Reconfigurable model execution in the OpenMDAO framework

John T. Hwang *
NASA Glenn Research Center, 21000 Brookpark Rd, Cleveland, OH 44135
Peerless Technologies Corporation, 2300 National Rd, Beavercreek, OH 45324

NASA's OpenMDAO framework facilitates constructing complex models and computing their derivatives for multidisciplinary design optimization. Decomposing a model into components that follow a prescribed interface enables OpenMDAO to assemble multidisciplinary derivatives from the component derivatives using what amounts to the adjoint method, direct method, chain rule, global sensitivity equations, or any combination thereof, using the MAUD architecture. OpenMDAO also handles the distribution of processors among the disciplines by hierarchically grouping the components, and it automates the data transfer between components that are on different processors. These features have made OpenMDAO useful for applications in aircraft design, satellite design, wind turbine design, and aircraft engine design, among others. This paper presents new algorithms for OpenMDAO that enable reconfigurable model execution. This concept refers to dynamically changing, during execution, one or more of: the variable sizes, solution algorithm, parallel load balancing, or set of variables—i.e., adding and removing components, perhaps to switch to a higher-fidelity sub-model. Any component can reconfigure at any point, even when running in parallel with other components, and the reconfiguration algorithm presented here performs the synchronized updates to all other components that are affected. A reconfigurable software framework for multidisciplinary design optimization enables new adaptive solvers, adaptive parallelization, and new applications such as gradient-based optimization with overset flow solvers and adaptive mesh refinement. Benchmarking results demonstrate the time savings for reconfiguration compared to setting up the model again from scratch, which can be significant in large-scale problems. Additionally, the new reconfigurability feature is applied to a mission profile optimization problem for commercial aircraft where both the parametrization of the mission profile and the time discretization are adaptively refined, resulting in computational savings of roughly 10% and the elimination of oscillations in the optimized altitude profile.

I. Introduction

Modern gradient-based optimization techniques enable engineers to optimize models with thousands of design variables [1, 2, 3]. For optimizers based on sequential quadratic programming (SQP), the number of iterations required to solve the optimization problem often scales better than linearly with the number of design variables. The adjoint method yields further efficiency gains as it computes the derivatives required in each iteration at a fixed cost that does not increase appreciably with the number of design variables. Therefore, pairing an SQP optimizer with the adjoint method makes it feasible to optimize in, for instance, a ten-thousand-dimensional design space at the cost of running the model only hundreds of times, in practice.

As models mature and optimization techniques improve, there is a push towards more disciplines and higher model complexity to improve their predictive capabilities. In this context, higher *complexity* refers to having more parts that are inter-connected. For instance, a single-discipline model may be complex because it contains an ordinary differential equation (ODE), a surrogate model, and a system of algebraic equations—all of which are coupled. On the other hand, a multidisciplinary model may contain straightforward blackbox sub-models, but in this case, the challenge may come from data exchange between disciplines and resolving feedback loops between disciplines.

Given a model that is multidisciplinary, complex, or both, performing large-scale optimization adds an additional challenge because the adjoint method is an invasive technique that requires extensive modification of the model, unlike simpler methods for computing derivatives. This encourages striving for scalability and modularity, to ensure the efficiency and maintainability of the code stay manageable as the number of disciplines and the complexity of the model increases.

NASA's OpenMDAO software framework is motivated by this goal of taking a modular approach to large-scale optimization. OpenMDAO is a NASA-funded open-source framework for gradient-based multidisciplinary design optimization, written in Python [4]. There are many existing open-source and commercial frameworks that also seek to

*Research engineer (contractor at NASA GRC)

enable a modular approach to multidisciplinary design optimization: Phoenix Integration's ModelCenter/CenterLink, Dassault Systèmes' Isight/SEE, Esteco's modeFRONTIER [5], TechnoSoft's AML suite, MathWorks' MATLAB/Simulink, Noesis Solutions' Optimus, Vanderplaats' VisualDOC [6], and SORCER [7]. However, OpenMDAO is unique because it is designed for gradient-based optimization—it facilitates efficient and accurate derivative computation using methods such as the adjoint method. It does this by using the modular analysis and unified derivatives (MAUD) architecture [8], a formulation that unifies all methods for computing derivatives using a single linear equation and solves the multidisciplinary systems using a parallel, hierarchical approach. MAUD and OpenMDAO have been used to solve design problems for wind turbines [9, 10, 11, 12], satellites [3], aircraft wings [13, 2, 14, 15], aircraft mission and allocation [16, 17, 18, 2, 19], and aircraft engines [20, 21].

As is the case with other frameworks, OpenMDAO's fundamental paradigm divides the model into components (which can represent a discipline or one part of a discipline). The components are in turn grouped hierarchically, both for convenience and for efficient execution. OpenMDAO is designed for parallel computing, so components can be run in parallel themselves, or the model can be parallelized across components, which means different components are assigned to different processors. OpenMDAO passes data between components, performing the necessary interprocessor communication. A key source of efficiency is pre-computing several data structures during *setup*, including these parallel communication patterns, so that operations carried out during execution are as efficient as possible.

This paper focuses on reformulating the core design and algorithms in OpenMDAO to enable dynamic *reconfiguration*. Reconfiguration refers to changing the model during execution—in the middle of converging the multidisciplinary system or at the start of each optimization iteration—and subsequently repeating the parts of the setup process that must be updated. This term is also used in a different, but related context: reconfigurable engineering systems, where, according to Ferguson et al. [22], the three motivating factors are multi-ability (performing different functions at different times), evolution (morphing into unplanned configurations), and survivability (handling unexpected failures). Here, only the model is reconfiguring, and not the engineering system.

Specifically, reconfigurability in this context allows one of four model properties to change in one or more components: sizes of variables, the division of processors, the hierarchical grouping of components, and the set of variables (i.e., adding or removing variables). The aim is to determine the correct sequence of 're-setup' that must occur after a component or a set of components reconfigure, to ensure that the rest of the model remains synchronized. For instance, if component B depends on component A, and component A changes variable sizes while running in parallel with component B, component B must be updated with the new information at the right time for the overall model to remain correct and consistent. The secondary aim is to do this while minimizing the amount of setup that is repeated, especially when a small part of the model reconfigures in a large, complex overall model.

Potential applications of reconfigurability include adaptive discretizations, adaptive solvers, and multi-fidelity approaches, among others. Problems with adaptive discretizations require changing variable sizes, which may occur during a nonlinear system or at the start of each optimization iteration, e.g., to refine a mesh. Adaptive solvers would not add or remove variables or change their sizes, but they could change the hierarchical grouping of components, the processor distribution, or both. Multi-fidelity approaches would add and remove variables to switch been low- and high-fidelity sub-models over the course of a simulation or optimization.

II. OpenMDAO

This section presents an overview of the mathematical formulation and core operations of OpenMDAO using a simple example that will be used throughout. The intent is to provide a graceful introduction to OpenMDAO for readers who have limited experience with it, and to provide context for the discussion on the methodology for reconfigurability.

A. A simple illustrative model

The example is a model with a single design variable, two coupled disciplines, and an objective function, as shown in Tab. 1. In this example, the two disciplines explicitly define their state variables, u_2 and u_3, and each discipline depends on the other, so we characterize them as *coupled*. The objective function, u_4, depends on the two state variables as well as the design variable, u_1. The discussion that follows will reference this simple example to aid in explaining of how OpenMDAO formulates models.

B. Monolithic formulation for derivative computation

OpenMDAO uses the *modular analysis and unified derivatives* (MAUD) architecture [8]. MAUD is a unique way of formulating the model mathematically that simplifies derivative computation for the framework, and it also provides a modular structure for applying nonlinear and linear solution techniques.

Component	Outputs	Inputs	Equation	Residual
Design variable:	u_1		$u_1 = 3$	$R_1(u_1, u_2, u_3, u_4) = u_1 - 3$
First discipline:	u_2	u_1, u_3	$u_2 = 2u_3 + u_1$	$R_2(u_1, u_2, u_3, u_4) = u_2 - 2u_3 - u_1$
Second discipline:	u_3	u_1, u_2	$u_3 = 2u_2 + u_1$	$R_3(u_1, u_2, u_3, u_4) = u_3 - 2u_2 - u_1$
Objective:	u_4	u_1, u_2, u_3	$u_4 = u_1 + u_2 + u_3$	$R_4(u_1, u_2, u_3, u_4) = u_4 - u_1 - u_2 - u_3$

Table 1: The simple example used throughout Sec. II. The model has 4 components, including the design variable, two coupled disciplines with one state variable each, and an objective function variable. The model is evaluated at a design variable value of 3. OpenMDAO is unique because it uses the MAUD architecture [8], which treats design variables as components and assigns residuals to each variable, allowing the formulation of the model as a nonlinear system of equations. This unifies all methods for computing multidisciplinary derivatives, and it also unifies all solution approaches as types of nonlinear solvers.

In addition to presenting the example, Tab. 1 also illustrates MAUD's unique model formulation. The idea is to treat all variables—design, state, objective, constraint, etc.—in the same way, as simply the output of a component. This is why there is a component that 'computes' the design variable, which is not customary in other frameworks or settings, and the variables are named (u_1, u_2, u_3, u_4) rather than, say, (x, y_1, y_2, f). The equation, $u_1 = 3$, indicates that the model is being evaluated with the design variable set to a value of 3. For each variable, a residual function is defined so that solving the resulting $n \times n$ nonlinear system yields the result of running the model. In Tab. 1, it is easy to see that driving the residuals to zero is equivalent to running this model—solving the coupled disciplines and evaluating the objective would yield the same values of (u_1, u_2, u_3, u_4). In this manner, any model can be formulated as a single nonlinear system,

$$R(u) = 0, \tag{1}$$

where in this example, $R = (R_1, R_2, R_3, R_4)$ and $u = (u_1, u_2, u_3, u_4)$.

The primary benefit of formulating this nonlinear system is that it leads to an equation that unifies all the methods for assembling the *model derivatives*, e.g., du_4/du_1, given user-provided or approximated *component derivatives*, e.g., $\partial R_2/\partial u_3$. Model derivatives are total derivatives that factor in the dependencies between components, while component derivatives are partial derivatives that are local to the component. Methods for assembling the model derivatives including the chain rule, the direct and adjoint methods, a coupled version of the chain rule (GSE2 [23]), or hybrid methods that combine any of the above. All of these methods are unified by the following equation [24], which is derived from the nonlinear system using the inverse function theorem [8]:

$$\frac{\partial R}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}r} = \mathcal{I} = \frac{\partial R}{\partial u}^T \frac{\mathrm{d}u}{\mathrm{d}r}^T, \tag{2}$$

where $\partial R/\partial u$ is the matrix of consisting of component (partial) derivatives, and du/dr contains the model (total) derivatives. This equation is called the *unifying derivatives equation*. Note that the r in du/dr is reflective of the fact that $du/dr = \partial R^{-1}/\partial r$ [8]; in this example, all four variables are explicitly deflined so the r is replaced with u in the denominator in the equations that follow.

For the current example, the left equality of (2) expands as follows:

$$\begin{bmatrix} \frac{\partial R_1}{\partial u_1} & \frac{\partial R_1}{\partial u_2} & \frac{\partial R_1}{\partial u_3} & \frac{\partial R_1}{\partial u_4} \\ \frac{\partial R_2}{\partial u_1} & \frac{\partial R_2}{\partial u_2} & \frac{\partial R_2}{\partial u_3} & \frac{\partial R_2}{\partial u_4} \\ \frac{\partial R_3}{\partial u_1} & \frac{\partial R_3}{\partial u_2} & \frac{\partial R_3}{\partial u_3} & \frac{\partial R_3}{\partial u_4} \\ \frac{\partial R_4}{\partial u_1} & \frac{\partial R_4}{\partial u_2} & \frac{\partial R_4}{\partial u_3} & \frac{\partial R_4}{\partial u_4} \end{bmatrix} \begin{bmatrix} \frac{\mathrm{d}u_1}{\mathrm{d}u_1} & \frac{\mathrm{d}u_1}{\mathrm{d}u_2} & \frac{\mathrm{d}u_1}{\mathrm{d}u_3} & \frac{\mathrm{d}u_1}{\mathrm{d}u_4} \\ \frac{\mathrm{d}u_2}{\mathrm{d}u_1} & \frac{\mathrm{d}u_2}{\mathrm{d}u_2} & \frac{\mathrm{d}u_2}{\mathrm{d}u_3} & \frac{\mathrm{d}u_2}{\mathrm{d}u_4} \\ \frac{\mathrm{d}u_3}{\mathrm{d}u_1} & \frac{\mathrm{d}u_3}{\mathrm{d}u_2} & \frac{\mathrm{d}u_3}{\mathrm{d}u_3} & \frac{\mathrm{d}u_3}{\mathrm{d}u_4} \\ \frac{\mathrm{d}u_4}{\mathrm{d}u_1} & \frac{\mathrm{d}u_4}{\mathrm{d}u_2} & \frac{\mathrm{d}u_4}{\mathrm{d}u_3} & \frac{\mathrm{d}u_4}{\mathrm{d}u_4} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & -2 & 0 \\ -1 & -2 & 1 & 0 \\ -1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{\mathrm{d}u_2}{\mathrm{d}u_1} & 1 & \frac{\mathrm{d}u_2}{\mathrm{d}u_3} & 0 \\ \frac{\mathrm{d}u_3}{\mathrm{d}u_1} & \frac{\mathrm{d}u_3}{\mathrm{d}u_2} & 1 & 0 \\ \frac{\mathrm{d}u_4}{\mathrm{d}u_1} & \frac{\mathrm{d}u_4}{\mathrm{d}u_2} & \frac{\mathrm{d}u_4}{\mathrm{d}u_3} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4}$$

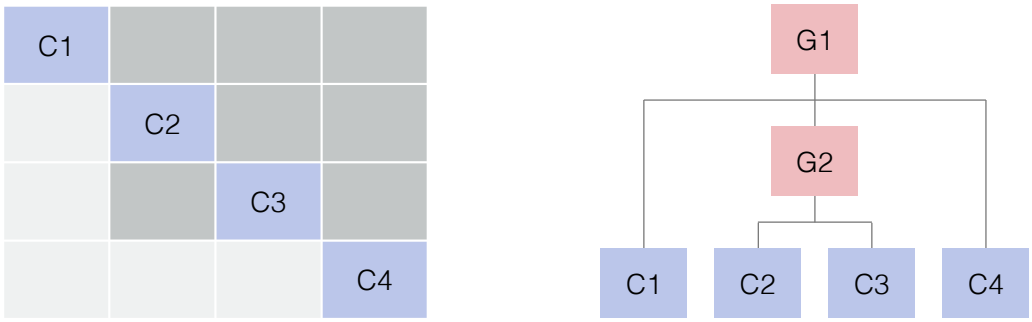
Therefore, the derivative of the objective with respect to the design variable, du_4/du_1, can be computed by assembling the $\partial R/\partial u$ matrix and solving the linear system with the correct right-hand side.

A secondary outcome due to formulating the model as a nonlinear system is that solution approaches used to run the model are also unified, under the umbrella of 'nonlinear solvers'. In the current example, running the model would conventionally consist in first assigning 3 to u_1, then solving for u_2 and u_3 together since they are coupled, and finally evaluating u_4. This can be interpreted as a single iteration of the nonlinear block Gauss–Seidel method, where the three blocks are (u_1), (u_2, u_3), and (u_4). While this is a trivially simple example, the interpretation of all solution approaches as simply a nonlinear or linear solver simplifies the framework operations appreciably, in larger and more complex models.

C. Parallel, hierarchical decomposition

While treating the full model monolithically has mathematical benefits, it can be inefficient with a naive approach. For instance, it would not make sense to apply a Newton solver to the full nonlinear system with all 4 variables included, in the simple example. Therefore, MAUD partitions the nonlinear system hierarchically, just as components are commonly grouped hierarchically in traditional software frameworks. In OpenMDAO, there are *Component* and *Group* classes, both of which inherit from a base *System* class.

Figure 1 shows the dependency graph and hierarchy tree for our simple example. Since components C2 and C3 are coupled, we assign them to group G2, and group G1 is the root group containing component C1, group G2, and component C4. The solution approach would then be to run a single iteration of nonlinear block Gauss–Seidel in group G1, and group G2 could either run a Newton solver, its own nonlinear Gauss–Seidel iteration, or any other custom or OpenMDAO-provided nonlinear solver.



Dependency graph Hierarchy tree

Figure 1: The dependency graph suggests an obvious way to group the components. Group G2 contains C2 and C3, and at the top, group G1 contains C1, G2, and C4. Note: the dependency graph is the transpose of the Jacobian $\partial R/\partial u$ because figures of this kind show feed-forward dependencies above the diagonal and feed-back dependencies below the diagonal.

Each intermediate group in a model 'owns' or is responsible for a subset of the residuals in the global nonlinear system. Group G2's nonlinear system would be

$$\begin{aligned} R_2(u_1, u_2, u_3, u_4) &= 0 \\ R_3(u_1, u_2, u_3, u_4) &= 0, \end{aligned} \tag{5}$$

where the unknowns of the nonlinear system are u_2 and u_3, while the parameters are u_1 and u_4. Therefore, during group G1's nonlinear block Gauss–Seidel iteration, G2 would solve this nonlinear system when called to block-solve itself.

The model can be run with distributed-memory parallel computing in one of two ways: parallel components or parallel groups. Parallel components are those that are assigned multiple processors and distribute its output among the processors. For instance, if the component has 10 processors and it computes a mesh of size 1000, each processor would be responsible for computing 100 of the 1000 nodes in the mesh. In contrast, non-parallel components compute the same output on all processors. Parallel groups are those that are assigned multiple processors and distribute its processors among its children with no overlap, but not necessarily uniformly. If a parallel group has 4 processors and two children, it could split the processors into 1 and 3, 2 and 2, or 3 and 1. Alternatively, a non-parallel group gives

all of its processors to all of its children, so both of the children would be assigned 4 processors in this case. Parallel groups cannot run nonlinear or linear block Gauss–Seidel solvers; instead, they would use nonlinear or linear block Jacobi solver, or a non-hierarchical solver such as Newton or a Krylov subspace method.

D. Data structures and setup

A core aspect of MAUD and OpenMDAO that leads to efficiency gains is the concatenation of variables into contiguous arrays to enable faster vector operations in compiled languages, as shown in Fig. 2. As an example, vector addition would consist in a single for loop over all entries of the vector in a compiled language such as C, rather than looping over each variable in Python and performing separate additions. The latter is much slower if there are a large number of variables because of the inefficiency of for loops in scripted languages. This approach does not compromise convenience as components still access individual variables by their string names, so for example input u_1 is accessed from component C2 via `inputs['u_1']`. Internally, this is achieved by making `inputs` a dictionary where each key is a variable name and the value is a *NumPy view*, i.e., a pointer onto a part of the larger vector. The vector attributes of all groups and components are pointers to sub-vectors of the full underlying vector containing all variables from the entire model. Note: Fig. 2 shows outputs with arrows pointing into the component because outputs are considered arguments to the residual function. However, they can also be interpreted as their namesake—outputs to the component—since the residuals implicitly define the outputs as a function of the inputs.

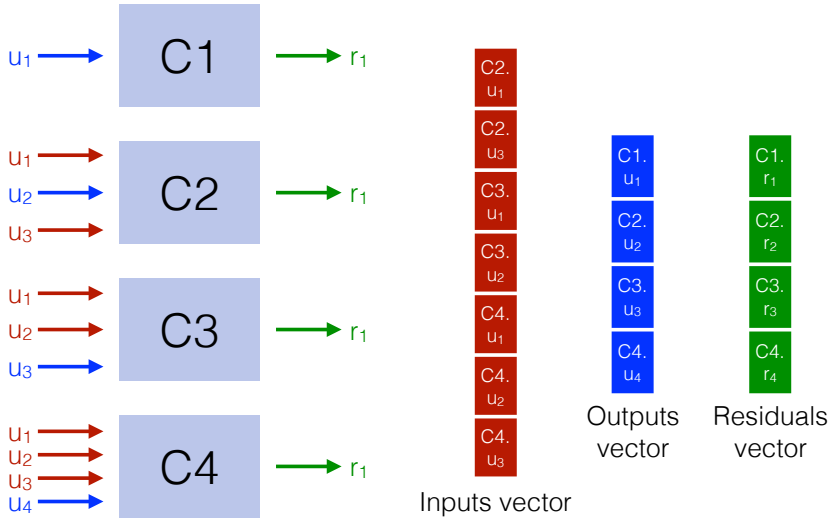


Figure 2: Variables are concatenated into arrays that are contiguous in memory to enable fast vector operations. Note: the outputs have arrows pointing into the component because they are considered arguments to the residual function.

Data transfers are operations on these concatenated vectors that are also designed to avoid slow for loops in Python. Transfers pass data from the output of one component to the input of another. In general, the source output is not always on the same processor as the target input; alternatively, parts of the requested output can be on different processors if the output is defined in a parallel component, as discussed in the previous section. During setup, the framework determines the processor(s) and the indices of the data in the output vectors for each input. Moreover, multiple transfers are performed simultaneously, partly for the speed of vector operations once again, but also to maximize interprocessor communication efficiency. However, it is important to note that transfers occur in groups; in particular, they occur in the *lowest common ancestor*, which is the lowest-level group containing both the input and the output, as Fig. 3 shows.

The takeaway from this section is that OpenMDAO performs key setup operations to allocate these concatenated vectors, assign to each system (i.e., group or component) a view into part of the concatenated vector, compute the dictionaries to enable string access to individual variables in components, and compute the data transfer indices, including any necessary parallel communication patterns. This setup time can be significant (on the order of minutes) in large models with hundreds of variables and processors, e.g., parallel aircraft allocation-mission-design optimization [2].

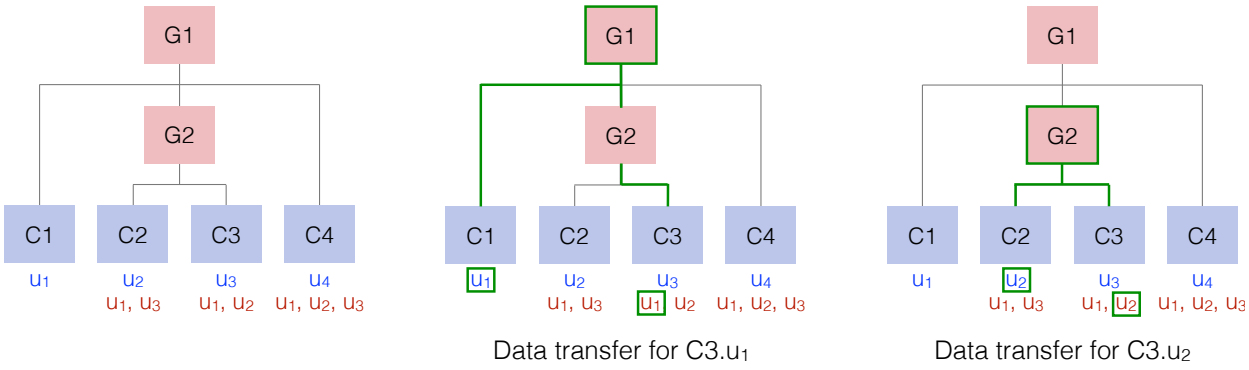


Figure 3: Transfers are performed by the lowest common ancestor for the input and source output. For instance, the $C1.u_1$-to-$C3.u_1$ transfer is performed by group G1, while the $C2.u_2$-to-$C3.u_2$ transfer is performed by group G2.

Updating and re-computing the setup operations after reconfiguration is the focus of the reconfiguration methodology presented in this paper, and this is discussed in the next section.

III. Reconfigurability

This section discusses the motivation and methodology for reconfigurability, and presents benchmarking results for reconfiguration setup. The primary goal for the reconfiguration methodology is to ensure the correct behavior and timing of reconfiguration operations, especially when an intermediate group or component reconfigures, and the secondary goal is minimizing the setup operations that must be repeated when a small part of the model changes.

A. Motivation and applications

Fundamentally, reconfiguration refers to changing one of four model properties during execution: variable sizes, the processor distribution, the hierarchical grouping of components, and the set of variables (i.e., adding or removing variables). More than one of these four can change simultaneously, and in fact, changing the hierarchical grouping changes the processor distribution in parallel groups, which in turn changes local variable sizes in parallel components. There are other aspects of the model that may change during execution, e.g., changing solvers, changing the finite-difference step size, convergence tolerance, etc. However, these changes amount to only logistical details and do not present the same technical challenges as the four aforementioned properties, which affect the dependency graph, hierarchy tree, or both.

Problems and applications that motivate reconfiguration are those in which building the model again from scratch and performing a full setup is not feasible or desirable. In parallel models, reconfiguration may occur on a subset of the processors; however, the full setup requires all processors to participate. Similarly, reconfiguration may occur in an intermediate group or a component multiple times in a solver iteration before returning to the root group, so waiting until then to perform a full setup would be too late. Even if neither of these cases applies, a full setup may be undesirable in problems in which a small part of a large model reconfigures, as the impact of the change would be small, and most of the setup would be unnecessary to repeat.

1. VARIABLE SIZES Applications for reconfiguring variable sizes include adaptive time-stepping, adaptive mesh refinement, and adaptive parametrizations. Adaptive time-stepping in OpenMDAO requires reconfigurability because the variable size for the time-series variable is in general different in each execution of the time-stepping ordinary differential equation (ODE) solver. Adaptive mesh refinement may occur at the start of each optimization iteration or in the middle of running the model, in which case a full setup is not possible. Even if it occurs at the start of the optimization iteration, the impact of the change may be limited to the functionals that are computed on the mesh, making a full setup wasteful, which is why a reconfiguration setup algorithm is desirable.

2. PROCESSOR DISTRIBUTION Applications for adaptive processor distributions involve models with parallel groups. An example is aerostructural analysis consisting of high-fidelity computational fluid dynamics (CFD) and finite element analysis (FEA) solvers. If both solvers are flexible with respect to the number of processors, it can be

beneficial to adjust the division of processors based on each solver's performance. Given that CFD solvers are nonlinear (and thus may require a startup sequence) and FEA solvers are often sufficiently accurate with a linear formulation, it would make sense, for instance, to allocate more processors to the CFD solver early on in the optimization when there are large design changes. Later, the algorithm would assign a more equal distribution of processors, closer to optimization convergence, because solving the coupled Newton system would likely dominate the model analysis time at that point. Alternatively, the solution algorithm in this or another application could recompute the processor distribution each time the model evaluates, based on how much time each discipline required in the previous optimization iteration.

3. HIERARCHICAL GROUPING Most applications for changing the hierarchy tree dynamically involve adaptive solvers. In many problems, groups can benefit from starting with a nonlinear block Gauss–Seidel then switching to a coupled Newton approach. The former provides globalization and fast convergence in weakly coupled problems [25], while the latter provides fast convergence when given a good starting point. An adaptive solver may also determine optimal clusters of components and add groups dynamically, to enable faster block solution approaches.

4. ADDING AND REMOVING VARIABLES This last type of reconfiguration involves adding and removing variables and potentially adding and removing components. One application is shape optimization involving large design changes, e.g., overset CFD optimization. Since overset CFD involves a finite number of sub-meshes based on intersections between geometric objects, sub-meshes could appear and disappear when large design changes are permitted. Another application is multi-fidelity optimization; switching from a low-fidelity model to a higher fidelity model during the optimization would involve entire components and variables being removed and added.

B. Methodology

When a component or group declares that it wants to change its variables' sizes, processor distribution, etc., the 'reconfiguration' operation that OpenMDAO must run is a subset of the setup process. OpenMDAO's setup process initializes solvers, assembles lists of variable names, and computes several other data structures that describe the model. However, the two most important parts of setup are allocating the concatenated vectors shown in Fig. 2 and computing the transfer information. Transfer information consists of the indices of the source output vector and the indices of the target input vector along with the processors that are involved in the transfer. The reconfiguration setup operation repeats many of the same operations performed in the traditional setup process, which is performed at the beginning of every OpenMDAO script. However, reconfiguration repeats only the parts that must be updated, and it runs in multiple stages, which are described in this section.

The reconfiguration methodology consists of 3 types of setup operations: full, reconfiguration, and update setup. Full setup is the normal setup operation that must always be performed prior to running any OpenMDAO model. It must always be called from the top-level group. Reconfiguration setup is called by the group or component that initiates reconfiguration, and update setup is called by all ancestors (i.e., all groups containing the initiating group or component) when their descendants are finished running their Gauss–Seidel or Jacobi iterations and the call stack passes the baton back to the ancestor.

Tab. 2 presents the specific functions of each operation. Reconfiguration setup resizes the vectors rather than allocating new vectors, but is otherwise identical to full setup. Update setup does not recurse into its subsystems except to initialize the vector views that are discussed in Sec. II.D.

Setup operation	Initializes variable data	Initializes vectors	Initializes transfers	Initializes solvers, derivs.
Full	Yes, recursively	Allocates vectors *	Yes, recursively	Yes, recursively
Reconfiguration	Yes, recursively	Resizes vectors *	Yes, recursively	Yes, recursively
Update	Yes; not recursively	*	Yes; not recursively	Yes; not recursively

* Initializes views recursively, but allocating or resizing vectors (if applicable) is not recursive.

Table 2: Description of the setup operations. The last column includes nonlinear solvers, linear solvers, finite-difference approximations, and Jacobian matrices.

Figure 4 illustrates the sequence of reconfiguration and update setup via an example. In the model, Group 4 is the one initiating reconfiguration to change variable sizes, hierarchy, processor distribution, its set of variables, or any combination thereof. It recurses into Group 5 and Components 1, 2, and 3, and performs effectively a full setup, other than resizing instead of allocating vectors so that the current-iteration values from Components 4, 5, and 6 are

not lost. Subsequently, once Group 4 and Component 4 complete execution, Group 2 performs an update setup, after detecting that Group 4 has reconfigured. During Group 2's update setup, it is not necessary to recurse into Component 4 since it has not changed; only the vector views must be re-initialized since all indices would have shifted if Group 4 changed variable sizes. Similarly, once Group 2 and Group 3 complete execution, Group 1 performs an update setup, after detecting that Group 2 has changed. Again, during Group 1's update setup, a full recursion into Group 3, Component 4, and Component 6 is unnecessary since their variable names, sizes, etc. are still up-to-date. Recursion is only performed to update their vector views in case the reconfiguration in Group 4 involved changes in variable sizes.

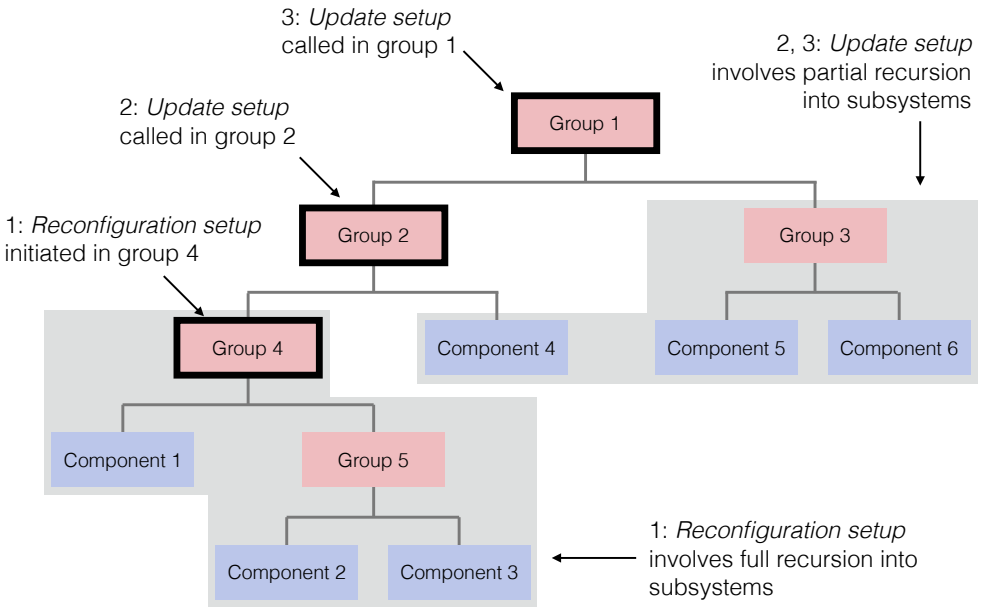


Figure 4: Reconfiguration sequence in an example model. Group 4 is assumed to change variable sizes, hierarchy, processor distribution, or set of variables.

Algorithms 1 and 2 present the pseudocode for the nonlinear block Gauss–Seidel and Jacobi solvers with reconfiguration incorporated. The two solvers are presented as methods on the System class, which is the base class for Group and Component. In both cases, the first step is to check if the current group wants to reconfigure. If so, a reconfiguration setup is triggered and a flag is set to inform the immediate parent group later that it will need to update. The Gauss–Seidel iteration loops over subsystems, and for each subsystem, it performs a transfer to pass data from the outputs of all other subsystems to the relevant inputs of the subsystem, before calling the subsystem to solve its nonlinear system. Next, an update setup is performed if the subsystem has reconfigured, in which case the system's own reconfigured flag is turned on. For Jacobi, a single transfer is first performed to pass data from the outputs of all subsystems to the inputs of all subsystems at once, before all subsystems solve their nonlinear systems. Next, an update setup is performed if any subsystem has reconfigured, and if so, the system's own reconfigured flag is turned on. For groups whose subsystems are sequentially dependent on each other or are fully decoupled, it is sufficient to run a single iteration of this Gauss–Seidel (Alg. 1) or Jacobi (Alg. 2) algorithm, respectively, and the reconfiguration operations would work the same way. Algorithms for reconfigurable linear block Gauss–Seidel and Jacobi are not presented here because reconfiguration is not permitted during a linear solution.

Algorithm 2 handles the synchronization that is required when reconfiguration occurs in a group or component while running in parallel with other groups or components. If a model is run with n processors in total, and the reconfiguring system has m processors where $m < n$, this means that at least one of the ancestor groups is a parallel group. Without loss of generality, let us assume that the immediate parent is the parallel group, there is one other component within that immediate parent called Component B, and the reconfiguring system is a component called Component A. The challenge is that Component B continues to execute in parallel while Component A reconfigures, perhaps even multiple times in a row, despite the fact that Component B might depend on Component A. As long as the variable in Component A that Component B depends on does not disappear, the reconfiguration can be valid, even if the variable is allocated on different processors post-reconfiguration. Algorithm 2 handles this situation by synchronizing after the for loop in line 8. By the time line 11 is reached, all subsystems—Components A and B in this

case—have completed their execution, and if any of them has reconfigured, update setup can be performed since all processors of the current system are synchronized. Prior to that, in line 9, Component B continues to run with the old values of the variable from Component A that it has as an input. The new values are not needed until the next transfer in line 7 of the next iteration, but prior to that, update setup is performed in line 12, if necessary. The same logic and algorithm work recursively if multiple ancestors of a reconfiguring system are parallel groups.

Algorithm 1 Nonlinear block Gauss–Seidel with reconfiguration

```
1:  function SOLVE_NONLINEAR_BLOCK_GS(self)        ▷ This is a method on System (Group or Component)
2:      if self.reconfigure() then                  ▷ Check if the current system wants to reconfigure
3:          self.reconfiguration_setup()
4:          self.has_reconfigured ← True            ▷ Flags to the parent group to update
5:      end if
6:      while Not converged and not out of iterations do
7:          for all subsys in self.subsystems do
8:              self.transfer(subsys)               ▷ Transfer inputs to subsys only
9:              subsys.solve_nonlinear()
10:             if subsys.has_reconfigured then     ▷ Check if subsys has reconfigured
11:                 self.update_setup()
12:                 self.has_reconfigured ← True    ▷ Flags to the parent group to update
13:                 subsys.has_reconfigured ← False
14:             end if
15:         end for
16:     end while
17: end function
```

Algorithm 2 Nonlinear block Jacobi with reconfiguration

```
1:  function SOLVE_NONLINEAR_BLOCK_JACOBI(self)    ▷ This is a method on System (Group or Component)
2:      if self.reconfigure() then                  ▷ Check if the current system wants to reconfigure
3:          self.reconfiguration_setup()
4:          self.has_reconfigured ← True            ▷ Flags to the parent group to update
5:      end if
6:      while Not converged and not out of iterations do
7:          self.transfer()                         ▷ Transfer inputs to all subsystems only
8:          for all subsys in self.local_subsystems do   ▷ Only the subsystems on this processor
9:              subsys.solve_nonlinear()
10:         end for
11:         if subsys.has_reconfigured for any subsys on any proc. then   ▷ Check if any subsys has reconfigured
12:             self.update_setup()
13:             self.has_reconfigured ← True        ▷ Flags to the parent group to update
14:         end if
15:         for all subsys in self.local_subsystems do   ▷ Only the subsystems on this processor
16:             subsys.has_reconfigured ← False
17:         end for
18:     end while
19: end function
```

C. Benchmarking results

As stated in the introduction, the primary need for the reconfigurability feature is to enable any component or group to change immediately during a nonlinear solution without waiting to return to the top-level group, where all processors are available and all parts of the model are known. A secondary motivation for reconfigurability is to avoid repeating the full set of setup operations when only a small part of the model changes. This section presents timings for full and reconfiguration setup, to address the secondary motivation and explore how much efficiency is gained with the

methodology presented in this paper. Setup times are typically not noticeable in small- and medium-sized problems but they can be on the order of minutes in large, parallel models, and small differences in time can magnify over the course of multiple reconfigurations in a simulation or multiple simulations within an optimization.

The benchmarking results are generated on a scalable toy problem. The root (top-level) group contains a component that reconfigures and a group that contains an variable number of components. All components compute a single output variable, the size of which can be varied, and the number of inputs is controlled by a sparsity parameter. Therefore, there are three parameters in total—number of components, sparsity parameter, and variable size—and the first two parameters can be visualized by referring to Fig. 5.

The plots in Fig. 5 compare the full setup and reconfiguration time (reconfiguration setup and update setup combined) versus number of components. The curves are generated for variable sizes of 1 and 100 and sparsity parameters of 1 and 10. We can see that in all cases, reconfiguration is faster than full setup by a factor of roughly 4, and more significantly, the gap grows as the problem becomes larger. Although one might expect the factor to be the number of components, savings to this extent are not possible because of the update setup that is performed in the top-level group and the partial recursion into all components in the model. Note that these results are generated with a preliminary, unreleased version of OpenMDAO v2, so the factor and absolute setup times are likely to change prior to release of v2. Unsurprisingly, sparsity has a large effect on setup time, but variable size has a neglible effect, likely due to setup operation times being dominated by Python-level operations rather than vector operations.

IV. Application: adaptive refinement

This section presents an application of reconfiguration in an aircraft mission optimization problem. In this problem, reconfiguration is used to adaptively refine the parametrization and discretization in time with the goal of reducing the required number of optimization iterations.

A. The aircraft mission design problem

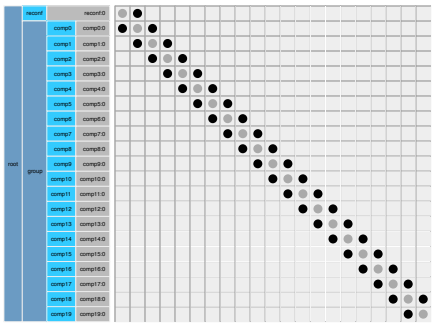
The mission analysis model [16, 2] is multidisciplinary; it incorporates aircraft aerodynamics, propulsion, the equations of motion over the mission profile, and atmospherics sub-models. It uses surrogate models for the aerodynamics and propulsion, and solves an ODE to compute the fuel weight profile from the fuel burn rate. There is coupling because the aircraft weight determines the lift it requires, which determine its drag, required thrust, and resulting fuel consumption, which in turn affects the aircraft weight. The dependency graph in Fig. 6 demonstrates the complexity of the model, which is why a modular implementation in OpenMDAO is necessary given that derivatives are computed using the adjoint method.

The mission optimization problem is posed as follows. The objective function is fuel burn, and the design variables are the cruise Mach number and the parametrized altitude profile variables. The constraints enforce minimum and maximum slopes for each point of the altitude profile, as well as minimum and maximum thrust over the profile. The thrust constraints are aggregated using Kreisselmeier–Steinhauser functionals [26] because they are nonlinear functions of the design variables, and thus explicit enforcement of the constraint at each discretization point would require computing the derivatives of each, which can be very expensive even with the adjoint method.

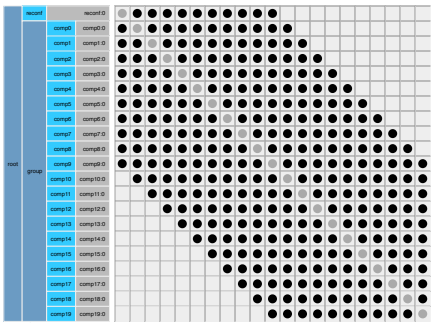
The altitude profile is parametrized using 4th order B-splines with open uniform knot vectors. The number of B-spline control points is n_c, and the number of discretization points is n_p. B-splines have the effect of smoothing gradients for shape or curve optimization, so as a rule of thumb, $n_p = 4n_c$ is used given a desired n_c. The optimization problem is summarized below:

minimize	fuel burn	
with respect to	cruise Mach number	1
	altitude profile	n_c
subject to	minimum slope constraints	n_p
	maximum slope constraints	n_p
	aggregated minimum thrust constraint	1
	aggregated maximum thrust constraint	1.

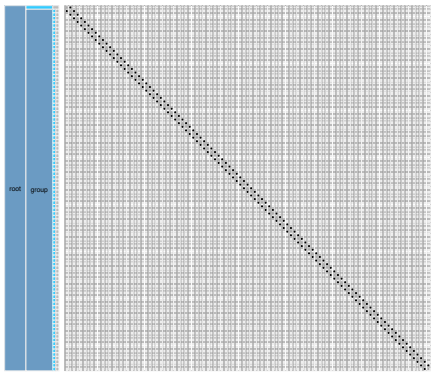
ENABLING REFINEMENT WITH A FIXED OPTIMIZATION PROBLEM The optimization problem is solved using SNOPT [27], a large-scale sparse SQP algorithm for nonlinear constrained optimization problems. As with most existing optimizers, SNOPT does not allow for changing the design variables or constraints. This poses a challenge for performing refinement because changing n_c would change the number of design variables, and changing n_p would



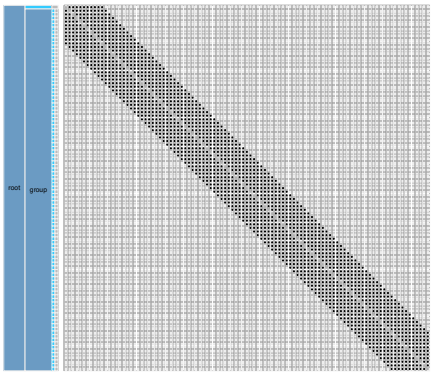
20 components; sparsity parameter = 1



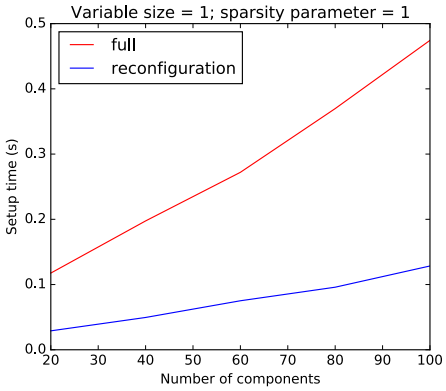
20 components; sparsity parameter = 10

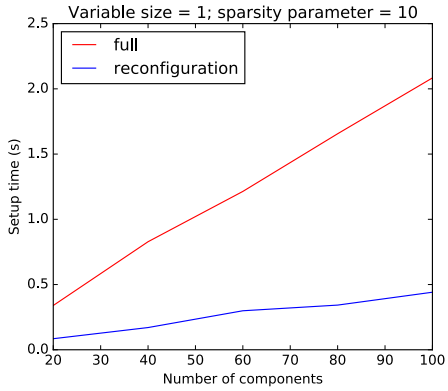


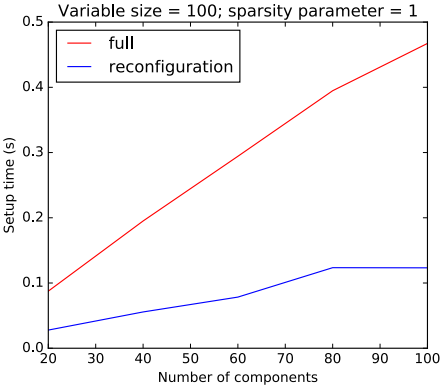
100 components; sparsity parameter = 1



100 components; sparsity parameter = 10







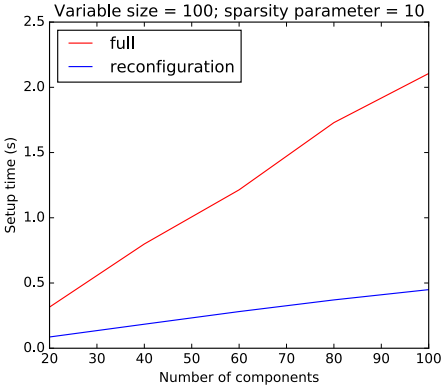


Figure 5: Comparison of full setup and reconfiguration times for varying number of components, degree of sparsity, and variable sizes. Note: 'reconfiguration' includes both reconfiguration setup and update setup. We see that reconfiguration is consistently about 4 times faster, and the gap grows with the size of the model.

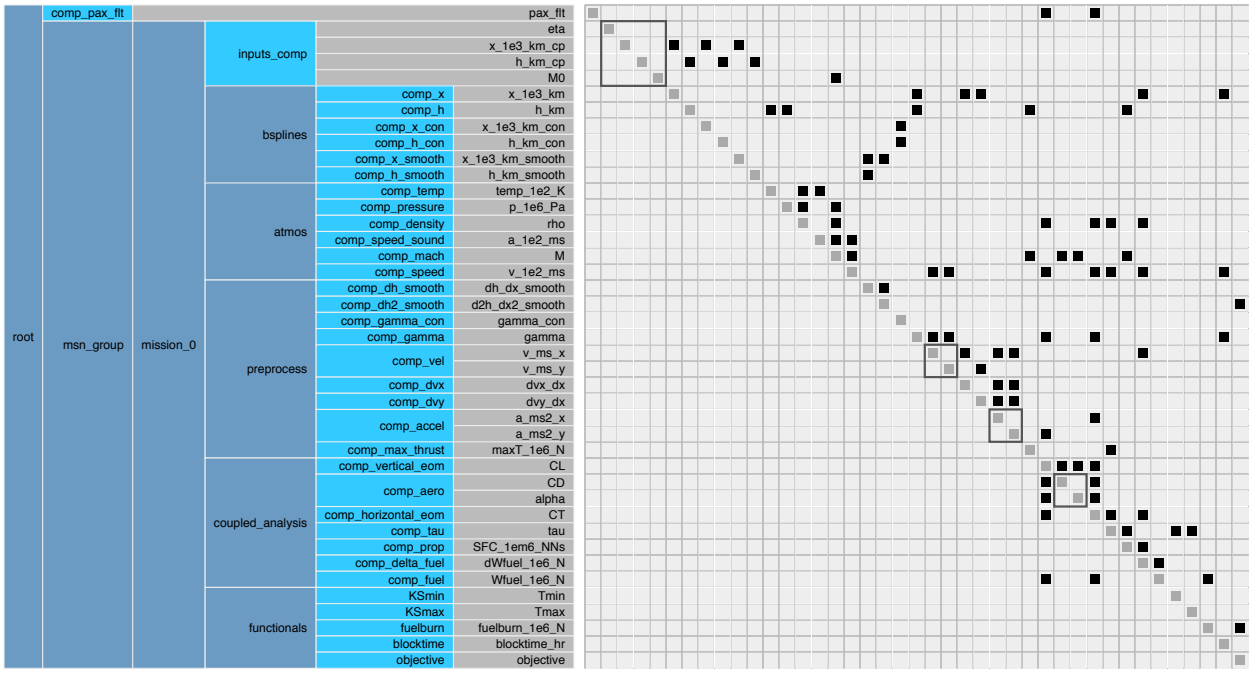


Figure 6: Dependency graph for the mission analysis model.

change the number of constraints. This is a universal problem that is always present in adaptive refinement applications, and it must be overcome to be able to take advantage of reconfigurability in shape or curve optimization.

The solution is a bi-level parametrization and the use of two discretizations of the curve. First, let us introduce two new variables, $n_{\bar{c}}$ and $n_{\bar{p}}$. We shall evaluate a B-spline with n_c control points at $n_{\bar{c}}$ points, then evaluate another B-spline with $n_{\bar{c}}$ control points at n_p and $n_{\bar{p}}$ points. Let $B_{\bar{c}}$ be an $n_{\bar{c}} \times n_c$ matrix representing the evaluation of a B-spline with n_c control points at $n_{\bar{c}}$ uniformly-spaced points. Let B_p and $B_{\bar{p}}$ be $n_p \times n_{\bar{c}}$ and $n_{\bar{p}} \times n_{\bar{c}}$ matrices representing the evaluation of a B-spline with $n_{\bar{c}}$ control points at n_p and $n_{\bar{p}}$ uniformly-spaced points, respectively. If $c \in \mathbb{R}^{n_c}$ is the vector of altitude design variables, the discretized altitude profile used for the mission analysis is $\bar{p}$ and the discretized altitude profile used for evaluating the slope constraints is p, then we have

$$p = B_p B_{\bar{c}} c \tag{6}$$

$$\bar{p} = B_{\bar{p}} B_{\bar{c}} c. \tag{7}$$

With this formulation, $\bar{c}$ can be arbitrarily varied to control the resolution of the parametrization and $\bar{p}$ can be arbirarily varied to control the resolution of the discretization. Meanwhile, c and p can be held fixed to ensure that the number of design variables and the number of constraints do not change. Normally, $\bar{c} < c$, therefore, the Hessian matrix is singular with respect to the n_c control points, which can cause failure in some optimizers. Moreover, it can result in oscillations in the n_c control points despite a smooth profile because of the additional degrees of freedom that have no impact on the altitude profile. To address both of these issuues, a penalty term is added to the objective function to minimize the integral of the second derivative of the altitude profile given by a direct c to p B-spline map. This approach is used to adaptively refine the parametrization and discretization in the mission model, while maintaining a fixed optimization problem.

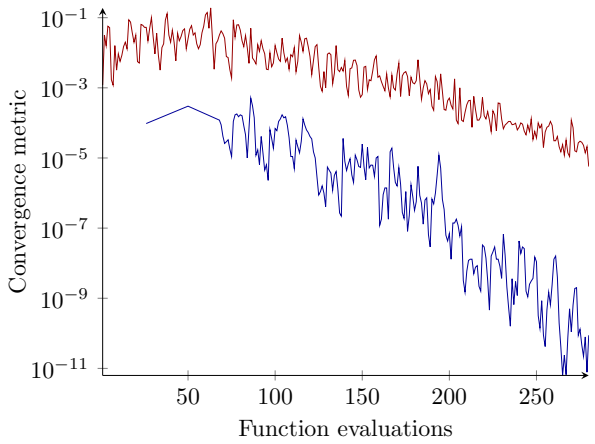
RESULTS Four optimization problems are solved: one with no refinement, and one with one, two, and four refinements. The optimization problem with no refinement does not use the bi-level parametrization, and it maps the n_c control points directly to the n_p discretization points. The four optimization problems are summarized in the following table:

In all 4 problems, refinement occurs when the optimality and feasibility are within a factor of 10 of the optimization convergence tolerances, which are 1e-5 for optimality and 1e-6 for feasibility. Figure 7 shows the effect of refinement on the optimization convergence metrics. As one would expect, there is an immediate increase in optimality and feasibility after refinement, followed by rapid convergence. Optimization with one refinement is faster than optimization

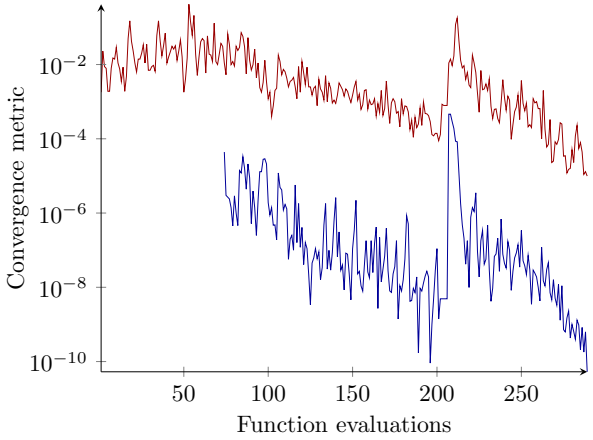
Problem	n_c	$n_{\bar{c}}$	$n_{\bar{p}}$	n_p
No refinement	100	-	-	400
One refinement	100	$40 \rightarrow 100$	$160 \rightarrow 400$	400
Two refinements	100	$20 \rightarrow 60 \rightarrow 100$	$80 \rightarrow 240 \rightarrow 400$	400
Four refinements	100	$20 \rightarrow 40 \rightarrow 60 \rightarrow 80 \rightarrow 100$	$80 \rightarrow 160 \rightarrow 240 \rightarrow 320 \rightarrow 400$	400

with no refinement, despite the fact that they have roughly the same number of function evaluations, because in the former case, most of the iterations are performed with the coarse discretization, which runs faster. Optimization with two refinements takes about 25% more iterations, but this is offset by the cheaper time per evaluation due to the coarser discretizations. The clear trend in this limited set of results is that fewer refinements is more efficient.

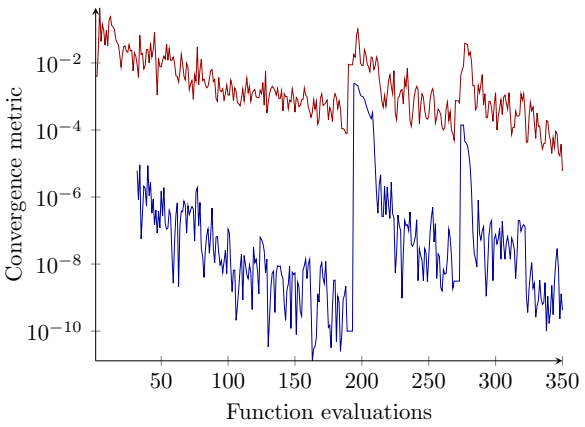
Another benefit for the adaptive refinement approach is evident from Fig 8. The optimization without refinement converges to a solution with oscillations, even though the convergence tolerances are the same across all problems. This is not atypical in fine-resolution curve or shape optimization problems, and it is a product of low sensitivity to the shape near the optimum and having a large number of design variables. However, all profiles found via optimization with refinement do not have oscillations. An advantage of converging first to a partial optimum with a coarser parametrization is that the high-dimensional optimization is given a good starting point that is less likely to have oscillations.



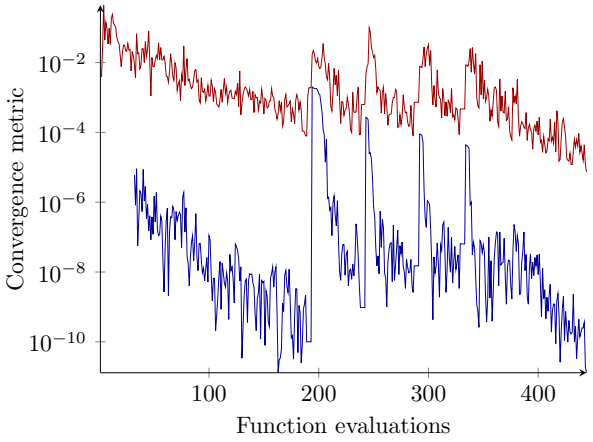
Optimization with no refinement (520 s)



Optimization with one refinement (470 s)

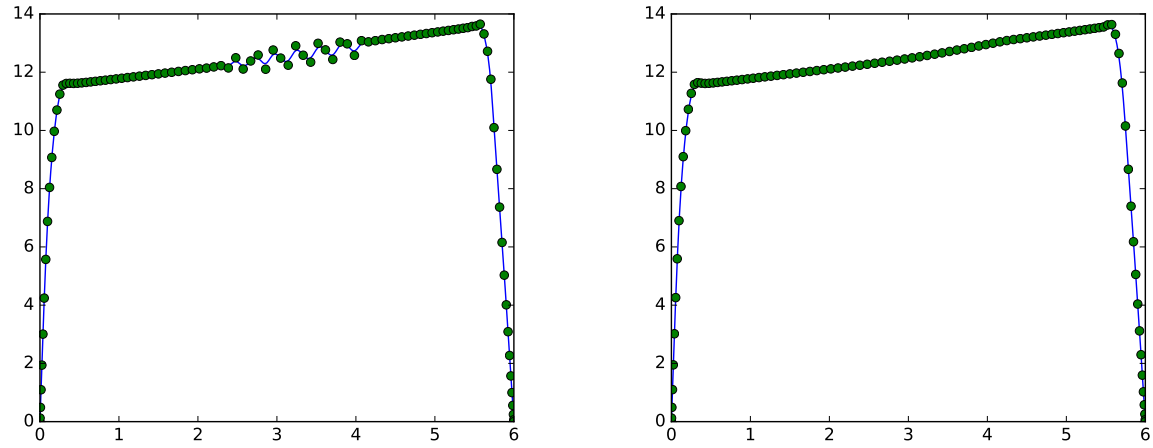


Optimization with two refinements (550 s)



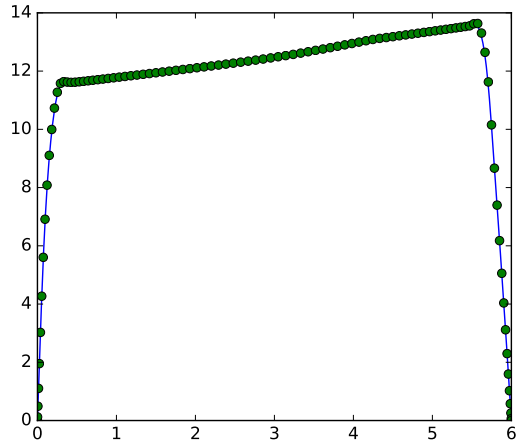
Optimization with four refinements (720 s)

Figure 7: Optimization convergence histories. Feasibility (constraint violation) is shown in blue and optimality (gradient norm of the Lagrangian) is shown in red. Some parts of the feasibility plot appear to be cut off because the constraint violation is 0 and the logarithm is shown. Optimization with one refinement has roughly the same computational cost as the optimization with no refinement.

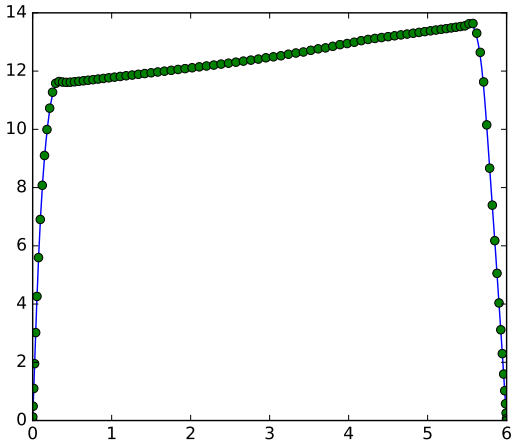


Optimization with no refinement

Optimization with one refinement



Optimization with two refinements



Optimization with four refinements

Figure 8: Optimized altitude profiles with the n_c control points shown in green. All optimizations with refinement converge to profiles without oscillations.

More studies are necessary to make definitive conclusions on when adaptive refinement is beneficial and how many are optimal, but these results show that adaptive refinement can lead to a better result in less time in some cases. A key advantage is that the bi-level parametrization and dual discretization approach enable adaptive refinement to be implemented in curve or shape optimization problems without a specialized optimizer, since the optimization problem is constant through refinement.

V. Conclusion

OpenMDAO is a NASA-developed software framework that facilitates the implementation of multidisciplinary or complex models as a set of components grouped hierarchically, and aids in computing the derivatives needed for optimization. OpenMDAO controls the execution of the components, performs parallel data transfers between components, and computes total derivatives for the full model given partial derivatives for each component. The required data structures and parallel communication patterns are pre-computed during an initial *setup* operation, which can take as long as minutes in very large problems. Therefore, OpenMDAO's design does not naturally permit models to reconfigure, i.e., to dynamically change during execution its variable sizes, processor distribution, component hierarchy, or list of variables.

This paper presented a methodology for enabling reconfiguration in the OpenMDAO framework, and applied it to enable adaptive refinement in an aircraft mission profile optimization problem. This methodology enables any

component or group of components to reconfigure at any time in the middle of execution, even when they are running in parallel with other components in the model, in a simulation involving multiple processors. The algorithms presented in this paper update the affected data structures and parallel communication patterns, and are designed to minimize the amount of operations that must be repeated from the initial setup process. Benchmarking results with a flexible toy problem show that reconfiguration takes less than a quarter of the total setup time, demonstrating that using the new reconfiguration algorithm can be much more efficient than re-initializing an OpenMDAO model with the new settings, even when that is an option. The results are generated using an early version of OpenMDAO v2, and the reconfigurability features from this paper will be included with the upcoming release of OpenMDAO v2.

This paper also provided a demonstration of the reconfiguration feature through an adaptive refinement approach to an aircraft mission profile optimization problem. This problem models the equations of motion for a commercial aircraft with aerodynamics and propulsion surrogate models embedded. In previous work, the continuous altitude profile is optimized using a B-spline curve with a fixed number of control points and evaluation points. In this paper, the reconfigurability feature is used to refine the number of control points and evaluation points—the parametrization and discretization, respectively—resulting in a reduction in optimization time of about 10 %. Moreover, the profile optimized without refinement has oscillations, but all profiles optimized with refinement are free of oscillations because they partially converge first with coarse parametrizations. Since black-box optimizers do not permit changing the set of design variables or constraints during optimization, a novel bi-level parametrization is used to keep the number of design variables fixed, but change the number of true degrees of freedom during refinement. This technique is applicable to all shape and curve optimization problems where an adaptive parametrization is desired with a black-box optimizer.

The directions for future work consist of adaptive solvers and new applications that are enabled by the ability to reconfigure models in OpenMDAO. Adaptive time-stepping applications are now possible since a time-stepping component or group of components can now reconfigure each time they are evaluated to reflect the variable number of time steps. Adaptive mesh refinement and adaptive parametrization refinement applications such as the one presented in this paper are also new problems that can now be solved while still benefiting from the derivative computation and parallel communication features of OpenMDAO. Reconfigurability also enables adaptive solvers that utilize optimal groupings of components and their processor distributions, based on the computation times of components in previous iterations or their coupling structure. The ability to add and remove components during reconfiguration enables applications such as optimization with overset computational fluid dynamics and multi-fidelity optimization in which the model switches between sub-models of varying fidelities. Finally, the method used to optimize the aircraft mission profile using an adaptive parametrization can be applied to any application since it allows the effective parametrization to be refined without changing the number of design variables or constraints seen by the optimizer.

VI. Acknowledges

This work was supported by the NASA ARMD Transformational Tools and Technologies Project. The author would also like to acknowledge and thank Justin Gray and the OpenMDAO development team for insightful discussions and their work on OpenMDAO.

References

[1] Burdette, D. A., Kenway, G. K., and Martins, J., "Performance Evaluation of a Morphing Trailing Edge Using Multipoint Aerostructural Design Optimization," *57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2016, p. 0159.

[2] Hwang, J. T. and Martins, J. R. R. A., "Allocation-mission-design optimization of next-generation aircraft using a parallel computational framework," *57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, San Diego, CA, Jan 2016.

[3] Hwang, J. T., Lee, D. Y., Cutler, J. W., and Martins, J. R. R. A., "Large-Scale Multidisciplinary Optimization of a Small Satellite's Design and Operation," *Journal of Spacecraft and Rockets*, Vol. 51, No. 5, September 2014, pp. 1648–1663. doi:10.2514/1.A32751.

[4] Gray, J., Moore, K. T., and Naylor, B. A., "OpenMDAO: An Open Source Framework for Multidisciplinary Analysis and Optimization," *Proceedings of the 13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference*, Fort Worth, TX, Sept. 2010, AIAA 2010-9101.

[5] Nardin, L., Srensen, K., Hitzel, S., and Tremel, U., "modeFRONTIER, a Framework for the Optimization of Military Aircraft Configurations," *MEGADESIGN and MegaOpt - German Initiatives for Aerodynamic Simulation and Optimization in Aircraft Design*, edited by N. Kroll, D. Schwamborn, K. Becker, H. Rieger, and F. Thiele, Vol. 107 of *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*, Springer Berlin Heidelberg, 2009, pp. 191–205. doi:10.1007/978-3-642-04093-1_14.

[6] Balabanov, V., Charpentier, C., Ghosh, D., Quinn, G., Vanderplaats, G., and Venter, G., "VisualDOC: A Software System for General Purpose Integration and Design Optimization," *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Atlanta, GA, 2002.

[7] Kolonay, R. M. and Sobolewski, M., "Service ORiented Computing EnviRonment (SORCER) for Large Scale, Distributed, Dynamic Fidelity Aeroelastic Analysis," *Optimization, International Forum on Aeroelasticity and Structural Dynamics, IFASD 2011, 26–30*, 2011.

[8] Hwang, J. T., *A modular approach to large-scale design optimization of aerospace systems*, Ph.D. thesis, University of Michigan, 2015.

[9] Gray, J., Hearn, T., Moore, K., Hwang, J. T., Martins, J. R. R. A., and Ning, A., "Automatic Evaluation of Multidisciplinary Derivatives Using a Graph-Based Problem Formulation in OpenMDAO," *Proceedings of the 15th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Atlanta, GA, June 2014. doi:10.2514/6.2014-2042.

[10] Graf, P., Dykes, K., Scott, G., Fields, J., Lunacek, M., Quick, J., and Rethore, P.-E., "Wind Farm Turbine Type and Placement Optimization," *Journal of Physics: Conference Series*, Vol. 753, IOP Publishing, 2016, p. 062004.

[11] Ning, A. and Petch, D., "Integrated design of downwind land-based wind turbines using analytic gradients," *Wind Energy*, 2016.

[12] Barlas, A. K., Tibaldi, C., Zahle, F., and Madsen, H. A., "Aeroelastic Optimization of a 10 MW Wind Turbine Blade with Active Trailing Edge Flaps," *34th Wind Energy Symposium*, 2016, p. 1262.

[13] Leal, P. B., Hartl, D. J., and Bertagne, C. L., "Aero-structural Optimization of Shape Memory Alloy-based Wing Morphing via a Class/Shape Transformation Approach," *23rd AIAA/AHS Adaptive Structures Conference*, 2015, p. 0731.

[14] Cook, L. W., Jarrett, J. P., and Willcox, K. E., "Horsetail Matching for Optimization Under Probabilistic, Interval and Mixed Uncertainties," *19th AIAA Non-Deterministic Approaches Conference*, 2017, p. 0590.

[15] Friedman, S., Ghoreishi, S. F., and Allaire, D. L., "Quantifying the Impact of Different Model Discrepancy Formulations in Coupled Multidisciplinary Systems," *19th AIAA Non-Deterministic Approaches Conference*, 2017, p. 1950.

[16] Kao, J. Y., Hwang, J. T., Martins, J. R. R. A., Gray, J. S., and Moore, K. T., "A modular adjoint approach to aircraft mission analysis and optimization," *56th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Jan 2015. doi:10.2514/6.2015-0136.

[17] Hwang, J. T., Roy, S., Kao, J. Y., Martins, J. R. R. A., and Crossley, W. A., "Simultaneous aircraft allocation and mission optimization using a modular adjoint approach," *56th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Jan 2015. doi:10.2514/6.2015-0900.

[18] Hwang, J. T. and Martins, J. R. R. A., "Parallel allocation-mission optimization of a 128-route network," *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Jun 2015.

[19] Roy, S., Moore, K., Hwang, J. T., Gray, J. S., Crossley, W. A., and Martins, J., "A Mixed Integer Efficient Global Optimization Algorithm for the Simultaneous Aircraft Allocation-Mission-Design Problem," *58th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2017, p. 1305.

[20] Hearn, T. A., Hendricks, E., Chin, J., and Gray, J. S., "Optimization of Turbine Engine Cycle Analysis with Analytic Derivatives," *17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2016, p. 4297.

[21] Gray, J. S., Chin, J., Hearn, T., Hendricks, E. S., Lavelle, T. M., and Martins, J., "Thermodynamics For Gas Turbine Cycles With Analytic Derivatives in OpenMDAO," *57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2016, p. 0669.

[22] Ferguson, S., Siddiqi, A., Lewis, K., and de Weck, O. L., "Flexible and reconfigurable systems: Nomenclature and review," *ASME 2007 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, American Society of Mechanical Engineers, 2007, pp. 249–263.

[23] Sobieszczanski-Sobieski, J., "Sensitivity of Complex, Internally Coupled Systems," *AIAA Journal*, Vol. 28, No. 1, 1990, pp. 153–160. doi:10.2514/3.10366.

[24] Martins, J. R. R. A. and Hwang, J. T., "Review and Unification of Methods for Computing Derivatives of Multidisciplinary Computational Models," *AIAA Journal*, Vol. 51, No. 11, November 2013, pp. 2582–2599. doi:10.2514/1.J052184.

[25] Chauhan, S., Hwang, J., and Martins, J., "Benchmarking approaches for the multidisciplinary analysis of complex systems using a Taylor series-based scalable problem," Submitted to the 12th World Congress on Structural and Multidisciplinary Optimization.

[26] Kreisselmeier, G. and Steinhauser, R., "Systematic Control Design by Optimizing a Vector Performance Index," *International Federation of Active Controls Syposium on Computer-Aided Design of Control Systems, Zurich, Switzerland*, 1979.

[27] Gill, P., Murray, W., and Saunders, M., "SNOPT: An SQP algorithm for large–scale constraint optimization," *SIAM Journal of Optimization*, Vol. 12, No. 4, 2002, pp. 979–1006.