AIAA SciTech Forum
8-12 January 2024, Orlando, FL
AIAA SCITECH 2024 Forum

10.2514/6.2024-1803

Static Task Scheduling for Parallel Execution of Large-Scale Multidisciplinary Design Optimization Models

Mark Z. Sperry*
University of California San Diego, 9500 Gilman Dr, La Jolla, CA 92093

Kavish Kondap†
Francis Parker School, 6501 Linda Vista Rd, San Diego, CA 92111

John T. Hwang‡
University of California San Diego, 9500 Gilman Dr, La Jolla, CA 92093

Multidisciplinary design optimization (MDO) models are built by assembling sub-models that represent varying disciplines and design conditions. Consequently, such models are often computationally expensive, requiring their evaluation in a parallel computing environment to decrease solution time. MDO modeling frameworks, which are software tools to allow construction of MDO models, generally require manual user input to specify where to parallelize their code. However, this process can be tedious if the model is large or complex. In this paper, we propose a method heavily inspired by scheduling theory that aims to fully automate parallelization of MDO models. We use static task scheduling to partition mathematical operations in the model to different processors prior to model execution. Additionally, our approach supports adjoint-based derivative computation to enable the use of gradient-based optimization algorithms. Our results show a significant performance improvement when applying our method to several MDO applications. We believe our approach enables MDO modelers to solve computationally expensive problems in a more efficient manner.

I. Nomenclature

G	=	Computational graph
V	=	Set of all nodes in G
E	=	Set of all edges in G
v	=	Node in V representing a task/operation
e	=	Directed edge in E representing a set of variables relating two tasks/operations
M	=	Number of processors
$\tilde{G}$	=	A clustered graph
$w(v)$	=	Computational cost of node v
$c(e)$	=	Communication cost of edge e

II. Introduction

Multidisciplinary design optimization (MDO) is used to aid in engineering design and analysis. MDO models are built by assembling a variety of sub-models, each often representing different disciplines and design conditions [1]. As a result, MDO models are commonly complex, making them expensive to evaluate. However, due to the use of different disciplines and design conditions, these models often possess inherent parallel properties, allowing concurrent computation of sub-models. Leveraging this parallelism is necessary to solve expensive MDO problems in a reasonable amount of time.

*Ph.D. Student, Department of Mechanical and Aerospace Engineering, AIAA student member
†High School Student, AIAA High School Student Member
‡Assistant Professor, Department of Mechanical and Aerospace Engineering, AIAA senior member

MDO modeling frameworks are software tools that allow developers to construct MDO models, evaluate them, perform sensitivity analysis, and allow integration with optimization algorithms among other tasks. However, existing frameworks do not automatically take advantage of parallelism ingrained in MDO models and require some amount of effort to set up. For example, a widely used MDO framework is OpenMDAO, which enables parallelization through a dedicated message passing template model [2]. GEMSEO uses a similar approach where the users can use models specifically made for parallelization [3]. Further, KADMOS allows parallel evaluations depending on MDO architectures [4]. Consequently, a way to automate parallel execution of MDO model evaluation and derivative computation is necessary.

The parallelization of general programs to minimize computation time has been studied extensively for decades [5]. A common approach to evaluate a numerical model in parallel involves the developers identifying independent sections of their program and manually implementing a parallel programming paradigm such as multithreading or message passing. However, this manual process can be difficult and time-consuming. Automatic parallelization techniques circumvent this issue by enabling sequential code to be executed in parallel with little effort. These methods analyze the structure of the program to identify opportunities for parallelization. However, this is a challenging task as it must handle arbitrary programs and account for performance factors such as communication costs which can introduce overhead to parallel code. A large number of effective heuristic algorithms dedicated to solving this problem are well-known in the field of computer science and engineering as well as operations research but have seen limited use in the field of MDO.

Thus, the objective of this paper is to present a method based on heuristic parallelization algorithms to parallelize MDO models automatically. Our approach performs static task scheduling by distributing operations in the computational graph to different processors before model execution. The method is based on widely-used static scheduling concepts such as graph coarsening and list scheduling. To evaluate the model, the computed schedules are executed in parallel by different processors. Further, necessary data transfer is performed using MPI point-to-point communication. We enable adjoint-based derivative computation by performing reverse-mode differentiation on the parallelized schedules. Finally, we utilize the three-state compiler framework and `CSDL` from Gandarillas et al. to retrieve the computational graph [6].

For the rest of this paper, we will present an overview of the field of parallelization and MDO, our methodology, as well as some results.

III. Background

In this section, we will first review the field of scheduling as well as some widely used algorithms and then introduce modeling frameworks for multidisciplinary design optimization (MDO) as well as the framework we base our work on.

A. Parallelization

Scheduling theory, a branch of operations research, is concerned with building optimal schedules of jobs to maximize or minimize some performance criteria. A well-known problem in this space is that of determining the optimal distribution of a set of jobs to a set of workers and is relevant in applications such as parallel computing, operating systems and manufacturing to name a few [5, 7]. Determining an optimal parallelization scheme falls under this general category of problems and the particular variation we are interested in is that of task parallelism. General workflows can often be represented by a task graph: a directed acyclic graph where nodes represent tasks to perform while directed edges represent precedence relationships between tasks. Specifically, a task that directly succeeds a set of preceding tasks cannot initiate until all preceding tasks have reached completion. Given such a graph, a common problem is the allocation of tasks to different workers to minimize the final time required to complete all tasks. Finding a globally optimal schedule is well known to be NP-complete, meaning that for practical purposes, an approximation algorithm is necessary [5].

In literature, many scheduling algorithms have been proposed across several decades with varying simplifying constraints imposed on the problem. A common simplifying assumption is that all tasks have identical costs. Notable algorithms that assume this constraint are Hu's algorithm [8] and the Coffman-Graham algorithm for special cases [9]. However, for the more realistic case of varying task costs, a wider range of algorithms have been proposed. Among these are greedy algorithms like the HLFET algorithm which prioritizes tasks based on the longest weighted path from the task to an exit [5] and linear programming algorithms such as the one proposed by Levey and Rothvoss [10].

In parallel computing environments, data transfer between nodes or processors may take a non-negligible amount of time. As a consequence, a task may not start immediately after preceding tasks if they are allocated to separate workers, leading to cases where concurrent evaluation may not be optimal. Fortunately, many algorithms take into account this additional complexity. These include clustering algorithms such as edge zeroing which clusters tasks based on

communication cost [5] or METIS' multilevel graph partitioning method for undirected task graphs [11]. Multilevel graph partitioning methods are also used to control the granularity of parallelization to reduce communication costs. The term granularity in this context denotes the degree of low-level parallelism performed [12]. Other approaches such as linear programming have also been proposed such as Kulkarni et al.'s framework which uses linear programming hierarchies [13].

More advanced cases include task duplication, which allows the same tasks to be allocated to multiple workers to reduce communication overhead, and moldable tasks, which are tasks that can be ran with multiple workers to decrease its cost [14]. Another variation is preemptive vs non-preemptive schduling where tasks can be paused mid-execution and continued on a different processor [5]. Further, it has been assumed thus far that workers are equivalent, meaning that completing a task takes the same amount of time regardless of where it has been allocated. However, for cases such as heterogeneous computing environments where workers are a mix of GPUs and CPUs, a task (as well as any transfer of any data) may have different execution speeds depending on if they are scheduled on a GPU or CPU. The HEFT and CPOP algorithms are two widely-used techniques that take heterogeneity into account [15]. Finally, we note that most aforementioned algorithms are for static scheduling, where the task graph is known before execution. However, in some cases task information may only be available during execution, where dynamic scheduling algorithms are needed. Further, dynamic scheduling can often lead to more robust schedules at the cost of overhead performance costs during execution. For example, this is the approach taken by the Dask Python package [16].

The aforementioned algorithms are independent of the parallelization environment in that they can be applied to an arbitrary problem that can be represented by a task graph. For computing, MPI and its implementations are well-used tools that enables programs to be executed on multiple machines. MPI implementations give programmers control of processors for parallel computing by providing operations such as synchronization, collective data transfer and point-to-point data transfer.

Most MPI communication operations can be divided into two categories: non-blocking operations and blocking operations. Blocking operations require synchronization between processors where a processor performing an outward MPI data transfer must wait for any receiving processors to complete the corresponding data reception. Conversely, non-blocking operations allow the processors to progress independently without the need to wait for inter-processor handshakes. The use of non-blocking operations are favored in many cases as it allows computations to overlap with communication. However, several challenges come with the use of non-blocking operations. Firstly, the behavior and performance of non-blocking operations are particularly dependent on the MPI implementation installed and the hardware of the parallel computing environment [17]. For example, a non-blocking communication operation can seemingly unknowingly swap to a blocking operation by the MPI implementation installed if it determines the system or application buffer is not large enough which can significantly degrade performance [17]. Because MPI implementations may manage communication buffers using different strategies, it can be difficult to predict when these transitions might occur. Further, in order to take full advantage of non-blocking communication, asynchronous communication is necessary. As processors post non-blocking communication operations, data transfer does not necessarily occur as they are called. Ideally, the MPI implementation or parallel program continuously monitors for data transfer activity and performs communication operations concurrently during other computations. However, full asynchronous communication is a non-trivial task and often requires work to configure to a parallel computing environment as they are not supported by all MPI implementations [18, 19]. Additionally, in the general case, it is possible that the utilization of non-blocking communication could lead to a significant increase in memory usage compared to the use of their blocking counterparts. This is attributed to the fact that the receiving data must be pre-allocated earlier than that of blocking communications, potentially resulting in increased memory usage when a large number of overlapping non-blocking receives are posted.

B. MDO Modeling

MDO modeling frameworks are software libraries that enable the construction of complex MDO models by assembling a series of sub-models which often consist of different disciplines and design conditions. These frameworks should also feature tools to help define and solve the optimization problem such as a method to interface to optimization algorithms, apply common numerical solvers and perform sensitivity analysis. Further, these types of models are often computationally complex, meaning parallel computing plays an important role in their evaluation. Thus, MDO modeling frameworks should also have the capability to enable parallel computation of their models.

Fortunately, MDO models often have inherent parallelism ingrained in them due to the use of different disciplines and design conditions. For example, Sarojini et al., performed a large scale optimization of NASA's lift plus cruise

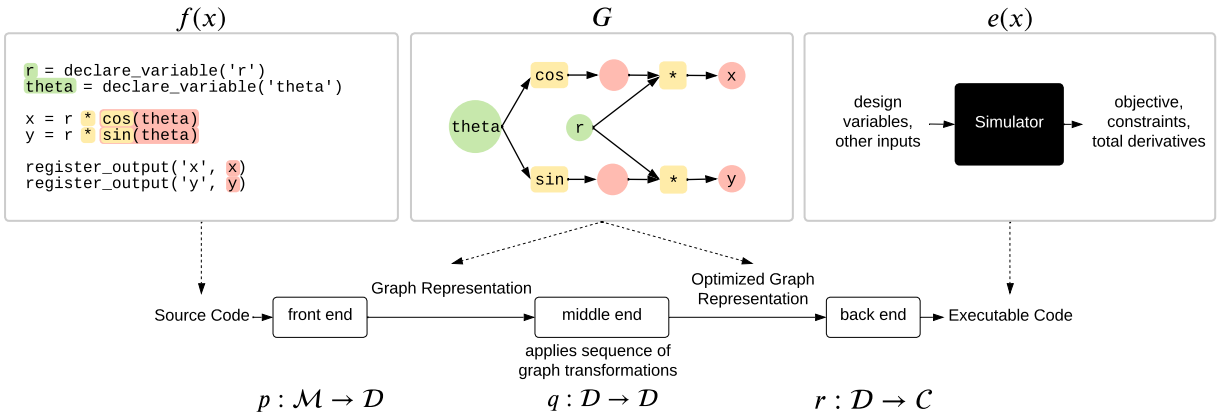

Fig. 1 From Gandarillas et al., the three-stage compiler used for CSDL [6].

vehicle [20] which included several independent design conditions. Ruh et al. also ran a large scale optimization of the same vehicle with 18 independent design conditions [21]. Further, some MDO architectures such as the "all-at-once" formulation allow concurrent evaluations of disciplines regardless of the model [22]. This is because coupling between disciplines may be defined through constraints imposed on the problem as opposed to being ingrained in the model itself [22].

However, as discussed in the introduction, current MDO frameworks require some amount of effort from the user to take advantage of the parallelism ingrained in their model such as manually defining which sub-models are to be parallelized. Further, the use of different disciplines in MDO models often make them heterogeneous in terms of computational complexity as some solvers may be much more expensive to evaluate than others [1]. This puts extra effort on the model developer as not only are they tasked with implementing the parallelization of the model themselves, but identifying how to parallelize it efficiently as well. Additionally, the developer may have to take into account technical factors such as data transfer communication costs and load balancing. Finally, the parallelization must also be enabled for sensitivity analysis as well.

C. Computational System Design Language

We base our work on a recent modeling framework described by Gandarillas et al. [6]. The framework uses a three-stage compiler architecture that allows efficient modeling of MDO models using the Computational System Design Language (CSDL). We will briefly review the framework architecture.

CSDL is a novel modeling language tailored for MDO modeling [6]. It makes up part of the first stage of the three-stage compiler framework and builds a computational graph of the MDO model in the form of a directed acyclical graph. The graph data structure contains all mathematical operators in the MDO model as well as their dependency relationships. The second stage involves graph transformations to enable additional analysis or performance features such as sensitivity analysis, uncertainty quantification, or memory management. Finally, the third stage involves the creation of executables for model evaluation and derivatives. The methodology presented in this paper builds upon the second stage of the aforementioned framework.

IV. Methodology

In this paper, we introduce an automatic static scheduling strategy for the parallel execution of multidisciplinary design optimization (MDO) models. Our method draws inspiration from graph clustering and list scheduling, as detailed in section III.A. We utilize the three-stage compiler framework described in section III.C to retrieve a computational graph of the MDO model defined using CSDL with which we can apply our algorithm. The third stage of the compiler is used to execute the parallelized model using MPI operations.

We enable gradient-based optimization by automatically performing adjoint-based sensitivity analysis on the parallelized schedule. We also use blocking point-to-point MPI communication operations as opposed to their non-blocking counterparts, motivated by the issues discussed in section III.A. Further, we aim to leverage the inherent

parallelism ingrained in MDO models and thus decide use task parallelism, where the directed acyclic graph (DAG), which represents a task graph we call $G = (V, E)$, is simply the computational graph obtained from the three-stage compiler framework [6] where vertices v represent tasks (mathematical operations) and edges e represent dependency relationships between them (variables).

We define our task scheduling problem as determining an ordered sequence of tasks v for each processor such that the partial ordering of G is preserved to minimize the makespan, or the completion time of the last task. We let M denote the number of processors and thus the algorithm will compute M schedules. Further, all tasks in V must occur in a schedule on a processor and a task v_i preceding task v_j must complete before v_j starts. Our task scheduling problem assumes that every task $v \in V$ has a computational cost $w(v)$, representing how long it takes to execute v on a single processor. Further, each directed edge $e \in E$ has a communication cost $c(e)$, representing the time it takes to transfer the variable(s) e across processors if need be. We also assume that w and c do not depend on the processor they are scheduled on, thus, the processors are homogeneous. Finally, we assume that tasks are not moldable, non-preemptive, and cannot be duplicated across machines.

It is important to note that standard scheduling techniques discussed in section III.A assume full asynchronous data transfer. Thus, applying existing scheduling algorithms using blocking MPI communication could potentially lead to a large loss in performance. This is because many scheduling algorithms assume data transfer can take place as needed, regardless of the status of the sending or receiving processors. However, blocking data transfer generally require both sending and receiving processors to have posted their respective MPI calls before communication occurs. Thus, standard greedy list scheduling can lead to inefficiencies as they do not consider the effects of synchronized data transfers on other processors. Expanding the scheduling problem to consider the coordination of synchronous data transfer is possible. However, this adds complexity to the problem as both sending and receiving processors must be idle in order to schedule a data transfer. The aforementioned behavior can be seen in figure 2 which shows an application of a standard greedy list scheduling algorithm applied to a heterogenous task graph using blocking and nonblocking MPI communication operations. Our approach must be able to handle the issues that arise from trying to schedule blocking data transfer without significantly compromising the time complexity of the algorithm.

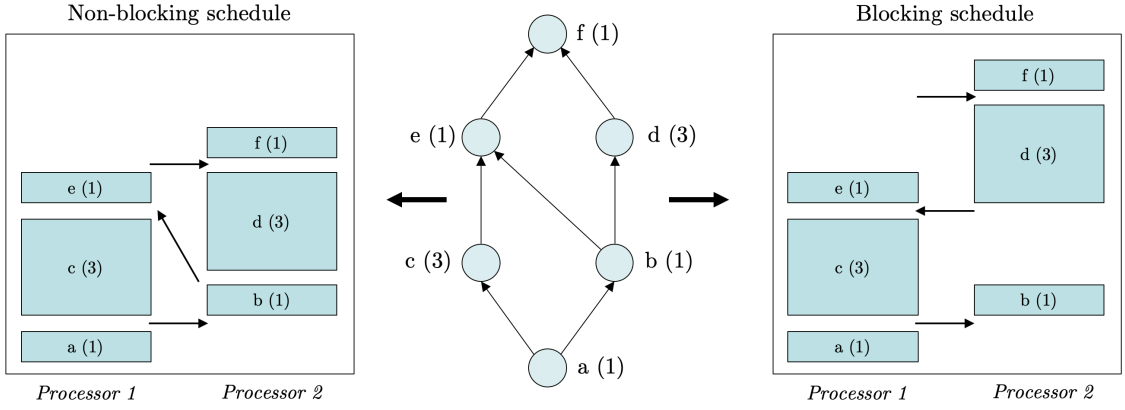

Fig. 2 A comparison of a schedule executed using non-blocking and blocking data transfer if a standard greedy list scheduling algorithm is used. Blocking data transfer requires both sending and receiving processors to be idle when communication occurs. The value in the parentheses next to each task represents the computational cost of that task.

We now introduce our approach which consists of two major steps: graph coarsening (section IV.A) followed by task scheduling (section IV.B). The first step attempts to homogenize the tasks in the computational graph in terms of computational costs. This is done by applying a DAG to DAG coarsening procedure that iteratively merges nodes in G. We then apply a scheduling algorithm that allocates nodes in the coarsened task graph to processors in order to create the final schedule. Finally, we aim for our algorithm to have a computational cost with near linear scaling in terms of the number of nodes in G.

A. Graph Coarsening

The main motivation behind graph coarsening is to mitigate the negative impact on performance from the use of blocking MPI operations as discussed in the previous section. Figure 3 shows a scheduling of the task graph from figure 2 if all tasks in the graph are homogeneous in terms of computational cost. A homogeneous computational graph can reduce idle time significantly as processors can start and finish tasks in unison. However, as discussed earlier, MDO models are often inherently heterogeneous. Thus, we attempt to address this issue by utilizing a graph coarsening approach to homogenize the task graph. Fortunately, graph coarsening has additional benefits such as reducing the amount of data transfer as well as reducing the dimensionality of the problem for the scheduling procedure.

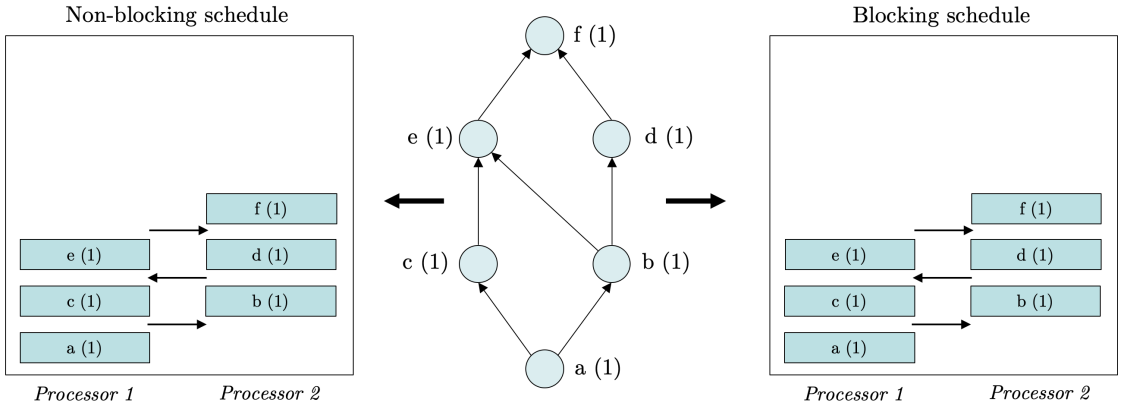

Fig. 3 A comparison of scheduling the computational graph from figure 2 if the computational graph is homogeneous in terms of costs. The blocking schedule is as efficient as the non-blocking schedule.

Graph coarsening techniques are commonly used in the automatic parallelization of task graphs as discussed in section III.A. The procedure involves partitioning nodes in the graph into disjoint sets and is used in a diverse array of applications such as network analysis, scheduling and machine learning [23] [24]. The choice of objective and constraints for the clustering varies by application, such as keeping the cost of each cluster below a specific value or minimizing the number of edges between clusters. It is also important to note that graph coarsening methods commonly found in literature are often designed for undirected graphs as opposed to DAGs.

We now define a coarsened graph. Let $\tilde{G} = (\tilde{V}, \tilde{E})$. A node $\tilde{v} \in \tilde{V}$ is a cluster containing a subgraph of G. Hence, the subgraph of $\tilde{v} \in \tilde{V}$ is $\tilde{v}_s = (V(\tilde{v}), E(\tilde{v}))$ where $V(\tilde{v}) \subseteq V$ and $E(\tilde{v}) \subseteq E$. An edge $\tilde{e} \in \tilde{E}$ connects two vertices $\tilde{v_i}, \tilde{v_j} \in \tilde{V}$ and contains the set of all edges $\tilde{e}_s \subseteq E$ between the nodes $V(\tilde{v_i})$ and $V(\tilde{v_j})$. Further, $\tilde{G}_i = (\tilde{V}_i, \tilde{E}_i)$, where i is an integer, denotes a coarsened graph as a result of clustering $\tilde{G}_{i-1}$. Finally, we define clustered node and edge costs as

$$w(\tilde{v}) = \sum_{v \in V(\tilde{v})} w(v), \qquad c(\tilde{e}) = \sum_{e \in \tilde{e}_s} c(e), \tag{1}$$

which represents the total computation time of executing subgraph $\tilde{v}_s$ and the total data transfer cost of all variables in $\tilde{e}_s$. We also add a granularity constraint that $w(\tilde{v}) \leq K(G)\ \forall \tilde{v} \in \tilde{V}$ where $K(G)$ is a tuning parameter limiting the maximum computational cost of a cluster $\tilde{v}$.

For the coarsening procedure in this paper, we aim to partition nodes in the graph G such that the nodes in G are similar in size in terms of cost. At the same time, we aim to preserve the DAG structure of the graph, ensuring that the final coarsened graph $\tilde{G}_n$, where n is the total number of coarsening iterations maintains the acyclic property from the original uncoarsened graph G. Preserving the DAG property allows standard task scheduling procedures on $\tilde{G}_n$. However, this DAG preservation constraint increases the complexity of the coarsening process and attention must be made to prevent nodes clustering together that leads to cycles. Herrmann et al. developed a multilevel partitioning scheme ensuring the DAG property is maintained throughout the entire scheduling procedure that includes a graph coarsening algorithm for clustering DAGs to DAGs [25].

Our approach uses the fundamental ideas introduced by Herrmann et al., namely the rule for clustering nodes together to prevent cycles. We briefly introduce Theorem 4.2 from Herrmann et al. [25]. They determined that in order to merge two clusters in a coarsened graph, the rule

$$|\text{top}[u] - \text{top}[v]| \leq 1 \quad \forall u, v \in C_i \tag{2}$$

must be satisfied where C_i is any cluster of nodes in a clustered graph and top$[v]$ denotes the top level value of a node v [25]. The top level value of a node v is the longest path length from a source of G to v where a path length is the number of vertices in a path that connects two nodes [25]. Importantly, the top level value for all nodes can be computed in linear time. Additionally, any edges between two clusters must satisfy top$[u] \neq$ top$[v] - 1$ for all $u \in C_i$ and $v \in C_j$ where C_j and C_i are two different clusters [25]. If the those conditions are met, Herrmann et al. proved that the clustered directed graph is acyclic [25].

Thus, the clustering algorithm consists of visiting all nodes in $\tilde{V}$ and checking their neighbors to see if the aforementioned conditions are met [25]. When visiting a node $\tilde{v} \in \tilde{V}$ during the traversal, the algorithm selects a suitable neighboring node and includes it in the cluster associated with $\tilde{v}$. When selecting a suitable node $\tilde{u}$ to cluster, the algorithm also checks to make sure that clustering that node will not exceed the predefined threshold by checking $w(\tilde{u}) + w(\tilde{v}) \leq K(G)$. Thus, no cluster cost will exceed $K(G)$. This computation can also be done in linear time.

Algorithm 1 Coarsen Graph

```
Set G̃_0 = G
for i = 0, i++, i < n do
    Compute top[ṽ] ∀ṽ ∈ Ṽ_i
    Set G̃_{i+1} = G̃_i
    for ṽ ∈ Ṽ_{i+1} do
        Find all valid neighbors of ṽ using top and K(G)
        Choose best neighbor ũ
        Cluster ũ and ṽ
    end for
end for
```

The full coarsening procedure is performed by iteratively performing the coarsening procedure until the graph coarsening is complete. This is shown in algorithm 1 and outputs the final coarsened graph $\tilde{G}_n$. The algorithm is initialized by nominally clustering G to itself such that $V(\tilde{v_i}) = \{v_i\}$ and $E(\tilde{v_i}) = \emptyset$ for all v_i in V to build $\tilde{G}_0$. Generally, the larger G is, the larger n needs to be to have a suitable clustering. In practice, we observed that the coarsening procedure is usually sufficient after around 10 to 15 iterations for large graphs.

B. Scheduling

The main job of the scheduling procedure is to distribute tasks to processors. This is done by using the coarsened graph $\tilde{G}_n$ where a clustered task $\tilde{v} \in \tilde{V}_n$ contains a set of lower level tasks from the original task graph G. We treat the clustered task $\tilde{v}$ as a single task for the purposes of this algorithm meaning all tasks in $V(\tilde{v})$ are to be executed on the processor that they are eventually assigned to. Thus, no inter-processor data transfer is required for edges in $E(\tilde{v})$ and the order in which the tasks in $V(\tilde{v})$ are executed do not matter as long as the satisfy their dependencies.

As mentioned earlier, the algorithm generates a sequence of tasks for each processor. Each sequence represents the order in which each processor executes each task. In essence, the scheduling procedure allocates a clustered task $\tilde{v}$ to a processor m where $1 \leq m \leq M$ by iterating through the nodes of $\tilde{G}_n$ in topological order.

When building these schedules, the algorithm must also be aware of a few keys factors. Firstly, as was discussed in section III.A, performing MPI communication operations introduces overhead costs regardless of whether blocking or non-blocking data transfer is performed. Hence, if data transfer exists between tasks, there are scenarios where executing them sequentially on a single processor instead of distributing them across processors can be more efficient as it avoids incurring communication overhead costs.

Also, a well-known pitfall in MPI programming it deadlock. Deadlock scenarios arise when multiple processors simultaneously find themselves waiting for the completion of tasks from one another. In the classic MPI deadlock scenario with two processors, processor A waits for data from processor B while processor B waits for data from processor A. This results in the program locking up as neither processor can progress. The scheduling algorithm must consider this to prevent such situations.

Finally, the algorithm must be aware of idle waiting times incurred by data transfer. As processors perform blocking data transfer, the algorithm must consider their effects on the other processors' schedules. This concept was illustrated in figure 2 as it can be seen that data transfer can potentially directly cause a processor to be idle. Although the graph

clustering pre-processing step attempts to naturally address this issue, the scheduling algorithm must also take this into account.

The algorithm to perform the scheduling is heavily inspired by standard list scheduling algorithms by iteratively traversing $\tilde{G}$ in topological order. During each iteration, the algorithm identifies the least occupied processor, and appends a task to that processor's schedule. The issue of characterizing the effects of performing data transfers across processors and the concern of deadlock is addressed by designating specific points in the schedule where all processors synchronize. These designated synchronization points serve as intervals where all processors perform any necessary data transfers, ensuring that all processors reach the same state upon completion of the data transfers. By allocating these points to the schedules, the adverse effects of issuing data transfers on other processors are much more predictable as all processors are prepared to send and receive data at the same time.

Algorithm 2 shows a high-level overview of the procedure. Following the standard approach of list scheduling algorithms, we maintain a "ready set", denoted as A, comprising of tasks where all the preceding tasks have already been scheduled on a processor. However, unlike standard list scheduling algorithms, A is not updated after every iteration. Instead, the algorithm attempts to schedule all currently ready tasks to a processor first. Once all processors are occupied or there are no more ready tasks, we designate a synchronization point across all processors and update A before proceeding with the algorithm.

Algorithm 2 Schedule Tasks

```
Set A as the set of all entry tasks in G̃_n
while A do
    for ṽ ∈ Ṽ_n in A do
        Find least occupied processor m to schedule ṽ or break loop if all processors has been assigned task
        Add ṽ to schedule m
        Update processor m availability
    end for
    Add a data synchronization point across all processors
    Delete scheduled tasks from A and update A with newly available tasks
end while
```

At every outer iteration (the outermost loop in algorithm 2), the algorithm loops over the set of all currently available tasks. Note that these tasks are guaranteed to require no data transfer between one another as all their predecessors have been scheduled by this point. During each loop, the algorithm attempts to allocate the available task $\tilde{v}$ to an optimal processor. Using the least occupied processor to schedule $\tilde{v}$, which corresponds to the processor with the earliest latest task completion time, is a standard list scheduling heuristic to determine the optimal processor. We also take into account communication costs in the case of tiebreakers. To find the optimal processor, we maintain a record of the completion time of the latest task scheduled on each processor m, denoted as $T[m]$. Identifying the processor with the earliest final task completion time is done by

$$m_{earliest} = \underset{1 \le m \le M}{\arg\min}(T[m] + C(m, \tilde{v})), \tag{3}$$

where C is an adjustment factor that takes into account inter-processor communication costs. C intends to capture the negative effects of communication costs depending on which processors the predecessors of $\tilde{v}$ were scheduled to. This is done by the heuristic

$$C(m, \tilde{v}) = \sum_{\tilde{e} \in P(\tilde{v}, m)} c(\tilde{e}), \tag{4}$$

where $P(\tilde{v}, m)$ returns the the set of edges $(\tilde{u}, \tilde{v})$ for all predecessor tasks $\tilde{u}$ not scheduled on m. Updating the schedule of the selected processor, $m_{earliest}$, involves simply appending $\tilde{v}$ to schedule of $m_{earliest}$ and updating the completion time by $T[m_{earliest}] = T[m_{earliest}] + w(\tilde{v})$. Once the loop terminates or all processors have had a task assigned to them at the current outer iteration, the scheduling process is temporarily paused. At this point, a synchronization point is added to all schedules, where we assume all processors require data transfer. The latest task completion time for all processors are then updated by

$$T[m] = \max_{1 \le m \le M}(T[m]). \tag{5}$$

Once all tasks in $\tilde{G}_n$ have been scheduled, the scheduling procedure is complete as all tasks in $\tilde{G}_n$ have been assigned to a processor. Further, the running time of the scheduling procedure generally scales linearly with the number of nodes in $\tilde{G}$ as nodes are visited once and scheduled greedily. The evaluations of the schedules are handled by the backend of the three-stage compiler framework denoted in section III.C. Each processor initializes a separate backend instance containing their respective schedules. Following this, each processor's backend instance iterates through their schedule, sequentially executing each task. Finally, data transfer instructions are handled by MPI send and receive operations as needed.

C. Node and Edge Costs

Thus far, the assumption is that the node costs $w(v)\ \forall v \in V$ and edge costs $c(e)\ \forall e \in E$ are known. However, the precise computational cost of each task is not known a priori to execution. Moreover, the computational cost may vary significantly from one run to another. This inability to dynamically adjust the schedule on a run by run basis is one of the drawbacks of static scheduling algorithms. One valid strategy to address this issue is to measure the computational cost of each task once during the first evaluation [26]. However, it is beneficial to have an a priori estimate of the computational costs known beforehand, to avoid the need to wait for a forward evaluation to run.

Our approach to estimate the costs of all tasks offline before executing the model leverages CSDL's predefined standard math library [6]. Because the majority of nodes in the computational graph defined by a CSDL multidisiplinary design optimization (MDO) model consist of operations from CSDL's standard math library, a function to estimate the costs can be pre-built for each operation class. This is done by building a surrogate model for each operation class where the inputs to each surrogate model are selected on an operation by operation basis. Training the surrogate model for an operation consists of evaluating and timing the execution time over a range of inputs for that operation.

We illustrate by using CSDL's addition operation as an example. The addition operation class relies on a single parameter for predicting the computational cost of each addition instance: the number of elements of an input tensor variable. Offline, the CSDL addition operation is evaluated multiple times over a wide range of parameter values, during which the execution times are recorded. Subsequently, the gathered data is then used to train a surrogate model which is then saved offline and can be loaded in as needed. We use the Surrogate Modeling Toolbox (SMT), to train, evaluate and save the surrogate models [27]. An example of a surrogate model built for the matrix-vector operation is seen in figure 4.

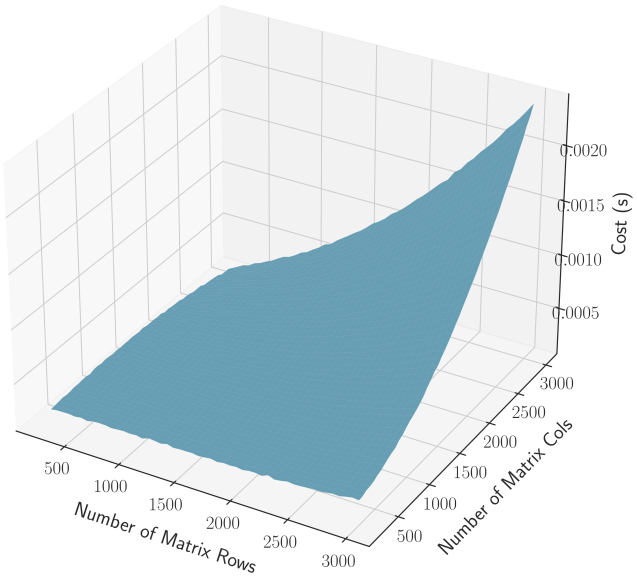

Fig. 4 The estimated computational cost of the matrix-vector product operation in the CSDL standard library using a surrogate model with two parameters.

The computational cost estimation workflow for a CSDL model before the graph coarsening step is shown in figure

5. Because of the three-stage compiler framework, all surrogate modeling parameters for standard operations are known when the MDO model is defined, before execution. Thus, we have a pre-processing procedure where we load in the relevant trained surrogate models and evaluate them to obtain $w(v)\ \forall v \in V$. We also accommodate for varying performance across different hardware by providing an interface to either recompute the surrogate model training data or use an adjustment factor to calibrate the data for a specific device.

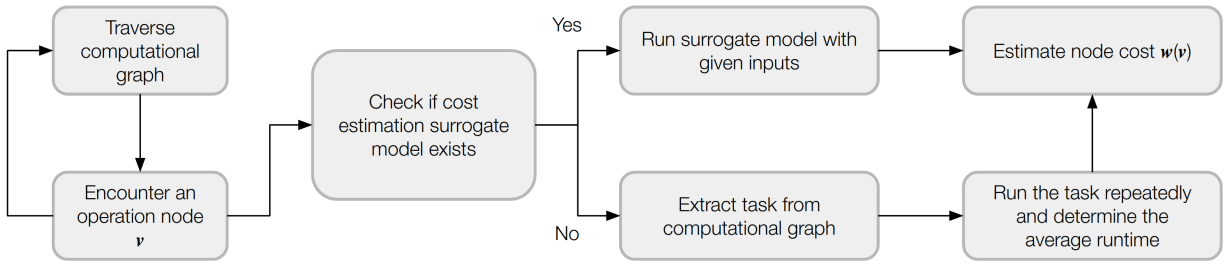

Fig. 5 The workflow to assign estimated computation costs $w(v)$ for all operations v in the computational graph. Estimations of computational costs for operations without surrogate models stored offline are approximated by measuring them right before model evaluation.

However, we note that the computational cost of certain CSDL operation classes cannot be predicted by a surrogate model. For example, some standard operations such as the einsum operation require abstract parameters such as the einstein summation notation defined by the CSDL user. Parameters such as these make it difficult to build robust surrogate models. Further, implicit function nodes which involve a nonlinear solver are also difficult to predict as the number of iterations required for convergence are not known in advance. Finally, custom operation nodes where the evaluation is performed by a user-defined function cannot have their computational costs estimated before running. Instead of estimating the computational costs of the aforementioned operations offline, they are estimated by measuring their evaluation time right before model execution.

For estimating data transfer costs $c(e)$, a linear function of tensor size is used. In practice, we observed that these communication costs are usually relatively low compared to $w(v)$ unless the tensors are unusually large.

D. Sensitivity Analysis

As discussed in section III.B, multidisciplinary design optimization (MDO) problems are usually performed using gradient-based optimization. Thus, in addition to evaluating the MDO model, the ability to automatically compute derivatives of the model is also necessary. Further, for MDO problems, adjoint-based derivative computation techniques are commonly used due to their efficiency in problems with a large number of independent variables and a small number of dependent variables [28]. This is often the case for large scale MDO problems. Fortunately, the backend of the three-stage compiler framework introduced in section III.C has the ability to perform adjoint-based derivative computation [29].

Section IV.A and IV.B overviews an approach to parallelize a forward model evaluation. Ideally, the backend is able to leverage parallelism for adjoint-based derivative computation as well. However, this poses a challenge as adjoint-based sensitivity analysis techniques such as computing vector-Jacobian products and performing reverse-mode differentiation to compute total derivatives traverses the computational graph differently than that of the forward model evaluation. Thus, the schedule computed from section IV.B cannot be used directly.

Because adjoint-based derivative computation can be performed by backtracking through the computational graph, we can perform reverse-mode differentiation of the parallelized MDO model by reversing the parallelized schedule. This is done by evaluating

$$\frac{df}{dp} = \sum_{s \in S(v)} \bar{s} \frac{\partial s}{\partial p}, \tag{6}$$

for every operation v in the computational graph in reverse order where $S(v)$ is the set of output variables of v, p is an input to v, f is the dependent variable of interest in the MDO model, and $\bar{s}$ is df/ds [29]. In the general case, $\bar{s}$ does not

have to be df/ds such as when computing vector-Jacobian products. The matrix/vector multiplication function on the right hand side of the equation above is evaluated by an adjoint function defined for every operation.

This procedure is automatically done by the backend, thus the only remaining task is defining the adjoint function for data transfer operations, specifically, MPI send and receive operations. For an MPI send operation of variable x to processor m, the adjoint is simply an MPI receive operation of $\bar{x}_m$ where the computation of $\bar{x}_m$ is performed on the receiving processor. Likewise, the adjoint of the MPI receive operation of variable x on processor m is an MPI send operation of $\bar{x}_m$ where the computation of $\bar{x}_m$ is performed on processor m. Because every MPI send operation is paired with a corresponding MPI receive operation on another processor, their adjoint functions serve a means of transferring accumulated derivatives across processors in the opposite direction of the original evaluation. Figure 6 highlights the parallelized derivative computation procedure for a parallelized schedule using adjoint MPI operations.

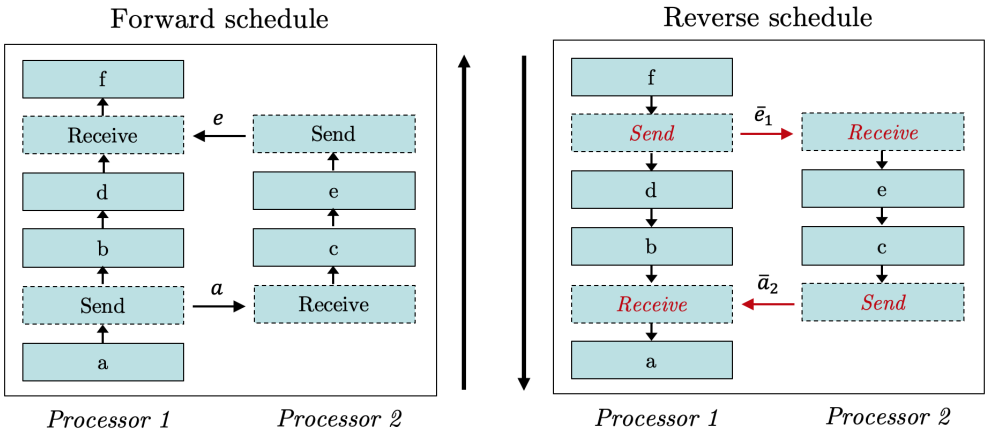

Fig. 6 A sample schedule showing the order of tasks executed and the flow of data in the forward evaluation compared to the reverse derivative computation.

The aforementioned technique allows parallelization of derivative computation without the need to perform additional scheduling procedures. Note that to evaluate total derivatives, $\bar{x}$ as used thus far can be replaced by propagated derivatives df/dx instead. Finally, it is important to note that the computational cost to evaluate an operation's adjoint function may differ substantially than their respective forward evaluation. For example, in order to propagate derivatives through a non-linear solver, a linear solver can be used [29]. However, the computational cost of solving a linear system is not equal to that of a non-linear solver. Therefore, an efficient parallel schedule for the forward evaluation of an MDO model may not necessarily be efficient for its derivative calculation.

V. Numerical Experiments

In this section, we present numerical results from several practical multidisciplinary design optimization (MDO) problems that showcase the performance gain from our automatic parallelization of the methods described in section IV. The collection of problems are real-world examples that have been successfully built using `CSDL` and solved for various research projects. We showcase the performance gain on three problems: a rotor design tool by Ruh and Hwang [30] using blade element momentum theory analyzing 8 rotors, an unsteady vortex lattice method solver, and a large-scale MDO problem of NASA's lift-plus-cruise concept [21].

In the following subsections, we present the computation times of model evaluation and gradient evaluation for each application as we increase the number of processors. Then, we analyze the results by taking into account the methodology presented in this paper as well as specific characteristics of the MDO model. We use OpenMPI as the MPI implementation for all applications [31].

A. Blade Element Momentum

We apply our automatic scheduling algorithm to a rotor design tool based on blade element momentum (BEM) theory from Ruh and Hwang [30]. The BEM tool is already implemented in `CSDL`, allowing us to easily apply our method. For our tests, we defined a model with 8 independent BEM solver instances, simulating an optimization

problem of an aircraft with 8 rotors. In total, the computational graph consistsed of over 2,800 operations, where the average tensor variable had 165 elements. The model was constructed without any consideration of parallelization and contains no information on how to parallelize the model. No effort was required to execute the model in parallel apart from running the program with OpenMPI and specifying the number of processors. Finally, we note that analyzing one rotor required a relatively expensive nonlinear solver that took up a large percentage of the execution time.

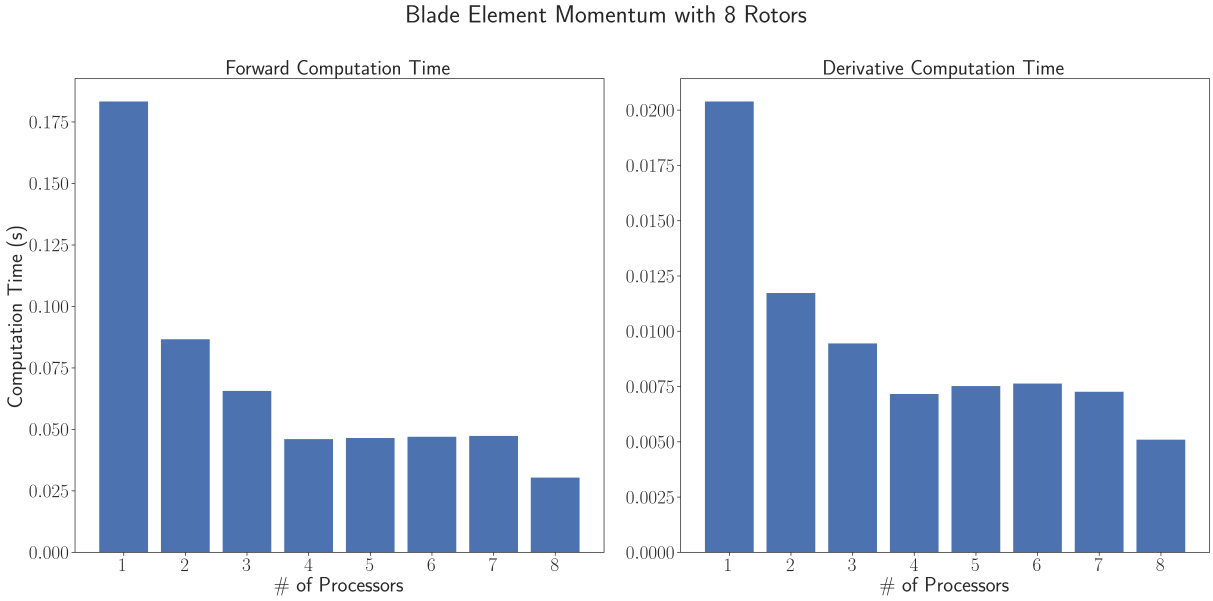

Fig. 7 Model evaluation and derivative computation times of a blade element momentum (BEM) model with 8 rotors using multiple processors. Each rotor analysis contains an expensive nonlinear solver operation. The computational cost with 4 through 7 processors is similar as at least one processor must evaluate two BEM solvers for all cases.

We evaluated the model specifying 1 through 8 processors and achieved significant performance improvements as shown in figure 7. Because the model analyzes 8 independent rotors each with a relatively expensive nonlinear solver, an 8 times speedup is a theoretical upper bound for the optimal speedup. In practice, around a 6 times speedup with 8 processors was observed using our approach, a significant improvement over the single processor case considering the complete automation of parallelization. The parallel schedule of this problem using our approach with 1, 4 and 8 processors can be seen in figure 8. We can see that the algorithm successfully allocates tasks to different processors with a mostly even distribution. Despite occasional idle intervals in the parallel schedules, the majority of the time was spent performing concurrent calculations. Finally, we also observed a speedup for the derivative computation as well although not as optimal as the forward evaluation.

There are multiple contributing factors as to why the maximum speedup experienced is around 6 despite there being 8 independent rotors. Firstly, as expected, the schedule is suboptimal. Notably, there are intervals in the schedule where processors are idle, leading to inefficiencies. This is an expected byproduct of attempting to schedule the model using blocking communication and is an area of possible improvement. Further, there are some calculations of the model and some startup costs of the backend that are not parallelizable, leading to a slower than expected speedup.

B. Unsteady Vortex Lattice Method

We also tested our approach on a model representing a single iteration of an unsteady vortex lattice method solver of a coupled propeller-wing system. This problem is of a much larger scale than that of the BEM solver with over 37,500 nodes in the computational graph where the average tensor variable size has over 11,000 elements. This model was also built without parallelization in mind. However, in contrast to the previous application where the model contained easily identifiable parallelizable sections (8 near-identical independent rotor analysis models), this model contained no such regions. The complexity of models like these showcases the usefulness of the automatic scheduling scheme.

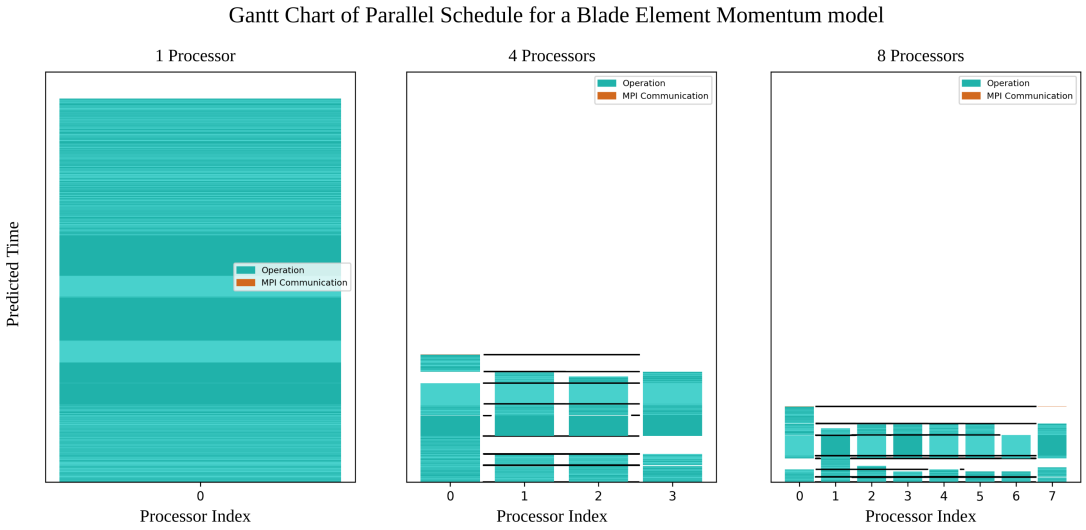

Fig. 8 Gantt chart of the parallelized schedule for the BEM model analyzing 8 rotors with 1,4, and 8 processors. Each cyan block consists of a task where the height of the block represents the estimated computational cost of that task using our approach discussed in section IV.C. One column represents a schedule for one processor where tasks are scheduled in order from bottom to top as described in section IV.B. The black lines represent data transfer across processors and white areas represent idle intervals in the schedule.

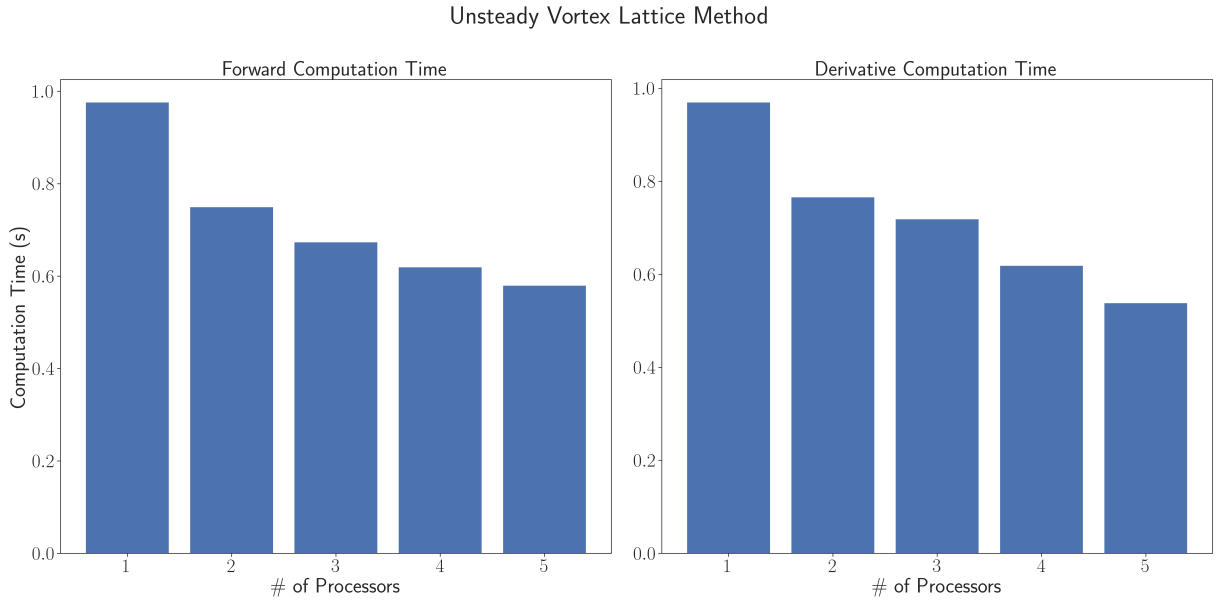

Fig. 9 Model evaluation and derivative computation times of one iteration of an unsteady vortex lattice method solver parallelized across 1 through 5 processors.

Figure 9 shows the performance improvements through parallelizing across 1 through 5 processors. We observe a maximum performance improvement of around 40% for both the forward evaluation and the derivative computation. It is believed that the parallelization comes from the fact that there are multiple surfaces and thus the computations can be parallelization between them.

C. NASA lift-plus-cruise

Finally, we studied a large-scale optimization of NASA's lift-plus-cruise concept based on Ruh et al. [21] and is the largest problem thus far with over 200,000 nodes in the computational graph. The MDO model consists of low and mid-fidelity physics-based solvers including models for rotor and wing aerodynamics, structural analysis, aeroacoustics, motor analysis and battery analysis [21]. Further, the model considers 18 design conditions over a wide range of cases with the goal of minimizing gross weight [21]. The design conditions use different combinations of the solvers listed above, meaning different design conditions incur different computational costs, making it difficult to parallelize manually. Again, this model was built without parallelization in mind and can be ran in parallel with little effort.

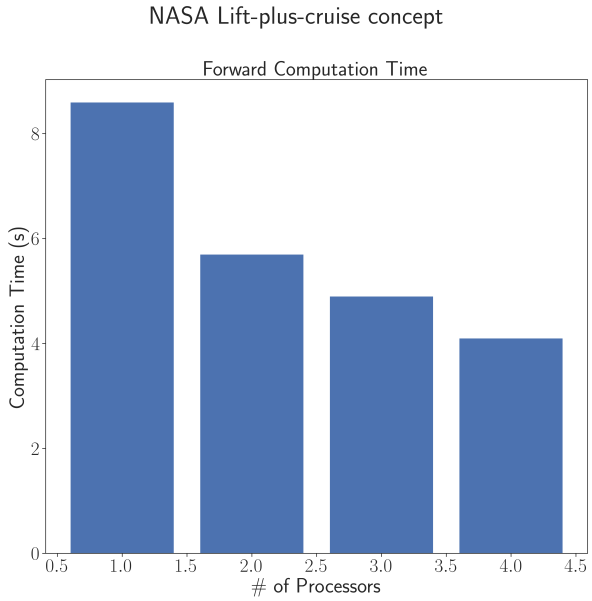

Fig. 10 Model evaluation computation times of a large-scale MDO of NASA's lift-plus-cruise concept parallelized across 1 through 4 processors.

Figure 10 shows the performance improvements through parallelizing across 1 through 4 processors. We observe a significant performance improvement of over 50% for the forward evaluation with 4 processors considering the complexity of the model and the automation of parallelization. However, due to the issues discussed in section IV.D, the derivative computation time did not see a significant decrease in computation time as the number of processors were increased. Further, a large portion of the derivative computation time is spent during non-parallelizable sections, causing the parallelization efficiency to decrease.

VI. Conclusion

In this paper, we presented a method to automatically evaluate multidisciplinary design optimization (MDO) models in parallel. Our method performs static task scheduling by allocating tasks (operations) in the computational graph to processors before model execution. We use well-known concepts in static scheduling including graph coarsening and list scheduling. We also present a framework to predict the computational costs of tasks before executing them. Because MDO problems are often solved using gradient-based optimization, we also enable parallelized adjoint-based derivative computation by reversing the parallelized computational graph and utilizing adjoints of data transfer operations. Finally, we use MPI blocking point-to-point communication operations to perform the data transfer across processors.

We tested our approach by utilizing our method on three applications. We observed significant speedups on all MDO models before and after parallelization. Further, we were able to achieve these speedups automatically without building the MDO models with parallelization in mind. For cases where opportunities for parallelization were easily identifiable, we saw expected performance gains. However, we also saw increased performance when opportunities for parallelization were not clear.

There are many directions for future work. It is evident that the utilization of MPI blocking communication is a

significant obstacle in the way of reaching optimal performance as it causes processors to be unnecessarily idle and it makes the static scheduling problem much more complex. The graph coarsening step taken to minimize the idle time may also reduce the parallelization efficiency as parallelizable nodes may be clustered to one processor. Thus, one possible area for future work includes extending our method to non-blocking operations or improving the efficiency of our algorithm.

Further, the possible inefficiencies with computing derivatives were also discussed. Because the same task allocation in the forward evaluation is used for the derivative computation, the parallelization for computing derivatives may not be efficient. Therefore, it would be beneficial to consider the differences in computational costs between the model evaluation and derivative computation. Also, static scheduling algorithms are beneficial because of their lack of overhead costs during runtime. However, dynamic scheduling can be more efficient in many cases and would be a useful feature to support in the future. Finally, state of the art parallelization utilize GPUs to speed up large tensor operations. Thus, improving the algorithm to support moldable tasks and heterogeneous computing environments is also an area of future work.

Acknowledgments

The material presented in this paper is, in part, based upon work supported by NASA under award No. 80NSSC21M0070.

References

[1] Hwang, J. T., and Martins, J. R., "A Computational Architecture for Coupling Heterogeneous Numerical Models and Computing Coupled Derivatives," *ACM Trans. Math. Softw.*, Vol. 44, No. 4, 2018. https://doi.org/10.1145/3182393, URL https://doi.org/10.1145/3182393.

[2] Gray, J. S., Hwang, J. T., Martins, J. R. R. A., Moore, K. T., and Naylor, B. A., "OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization," *Structural and Multidisciplinary Optimization*, Vol. 59, No. 4, 2019, pp. 1075–1104. https://doi.org/10.1007/s00158-019-02211-z.

[3] Gallard, F., Vanaret, C., Guénot, D., Gachelin, V., Lafage, R., Pauwels, B., Barjhoux, P.-J., and Gazaix, A., "GEMS: A Python Library for Automation of Multidisciplinary Design Optimization Process Generation," *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2018.

[4] van Gent, I., La Rocca, G., and Veldhuis, L. L., "Composing MDAO symphonies: graph-based generation and manipulation of large multidisciplinary systems," *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2017, p. 3663.

[5] Kwok, Y.-K., and Ahmad, I., "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Comput. Surv.*, Vol. 31, No. 4, 1999, p. 406–471. https://doi.org/10.1145/344588.344618, URL https://doi.org/10.1145/344588.344618.

[6] Gandarillas, V., Joshy, A. J., Sperry, M. Z., Ivanov, A. K., and Hwang, J. T., "A graph-based methodology for constructing computational models that automates adjoint-based sensitivity analysis," *Structural and Multidisciplinary Optimization*, (under review).

[7] Reza Hejazi*, S., and Saghafian, S., "Flowshop-scheduling problems with makespan criterion: a review," *International Journal of Production Research*, Vol. 43, No. 14, 2005, pp. 2895–2929.

[8] Hu, T. C., "Parallel Sequencing and Assembly Line Problems," *Operations Research*, Vol. 9, No. 6, 1961, pp. 841–848. URL http://www.jstor.org/stable/167050.

[9] Coffman, E. G., and Graham, R. L., "Optimal scheduling for two-processor systems," *Acta informatica*, Vol. 1, 1972, pp. 200–213.

[10] Levey, E., and Rothvoss, T., "A (1+epsilon)-Approximation for Makespan Scheduling with Precedence Constraints Using LP Hierarchies," *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, Association for Computing Machinery, New York, NY, USA, 2016, p. 168–177. https://doi.org/10.1145/2897518.2897532, URL https://doi.org/10.1145/2897518.2897532.

[11] Karypis, G., and Kumar, V., "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, Vol. 20, No. 1, 1998, pp. 359–392. https://doi.org/10.1137/S1064827595287997, URL https://doi.org/10.1137/S1064827595287997.

[12] Gerasoulis, A., and Yang, T., "On the granularity and clustering of directed acyclic task graphs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 6, 1993, pp. 686–701. https://doi.org/10.1109/71.242154.

[13] Kulkarni, J., Li, S., Tarnawski, J., and Ye, M., *Hierarchy-Based Algorithms for Minimizing Makespan under Precedence and Communication Constraints*, ????, pp. 2770–2789. https://doi.org/10.1137/1.9781611975994.169, URL https://epubs.siam.org/doi/abs/10.1137/1.9781611975994.169.

[14] Zhang, J., Luo, J., and Dong, F., "Scheduling parallel task graphs on non-dedicated heterogeneous multicluster platform with moldable task duplication," *Proceedings of the 2013 IEEE 17th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, IEEE, 2013, pp. 313–318.

[15] Topcuoglu, H., Hariri, S., and Wu, M.-Y., "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 3, 2002, pp. 260–274. https://doi.org/10.1109/71.993206.

[16] Matthew Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," *Proceedings of the 14th Python in Science Conference*, edited by Kathryn Huff and James Bergstra, 2015, pp. 126 – 132. https://doi.org/10.25080/Majora-7b98e3ed-013.

[17] Saif, T., and Parashar, M., "Understanding the behavior and performance of non-blocking communications in MPI," *European Conference on Parallel Processing*, Springer, 2004, pp. 173–182.

[18] Si, M., and Balaji, P., "Process-Based Asynchronous Progress Model for MPI Point-to-Point Communication," *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2017, pp. 206–214. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.27.

[19] Wittmann, M., Hager, G., Zeiser, T., and Wellein, G., "Asynchronous MPI for the Masses," , 2013.

[20] Sarojini, D., Ruh, M. L., Joshy, A. J., Yan, J., Ivanov, A. K., Scotzniovsky, L., Fletcher, A. H., Orndorff, N. C., Sperry, M., Gandarillas, V. E., Asher, I., Chambers, J. T., Gill, H., Lee, S., Cheng, Z., Rodriguez, G., Zhao, S., Mi, C., Nascenzi, T., Cuatt, T., Winter, T. F., Guibert, A. T., Cronk, A., Kim, H. A., Meng, S., and Hwang, J. T., *Large-Scale Multidisciplinary Design Optimization of an eVTOL Aircraft using Comprehensive Analysis*, ???? https://doi.org/10.2514/6.2023-0146, URL https://arc.aiaa.org/doi/abs/10.2514/6.2023-0146.

[21] Ruh, M. L., Fletcher, A., Sarojini, D., Sperry, M., Yan, J., Scotzniovsky, L., van Schie, S. P. C., Warner, M., Orndorff, N. C., Xiang, R., Joshy, A. J., Zhao, H., Krokowski, J. J., Gill, H., Lee, S., Cheng, Z., Cao, Z., Mi, C., Silva, C., Wolfe, L., Chen, J. S., and Hwang, J. T., "Large-scale multidisciplinary design optimization of a NASA air taxi concept using a comprehensive physics-based system mode," *AIAA SCITECH 2024 Forum*, 2024.

[22] Martins, J. R. R. A., and Lambe, A. B., "Multidisciplinary Design Optimization: A Survey of Architectures," *AIAA Journal*, Vol. 51, No. 9, 2013, pp. 2049–2075. https://doi.org/10.2514/1.J051895, URL https://doi.org/10.2514/1.J051895.

[23] Safro, I., Sanders, P., and Schulz, C., "Advanced Coarsening Schemes for Graph Partitioning," *ACM J. Exp. Algorithmics*, Vol. 19, 2015. https://doi.org/10.1145/2670338, URL https://doi.org/10.1145/2670338.

[24] Chen, J., Saad, Y., and Zhang, Z., "Graph coarsening: from scientific computing to machine learning," *SeMA Journal*, 2022, pp. 1–37.

[25] Herrmann, J., Özkaya, M. Y., Uçar, B., Kaya, K., and Çatalyürek, U. V., "Multilevel Algorithms for Acyclic Partitioning of Directed Acyclic Graphs," *SIAM Journal on Scientific Computing*, Vol. 41, No. 4, 2019, pp. A2117–A2145. https://doi.org/10.1137/18M1176865, URL https://doi.org/10.1137/18M1176865.

[26] Torres, H. C., and Murman, S. M., *Automatic Runtime Scheduling Via Directed Acyclic Graphs for CFD Applications*, ???? https://doi.org/10.2514/6.2023-3426, URL https://arc.aiaa.org/doi/abs/10.2514/6.2023-3426.

[27] Bouhlel, M. A., Hwang, J. T., Bartoli, N., Lafage, R., Morlier, J., and Martins, J. R. R. A., "A Python surrogate modeling framework with derivatives," *Advances in Engineering Software*, 2019, p. 102662. https://doi.org/https://doi.org/10.1016/j.advengsoft.2019.03.005.

[28] 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, 2013, pp. 2582–2599. https://doi.org/10.2514/1.J052184.

[29] Sperry, M., Kondap, K., and Hwang, J. T., "Automatic adjoint sensitivity analysis of models for large-scale multidisciplinary design optimization," *AIAA AVIATION 2023 Forum*, 2023, p. 3721.

[30] Ruh, M. L., and Hwang, J. T., "Robust modeling and optimal design of rotors using blade element momentum theory," *AIAA AVIATION 2021 FORUM*, 2021, p. 2598.

[31] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al., "Open MPI: Goals, concept, and design of a next generation MPI implementation," *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*, Springer, 2004, pp. 97–104.