The three *driver* programs that are presented are:

1. A scalability driver program that was used to produce the scalability graphs in Section B.
2. A driver program for the existing integration capabilities in CHIMPS.
3. A driver program and an explanation for the pitching airfoil test case.

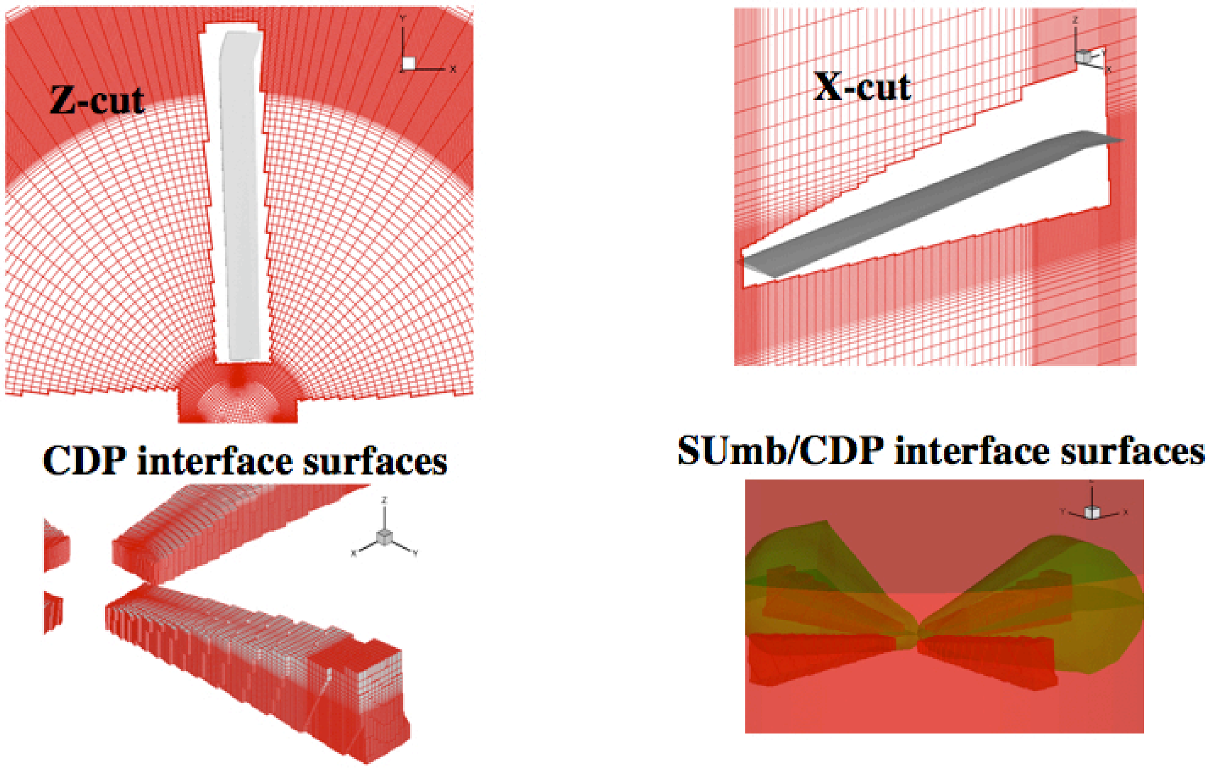

Figure 21. Mesh topologies used in the exchange of information between the SUmb and CDP solvers in the HUSH environment. Holes are cut into the CDP mesh (red). The blades flap and distort within the SUmb domain (green). CHIMPS determines all connectivity and passes information between solvers.

All codes are completely functional CHIMPS programs that can be compiled and run with the use of the CHIMPS library and MPI where appropriate.

A. Scalability driver program

```
program ScalabilityDriver
  use chimps_m
  use codeA_m
  use codeB_m
  implicit none

  include 'mpif.h'

  character(len=32) :: groupName, varNames(2), interfaceNames(1)
  integer :: ierr, myrank, nprocs, groupComm, myGroupRank, iter
  integer(IP) :: nNodes, nTet, nPyra, nPrism, nHex, nPoints, nVars, nInterfaces
  real(RP),    allocatable, dimension(:,:) :: xyz, data
  integer(IP), allocatable, dimension(:,:) :: tetConn, pyraConn, prismConn, hexaConn

  ! for timing
  real(RP), dimension(2) :: ts,dt,dtmean

  ! Initialize mpi...
  call mpi_init(ierr)
  call MPI_comm_rank(MPI_COMM_WORLD, myrank, ierr)
  call MPI_comm_size(MPI_COMM_WORLD, nprocs, ierr)

  ! timing
  dt = 0.0_RP

  ! initialize chimps on all processors using the same group name.
  groupName = 'CODEA_CODEB'
  call chimps_initialize(groupComm,groupName,MPI_COMM_WORLD)
  call chimps_setParam('HIGH','VERBOSITY')
  call codeA_init(groupComm)
  call codeB_init(groupComm)

  call MPI_comm_rank(groupComm, myGroupRank, ierr)

  ! code A mesh registration
  call codeA_getMeshSize(nNodes,nTet,nPyra,nPrism,nHex)
  allocate(xyz(3,nNodes),tetConn(4,nTet),pyraConn(5,nPyra))
  allocate(prismConn(6,nPrism),hexaConn(8,nHex))
  call codeA_getMeshGeom(xyz,hexaConn)
  call chimps_setMeshGeom(xyz,nNodes,tetConn,nTet,pyraConn, &
  nPyra,prismConn,nPrism,hexaConn,nHex,"CODEA_MESH")
  deallocate(xyz,tetConn,pyraConn,prismConn,hexaConn)

  ! code B point registration
  call codeB_getPointSize(nPoints)
  allocate(xyz(3,nPoints))
  call codeB_getPointGeom(xyz)
  call chimps_setPointGeom(xyz,nPoints,"CODEB_POINTS")

  ! register interpolation interface
  call chimps_setInterface("MY_INTERFACE",CHIMPS_INTERPOLATE, &
       "CODEA_MESH"  , "CODEA_CODEB", &  ! src
       "CODEB_POINTS", "CODEA_CODEB") ! dest

  ! register requested variable names
  varNames(1) = 'PHI'
  varNames(2) = 'PSI'
  nVars = 2
  call chimps_setPointRequest(varNames,nVars,'CODEB_POINTS')

  ! MAIN LOOP
  do iter = 1,3

     ! register solution on mesh for code A
     varNames(1) = 'PHI'
     varNames(2) = 'PSI'
     nVars = 2
     allocate(data(nVars,nNodes))
     call codeA_getMeshData(data)
     call chimps_setMeshData(data,varNames,nVars,'CODEA_MESH')
     deallocate(data)

     ! timing main work
     ts(1) = MPI_WTIME()

     ! perform interpolation
     interfaceNames(1) = "MY_INTERFACE"
     nInterfaces = 1
     call chimps_updateInterface(interfaceNames,nInterfaces)

     ! timing main work
     if (iter == 1) then
        dt(1) = MPI_WTIME() - ts(1)
     else
        dt(2) = dt(2) + MPI_WTIME() - ts(1)
     end if

     ! retrieve interpolated data
     varNames(1) = 'PHI'
     varNames(2) = 'PSI'
     nVars = 2
     allocate(data(nVars,nPoints))
     call chimps_getPointData(data,varNames,nVars,'CODEB_POINTS')
     call codeB_setPointData(data,varNames,nVars,'CODEB_POINTS')
     deallocate (data)

  end do

  ! output of timing information
  call MPI_REDUCE(dt(1),dtMean(1),2,MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,ierr)

  if (myrank == 0) then
     dtMean = dtMean/real(nprocs,RP)
     dtMean(2) = dtMean(2)/2.0_RP
     write (*,*) '-----------------------------------------------'
     write (*,*) ' Timing results:'
     write (*,'(a,3f12.5)') ' Chimps 1st     updateInterface: ',dtMean(1)
     write (*,'(a,3f12.5)') ' Chimps 2-n avg updateInterface: ',dtMean(2)
  end if

  call chimps_finalize()
  call codeA_finalize()
  call codeB_finalize()
  call mpi_finalize(ierr)

end program ScalabilityDriver
```

B. Integration interface driver

```
program IntegrationDriver
  use chimps_m
  use codeA_m
  use codeB_m
  implicit none

  include 'mpif.h'

  character(len=32) :: groupName, varNames(3), interfaceNames(1)
  integer :: ierr, myrank, nprocs, groupComm, myGroupRank, iter
  integer(IP) :: nNodes, nTet, nPyra, nPrism, nHex, nPoints, nVars, nInterfaces
  real(RP),    allocatable, dimension(:,:) :: xyz, data
  integer(IP), allocatable, dimension(:,:) :: tetConn, pyraConn, prismConn, hexaConn

  ! Initialize mpi...
  call mpi_init(ierr)
  call MPI_comm_rank(MPI_COMM_WORLD, myrank, ierr)
  call MPI_comm_size(MPI_COMM_WORLD, nprocs, ierr)

  ! initialize chimps on all processors using the same group name.
  groupName = 'CODEA_CODEB'
  call chimps_initialize(groupComm,groupName,MPI_COMM_WORLD)
  call chimps_setParam('HIGH','VERBOSITY')
  call codeA_init(groupComm)
  call codeB_init(groupComm)

  call MPI_comm_rank(groupComm, myGroupRank, ierr)

  ! code A mesh registration
  call codeA_getMeshSize(nNodes,nTet,nPyra,nPrism,nHex)
  allocate(xyz(3,nNodes),tetConn(4,nTet),pyraConn(5,nPyra),  &
  prismConn(6,nPrism),hexaConn(8,nHex))
  call codeA_getMeshGeom(xyz,hexaConn)
  call chimps_setMeshGeom(xyz,nNodes,tetConn,nTet,pyraConn, &
  nPyra,prismConn,nPrism,hexaConn,nHex,"CODEA_MESH")
  deallocate(xyz,tetConn,pyraConn,prismConn,hexaConn)

  ! code B mesh regustration as point entity
  call codeB_getPointSize(nPoints)
  allocate(xyz(3,nPoints))
  call codeB_getPointGeom(xyz)
  call chimps_setPointGeom(xyz,nPoints,"CODEB_POINTS")

  ! register interpolation interface
  call chimps_setInterface("MY_INTERFACE",CHIMPS_INTEGRATE, &
       "CODEA_MESH"  , "CODEA_CODEB", &  ! dest
       "CODEB_POINTS", "CODEA_CODEB") ! src

  ! register requested variable names
  varNames(1) = 'PHI'
  varNames(2) = 'PSI'
  nVars = 2
  call chimps_setPointRequest(varNames,nVars,'CODEA_MESH')

  ! MAIN LOOP
  do iter = 1,3

     ! register solution on points for code B
     varNames(1) = 'PHI'
     varNames(2) = 'PSI'
     varNames(3) = 'CELL_VOLUME'
     nVars = 3
     allocate(data(nVars,nNodes))
     call codeB_getPointData(data)
     call chimps_setPointData(data,varNames,nVars,'CODEB_POINTS')
     deallocate(data)

     ! timing main work
     ts(1) = MPI_WTIME()

     ! perform interpolation
     interfaceNames(1) = "MY_INTERFACE"
     nInterfaces = 1
     call chimps_updateInterface(interfaceNames,nInterfaces)

     ! retrieve integrated data
     varNames(1) = 'PHI'
     varNames(2) = 'PSI'
     nVars = 2
     allocate(data(nVars,nPoints))
     call chimps_getMeshData(data,varNames,nVars,'CODEA_MESH')
     call codeA_setPointData(data,varNames,nVars,'CODEA_MESH')
     deallocate (data)

  end do

  call chimps_finalize()
  call codeA_finalize()
  call codeB_finalize()
  call mpi_finalize(ierr)

end program IntegrationDriver
```

C. Pitching Airfoil Test Case

1. *Overall structure of a driver program*

As described in Section II, the final executable to run a multi-code integrated simulation is constructed by writing and compiling a driver program, which is the outermost layer that actually couples the entire set of programs. All the solver and CHIMPS modules are loaded and API routines are called in it.

Figure 22 shows the typical structure of a driver program which would be common to most applications. The sequence of operations in a driver is as follows:

1–2. As is common to all MPI applications, the first step is the MPI initialization, which is followed by the initialization of CHIMPS and each solver. Note that, along with the global communicator for driver and CHIMPS, a local communicator for each solver should be also defined here. In CHIMPS, it is actually handled by `chimps_initialize` which returns a local communicator as an output, and hence it is not necessary to call `mpi_comm_split` separately within the driver.

3. After initialization, the first thing to do is to transfer information on all the involved geometric objects (such as interface points or volume meshes) from each solver and register them to CHIMPS, which is done by calling `solver_getPoint(Mesh)Size`, `solver_getPoint(Mesh)Geom` and `chimps_setPoint(Mesh)Geom` in this order.

4. Once all the points and volume meshes are registered to CHIMPS, CHIMPS should construct interfaces, which are the relationships between named point and mesh objects, via `chimps_setInterface`.

5. Now the list of flow variables requested by each solver for its interfaces should be registered to CHIMPS for later interpolations, which is done by `chimps_setPointRequest`. Note that CHIMPS uses CGNS names to identify flow variables.

6. The list of all the relevant flow variables and actual solution data at the entire mesh points are transferred from each solver to CHIMPS by `solver_getMeshData` and `chimps_setMeshData`. Here, it is users' responsibility to guarantee that all the participating solvers provide proper flow variables to CHIMPS in order to prevent the interpolation failure.

7. Given all the transferred data and interfaces constructed, CHIMPS performs searches and interpolations via `chimps_updateInterface`.

8. With searches and interpolations completed, the interpolated data are transferred from CHIMPS back to each solver by `chimps_getPointData` and `solver_setPointData`. Within each solver, these data are employed as a boundary condition.

9. Given the interface data, each solver performs a time marching.

10. If the final time step is reached, each solver and CHIMPS are finalized. Otherwise, repeat 6–9

in case that all the relevant geometric objects are stationary. If the problem involves moving grids in time, repeat 3–9 to reflect changes in geometry at every time step.

Although CHIMPS is entirely written in Fortran90, a driver can be written in other programming languages in principle, more preferably in script languages like Python or Perl. Currently, CHIMPS supports both Fortran- and Python-based APIs. Note that f2py (http://cens.ioc.ee/projects/f2py2e/) provides an efficient environment which enables all the Fortran-based API routines accessible from Python without any necessity of rewrite.

2. Example of a driver program

As an example, we provide an actual driver program in this section, which was used for the coupled simulation of dynamic stall, so that users can utilize it as a template in writing their own ones. For this specific problem, the following names are prescribed to identify geometric objects and interfaces:

(1) Names internally defined within each solver
- These internal names should be used as an argument of solver API routines.
- `SUmbRhouvwInterfaceFamily`: Name for SUmb interface points used within SUmb.
- `SUmbAirfoil`: Name for SUmb volume meshes (coordinates and connectivity) used within SUmb.
- `cdp_interface`: Name for CDP interface points used within CDP.
- `default-interior`: Name for CDP volume meshes (coordinates and connectivity) named within CDP.

(2) Names newly created during registration to CHIMPS
- These names should be used as an argument of CHIMPS API routines.
- Users can also register the same names as in (1) to prevent the confusion.
- `SUMB_POINTS`: Name registered to CHIMPS for SUmb interface points.
- `CDP_POINTS`: Name registered to CHIMPS for CDP interface points.
- `SUMB_MESH`: Name registered to CHIMPS for SUmb volume meshes.
- `CDP_MESH`: Name registered to CHIMPS for CDP volume meshes.
- `SUMB_INTERFACE`: Name registered to CHIMPS for the interface where CDP is the source and SUmb is the destination (i.e. outer boundary of the SUmb domain).
- `CDP_INTERFACE`: Name registered to CHIMPS for the interface where SUmb is the source and CDP is the destination (i.e. inner-hole boundary of the CDP domain).

Also note that the assignment of constant free-stream density is manipulated within the driver program, instead of directly transferring it from CDP via CHIMPS. For minute details, the code is annotated in red below.

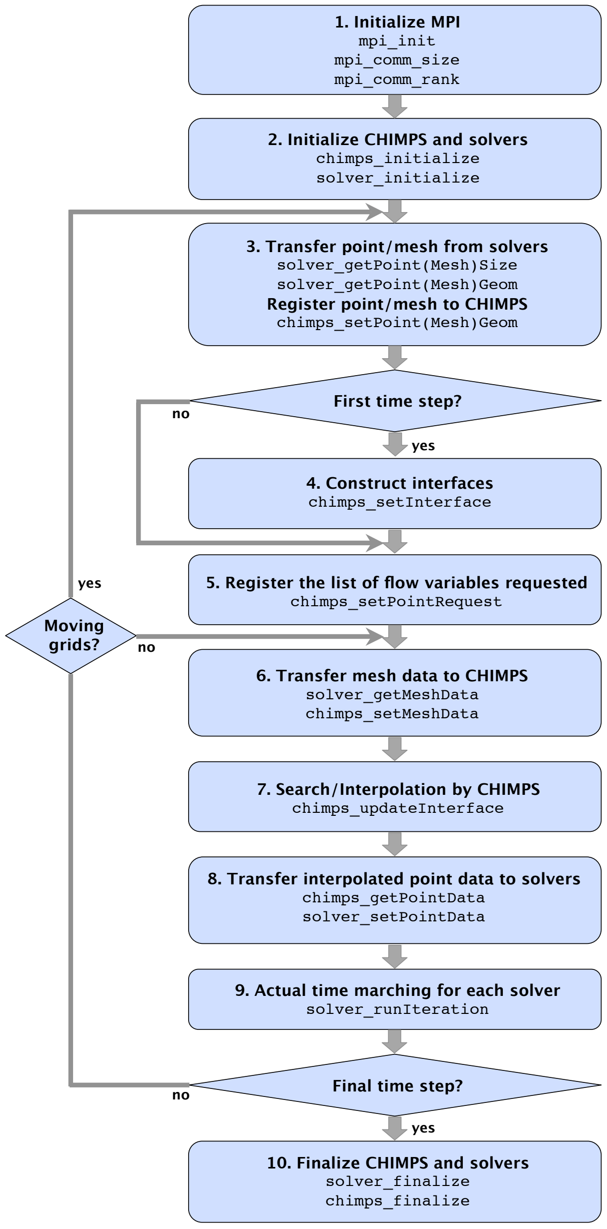
1. Initialize MPI
mpi_init
mpi_comm_size
mpi_comm_rank
2. Initialize CHIMPS and solvers
chimps_initialize
solver_initialize
3. Transfer point/mesh from solvers
solver_getPoint(Mesh)Size
solver_getPoint(Mesh)Geom
Register point/mesh to CHIMPS
chimps_setPoint(Mesh)Geom
First time step?
no
yes
4. Construct interfaces
chimps_setInterface
yes
5. Register the list of flow variables requested
chimps_setPointRequest
Moving grids?
no
6. Transfer mesh data to CHIMPS
solver_getMeshData
chimps_setMeshData
7. Search/Interpolation by CHIMPS
chimps_updateInterface
8. Transfer interpolated point data to solvers
chimps_getPointData
solver_setPointData
9. Actual time marching for each solver
solver_runIteration
Final time step?
no
yes
10. Finalize CHIMPS and solvers
solver_finalize
chimps_finalize

Figure 22. Typical structure of a driver program.

```
!
!  Driver to couple CDP and SUmb for flow over a pitching airfoil.
!  - SUmb covers the near-blade domain pitching with the airfoil.
!  - CDP covers the stationary outer domain.
!

  program coupled_pitching_af

  use mpi
  use cdp_if_coupler_m This module contains all the CDP API routines.
  use sumb_coupler_m   This module contains all the SUmb API routines.
  use chimps_m         This module contains all the CHIMPS API routines.

  implicit none

  ! ******* Beginning of driver parameters *******
  ! Variables needed for this driver are specified
  ! as parameters here...

  integer(IP), parameter :: nSteps            = 1000, &
                            Number of unsteady time steps for this session.
                            nProcsSUmb         =    8, &
                            Number of processors alloted to SUmb.  The rest is
for CDP.
                            nWriteStepsVolSUmb =    2, &
                            Writing interval for SUmb volume flow fields.
                            nWriteStepsSurSumb =    2
                            Writing interval for SUmb surface flow fields.
  logical, parameter :: movGridSUmb  = .true. , & Whether SUmb grids are moving
or not.
                        movGridCDP   = .false., & Whether CDP grids are moving
or not.
                        adjustDt     = .false.
                        Whether the driver readjusts the unsteady time step of
each solver.

  ! ********** End of driver parameters **********

  integer(IP) :: ierr, nProcsDrv, myIDDrv, groupComm,           &
                 nPoints, nNodes, nTetra, nPyra, nPrism, nHexa, &
                 ii, n, nVars, nInterfaces, nIterSUmb, nIterCDP
  integer(IP), allocatable :: tetraConn(:,:), pyraConn(:,:), &
                              prismConn(:,:), hexaConn(:,:)
  real(RP) :: tmpDrv1, tmpDrv2, rhoConstDrv, dtSUmb, dtCDP
  real(RP), allocatable :: xyz(:,:), data(:,:)
  character(len=80) :: groupName
  character(len=80), allocatable :: varNames(:), interfaceNames(:)
  character(len=4) :: ntstStr
  character(len=2) :: myIDStr

  ! ********** Beginning of the program **********

  ! Initialize mpi...

  call mpi_init(ierr)
  call mpi_comm_size(mpi_comm_world, nProcsDrv, ierr)
  call mpi_comm_rank(mpi_comm_world, myIDDrv, ierr)

  ! Define groupName...

  if(myIDDrv <  nProcsSUmb) groupName = 'SUMB'
  if(myIDDrv >= nProcsSUmb) groupName = 'CDP'

  ! Initialize CHIMPS...

  call chimps_initialize(groupComm, groupName, mpi_comm_world)
  chimps_initialize returns groupComm, which is a local communicator for each
solver.

  ! Initialize participating solvers...

  if(groupName == 'SUMB') then
    call sumb_initialize('param.in', groupComm)
  else if(groupName == 'CDP') then
    call cdp_initialize('cdp_if.in', groupComm)
  endif


  ! Get some parameters from SUmb for later use...

  tmpDrv1 = 0.

  if(groupName == 'SUMB') then
    call sumb_getParam(tmpDrv1, 'density for initialization')
    density for initialization is the SUmb keyword for free-stream density.
    Its value is obtained via sumb_getParam.
  endif

  call mpi_allreduce(tmpDrv1, rhoConstDrv, 1, mpi_real8, mpi_max, &
                     mpi_comm_world, ierr)
  The obtained value is shared by all processors.

  ! Set stationary point- and mesh-stuffs...

  if(groupName == 'SUMB') then

    ! Setup the point stuffs where data will be requested by SUmb...

    call sumb_getPointSize(nPoints, 'SUmbRhouvwInterfaceFamily')

    allocate(xyz(3,nPoints))

    call sumb_getPointGeom(xyz, 'SUmbRhouvwInterfaceFamily')

    call chimps_setPointGeom(xyz, nPoints, 'SUMB_POINTS')

    deallocate(xyz)

    ! Setup the SUmb mesh stuffs...

    call sumb_getMeshSize(nNodes, nTetra, nPyra, nPrism, nHexa, &
                          'SUmbAirfoil')
```

```
    allocate(xyz(3,nNodes), tetraConn(4,nTetra), pyraConn(5,nPyra), &
             prismConn(6,nPrism), hexaConn(8,nHexa))

    call sumb_getMeshGeom(xyz, tetraConn, pyraConn, prismConn, &
                          hexaConn, 'SUmbAirfoil')

    call chimps_setMeshGeom(xyz, nNodes, tetraConn, nTetra,     &
                            pyraConn, nPyra, prismConn, nPrism, &
                            hexaConn, nHexa, 'SUMB_MESH')

    deallocate(xyz, tetraConn, pyraConn, prismConn, hexaConn)

  else if(groupName == 'CDP') then

    ! Setup the point stuffs where data will be requested by CDP...

    call cdp_getPointSize(nPoints, 'cdp_interface')

    allocate(xyz(3,nPoints))

    call cdp_getPointGeom(xyz, 'cdp_interface')

    call chimps_setPointGeom(xyz, nPoints, 'CDP_POINTS')

    deallocate(xyz)

    ! Setup the CDP mesh stuffs...

    call cdp_getMeshSize(nNodes, nTetra, nPyra, nPrism, nHexa, &
                         'default-interior')

    allocate(xyz(3,nNodes), tetraConn(4,nTetra), pyraConn(5,nPyra), &
             prismConn(6,nPrism), hexaConn(8,nHexa))

    call cdp_getMeshGeom(xyz, tetraConn, pyraConn, prismConn, &
                         hexaConn, 'default-interior')

    call chimps_setMeshGeom(xyz, nNodes, tetraConn, nTetra,     &
                            pyraConn, nPyra, prismConn, nPrism, &
                            hexaConn, nHexa, 'CDP_MESH')

    deallocate(xyz, tetraConn, pyraConn, prismConn, hexaConn)
  endif

  ! Setup the interfaces which are relationships between named point-
  ! and/or mesh-stuffs...
  ! This should be called by all participating processers...

  call chimps_setInterface('SUMB_INTERFACE',            &
                           CHIMPS_INTERPOLATE_FAILSAFE, &
                           'CDP_MESH', 'CDP',           &
                           'SUMB_POINTS', 'SUMB')


  call chimps_setInterface('CDP_INTERFACE',             &
                           CHIMPS_INTERPOLATE_FAILSAFE, &
                           'SUMB_MESH', 'SUMB',         &
                           'CDP_POINTS', 'CDP')


  ! Main iteration loop.

  iter_loop: do n=1,nSteps

    ! Solvers determine the flow variables they need
    ! or should provide from their local domains...

    if(groupName == 'SUMB') then

      ! Data SUmb requests...

      nVars = 7

      allocate(varNames(nVars))

      CGNS names are used to identify flow variables.
      varNames(1) = 'VelocityX'
      varNames(2) = 'VelocityY'
      varNames(3) = 'VelocityZ'
      varNames(4) = 'TurbulentEnergyKinetic'
      varNames(5) = 'TurbulentDissipation'
      varNames(6) = 'TurbulentScalarV2'
      varNames(7) = 'TurbulentScalarF'

      call chimps_setPointRequest(varNames, nVars, 'SUMB_POINTS')

      deallocate(varNames)

      ! Data SUmb should provide...

      nVars = 7

      allocate(varNames(nVars))

      varNames(1) = 'VelocityX'
      varNames(2) = 'VelocityY'
      varNames(3) = 'VelocityZ'
      varNames(4) = 'TurbulentEnergyKinetic'
      varNames(5) = 'TurbulentDissipation'
      varNames(6) = 'TurbulentScalarV2'
      varNames(7) = 'TurbulentScalarF'

      allocate(data(nVars,nNodes))

      call sumb_getMeshData(data, varNames, nVars, 'SUmbAirfoil')

      call chimps_setMeshData(data, varNames, nVars, 'SUMB_MESH')

      deallocate(varNames, data)

    else if(groupName == 'CDP') then

      ! Data CDP requests...
  nVars = 7

  allocate(varNames(nVars))

  varNames(1) = 'VelocityX'
  varNames(2) = 'VelocityY'
  varNames(3) = 'VelocityZ'
  varNames(4) = 'TurbulentEnergyKinetic'
  varNames(5) = 'TurbulentDissipation'
  varNames(6) = 'TurbulentScalarV2'
  varNames(7) = 'TurbulentScalarF'

  call chimps_setPointRequest(varNames, nVars, 'CDP_POINTS')

  deallocate(varNames)

  ! Data CDP should provide...

  nVars = 7

  allocate(varNames(nVars))

  varNames(1) = 'VelocityX'
  varNames(2) = 'VelocityY'
  varNames(3) = 'VelocityZ'
  varNames(4) = 'TurbulentEnergyKinetic'
  varNames(5) = 'TurbulentDissipation'
  varNames(6) = 'TurbulentScalarV2'
  varNames(7) = 'TurbulentScalarF'

  allocate(data(nVars,nNodes))

  call cdp_getMeshData(data, varNames, nVars, 'default-interior')

  call chimps_setMeshData(data, varNames, nVars, 'CDP_MESH')

  deallocate(varNames, data)

endif


! Do the actual exchange...

nInterfaces = 2

allocate(interfaceNames(nInterfaces))

interfaceNames(1) = 'SUMB_INTERFACE'
interfaceNames(2) = 'CDP_INTERFACE'

call chimps_updateInterface(interfaceNames, nInterfaces)


deallocate(interfaceNames)

! Get the data from CHIMPS and provide them to the appropriate solver...

if(groupName == 'SUMB') then

  nVars = 7

  ! A little bit of trick to provide the constant density to SUmb...

  allocate(varNames(nVars+1))
```
varNames(8) is reserved for density.
```
  varNames(1) = 'VelocityX'
  varNames(2) = 'VelocityY'
  varNames(3) = 'VelocityZ'
  varNames(4) = 'TurbulentEnergyKinetic'
  varNames(5) = 'TurbulentDissipation'
  varNames(6) = 'TurbulentScalarV2'
  varNames(7) = 'TurbulentScalarF'

  allocate(data(nVars+1,nPoints))
```
data(8,:) is reserved for density.
```
  call chimps_getPointData(data, varNames, nVars, 'SUMB_POINTS')
```
A constant density is directly specified here.
```
  varNames(nVars+1) = 'Density'
  data(nVars+1,:) = rhoConstDrv

  call sumb_setPointData(data, varNames, nVars+1, 'SUmbRhouvwInterfaceFamily')

  deallocate(varNames, data)

else if(groupName == 'CDP') then

  nVars = 7

  allocate(varNames(nVars))

  varNames(1) = 'VelocityX'
  varNames(2) = 'VelocityY'
  varNames(3) = 'VelocityZ'
  varNames(4) = 'TurbulentEnergyKinetic'
  varNames(5) = 'TurbulentDissipation'
  varNames(6) = 'TurbulentScalarV2'
  varNames(7) = 'TurbulentScalarF'

  allocate(data(nVars,nPoints))

  call chimps_getPointData(data, varNames, nVars, 'CDP_POINTS')

  call cdp_setPointData(data, varNames, nVars, 'cdp_interface')
  deallocate(varNames, data)
endif

! Determine the unsteady time step...

if(adjustDt .and. n == 1) then
  tmpDrv1 = 0.
  tmpDrv2 = 0.
```
Time-step sizes for the two solvers are obtained via solver_getParam.
```
  if(groupName == 'SUMB') then
    call sumb_getParam(tmpDrv1, 'unsteady time step (in sec)')
  else if(groupName == 'CDP') then
    call cdp_getParam(tmpDrv2, 'DT')
  endif
```
Obtained values are shared by all processors.
```
  call mpi_allreduce(tmpDrv1, dtSUmb, 1, mpi_real8, mpi_max, &
                     mpi_comm_world, ierr)
  call mpi_allreduce(tmpDrv2, dtCDP , 1, mpi_real8, mpi_max, &
                     mpi_comm_world, ierr)
```
Readjust two Δt's so that one becomes an integer factor of the other.
```
  if(dtSUmb >= dtCDP) then
    nIterSUmb = 1
    nIterCDP  = int(dtSUmb/dtCDP)
    dtSUmb = dtCDP*real(nIterCDP)
  else
    nIterCDP  = 1
    nIterSUmb = int(dtCDP/dtSUmb)
    dtCDP = dtSUmb*real(nIterSUmb)
  endif
```
Reset Δt to the adjusted value via solver_setParam.
```
  if(groupName == 'SUMB') then
    call sumb_setParam(dtSUmb, 'unsteady time step (in sec)')
  else if(groupName == 'CDP') then
    call cdp_setParam(dtCDP, 'DT')
  endif

else if((.not. adjustDt) .and. n == 1) then
    nIterSUmb = 1
    nIterCDP  = 1
endif

! Run actual iterations...

if(groupName == 'SUMB') then
  call sumb_runIteration(nIterSUmb)
else if(groupName == 'CDP') then
  call cdp_runIteration(nIterCDP)
endif


if(groupName == 'SUMB') then
  if(mod(n,nWriteStepsVolSUmb) == 0) &
    call sumb_writeVolumeSolutionFile
  if(mod(n,nWriteStepsSurSUmb) == 0) &
    call sumb_writeSurfaceSolutionFile
endif

if(((.not. movGridSUmb) .and. (.not. movGridCDP)) &
   .or. (n == nSteps)) cycle iter_loop

if(movGridSUmb .and. groupName == 'SUMB') then

  ! Setup the point stuffs where data will be requested by SUmb...

  call sumb_getPointSize(nPoints, 'SUmbRhouvwInterfaceFamily')

  allocate(xyz(3,nPoints))

  call sumb_getPointGeom(xyz, 'SUmbRhouvwInterfaceFamily')

  call chimps_setPointGeom(xyz, nPoints, 'SUMB_POINTS')

  deallocate(xyz)

  ! Setup the SUmb mesh stuffs...

  call sumb_getMeshSize(nNodes, nTetra, nPyra, nPrism, nHexa, &
                        'SUmbAirfoil')

  allocate(xyz(3,nNodes), tetraConn(4,nTetra), pyraConn(5,nPyra), &
           prismConn(6,nPrism), hexaConn(8,nHexa))

  call sumb_getMeshGeom(xyz, tetraConn, pyraConn, prismConn, &
                        hexaConn, 'SUmbAirfoil')

  call chimps_setMeshGeom(xyz, nNodes, tetraConn, nTetra,     &
                          pyraConn, nPyra, prismConn, nPrism, &
                          hexaConn, nHexa, 'SUMB_MESH')

  deallocate(xyz, tetraConn, pyraConn, prismConn, hexaConn)

else if(movGridCDP .and. groupName == 'CDP') then

  ! Setup the point stuffs where data will be requested by CDP...

  call cdp_getPointSize(nPoints, 'cdp_interface')

  allocate(xyz(3,nPoints))

  call cdp_getPointGeom(xyz, 'cdp_interface')

  call chimps_setPointGeom(xyz, nPoints, 'CDP_POINTS')

  deallocate(xyz)

  ! Setup the CDP mesh stuffs...

  call cdp_getMeshSize(nNodes, nTetra, nPyra, nPrism, nHexa, &
                       'default-interior')

  allocate(xyz(3,nNodes), tetraConn(4,nTetra), pyraConn(5,nPyra), &
           prismConn(6,nPrism), hexaConn(8,nHexa))

  call cdp_getMeshGeom(xyz, tetraConn, pyraConn, prismConn, &
                       hexaConn, 'default-interior')
    call chimps_setMeshGeom(xyz, nNodes, tetraConn, nTetra,     &
                            pyraConn, nPyra, prismConn, nPrism, &
                            hexaConn, nHexa, 'CDP_MESH')

    deallocate(xyz, tetraConn, pyraConn, prismConn, hexaConn)

  endif

enddo iter_loop

! Finalize each solver...

if(groupName == 'SUMB') then
  call sumb_finalize()
else if(groupName == 'CDP') then
  call cdp_finalize()
endif


! Finalize CHIMPS...

call chimps_finalize()


! Terminate the MPI session...

call mpi_finalize(ierr)

! ********** End of the program **********

end program coupled_pitching_af
```