AIAA AVIATION Forum
August 2-6, 2021, VIRTUAL EVENT
AIAA AVIATION 2021 FORUM

10.2514/6.2021-2378

Check for updates

Coded Distributed Path Planning for Unmanned Aerial Vehicles

Cameron Douma*, Baoqian Wang†, Junfei Xie ‡

To enable urban air mobility (UAM), an efficient shortest path planning algorithm is required to ensure safe UAV navigation in large-scale urban environments. Existing optimal shortest path planning algorithms, e.g., the Dijkstra's algorithm, are computationally infeasible for complicated scenarios and large-scale problems. This paper aims to conquer this challenge by exploring a novel distributed implementation of the classical centralized Dijkstra's algorithm. The proposed algorithm explores the coding theory and the idea of load balancing to address the practical issues prominent in UAV-based distributed computing systems, including uncertain disturbances and node heterogeneity. Comprehensive experimental studies on Amazon EC2 demonstrate the high resiliency of the proposed algorithm to uncertain disturbances and its high efficiency compared with existing solutions.

I. Introduction

Advances in aviation technologies enable a variety of unmanned aerial vehicle (UAV) based applications, such as package delivery, search and rescue, surveillance and reconnaissance,[1–3] etc. Recently, urban air mobility (UAM)[4,5] was proposed to utilize UAVs to transport passengers or cargo within urban and suburban areas, which is envisioned to significantly reduce traffic congestion at the ground and offer people much more affordable, fast, and accessible air transit services. To realize UAM, efficient path planning algorithms that allow UAVs to safely navigate in a large-scale urban environment with many obstacles and to quickly reach the destination are critical.

Path planning has been extensively studied in the literature. Depending on the application needs, path planning problems can be classified into three categories: the shortest path planning problem,[6] coverage path planning problem,[7,8] and travelling salesman problem/vehicle routing problem.[9] Among them, the shortest path planning problem is the most prominent one, which involves finding the optimal path (in term of e.g., travel distance or travel time) from a starting position to the target position while satisfying a set of constraints (e.g., collision avoidance). Classical approaches that guarantee optimal solutions to this problem include the Dijkstra's algorithm,[10] dynamic programming,[11] and Floyd-Warshall algorithm,[12] etc. However, these approaches suffer from the scalability issue and are computationally infeasible for complicated scenarios and large-scale problems. To improve efficiency, many heuristic approaches have been proposed, such as A*,[13] D*,[14] probabilistic roadmap,[15] RRT,[16] RRT*.[17] However, these approaches generate sub-optimal paths which can be unsatisfactory.

To enhance the efficiency of optimal shortest path planning algorithms while not sacrificing their accuracy, distributed/parallel computing is a promising solution,[18] which partitions the computation load into several parts and leverages extra computing devices/processors to execute the parts simultaneously. Although there have been a few studies that investigate the distributed/parallel implementation of the graph-based path planning algorithms including the Dijkstra's algorithm and Floyd-Warshall algorithm,[18] distributed/parallel path planning in the context of UAVs or other unmanned vehicles has not been well studied. One reason for the lack of study is that previous designs of UAVs have mainly focused on the control, communication and networking aspects, with the computing aspect being largely ignored.[19] Therefore, existing studies on path

*Undergraduate student, Department of Computer Engineering, San Diego State University
†Ph.D. student, Department of Electrical and Computer Engineering, University of California-San Diego, San Diego State University
‡Assistant professor, Department of Electrical and Computer Engineering, San Diego State University, AIAA member. Email: jxie4@sdsu.edu.

Copyright © 2021 by the American Institute of Aeronautics and Astronautics, Inc. All rights reserved.

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

planning for UAVs have been focused on centralized off-line algorithms[20] or simple but less-accurate on-line algorithms.[20]

In recent years, the advance of embedded systems and the emergence of networked airborne computing,[19,21] mobile edge computing (MEC),[22] and Internet of Things (IoT)[23] make it feasible to execute computation-intensive tasks directly onboard of UAVs. By offloading tasks to nearby servers, UAVs, mobile devices or remote clouds, the computing performance can be further enhanced. Relying on these techniques, it is thus possible to design more effective and efficient shortest path planning algorithms for UAVs to navigate in a large-scale and complicated urban environment. In this paper, we aim to design such an algorithm by exploring distributed computing techniques, where the UAV is assumed to be able to leverage the computing resources from other nodes (e.g., neighboring UAVs, ground servers, nearby mobile devices or remote clouds) to plan its path.

Existing distributed path planning algorithms assume perfect communication and computing systems, with each computing node having the same computing power.[18] Nevertheless, this is not true in a UAV-based distributed computing system, where the communication between computing nodes is subject to uncertain disturbances due to UAV mobility, line-of-sight effect, and wireless interference. Furthermore, computing nodes, which can be neighboring UAVs, ground servers, mobile devices or remote clouds, may have different computing powers. New algorithms that address these practical issues are thus needed.

In this preliminary study, we investigate the classical Dijkstra's algorithm and propose a Load-Balanced Coded (LBC) load allocation scheme to enhance system resilience to uncertain disturbances and address node heterogeneity. This innovative load allocation scheme explores the coding theory and the idea of load balancing to smartly partition and assign computation loads with redundancy to the computing nodes. To further speed up computation, we also develop a dynamic batch processing scheme, which allows partial results to be returned early and dynamically adjusts the frequency of data return to achieve the optimal performance. The resulting distributed Dijkstra's algorithm that integrates the LBC and dynamic batch processing schemes achieves higher efficiency than existing solutions at the presence of stragglers, as demonstrated by the experimental results generated using Amazon EC2 computing clusters.

In the rest of the paper, we first briefly review the Dijkstra's algorithm and its distributed implementation by traditional approaches in Section II. Section III describes the proposed coded distributed path planning algorithm. Our experimental results are then presented in Section IV. Finally, we summarize our paper in Section V.

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

II. Preliminaries

In this section, we first describe the problem to be solved. We then briefly review the Dijkstra's algorithm and a traditional distributed implementation of this algorithm.

II.A. Problem Description

Consider the scenario where a UAV navigates in a large-scale urban environment with the presence of various obstacles such as trees and buildings. It aims to reach a target position s_g, starting from an initial position s_l. Suppose the UAV flies at the same altitude, so that its movement can be described in a 2-dimensional (2-D) space. The shortest path planning problem aims to find a sequence of waypoints $S = \{s_i\}_{i\in[n]}$ to the target position, such that the total travel cost J is minimized and there is no collision with the obstacles. Here s_i is the i-th waypoint, n is the total number of waypoints, and $[n] := \{1, 2, \ldots, n\}$. Mathematically, the problem can be formulated as follows:

$$\begin{aligned} \underset{S=\{s_i\}_{i\in[n]}}{minimize} \quad & J = \sum_{i=1}^{n-1} \mathrm{d}(s_i, s_{i+1}) \\ subjectto \quad & s_i \in C_{free}, \quad \forall i \in [n] \\ & s_1 = s_l \\ & s_n = s_g \end{aligned} \tag{1}$$

where $\mathrm{d}(s_i, s_{i+1})$ measures the cost to travel from s_i to s_{i+1}. C_{free} denotes the free space that the UAV can safely fly, which can be derived by $C_{free} = \{s \mid \|s - o\|_2 \leq \epsilon, \forall o \in C_{obs}\}$, where C_{obs} is the obstacle space and $\epsilon > 0$ is the minimum distance the UAV should maintain with any obstacle to avoid collision.

The above optimization problem can be solved by discretizing the map and applying the classical Dijkstra's algorithm. However, the computation cost grows exponentially when the size of the map increases. In this study, we explore distributed computing techniques to reduce the running time of the Dijkstra's algorithm, by assuming that the UAV can offload its computation task to nearby UAVs, servers, mobile computing devices, or remote clouds. Nevertheless, due to node mobility and the harsh communication environment in the aerial layer, task offloading is subject to significant uncertain disturbances, causing slowdowns in computation. Moreover, the offloadees often have different computing capabilities, which have been overlooked in existing solutions. These practical issues will be addressed in the next Section.

II.B. Dijkstra's Algorithm

In this subsection, we briefly describe how to use the Dijkstra's algorithm to solve the shortest path planning problem in (1). The Dijkstra's algorithm[24] is a graph-based greedy algorithm that finds the shortest path between two vertices in a graph, by making locally optimal choices. To construct the graph $\mathcal{G}(V, E)$, we first decompose the free space C_{free} into grids with the size of each grid to be $\beta \times \beta$, $\beta > 0$. Then the centroids of the grids become vertices V in the graph, and adjacent grids are connected to form edges E. Let $s_i \in V$ be a vertex and $\mathbf{A}$ be a distance matrix that captures the relationship among the vertices and its (i, j)-th entry is defined as follows

$$a_{ij} = \begin{cases} \mathrm{d}(s_i, s_j), & If vertices\ s_i\ and\ s_j\ formanedge \\ 0, & If\ i = j \\ \infty, & Otherwise \end{cases} \tag{2}$$

Given a start position $s_l \in V$ and target position $s_g \in V$, the procedures to find the shortest path from s_l to s_g using the Dijkstra's algorithm are summarized in Algorithm 1. In particular, Line 1 introduces a set $Visited$ to maintain the list of vertices that have been visited, and set $Unvisited$ to maintain the list of vertices that have not been visited. Line 2 introduces a vector $D = \{D(s_i)\}$ to store the estimate of the shortest distance from the start position s_l to each vertex $s_i \in V$, and Line 3 introduces a vector $P = \{P(s_i)\}$ to store the last vertex (before s_i) in the estimated shortest path from s_l to s_i. Lines 4-12 then iteratively search for the shortest path from s_l to s_g by visiting each vertex $s_i \in V$ and improving the estimate of the shortest distance from s_l to each vertex s_i. With optimized D and P, Lines 13-15 finally constructs the shortest path S from s_l and s_g.

II.C. Traditional Distributed Dijkstra's Algorithm

A traditional distributed implementation of the Dijkstra's algorithm[18] is to partition and distribute the load for computing the vectors D and P, by dividing the set of vertices V into N equally-sized non-overlapping groups, denoted $\mathcal{L}_i \subseteq V$, $i \in [N]$, such that each worker node $i \in [N]$ computes $D_i = \{D(s_j)\}_{s_j \in \mathcal{L}_i}$ and $P_i = \{P(s_j)\}_{s_j \in \mathcal{L}_i}$, where N is the total number of worker nodes. Let $K = |V|$ be the total number of vertices. Then $|\mathcal{L}_i| = \lfloor \frac{K}{N} \rfloor$, $\forall i \in [N-1]$ and $|\mathcal{L}_N| = K - (N-1)\lfloor \frac{K}{N} \rfloor$. Algorithm 2 summarizes the procedures followed by the master node (UAV) (Lines 1-21) and the worker nodes (Lines 22-34).

III. Coded Distributed Path Planning

In this section, we introduce a novel coded distributed Dijkstra's algorithm that addresses the limitations of the traditional distributed Dijkstra's algorithm and enhances its efficiency under uncertain disturbances. Before we describe the proposed algorithm, let's first discuss the limitations of the traditional distributed Dijkstra's algorithm so as to motivate our approach.

III.A. Limitations of the Traditional Distributed Dijkstra's Algorithm

Uncertain disturbances (e.g., node/link failures, communication bottlenecks and data losses) are prominent in a UAV-based distributed computing system, which will significantly delay or even fail the transmission of data between two nodes. The traditional distributed Dijkstra's algorithm, unfortunately, is vulnerable to such uncertain disturbances. In particular, as shown in Algorithm 2, in the traditional distributed Dijkstra's

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

Algorithm 1: Centralized Dijkstra's algorithm

Input: s_l, s_g, $\mathcal{G}(V, E)$
Output: shortest path S
// Step 1: Initialization
1 $Visited \leftarrow \emptyset$; $Unvisited \leftarrow V$
2 $D(s_l) \leftarrow 0$; $D(s_i) \leftarrow \infty$, $\forall s_i \in V \setminus \{s_l\}$
3 $P(s_i) \leftarrow \emptyset$, $\forall s_i \in V$
// Step 2: Search the environment
4 $s_c \leftarrow s_l$
5 **while** $s_g \notin Visited$ **do**
6 **foreach** $s_j \in Unvisited$, $s_j \notin Visited$ **do**
7 **if** $D(s_j) > D(s_c) + a_{cj}$ **then**
8 $D(s_j) \leftarrow D(s_c) + a_{cj}$
9 $P(s_j) \leftarrow s_c$
10 $Visited \leftarrow Visited \cup \{s_c\}$
11 $Unvisited \leftarrow Unvisited \setminus \{s_c\}$
12 $s_c \leftarrow \text{argmin}_{s_i \in Unvisited} D(s_i)$
// Step 3: Construct the shortest path
13 $s \leftarrow s_g$; $S \leftarrow s$
14 **while** $s_l \notin S$ **do**
15 $S \leftarrow P(s) \cup S$
16 $s \leftarrow P(s)$
17 **Return** S

algorithm, the master node cannot update the intermediate variable s_c in Line 12 until it has received all results from the worker nodes in each iteration. Therefore, any delay of results will increase the running time of this algorithm.

Furthermore, the traditional distributed Dijkstra's algorithm assumes that all computing nodes are homogeneous with the same computing capability and thus it assigns each node with a same load. However, real distributed computing systems can often be heterogeneous with nodes of differing computing capabilities. This will result in some worker nodes completing their tasks sooner than others. Nodes that finish early have to idly wait for others to finish, which can create long delays.

III.B. Coded Distributed Dijkstra's Algorithm

In order to address the limitations of the traditional distributed Dijkstra's algorithm, we develop a novel load allocation scheme to smartly partition and assign the loads to the worker nodes. To further speed up the computation, we develop a dynamic batch processing scheme that allows worker nodes to return partial results early. In the following subsections, we first introduce the load allocation scheme and then describe the dynamic batch processing scheme. The complete algorithm is provided at the end.

III.B.1. Load-Balanced Coded Load Allocation Scheme

Our load allocation scheme employs the repetition codes to enhance the resilience of the distributed computing system to uncertain disturbances. Particularly, denote γ as the number of *stragglers* (worker nodes that suffer from uncertain disturbances) we seek tolerance to. With repetition codes, redundancies are introduced into the computation by assigning each vertex to $(\gamma + 1)$ worker nodes,[25] meaning that each vertex will be processed for γ redundant times. γ thus reflects the level of repetition or redundancy. To partition the load, we first divide the N worker nodes into $(\gamma+1)$ groups, with each group consisting of $M = \frac{N}{\gamma+1}$ nodes, where N is assumed to be a multiple of $\gamma + 1$. Then, by assigning the whole set of K vertices to each group, each vertex will be processed by $(\gamma + 1)$ worker nodes.

To determine the set of vertices assigned to each worker node within each group, we employ the idea of load balancing[25] to partition the load proportionally according to the nodes' computing capabilities, such

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

Algorithm 2: Traditional Distributed Dijkstra's algorithm

Input: s_l, s_g, $\mathcal{G}(V, E)$
Output: shortest path S
`// Master node (UAV):`
$Visited \leftarrow \emptyset$; $Unvisited \leftarrow V$; $Received \leftarrow \emptyset$; $D \leftarrow \emptyset$; $P \leftarrow \emptyset$; $s_c \leftarrow s_l$
while $s_g \notin Visited$ **do**
  $Visited \leftarrow Visited \cup \{s_c\}$
  $Unvisited \leftarrow Unvisited \setminus \{s_c\}$
  Send $Visited$, $Unvisited$, s_c to all worker nodes, and then listen to the channel
  while $|Received| \neq N$ **do**
    if *Receiving* $\hat{s}$ *and* $D(\hat{s})$ *from worker node* i **then**
      $Received \leftarrow Received \cup \hat{s}$
      $D \leftarrow D \cup D_i(\hat{s})$
  $s_c \leftarrow \text{argmin}_{s \in Received} D(s)$
  Received $\leftarrow \emptyset$
for $i = 1 : N$ **do**
  Receive P_i from worker node i.
  $P = P \cup P_i$
$s \leftarrow s_g$; $S \leftarrow s$
while $s_l \notin S$ **do**
  $S \leftarrow P(s) \cup S$
  $s \leftarrow P(s)$
Return S
`// Worker node` i`:`
$P_i(s) \leftarrow \emptyset$, $Visited \leftarrow \emptyset$, $\forall s \in \mathcal{L}_i$; $D_i(s) \leftarrow \infty$, $\forall s \in \mathcal{L}_i$
if $s_l \in \mathcal{L}_i$ **then**
  $D_i(s_l) \leftarrow 0$
while $s_g \notin Visited$ **do**
  Upon receiving $Visited$, $Unvisited$, s_c *from the master node:*
  foreach $s_j \in Unvisited \cap \mathcal{L}_i$, $s_j \notin Visited$ **do**
    if $D_i(s_j) > D_i(s_c) + a_{cj}$ **then**
      $D_i(s_j) \leftarrow D_i(s_c) + a_{cj}$
      $P_i(s_j) \leftarrow s_c$
  $\hat{s} \leftarrow \text{argmin}_{s \in Unvisited \cap \mathcal{L}_i, s \notin Visited} D_i(s)$
  Send $\hat{s}$ and $D_i(\hat{s})$ to master.
Send P_i to the master node

that more powerful nodes are assigned more vertices than slower nodes. In particular, suppose the CPU frequency of worker node $i \in G_p$ is $C_i > 0$, where G_p is the set of vertices that belong to group $p \in [\gamma + 1]$ and $|G_p| = M$. The number of vertices assigned to worker node i is then given by

$$|\mathcal{L}_i| = \left\lfloor K \frac{C_i}{\sum_{i \in G_p} C_i} \right\rceil \tag{3}$$

where $\lfloor \rceil$ rounds the included value to its nearest integer. To ensure the total number of vertices assigned to the group is K, we determine $\mathcal{L}_i$ one by one using (3) and make the number of vertices assigned to the last node be K subtracting the total number of vertices assigned to the previous $M-1$ nodes. Note that it does not matter which vertices should be assigned to a worker node.

Figure 1 shows an example of load allocation for $N = 4$, $\gamma = 1$, $K = 4000$, $C_1 = C_3 = 2.3$ GHz and $C_2 = C_4 = 3.4$ GHz. In this example, the four worker nodes are divided into $\gamma + 1 = 2$ groups, where the first group consists of the first two nodes and the second group consists of the other two nodes. Using (3), we can obtain the number of vertices assigned to each worker node. Particularly, $|\mathcal{L}_1| = |\mathcal{L}_3| = 1,614$ and

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

$|\mathcal{L}_2| = |\mathcal{L}_4| = 2,386$. The full set of vertices assigned to each group has a size of $K = 4,000$ and make up the complete map to be processed.

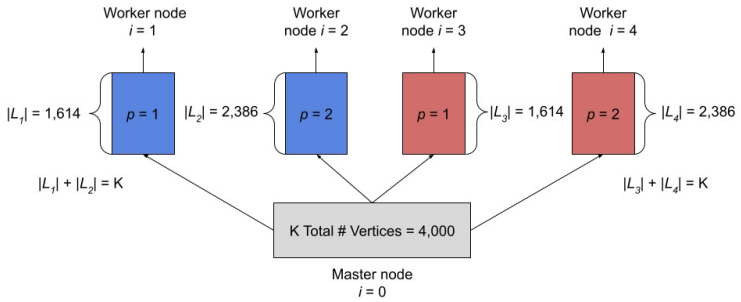


Figure 1: Load-Balanced Coded Scheme for load allocation with $N = 4$, $\gamma = 2$ and $K = 4,000$.

III.B.2. Dynamic Batch Processing

To further speed up the computation, we extend the batch processing procedure proposed in our previous studies.[26–28] The key idea is to partition the set of vertices to be processed by each worker node i at each iteration, i.e., $Unvisited \cap \mathcal{L}_i$ in Line 28 of Algorithm 2, into subgroups, called *batches*. In the following sections, we let $U_i[k]$ represent the list of vertices to be processed by worker node i at the k-th iteration. Each worker node i will process these vertices batch by batch and return results to the master node whenever a batch has been processed. As worker nodes do not need to wait to return results until all assigned vertices have been processed, the master node is expected to get the desired amount of results earlier. Decomposing the set of vertices into small batches is also expected to greatly enhance the system resilience to uncertain stragglers, as loss/delay of small amount of results has a small impact on the whole task.

To determine the number of batches for each worker node i, denoted as b_i, the original batch processing procedure[26–28] sets it as a constant that stays unchanged throughout the entire program. The value of b_i can be determined by minimizing the overall task completion time. It was observed[27] that more batches of smaller size improves system efficiency to some extent but the creation and transmission of batches introduce overhead.

Setting the number of batches as constants works well for matrix multiplication problems considered in our previous studies where each sub-matrix is processed by a worker node only once. However, in the distributed Dijkstra's algorithm, the list of vertices being processed by each worker node i, i.e., $U_i[k]$, decreases one by one after each iteration until there are no more nodes left to visit. As the load changes over time, using the same batch number at all iterations may not lead to the best performance, considering the overhead introduced by batches. Through experiments, we observed that the optimal number of batches generally decreases as the list $U_i[k]$ becomes smaller, and equals 1 when the time required for processing $U_i[k]$ is roughly equal to or smaller than the time required for transmitting the result back to the master node.

To account for this observation, we develop a heuristic algorithm to dynamically adjust the number of batches for each worker node as its load changes over time. In particular, denote $b_i[k]$ as the number of batches for worker node i at the k-th iteration. The number of vertices in each batch is then $\frac{|U_i[k]|}{b_i[k]}$. Let $t_{m,i}$ be the transmission time of the result returned by worker node i to the master node at each iteration. As two values, $\hat{s}$ and $D_i(\hat{s})$ (see Line 32 of Algorithm 2), are transmitted at each iteration, k is removed from the notation. Based on the hypothesis that the optimal number of batches equals 1 when the processing time of the list of vertices is equal to or smaller than the transmission time of the results, the optimal batch size can be approximated by $\frac{t_{m,i}C_i}{\alpha}$, where α is the average number of CPU cycles required for processing a vertex. We then approximate $b_i[k]$ by

$$b_i[k] = \max\left\{\left\lfloor \frac{|U_i[k]|\alpha}{t_{m,i}C_i} \right\rfloor, 1\right\} \tag{4}$$

where $\lfloor\rfloor$ is the floor function. To estimate the value of $t_{m,i}$, we record the transmission time at each iteration and apply the exponentially weighted moving average[29] to get an average value. Furthermore, as $|U_i[k]|$ only

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

Algorithm 3: Coded Distributed Dijkstra's algorithm

Input: s_l, s_g, $\mathcal{G}(V,E)$, f, α, $\{C_1, C_2, \ldots, C_N\}$, $\{b_1, b_2, \ldots, b_N\}$
Output: shortest path S
`// Master node (UAV):`
$Visited \leftarrow \emptyset$; $Unvisited \leftarrow V$; $Received \leftarrow \emptyset$; $D \leftarrow \emptyset$; $P \leftarrow \emptyset$; $s_c \leftarrow s_l$
while $s_g \notin Visited$ **do**
  $Visited \leftarrow Visited \cup \{s_c\}$
  $Unvisited \leftarrow Unvisited \setminus \{s_c\}$
  Send $Visited$, $Unvisited$, s_c to all worker nodes, and then listen to the channel
  while $|Received| \neq N$ **do**
    if *Receiving* $\hat{s}$ *and* $D(\hat{s})$ *from worker node* i **then**
      if $\hat{s} \notin Received$ **then**
        $Received \leftarrow Received \cup \hat{s}$
        $D \leftarrow D \cup D_i(\hat{s})$
  Send acknowledgements to all worker nodes.
  $s_c \leftarrow \text{argmin}_{s \in Received} D(s)$
  Received $\leftarrow \emptyset$
for $i = 1 : N$ **do**
  Receive P_i from any worker node i.
  $P = P \cup P_i$
$s \leftarrow s_g$; $S \leftarrow s$
while $s_l \notin S$ **do**
  $S \leftarrow P(s) \cup S$
  $s \leftarrow P(s)$
Return S
`// Worker node` i`:`
$P_i(s) \leftarrow \emptyset$, $Visited \leftarrow \emptyset$, $\forall s \in \mathcal{L}_i$; $D_i(s) \leftarrow \infty$, $\forall s \in \mathcal{L}_i$
$k \leftarrow 1$
if $s_l \in \mathcal{L}_i$ **then**
  $D_i(s_l) \leftarrow 0$
while $s_g \notin Visited$ **do**
  Upon receiving $Visited$, $Unvisited$, s_c *from the master node:*
  if $(k \mod f) = 0$ **then**
    $b_i \leftarrow \max\left\{\left\lfloor \frac{|Unvisited \cap \mathcal{L}_i|\alpha}{t_{m,i} C_i} \right\rfloor, 1\right\}$
  Divide set $Unvisited \cap \mathcal{L}_i$ into b_i subsets of equal size, denoted as $\{U_{i,1}, \ldots, U_{i,b_i}\}$
  foreach $h \in [b_i]$ **do**
    foreach $s_j \in U_{i,h}$, $s_j \notin Visited$ **do**
      if $D_i(s_j) > D_i(s_c) + a_{cj}$ **then**
        $D_i(s_j) \leftarrow D_i(s_c) + a_{cj}$
        $P_i(s_j) \leftarrow s_c$
      if *Acknowledgement Received* **then**
        $k \leftarrow k + 1$
        break
    $\hat{s} \leftarrow \text{argmin}_{s \in U_{i,h}, s \notin Visited} D_i(s)$
    Send $\hat{s}$ and $D_i(\hat{s})$ to master.
    Record the transmission time and update $t_{m,i}$ using the exponentially weighted moving average.
Send P_i to the master node

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

reduces by one after each iteration, we introduce a variable f to control how often the number of batches is updated using (4).

With the batch processing procedure, unlike the traditional distributed Dijkstra's algorithm, the worker nodes do not need to process all vertices assigned to it. As soon as the master node receives sufficient results required for updating the intermediate variable s_c (see Line 12 in Algorithm 2), it will notify all worker nodes, who will then skip from the current iteration and directly move to the next iteration.

In particular, let $\mathcal{I}$ be the set of worker nodes, whose results have been received by the master node by a certain time. The master node is able to update s_c when $\{s_j | s_j \in \mathcal{L}_i, i \in \mathcal{I}\} = M$.

Algorithm 3 summarizes the procedures followed by the master node (Lines 1-21) and the worker nodes (Lines 22-42). The initial batch numbers are calculated using the historical data.

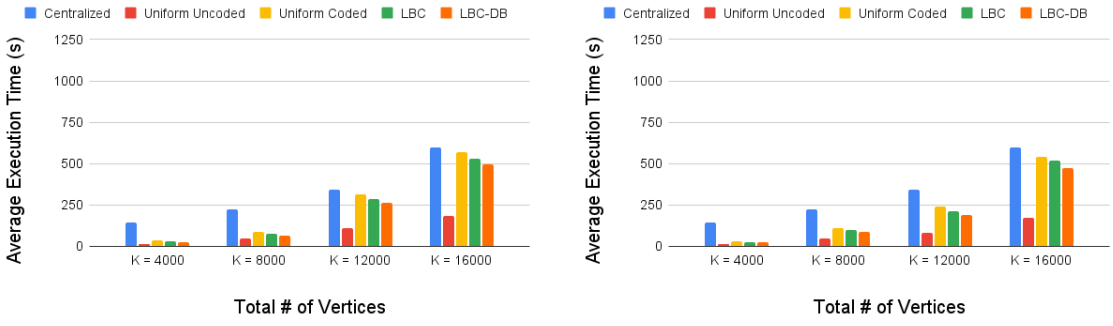
IV. Experimental Studies

In this section, we conduct experiments to evaluate the performance of the proposed coded distributed Dijkstra's algorithm.

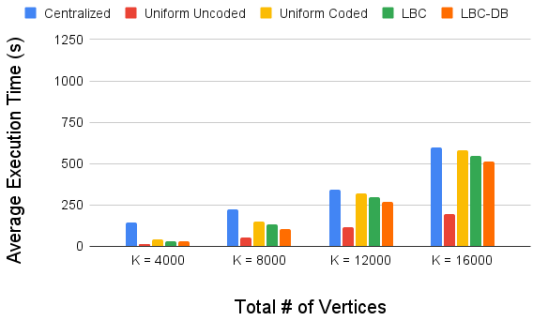
IV.A. Experiment Settings

To evaluate the performance of the proposed coded distributed Dijkstra's algorithm, we consider three different scenarios, each containing a *i3.large* instance type master node and a different combination of worker nodes.

- **Scenario 1** consists of two worker nodes of instance type *r4.xlarge* and two worker nodes of instance type *m5zn.xlarge*.
- **Scenario 2** consists of two worker nodes of instance type *r4.xlarge*, two worker nodes of instance type



(a) (b)



(c)

Figure 2: Average execution time of different algorithms in a) Scenario 1, b) Scenario 2, and c) Scenario 3 when $o = 0$ stragglers exist.

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

c5.lxlarge, and two worker nodes of instance type *m5zn.xlarge*.

- **Scenario 3** consists of two worker nodes of instance type *r4.xlarge* and two worker nodes of instance type *c5.xlarge*.

The CPU frequencies of the different instance types used in this study are 2.3 GHz for *i3.large* instance, 2.3 GHz for *r4.xlarge* instance, 3.4GHz for *c5.xlarge* instance, and 4.5 GHz for *m5zn.xlarge* instance.

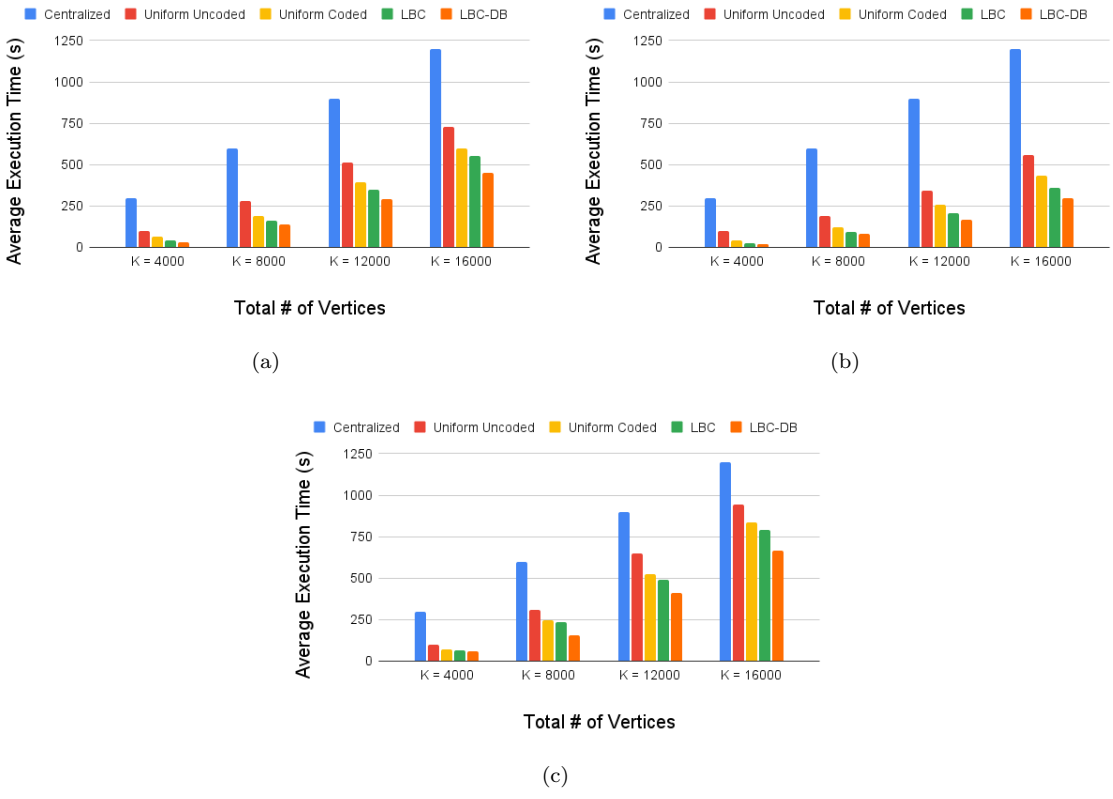
IV.B. Comparison Results

To evaluate the efficiency of the proposed algorithm, we compare it with the following four benchmark algorithms.

- **Centralized**: The centralized Dijkstra's algorithm (Algorithm 1) is implemented on the master node of instance type *i3.large* with no other worker nodes.
- **Uniform Uncoded**: This is the traditional distributed Dijkstra's algorithm (Algorithm 2) that divides the load among the worker nodes evenly without introducing any redundancy into the computation.
- **Uniform Coded**: This is an extension of the traditional distributed Dijkstra's algorithm, which adds redundant computations using the repetition codes described in Section III.B with $\gamma = 1$.
- **Load-Balanced Coded (LBC)**: This algorithm extends the Uniform Coded algorithm to further implement the load balancing idea described in Section III.B.

In the proposed algorithm, the parameters are set as $\gamma = 1$ and $f = 50$.

To evaluate the efficiency of the five different algorithms, we measure the execution time taken by each algorithm to process a grid map of different sizes. Each experiment is repeated for 10 times and the average



(a) (b)

(c)

Figure 3: Average execution time of different algorithms in a) Scenario 1, b) Scenario 2, and c) Scenario 3 when $o = 1$ stragglers exist.

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

execution time was recorded. To evaluate the resilience of these algorithms to uncertain stragglers, we introduce unexpected stragglers that are randomly chosen. In particular, we randomly pick o worker nodes as stragglers at each iteration and make each of these nodes delay for 0.01s before returning the results. Figure 2 shows the mean execution time of different algorithms when the size of the grid map (K) increases and when no stragglers are present. Figure 3 shows their performances when $o = 1$ straggler is present. Note that distributing the load does not hurt the optimality of the generated paths, and all algorithms are able to find the shortest paths.

From Figure 2 and Figure 3, we can observe that leveraging surrounding worker nodes to share the computation load significantly reduces the time required for finding the optimal path. Comparing the four distributed Dijkstra's algorithm, we can observe that the Uniform Uncoded algorithm (i.e., the traditional distributed Dijkstra's algorithm) is more efficient than the three coded algorithms when no stragglers exist (see Figure 2), as it does not involve any computation redundancy. However, when stragglers are present (see Figure 3), the uncoded algorithm performs the worst, demonstrating its vulnerability to uncertain disturbances. Comparing the Uniform Coded with the two Load-Balanced Coded algorithms, we can observe that it always performs worse than the Load-Balanced Coded algorithms, as it ignores the node heterogeneity. Furthermore, from the two figures, we can see that our algorithm always outperforms the two coded algorithms, and the centralized algorithm of course.

IV.C. Effectiveness of the Dynamic Batch Processing Scheme

In order to evaluate the effectiveness of the proposed dynamic batch processing scheme, we compare the proposed algorithm with two benchmark algorithms:

- **Load-Balanced Coded (LBC):** This is the algorithm described in the previous section, which does not introduce any batches or can be considered as a special case of our algorithm with $b_i[k] = 1$, $\forall i \in [N]$.

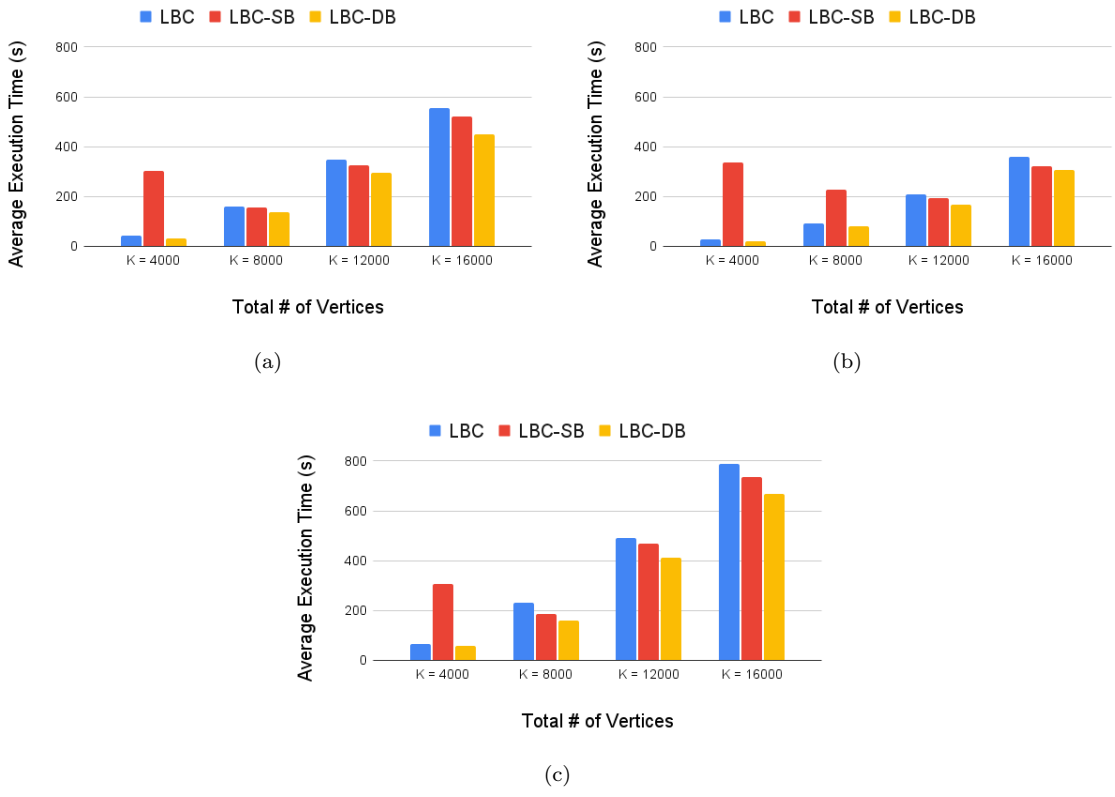


Figure 4: Average execution time of the distributed Dijkstra's algorithm equipped with different batch processing schemes in a) Scenario 1, b) Scenario 2, and c) Scenario 3 when $o = 1$ stragglers exist.

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

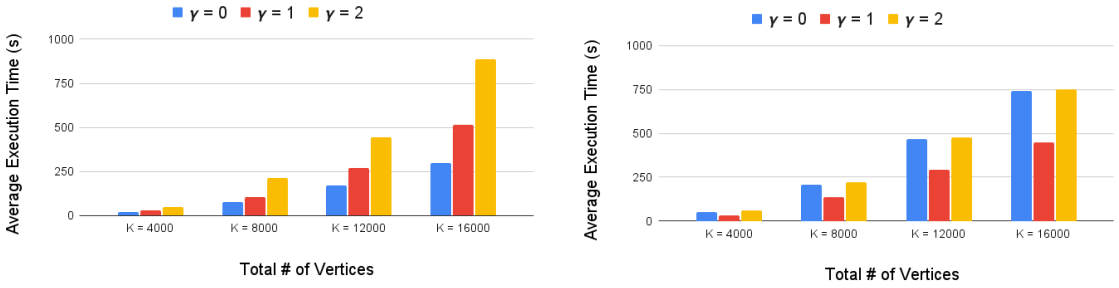
- **Load-Balanced Coded algorithm with Static Batches (LBC-SB)**: This is a variant of our algorithm that has the number of batches for each worker node being unchanged over time. In this study, we set the number of batches in this algorithm as 100 for each worker node, i.e., $b_i[k] = 100$, $\forall i \in [N]$.

Figure 4 shows the performance of the different algorithms in different scenarios when $o = 1$ uncertain straggler exists. Of interest, comparing LBC and LBC-SB, we can observe that the effect that the number of batches has on an algorithm differs greatly based on each scenario and different map sizes. In general, the addition of batches for a worker node improves the overall performance, but only when the load assigned to it is large enough. This is why we can observe a spike in execution time in Figure 4(a) when $K = 4,000$, in Figure 4(b) when $K = 4,000$ and $K = 8,000$, and in Figure 4(c) when $K = 4,000$. The proposed dynamic batch processing scheme takes this into consideration and adjusts the number of batches dynamically, so that a large batch number is used at the beginning when the worker node has a large load, and a gradually decreasing value is used as the load reduces. This makes our algorithm outperform the benchmark algorithms in all cases as shown in Figure 4.

IV.D. Impact of the Level of Redundancy

What is the proper amount of redundancy we should introduce into the computation? To answer this question, we vary the value of the level of repetition γ and evaluate its impact on the performance of the proposed algorithm.

Figure 5(a) and Figure 5(b) show the average execution time of our algorithm at different values of γ in Scenario 1 when no straggler ($o = 0$) and one straggler ($o = 1$) is present, respectively. As expected, when no stragglers exist, the performance of our algorithm keeps degrading with the increase of γ, due to the extra time required for processing redundant load. When stragglers exist, the proposed algorithm performs the best when $\gamma = 1$, suggesting that $\gamma = 1$ is the proper value to use.



(a) (b)

Figure 5: Average execution time of our algorithm with the increase of γ when there are a) $o = 0$ and b) $o = 1$ stragglers.

V. Conclusion

This paper extends the traditional distributed Dijkstra's algorithm to address the practical issues prominent in UAV-based distributed computing systems including uncertain disturbances and node heterogeneity. The proposed coded distributed Dijkstra's algorithm is featured by 1) a Load-Balanced Coded (LBC) scheme that smartly distributes loads among heterogeneous computing nodes under uncertain disturbances and 2) a dynamic batch processing procedure that enables computing nodes to return results early for even higher efficiency. To demonstrate the performance of the proposed algorithm, comprehensive experimental studies were conducted on Amazon EC2. The results show that distributing the load significantly reduces the computation time, especially for large-scale problems. The proposed algorithm is more resilient to uncertain stragglers than the traditional distributed Dijkstra's algorithm. Moreover, the LBC and dynamic batch processing schemes improve computation efficiency significantly. In the future, we will extend our study to explore the distributed implementation of other path planning algorithms.

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378

Acknowledgement

We would like to thank the National Science Foundation (NSF) under Grants CI-1953048 and CAREER-2048266, and the San Diego State University under the University Grants Program for the support of this work.

References

[1]J. Stolaroff, "The need for a life cycle assessment of drone-based commercial package delivery," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2014.

[2]P. Doherty and P. Rudol, "A uav search and rescue scenario with human body detection and geolocalization," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2007, pp. 1–13.

[3]E. Kuiper and S. Nadjm-Tehrani, "Mobility models for uav group reconnaissance applications," in *2006 International Conference on Wireless and Mobile Communications (ICWMC'06)*. IEEE, 2006, pp. 33–33.

[4]A. Mathur, K. Panesar, J. Kim, E. M. Atkins, and N. Sarter, "Paths to autonomous vehicle operations for urban air mobility," in *AIAA Aviation 2019 Forum*, 2019, p. 3255.

[5]L. Gipson, "Nasa embraces urban air mobility, calls for market study," *NASA. November*, vol. 7, 2017.

[6]K. Jiang, L. D. Seneviratne, and S. Earles, "A shortest path based path planning algorithm for nonholonomic mobile robots," *Journal of Intelligent and Robotic Systems*, vol. 24, no. 4, pp. 347–366, 1999.

[7]J. Xie, W. Zhang, and J. Chen, "Path planning for multiple energy constrained unmanned aerial vehicles to cover multiple regions," in *AIAA Aviation 2020 Forum*, 2020, p. 2887.

[8]J. Xie, L. Jin, and L. R. Garcia Carrillo, "Optimal path planning for unmanned aerial systems to cover multiple regions," in *AIAA Scitech 2019 Forum*, 2019, p. 1794.

[9]J. Xie, L. R. G. Carrillo, and L. Jin, "An integrated traveling salesman and coverage path planning problem for unmanned aircraft systems," *IEEE control systems letters*, vol. 3, no. 1, pp. 67–72, 2018.

[10]Y. Singh, S. Sharma, R. Sutton, and D. Hatton, "Optimal path planning of an unmanned surface vehicle in a real-time marine environment using a dijkstra algorithm," in *Marine Navigation*. CRC Press, 2017, pp. 399–402.

[11]K. H. Sedighi, K. Ashenayi, T. W. Manikas, R. L. Wainwright, and H.-M. Tai, "Autonomous local path planning for a mobile robot using a genetic algorithm," in *Proceedings of the 2004 congress on evolutionary computation (IEEE Cat. No. 04TH8753)*, vol. 2. IEEE, 2004, pp. 1338–1345.

[12]S. Hougardy, "The floyd–warshall algorithm on graphs with negative cycles," *Information Processing Letters*, vol. 110, no. 8-9, pp. 279–281, 2010.

[13]P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[14]D. Ferguson, M. Likhachev, and A. Stentz, "A guide to heuristic-based path planning," in *Proceedings of the international workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling (ICAPS)*, 2005, pp. 1–10.

[15]L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[16]S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[17]S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[18]A. Pradhan and G. Mahinthakumar, "Finding all-pairs shortest path for a large-scale transportation network using parallel floyd-warshall and parallel dijkstra algorithms," *Journal of Computing in Civil Engineering*, vol. 27, no. 3, pp. 263–273, 2013.

[19]B. Wang, J. Xie, S. Li, Y. Wan, Y. Gu, S. Fu, and K. Lu, "Computing in the air: An open airborne computing platform," *IET Communications*, vol. 14, no. 15, pp. 2410–2419, 2020.

[20]Z. Shiller, *Off-Line and On-Line Trajectory Planning*, 02 2015, vol. 29, pp. 29–62.

[21]B. Wang, J. Xie, S. Li, Y. Wan, S. Fu, and K. Lu, "Enabling high-performance onboard computing with virtualization for unmanned aerial systems," in *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2018, pp. 202–211.

[22]Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[23]L. Tan and N. Wang, "Future internet: The internet of things," in *2010 3rd international conference on advanced computer theory and engineering (ICACTE)*, vol. 5. IEEE, 2010, pp. V5–376.

[24]T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[25]R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *International Conference on Machine Learning*, 2017, pp. 3368–3376.

[26]B. Wang, J. Xie, K. Lu, Y. Wan, and S. Fu, "Coding for heterogeneous uav-based networked airborne computing," in *2019 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2019, pp. 1–6.

[27]——, "On batch-processing based coded computing for heterogeneous distributed computing systems," *arXiv preprint arXiv:1912.12559*, 2019.

[28]——, "Multi-agent reinforcement learning based coded computation for mobile ad hoc computing," *arXiv preprint arXiv:2104.07539*, 2021.

[29]R. G. Brown, "Exponential smoothing for predicting demand," in *Operations Research*, vol. 5, no. 1, 1957, pp. 145–145.

Downloaded by 66.75.47.237 on July 28, 2021 | http://arc.aiaa.org | DOI: 10.2514/6.2021-2378