46th AIAA Aerospace Sciences Meeting and Exhibit
7 - 10 January 2008, Reno, Nevada

AIAA 2008-927

A Multi-Code Python-Based Infrastructure for Overset CFD with Adaptive Cartesian Grids

Andrew M. Wissink*
Ames Research Center, Moffett Field, CA 94035

Jayanarayanan Sitaraman†
Langley Research Center, Hampton, VA 23666

Venkateswaran Sankaran‡
Ames Research Center, Moffett Field, CA 94035

Dimitri J. Mavriplis§
University of Wyoming, Laramie, WY 82071

Thomas H. Pulliam¶
NASA Ames Research Center, Moffett Field, CA 94035

This paper describes a computational infrastructure that supports Chimera-based interfacing of different CFD solvers - a body-fitted unstructured grid solver with a block-structured adaptive cartesian grid solver - to perform time-dependent adaptive moving-body CFD calculations of external aerodynamics. The goal of this infrastructure is to facilitate the use of different solvers in different parts of the computational domain - body fitted unstructured to capture viscous near-wall effects, and cartesian adaptive mesh refinement to capture effects away from the wall. The computational infrastructure, written using Python, orchestrates execution of the different solvers and coordinates data exchanges between them, controlling the overall time integration scheme. Details about the infrastructure used to integrate the codes, the parallel implementation, and results from demonstration calculations are presented.

I. Introduction

An emerging paradigm in computational fluid dynamics (CFD) simulation is the use of computational infrastructures to orchestrate large-scale multi-disciplinary computations. Such infrastructures are typically composed of several independent codes or modules, coupled together to facilitate exchange of relevant data and to advance the combined solution in time. There are many reasons for such an approach, the most obvious being the need to integrate existing capabilities from multiple disciplines. Some examples relevant to aeronautics include coupled CFD and CSD (computational structural dynamics) simulations of aero-structure interactions, and coupled RANS (Reynolds-Averaged Navier-Stokes) and LES (Large Eddy Simulation) simulations. The algorithms and data structures used in each model are distinct and often do not lend themselves to one another. Hence, integrating them into a single large monolithic code is complex and usually relegates at least one of the models to be less accurate, less optimized, or less flexible than the original standalone solver. Coupling existing mature simulation codes through a common high-level infrastructure provides a natural way to reduce the complexity of the coupling task and to leverage the large amount of verification, validation, and user experience that typically goes into the development of each separate model.

Coupling multiple discipline codes also facilitates the use of different grid or meshing paradigms within a single computational platform. Traditional monolithic CFD codes are usually written for a single gridding paradigm, such as structured-cartesian, structured-body-fitted or unstructured (tetrahedral, hexahedral or prismatic) grids. Each meshing paradigm has specific advantages and disadvantages. For example, cartesian grids are easy to generate, to adapt, and to extend to higher-order spatial accuracy, but they are not

*Senior Research Scientist, ELORET Corp., MS 215-1, andrew.wissink@nasa.gov, AIAA Member
†Research Scientist - II, NIA, 100 Exploration Way, j.sitaraman@larc.nasa.gov, AIAA Member
‡Senior Scientist, UARC, UC Santa Cruz, MS 215-1, vsankaran@mail.arc.nasa.gov, AIAA Member
§Professor, Department of Mechanical Engineering, mavripl@uwyo.edu, AIAA Associate Fellow
¶Senior Research Scientist, MS T27B, tpulliam@mail.arc.nasa.gov, AIAA Associate Fellow

Copyright © 2008 by the American Institute of Aeronautics and Astronautics, Inc.
The U.S. Government has a royalty-free license to exercise all rights under the copyright claimed herein for Governmental purposes.
All other rights are reserved by the copyright owner.

suited for resolving boundary layers around complex geometries. Body-fitted structured grids work well for resolving boundary layers, but the grid generation process for complex geometries remains tedious and requires considerable user expertise. General unstructured grids are well-suited for complex geometries and are relatively easy to generate, but their spatial accuracy is often limited to second-order, and the associated data structures tend to be less computationally efficient than their structured-grid counterparts. Thus, while a single gridding paradigm brings certain advantages in some portions of the flowfield, it also imposes an undue burden on others. A computational platform that embodies multiple gridding paradigms provides the potential for optimizing the gridding strategy on a local basis for the particular problem at hand.

A notable example of such a multiple-grid paradigm is the OVERFLOW code,[4] which employs structured body-fitted grids in the near-body region and multi-level cartesian grids in the off-body region. The two grid zones are coupled together using Chimera overset grid procedures with associated domain connectivity modules for exchanging data between the grids. In this paper, we consider a similar multiple grid paradigm, involving *unstructured* near-body grids and block-structured adaptive cartesian off-body grids, together with a generalized domain connectivity approach for data exchange. The unstructured grids facilitate ease of grid generation around complex geometries, while the adaptive cartesian grids permit the use of high-order spatial schemes and time-dependent mesh refinement. Instead of developing a single unified code that contains these capabilities, we employ a set of existing codes and couple them through a Python infrastructure: (a) the parallel NSU3D code[15] for the unstructured near-body solver, (b) the SAMRAI framework[12] for the off-body cartesian grid generation, adaptation, and parallel communication, (c) the serial high-order ARC3DC code for solution on the cartesian blocks, and (d) a domain connectivity module which includes Chimera-style hole cutting capability together with the CHIMPS[32] software for communication between overlapping grid systems. In all cases, the individual codes have been previously validated for stand-alone operation. The present work focuses on their integration within a coupled infrastructure, while a companion article,[34] also presented at this meeting, focuses on the testing and evaluation of the overall multi-code platform discussed here.

The concept of a computational infrastructure to support data exchange between different codes is not new, and a number of such infrastructures exist. They can generally be classified into two categories; *high-level* execution managers that coordinate the execution of standalone legacy codes, and *low-level* frameworks that provide a common data format and communication protocol from which the higher-level executable may be built. Examples of the former include FD-CADRE[28] and MDICE,[14] which are used to manage execution of different codes for multidisciplinary overset moving-body problems. Each code is run in a standalone fashion, maintaining its own parallel implementation and exchange data through file I/O. Examples of the latter include SIERRA[6,37] and Overture,[9] which provide a meshing paradigm, solver support, and MPI-based parallel communication support. The solvers in this case are not executed as standalone modules; they must be rewritten to use the data structures and communication support provided by the framework. Both approaches have their advantages and disadvantages. The advantage of the high-level approach is that it requires little change to existing legacy codes, although the cost of the file-based data exchange typically limits the efficiency and parallel scalability of the coupled code execution. The low-level approach circumvents this problem by exchanging information within the framework itself, and additionally, it encourages the use of common algorithms and load balance strategies across multiple codes. However, it usually requires substantial rewriting of the existing code to translate it into a format that incorporates the data structures and communication protocols supported by the framework. It is also less flexible in the support of alternate gridding strategies which the framework may not have been designed to handle.

The infrastructure presented in this work can be considered to be *intermediate-level*. Like the high-level execution managers, it links existing simulation codes that do not have to be restructured. However, it exchanges data between the codes through Python-based software that is object-oriented and scalable. Data exchanges are through memory rather than file I/O. Recent successful efforts integrating separate codes for multi-disciplinary simulation have used a similar approach. Alonso et al.[1] have used a Python-based infrastructure for multi-disciplinary design optimization. Schluter et al.[31] have also demonstrated coupled LES-RANS simulations on large-scale parallel computer systems with a similar infrastructure. Further, Gopalan et al.[7] have used it for multi-disciplinary simulations consisting of CFD, CSD, and acoustic codes applied to helicopter aeromechanics and noise. The specific advantage of the Python-based approach is that it allows each code module to be treated as an object, providing a convenient way to assemble a complex multi-disciplinary simulation in an object-oriented fashion. Use of existing simulation codes facilitates software re-use and also reduces software maintenance, since it is possible to leverage all the debugging efforts that have

gone into the simulation code. The Python infrastructure orchestrates operations in the solvers by making calls through a socket-like interface layer, which also manages the translation of data between Python and the native language (e.g. C or Fortran) of the component solvers. Data exchanges can be done without memory copies or file I/O, and the infrastructure can be run in parallel on large multi-processor computer systems. As long as the Python interfaces are used only at the high-level, primarily for exchanging data and not for computing numerics, the overhead introduced by the Python layer is minimal.

In the present work, the cartesian AMR meshing framework (SAMRAI) is written in C++, while the near-body CFD code (NSU3D), off-body CFD single-block solver (ARC3DC) and the domain connectivity software (CHIMPS + hole cutting software) are written in Fortran90. The individual modules are wrapped in Python with standardized interfaces that optionally allow plugging in additional modules or swapping one or more modules with appropriate replacements. The overall control of the iterative or time-advancement process resides within the main Python script. The script orchestrates the time-stepping of each of the solvers and controls the frequency of the domain connectivity operations. The inherent flexibility of the scriptable Python code makes it straightforward to accommodate different time-stepping schemes in the solution process.

The paper is organized as follows. Section II describes the motivation and implementation details of the dual-mesh paradigm, i.e., unstructured near-body coupled to block-structured adaptive cartesian off-body meshes. Section III discusses the flow solvers used for the different mesh types. Section IV presents the interfaces developed for the solvers to facilitate time-stepping algorithms in the infrastructure. Section V presents details of the Python infrastructure and its parallel implementation. Finally, Section VI provides some sample results of calculations that have been performed using the infrastructure.[34]

II. Meshing Paradigm

The infrastructure is designed to link solvers to support an overset meshing approach that uses unstructured grids near the body surface — the "near-body" grids — and cartesian adaptive grids away from the surface — the "off-body" region (Fig. 1). This near/off-body overset meshing approach allows the boundary layer effects to be captured near the surface while resolving far-field effects through efficient adaptive cartesian methods. Such a paradigm has been demonstrated previously by Meakin[20–23] using curvilinear structure grids near the body. Henshaw[10] followed a similar approach for 2D calculations that used fully adaptive off-body grids. The infrastructure developed here further extends these seminal developments with two important extensions. First, we use an *unstructured* grid for the near-body region, coupled with a structured-cartesian off-body grid. Second, rather than couple the dual-grid system in the same code, we use two *separate* solvers to perform the CFD computation.

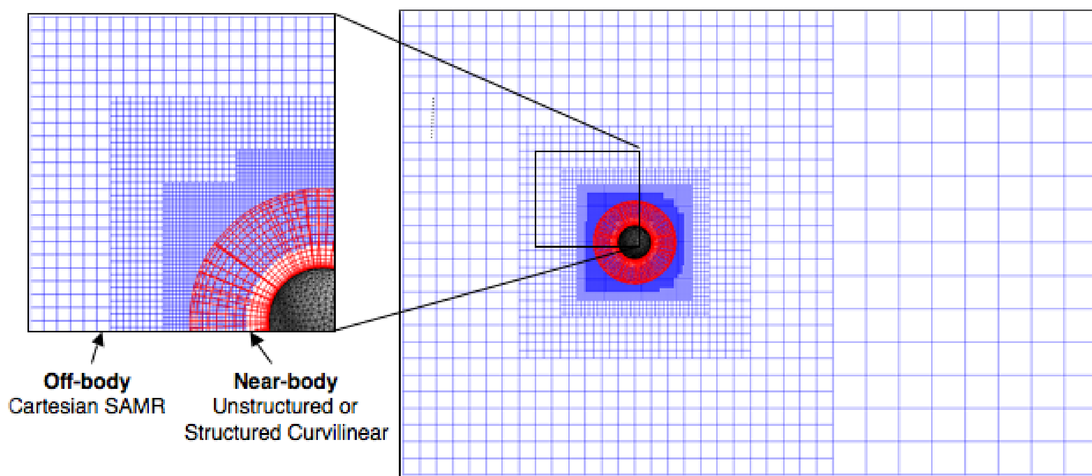


Figure 1. Overset near/off-body gridding. Unstructured or curvilinear grids to capture geometric features and boundary layer near body surface, adaptive block-structured cartesian grids to capture far-field flow features.

The starting point for the simulation is a near-body grid, generated external to the infrastructure, that

extends a short distance from the body. This grid can be a curvilinear grid or an unstructured tetrahedral or prismatic grid, extracted from a standard unstructured volume grid or generated directly from a surface triangulation using prisms with hyperbolic marching.[24] Either will work equally well in the infrastructure. The reason for using curvilinear or unstructured grids in the near-body region is to properly capture the geometry and viscous boundary layer effects, which are difficult to impossible to capture with cartesian grids alone. A short distance from the body, the solution is interpolated to a structured cartesian background mesh using standard overset hole-cutting and interpolation techniques.

Once the near-body grid is supplied to the infrastructure, all adaptive off-body gridding is performed automatically. The outer boundary points of the near-body grids will receive data from the background cartesian grids. These interpolation points are referred to as inter-grid boundary points (IGBP). They do not compute a solution, they simply receive data interpolated directly from the overlapping background grids. The locations (x, y, z coordinates) and resolution Δs (generally, the minimum distance to neighboring grid points) of the near-body IGBPs are used to guide construction of the off-body grid system.

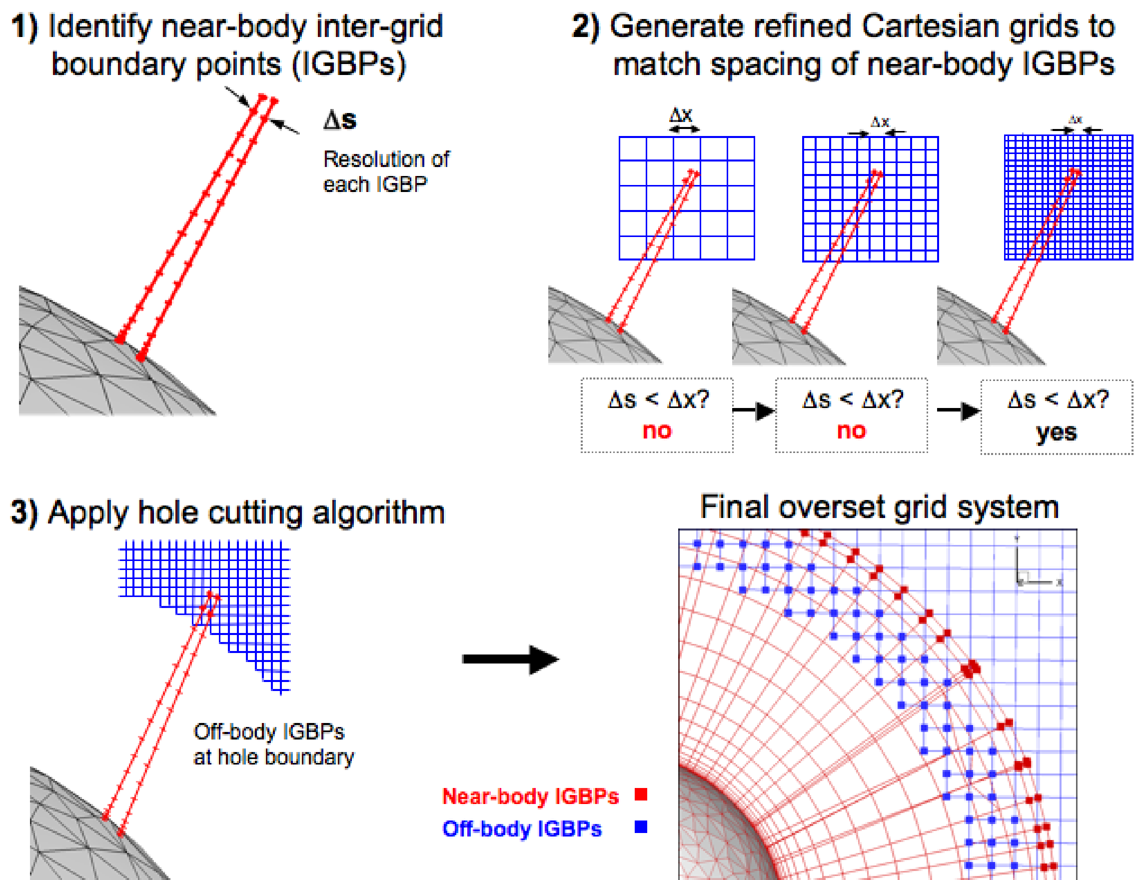


Figure 2. Adaptive cartesian off-body grid generation. The resolution of near-body IGBPs are used to drive refinement of cartesian grids. For the purposes of illustration, all figures use dual-fringes (i.e., two interpolation points). In practice, between one and three fringes are used, depending on the order of accuracy in the solver.

The off-body grid system uses a block-structured AMR (SAMR) grid. Levels are constructed from coarsest to finest. The coarsest level defines the physical extent of the computational domain. Each finer level is formed by selecting cells on the coarser level and then clustering the marked cells together to form the regions that will constitute the new finer level. Using the list of near-body IGBPs, identify all cartesian cells on a particular level that contain at least one IGBP. Then assess each of those cells to see whether the cartesian grid spacing Δx is greater than the resolution Δs of the near-body IGBP it contains. Those cartesian cells that do not have appropriate resolution (i.e., $\Delta x > \Delta s$) are marked for refinement and the next level is constructed. This process continues until no more cells are marked i.e., the resolution of all near-body IGBPs has been satisfied by the off-body cartesian grid system. At that point the overlap region is cut away from the cartesian grid and the off-body IGBPs are identified at the hole boundaries. Figure 2

outlines the off-body grid generation process.

III. Code Modules

This section discusses the different modules that are integrated within the infrastructure using the aforementioned meshing strategy. The body-fitted unstructured mesh solver NSU3D developed by Mavriplis et al.[15,17,19,41] is used to resolve near-wall effects — shocks, separation zones, boundary layers, etc. The block structured adaptive cartesian code SAMARC, which couples the parallel AMR library SAMRAI[11,12,40] with the high-order ARC3DC solver, is used in the off-body to resolve far-field wake effects. A domain connectivity module, whose purpose is to couple the near- and off-body solvers, manage Chimera grid hole cutting, and perform data interpolation and inter-processor communication between the solvers.

III.A. Near-Body Solver

The near-body solver, NSU3D, is an unstructured mesh multigrid Unsteady Reynolds-averaged Navier-Stokes (URANS) solver developed for high-Reynolds number external aerodynamic applications. The NSU3D discretization employs a second-order accurate vertex-based approach, where the unknown fluid and turbulence variables are stored at the vertices of the mesh, and fluxes are computed on faces delimiting dual control volumes, with each dual face being associated with a mesh edge. This discretization operates on hybrid mixed-element meshes, generally employing prismatic elements in highly stretched boundary layer regions, and tetrahedral elements in isotropic regions of the mesh. A single edge-based data structure is used to compute flux balances across all types of elements. The single-equation Spalart-Allmaras turbulence model,[36] as well as a standard $k - \omega$ two-equation turbulence model[38] are available within the NSU3D solver.

The NSU3D solution scheme was originally developed for optimizing convergence of steady-state problems. The basic approach relies on an explicit multistage scheme which is preconditioned by a local block-Jacobi preconditioner in regions of isotropic grid cells. In boundary layer regions, where the grid is highly stretched, a line preconditioner is employed to relieve the stiffness associated with the mesh anisotropy.[16] An agglomeration multigrid algorithm is used to further enhance convergence to steady-state.[15,17] The Jacobi and line preconditioners are used to drive the various levels of the multigrid sequence, resulting in a rapidly converging solution technique.

For time-dependent problems, first, second, and third-order implicit backwards difference time discretizations are implemented, and the line-implicit multigrid scheme is used to solve the non-linear problem arising at each implicit time step. NSU3D has been extensively validated in stand-alone mode, both for steady-state fixed-wing cases, as a regular participant in the AIAA Drag Prediction workshop series,[19] as well as for unsteady aerodynamic and aeroelastic problems,[42] and has been benchmarked on large parallel computer systems.[18]

For operation within an overset environment, "iblanking" capability has been added. The "iblanks" specify which nodes the solution variables are to be updated in the near-body solver (iblank = 1) and which nodes are not updated, i.e., fringes and holes (iblank = 0). The fringes correspond to the overset regions of the near-body grid, which are updated by the off-body code, while holes correspond to locations where the grid intersects solid objects.

For execution within the Python framework we modified NSU3D to be built as a dynamic shared library rather than a standalone executable. First, the subroutines and variables that need to be exposed to the framework are identified (see discussion in Sec. IV). The number of interface routines required is small because they only need to perform three main functions — initialize mesh and data, run an iteration or timestep, and save data and terminate. The amount of data exposed to the framework is also small — the grid (x, y, z) and solution data $(\rho,\rho u,\rho v,\rho w,e)$ plus turbulence model data for each node in the mesh. Second, we generate a Python signature file that "wraps" these interface routines and specifies functions to return the exposed data to the infrastructure. This signature file is used as an input to the `f2py`[27] tool which automatically builds the loadable Python module used by the infrastructure. The total time required to complete these two steps, by a skilled individual familiar with the Python wrapping process but not necessarily NSU3D, is about one week. The procedure is very similar to that used by Alonso et. al.[1] Further details are discussed in that reference.

III.B. Off-Body Solver

The adaptive cartesian off-body solver is built using the Structured AMR Applications Infrastructure (SAMRAI)[11,12,40] from Lawrence Livermore National Lab. SAMRAI uses a block-structured SAMR grid system that consists of a hierarchy of nested refinement levels with each level composed of a union of logically-rectangular grid regions (Fig. 3). All grid cells on a particular level have the same spacing, and the ratio of spacing between levels is generally a factor of two or four, although it is possible to use other refinement ratios as well. Grid generation, load balancing, and parallel adaptive data exchanges between blocks in the off-body solver are all performed by SAMRAI.

The flow solution on each SAMR block is performed by ARC3DC, a version of NASA's ARC3D[29,30] code with high-order algorithms optimized for cartesian grids. The code solves the Euler equations with a spatial central differencing scheme up to 6th-order accuracy and both 3rd-order and 5th-order accurate artificial dissipation terms. The code also has a number of high-order time integration schemes, although the one used exclusively so far in the adaptive context is explicit 3rd-order Runge-Kutta (RK).

Cartesian SAMR grids offer a number of computational advantages over traditional unstructured meshes. Grid storage costs are minimal because it is only necessary to store the boundaries of each grid block, which can be described by seven integers per block. In practice, a typical SAMR grid system can be stored in $O(10^3) - O(10^4)$ integers compared to $O(10^6) - O(10^9)$ reals for a comparable unstructured volume mesh. Use of the structured mesh allows simulation data to be stored in contiguous arrays with a defined order, without indirection, which minimizes memory overheads and leads to higher FLOP rates through efficient use of cache. Cartesian grids require no metric terms so the number of operations in the spatial difference operations are minimal. It is straightforward to implement high-order schemes on cartesian grids. Finally, cartesian grids permit a convenient multi-level representation that facilitates numerical algorithms like multi-grid. The multi-level representation also makes grid adaptivity relatively straightforward.

We are primarily interested in time-dependent problems where the grids dynamically adapt — refine and coarsen — to changing features in the flowfield. On a parallel computer system, each "adapt" step requires the grid be repartitioned for load balancing and data communication patterns re-established between processors. This is usually a complex time consuming task that is difficult to implement in a scalable manner with a fully unstructured mesh. As a result, while many unstructured solvers have used AMR successfully for steady-state problems (where the number of adapt cycles is small), few if any have been able to achieve scalable performance for problems where the grid is adapted frequently. The SAMR paradigm has a significant advantage because the block-structured grid description is very concise, so it is much faster to perform the re-gridding operations after each adapt step. Tests of time-dependent adaptive structured AMR calculations in SAMRAI in which the grid was adapted frequently (every other timestep) have shown scaling to over 1000 processors.[8,39]

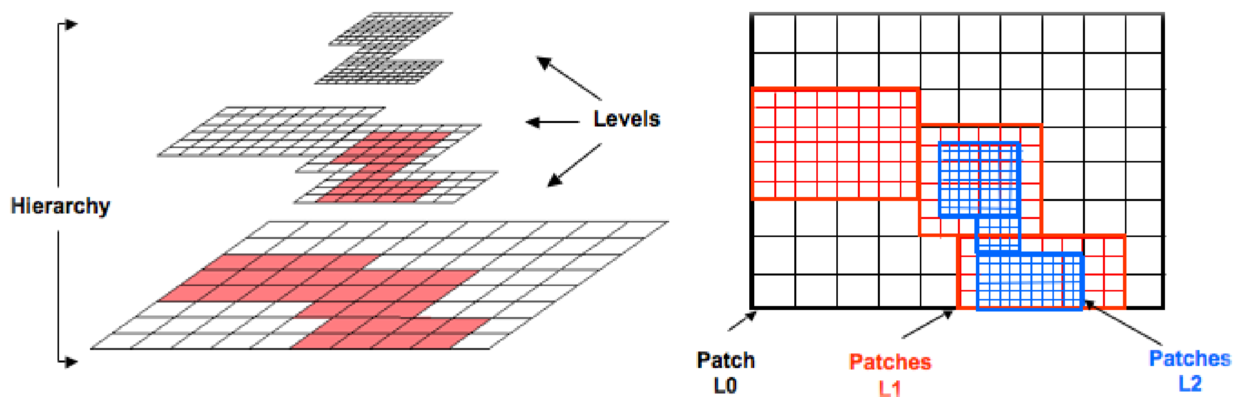


Figure 3. Block structured AMR grid composed of a hierarchy of nested levels of refinement. Each level contains uniformly-spaced logically-rectangular regions, defined over a global index space.

The initial adaptive off-body grid system is generated using the scheme discussed in Sec. II and shown in Fig. 2. Throughout the simulation, the grids are adapted to capture time-dependent features. Between adaptive gridding steps the governing equations are numerically integrated on the patches. All levels execute the explicit RK scheme with a uniform timestep. That is, the same timestep is applied across all grids at each

RK sub-step so the overall timestep is governed by the spacing on the finest level. We currently do not refine in time, although that is a consideration for future development. At the beginning of each RK sub-step, data on fine patch boundaries are first updated either through copying data from a neighboring patch on the same level, if one exists, or through interpolation of data from a coarser level. Second, the numerical operations to advance a single RK sub-step are performed simultaneously on each patch of each level. Third, data are injected into coarse levels wherever fine patches overlap coarse ones. The data exchanges performed to update patch boundary and interior information are done between processors using MPI communication. All parallel communication for these operations is managed by SAMRAI.

The off-body code that couples SAMRAI and ARC3DC is referred to as SAMARC. It is primarily written in C++ with some F77 routines for numerics. The mechanism to construct SAMARC as a shared object for the Python script is largely similar to the steps used for NSU3D. First, we identify the routines and data that need to be exposed to the framework. The routines include initializing the mesh and data, running a timestep, adapting the mesh, and shutting down at the end of the run. The data exposed is the grid, which can be defined in a concise integer-based block-structured format, and the solution data at each node of the mesh. From this information an interface definition is generated for the tool `swig`[3] to automatically build the files that make up the loadable Python module. There are minor differences in the wrapping procedure used by `swig` compared to that used by `f2py` for F90 codes, but the concept is similar. The final loadable module is then able to access the necessary routines and data. A further Python layer of object orientation is also built to further modularize the data exchange and execution.

III.C. Near/Off-Body Domain Connectivity

The Domain Connectivity Function (DCF) module has the role of coupling the near- and off-body solvers. It performs two main tasks; Chimera grid hole cutting to identify the points that exchange data, and the actual interpolation and inter-processor exchange of data. The latter step is done using the Coupler for High-performance Integrated Multi-Physics Simulations (CHIMPS)[32] package from Stanford University. This package has been used to couple distinct simulation codes in a number of applications, including LES/RANS solvers for turbulent flows,[31] and structured/unstructured solvers for helicopter aeromechanics calculations.[7]

Identification of the inter-grid boundary points (IGBPs) is done through Chimera-style hole cutting software (see section II for more discussion on this procedure). Once the near- and off-body IGBPs have been identified, CHIMPS performs the data interpolation between them. It manages any inter-processor exchanges that must take place to complete this operation. NSU3D and SAMARC use different scaling of their primitive variables and this is factored into the interpolation. CHIMPS is parallel and contains native Python interfaces so its integration into the infrastructure is straightforward.

IV. Interfaces

The Python infrastructure orchestrates execution of the different modules and thereby controls the global time-integration sequence. In this section, we discuss the interfaces developed for the flow solver and domain connectivity modules, in order to illustrate the role of the Python infrastructure in driving the simulation.

The interfaces created for the different CFD modules are designed to facilitate different levels of synchronization between the solvers. For instance, in the simplest case, synchronization occurs between NSU3D and SAMARC solvers after each completes a single iteration (i.e., performs a `runIteration()` step), followed by a call to the DCF module to exchange data at inter-grid boundary points. This sequence, used for steady-state problems, is illustrated in Fig. 4(a). Time dependent problems use the same steady integration procedure for the sub-iterations inside of a global time-stepping loop, as illustrated in Fig. 4(b). Many variations of this basic scheme can be constructed. For instance, the solvers may use different time-integration algorithms and be synchronized only after each physical timestep. Alternatively, data could be exchanged between solvers at the end of their respective sub-iteration steps, providing more frequent synchronization. Since this sequence is controlled by Python and is therefore scriptable, any combination can be easily constructed simply by modifying the order of the calls in the Python script.

The interfaces developed for the two flow solvers and the domain connectivity package are as follows:

NSU3D

`initialize()`	read input, read/generate initial grid, allocate memory

runIteration()　perform a single sub-iteration
runTimestep(dt)　perform a single timestep
runTimestep(dt,n)　perform a single timestep, using n sub-iterations
finalize()　de-allocate memory, clean up any open files

SAMARC

All the methods listed for NSU3D, plus:
adaptGrids(time)　regenerate off-body grid system

DCF

setScales(nsu3d,samarc)　sets the non-dimensional scaling parameters for each solver
checkNB(<ugrid>)　sets the unstructured grid used in NSU3D
checkOB(<sgrid>)　sets the SAMR grid used in SAMARC
setInterfaces()　computes IGBPs for all solvers
update()　interpolates IGBPs between the solvers
finalize()　de-allocate memory, clean up any open files

In the DCF module, the "check" functions let the module know the grid information. The "set interfaces" method determines the connectivity (IGBPs) between the two solvers, and the "update" function communicates the data. The check and set interfaces methods are only called at the initial time and after the grid configuration changes, such as when the near-body grid moves, or off-body grids are adapted. In between these steps when the grid configuration is static, only the update function is needed to exchange interface data between the solvers.

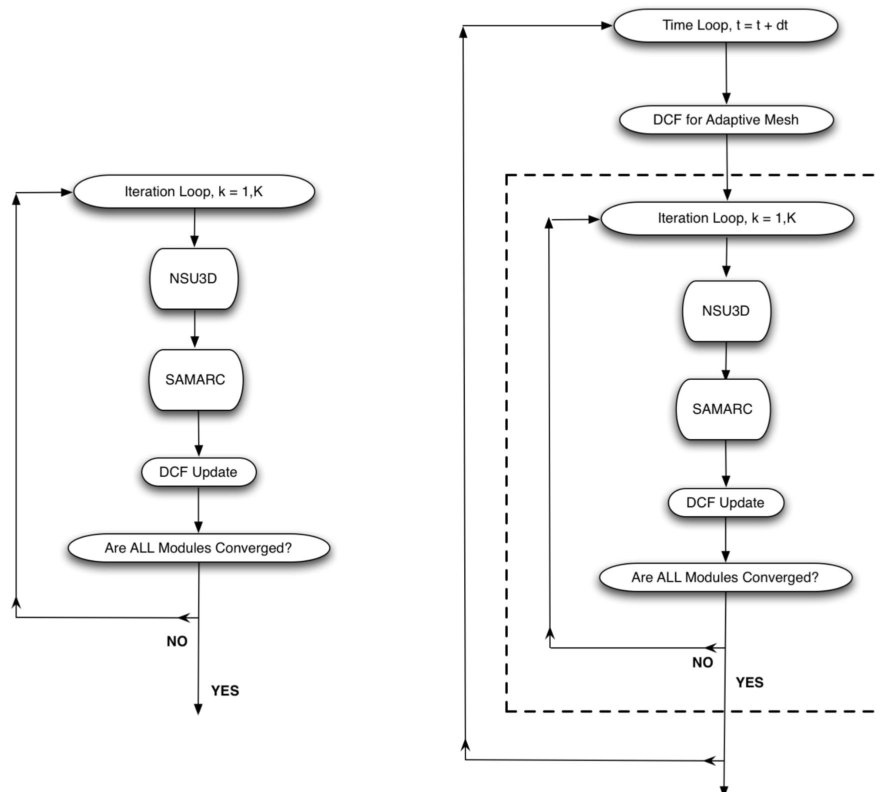


Figure 4. Time integration scheme. Integrating to steady state using sub-iterations (a), and using the same steady-state scheme within a time-stepping loop (b).

V. Infrastructure

The multi-code infrastructure, which we refer to as the software integration framework (SIF), consists of two parts: 1) a high-level Python script that orchestrates execution of different modules, and 2) a set of interfaces designed for each module that are called by the Python script. A discussion of the module interfaces is given in the previous section, and details about the way in which these interfaces are integrated with Python using the `f2py` and `swig` tools is given in Section III. In this section we focus on the high-level script and provide details of its parallel implementation.

Python supports object-orientation and the infrastructure uses this paradigm, treating each module as an object in the main Python script. Each module allocates the required memory for its grid and solution variables. Since the amount of data that must actually be exchanged between modules is small, most of the internal data structures for the modules are protected and are not accessible at the Python level. The interfaces are designed to expose only the variables of interest that need to be known to other modules.

An example of the Python script that executes a simple steady-state integration loop in the SIF is:

Python SIF script for Steady-State Integration

```
# Pull in modules
import nsu3d
import samarc
import dcf

# create NSU3D and SAMARC solvers & read input
nearbody = nsu3d('nsu3d.input')
offbody = samarc('samarc.input')

# access grid and solution information from the two solvers
[xNB,qNB,ilb,ilv] = nearbody.getData()
[gridOB,qOB,iblOB] = offbody.getData()

# set the non-dimensional scaling parameters for the two solvers
nbscaling = array(<nondimensional scaling of Q in NSU3D>)
obscaling = array(<nondimensional scaling of Q in SAMARC>)
dcf.setScales(nbscaling,obscaling)
dcf.checkNB(xNB[0],qNB[0],ilb[0])  # register unstructured grid with CHIMPS
dcf.checkOB(gridOB,qOB,iblOB,...)  # register cartesian grid data with CHIMPS
dcf.setInterfaces()                # compute near/off-body IGBPs

# Iteration to solution
nsteps = 10
for n in range(nsteps):
     nearbody.runIteration()
     offbody.runIteration()
     dcf.update()                  # exchange data at IGBPs

# Shut down
nearbody.finalize()
offbody.finalize()
dcf.finalize()
```

The preliminary steps create and initialize the modules. Grid and solution data are accessed directly from the respective solver modules without making memory copies. For NSU3D the grid data structures consist of a list of coordinates and an index list that expresses the connectivity of edges and cells. For SAMARC the grid data consists of bounding box coordinates, cell spacing and index bounds. The real x, y, z coordinates

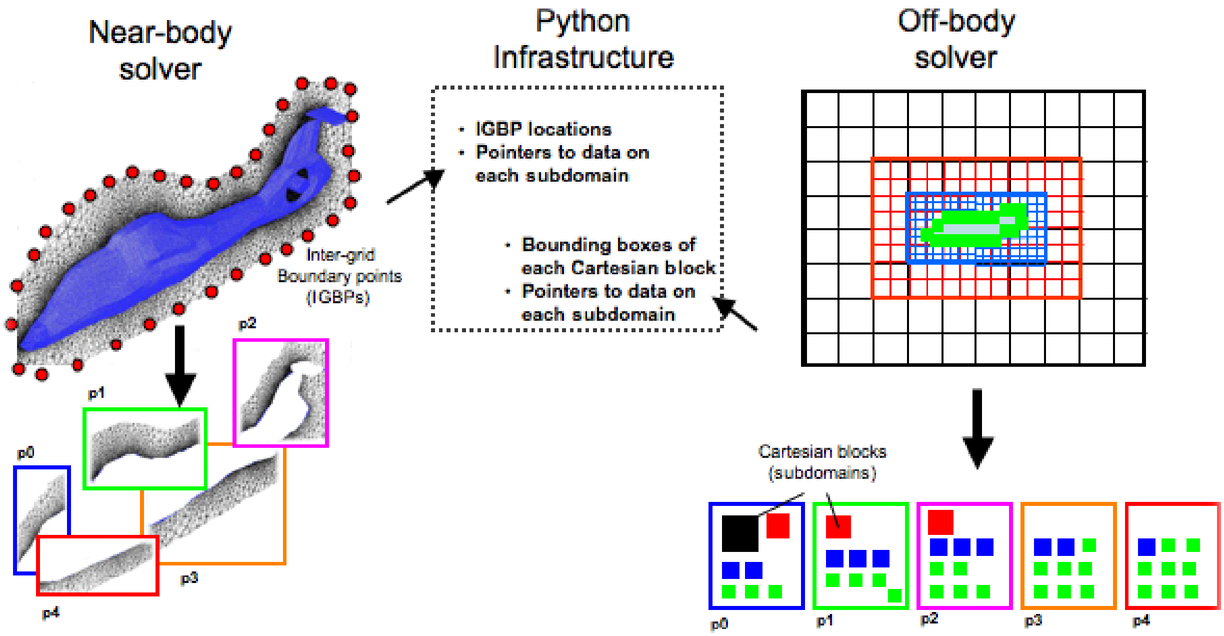


Figure 5. Code interfacing. Near and off-body solvers each run independently in parallel and exchange data only in their overlap region. Python maintains grid information and pointers to the data on the grids to facilitate inter-code data exchange.

of any grid point within a block can be deciphered from this information. Note that NSU3D does not use data from SAMARC, or vice-versa. All data exchanges that take place between the two are done through the DCF module, which decides the appropriate data interpolation for the state variables based on the the shared geometry data (Fig. 5). The arrays of grid and solution data are stored within Python as standard numerical arrays that can be operated on using Numeric[2] and SciPy[26] modules to perform fast and efficient numerical operations in Python. These modules have been shown in other works to be competitive with natively compiled code because they actually use a library of natively compiled functions (akin to BLAS routines in Fortran or C) for numerical operations.

The high-level Python script is quite concise and can be readily modified to perform more complex operations. For example, to invoke off-body grid adaptivity into the above example, the iteration loop in the SIF script could be modified as follows:

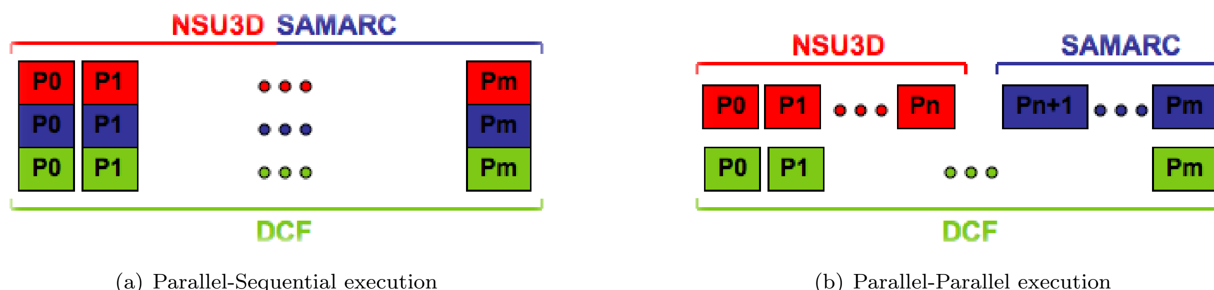
Python commands to incorporate off-body adaptivity

```
nadapt = 10   # adapt every ten iterations
for n in range(nsteps):
     # adapt grids
     if (n % nadapt == 0):
          offbody.adaptGrids()
          [gridOB,qOB,iblOB] = offbody.getData() # reset off-body grid/soln data
          dcf.checkOB(qOB,iblOB,...)             # register cartesian grid data with CHIMPS
          dcf.setInterfaces()                    # compute new near/off-body IGBPs
          dcf.update()                           # exchange data at IGBPs
     # continue iterative solution
     nearbody.runIteration()
     offbody.runIteration()
     dcf.update()                                # exchange data at IGBPs
```

The order of operations could also be easily modified to incorporate alternative time integration options. In practice, one needs to consider some peripheral operations like solution output, restart capability, residual convergence monitoring, etc. These can also be easily added to the interfaces of the respective modules as needed.

Note that the Python script is quite simple. The critical new capability of the infrastructure is the set of interfaces designed into the existing modules and the ability to orchestrate their operations through a simple concise Python script.

The example shown above can be executed on parallel computer systems using pyMPI[25] or myMPI,[13] allowing one to use the same parallel computer systems traditionally used for large-scale CFD calculations. At present, we are using the native load balancing capabilities of NSU3D and SAMARC to distribute the solver's data across processors, and then executing each module in a sequential fashion, as shown in Fig. 6(a). CHIMPS manages any inter-processor data communication that takes place when exchanging IGBP data between the two solvers. This has proven to be an effective strategy for the problems we have run to date. Figure 7 shows the execution time in each module, displayed using tools from the Tau package,[33] for a 16 processor adaptive time-dependent calculation of flow over a sphere (results shown in Sec. VI). It is clear from the figure that the level of load imbalance is relatively small. However, in the future as we move to larger problems run on more processors we envision adopting a strategy where different numbers of processors are assigned to the different solver modules, as shown in Fig. 6(b). This will require a more global load balancing strategy that assesses the computational load and synchronization points and re-partitions as needed.



(a) Parallel-Sequential execution

(b) Parallel-Parallel execution

Figure 6. Load balancing options. Each code could be distributed to all processors separately using its native load balancing scheme, and then run sequentially (a). Alternatively, the two codes could be assigned different numbers of processors (b). The latter option requires more coordination, balancing optimal load considerations with synchronization points.

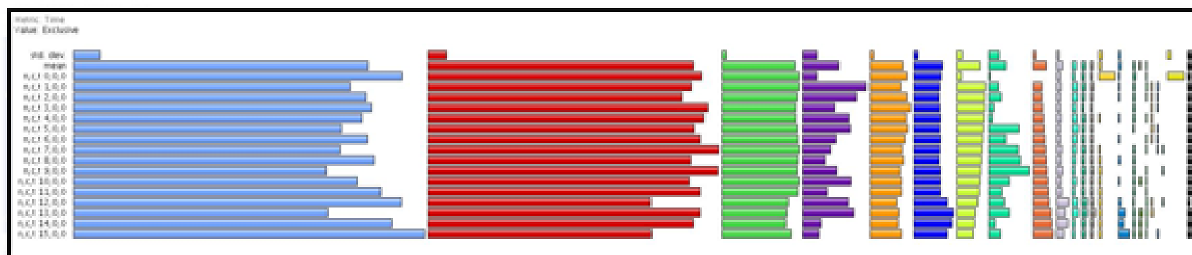


Figure 7. Display of execution time of the different modules for an adaptive calculation on 16 processors.

The modular approach used by the SIF makes substitution of alternative solvers reasonably straightforward. For example, suppose a user wishes to use the supported meshing strategy but with an alternative set of solvers. Since SAMARC adopts all of its adaptive meshing and parallel support from SAMRAI, it is straightforward to supply a new solver to replace ARC3DC. Any CFD code that operates on 3D structured cartesian grids can be incorporated with little modification. The capability to perform AMR and all parallel operations with the cartesian block solver are all provided within the infrastructure by SAMRAI. Likewise, it is straightforward to supply an alternative near-body solver by adding the wrapper code that provides the interfaces (described in Section IV). One difference, however, is that the SIF infrastructure assumes the near-body code manages its own parallel communication, so any near-body code used in the infrastructure

must, therefore, already be parallel. All Python interface software and the DCF module that provides the inter-grid interfacing between the cartesian adaptive off-body grids and the unstructured near-body grids can be used as is, requiring no modification. By packaging the Python SIF with SAMRAI and DCF, as shown in Fig. 8, the entire package is capable of providing parallel adaptive cartesian gridding support for nearly any combination of unstructured and structured cartesian solvers.

The flexibility and ease of reconfiguration is evaluated within the current infrastructure by plugging in a structured curvilinear grid solver UMTURNS[35] in place of NSU3D as the near body solver. Because of the standard APIs used in the code interfaces of the SIF, it was possible to plug in the new solver with only minimal effort to add the appropriate interfaces to the SIF and with no changes to the infrastructure itself. The UMTURNS code utilizes a 3rd-order upwind approach for spatial discretization and implicit second-order backward difference in time for integration of the discrete equations.

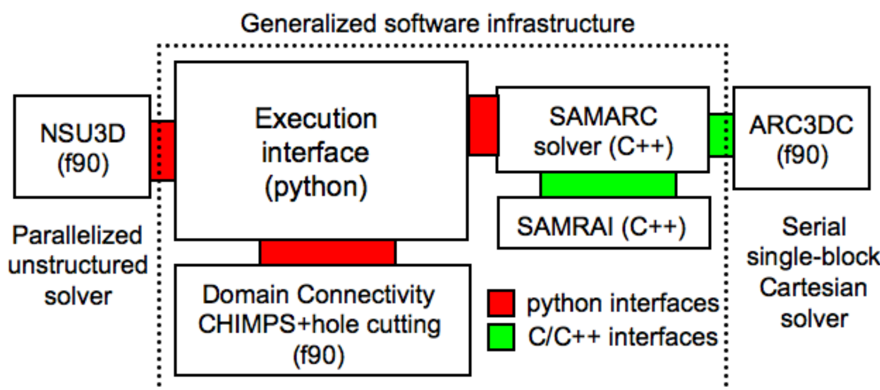


Figure 8. Software infrastructure. Different modules are connected through Python and C/C++ interfaces. Execution is controlled by Python, parallel AMR support for the cartesian off-body grid system is provided by SAMRAI, and interfacing between the near-body and off-body grid system is provided by a Domain Connectivity module. Outside plug-in modules include a parallelized unstructured flow solver and a serial single block cartesian grid solver.

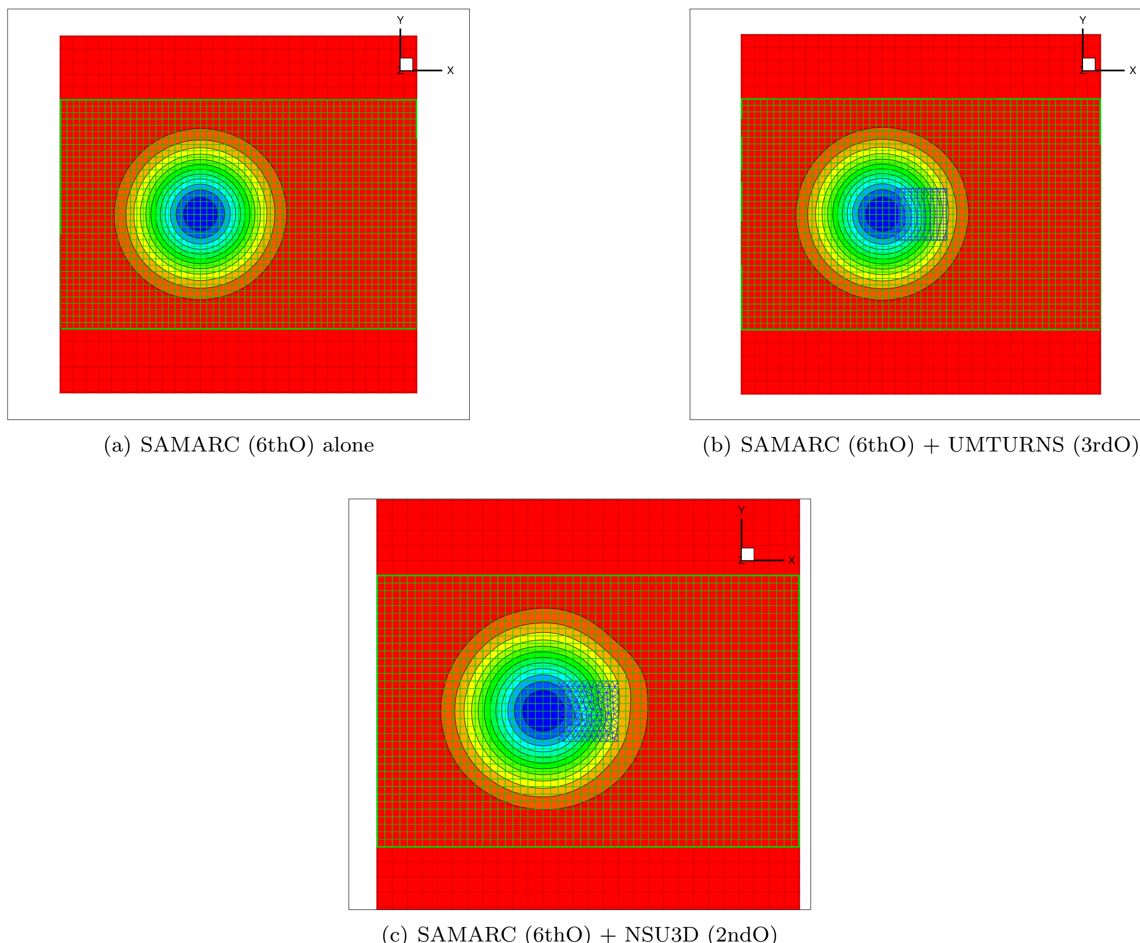
The flexibility to supply different solvers is useful for validating new software. New code modules can be easily substituted into the infrastructure and compared to known results of test problems run with the original module(s). Although this can in principle be done with two separate standalone solvers, this is often complicated when the codes use different problem setup steps (i.e., grid generation, scaling, input parameters, etc.). Being able to easily substitute different modules within the SIF, through the use of standard inputs and APIs, makes comparison of the results obtained with different modules more straightforward.

VI. Example Results

The infrastructure was evaluated for several several test problems to assess the operational and accuracy considerations of the multi-code paradigm. Descriptions of the validation problems here is cursory because a detailed validation study of this multi-code approach is presented in another paper at this conference.[34]

The first test case is convection of a Lamb vortex, for which the exact solution is known. The simulation consists of initializing the exact solution of the vortex and letting it evolve with time. The vortex continually moves through the domain in the x direction, with periodic boundary conditions at the upstream and downstream x boundaries — see Fig. 9. This problem is a good test of the amount of numerical diffusion present in spatial discretization schemes for inviscid fluid fluxes, since a more dissipative scheme will diffuse the vortex faster. It also tests the effect of moving the vortex feature between grid systems. Figure 9(a) shows the vortex structure after two passes through a two-level fixed cartesian grid system using the 6th-order accurate central-difference scheme in SAMARC alone. Approximately 20 points are used across the vortex core on the finest level. The vortex structure is preserved quite accurately, owing to the low dissipation numerical algorithm. Figure 9 (b) shows results with two codes using a small interior cartesian grid (shown in blue) region run with the 3rd-order upwind structured CFD solver (UMTURNS), and 6th-order SAMARC scheme used everywhere else. Although two different grids are used, both are cartesian with the resolution between the near- and off-body cartesian grids the same, so this test primarily demonstrates the effects of using two different solvers. Note that the vortex structure for this multi-code result is essentially identical to the SAMARC alone results. Figure 9 (c) shows the same case using a tetrahedral mesh with

the unstructured 2nd-order CFD solver NSU3D in an interior region and the 6th-order SAMARC scheme everywhere else. There is some slight deformation in this case, due to differences in both the numerical scheme and data interpolation (i.e., tetrahedral-cartesian rather than cartesian-cartesian) but the overall vortex structure remains largely intact with only slight distortion. Understanding the algorithmic effects of combining disparate numerical schemes and their optimal data exchange strategy is very complex and requires much more study, but these initial results are encouraging.



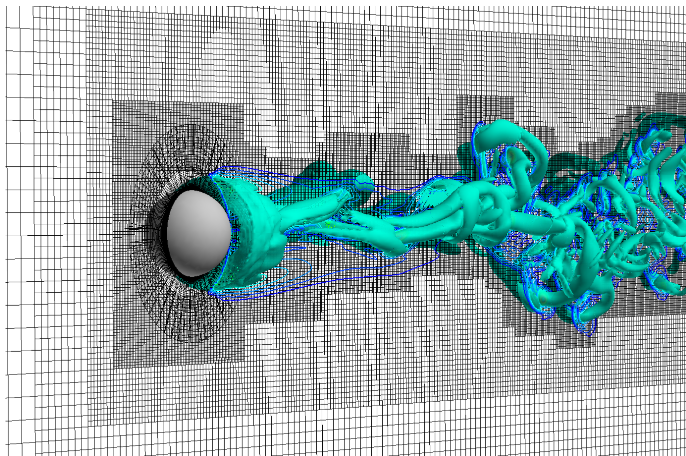
(a) SAMARC (6thO) alone

(b) SAMARC (6thO) + UMTURNS (3rdO)

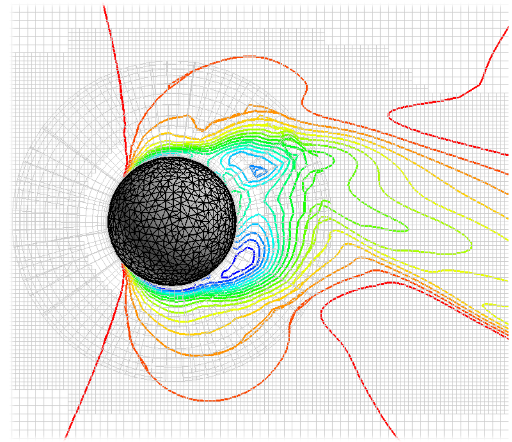
(c) SAMARC (6thO) + NSU3D (2ndO)

Figure 9. Density contours of Lamb vortex structure after two passes through the domain.

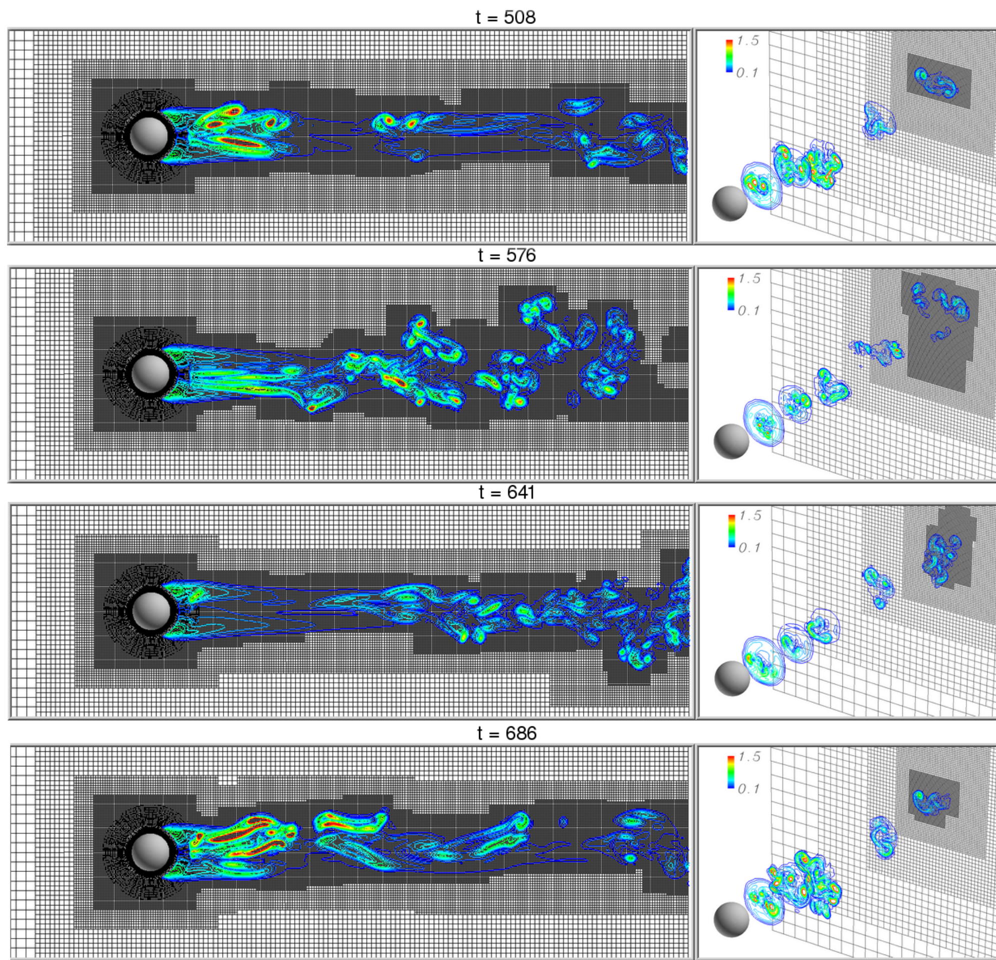
The next test problem is that of unsteady flow around a sphere. The physical characteristics of this problem, such as onset of instabilities and shedding frequency at different Reynolds numbers, are well known and documented both experimentally and computationally. The wealth of validation data available makes this problem useful to evaluate the accuracy of the multi-code paradigm. This case uses NSU3D in the near-wall region around the sphere, and adaptive grids in SAMARC in the farfield. A range of Reynolds numbers ($\mathrm{Re} = \frac{U_\infty D}{\nu}$), from Re=40 to 10000 was tested. The NSU3D grid contains approximately 50K prisms. The SAMARC grid used 6 levels of refinement and adapted time-dependently, with its size varying from 8M to 13M nodes. All runs were performed on 16 processors of an Opteron-based Linux cluster. A formal parallel scalability analysis was not conducted but Figure 7 shows that the multi-processor execution of the different modules is reasonably well load balanced, particularly considering that the grids adapt frequently. At $\mathrm{Re} < 160$ the flow is steady. Analysis of the separation angle and the location and size of the separation bubbles, shown elsewhere,[34] compares favorably with other experimental and computational results. Figure 10 shows results for Re=1000, which correctly predicts unsteady shedding. Part (a) shows an iso-surface of the 3D vortical structures overlaid on the adaptive off-body grids which adapt in time to regions of high vorticity. Part (b) shows the density contours at the grid overlap region between the two codes, showing the shedding characteristics and uniform transition of data in the near/off-body overlap region. Part (c) shows a 2D cross section of vorticity contours over a time sequence of approximately one



(a) Iso-surface of vorticity at $\omega = 0.55$



(b) Density contours

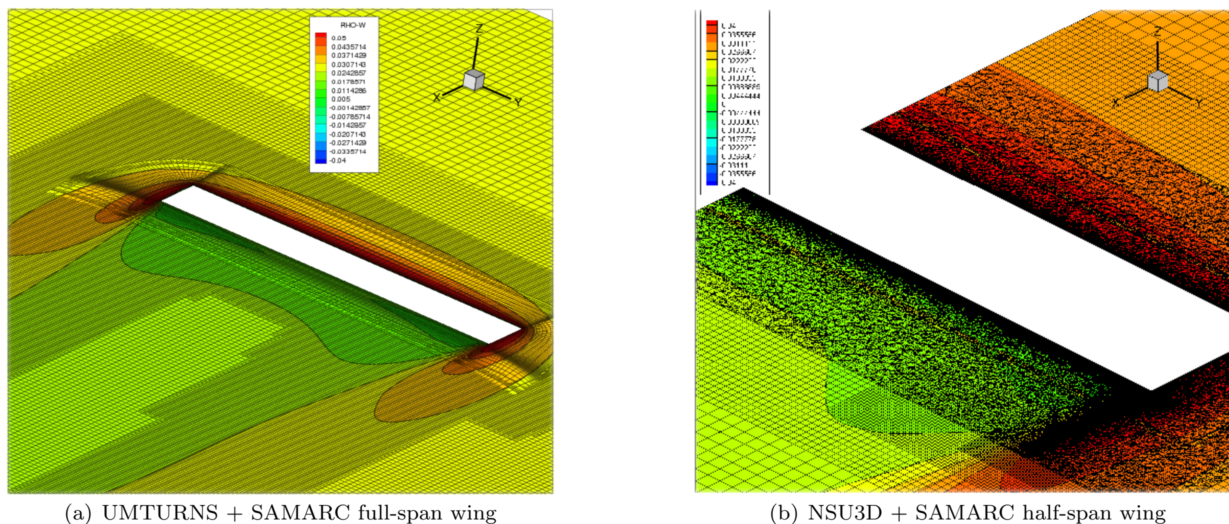


(c) Vorticity contours over one shedding period

Figure 10. Unsteady shedding with flow over sphere at Re=1000 using NSU3D + SAMARC. Cartesian grids adapt to regions of high vorticity.

shedding period. Parts (a) and (c) both show what appears to be larger structures transitioning to smaller-scale turbulent-like structures downstream. Transition to turbulence has been shown experimentally to take place at around Re=800. Analysis of the shedding frequency is currently underway.

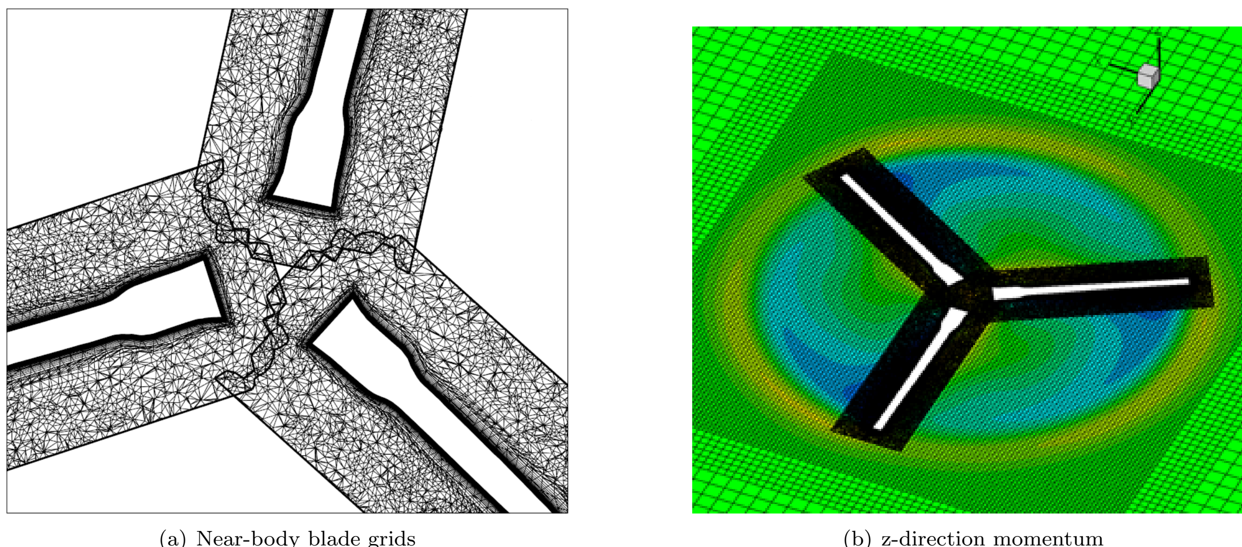
The next test problem is steady flow around a NACA0015 wing at Mach number 0.1235 with 12 degrees angle of attack. This case is tested with two different near-body grid approaches — a structured curvilinear full-span grid using the UMTURNS code, and an unstructured tetrahedral half-span grid using NSU3D. SAMARC is used as the off-body solver for both. This case assesses differences that arise from different near-body solution strategies (i.e. unstructured vs. curvilinear structured). It is also a good case to evaluate the ability of the off-body refinement strategy in SAMARC to capture the tip vortices downstream. The off-body grid used for this case is somewhat small with a total of three refinement levels. The problem is run on 8 processors. The off-body grids are adapted to regions of high vorticity. Figure 11 shows the downwash (i.e., z-direction momentum) for UMTURNS and NSU3D. Analysis of the loading (lift profiles) at different spanwise locations, shown elsewhere,[34] shows results between the two codes are comparable. Figure 11 (a) shows that the off-body grids are adapting well to locations of the tip vortex, although more refinement levels are needed.



(a) UMTURNS + SAMARC full-span wing

(b) NSU3D + SAMARC half-span wing

Figure 11. Contours of z-direction momentum in the z=0 plane, representing the downwash distribution, overlaid on adapted grid system $M = 0.1235, Re = 1.5e6, \alpha = 12^o$**.**



(a) Near-body blade grids

(b) z-direction momentum

Figure 12. Quarter-scale V-22 (TRAM) rotor in hover. Left: overset unstructured near-body grids. Right: z-direction momentum (downwash) contours overlaid on adaptive cartesian off-body grid system.

The final test case is the quarter-scale V-22 (TRAM) rotor in hover. This case is relevant to rotorcraft aeromechanics predictions, the class of problems for which this infrastructure is intended. Moving and rotating grid terms were added to both NSU3D and SAMARC in order to perform this steady-state simulation. The NSU3D grid consists of three tetrahedral grids around the blades, each with 256K nodes, giving a total of 800K near-body points — see Figure 12(a). The SAMARC grid contains 6 levels of refinement, ranges from 7M to 11M nodes, and adapts to regions of high vorticity. The case is run on 16 processors. Figure 12(b) shows contours of z-direction momentum (downwash). We are currently analyzing the blade loading and tip vortex characteristics, comparing to to experimental results.

VII. Concluding Remarks

This paper discusses a multi-code infrastructure that uses different gridding paradigms in different parts of the domain. The unstructured solver NSU3D is applied in the "near-body" region to capture geometry and near-wall boundary layer effects, while a high-order adaptive cartesian solver is used in the "off-body", a short distance from the wall into the far field. The block-structured AMR off-body solver is built from coupling the SAMRAI parallel AMR infrastructure with ARC3DC, a single block high-order cartesian solver. The motivation for this meshing approach is to create a CFD solution methodology which exercises the best features of both solvers — i.e., body-fitted grids to capture the geometry and boundary layer effects, and efficient adaptive cartesian methods for flow outside the boundary layer. The infrastructure utilizes existing standalone codes that are coupled through a Python infrastructure, thus avoiding the complex task of merging their respective algorithms into a single large monolithic code. The critical feature of the infrastructure is its ability to couple well-established and documented existing CFD codes to capture the near-wall viscous and turbulent effects, together with all the advantages of structured cartesian grids away from the wall (automatic grid generation, numerical efficiency, high-order accuracy, time-dependent AMR). The focus of this paper is primarily on the design of the infrastructure itself and the mechanisms used to turn existing standalone codes into modules that may be used in the infrastructure. Some example results are shown, but a more detailed analysis of the evaluation studies is presented in an accompanying paper.[34]

The multi-code infrastructure is evaluated for a suite of test problems to assess accuracy and performance. Validation problems include an analytic advecting Lamb vortex with a known exact solution, unsteady flow over a sphere at a range of Reynolds numbers, flow over a NACA 0015 wing, and a quarter-scale V-22 rotor in hover. In all the cases we used various near-body solution strategies (structured curvilinear vs. unstructured) and off-body grid refinement techniques. Preliminary results from these test problems indicate the infrastructure provides accurate and efficient performance.

Parallel scalability and performance must be addressed in future work. While we are cautiously optimistic because each of the code modules used in the infrastructure have separately been shown to scale to over 1000 processors, it nevertheless must be verified that this holds true when the codes are executed in a coupled fashion in the Python infrastructure.

The Python-based approach is an effective technique for merging existing existing flow solvers in a coupled simulation. By "Python wrapping" a standalone CFD code, one effectively builds an object-oriented module out of what was formerly a procedural-based code. This module can then be used in many different ways for different problems, facilitating large complex simulations that involve many interacting modules. It also permits software re-use and reduces maintenance through adoption of legacy solvers, rather than writing a new solver from scratch. The commercial software industry adopted an object-based approach to software development over two decades ago to manage large complex software packages. Likewise, as the scientific community pursues larger and more complex simulations, often involving different physics modeling or meshing strategies, there will likely be increased motivation to adopt similar object-based paradigms.

Acknowledgments

Development was performed at the HPC Institute for Advanced Rotorcraft Modeling and Simulation (HIARMS) located at the US Army Aeroflightdynamics Directorate at Moffett Field, CA, which is supported by the Department of Defense's High Performance Computing Modernization Office (HPCMO). The authors gratefully acknowledge the contributions of William Chan, Buvana Jayaraman, Aaron Katz, Robert Meakin, Robert Ormiston, Mark Potsdam, and Roger Strawn.

References

[1]Alonso, J., P. LeGresley, E. v. d. Weide, J. Martins, J. Reuther, "pyMDO: A Framework for High-Fidelity Multi-Disciplinary Optimization," AIAA-2004-4480, 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Albany NY, Aug 2004.

[2]Ascher, D., P. F. Dubois, K. Hinsen, J. Huginin, and T. Oliphant, "Numerical Python," Lawrence Livermore National Laboratory report UCRL-MA-128569. Also see `http://numpy.scipy.org/numpydoc/numpy.html`.

[3]Beazley, D. M., "SWIG: An Easy-to-use Tool for Integrating Scripting Languages with C and C++," 4th Annual Tcl/Tk Workshop, Monterey CA, July 1996. Also see `http://www.swig.org`.

[4]Buning, P. G., et. al. "OVERFLOW Users Manual," NASA Langley Research Center, July 2003.

[5]Berger, M. J., and P. Colella, "Local Adaptive Mesh Refinement for Shock Hydrodynamics," *J. Comp. Phys.*, **82**, 1989, pp. 65–84.

[6]Edwards, H. C., and J. R. Stewart, "SIERRA, a software environment for developing complex multiphysics applications," *Computational Fluid and Solid Mechanics: Proceedings of the First MIT Conference*, K.J. Bathe (Ed.), , Cambridge, MA, 2001, Elsevier, Oxford, UK, 2001, pp. 1147-1150.

[7]Gopalan, G., J. Sitaraman, J. D. Baeder and F. H. Schmitz, "Aerodynamic and Aeroacoustic Prediction Methodologies with Application to the HART II Model Rotor," Presented at the 62nd American Helicopter Society Annual Forum, Phoenix AZ, May 2006.

[8]Gunney, B. T. N., A. M. Wissink, and D. A. Hysom, "Parallel Clustering Algorithms for Structured AMR," *J. Parallel. Dist. Computing*, **66**, 2006, pp. 1419–1430.

[9]Hensaw, W. H., "Overture: An Object-Oriented Framework for Overlapping Grid Applications," AIAA-2002-3189, 32nd AIAA Fluid Dynamics Conference and Exhibit, St. Louis MO, June 2002.

[10]Henshaw, W. H., and D. W. Schwendeman, "Moving overlapping grids with adaptive mesh refinement for high-speed reactive and non-reactive flow," *J. Comp. Phys.*, **216**, Issue 2, Aug 2006, pp. 744–779.

[11]Hornung, R. D., and S. R. Kohn, "Managing Application Complexity in the SAMRAI object-oriented framework," *Concurrency and Computation: Practise and Experience*, Vol. 14, 2002, pp. 347-368.

[12]Hornung, R. D., A. M. Wissink, and S. R. Kohn, "Managing Complex Data and Geometry in Parallel Structured AMR Applications," *Engineering with Computers*, Vol. 22, No. 3-4, Dec. 2006, pp. 181-195. Also see `www.llnl.gov/casc/samrai`.

[13]Kaiser, T. H., "Using a Generalized MPI Interface for Python," SciPy 2005 Conference: Python for Scientific Computing, CalTech, Pasadena CA, Sept. 2005. Also see `http://peloton.sdsc.edu/ tkaiser/mympi/`.

[14]Kingsley, G., J. M. Siegel, V. J. Harrand, C. Lawrence, J. J. Luker, "Development of a Multi-Disciplinary Computing Environment (MDICE)," AIAA-98-4738, AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis MO, Sept 1998.

[15]Mavriplis, D. J., and V. Venkatakrishnan, "A Unified Multigrid Solver for the Navier-Stokes Equations on Mixed Element Meshes," *International Journal for Computational Fluid Dynamics*, Vol. 8, 1997, pp. 247-263.

[16]Mavriplis, D. J., "Multigrid Strategies for Viscous Flow Solvers on Anisotropic Unstructured Meshes," *Journal of Computational Physics*, Vol. 145, No. 1, Sept 1998, pp. 141-165.

[17]Mavriplis, D. J., and S. Pirzadeh, "Large-Scale Parallel Unstructured Mesh Computations for 3D High-Lift Analysis," *AIAA Journal of Aircraft*, Vol. 36, No. 6, 1999, pp. 987-998.

[18]Mavriplis, D. J., M. Aftosmis, and M. Berger, "High-Resolution Aerospace Applications using the NASA Columbia Supercomputer," *Paper presented at the 2005 Supercomputing Conference*, Seattle WA, Nov 2005.

[19]Mavriplis, D. J., "Results from the Third Drag Prediction Workshop using the NSU3D Unstructured Mesh Solver," AIAA-2007-0256, 45th AIAA Aerosciences Conference, Reno NV, Jan 2007. To appear, AIAA J. of Aircraft.

[20]Meakin, R., "An Efficient Means of Adaptive Refinement within Systems of Overset Grids," AIAA 95-1722-CP, 12th AIAA Computational Fluid Dynamics Conference, San Diego CA, June 1995.

[21]Meakin, R., "On Adaptive Refinement and Overset Structured Grids," AIAA-97-1858-CP, 13th AIAA Computational Fluid Dynamics Conference, Snowmass CO, June 1997.

[22]Meakin, R., and A. M. Wissink, "Unsteady Aerodynamic Simulation of Static and Moving Bodies using Scalable Computers," AIAA 99-3302-CP, 14th AIAA CFD Conference, Norfolk VA, July 1999.

[23]Meakin, R., "Automatic Off-body Grid Generation for Domains of Arbitrary Size," AIAA-2001-2536, 15th AIAA CFD Conference, Anaheim CA, June 2001.

[24]Meakin, R., A. M. Wissink, W. M. Chan, S. A. Pandya, and J. Sitaraman, "On Strand Grids for Complex Flows," AIAA-2007-3834, 18th AIAA Computational Fluid Dynamics Conference, Miami FL, June 2007.

[25]Miller, P., "pyMPI - An introduction to parallel Python using MPI," UCRL-WEB-150152. Also see `http://pympi.sourceforge.net`.

[26]Jones, E., T. Oliphant, P. Peterson, and others, "SciPy: Open source scientific tools for Python," 2001–. See `http://www.scipy.org`.

[27]Peterson, P. "F2PY Users Guide and Reference Manual," See `http://cens.ioc.ee/projects/f2py2e/`.

[28]Power, G. D., and J. A. Calahan, "A Flexible System for the Analysis of Bodies in Relative Motion," AIAA 2005-5120, 35th AIAA Fluid Dynamics Conference and Exhibit, Toronto Ontario Canada, June 2005.

[29]Pulliam, T. H., "Solution Methods in Computational Fluid Dynamics," *von Karmon Institute for Fluid Mechanics Lecture Series, Numerical Techniques for Viscous Flow Computations in Turbomachinery*, Rhode-St-Genese, Belgium, Jan 1986. See `http://people.nas.nasa.gov/ pulliam/mypapers/vki_notes/vki_notes.html`.

[30]Pulliam, T. H., "Euler and Thin-Layer Navier-Stokes Codes: ARC2D, and ARC3D" Computational Fluid Dynamics Users Workshop, The University of Tennesse Space Institute, Tullahoma TN, March 1984.

[31]Schluter, J. U., X. Wu, S. Kim, J. J. Alonso, and H. Pitsch, "Integrated Simulations of a compressor/combustor assembly of a gas turbine engine," ASME paper No. GT2005-68204, ASME Turbo Expo 2005, Reno NV, June 2005.

[32]Schluter, J. U., X. Wu, E. v. d. Weide, S. Hahn, J. J. Alonso, and H. Pitsch, "Multi-Code Simulations: A Generalized Coupling Approach," AIAA 2005-4997, 17th AIAA Computational Fluid Dynamics Conference, Toronto Ontario Canada, June 2005.

[33]S. Shende, and A. D. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, SAGE Publications, Vol. 20, No. 2, 2006, pp. 287–331.

[34]Sitaraman, J., A. Katz, B. Jayaraman, A. Wissink, V. Sankaran, "Evaluation of a Multi-Solver Paradigm for CFD using Unstructured and Structured Adaptive Cartesian Grids," AIAA-2008-0660, 46th AIAA Aerosciences Conference, Reno NV, Jan 2008.

[35]Sitaraman, J., and J. D. Baeder, "Evaluation of the Wake Prediction Methodologies used in CFD Based Rotor Airload Computations," AIAA-2006-3472, 24th AIAA Applied Aerodynamics Conference, San Francisco CA, June 2006.

[36]Spalart, P. R., and S. R. Allmaras, "A One-equation Turbulence Model for Aerodynamic Flows," *La Recherche Aérospatiale*, Vol. 1, 1994, pp. 5–21.

[37]Stewart, J. R., and H. C. Edwards, "The SIERRA framework for developing advanced parallel mechanics applications," in: *Large-Scale PDE-Constrained Optimization*, LL. Biegler, O. Ghattas, M. Heinkenschloss, B. van Bloemen Waanders (Eds.), Vol. 30, Springer, Berlin, July 2003.

[38]Wilcox, D. C., "Re-assessment of the Scale-determining Equation for Advanced Turbulence Models", *AIAA Journal*, Vol. 26, 1988, pp. 1414–1421.

[39]Wissink, A. M., D. A. Hysom, and R. D. Hornung, "Enhancing Scalability of Parallel Structured AMR Calculations", Proceedings of the 17th ACM International Conference on Supercomputing (ICS03), San Francisco CA, June 2003, pp. 336–347.

[40]Wissink, A. M., R. D. Hornung, S. Kohn, S. Smith, and N. Elliott, "Large-Scale Parallel Structured AMR Calculations using the SAMRAI Framework", Proceedings of Supercomputing 2001 (SC01), Denver CO, Nov 2001.

[41]Yang, Z., and D. Mavriplis, "Unstructured Dynamic Meshes with Higher-order Time Integration Schemes for the Unsteady Navier-Stokes Equations," AIAA-2005-1222, 43rd AIAA Aerospace Sciences Meeting and Exhibit, Reno NV, Jan 2005.

[42]Yang, Z., and D. Mavriplis, "Higher-order Time Integration Schemes for Aeroelastic Applications on Unstructured Meshes," AIAA-2006-0441, 44th AIAA Aerospace Sciences Meeting and Exhibit, Reno NV, Jan 2006.