Enhancing Scalability for FUN3D Rotorcraft Simulations with Yoga: an Overset Grid Assembler

Cameron T. Druyor,
NASA Langley Research Center
Hampton, VA 23681, USA

FUN3D, an unstructured grid Navier-Stokes CFD code is capable of overset grid simulations, but does not have an internal method for assembling an overset grid system from a group of component grids. FUN3D currently relies on the third party codes Suggar++ and DiRTlib to perform domain assembly and provide intergrid connectivity for overset simulations. Rotorcraft simulations with moving, deforming blades require domain assembly and mesh deformation at each time step. For these simulations, the three primary drivers of computational cost for each time step are: deforming the mesh, performing domain assembly, and performing subiterations of the flow solver. FUN3D exhibits strong and weak scalability for the flow solver subiterations and the mesh deformation. However, FUN3D is currently hardwired directly to the serial version of Suggar++, which has a fixed cost for a given mesh system. Therefore, domain assembly begins to dominate the total cost of each time step as grid systems become larger. An integrated method for parallel domain assembly is presented that addresses scalability for large grid systems. Restructuring within FUN3D to accommodate integrated domain assembly is also discussed, which could enable use of the parallel Suggar++ library.

I. Introduction

The overset, or chimera [1], grid method enables dynamic CFD simulations with multiple bodies in relative motion by allowing the domain to be spanned by multiple overlapping, but disjoint, component grids. The individual components are free to move relative to one another, provided that their union spans the computational domain. Additionally, some work is required to transfer solution information between grids and achieve a consistent solution across the composite domain. Overset domain assembly is the process of determining appropriate locations for intergrid boundaries and dividing component grids into regions of active, inactive, and receptor points. These are also commonly referred to as in, out, and fringe, respectively. Overset domain assemblers can use various geometric quantities to determine appropriate locations for intergrid boundaries such as relative cell size and wall distance. For points in active regions, the CFD solver can solve for the solution using the same method it uses in a nonoverset context. Inactive regions are not part of the solution space, so the CFD solver can ignore points in these regions entirely. Finally, receptor regions are where the solution from an active region of another component grid is injected into the current grid. Defining these boundaries requires information about how each component grid overlaps with the others. Specifically, for each receptor, a list of candidate donor elements from other components must be identified via a parallel geometric search. The information generated during domain assembly is used to to define relationships and describe connectivity between component grids. Specifically, donor points are identified for each receptor, such that the solution at the donor points can be interpolated to the receptor. A comprehensive survey of overset methods and history can be found in Ref. [2].

FUN3D has existing overset capability [3] and uses Suggar++ [4] and Dirtlib [5] to perform overset domain assembly and provide connectivity information, respectively. Grid resolutions required to resolve the relevant physics of current engineering problems lead to large grid systems containing hundreds of millions or billions of points and the simulations must be solved on thousands or tens of thousands of CPU cores. CFD codes, such as FUN3D, have adapted to operate efficiently at this scale, but overset domain assembly can become the dominant cost factor if it does not exhibit the same scaling behavior as the fluid solver. This paper discusses the scalability challenges presented by parallel domain assembly, and how these challenges influenced Yoga's design.

II. Yoga: is an Overset Grid Assembler

Yoga (first described in Ref. [6]), is designed to meet multiple criteria concerning repeatability and performance. FUN3D and Yoga should both use the same MPI ranks during a simulation such that no processes are idle during the flow solve, and no processes are idle during the domain assembly. Therefore, Yoga should minimize memory overhead as much as possible so that the solver and assembly data fit on each MPI rank, e.g., Yoga must not store a global image of the grid on a single process. Yoga should also give the same result, regardless of the partitioning of the flow solver (i.e., the assembly should be invariant to partitioning). Yoga should perform well at various scales in terms of both grid sizes and number of MPI ranks. Specifically, the cost of assembly should ideally be less than the cost of the flow solver subiterations for each time step, the cost should scale linearly with the number of grid points per MPI rank, and the cost should not be sensitive to the initial partitioning of the flow solver.

A. Cost Uncertainty

Parallel fluid solvers typically need to communicate very little information at partition boundaries, and the communication pattern for those boundaries is often known at the beginning of the simulation and remains fixed throughout. Partitions are usually constructed such that each has a similar computational cost, which is possible because the cost can be effectively estimated a priori based on the number of nodes and elements. Therefore, achieving parallel scalability in the fluid solver is relatively straightforward. Whereas nodes and elements each contribute a consistent amount to the cost of a fluid solver across partitions, their contribution to the cost of domain assembly can vary significantly due to its highly geometric nature. Nodes and elements outside of regions where multiple grids overlap have effectively zero cost, while the cost of nodes and elements in overlap regions depends on grid density and choice of geometric search algorithm. Furthermore, for moving grids, the overlap regions (and the elements contained in those regions) will change with each time step. The initial partitioning of the fluid solver can be extremely poor for overset domain assembly in terms of total work distribution.

1. Motivation for Overdecomposition and Dynamic Load Balancing

Overdecomposition and dynamic load balancing are the two primary tactics Yoga uses in its parallel strategy to mitigate cost uncertainty. In this section, a simple Monte Carlo simulation is used to empirically demonstrate the effectiveness of both tactics. For the simulation, a prescribed total amount of work is divided into work units and distributed to processors. The "work" in this case is simply to sleep for a specified amount of time. Figure 1 shows measurements from Monte Carlo simulations with different levels of overdecomposition and with different amounts of uncertainty in the cost of each work unit. Overhead, plotted on the y-axis, is the ratio of the actual recorded wall clock cost to the theoretical wall clock cost if the problem had been perfectly load balanced. Uncertainty in predicted cost is modeled by sampling a normal distribution with mean equal to the target average cost, and varying the standard deviation. On the left side of the plot, the standard deviation is set to zero, and on the right side, the standard deviation is 200 percent of the mean. This provides a way to demonstrate the relationship between error in predicted cost, and overhead incurred. For each level of overdecomposition, both the average and worst case measurements are plotted with solid and dashed lines respectively.

Figure 1 shows how overhead incurred by predictive error increases non-linearly with the size of the predictive error and demonstrates how overdecomposition can reduce the variance in cost, even with the same amount of predicted error. When the work is divided into 100 units per process, only a small amount of overhead is incurred, even when the predicted cost per work unit is wrong by a factor of 2. This property makes overdecomposition a powerful tool for mitigating uncertainty in predicted costs. Whereas the Monte Carlo simulation does not account for overhead associated with overdecomposing the work, real-world problems, such as overset domain assembly are not amenable to such extreme overdecomposition. Therefore, it is practical to combine moderate overdecomposition with dynamic load balancing to increase its effectiveness. Figure 2 shows that a similar level of performance can be achieved with ten times less work units per process if the work units are dynamically load balanced. For these simulations, dynamic load balancing is simulated by having each process request a new work unit once it completes the previous one.

B. Task Based Parallelism

Yoga employs task based parallelism to hide communication costs by overlapping communication and computation. Yoga's task system is based on remote procedure calls. The task system runs on its own thread, which allows communication to be independent of the work being done on the main thread. This allows communication to be

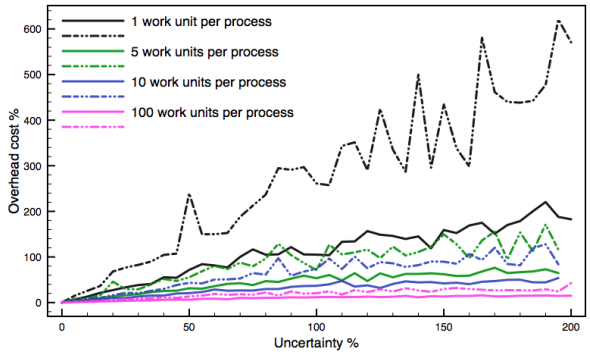


Fig. 1 Monte-Carlo simulation of parallel performance with varying degrees of uncertainty in work unit cost, and different amounts of overdecomposition.

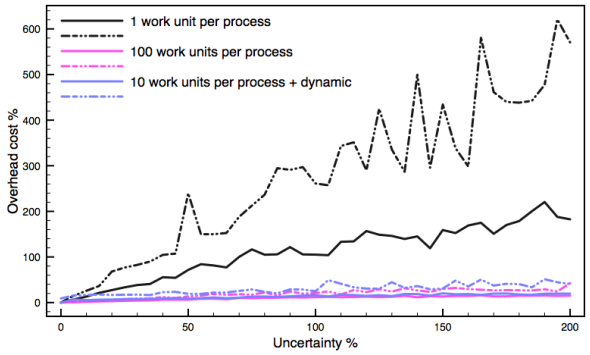


Fig. 2 Comparison of performance between extreme overdecomposition with moderate overdecomposition and dynamic load balancing.

fully asynchronous, i.e., an incoming request can trigger a callback without waiting for the current operation on the main thread to complete. This approach relies on the operating system to switch out the oversubscribed threads appropriately. Though there is overhead associated with thread oversubscription, it is small relative to the latency incurred by serializing requests.

Upon construction, the task system takes a set of integer identifiers for all task types that can be created while the task system is running. The task system has the following minimal, but extremely flexible member functions, upon which all of Yoga's asynchronous communication patterns are built.

```
void registerCallBack(int message_type,
        std::function<void(MessagePasser::Message&&)> callback);
void start();
void push(int target_rank, int message_type,
        MessagePasser::Message&& msg);
void push(int target_rank, int message_type,
        std::shared_ptr<MessagePasser::Message> msg);
void stop();
```

The task system requires that a callback is registered for each task before start is called. The task system runs on its own thread, which is launched when start is called. Each callback must take a serialized message as its input and return nothing. All communication performed via the task system is one sided and asynchronous. A task consists of a target rank, task type identifier, and a serialized message. The *Message* class contains a contiguous vector of bytes, and has templated *pack* and *unpack* methods that simplify data marshaling for complex data structures. By dealing only in terms of *Message*, the task system remains agnostic to datatype. The communication is performed via ZeroMQ [7] within the task system, but processors are indexed by their MPI rank for consistency. The two methods for creating tasks differ only by *Message* ownership. The first method takes an rvalue reference [8] to a message (i.e., a sink parameter) which gives the task system the responsibility for freeing the *Message* contents after the task is completed. The second method takes a shared pointer, which allows the caller to manage the memory. The task system needs to be lightweight to avoid stealing resources from the main thread performing the actual work, so its duties are strictly limited to receiving messages, invoking callbacks, and sending messages.

Yoga's task system implements asynchronous communication via a reverse push-pull pattern with ZeroMQ. The push-pull pattern typically defines a one-to-many relationship between a producer and its consumers. The producer (i.e., server) has a *PUSH* socket that is bound to a predefined TCP port, and the consumers (i.e., clients) each have a *PULL* socket that connects to the *PUSH* socket of the producer. When messages are pushed by the producer, ZeroMQ load balances them across the consumers. In Yoga, the client server relationship is reversed. Each instance of the task system defines a server by binding a *PULL* socket to a predefined TCP port, and defines a client by creating a *PUSH* socket that can be connected to any server. This pattern creates a one-to-many client-server relationship in which each client can arbitrarily push messages to any server on any rank (including itself). Figure 3 shows a snapshot of tasks

being communicated between instances of the task system on four ranks. Tasks are represented by a message and an integer for the target rank. Any rank can push tasks to any other rank and an arbitrary number of tasks can be pushed to a given rank simultaneously (i.e., multiple messages to a particular rank can be in flight at the same time).

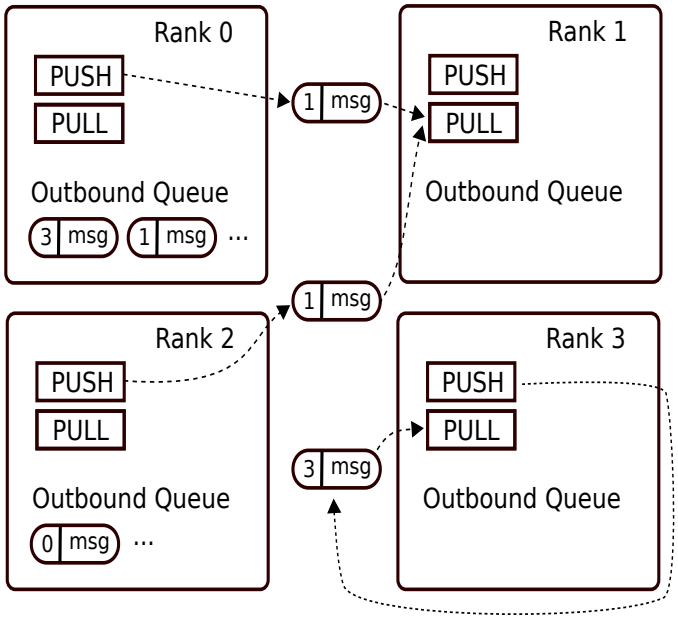


Fig. 3 A snapshot of Yoga's task system is shown with multiple messages in flight. Tasks are depicted as integer and message pairs, where the integer is the target rank for the task. Each rank has an instance of the task system, and each instance has a *PUSH* socket for sending messages, a *PULL* socket for receiving messages, and a queue of messages that need to be sent.

While running, the task system alternates between sending and receiving messages. If the outbound queue is not empty, the task system sends the first message and pops it from the queue. Then the task system checks for incoming messages on the *PULL* socket. If the socket has a message, the task system pulls the message, unpacks it, and executes the appropriate callback. Since the task system cannot send or receive messages while executing a callback, it is desirable to have only have lightweight or very high priority operations. For example, the callback may simply store the incoming message for later retrieval by the main working thread. Simple storage callbacks like this can be implemented by leveraging Yoga's thread-safe *MailBox* class. *MailBox* has the following member functions, which enable thread-safe storage, retrieval, and synchronization of messages:

```
void store(int message_type,MessagePasser::Message&& m);
std::shared_ptr<MessagePasser::Message> wait(int message_type);
std::shared_ptr<MessagePasser::Message> retrieve(int inbox_id);
bool isThereAMessage(int message_type) const;
void lock();
void unlock();
```

The store method simply copies the message and returns immediately, freeing up the task system to resume sending and receiving messages. For example, a callback that stores messages of type N can be captured as a lambda closure that leverages the *store* method of *MailBox* as follows:

```
auto callback = [&](MessagePasser::Message&& m){
        mailbox.store(N,std::move(m));}
```

Where N is the integer that identifies the message type. In this way, *MailBox* provides a template for storing different message types (i.e., different kinds of tasks) that each require separate inboxes while maintaining thread safety. Once a callback has been defined, it can in turn be registered with the task system as:

```
task_system.registerCallback(N,callback);
```

Leveraging closures in this way yields a high level of flexibility and composability when designing, testing, and making large changes to communication patterns.

C. Yoga High Level Algorithm

Yoga has three phases: preprocessing, geometric search, and postprocessing, where the geometric search phase represents the greatest computational cost and is therefore the target of the dynamic load balancing strategy. Note that, while Yoga repartitions the grid dynamically within each time step, the partitioning of the flow solver remains intact. Yoga propagates all domain assembly information back to the flow solver's original partitions before returning control to the solver.

1. Preprocessing

During the preprocessing phase, Yoga generates metadata for the grid system to help accelerate the parallel geometric search. Yoga creates point clouds for the surface nodes of all bodies in the grid system, and saves local copies on each MPI rank. Yoga then constructs nearest neighbor trees for each body that will later be used to generate approximate wall distance for points in overlap regions. Yoga also uses the surface point clouds to create hole maps [9] to serve as approximate representations of each body. The hole maps can be used to efficiently identify points that might lie inside bodies (false positives are filtered out during the donor search). Yoga makes heavy use of AABBs (Axis Aligned Bounding Boxes) for their small storage size and fast intersection detection. First, Yoga generates AABBs for each solid body based on the boundary conditions from the solver, and stores them on every MPI rank. These provide an approximate view of the relative spatial locality of each solid surface in the grid system on every rank. Hole maps are then constructed by first creating a structured auxiliary grid for each body, as shown in Figure 4. Then, the cells of the structured auxiliary grid which intersect or are fully contained by the body are marked in a bit-mask, as shown in Figure 5. Using the same procedure, Yoga generates AABBS for each partition, along with structured auxiliary grids

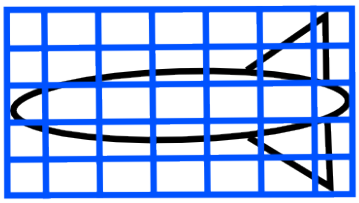


Fig. 4 Given an AABB, a structured auxiliary grid defines an implied discretization, which can be used for constant time spatial lookups.

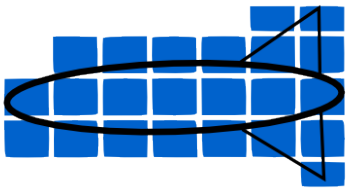


Fig. 5 A hole map is defined by marking cells in a structured auxiliary grid that intersect or are contained by the geometry.

with bit-masks to signify if the partition contains any nodes or cells that cross each cell of the auxiliary grid. These bit masks provide a representation of the shape of each partition within its AABB, without adding much storage overhead. The AABBs and bit masks are small enough that they can be stored on every MPI rank, which gives every rank the ability to make a local query to determine which MPI ranks may own grid information in any particular region of space. Yoga also decomposes each partition into small grid fragments via Recursive Coordinate Bisection [10], and stores AABBs for each fragment. These grid fragments and their AABBs facilitate low latency remote grid retrieval during the geometric search phase.

The most critical part of the preprocessing phase is the construction of the load balancer, which will be responsible for efficiently distributing the cost of the geometric search across all MPI ranks. In Yoga, a work unit is an AABB that describes a region of space that potentially contains grid points from multiple component grids. The grid points in a particular region of space may live on different MPI ranks, so communication will be required to complete each work unit, but it can be simply described by an AABB for the purposes of load balancing. When constructing the load

balancer, Yoga must generate a set of AABB work units that completely spans the space of all overlap regions and estimate the cost of each work unit.

Yoga uses mesh density as an approximation of geometric searching cost, because the number of nodes and cells in a given region of space is correlated to both the number of searches required and the cost of each search. However, determining mesh density exactly for each work unit could incur large communication costs because nodes and cells in the AABB could reside on an arbitrary set of MPI ranks. Therefore Yoga creates a surrogate model for mesh density that can be stored on the root, and allow all work units to be generated without further communication. The model is constructed as a Cartesian image that contains all regions where multiple grids overlap, and each volumetric pixel of the image contains the total number of cells that intersect it. The image is generated by first creating an AABB that contains all overlap regions, counting overlapping cells for each voxel, and performing a single parallel sum over the counts.

Yoga uses the surrogate density model to estimate costs while creating work units. Yoga first identifies all non-empty voxels in the image and extracts them as AABBs. These AABBs of the extracted voxels cannot be used directly as work units due to the wide variation in overlap counts (and hence costs). Many of the extracted voxels will not contain enough work to outweigh the overhead of dynamic load balancing. Furthermore, some voxels may have extremely high costs that would cause a small number of work units to be responsible for a large amount of the overall geometric search cost. Yoga handles both issues separately. First, Yoga normalizes the overlap counts for every voxel, then agglomerates low cost voxels via weighted Recursive Coordinate Bisection, using the voxel centers as points. Then Yoga subdivides high cost voxels into smaller AABBs that are estimated to be under the target cost. After agglomerating low cost voxels and subdividing high cost voxels, Yoga has a set of AABBs that span the space of all overlap regions and estimates of their costs. Yoga then stores these into a priority queue based on their estimated cost, so that the work units that are expected to be the most expensive will be completed first, less expensive work units can fill in the gaps later.

2. Geometric Search

Yoga uses several asynchronous client-server communication patterns built on top of its task system to interleave communication and computation during the geometric search phase. The first client-server pattern arises from the distribution of work units that were generated during the preprocessing phase. Each MPI rank (i.e., client) must query a work server to get work units. Since work units are simply AABBs, they could be stored on a single server rank. However, this can cause latency issues for simulations with thousands of MPI ranks due to the large number of simultaneous requests. Therefore Yoga distributes the load across multiple work servers. By default, Yoga creates one work server for every 50 MPI ranks, and work units are distributed round-robin from the full priority queue (each work server in turn stores its subset of work units in a priority queue). Yoga's task system uses one-sided communication, so the client-server relationship is broken into two pieces. On the client side, a request to a particular work server is pushed into the task system:

```
MessagePasser::Message work_request;
work_request.pack(my_rank);
task_system.push(rank_of_my_load_balancer, MessageTypes::WorkRequest,
                 std::move(work_request));
```

Where the request message simply contains the rank of the requesting process, so that the reply can be routed appropriately. Then the client can wait on the reply to appear in the corresponding $MailBox$.

```
next_work_unit_ptr = mailbox.wait(MessageTypes::WorkExtentBox);
```

Note that the task system continues to field requests while the client thread is waiting on a reply to arrive. On the server side, the subset of ranks that have work units to distribute must register a callback that enables the task system to handle incoming work requests. The callback is succinctly defined as the following closure:

```
auto load_balance_callback = [&](MessagePasser::Message&& msg){
        MessagePasser::Message reply;
    int sending_rank = 0;
    msg.unpack(sending_rank);
    int n_remaining = distributed_load_balancer.getRemainingVoxelCount();
    reply.pack(n_remaining);
    reply.pack(distributed_load_balancer.getWorkVoxel());
```

```
    task_system.push(sending_rank,MessageTypes::WorkExtentBox,
                     std::move(reply));
};
```

Where *distributed_load_balancer* is an instance of the class that contains the priority queue of work units discussed earlier, and *task_system* is an instance of Yoga's asynchronous task system, both of which are captured implicitly by reference in the lambda signature. The incoming *Message* for this type of request only contains a single integer that indicates which rank sent the request. The work server uses this rank when pushing the reply back into the task system. Two pieces of information are needed to fulfill a work request. First, the number of work units left in the resident priority queue, and second, the AABB that defines the current work unit. Both pieces of information are packed into a reply *Message*, and pushed into the task system tagged as a *WorkExtentBox* message type and target rank. Then the closure returns, and the task system resumes.

The second client-server pattern involves gathering the necessary nodes and cells to complete a work unit. Each rank that owns nodes and cells that overlap with another component grid has a grid server that responds to spatial requests. The grid server is implemented in Yoga's task system as the following closure:

```
auto grid_request_callback = [&](MessagePasser::Message&& msg){
   int sending_rank = 0;
   msg.unpack(sending_rank);
   auto reply = grid_fetcher.doWork(msg);
   task_system.push(sending_rank,
                    MessageTypes::GridFragment,
                    std::move(reply));
};
```

Similarly to the load balancing callback, the grid request callback first unpacks the sending rank from the incoming message in order to send the reply back to the appropriate rank. In addition to the sending rank, a grid request message also contains an AABB, which is forwarded on to the *grid_fetcher* instance contains the grid fragments of the overdecomposed partition. When given the message containing an AABB, the *grid_fetcher* will identify all fragments that intersect the AABB and pack the fragments into a message. The returned message is then pushed into the task system, to be delivered back to the requesting rank.

On the client side, after a worker receives a work unit, the worker identifies which ranks may own nodes or cells that overlap with the work unit, using the partition AABBs and bit-masks that were created in the preprocessing phase. Then the worker sends a grid request to the grid server on each of those ranks. After pushing all of its grid requests into the task system, the worker waits for replies to arrive. The communication is asynchronous, and replies come in arbitrary order. As each reply arrives, the worker unpacks the grid information from the serialized message and adds it to a temporary local grid. Once all grid requests have been fulfilled, the worker has enough information to complete the geometric search. The worker then creates an ADT (Alternating Digital Tree [11]) for each component grid in the work-unit. The worker then finds all possible donor cells for all nodes in the work unit by searching the ADT of each component grid to which the node does not belong. In addition to performing the donor search, the worker also generates node connectivity graphs and an approximate distance to the wall for each node in the work-unit. This information will be needed during the type assignment phase, and by performing this work on a per work unit basis, the cost of node connectivity and distance field generation is load balanced along with the donor search. Finally, the worker packages all the generated donor information and connectivity for each node into a serialized message, and sends it to the owning rank (of each node). Communication of donor information is implemented via the same pattern discussed for earlier operations with one exception. The callback registered for storing a *DciUpdate* also sends a receipt back to the rank that sent the update:

```
task_system.registerCallBack(DciUpdate,[&](MessagePasser::Message&& m){
        int sending_rank = 0;
        m.unpack(sending_rank);
        mailbox.store(DciUpdate,std::move(m));
        post_man.push(sending_rank,DciReceipt,MessagePasser::Message());
});
```

This receipt ensures that all updates have been properly stored before shutting down the task systems on every rank. When the main working thread finishes all of its work and has received a receipt for each *DciUpdate* that it has sent, it

notifies the root process that it is finished, but leaves its task system running to process further requests. The notification to the root occurs hierarchically to avoid many ranks pushing requests to the root process simultaneously. Groups of MPI ranks send their notifications to a local collecting rank, which only notifies the root process after it has received notifications from all of its subordinates. When all processes are accounted for, the root process notifies all processes (via the same hierarchy) that their task systems can safely shut down. The need for such a shutdown procedure is required because a rank cannot predict a priori how many requests it will need to handle.

3. Postprocessing

During the postprocessing phase, Yoga determines appropriate intergrid boundary locations, and assigns one of the following statuses to each node in the grid system: In, Out, Receptor, or Orphan. The status of some points can be set directly, such as hole-points that lie within solid surfaces, which can be identified and set to Out. Similarly, surface nodes, and nodes whose distance is smaller than that of their donor candidates are all set to In. For more complex relationships, Yoga uses several intermediate statuses that eventually get turned into one of the four final statuses. For example, Yoga identifies "mandatory" receptors, which are ineligible to be donors. These include nodes on surfaces with interpolation boundary conditions, and potentially several layers of their neighbors, depending on the number of interpolation layers needed. For each mandatory receptor, Yoga identifies its donors and attempts to convert them to In points if possible. This modification reduces or eliminates Orphans when many component grids overlap in the same volume by increasing the availability of valid donors. Yoga sets mandatory receptors to Out if any of their donors have a smaller distance value, otherwise Yoga sets them to Receptor if they have valid donors. Yoga then identifies points with unknown status that have neighbors marked as In, and marks them as candidate receptors. Candidate receptors that have valid donors are converted to Receptors, otherwise they are converted to Orphans. Finally, Yoga sets all remaining points to Out.

III. Results

A. Verification and Validation

A 4 blade UH60 test case (18 million nodes) is used for code-on-code verification between Yoga and Suggar++. Overset domain assembly results and flow solutions are compared. Figures 6 and 7 show x and y slices of the mesh system as generated by Suggar++ and Yoga, respectively (i.e., a slice down the centerline of the fuselage, and another perpendicular slice through the rotor hub). Intergrid boundaries are made visible by removing Out points from each component and coloring the grid by component id. The assembled grid systems differ, due to differences in Yoga's modified distance criteria and the overlap-minimization option in Suggar++. However, the differences do not greatly affect the resulting flow solution.

Figures 8 and 9 show x and y slices of vorticity magnitude to demonstrate the qualitative similarity of large scale flow features between FUN3D plus Suggar++ and FUN3D plus Yoga, respectively. FUN3D generates similar large scale flow features when coupled with either Suggar++ or Yoga. Additional, quantitative comparisons are shown in Figure 10. Here, airloads are compared at $r/R = 0.920$ for the UH60-A by plotting normal force, pitching moment, and chord force. Computational results generated by FUN3D plus Suggar++ and FUN3D plus Yoga are compared with experimental data. Airloads obtained on the grid system assembled with Yoga agree well with those obtained using the grid system assembled by Suggar++.

B. Invariance to Partitioning

One of Yoga's design requirements is that the assembled grid system should be invariant to the solver's initial partitioning. For each case presented in section III.C and III.D, this property is verified by running the case on different numbers of partitions, and different partitioning options (ParMETIS with and without weights, and Recursive Coordinate Bisection) and checking that the resulting grid system had the same number of In, Out, and Receptor points. Yoga provides the same result for each combination of MPI ranks and partitioning options, and is therefore invariant to partitioning.

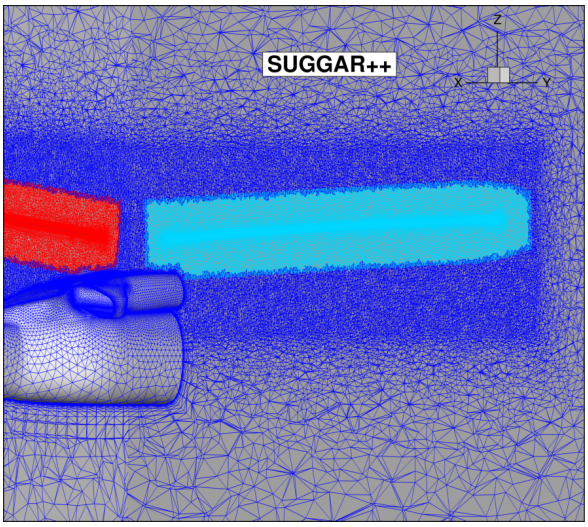


Fig. 6 A slice of the grid system is plotted, colored by component grid id to show the locations of intergrid boundaries generated by Suggar++.

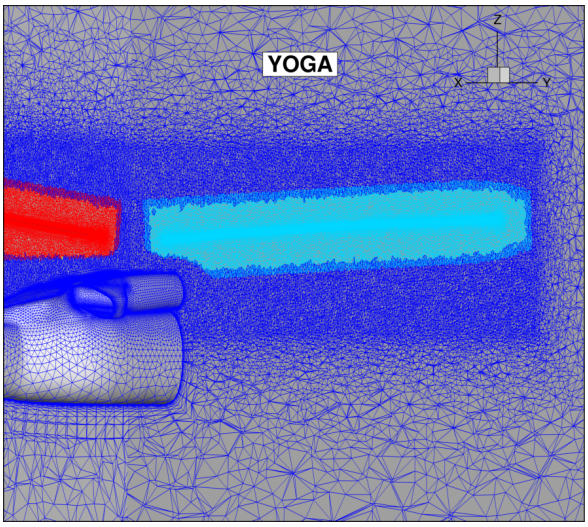


Fig. 7 The same plot is generated with FUN3D coupled with Yoga to compare intergrid boundary locations.

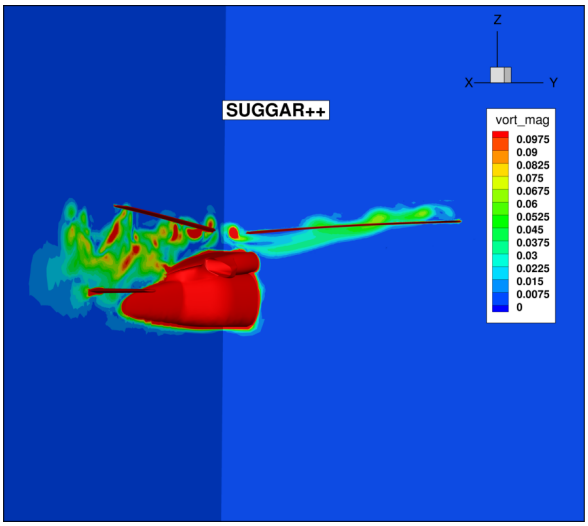


Fig. 8 Slices with vorticity magnitude are plotted for the baseline configuration of FUN3D with Suggar++.

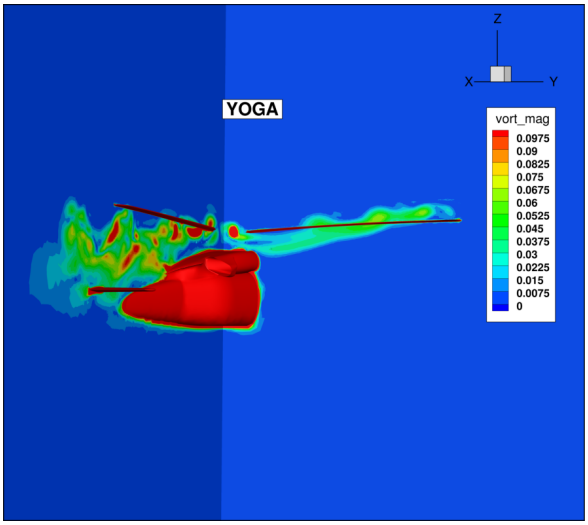


Fig. 9 Solution data from FUN3D and Yoga are plotted for comparison.

C. Strong Scaling

An MPI program that exhibits strong scaling will have a wall clock cost that is proportional to the number of processors for a problem of fixed size (e.g., a problem that costs 10 seconds in serial should cost 1 second on 10 MPI ranks.) FUN3D has been shown [12] to scale well on many thousands of MPI ranks for single grid simulations. For rotorcraft simulations with flexible blades, FUN3D has two additional components that influence overall scaling: overset domain assembly, and grid motion (including deformation). Results of a strong scaling study that captures measurements of the cost of all three components is presented in (III.C.1).

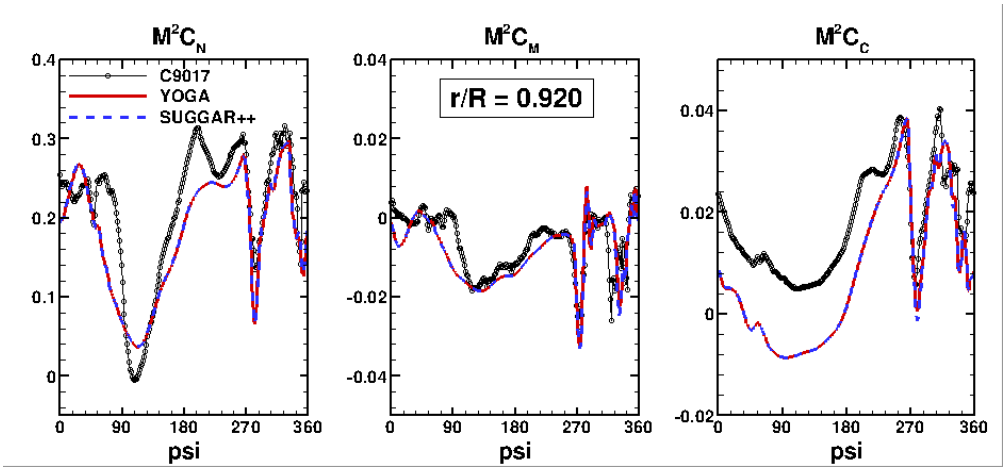


Fig. 10 UH60-A, C9017 airloads comparison at r/R = 0.920, CFD/CSD results computed using SUGGAR++ and YOGA. Normal force, pitching moment, and chord force are shown above from left to right.

1. HART-II Case Study

A strong scaling study is performed on a series of grid systems of varying grid size for the HART-II wind tunnel test geometry. Timing measurements are averaged over one quarter revolution to capture the full range of relative blade positions and make the sample representative of a full simulation. The first time Suggar++ is called from FUN3D, the domain assembly cost can be much higher than the cost of subsequent iterations. However, this cost has little impact on production simulations because it is amortized over hundreds or thousands of time steps. Therefore, timing measurements for the first time step of each test case are discarded to make the study more representative of a typical simulation. Each plot contains a line representing linear (i.e., perfect) strong scaling. Three scenarios are compared for each plot:

- FUN3D and Suggar++
- FUN3D and Yoga
- FUN3D and Yoga-unweighted

The baseline configuration, which is currently standard practice, is FUN3D coupled with Suggar++. The second configuration is FUN3D coupled with Yoga, where FUN3D's partition weight calculations are all set to their defaults. FUN3D has several options for calculating node, edge, and element weights to influence the quality of partitions generated by ParMETIS. These options can affect performance of the solver and overset domain assembly. Therefore, the final configuration is FUN3D coupled with Yoga, but with all partitioning weights turned off. For each case, all measurements are normalized based on the results of running FUN3D with Suggar++ on the smallest number of MPI ranks possible for that grid system size in order to measure strong scaling against current standard practice for FUN3D.

Figure 11 shows strong scaling of the FUN3D solver and mesh deformation operations in isolation. Ideal strong scaling is plotted in blue for reference, and the FUN3D solver and mesh deformation are plotted in yellow and green, respectively. From left to right, the number of MPI ranks ranges from 28 to 448, where the number of grid points per rank are roughly 250k and 15k. The FUN3D solver and mesh deformation both scale nearly linearly up to 30k grid points per MPI rank, which is in the range of standard practice. The FUN3D solver demonstrates further scaling to 15k nodes per MPI rank, while the scaling of the mesh deformation operation falls off. This is not surprising, given that only the fuselage grid is fixed, and only the blade grids are moving and deforming due to the flexibility of the blade geometries during the simulation (i.e., the mesh deformation problem itself is not load balanced.) The reduced scaling of the mesh deformation operation has a small impact on overall scalability of rotorcraft simulations however, because the wall clock cost is small relative to the solver across this range. For the largest number of MPI ranks for this case, the ratio of solver cost to mesh deformation is roughly 10:1. The combined scaling of the solver and mesh deformation is plotted in red to demonstrate the small impact of the mesh deformation due to its low relative cost. While the FUN3D solver and mesh deformation operations scale well across the selected range, the scalability of a full time step for a rotorcraft simulation is affected by the cost of domain assembly also. Figure 12 shows the scaling of a complete time step that includes all operations. For small numbers of MPI ranks, the cost of domain assembly is small compared to that of the solver, and scalability is dominated by the cost of the solver. However, since FUN3D calls Suggar++ in

serial, the assembly cost remains constant regardless of MPI ranks, and therefore reduces overall scalability.

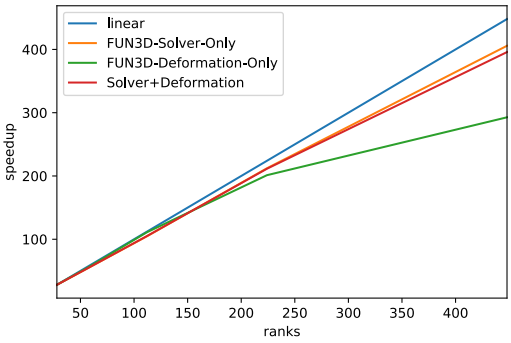


Fig. 11 Strong scaling measurements of the FUN3D solver and mesh deformation operations. The mesh deformation does not scale as well as the solver, but its baseline cost relative to the solver is low, so its impact on overall scalability is minimal.

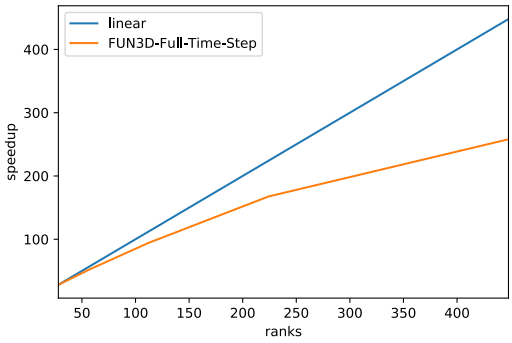


Fig. 12 Strong scaling measurement of a complete FUN3D time step, including the solver, mesh deformation, and overset domain assembly. When domain assembly is included in the full cost, scalability is decreased significantly.

Figure 13 shows a strong scaling plot for full time steps with FUN3D coupled with Suggar++, and FUN3D coupled with Yoga. FUN3D has several options for generating weights for the partitioner, and the resulting partition shapes have been shown to affect parallel performance in Yoga, therefore the cases with FUN3D coupled with Yoga are divided into two configurations. For the baseline configuration, FUN3D defaults to adding weights based on edges and component grid number. For the unweighted case, these weights are turned off explicitly via FUN3D's configuration options. As in previous plots, linear scaling is plotted in blue for comparison. Overall scalability for this case is only slightly improved for FUN3D and Yoga due to the overhead incurred for dynamic load balancing for such a small case. However, scalability improves further when FUN3D's partitioning weights are turned off. Partition shapes affect Yoga's communication cost, because Yoga must determine which ranks to query for grid information for every work unit. FUN3D's default partitioning weights tend to make very high aspect ratio partitions in the boundary layers of the blade component grids. Furthermore, blade grid partitions tend to be relatively concentric, where the AABB of a partition in the coarse outer portion of the blade grid fully contains the AABBs of multiple inner partitions. This configuration makes it difficult for Yoga to accurately determine which MPI ranks need to be queried for grid information based purely on partition AABBs and bit-masks, which causes numerous false positives and incurs significant parallel overhead. When the partitioning weights are turned off, partitions tend closer to unit aspect ratio and have less spatial overlap with other partitions of the same component grid. This enables Yoga to define its communication requirements more accurately, and ultimately leads to higher scalability.

Figure 14 shows a similar strong scaling plot for a 14 million node grid system on a range of MPI ranks from 56 to 896. Since this case is twice as large as the previous case, the cost of domain assembly is higher in absolute wall clock cost and cost relative to the solver. Therefore the assembly has a higher impact on overall scalability. Running the case with Yoga improves scaling over the range of MPI rank counts, but the improvement is reduced for the widest run (again, the widest run has roughly 15k grid points per MPI rank). The impact of FUN3D's partitioning weights is even more pronounced for this case.

Figure 15 shows a strong scaling plot for a 30 million node grid system on a range of MPI ranks from 224 to 1792. The effect of overset domain assembly on overall scalability continues to increase with grid system size. The effect of partitioning is even more pronounced at this grid size. For this case, scaling for the widest run is worse with Yoga with the default partitioning weights than with Suggar++. However, turning off the weights yields a marked improvement.

Figure 16 shows a strong scaling plot for a 150 million node grid system.

D. Weak Scaling

An MPI program that exhibits ideal weak scaling will have the same wall-clock cost regardless of problem size, provided that the number of processing units is proportional to the problem size (e.g., if an iteration costs 10 seconds

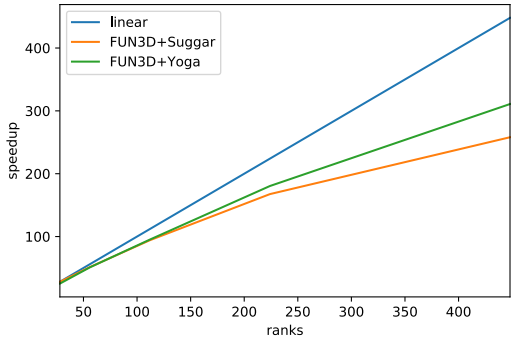


Fig. 13 Strong scaling measurements for a 7m node grid system on 28 to 448 MPI ranks.

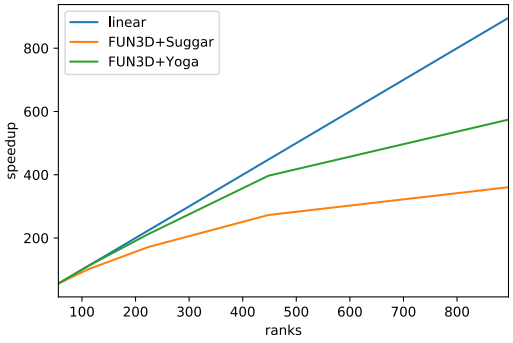


Fig. 14 Strong scaling measurements for a 14m node grid system on 56 to 896 MPI ranks.

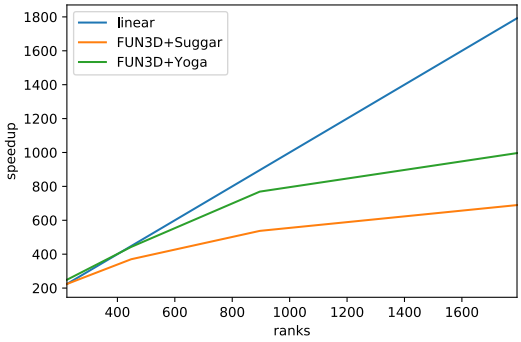


Fig. 15 Strong scaling measurements for a 30m node grid system on 224 to 1720 MPI ranks.

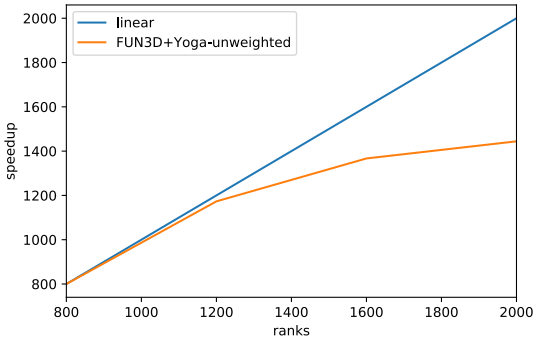


Fig. 16 Strong scaling measurements for a 150m node grid system on 800 to 2000 MPI ranks.

for a grid of 5 million nodes on 100 MPI ranks, an iteration should also cost 10 seconds for a grid of 50 million nodes on 1000 MPI ranks). Figure 17 shows weak scaling performance of FUN3D based on measurements taken for the strong scaling study. For each grid, the amount of work per processor is held roughly constant at 30 thousand grid points per rank, and the Yoga results are normalized with respect to the baseline Suggar++ results. Hence, the measurements for Yoga for the smallest grid are greater than 1 because the total cost was lower than the baseline. Both Yoga configurations demonstrate improved weak scaling compared to the baseline.

IV. Conclusion

Code on code verification was performed between FUN3D coupled with Suggar++ and FUN3D coupled with Yoga to demonstrate the viability of Yoga's assembly of overset grid systems. Yoga and Suggar++ use different criteria to determine intergrid boundary locations, therefore, domain assembly results are different. However, the solutions obtained by FUN3D using both assembly methods are qualitatively similar, and quantitative airload measurements also agree. Yoga's domain assembly has been verified to be invariant to partitioning, i.e., the location of intergrid boundaries, the number of receptor points, donor points, in points, and out points is the same on different numbers of MPI ranks. Yoga meets its design criteria of performing domain assembly in less time than an unsteady time step in the FUN3D solver for a range of grid system sizes and number of MPI ranks. Enhanced strong scalability for coupled rotorcraft simulations is demonstrated by strong scaling measurements that compare against FUN3D's baseline configuration. Enhanced weak scaling is also demonstrated by fixing the ratio of grid points to MPI ranks,

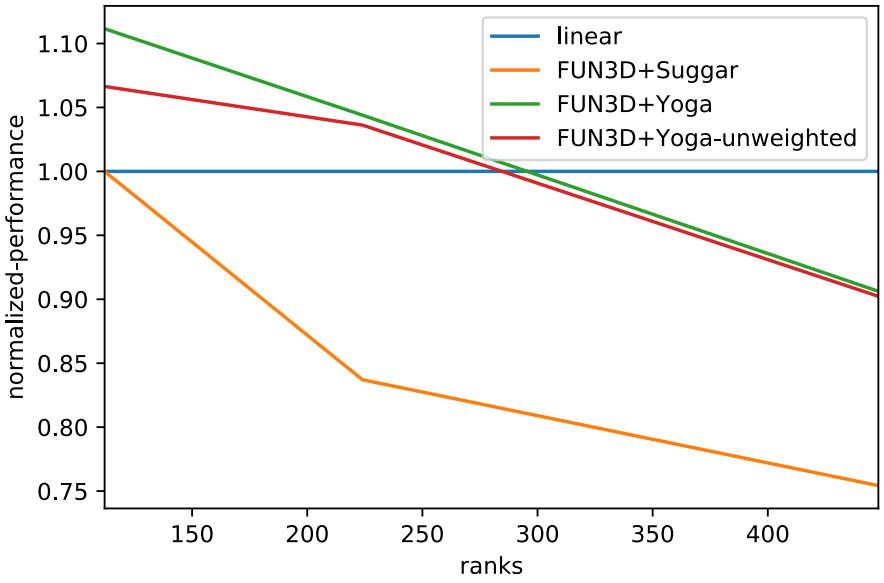


Fig. 17 Normalized weak scaling measurements, approximately 30k grid points per MPI rank.

and comparing total cost of FUN3D with Yoga and the baseline configuration of FUN3D with Suggar++. To make the cases presented representative of typical coupled rotorcraft simulations with FUN3D, grid sizes for the study range from 7 million to 150 million grid points, and processor loads range from 30 to 250 thousand grid points per MPI rank.

V. Acknowledgements

The author would like to thank Dr. Robert Biedron for his contributions to the effort of coupling FUN3D with Yoga, setting up test cases, and verifying results.

References

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

[2] Druyor, C. T., Karman, S. L., and Jones, W. T., "A Survey of Overset Domain Assembly Methods," AIAA Paper 2015–0910, 2015.

[3] Biedron, R. T., and Thomas, J. L., "Recent Enhancements to the FUN3D Flow Solver for Moving-Mesh Applications," AIAA Paper 2009–1360, 2009.

[4] Noack, R. W., "A Direct Cut Approach for Overset Hole Cutting," AIAA Paper 2007–3835, 2007.

[5] Noack, R. W., "DiRTlib: A Library to Add an Overset Capability to Your Flow Solver," AIAA Paper 2005–5116, 2005.

[6] Druyor, C. T., Jr., "Advances in Parallel Overset Domain Assembly," Ph.D. thesis, University of Tennessee at Chattanooga, Aug. 2016.

[7] Hintjens, P., *ZeroMQ: Messaging for Many Applications*, Oreilly and Associate Series, O'Reilly Media, Incorporated, 2013. URL `https://books.google.com/books?id=KWtT5CJc6FYC`.

[8] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*, 3rd ed., 2011. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`.

[9] Meakin, R. L., "A New Method for Establishing Intergrid Communication Among Systems of Overset Grids," AIAA Paper 91–1586, 1991. URL http://arc.aiaa.org/doi/pdf/10.2514/6.1991-1586.

[10] Williams, R. D., "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency: Practice and Experience*, Vol. 3, No. 5, 1991, pp. 457–481. doi:10.1002/cpe.4330030502, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4330030502.

[11] Bonet, J., and Peraire, J., "An Alternating Digital Tree (ADT) Algorithm for 3D Geometric Searching and Intersection Problems," *International Journal for Numerical Methods in Engineering*, Vol. 31, No. 1, 1991, pp. 1–17.

[12] Lee-Rausch, E. M., Hammond, D. P., Nielsen, E., Pirzadeh, S. Z., and Rumsey, C. L., "Application of the FUN3D Solver to the 4th AIAA Drag Prediction Workshop," *AIAA Journal of Aircraft*, Vol. 51, No. 4, 2014, pp. 1149–1160. doi:10.2514/1.C032558.