47th AIAA Aerospace Sciences Meeting, 5-9 Jan. 2009, Orlando, FL

Improvements to SUGGAR and DiRTlib for Overset Store Separation Simulations

Ralph W. Noack*
David A. Boger†
Applied Research Laboratory, The Pennsylvania State University, USA

SUGGAR is a general overset grid assembly capability that is targeted at moving body simulations. DiRTlib is a library that encapsulates the functionality required to perform the overset interpolation and communications required in an overset composite grid solution. This paper describes enhancements that have been made to SUGGAR and DiRTlib to improve the performance and capability for moving body problems. A procedure to adapt the grids to overlap requirements is demonstrated and found effective in reducing or eliminating the orphans due to insufficient overlap. In addition, SUGGAR can also be linked into the flow solver as a library to provide a capability integral to the flow solver. The paper documents the library interface and discusses modifications required to the flow solver to utilize SUGGAR as a library. Finally, a new overset domain connectivity code, Suggar++, is being developed, and this paper briefly discusses the improvements it offers relative to SUGGAR.

I. Introduction

The overset, or chimera, grid methodology[1] utilizes a set of overlapping grids to discretize the solution domain. Grid generation can be greatly simplified for complex geometries by constructing component grids for portions of the geometry with minimal regard for other portions of the geometry. The result is a flexible computational simulation framework that can be an enabling force in many situations. The overset approach has been widely used to simplify the structured grid generation requirements for complex geometries.

The overlapping grid system must be processed to form a composite grid where the solution from one grid is linked to the solution on an overlapping grid. This process begins by identification of "hole" points in a component grid that are outside the domain of interest and should be excluded from the flow solver computations (commonly called OUT points) such as points inside a body or behind a symmetry plane. Next, the grid points adjacent to the holes along with points on grid boundaries in the overlap regions become intergrid boundary points termed receptor or fringe points. Finally, appropriate donor elements are found to function as the interpolation sources for the boundary values required by the flow-field solution at the fringe points. The overset domain connectivity information (DCI) consists of the specification of hole and fringe points along with donor interpolation information that is required to form the overset composite grid and solution and is produced by the overset grid assembly process.

The use of an overset grid system is an enabling technology for the simulation of bodies in relative motion, such as store separation[2,3] and rotorcraft.[4] The component grids move rigidly with their associated body

*Senior Member AIAA
†Member AIAA

components such as the store or rotor blade. The grid assembly process must be performed during the flow solver time stepping to update the overset domain connectivity information as the grids change position. The motion of the bodies will either be specified for known motion, such as a rigid rotor blade, or else the motion will be calculated as part of the solution by using forces and moments predicted by the flow solver as inputs to a rigid body dynamics simulation.

The current production capabilities for overset CFD store separation simulation primarily use structured grids. The BEGGAR code[5] along with Overflow2[6] have an integrated overset grid assembly process for moving body problems. A loosely coupled/non-integrated capability has been used[7] with NXAIR[8] for the flow solver, PEGSUS4[9] for the overset grid assembly and FDcadre[10] as the simulation executive to control the various codes. Significant interest in the use of overset unstructured grids for store separation analysis is emerging. The current demonstrated unstructured store separation capability in a production environment[11] uses the USM3D flow solver[12,13] and SUGGAR[14] for the overset grid assembly.

Currently SUGGAR is the only overset grid assembly capability for use with structured, unstructured, and other grid topologies. It has been applied to a wide range of moving body problems including store separation,[15] rotorcraft,[16] and general ship motion.[17] While SUGGAR has enabled new simulation capabilities, some deficiencies have limited its wider use in the past. These include the use of a named pipe/FIFO for communication with the flow solver, no overlap minimization for unstructured grids, overlap requirements for unstructured grids, and parallel execution performance. This paper will address improvements to SUGGAR that remove some of these deficiencies. In addition, this paper will briefly introduce Suggar++, which is a new code intended to provide the same basic capabilities as SUGGAR but without the deficiencies.

II. New SUGGAR Capability

Significant new capability has been added to SUGGAR to improve the capability for simulation of moving body problems. This section will discuss these improvements and demonstrate their effectiveness.

A. Overlap Minimization for Unstructured Grids

SUGGAR's capability for minimizing the overlap between component grids has been available for structured grids, and this capability was recently extended to unstructured grids. Minimizing the overlap between component grids can improve the solution quality by having the donor element and fringe point elements as close as possible in size. Without the overlap minimization procedure, the user must adjust the size of the hole cutting geometry to manually reduce the overlap. Having the automatic overlap minimization procedure for unstructured grids significantly simplifies SUGGAR's inputs and reduces the time-consuming trial-and-error procedure required to adjust the hole cutting parameters to blank more of the grid in the overlap region. Figure 1 shows a cutplane through the grid system for a helicopter rotor blade without using the minimization process. The blade geometry is shown in gray, the grid surrounding the blade is colored blue, and the background grid is red. The points marked as OUT are not displayed and one can see that while the hole cut in the background grid by the blade is large there is still significant overlap. Figure 2 shows a cutplane through the same grid system after enabling the overlap minimization procedure. One can see that the procedure was effective in significantly reducing the overlap between component grids. In addition, there are small isolated patches of elements in the background grid that overlap the blade grid but have not been removed. These patches results from using the cell volume as the donor suitability function (DSF)[18] in the minimization process and the non-smoothness of the cell volume in the tetrahedral mesh. SUGGAR allows the user to set the DSF in the mesh as a means to control the overlap minimization. Setting the DSF to zero in the blade grids will force the blade grid to have preference over the background grid and will eliminate the small isolated patches of elements as shown in Figure 3.

B. Reducing Orphans in Unstructured Grids by Adaption

The ideal overset system of component meshes will have each mesh with elements of the same size in the overlap region along with fringe donor elements that are high quality, i.e. the donor interpolation stencil does not include any members that are also fringe locations. This can be difficult to achieve, especially with unstructured grids where large surface faces will typically result in large tetrahedron at the end of a viscous prism layer. Controlling this rapid stretching can be difficult in some grid generation packages. A common approach within the structured overset community is to simply insert an additional interface mesh in the region where the overlap is insufficient to maintain a quality overset composite grid. Unstructured interface grids can be added, as in Reference 13, to provide additional resolution in appropriate regions and eliminate the orphans. Generating the interface grids can be time consuming at best.

A more automated approach uses unstructured grid adaption to refine the component meshes in the region of the orphans. One first such attempt[11] successfully implemented such an approach where the orphan elements are adapted. A better approach is to refine any candidate donor elements that were found for the orphan but were rejected because of poor quality. This requires more information from the overset grid assembly code than is typically available in the overset domain connectivity information. Refining only the orphans fails to address the underlying cause of the problem, which is that the donor element in the region is low quality because it is too large. A successful refinement procuedure must adapt the candidate donors for the orphans, which requires additional information from the overset grid assembly code.

SUGGAR can now assist directly in the adaptive refinement process by outputting lists of elements that need to be refined to improve the overlap. These lists include orphans elements, candidate donors for the orphans, and candidate donors that are too large for their fringe point. These files are used as input to the NASA LaRC H-refinement code[19] to refine the specified elements and improve the overlap. Iteratively applying the process can significantly reduce or eliminate the orphans in the unstructured overset composite grid.

The adaption to overlap requiements is demonstrated here using the Eglin Wing/Pylon/Store configuration shown in Figure 4. The baseline grid system consists of one grid encompassing the wing with pylon and extending to the farfield and a second grid around the store extending out a relatively short distance from the body as shown in Figure 5. Using SUGGAR as the grid assembly process for this baseline configuration produces overset domain connectivity information with a large number of orphans in the region between the pylon and the store as shown in Figure 6. The orphans are represented as small squares with the wing grid orphans colored cyan and the store orphans colored red. A cutplane through the grid system is shown in Figure 7 to illustrate the insufficient overlap resulting from the rapid stretching in the component grids. Figure 8 shows a complicated system of interface grids that was used in a previous effort[13] to reduce the orphan count. Generating the individual interface grids was time consuming because each iteration required generating a mesh, performing the overset assembly, diagnosing the remaining orphans, and then inserting another grid or modifying an existing one.

An automated adaptive process to eliminate orphans can be scripted as follows:

1. Run SUGGAR using a system of overset component grids and output the list of elements to refine.
2. Stop if the number of orphans is acceptable.
3. Refine each component grid by using the list of elements to refine as input to the H-refinement code.
4. Create a new SUGGAR input file that uses the refined component grids
5. Go back to 1 and repeat.

Applying this automated adaption process to eliminate orphans for the Wing/Pylon/Store configuration produces the results shown in Table 1 for three different refinement options. The first option that was tested refined only the orphan elements and resulted in a relatively small decrease in the orphan count. Refining only the orphans does not directly address the cause of the orphans, which is poor quality donors.

Refining the orphans can actually make the number of orphans increase if the candiate donor elements are not improved and remain poor quality.

The second option that was tested refined the orphans and their candidate donors and showed a dramatic reduction in the orphan count with a modest increase in grid size. Neighboring elements to the donor elements were also included to try to obtain a smoother adapted grid and to expand the refinement region. This approach directly addresses the cause of the orphans by reducing the size of the donor elements. Refining the mesh in the region of the donors will result in a tighter hole cut because the elements being cut are smaller. Hence the donor elements have an improved chance of having adequate quality. A cutplane through the grid system after two adaption cycles is shown in Figure 9, and comparing this with the baseline grid system shown in Figure 7 clearly shows the improved overlap in the adapted grids. The orphans remaining after the second adaption cycle are shown in Figure 10.

The third option that was tested refined the orphans and their candidate donors along with any donor that was too large by a factor $f = 2.5$ than the fringe point/element. Again, a dramatic reduction in the orphan count was achieved but at the cost of a significant increase in grid size. Refining the donors that are "too large" refines the mesh in the region of the outer boundary of the store grid in addition to the region of the orphans. The exponential increase in grid size results from the newly refined elements being too small relative to the other mesh causing the other mesh to be refined in the next iteration. Attempting to refine based upon donors that are too large must be exercised with care to prevent an explosize growth in the grid size. The primary goal of the refinement is to improve the grid to reduce the orphan count, and any other refinement should probably be based upon flow solver solution accuracy requirements.

This adaptive process is being used during a time dependent store separation simulation[20] to maintain a resonable quality overset composite grid throughout the separation trajectory. The need for a complicated derefinement process is avoided by starting each refinement cycle using the set of baseline grids.

C. Using SUGGAR as a Library

An overset grid simulation of moving body problems requires the overset domain connectivity information to be recomputed during the simulation, typically at every time step. This requires the current position of moving bodies to be communicated to the overset grid assembly process, where the new DCI must be computed and then transferred back to the flow solver.

The process that has been used with SUGGAR[14] will only be summarized here. SUGGAR is run as a separate process from the flow solver with communication of the position of the moving body along with synchronization occurring via a standard Unix named pipe. A DCI file is written by SUGGAR and read by the flow solver. The advantage of this approach is that modifications to the flow solver to interface with SUGGAR are limited to reading and writing a few files. Using the named pipe and DCI file, while simple and effective, imposes some penalties and limits the acceptable computing environment. The use of the named pipe requires the SUGGAR process to reside on the same compute node as the flow solver process communicating with SUGGAR. This limits SUGGAR to executing on compute nodes with sufficient memory for both SUGGAR and the flow solver at the same time. In addition, writing and then reading the DCI file can take significant wall clock time for large DCI files and slow filesystems.

To eliminate these problems, SUGGAR has been modified to allow it to be linked as a library into the flow solver executable. The flow solver code makes function calls to the library to control the execution of SUGGAR and to directly transfer the overset domain connectivity information from libSuggar to the flow solver. This capability eliminates the use of the named pipe for communication and synchronization between the flow solver and SUGGAR. In addition, the SUGGAR execution and hence memory and CPU resources can be placed on a dedicated rank to eliminate contention with the flow solver.

DiRTlib has also been modified to make calls to libSuggar and directly obtain the overset domain connectivity information from libSuggar rather than reading it from a file. Solvers using DiRTlib benefit by the simpler interface that DiRTlib provides relative to making all the libSuggar calls required to transfer the data. In addition, DiRTlib will now work with a solver where a single structured grid is decomposed into

Baseline: No Refinement			
	Total Number		
	Orphans	Points	Elements
	67905	1128740	6580833

Refining: Orphans only					
Adapt Cycle	Total Number			Ratio to Baseline	
	Orphans	Points	Elements	Points	Elements
1	57197	1128740	7153857	1.00	1.09
2	14733	1359764	7909248	1.20	1.20
3	6253	1402943	8150710	1.24	1.24

Refining: Orphans and Candidate Donors					
Adapt Cycle	Total Number			Ratio to Baseline	
	Orphans	Points	Elements	Points	Elements
1	6569	1249344	7284689	1.11	1.11
2	399	1285386	7490352	1.14	1.14
3	262	1289123	7511166	1.14	1.14

Refining: Orphans, Candidate Donors, Too large donors($f = 2.5$)					
Adapt Cycle	Total Number			Ratio to Baseline	
	Orphans	Points	Elements	Points	Elements
1	6472	2264136	13263031	2.01	2.02
2	384	5760156	33425762	5.10	5.08
3	167	11143409	63954604	9.87	9.72

Table 1. Effect of adaption cycles on orphan count for Wing/Pylon/Store

subgrids for parallel execution.

Some significant modifications to the flow solver are required to interface to libSuggar. For this discussion the flow solver is assumed to utilize MPI for its message passing communications during parallel execution. The following steps are suggested as a way to incrementally integrate libSuggar into the flow solver and allow testing at each stage. This discussion will assume that the flow solver already uses DiRTlib for its overset capability. libSuggar has methods to transfer the DCI directly for those solvers with an internal overset capability that are not using DiRTlib. This paper will not discuss the additional complexity of using libSuggar without using DiRTlib.

The discussion that follows will present a general application programmer interface (API) that could be used by other domain connectivity codes. The discussion will present the C API that is callable from C or C++ codes. A Fortran API is also available with very similar names and arguments. libSuggar provides an implementation of the API discussed, and a C header file and Fortran 90 module file that provide prototypes for the API are available to allow the compiler to check function names and arguments. The API is broken into two subsets. The first set of functions controls the execution of the domain connectivity code, and all functions in this first set begin with a prefix of *"dc_"* for the C API and *"dcf_"* for the Fortran API. The second set of functions controls the transfer of information from the domain connectivity code to the flow solver, and all functions in this second set begin with a prefix of *"dcx_"* for the C API and *"dcxf_"* for the Fortran API.

1. Validate Use of DiRTlib for Moving Body Problems

If the flow solver does not have an internal overset capability then the addition of calls to DiRTlib to add the overset capability is required. Reference 21 documents the use of DiRTlib within the flow solver. The suggested approach is to modify the flow solver to make the appropriate calls to DiRTlib and test the overset capability first for static problems and then for moving body problems. A simple approach for testing the capability for moving bodies is to use a test case with prescribed motion such as an oscillating airfoil or the Eglin Wing/Pylon/Store trajectory obtained from wind tunnel tests. For problems with prescribed motion the set of DCI files can be pre-generated for each time step. One approach to accomplish this is to generate the motion history input required by SUGGAR using a simple script that writes the history to a file. This is the dynamic body positioning information that would have been written to the Fifo_SUGGAR.inp file in Reference 14. It is simply a set of XML elements that specify the dynamic body hierarchy and the transformation that should be applied to each body. In addition, XML elements can be included to select writing the DCI to a specified file that should be unique for each time step. This motion history input file can be provided to the SUGGAR executable to play the motion and generate the set of DCI files. The flow solver can then load the DCI file that is appropriate for each time step while computing the solution for the prescribed motion. With this capabiliity, the use of DiRTlib for moving body problems is validated by loading a new DCI file at each time step.

2. Validate Execution of libSuggar at Each Time Step

Next the calls to execute libSuggar at each time step are added. libSuggar will still write the DCI file, and the DiRTlib calls used in the previous section are still used to read the DCI file.

A simple domain connectivity (DC) application programmer interface (API) has been developed to control the execution of the domain connectivity code. In the present effort, the capabilities of SUGGAR are exposed as libSuggar through this API. The discussion that follows will present the functions only for the C API. The following stages in execution of the DC code and the corresponding library calls are as follows:

1. **Initialize the library and the current configuration.** This stage should initialize the library, read any grids, compute any required data structures, and be ready to compute the DCI for the first time step. A file name for a DC code-specific input file is passed to the library to simplify this initialization.

This file will typically be the same file as provided to the DC code when running as a stand-alone process, which allows the input file to be prepared and tested outside of execution with the flow solver.

The C prototype for the initialization functions are

```
void dc_init(int argc, char **argv);
void dc_init_no_args(char* input_filename);
```

where *argc* is the number of command line arguments, *argv* is the list of command line arguments, and *input_filename* is a null-terminated character string containing the filename of the code- and configuration-specific input file.

2. **Motion input.** The next set of routines provides a mechanism to pass information required to specify the positioning of any dynamic bodies. To maximize the generality of the approach, all information is passed as individual character strings that each domain connectivity code can interpret appropriately. For libSuggar, each character string will be a line of the XML file that would have been written to the named pipe/FIFO.

   The C prototype for the functions to specify the motion positioning information are

   ```
   void dc_begin_motion_input(void);
   void dc_add_motion_input(char* buffer);
   void dc_end_motion_input(void);
   void dc_parse_motion(void);
   ```

   where *buffer* is a null-terminated character string containing a portion of the motion input specification. A new motion input specification is initiated with a call to *dc_begin_motion_input*. Multiple calls to the *dc_add_motion_input* function may be required to fully specify the position information, and the domain connectivity code should copy and assemble the data passed through multiple calls appropriately. The *dc_end_motion_input* function is called when all of the positioning information has been passed through the calls to *dc_add_motion_input*. Finally, the *dc_parse_motion* function is called to allow the domain connectivity code to process the positioning information and setup appropriate data structures for computing the DCI with the dynamic bodies in the new positions.

3. **Compute DCI at a timestep.** Once the new positions of the dynamic bodies have been configured, the DCI for this new configuration can be computed.

   The C prototype for the function to compute the DCI for the new configuration is

   ```
   void dc_compute_dci(int output_dci);
   ```

   where a value of *1* for *output_dci* will write the new DCI to a file and a value *0* will prevent the DCI file from being written.

4. **Release memory used in computing DCI.** Once the DCI has been written (or transferred to the flow solver as discussed below), the temporary memory used by the DC library must be released or freed to prevent memory leaks.

   The C prototype for the function to release or free any temporary memory used to compute the DCI is

   ```
   void dc_release_dci(void);
   ```

5. **Release ALL memory used by the library.** At the end of the moving body simulation the flow solver may wish to release all of the memory utilized by the DC library. This call is unnecessary if it is called immediately before program termination; however, if the solver will discontinue calling the DC library before the end of execution it may be useful to free the memory for reuse by the solver.

   The C prototype for the function to release or free all memory used by the library is

   ```
   int dc_terminate(void);
   ```

3. Modify Flow Solver to Execute Domain Connectivity on a Dedicated Rank

The domain connectivity library can require significant memory and processor resources. Some computational environments may not have sufficient system memory to execute both in the same rank/process. Therefore, it is desirable to have the DC library execute on a dedicated rank, which will require more extensive modifications to the flow solver.

The first step is to modify the flow solver to split *MPI_COMM_WORLD* and use the new communicator. This splitting will be such that the "flow solver only" ranks will be in one communicator and the "DC only" rank will be in a separate communicator. *MPI_COMM_WORLD* can still be used to communicate between a flow solver rank and the DC rank, such as may be required to communicate the new dynamic body positions computed by the flow solver rigid body dynamics to the DC procedure. Figure 11 illustrates the decomposition of the process along with the split communicators.

The MPI function to split *MPI_COMM_WORLD* is

```
MPI_Comm_split(MPI_COMM_WORLD,color,key,&new_comm);
```

where *color* is an integer that will have one value for the flow solver ranks and another for the DC ranks. The *key* is another integer that can be used to reorder the ranks within the new communicator. Finally *new_comm* is the new split communicator that the flow solver should use. By using the new communicator, the flow solver will continue to work as before with collective operations operating on only the flow solver ranks.

The next step is to reorganize the flow solver routines so that the flow solver functions are called on the ranks dedicated to the flow solver and the routines in the DC library discussed in Section 2 are called on the ranks dedicated to domain connectivity. Depending on the modularity and complexity of the flow solver software, this may be easy or difficult, but it is beyond the scope of this paper.

For this stage in the development process, the DCI should still be exchanged via a file by calling *dc_compute_dci(1)* and having DiRTlib read the file.

4. Modify the Calls to Directly Transfer the DCI

The final stage in the integration of libSuggar is to enable the transfer of the DCI via MPI from the DC rank to DiRTlib and the flow solver rather than writing and reading the DCI file. The domain connectivity exchange portion of the API (function calls with a prefix of *dcx_*) provides several different styles of data transfer between the DC library and the flow solver. This discussion is limited to solvers that use DiRTlib. Such solvers require a simpler set of function calls because DiRTlib encapsulates the data transfer function calls.

The first step is to define which ranks will host the flow solver work and which rank will host the SUGGAR process. Using more than one rank for the domain connectivity execution is allowed by the API but has not been tested in libSuggar.

Since DiRTlib will have calls executing on the flow solver and DC ranks, some additional initialization calls are required. DiRTlib and libSuggar must know if they are being called from a flow solver or a DC only rank. The following API calls inform the libraries as to the purpose of each rank:

```
dc_status dc_rank_dci_only(void);
dc_status dc_rank_not_flow_or_dci(void);
dc_status dc_rank_flow_and_dci(void);
dc_status dc_rank_flow_only(void);
```

One of these functions must be called on each rank before the *drt_pll_init* function call.

Next, the DC library must be informed as to which rank in *MPI_COMM_WORLD* is assigned to be the master rank of the DC ranks.

The C prototype for the function to set the master rank is

```
dcx_status dcx_set_dci_master_rank_comm_world(int rank);
```

This function must be called on each rank after the *dc_rank_* function call and before the *drt_pll_init* function call.

The final step is to change from having DiRTlib load the DCI from a file to passing it directly from the DC rank. This is achieved by changing the call to load the DCI file header, which was

```
call drtf_load_dci_file_header(dci_file_name)
```

to the call to directly pass the header information, which is

```
call drtf_get_dci_header()
```

and likewise, to change the call to load the dci file, which was

```
call drtf_load_dci_file_header(dci_file_name)
```

to the call to directly pass the domain connectivity information, which is

```
call drtf_get_dci()
```

III. Future Improvements

The new capability provided in DiRTlib and libSuggar enable a significantly improved store separation simulation environment. The ability to link the domain connectivity code in as a library significantly improves the execution because the DC can be executed on a dedicated node which eliminates resource contention with the flow solver. The use of DiRTlib and libSuggar provides an integrated overset domain connectivity capability to any flow solver, structured or unstructured, in a fashion similar to the integrated overset grid assembly capability in BEGGAR,[5] which is restricted to structured grids.

A. Deficiencies in SUGGAR

There are areas where additional improvements relative to SUGGAR are required. Two significant areas of deficiency are the memory requirements for hole cutting and the performance for parallel execution.

1. Memory Requirements

The hole cutting geometry in SUGGAR is created using an octree, which requires refinement in all directions even when there is an obvious preferential direction. This leads to large memory requirements where a highly accurate hole cut is required, such as the gap between the pylon and a store in a store separation simulation.

2. User Inputs to Control the Hole Cutting

SUGGAR can require significant user interaction to iteratively modify the inputs to improve the resolution of the hole cut octree to cut tighter holes where needed. The octree can not simply be refined everywhere because the memory requirements can become excessive.

3. Parallel Execution

Poor parallel performance is also a significant weakness of SUGGAR. The MPI parallel capability in SUGGAR duplicates all data structures on all ranks, which is prohibitive for large problems. SUGGAR can utilize multiple threads for parallel execution on shared memory machines, but load balancing deficiencies limit the scalability of the threads. In addition, SUGGAR has no mechanism to distribute the memory requirement across multiple nodes, which then requires a machine with large memory for large problems.

B. Improvements in Suggar++

A new overset domain connectivity capability is being developed in Suggar++.[22] This section will highlight the planned and implemented improvements in Suggar++ and provide some preliminary results.

The most significant change in Suggar++ relative to SUGGAR is the use of a direct cut approach to hole cutting.[22] This is a more complex and time consuming approach relative to the approximate geometry approach used in SUGGAR. The benefit is that the direct cut approach will cut much tighter holes because it uses the actual geometry rather than an approximate geometry. The result will be fewer orphans relative to SUGGAR in areas where gaps are small.

The benefit of the tighter hole cut is dramatically demonstrated by applying Suggar++ to the Wing/Pylon/Store grid system used in Section B. For the baseline unadapted grid system, Suggar++ produced 8581 orphans while SUGGAR produced 67905. Applying Suggar++ to the same grid system produced by the first adaption cycle resulted in 6 orphans in comparison to 6569 produced by SUGGAR. The adapted grid used with Suggar++ was the same grid that was used with SUGGAR and was produced with the output from SUGGAR.

Suggar++ also implements the domain connectivity API described in Section C. Using libSuggar++ in place of libSuggar is simply a matter of changing the set of libraries that the flow solver links with. Some small additional modifications to the flow solver will be required to take advantage of new capabilities within libSuggar++ such as parallel MPI execution.

The improvements in Suggar++ relative to the deficiencies in SUGGAR noted above are as follows:

1. Memory Requirements

The direct cut approach in Suggar++ does not require tunable parameters to adjust the hole cutting accuracy and hence the memory requirements will be fixed. For cases where accurate hole cuts are required, the direct cut in Suggar++ will cut a more accurate hole and use significantly less memory than SUGGAR.

2. User Inputs to Control the Hole Cutting

As mentioned in the previous section, Suggar++ does not require parameters to adjust the hole cutting so that the inputs for hole cutting will simply be the solid surfaces in the grid. Some grid formats include boundary condition specification so that this information would not have to be supplied by the user, resulting in a significant decrease in user input relative to SUGGAR. In addition, since the direct cut approach uses actual surfaces in the grid for hole cutting, the results are easier for users to understand.

3. Parallel Execution

Besides the new approach to hole cutting, the next major change for Suggar++ is the parallel execution. It can currently decompose a system of grids into subgrids, which can then be distributed to different MPI ranks and/or execute on different threads. Only lighter data is duplicated on all ranks to achieve good scalability in memory usage. Serial and parallel performance of the code is still being investigated and improved and will not be discussed in this paper.

IV. Conclusions

SUGGAR is a general overset grid assembly capability that is targeted at moving body simulations. This paper presented several significant enhancements that provide an improved capability. SUGGAR's capability for minimizing the overlap between component grids has been extended to unstructured grids resulting in improved overset composite grids and reduced user inputs. Results for the unstructured overlap minimization were demonstrated for a helicopter rotor blade grid system. An overlap grid adaption procedure was demonstrated that can be used to reduce or eliminate orphans due to insufficient overlap in unstructured grid systems. The effectiveness of the adaption process was demonstrated for an unstructured grid system for the Eglin Wing/Pylon/Store configuration. The best approach, which refined both the orphan elements and their cadidate donors, resulted in a rapid decrease in orphan count with only a modest increase in grid size. SUGGAR was also modified to allow it to be linked into the flow solver to provide an integrated overset grid assembly capability. The application programmer interface (API) was presented along with a discussion on how to modify a flow solver to utilize the library capability. Finally, the deficiencies in SUGGAR were discussed, and an improved capability provided by a new code called Suggar++ was briefly presented. These new capabilities in SUGGAR provide a significantly enhanced capability for overset moving body simulations such as store separations, and Suggar++ promises further improvements.

V. Acknowledgments

This material is based, in part, upon work supported by the National Aeronautics and Space Administration under Agreement No. NNX07AU75A issued through the Aeronautics Research Mission Directorate and the New 6DOF Environment project through the DoD HPC Institute for HPC Applications to Air Armament. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Aeronautics and Space Administration.

References

[1]Benek, J. A., Steger, J., and Dougherty, F., "A Flexible Grid Embedding Technique with Applications to the Euler Equations," Paper 83-1944, AIAA, 1983.

[2]Lijewski, L. E. and Suhs, N., "Chimera-Eagle Store Separation," Paper 92-4569, AIAA, 1992.

[3]R. L. Meakin, "Computations of the Unsteady Flow About a Generic Wing/Pylon/Finned-Store Configuration," Paper AIAA 92-4568, AIAA, AIAA Atmospheric Flight Mechanics Conference, 1992.

[4]R. L. Meakin, "Unsteady Simulation of the Viscous Flow About a V-22 Rotor and Wing in Hover," *AIAA Atmospheric Flight Mechanics Conf.*, 95-3463-CP, 1995, pp. 332–344.

[5]Belk, D. and Maple, R., "Automated Assembly of Structured Grids for Moving Body Problems," *Proceedings of 12th AIAA Computational Fluid Dynamics Conference*, AIAA Paper 95-1680-CP, San Diego, CA, 1995.

[6]Buning, P., "CFD Approaches for Simulation of Wing-Body Stage Separation," Paper 2004-4838, AIAA, 2004.

[7]Sickles, W., Power, G., Calahan, J., and Denny, A., "Application of a CFD Moving-Body System to Multicomponent Store Separation," Paper 2007-4073, AIAA, 25th AIAA Applied Aerodynamics Conference, Miami, FL, 2007.

[8]Tramel, R. W. and Nichols, R. H., "A Highly Efficient Numerical Method for Overset-Mesh Moving-Body Problems," Paper 97-2040, AIAA, 1997.

[9]Suhs, N. and Tramel, R., "PEGSUS 4.0 User's Manual," TR 91-8, AEDC, 1991.

[10]Power, G. and Calahan, J., "A Flexible System for the Analysis of Bodies in Relative Motion," Paper 2005-5120, AIAA, 17th AIAA Computational Fluid Dynamics Conference, Toronto, Ontario, 2005.

[11]Hooker, J. and Gudenkauf, J., "Application of the Unstructured Chimera Method for Rapid Weapon Trajectory Simulations," Paper 2007-75, AIAA, 45th AIAA Aerospace Sciences Meeting, Reno, Nevada, 2007.

[12]Pandya, M. J., Frink, N. T., and Chung, J. J., "Recent Enhancements to USM3D Unstructured Flow Solver for Unsteady Flows," Paper 2004-5201, AIAA, 2004.

[13]Pandya, M. J., Frink, N. T., and Noack, R. W., "Progress Toward Overset-Grid Moving Body Capability for USM3D Unstructured Flow Solver," Paper 2005-518, AIAA, 17th AIAA Computational Fluid Dynamics Conference, Toronto, Ontario, 2005.

[14]R. W. Noack, "SUGGAR: A General Capability for Moving Body Overset Grid Assembly," Paper 2005-5117, AIAA, 17th AIAA Computational Fluid Dynamic Conference, Toronto, Ontario, Canada, 2005.

[15]Power, G., Aboulmouna, M., Gudenkauf, J., and Masters, J., "Integration of USM3D Into the Store Separation Process: Current Status and Future Direction," Paper 2009-0339, AIAA, 47th AIAA Aerospace Sciences Meeting, Orlando, Florida, 2009.

[16]Biedron, R. and Lee-Rausch, E., "Rotor Airloads Prediction Using Unstructured Meshes and Loose CFD/CSD Coupling," Paper 2008-7341, AIAA, 26th AIAA Applied Aerodynamics Conference, Honolulu, Hawaii, 2008.

[17]Carrica, P., Wilson, R., Noack, R., and Stern, F., "Ship Motions Using Single-Phase Level Set with Dynamic Overset Grids," *Computers & Fluids*, Vol. 36, No. 9, 2007, pp. 1415–1433.

[18]R. W. Noack, "Resolution Appropriate Overset Grid Assembly for Structured and Unstructured Grids," Paper 2003-4123, AIAA, 16th AIAA Computational Fluid Dynamic Conference, Orlando, FL, 2003.

[19]Pao, S., Abdol-Hamid, K., Cambel, R., and Hunder, C., "Unstructured CFD and Noise Prediction Methods for Propulsion Airframe Aeroacoustics," Paper 2006-2597, AIAA, 2006.

[20]Power, G., Aboulmouna, M., Gudenkauf, J., and Masters, J., "Integration of USM3D Into the Store Separation Process: Current Status and Future Direction," Paper 2009-0339, AIAA, 2009.

[21]R. W. Noack, "DiRTlib: A Library to Add an Overset Capability to Your Flow Solver," Paper 2005-5116, AIAA, 17th AIAA Computational Fluid Dynamic Conference, Toronto, Ontario, Canada, 2005.

[22]R. W. Noack, "A Direct Cut Approach for Overset Hole Cutting," Paper 2007-3835, AIAA, 18th AIAA Computational Fluid Dynamic Conference, Miami, FL, 2007.

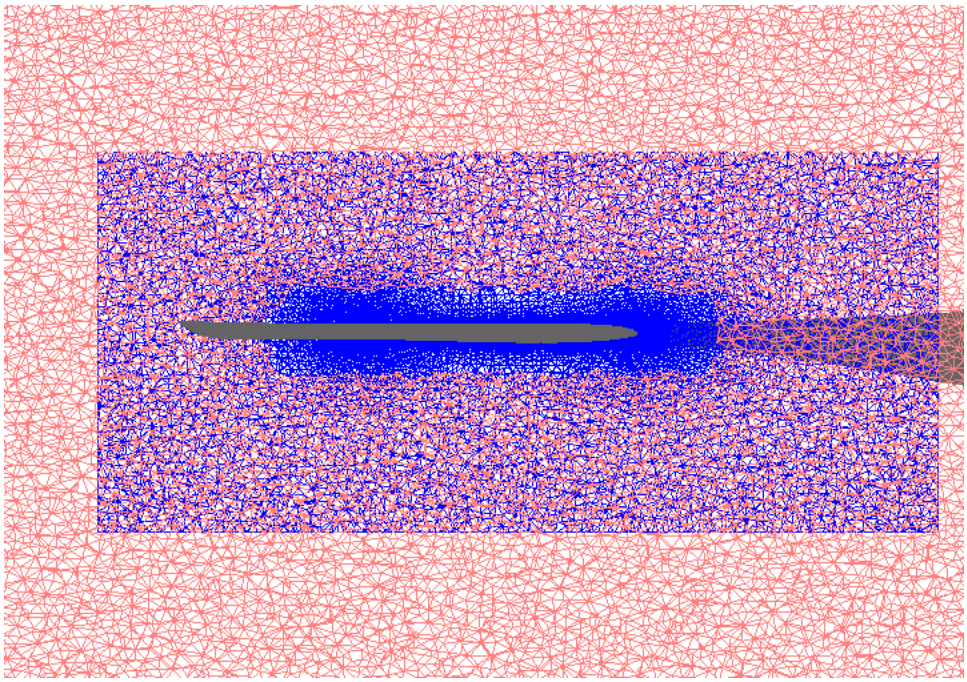

Figure 1. Overset grid system for rotor blade without overlap minimization

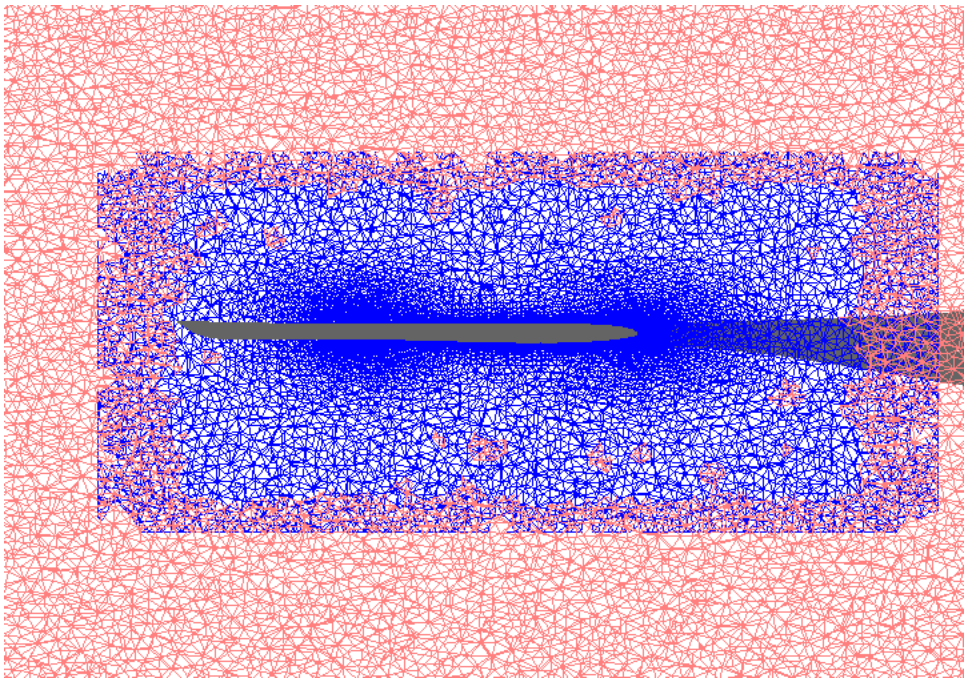

Figure 2. Overset grid system for rotor blade with overlap minimization

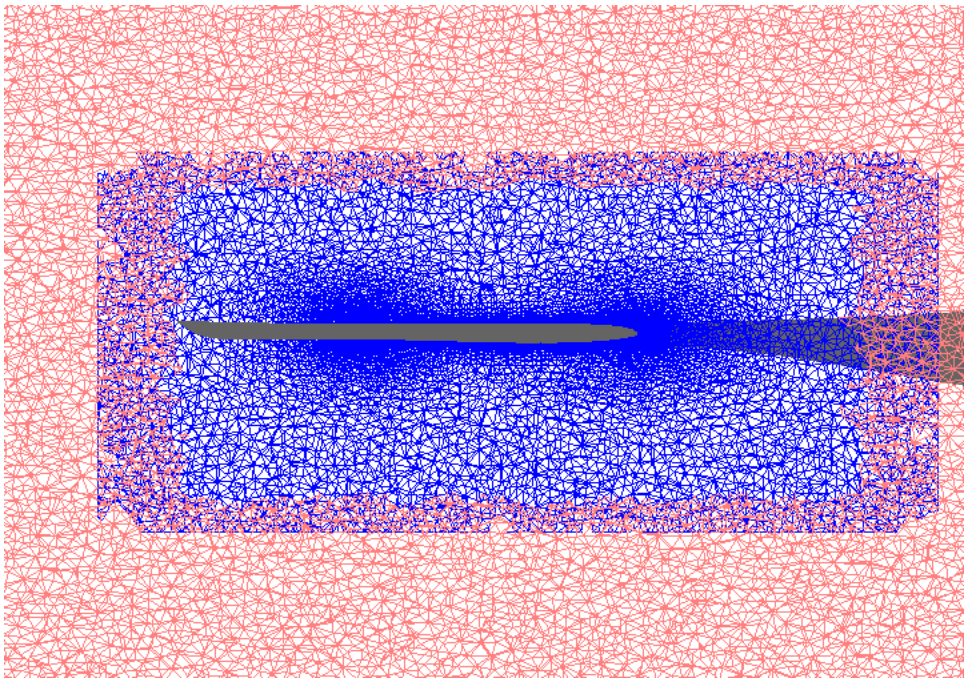

Figure 3. Overset grid system for rotor blade with overlap minimization and DSF set to 0 in blade grid

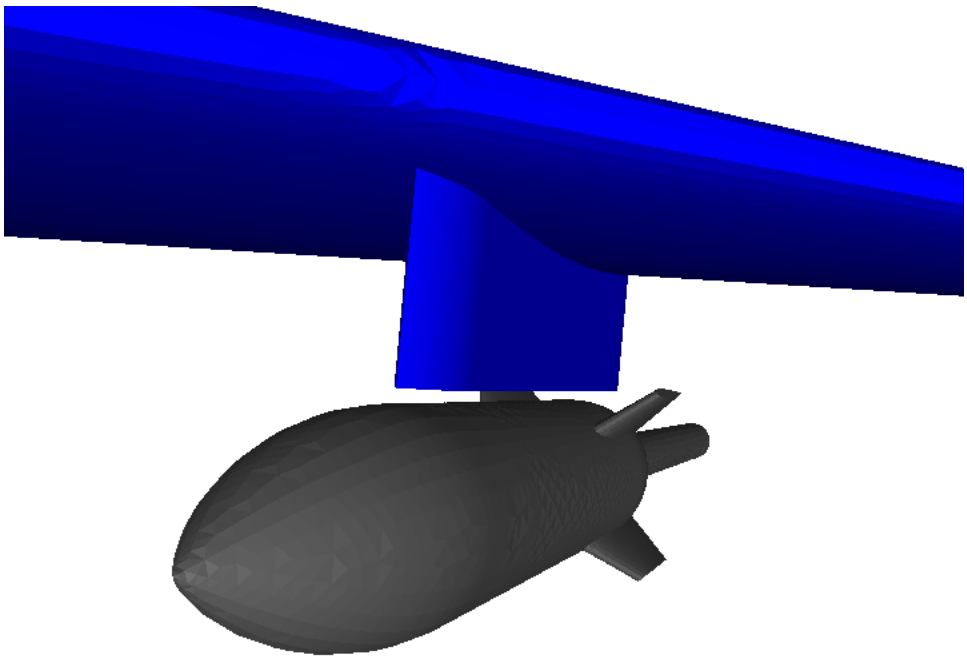

Figure 4. Eglin Wing/Pylon/Store geometry showing small gap between pylon and store

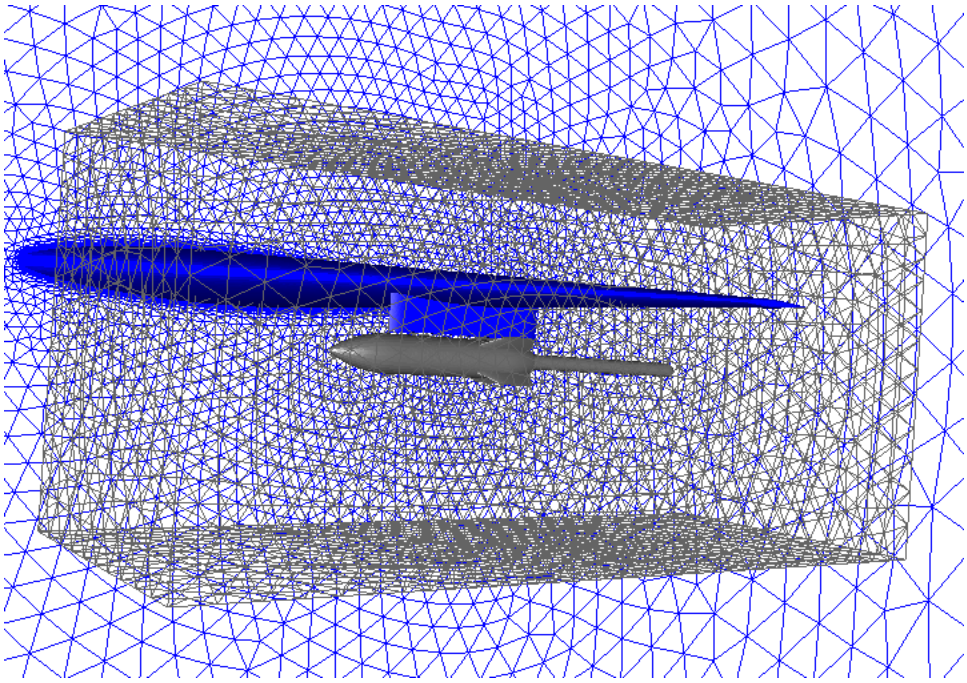

Figure 5. Overset grid system for Eglin Wing/Pylon/Store geometry

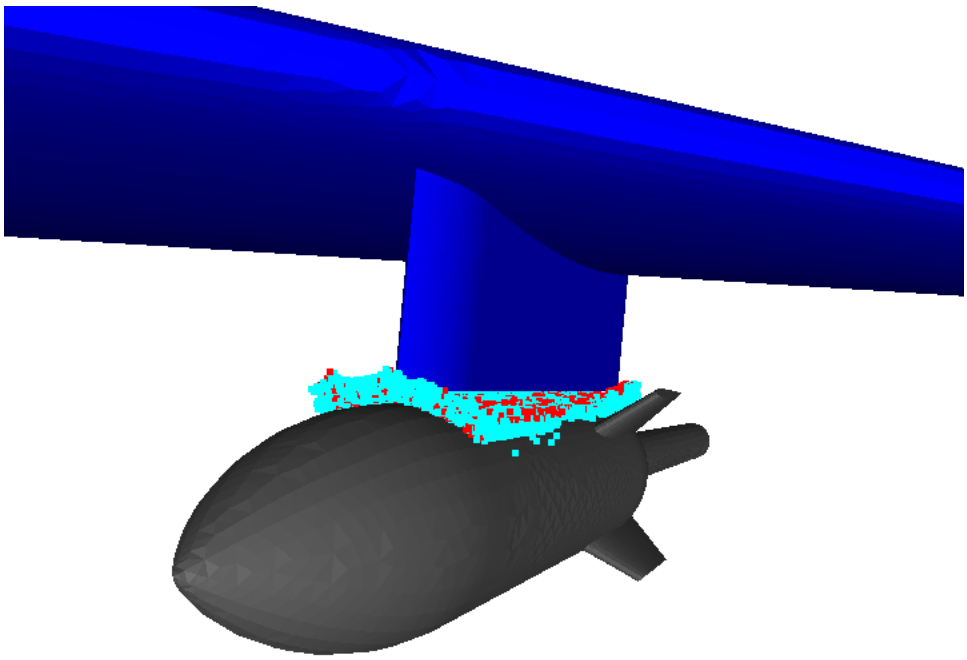

Figure 6. Eglin Wing/Pylon/Store geometry along with orphan locations

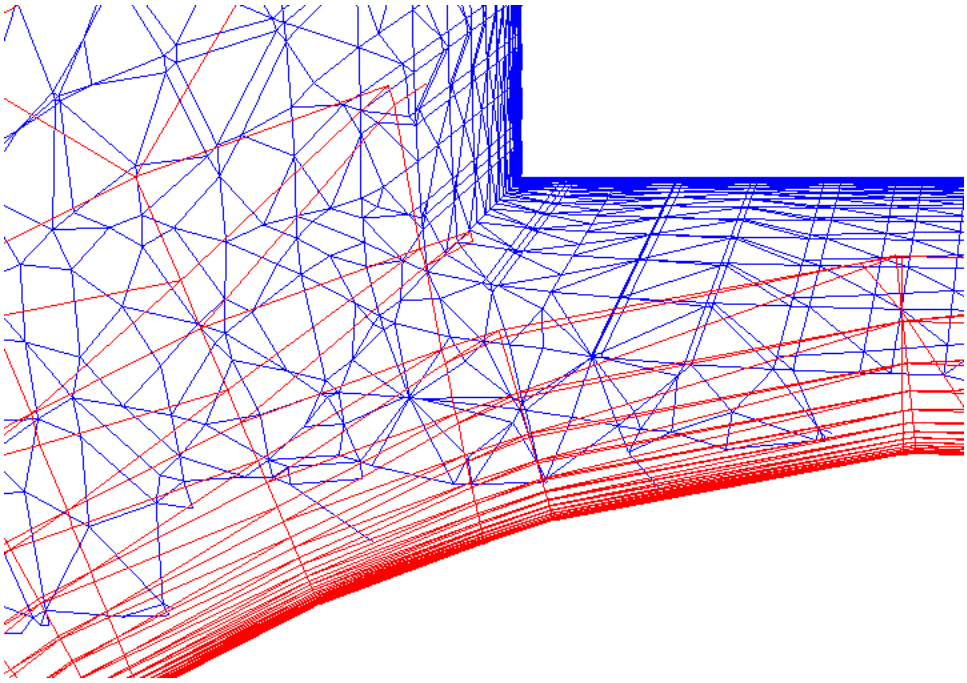

Figure 7. Cutplane through baseline overset grid system for Wing/Pylon/Store geometry

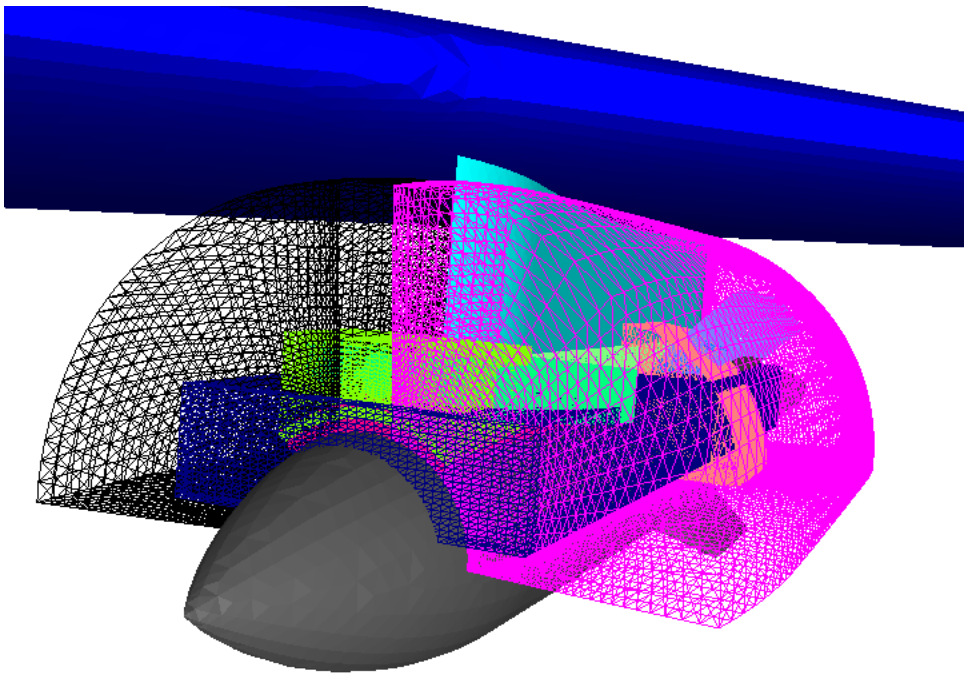

Figure 8. System of interface grid for Wing/Pylon/Store geometry

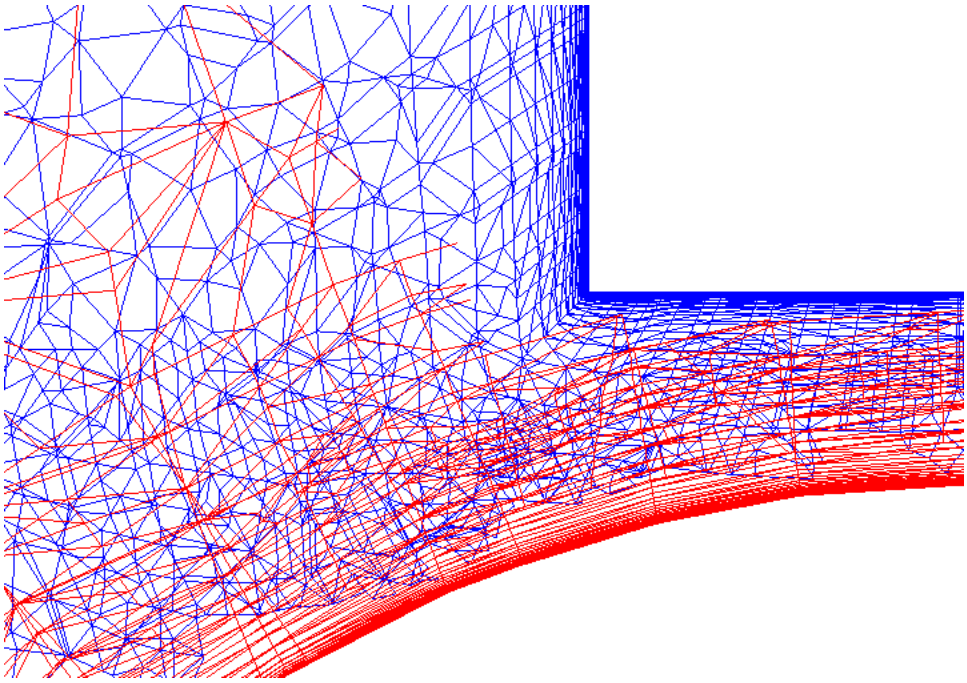

Figure 9. Cutplane through overset grid system after two adaption cycles for Wing/Pylon/Store geometry

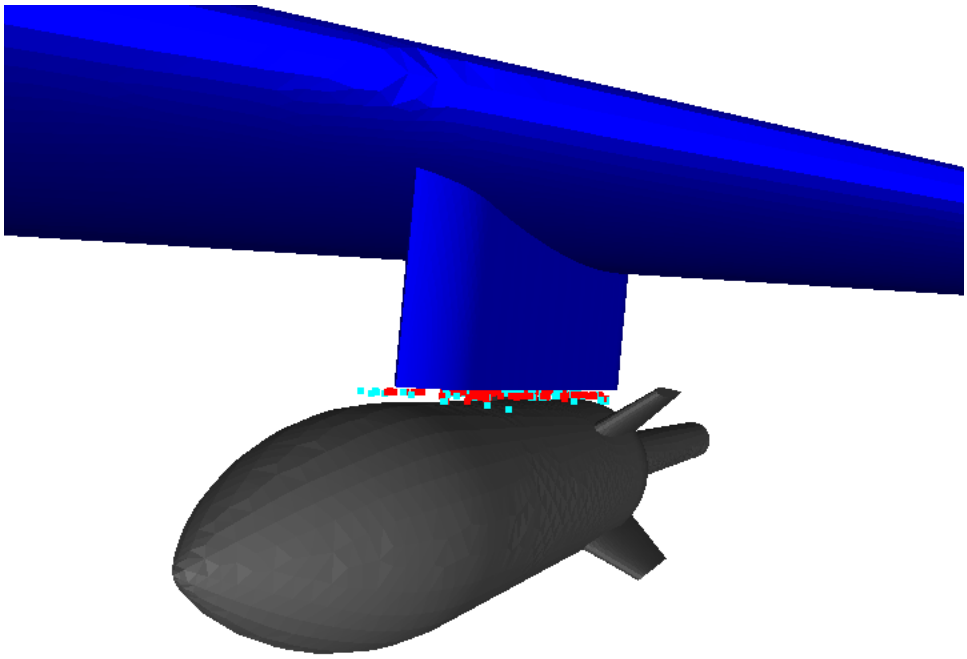

Figure 10. Eglin Wing/Pylon/Store geometry along with orphan locations after two adaption cycles

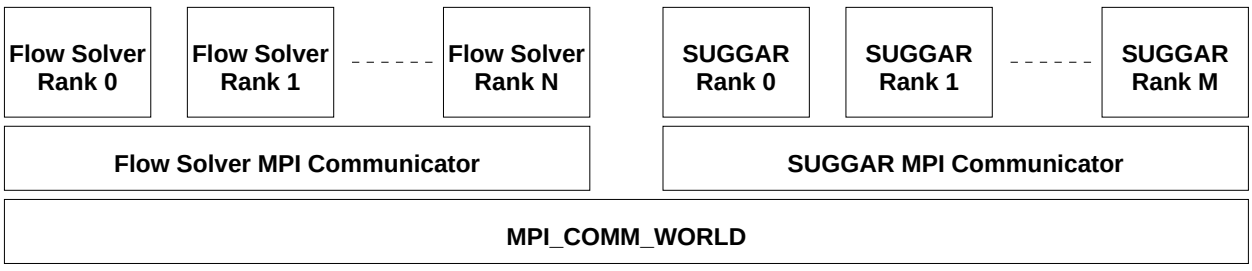

Figure 11. Flow Solver and SUGGAR processes and MPI Communicator Splitting

A. Example Use of DiRTlib and libSuggar in the PSU OVER-REL Code

This appendix describes in text and code snippets the use of DiRTlib and libSuggar in the PSU OVER-REL flow solver and is intended as a concrete example of the use of the libraries.

This solver executes the domain connectivity via libSuggar on a dedicated rank. The first step is to define which ranks will perform the flow solver work and which rank(s) will contain the DC work. Using more than one rank for libSuggar execution is not recommended but will be available with libSuggar++. Some initialization is required for both DiRTlib and libSuggar. The DCI update during the time step loop will require some changes to the calling sequence. Finally, libSuggar will be terminated just before the code execution completes.

A. Code Startup

OVER-REL begins by having all ranks initialize the MPI communication with the usual commands:

```
call mpi_init(ierr)
call mpi_comm_rank(mpi_comm_world,my_world_rank,ierr)
```

where *mpi_comm_world* represents the MPI world communicator that encompasses all of the available ranks.

1. Definition of the Ranks and Communicators

DiRTlib will need to split the MPI communicator between the ranks on which the flow solver will execute and the rank(s) on which the DC library will execute. The flow solver must also split the MPI communicator to obtain a communicator that is restricted to the flow ranks. The flow solver should use the split communicator instead of *MPI_COMM_WORLD* so that the flow ranks may communicate with one another without involving the libSuggar rank(s) and so flow solver collective operations encompass only the flow ranks. Typically the last M ranks will be dedicated to executing libSuggar. Currently, OVER-REL uses M=1.

The following Fortran code snippets demonstrate the assignment of the communication groups and responsibilities. Some optional logic is shown that might be useful for control depending on whether libSuggar is needed in a particular simulation and whether libSuggar is available (i.e. whether libSuggar has been linked to the solver). For Fortran codes, the DiRTlib Fortran interface module must be included to make the DiRTlib procedures available. Note that there are no calls yet to libSuggar during this phase.

```
use libdirt_interface

! if SUGGAR is needed and SUGGAR has been linked (and so is callable),
! then separate one dedicated processor for SUGGAR.

! get the total number of processors available (in mpi_comm_world):

call mpi_comm_size(mpi_comm_world,num_all_processors,ierr)

! use input to set the number of SUGGAR processors as one or zero and
! choose the rank on which to place the SUGGAR process (e.g. on
! the last rank):

num_suggar_processors = 1
suggar_rank = num_all_processors - 1

! calculate the number of processors that will contain flow
! computation processes for the flow solver
```

```
num_flow_processors = num_all_processors - num_suggar_processors

! ensure that every rank has access to num_flow_processors
! and suggar_rank.

! create separate communicators for the flow solver ranks
! and libSuggar rank. It is assumed at this point that libSuggar
! is linked to the flow solver and that a separate rank for
! SUGGAR is desired.

if (my_world_rank < num_flow_processors) then
! color for flow solver
  flow_node = .true.
  suggar_node = .false.
  color = 1
else
! color for SUGGAR
  suggar_node = .true.
  flow_node = .false.
  color = 0
end if

! next, split the communicator, creating "cfd_comm_world"
! subgroups, and establish a separate ranking system (my_comm_rank)
! within each sub-communicator.

call mpi_comm_split(mpi_comm_world, color, my_world_rank,
                    cfd_comm_world, ierr)
call mpi_comm_rank(cfd_comm_world,my_comm_rank,ierr)
```

B. Initialization Phase

Next, some initialization calls are required for DiRTlib (if this is an overset case) and for libSuggar (if it is linked and if it is required during the current simulation). Initializing both DiRTlib and libSuggar at the same time is required in order to allow them to perform an *mpi_comm_split* on *MPI_COMM_WORLD*. DiRTlib does this on the flow solver ranks, so the DC rank(s) must do it as well because it is a collective operation. If they both do it early on, then the flow ranks are freed to read the DCI file and begin initialization while the SUGGAR rank is freed to read the grid files, build the octree data structures, and conduct other initialization.

The following Fortran code snippets demonstrate the initialization phase. For Fortran codes, the DiRTlib Fortran interface module must be included to make the DiRTlib procedures available, while the libSuggar Fortran interface module must be included to make the libSuggar procedures available.

```
use libdirt_interface
use libSuggar_interface

#ifdef LINK_SUGGAR
if (suggar_node.and.flow_node) then
  call drtf_rank_flow_and_dci(ierr)
else if (suggar_node) then
```

```
  call drtf_rank_dci_only(ierr)
else if (flow_node) then
  call drtf_rank_flow_only(ierr)
else
  write(*,*) "ERROR! node is neither suggar_node or flow_node"
  stop
end if

! Everyone needs to be made aware of the rank number
! of the SUGGAR node:
call dcxf_set_dci_mrank_commwrld(suggar_rank,ierr)

#endif

! Everyone needs to call pll_init.  The arguments to this
! have been ignored for some time.
call drtf_pll_init(0,0)
call mpi_barrier(mpi_comm_world,ierr)
```

Setting the interpolation clipping is optional, but note that a flow-node-only barrier in that routine requires this to be called by only the flow nodes:

```
if (flow_node) then
  call drtf_clip_interp_to_min_max(interp_clip)
end if
```

Next, have every rank set the number of dependent variables:

```
num_data_values = 8
call drtf_set_ndata_val_all_grds(num_data_values)
allocate(mask(num_data_values))
mask = 1
```

The next snippet includes some optional steps: The SUGGAR rank changes directories, so that the SUGGAR input and output files may be separated from those of the flow solver. The SUGGAR rank then reopens the STDOUT and STDERR streams, so that the SUGGAR screen output may be separated from the flow solver screen output.

The required step in this section is the call to *dcf_init_no_args(input_file_name)*, which initializes libSuggar execution by first reading the specified SUGGAR input file and then computing the required internal data structures.

```
#ifdef LINK_SUGGAR
if (suggar_node) then
  ierr = chdir("SUGGAR")

  if (ierr /= 0) then
    write(*,*) "ERROR! chdir returned ierr = ",ierr
    stop
  end if
  call dcf_reopen_stdout("SUGGAR.stdout''//char(0))
  call dcf_reopen_stderr("SUGGAR.stderr"//char(0))
```

```
  dci_file_name = "../"//trim(dci_file_name)
  call dcf_init_no_args("Input/Input.xml"//char(0))

end if
#endif
```

Some codes may choose to have the flow solver ranks read a pre-existing DCI file at this point (for example, upon restart, to obtain the DCI information from the previous time step), although this is not generally needed:

```
if (flow_node) then
! must load the dci header before we can specify the decomposition
  call drtf_load_dci_file_header(dci_file_name)

! pass the flow solver paralle decomposition to DiRTlib
  call drtf_set_grid2rank_map(decomp_glb)

  call drtf_load_flex_grid_drt_file(dci_file_name)
end if
```

All nodes are required to initialize DiRTlib:

```
call drtf_Init(PutDataValue,GetDataValue,0,0,0)
```

If the DCI file was read, then it might be desirable to transfer the iblank values to a solver-defined array:

```
if (flow_node) then
  call drtf_set_iblank_for_frng_pnts( )
  call drtf_set_iblank_for_out_pnts( )
  call drtf_set_iblank_for_orph_pnts( )
endif
```

C. Time Step Loop

The initialization phase is now complete, and the time step loop is begun. The flow solver will generally communicate new instructions to libSuggar with each new time step. These instructions will normally include grid transformations and might also rename any desired output files.

The following Fortran code snippet demonstrates the calls that would be made for each time step. The flow solver ranks will generally need to wait for SUGGAR to complete prior to proceeding since they will need to obtain the new DCI information, so an mpi barrier that acts on *MPI_COMM_WORLD* is placed at the end of this section of code to provide the required synchronization.

The calls to begin, end, and parse the motion are all required, and all of the motion commands should be contained within (i.e. children of) a "<global>" element. Once the motion is specified and parsed, SUGGAR may be instructed to perform the assembly (i.e. compute the DCI information for the new configuration). In Fortran, `char(10)` represents a carriage return, which simply makes the log file that is written by SUGGAR more readable, while `char(0)` is required to end the string when it is passed from Fortran to C. For Fortran codes, the libSuggar Fortran interface module must be included to make the libSuggar procedures available.

```
use libSuggar_interface

if (suggar_node) then
```

```
#ifdef LINK_SUGGAR
  call dcf_begin_motion_input( )

  call dcf_add_motion_input("<global>"//char(10)//char(0))
! [...add more calls to communicate the grid transforms, etc.]
  call dcf_add_motion_input("</global>"//char(10)//char(0))

  call dcf_end_motion_input( )
  call dcf_parse_motion( )
  call dcf_compute_dci(0) ! 0 suppresses, 1 enables writing the DCI file
#endif

end if
call mpi_barrier(mpi_comm_world,ierr)
```

At this point, the new DCI information has been calculated, but the information is only known by libSuggar (and/or has been written to the disk in the form of a DCI file). The information needs to be transferred to DiRTlib, which can then provide it to the solver as needed. The DCI output file may be written from the SUGGAR rank at any time after the DCI information has been calculated and before the DCI information is released from memory. The DCI information must be released from memory before the end of each time step to avoid memory leaks.

```
use libdirt_interface,only: drtf_get_dci,drtf_get_dci_header
#ifdef LINK_SUGGAR
use libSuggar_interface,only: dcf_release_dci,dcf_write_dci_file
#endif

if (first_time) then
! only need to call the get header function one time
  call drtf_get_dci_header( )
  first_time = .false.
end if

! transfer the DCI from libSuggar to DiRTlib
call drtf_get_dci( )

if (suggar_node) then
#ifdef LINK_SUGGAR
! optional: write the DCI file for use in a restart
  call dcf_write_dci_file(out_dci_filename,ierr)

! required: release the DCI now that DiRTlib has it
  call dcf_release_dci( )
#endif
end if

! Final initialization call:
! Pass solver interface functions to DiRTlib
! Note that this must be called AFTER the DCI information
! is obtained and MUST be called EVERY time new DCI information
```

```
! is obtained
if (flow_node) then
  call drtf_Init(PutDataValue,GetDataValue,0,0,0)
  ! transfer the iblank information to solver-defined iblank arrays
  ! if desired
end if
```

D. Termination Phase

Once the time step loop has completed, the SUGGAR rank may terminate the SUGGAR process as follows:

```
if (suggar_node) then
#ifdef LINK_SUGGAR
  call dcf_terminate(ierr)
#endif
end if
```

Finally, all ranks may terminate the MPI processes:

```
call mpi_finalize(ierr)
```